

Joel Grus

Data science par la pratique

Fondamentaux avec Python

2^e édition

● Éditions
EYROLLES

O'REILLY®

Préparez-vous aux métiers du futur !

Data science par la pratique

2^e édition

Un ouvrage de référence pour les (futurs) data scientists

Les bibliothèques, les *frameworks*, les modules et les boîtes à outils sont parfaits pour faire de la data science. Ils sont aussi un bon moyen de plonger dans cette discipline sans la comprendre. Dans cet ouvrage, vous apprendrez comment fonctionnent les outils et algorithmes les plus fondamentaux de la data science, en les réalisant à partir de zéro.

Si vous êtes fort en maths et que vous connaissez la programmation, l'auteur, Joel Grus, vous aidera à vous familiariser avec les maths et les statistiques qui sont au cœur de la data science et avec les compétences informatiques indispensables pour démarrer comme data scientist. La profusion des données d'aujourd'hui contient les réponses aux questions que personne n'a encore pensé à poser. Ce livre vous enseigne comment obtenir ces réponses.

La deuxième édition, revue et augmentée, de cet ouvrage comporte des codes et exemples associés entièrement réécrits en Python 3.6 et intègre de nouveaux chapitres sur l'apprentissage profond (*deep learning*), les statistiques et le traitement en langage naturel.

- Suivez un cours accéléré de Python.
- Apprenez les fondamentaux de l'algèbre linéaire, des statistiques et des probabilités, et comprenez comment et quand les utiliser en data science.
- Collectez, explorez, nettoyez, bricolez et manipulez les données.
- Plongez dans les bases de l'apprentissage automatique.
- Implémentez des modèles comme les k plus proches voisins, la classification naïve bayésienne, les régressions linéaire ou logistique, les arbres de décision, les réseaux neuronaux et le *clustering*.
- Explorez les systèmes de recommandation, le traitement du langage naturel, l'analyse de réseau, MapReduce et les bases de données.

À qui s'adresse cet ouvrage ?

- Aux développeurs, statisticiens, étudiants et chefs de projet ayant à résoudre des problèmes de data science.
- Aux data scientists, mais aussi à toute personne curieuse d'avoir une vue d'ensemble de l'état de l'art de ce métier du futur.

Au sommaire

Qu'est-ce que la data science ? • Cours accéléré de Python • Visualisation des données • Algèbre linéaire • Statistique • Probabilités • Hypothèses et inférences • Descente de gradient • Collecte des données • Travail sur les données • Apprentissage automatique • Les k plus proches voisins • Classification naïve bayésienne • Régression linéaire simple • Régression multiple • Régression logistique • Arbres de décision • Réseaux neuronaux • Apprentissage profond • Partitionnement de données • Traitement automatique du langage naturel • Analyse des réseaux • Systèmes de recommandation • Bases de données et SQL • MapReduce • Éthique des données • En avant pour la data science

Joel Grus est ingénieur logiciel chez Google. Auparavant data scientist dans plusieurs start-up, il vit aujourd'hui à Seattle et participe régulièrement à des réunions de data scientists. Il blogue occasionnellement sur joelgrus.com et tweete toute la journée via le compte [@joelgrus](https://twitter.com/joelgrus).

www.editions-eyrolles.com

Data science par la pratique

SUR LE MÊME THÈME

Y. BENZAKI. – **Les data sciences en 100 questions/réponses.**

N°67951, 2020, 124 pages.

M.-R. AMINI. ET AL. – **Data science – Cours et exercices.**

N°67410, 2018, 272 pages.

H. WICKHAM, G. GROLEMUND. – **R pour les data sciences.**

N°67571, 2018, 496 pages.

F. PROVOST, T. FAWCETT. – **Data science pour l'entreprise.**

N°67570, 2018, 384 pages.

E. BIERNAT, M. LUTZ. – **Data science : fondamentaux et études de cas.**

N°14243, 2015, 312 pages.

W. MCKINNEY. – **Analyse de données en Python.**

N°14109, 2015, 488 pages.

M.-R. AMINI. – **Apprentissage machine, de la théorie à la pratique.**

N°13800, 2015, 272 pages.

DANS LA MÊME COLLECTION

K. NOVAK. – **Administration Linux par la pratique – Tome 2.**

N°67949, 2020, 418 pages.

C. DELANNOY. – **Le guide complet du langage C.**

N°67922, 2020, 876 pages.

J. LORIAUX, T. DEFOSSEZ. – **Emailing : développement et intégration.**

N°67849, 2020, 160 pages.

B. BARRÉ. – **Concevez des applications mobiles avec React Native.**

N°67889, 2^e édition, 2019, 224 pages.

S. RINGUEDÉ. – **SAS.**

N°67631, 4^e édition, 2019, 688 pages.

C. BLAESS. – **Développement système sous Linux.**

N°67760, 5^e édition, 2019, 1080 pages.

T. PARISOT. – **Node.js.**

N°13993, 2018, 472 pages.

Retrouvez nos bundles (livres papier + e-book) et livres numériques sur
<http://izibook.eyrolles.com>

Joel Grus

Data science par la pratique

Fondamentaux avec Python

2^e édition

● Éditions
EYROLLES

O'REILLY®

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Traduction autorisée de l'ouvrage en langue anglaise intitulé
Data Science from Scratch, 2nd edition
de Joel Grus, ISBN : 9781492041139,
publié par O'Reilly Media.
All Rights Reserved.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre français d'exploitation du droit de copie, 20, rue des Grands-Augustins, 75006 Paris.

© 2019 by Joel Grus / O'Reilly Media pour l'édition en langue anglaise

© Éditions Eyrolles, 2020 pour la présente édition, ISBN : 978-2-212-67907-6

© Traduction française : Tristan Kottelanne

Préface à la deuxième édition

Je suis particulièrement fier de la première édition de *Data science par la pratique*. Ce livre est tel que je l'envisageais. Toutefois, plusieurs années d'évolution de la data science, d'avancées dans l'écosystème Python et de progrès personnels en tant que développeur et formateur ont bel et bien *changé* mon point de vue sur la conception d'un premier ouvrage abordant ce sujet. Dans la vie réelle, on ne peut pas recommencer les choses. En rédaction, en revanche, il y a les deuxièmes éditions.

Ainsi, j'ai réécrit l'ensemble du code et des exemples en utilisant Python 3.6 (et nombre de ses nouvelles fonctions, comme les annotations de type). Dans le présent ouvrage, j'ai mis l'accent sur la rédaction d'un code propre. J'ai remplacé certains des exemples factices de la première édition par d'autres, plus réalistes, faisant appel à des ensembles de données « réels ». J'ai ajouté de nouvelles ressources sur des thèmes tels que l'apprentissage profond (*deep learning*), les statistiques et le traitement en langage naturel ; autant d'éléments correspondant aux contextes dans lesquels les experts actuels en data science sont amenés à travailler. J'ai également supprimé certaines ressources qui semblaient moins pertinentes. J'ai ensuite repassé l'ouvrage au crible, corrigé les bogues, reformulé les explications les moins claires et remis certaines plaisanteries au goût du jour.

La première édition était un excellent ouvrage ; celle-ci est encore meilleure. Amusez-vous bien !

Joel Grus
Seattle, WA, 2019

Conventions utilisées dans ce livre

Cet ouvrage utilise les conventions typographiques suivantes :

Italique gras

Présente les nouveaux termes ou ceux importants.

URL

Présente les URL et adresses électroniques.

Code dans le texte

Présente, dans le corps de texte, les noms et extensions de fichier, des éléments de programme, tels que noms de variable ou de fonction, bases de données, types de données, variables d'environnement, déclarations et mots-clés.

Code

Présente les listings de programme.

Code en gras

Présente des commandes ou autres éléments de texte que l'utilisateur doit saisir littéralement.

Conseil

Encadré présentant un conseil ou une suggestion.

Remarque

Encadré présentant une remarque générale.

Avertissement

Encadré présentant un avertissement ou une mise en garde.

Utilisation d'exemples de code

Des ressources supplémentaires (exemples de code, exercices, etc.) sont disponibles en téléchargement sur <https://github.com/joelgrus/data-science-from-scratch>.

Le présent ouvrage a pour objectif de vous aider à mener à bien une tâche. En général, si cet ouvrage offre un exemple de code, vous pouvez l'utiliser dans vos programmes et votre documentation. Il n'est pas nécessaire de nous contacter pour nous demander une autorisation, à moins que vous ne reproduisiez une partie considérable du code.

Remerciements

Avant tout, je tiens à remercier Mike Loukides, qui a accueilli favorablement ma proposition pour ce livre (et pour son insistance à me convaincre de le réduire à une taille raisonnable). Il lui aurait été si simple de dire « Qui est ce type qui m'envoie sans arrêt des extraits de chapitres et comment puis-je m'en débarrasser ? » Je suis heureux qu'il ne l'ait pas fait. Je voudrais remercier également mes éditrices à O'Reilly, Michele Cronin et Marie Beaugureau, qui m'ont guidé tout au long du processus de publication et qui ont porté ce livre à un niveau bien meilleur que tout ce à quoi je serais parvenu seul.

Je n'aurais pas pu écrire ce livre si je n'avais pas appris la data science et je n'aurais sans doute jamais étudié cette dernière sans l'influence de Dave Hsu, Igor Tatarinov, John Rauser et le reste de la bande Farecast, à une époque si lointaine qu'on ne parlait pas encore de data science. Les organisateurs de Coursera et DataTau méritent mes remerciements eux aussi.

Je suis reconnaissant aux lecteurs et aux réviseurs de ma version bêta. Jay Fundling a trouvé des erreurs à la pelle et a mis le doigt sur nombre d'explications obscures. Le livre est bien meilleur (et plus correct) grâce à lui. Debashis Ghosh est un héros qui a vérifié le bien-fondé de toutes mes statistiques. Andrew Musselman a suggéré de gommer le côté « ceux qui préfèrent R à Python n'ont aucun sens moral » de ce livre et je suis convaincu que c'était finalement un excellent conseil. Trey Causey, Ryan Matthew Balfanz, Loris Mularoni, Nuria Pujol, Rob Jefferson, Mary Pat Campbell, Zach Geary, Denise Mauldin, Jimmy O'Donnell et Wendy Grus m'ont tous donné des conseils infiniment avisés. Merci à tous les lecteurs et lectrices de la première édition et à toutes celles et tous ceux qui m'ont aidé à écrire un meilleur ouvrage. S'il subsiste quelques erreurs, j'en assume l'entière responsabilité.

Je dois beaucoup à la communauté Twitter *#datascience* qui m'a fait découvrir une multitude de nouveaux concepts, m'a mis en contact avec des personnes extraordinaires et m'a fait prendre conscience de mes imperfections au point que je les ai quittées et que j'ai écrit un livre pour compenser. Des remerciements spéciaux à Trey Causey (encore lui), qui m'a rappelé au passage d'inclure un chapitre sur l'algèbre linéaire, et à Sean J. Taylor, qui a pointé du doigt quelques lacunes sérieuses dans le chapitre *Travail sur les données*.

Par-dessus tout, je dois un immense merci à Ganga et à Madeline. La seule chose plus difficile que d'écrire un livre, c'est de vivre avec quelqu'un qui écrit un livre. Je n'aurais jamais pu achever cet ouvrage sans leur soutien.

Préface à la première édition

« Expert en data science » a été désigné comme « métier le plus sexy du XXI^e siècle »¹ par un journaliste qui n'avait sans doute jamais visité une caserne de pompiers. Il est vrai que la data science est un domaine de premier plan en pleine croissance et il n'y a pas à chercher bien loin pour trouver des analystes surexcités qui prévoient que, dans les dix prochaines années, nous aurons besoin de milliards d'experts en plus de ceux qui existent déjà.

Cependant, qu'est-ce que la data science ou science des données ? Après tout, comment former des experts du domaine – des data scientists – si nous ne savons pas répondre à cette question ? D'après un diagramme de Venn, devenu célèbre chez les spécialistes², la data science se situe à l'intersection des domaines suivants :

- compétences pointues en informatique ;
- connaissance des mathématiques et des statistiques ;
- expertise fonctionnelle.

Bien que ma première intention ait été d'écrire un livre qui couvrirait les trois, j'ai rapidement réalisé que le traitement approfondi de « l'expertise fonctionnelle » à lui seul nécessiterait des dizaines de milliers de pages. À ce stade, j'ai donc décidé de me concentrer sur les deux premiers points. Mon objectif est d'abord de vous aider à développer les compétences informatiques nécessaires pour démarrer en data science et aussi de vous familiariser avec les mathématiques et les statistiques, qui sont au cœur de cette discipline.

C'est une aspiration ambitieuse pour un livre, car la meilleure façon d'apprendre l'informatique reste la pratique. En lisant cet ouvrage, vous apprendrez à bien comprendre ma méthode de programmation, qui ne sera pas forcément la meilleure pour vous. Vous apprendrez à bien comprendre quelques-uns de mes outils, qui ne seront pas forcément les meilleurs dans votre propre cas. Vous apprendrez à bien comprendre comment j'aborde les problèmes de données, ce qui ne sera pas forcément la meilleure approche pour vous. Mon intention (et mon secret espoir) est que mes exemples vous inspirent au point que vous mettiez au point

1. <https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century>

2. <http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram>

votre propre méthode. Tout le code et toutes les données de ce livre sont disponibles sur GitHub³ pour vous aider à démarrer.

Si la meilleure façon d'apprendre les mathématiques consiste à faire des mathématiques, sachez que cet ouvrage n'est absolument pas un livre sur ce sujet et que, pour l'essentiel, nous n'allons pas « faire des maths ». Cependant, vous ne pouvez pas véritablement pratiquer la data science sans comprendre un peu les probabilités, les statistiques et l'algèbre linéaire. En d'autres termes, le cas échéant, nous plongerons dans les équations, l'intuition mathématique, les axiomes et les versions simplifiées des grandes idées mathématiques. J'espère que vous n'aurez pas peur de plonger avec moi.

Tout au long de notre périple, j'espère aussi vous apporter la sensation que jouer avec les données est amusant parce que, en fait... jouer avec les données est amusant ! (En particulier comparé à d'autres activités comme remplir sa déclaration de revenus ou travailler dans une mine de charbon.)

Partir de rien

Il existe pléthore de bibliothèques, *frameworks*, modules et boîtes à outils dédiés à la data science ; autant d'éléments qui mettent en œuvre les algorithmes et les techniques les plus courants dans ce domaine (et aussi les moins courants d'ailleurs). Vous pénétrerez dans l'intimité de NumPy, scikit-learn, pandas et toute une panoplie d'autres bibliothèques. Parfaites pour la data science, elles constituent aussi une bonne manière de débiter sans réellement comprendre le domaine.

Dans cet ouvrage, nous appréhenderons la data science en partant de zéro. Nous construirons des outils et nous réaliserons des algorithmes à la main afin de mieux les comprendre. J'ai beaucoup travaillé sur la mise en œuvre des exemples pour qu'ils soient clairs, bien commentés et lisibles. La plupart du temps, les outils que nous construirons seront parfaits pour comprendre, mais totalement inutilisables dans la réalité. Ils seront adaptés à de petits jeux de données, mais ne résisteront pas à ceux dimensionnés à l'échelle du Web.

Au fil du livre, je vous indiquerai les bibliothèques à employer pour appliquer ces techniques à plus grande échelle, mais nous ne nous en servirons pas ici.

Le débat fait toujours rage pour décider quel est le meilleur langage pour apprendre la data science.

Beaucoup pensent que c'est le langage de programmation statistique R. (Disons-le tout net, ils ont tort.) Certains suggèrent Java ou Scala. Cependant, à mon avis, le choix de Python saute aux yeux.

3. <https://github.com/joelgrus/data-science-from-scratch>

Python possède plusieurs fonctions qui le rendent particulièrement bien adapté à l'apprentissage (et la pratique) de la data science :

- il est gratuit ;
- il est assez facile à coder (et aussi à comprendre) ;
- il dispose de nombreuses bibliothèques en lien avec la data science.

J'hésite à appeler Python mon langage préféré. Il en existe d'autres que je trouve plus agréables, mieux conçus ou simplement plus amusants à utiliser. Et pourtant, pratiquement à chaque fois que je commence un projet de data science, je finis par revenir à Python. À chaque fois que j'ai besoin d'élaborer rapidement un prototype fonctionnel, je reviens à Python. Et à chaque fois que je veux démontrer des concepts de data science de manière claire et limpide, je reviens également à Python. Ce livre fait donc appel à Python.

L'objectif de cet ouvrage n'est pas de vous apprendre Python. Pendant un chapitre, je vais vous accompagner dans un cours accéléré pour mettre en évidence les fonctions les plus importantes pour nous. Toutefois, si vous ignorez tout de la programmation en Python (ou de la programmation tout court), vous aurez intérêt à compléter ce livre par un manuel du genre « Python pour débutants ».

Le reste de notre introduction à la data science suivra la même approche : entrer dans les détails quand il est crucial ou particulièrement instructif de le faire et, à d'autres moments, laisser de côté les détails pour que vous puissiez trouver vos propres solutions (ou consulter Wikipédia).

Je forme des data scientists depuis des années. Même si tous ne sont pas devenus des data gourous capables de changer le monde, je les ai tous quittés meilleurs qu'ils ne l'étaient lors de notre première rencontre. Et j'en suis venu à croire que quiconque possède quelques aptitudes aux mathématiques et à la programmation a toutes les compétences nécessaires pour pratiquer la data science. Tout ce qu'il faut, c'est un esprit avide de découverte, de la motivation et ce livre...

Table des matières

CHAPITRE 1

Introduction 1

L'origine des données	1
Qu'est-ce que la data science ?	2
L'hypothèse DataSciencester	3
À la recherche des connecteurs clés	3
Des experts que vous connaissez peut-être	6
Salaires et expérience	9
Les comptes payants	12
Les centres d'intérêt	12
À suivre	14

CHAPITRE 2

Cours accéléré de Python..... 15

Découvrir Python	15
Obtenir Python	15
Les environnements virtuels	16
Les principes zen de Python	17
La mise en page par les espaces	18
Les fonctions et modules	19
Les fonctions	19
Les modules	20
Les exceptions	21
Types complexes de données	22
Les chaînes de caractères	22
Les listes	23
Les tuples	25
Les dictionnaires	25
Defaultdict	27
Les compteurs	28
Les ensembles	28

Les structures de contrôle	29
Les éléments booléens	30
Trier	32
Les listes en compréhension	32
Les tests automatisés et instructions assert	33
La programmation orientée objet	34
Les générateurs et itérateurs	36
Les valeurs aléatoires	38
Les expressions rationnelles	39
Programmation fonctionnelle	40
zip et le déballage d'arguments	40
args et kwargs	41
Les annotations de type	42
Comment rédiger des annotations de type	44
Pour aller plus loin	46

CHAPITRE 3

Visualisation des données.... 47

matplotlib	47
Les graphiques à barres	49
Les courbes	51
Les nuages de points	52
Pour aller plus loin	54

CHAPITRE 4

Algèbre linéaire..... 55

Les vecteurs	55
Les matrices	59
Pour aller plus loin	63

CHAPITRE 5

Statistique..... 65

Décrire un unique jeu de données	65
Tendances centrales	67
La dispersion	69
La corrélation	71
Le paradoxe de Simpson	74
Les autres chausse-trappes	75
Corrélation et causalité	75
Pour aller plus loin	76

CHAPITRE 6

Probabilités..... 77

Probabilité conditionnelle	78
Théorème de Bayes	80
Variables aléatoires	81
Distributions continues	81
Distribution normale	83
Théorème de la limite centrale	85
Pour aller plus loin	87

CHAPITRE 7

Hypothèses et inférences..... 89

Test statistique d'une d'hypothèse	89
Exemple : lancer de pièce	90
p-values	93
Intervalle de confiance	94
P-hacking	95
Exemple : effectuer un test A/B	96
Inférence bayésienne	98
Pour aller plus loin	100

CHAPITRE 8

Descente de gradient..... 101

Idée cachée derrière la descente de gradient	101
Estimer le gradient	102
L'utilisation du gradient	105

Choisir le bon pas	106

S'adapter à des modèles	106

Mini-lot et descente du gradient stochastique	108

Pour aller plus loin	109

CHAPITRE 9

Collecte des données 111

stdin et stdout	111

Lecture de fichiers	113

Fondamentaux des fichiers de type texte	114

Fichiers à délimiteur	115

Ratissage du Web	117

Analyse syntaxique du HTML	117

Exemple : Congrès sous surveillance	119

Utilisation des API	122

JSON et XML	123

Utiliser une API non authentifiée	124

Trouver des API	125

Exemple : utilisation de l'API Twitter	125

L'accréditation	125

Utilisation de Twython	126

Pour aller plus loin	129

CHAPITRE 10

Travail sur les données 131

Explorer les données	131

Données à une dimension	131

Données à deux dimensions	133

Données à plusieurs dimensions	134

Utiliser NamedTuple	136

Dataclass	137

Nettoyer et transformer	138

Manipuler les données	140

Changer d'échelle	143

En aparté : tqdm	145

Réduire la dimension	147

Pour aller plus loin	151

CHAPITRE 11

Apprentissage**automatique 153**

Modélisation 153

Qu'est-ce que l'apprentissage
automatique ? 154

Surajustement et sous-ajustement 155

Exactitude 157

Le compromis entre le biais et la variance 160

Extraction et sélection des variables 161

Pour aller plus loin 162

CHAPITRE 12

Les k plus proches voisins... 163

Modèle 163

Exemple : jeu de données Iris 165

Malédiction de la dimension 168

Pour aller plus loin 171

CHAPITRE 13

Classification naïve**bayésienne 173**Un filtre antispam particulièrement
stupide 173

Un filtre antispam plus élaboré 174

La mise en œuvre 175

Tester le modèle 178

Utiliser le modèle 179

Pour aller plus loin 181

CHAPITRE 14

Régression linéaire simple.. 183

Modèle 183

Utiliser la descente de gradient 186

Estimer le maximum de vraisemblance .. 187

Pour aller plus loin 188

CHAPITRE 15

Régression multiple 189

Modèle 189

Autres hypothèses du modèle
des moindres carrés 190

Ajuster le modèle 191

Interprétation du modèle 193

Qualité de l'ajustement du modèle 193

Digression : l'amorce 194

Erreur standard des coefficients
de régression 195

Régulariser 197

Pour aller plus loin 200

CHAPITRE 16

Régression logistique 201

Problème 201

Fonction logistique 203

Appliquer le modèle 205

Bien ajuster le modèle 206

Séparateurs à vaste marge 207

Pour aller plus loin 210

CHAPITRE 17

Arbres de décision..... 211

Qu'est-ce qu'un arbre de décision ? 211

Entropie 213

Entropie d'une partition 215

Construire un arbre de décision 216

Synthèse 218

Forêts aléatoires 221

Pour aller plus loin 222

CHAPITRE 18

Réseaux neuronaux..... 223

Perceptron 224

Réseaux neuronaux à propagation
vers l'avant 225

Rétropropagation 228

Exemple : Fizz Buzz	230	Aparté : échantillonnage de Gibbs	280
Pour aller plus loin	233	Modélisation thématique	282
CHAPITRE 19		Vecteurs de mots	287
Apprentissage profond..... 235		Réseaux neuronaux récurrents	296
Tenseur	235	Exemple : utilisation d'un RNN	
Abstraction de couche	238	de niveau caractère	299
Couche linéaire	240	Pour aller plus loin	302
Réseaux neuronaux en tant que séquences		CHAPITRE 22	
de couches	242	Analyse des réseaux..... 303	
Perte et optimisation	243	Centralité internœud	303
Exemple : l'opérateur xor revisité	246	Centralité de vecteur propre	308
Autres fonctions d'activation	247	Multiplication matricielle	308
Exemple : Fizz Buzz revisité	248	Centralité	310
Fonctions softmax et entropie croisée ..	249	Graphes orientés et PageRank	312
Abandon	252	Pour aller plus loin	314
Exemple : MNIST	253	CHAPITRE 23	
Enregistrement et chargement		Systèmes	
des modèles	257	de recommandation..... 315	
Pour aller plus loin	258	Méthode manuelle	316
CHAPITRE 20		Recommander ce qui est populaire	316
Partitionnement		Filtrage collaboratif sur la base	
de données..... 259		des utilisateurs	317
Idee sous-jacente	259	Filtrage collaboratif sur la base	
Modèle	260	des articles	320
Exemple : rencontres	262	Factorisation de matrice	322
Choisir k	264	Pour aller plus loin	327
Exemple : partitionnement de couleurs ..	265	CHAPITRE 24	
Partitionnement hiérarchique de bas		Bases de données et SQL 329	
en haut	266	CREATE TABLE et INSERT	329
Pour aller plus loin	272	UPDATE	333
CHAPITRE 21		DELETE	334
Traitement automatique		SELECT	334
du langage naturel..... 273		GROUP BY	337
Nuages de mots	273	ORDER BY	339
Modèles n-grammes	275	JOIN	340
Grammaires	278	Sous-requêtes	343
		Index	343

Optimisation des requêtes	344	Données biaisées	363
NoSQL	344	Protéger les données	364
Pour aller plus loin	345	En résumé	365
		Pour aller plus loin	365
CHAPITRE 25		CHAPITRE 27	
MapReduce	347	En avant pour	
Exemple : compteur de mots	348	la data science	367
Pourquoi MapReduce ?	349	IPython	367
MapReduce vu plus généralement	350	Mathématiques	368
Exemple : analyser les mises à jour		Ne pas partir de rien	368
de statuts	352	NumPy	369
Exemple : multiplication matricielle	353	pandas	369
Aparté : les fonctions de combinaison	355	scikit-learn	369
Pour aller plus loin	356	Représentations graphiques	369
		R	370
CHAPITRE 26		Apprentissage profond	370
Éthique des données	357	Trouver des données	371
Qu'est-ce que l'éthique des données ?	357	Pratiquer la data science	371
Mais qu'est-ce que l'éthique des données ?	358	Hacker News	371
Dois-je me soucier de l'éthique		Camions de pompiers	372
des données ?	359	T-shirts	372
Élaborer de mauvais produits de données	359	Des tweets sur le globe	373
Compromis entre précision et équité	360	Et vous ?	373
Collaborer	361		
Interpréter	362	Index	375
Recommander	362		

1

Introduction

*Des données ! Des données ! Donnez-moi des données ! » s'écria-t-il avec impatience.
« Je ne peux pas faire de briques sans argile !*

Arthur Conan Doyle

L'origine des données

Nous vivons dans un monde noyé dans les données. Les sites web suivent à la trace chaque clic de chaque utilisateur. Tous les jours, votre smartphone enregistre votre localisation et votre vitesse à la seconde près. Des « accros aux données » dopés aux stéroïdes portent des podomètres qui enregistrent en permanence leur rythme cardiaque, leurs gestes, leur alimentation et leur sommeil. Les voitures intelligentes collectent les habitudes de conduite, les maisons intelligentes collectent les habitudes de vie et les marqueteurs intelligents collectent les habitudes d'achat. Internet lui-même n'est qu'un énorme diagramme de connaissances qui contient (entre autres) une gigantesque encyclopédie de références croisées, des bases de données spécialisées sur le cinéma, la musique, les résultats sportifs, les flippers de bistrot, les mêmes et les cocktails et, pour alimenter la réflexion, trop de statistiques officielles (certaines d'entre elles presque vraies !) publiées par de trop nombreux gouvernements.

Enfouies dans ces données se cachent les réponses aux innombrables questions que personne n'a jamais pensé à poser. Nous allons apprendre à les trouver grâce à ce livre.

Qu'est-ce que la data science ?

Connaissez-vous la blague pour définir un data scientist (autrement dit, un expert en data science) ? C'est quelqu'un qui s'y connaît mieux en statistiques qu'un informaticien et mieux en informatique qu'un statisticien – je n'ai pas dit que c'était drôle... Dans les faits, certains data scientists sont, pour des raisons pratiques, des statisticiens, alors que d'autres se fondent assez bien parmi les ingénieurs en informatique. Certains sont des experts de l'apprentissage machine (*machine learning*), alors que d'autres en sont aux balbutiements dans ce domaine. Certains sont titulaires de doctorats et collectionnent les publications tandis que d'autres n'ont jamais lu une thèse universitaire de leur vie (honte à eux quand même). En résumé, quelle que soit la définition que vous donnez à la data science, vous trouverez des praticiens auxquels elle ne s'applique absolument pas !

Ce n'est pas ce qui va nous empêcher d'essayer. Nous dirons qu'un data scientist est une personne capable d'extraire du sens d'un lot de données inextricables. Le monde d'aujourd'hui est peuplé de gens qui tentent d'apporter du sens aux données.

Par exemple, le site de rencontres OkCupid pose à ses membres de très nombreuses questions pour trouver le partenaire qui leur correspond le mieux. Cependant, il analyse aussi ces résultats pour élaborer des questions innocentes en apparence qui vous permettront de connaître vos chances de coucher avec elle ou lui dès le premier rendez-vous¹.

Facebook vous demande votre ville natale et celle où vous vivez actuellement, officiellement pour permettre à vos amis de vous retrouver plus facilement. Toutefois, il utilise aussi ces données pour repérer des schémas de déplacement des populations² et savoir où vivent les fans des différentes équipes de football³.

Certaines chaînes de supermarchés suivent vos achats et vos interactions, aussi bien en ligne que dans leurs magasins. Elles utilisent ces données pour construire un modèle prédictif⁴ ; par exemple, pour savoir quelles clientes sont enceintes et ainsi leur présenter plus efficacement les articles de périculture.

En 2012, l'équipe de Barack Obama a employé, pour sa campagne électorale, des dizaines de data scientists pour explorer les bases de données d'électeurs dans le but de trouver ceux qui devaient être particulièrement cajolés afin d'orienter les appels et programmes de levée de fonds vers certains donateurs et de concentrer les efforts là où ils seraient le plus utiles. De même, en 2016, la campagne de Donald Trump a été exploitée pour tester un panel impressionnant de publicités en ligne⁵ et analyser des données pour déterminer ce qui fonctionne et ce qui échoue.

1. <https://www.gwern.net/docs/psychology/okcupid/thebestquestionsforafirstdate.html>

2. <https://www.facebook.com/notes/facebook-data-science/coordinated-migration/10151930946453859>

3. <https://www.facebook.com/notes/facebook-data-science/nfl-fans-on-facebook/10151298370823859>

4. <https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html>

5. <https://www.wired.com/2016/11/facebook-won-trump-election-not-just-fake-news/>

Rassurez-vous tout de même : certains data scientists font parfois usage de leurs compétences pour de bonnes actions, notamment en utilisant des données pour rendre les actions gouvernementales plus efficaces⁶, pour aider les sans-abris⁷ ou encore pour améliorer la santé publique. Notez quand même que, si vous préférez imaginer des moyens d'inciter les gens à cliquer sur des publicités, votre carrière ne s'en portera pas plus mal.

L'hypothèse DataSciencester

Félicitations ! Vous venez d'être recruté pour conduire les efforts en faveur de la data science chez DataSciencester, le réseau social des experts du domaine.

Remarque

Lorsque j'ai écrit la première édition de cet ouvrage, je pensais qu'un réseau social dédié aux experts en science des données constituait une hypothèse de travail aussi amusante que ridicule. Depuis lors, de tels réseaux sociaux pour data scientists ont bel et bien été créés et ils ont généré bien plus d'argent de la part d'investisseurs en capital-risque que je n'en ai gagné avec mon livre. En matière d'hypothèses ridicules sur la data science et/ou de publication d'ouvrages, il est indéniable qu'il faut ici tirer de précieuses leçons.

Malgré son public cible, DataSciencester n'a jamais rien investi pour développer sa propre pratique de data science (pour être tout à fait honnête, DataSciencester n'a jamais vraiment non plus investi dans l'élaboration de son produit). Ce sera votre mission ! Tout au long de ce livre, vous allez découvrir les concepts de la data science en trouvant les solutions aux problèmes que vous rencontrerez au travail. Parfois, nous examinerons des données explicitement fournies par des utilisateurs, parfois nous observerons des données générées au moyen de leurs interactions avec le site et parfois nous examinerons des données issues des expériences que nous aurons conçues.

Et, comme DataSciencester affiche une très forte culture hostile à ce qui « ne vient pas de chez nous », nous construirons nos propres outils de A à Z. À la fin, vous aurez acquis une compréhension solide des bases de la data science et vous aurez davantage d'assurance pour mettre vos compétences au service d'une entreprise ou pour résoudre tout autre problème qui vous intéresse.

C'est parti, bienvenue et bonne chance !

À la recherche des connecteurs clés

C'est votre premier jour chez DataSciencester et déjà le responsable Réseautage a quantité de questions concernant vos utilisateurs. Jusqu'à présent, il n'avait aucun interlocuteur. Il est donc tout excité d'apprendre votre recrutement.

6. <https://www.marketplace.org/2014/08/22/tech/beyond-ad-clicks-using-big-data-social-good/>

7. <http://www.dssgfellowship.org/2014/08/20/tracking-the-paths-of-homelessness/>

En particulier, il veut que vous identifiiez les « connecteurs clés » parmi les experts en data science. Pour ce faire, il vous remet une extraction complète de l'ensemble du réseau DataSciencester.

Remarque

Dans la vraie vie, il est rare que quelqu'un vous remette ainsi les données qui vous sont nécessaires. La collecte des données sera abordée au chapitre 9.

À quoi ressemble cette extraction de données ? C'est une liste d'utilisateurs, chacun représenté par un `dict` contenant l'identifiant (`id`, au format numérique) et le nom de l'utilisateur (qui, sous l'effet d'une de ces coïncidences cosmiques extraordinaires, rime avec l'identifiant en anglais) :

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

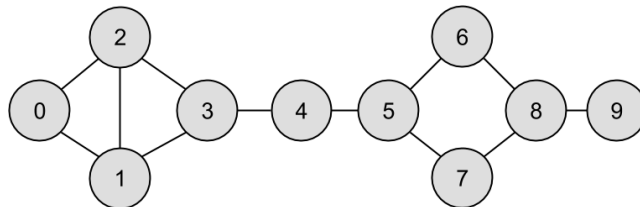
Il vous remet également les « relations » entre les inscrits, représentées par une liste de paires d'`id` comme suit :

```
friendship_pairs = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
                    (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Par exemple, le tuple `(0, 1)` indique que le data scientist identifié par `id 0` (Hero) et celui identifié par `id 1` (Dunn) sont amis. La figure 1–1 illustre le réseau.

Figure 1–1

Le réseau DataSciencester



Représenter les relations sous forme d'une liste de paires n'est pas le meilleur moyen de les exploiter. En effet, pour identifier toutes les relations de l'utilisateur 1, vous devrez alors faire des itérations sur chaque paire en recherchant celles qui contiennent 1. Si vos paires sont nombreuses, l'exercice risque de prendre du temps. Au lieu de cela, nous représentons nos

utilisateurs sous forme de `dict` et les valeurs sont des listes d'id de relations (d'amis). Consulter un `dict` est très rapide.

Remarque

Ne vous attardez pas trop sur le détail du code à ce stade. Vous aurez droit à un cours accéléré de Python au chapitre 2. Pour le moment, essayez simplement d'avoir une vue d'ensemble de la chose.

Pour créer le `dict`, il faut, bien sûr, examiner chaque paire, mais une seule fois. Les consultations suivantes seront bien moins coûteuses :

```
# initialiser le dict avec une liste vide pour chaque id d'utilisateur
friendships = {user["id"]: [] for user in users}

# et boucler sur les paires de relation pour le remplir
for i, j in friendship_pairs:
    friendships[i].append(j) # ajoute j comme ami de i
    friendships[j].append(i) # ajoute i comme ami de j
```

Une fois que chaque `dict` contient sa liste d'amis, il est facile pour nous de répondre à des questions comme « Quel est le nombre moyen de connexions ? »

Cherchons d'abord le nombre total de connexions en faisant la somme des longueurs de toutes les listes d'amis :

```
def number_of_friends(user):
    """combien d'amis l'utilisateur _user_ a-t-il ?"""
    user_id = user["id"]
    friend_ids = friendships[user_id]
    return len(friend_ids)

total_connections = sum(number_of_friends(user)
                        for user in users) # 24
```

Il suffit ensuite de diviser par le nombre total d'utilisateurs :

```
num_users = len(users) # longueur de la liste friend_ids
avg_connections = total_connections / num_users # 24 / 10 == 2.4
```

Il est tout aussi facile de trouver les personnes les plus connectées : ce sont les personnes qui ont le plus grand nombre d'amis.

Comme il n'y a pas beaucoup d'utilisateurs, nous pouvons les trier depuis « le plus d'amis » jusqu'à « le moins d'amis » :

```
# créer une liste (user_id, number_of_friends)
num_friends_by_id = [(user["id"], number_of_friends(user))
                      for user in users]

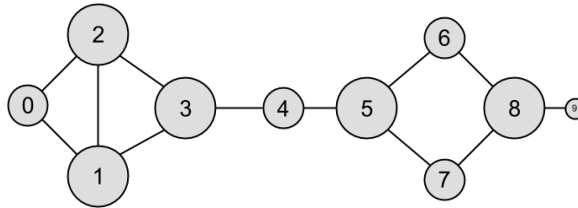
num_friends_by_id.sort( # trier la liste
    key=lambda id_and_friends: id_and_friends[1], # suivant num_friends
    reverse=True) # du plus grand au plus petit

# chaque paire vaut (user_id, num_friends)
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

Pour interpréter ce que nous venons de faire, on peut considérer que c'est une façon d'identifier les personnes centrales du réseau. En fait, nous venons de calculer le degré de centralité du réseau (figure 1-2).

Figure 1-2

Le réseau DataSciencester dimensionné par degré



L'avantage tient à la facilité de calcul, mais le résultat n'est pas toujours celui attendu. Par exemple, dans le réseau DataSciencester, Thor (id 4) a seulement deux connexions, alors que Dunn (id 1) en a trois. Et pourtant, quand on regarde le réseau, on a intuitivement l'impression que Thor est plus central. Au chapitre 22, nous examinerons les réseaux plus en détail et nous étudierons des notions de centralité plus complexes, plus ou moins en accord avec nos intuitions.

Des experts que vous connaissez peut-être

Alors que vous êtes en train de remplir votre dossier de nouvel embauché, la responsable de la Fraternisation débarque dans votre bureau. Elle veut encourager davantage de connexions au sein de vos relations.

Elle vous demande donc de concevoir un système pour suggérer « des experts que vous connaissez peut-être déjà ».

Votre première idée est de suggérer qu'un utilisateur peut connaître les amis de ses amis. C'est facile à calculer. Vous rédigez donc un code pour faire des itérations sur les amis d'un utilisateur et collecter les résultats ; les amis des amis :

```
def foaf_ids_bad(user):
    """foaf" est l'abréviation de "friend of a friend", autrement dit "ami d'un ami"""
    return [foaf_id
            for friend_id in friendships[user["id"]]
            for foaf_id in friendships[friend_id]]
```

Quand nous appelons cette fonction sur `users[0]` (Hero), elle récupère les informations suivantes :

```
[0, 2, 3, 0, 1, 3]
```

Cela inclut `user 0` deux fois, car Hero est ami avec ses deux amis. Le résultat inclut donc les utilisateurs 1 et 2, bien qu'ils soient déjà amis avec Hero. Et il inclut deux fois l'utilisateur 3, car Chi peut être atteint par l'intermédiaire de deux amis différents :

```
print(friendships[0]) # [1, 2]
print(friendships[1]) # [0, 2, 3]
print(friendships[2]) # [0, 1, 3]
```

Savoir que des personnes sont des amis d'amis de plusieurs manières semble être une information intéressante. Aussi, devrions-nous plutôt compter les amis communs. Et nous devrions surtout utiliser une fonction supplémentaire pour exclure les personnes déjà connues de l'utilisateur :

```
from collections import Counter # n'est pas chargé par défaut

def friends_of_friends(user):
    user_id = user["id"]
    return Counter(
        foaf_id
        for friend_id in friendships[user_id] # pour chacun de mes amis
        for foaf_id in friendships[friend_id] # trouve leurs amis
        if foaf_id != user_id # qui ne sont pas moi
        and foaf_id not in friendships[user_id] # et ne sont pas mes amis
    )

print(friends_of_friends(users[3])) # compteur({0: 2, 5: 1})
```

Ceci met en évidence que Chi (id 3) a deux amis communs avec Hero (id 0), mais seulement un avec Clive (id 5).

En tant que data scientist, vous aimeriez aussi rencontrer des utilisateurs qui partagent vos centres d'intérêt (un bon exemple de « domaine de prédilection » de la data science). Après

avoir demandé autour de vous, vous avez réussi à mettre la main sur ces données, sous la forme d'une liste de paires (`user_id`, `interest`) :

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"), (1, "NoSQL"),
    (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"), (1, "Postgres"),
    (2, "Python"), (2, "scikit-learn"), (2, "scipy"), (2, "numpy"),
    (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"),
    (4, "decision trees"), (4, "libsvm"), (5, "Python"), (5, "R"),
    (5, "Java"), (5, "C++"), (5, "Haskell"),
    (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"),
    (8, "deep learning"), (8, "Big Data"),
    (8, "artificial intelligence"), (9, "Hadoop"), (9, "Java"),
    (9, "MapReduce"), (9, "Big Data")
]
```

Par exemple, Hero (id 0) n'a aucun ami commun avec Klein (id 9), mais ils partagent un intérêt pour Java et les Big Data.

Il est facile de construire une fonction qui trouve les utilisateurs présentant un centre d'intérêt commun :

```
def data_scientists_who_like(target_interest):
    """trouve tous les utilisateurs partageant le centre d'intérêt cible"""
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

Cela fonctionne, mais il faut passer en revue la liste complète des centres d'intérêt à chaque recherche. Si nous avons beaucoup d'utilisateurs et de centres d'intérêt (ou si nous voulons effectuer beaucoup de recherches), il sera sans doute préférable de construire un index des centres d'intérêt par utilisateur :

```
from collections import defaultdict
# La clé est le centre d'intérêt, la valeur est la liste
# d'identifiants user_id ayant ce centre d'intérêt.
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```


Et un autre à partir des utilisateurs vers les centres d'intérêt :

```
# La clé est l'identifiant user_id, la valeur est la liste
# de centres d'intérêt pour ce user_id.
interests_by_user_id = defaultdict(list)

for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

Maintenant, il est facile de trouver qui a le plus de centres d'intérêt en commun avec un utilisateur donné :

- faire des itérations sur les centres d'intérêt de l'utilisateur ;
- pour chacun des centres d'intérêt, explorer par itération les autres utilisateurs concernés ;
- compter combien de fois nous voyons les autres utilisateurs.

Voici ce que cela donne sous forme de code :

```
def most_common_interests_with(user):
    return Counter(
        interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"]
    )
```

Nous pourrions alors utiliser ces résultats pour construire une fonctionnalité enrichie « les experts que vous pourriez connaître » basée sur une combinaison d'amis communs et de centres d'intérêt communs. Nous explorerons ce type d'application au chapitre 23.

Salaires et expérience

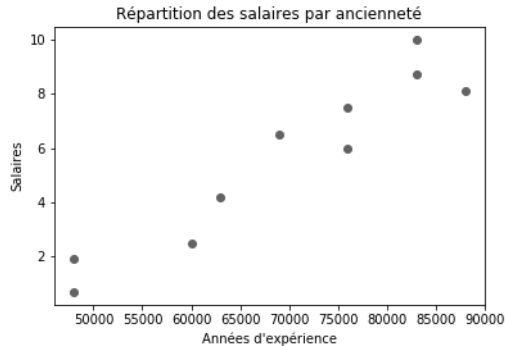
Vous êtes prêt à sortir déjeuner quand le responsable des Relations publiques vous demande si vous pouvez illustrer de manière ludique la question du salaire d'un expert en data science. Les salaires restent, bien sûr, des données sensibles, mais il réussit à vous procurer un jeu de données anonymes contenant le salaire de chaque utilisateur (en dollars) et son ancienneté au poste de data scientist (en années) :

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
                        (60000, 2.5), (83000, 10),
                        (48000, 1.9), (63000, 4.2)]
```

La première étape consiste naturellement à représenter graphiquement les données (nous verrons comment procéder au chapitre 3).

Figure 1–3

Salaire par années d'expérience



Il est évident que les personnes les plus expérimentées ont tendance à gagner davantage. Aussi, comment illustrer une telle réalité de manière ludique ? Votre première idée consiste à considérer le salaire moyen par nombre d'années d'ancienneté :

```
# La clé est l'année, la valeur est la liste de salaires
# pour chaque période d'affectation.
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)

# La clé est l'année, la valeur est le salaire moyen
# pour cette période.
average_salary_by_tenure = {
    tenure: sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

Finalement, ce n'est pas très utile, car aucun des utilisateurs n'a la même ancienneté et le rapport revient à énumérer simplement le salaire de chacun :

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
 6: 76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10: 83000.0}
```

Il serait sans doute plus intéressant de répartir l'ancienneté en catégories :

```
def tenure_bucket(tenure):
    if tenure < 2:
        return "less than two"
    elif tenure < 5:
        return "between two and five"
    else:
        return "more than five"
```

Pour ensuite regrouper les salaires dans les différentes catégories :

```
# La clé est la catégorie de poste, la valeur est la liste
# de salaires pour cette catégorie.
salary_by_tenure_bucket = defaultdict(list)
for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

Et enfin, calculer le salaire moyen de chaque catégorie :

```
# La clé est la catégorie de poste, la valeur est le salaire moyen
# pour cette catégorie.
average_salary_by_bucket = {
    tenure_bucket: sum(salaries) / len(salaries)
    for tenure_bucket, salaries
        in salary_by_tenure_bucket.items()
}
```

Le résultat est plus intéressant :

```
{'between two and five': 61500.0,
 'less than two': 48000.0,
 'more than five': 79166.66666666667}
```

Et voilà ce que vous annoncez : « Les data scientists ayant plus de cinq ans d'expérience gagnent 65 % de plus que ceux qui ont peu ou pas d'expérience. »

Cependant, nous avons choisi les catégories de manière totalement arbitraire. Nous préférons dégager une règle sur l'effet que peut avoir une année d'ancienneté supplémentaire sur le salaire (en moyenne). En plus de nous procurer une annonce plus percutante, cela nous permettrait de faire des prévisions sur des salaires que nous ne connaissons pas. Nous développerons cette idée au chapitre 14.

Les comptes payants

De retour à votre bureau, vous trouvez la responsable Recettes qui vous attend. Elle voudrait mieux appréhender quels utilisateurs paient pour avoir un compte et lesquels ne paient pas. Elle connaît leurs noms, mais ce n'est pas une information très facile à exploiter.

Vous remarquez qu'il semble exister une correspondance entre l'expérience et les comptes payants :

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6.0 unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10.0 paid
```

Si vous voulez modéliser la situation – même s'il n'y a vraiment pas assez de données pour établir un modèle – vous pourriez être tenté de prédire que les utilisateurs peu ou très expérimentés sont « payants », alors que les utilisateurs d'expérience moyenne sont « non payants » :

```
def predict_paid_or_unpaid(years_experience):
    if years_experience < 3.0:
        return "paid"
    elif years_experience < 8.5:
        return "unpaid"
    else:
        return "paid"
```

Évidemment, nous avons complètement déduit les seuils à l'œil nu.

Avec davantage de données (et plus de mathématiques), nous pourrions construire un modèle prédictif pour connaître la probabilité qu'un utilisateur paie, en nous basant sur son nombre d'années d'expérience. Nous étudierons ce genre de problème au chapitre 16.

Les centres d'intérêt

Alors que votre premier jour touche à sa fin, la responsable Stratégie de contenu voudrait obtenir des données sur les sujets qui intéressent le plus les utilisateurs, afin de planifier ses publications sur le blog en conséquence. Vous avez déjà les données brutes du projet précédent dont le but était de suggérer des experts qui pourraient être vos amis :

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"), (1, "NoSQL"),
    (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"), (1, "Postgres"),
```

```
(2, "Python"), (2, "scikit-learn"), (2, "scipy"), (2, "numpy"),
(2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
(3, "statistics"), (3, "regression"), (3, "probability"),
(4, "machine learning"), (4, "regression"),
(4, "decision trees"), (4, "libsvm"), (5, "Python"), (5, "R"),
(5, "Java"), (5, "C++"), (5, "Haskell"),
(5, "programming languages"), (6, "statistics"),
(6, "probability"), (6, "mathematics"), (6, "theory"),
(7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
(7, "neural networks"), (8, "neural networks"),
(8, "deep learning"), (8, "Big Data"),
(8, "artificial intelligence"), (9, "Hadoop"), (9, "Java"),
(9, "MapReduce"), (9, "Big Data")
]
```

Une manière simple (mais pas très excitante) de trouver les sujets les plus populaires consisterait à compter les mots :

- 1 Passer chaque sujet en minuscules (car différents utilisateurs ont pu utiliser ou non des majuscules).
- 2 Isoler les mots de chaque sujet.
- 3 Compter les résultats.

Voici ce que cela donne sous forme de code :

```
words_and_counts = Counter(word
                             for user, interest in interests
                             for word in interest.lower().split())
```

Cela permet, par exemple, d'identifier facilement les mots qui ont plus d'une occurrence :

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print(word, count)
```

Cela vous donne le résultat prévu (à moins que vous ne vous attendiez à séparer en deux mots « scikit-learn ») :

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
```

```
machine 2  
neural 2  
scikit-learn 2  
r 2
```

Nous examinerons des méthodes d'extraction plus élaborées au chapitre 21.

Remarque

Notez qu'il faudra penser à traiter les mots vides de sens, comme le « de » en français (*ndt*).

À suivre

Ce premier jour s'est plutôt bien déroulé ! Épuisé, vous vous glissez hors de l'immeuble avant que quelqu'un ne vienne encore vous solliciter. Reposez-vous bien, car demain, ce sera la journée d'intégration des nouveaux employés. Oui, vous venez de vivre une journée de travail bien remplie avant même de participer à la journée d'intégration. Il faudra en parler au DRH !

2

Cours accéléré de Python

Les gens sont encore dingues de Python vingt-cinq ans après, ce que j'ai du mal à croire.

Michael Palin

Tous les nouveaux embauchés chez DataSciencester sont tenus d'assister à la journée d'intégration, dont la partie la plus intéressante consiste en un cours accéléré de Python.

Ce n'est pas un tutoriel Python complet. Il s'agit seulement de souligner les parties du langage qui nous concernent le plus (dont certaines sont rarement mises en avant dans les tutoriels Python). Si vous n'avez encore jamais utilisé Python, vous complèterez sans doute avantageusement ce qui suit avec un tutoriel pour débutant de votre choix.

Découvrir Python

Obtenir Python

Remarque

Sachant que le processus d'installation des éléments peut évoluer – ce qu'un ouvrage ne peut plus faire une fois imprimé – des instructions actualisées sur l'installation de Python sont disponibles dans le référentiel GitHub de l'ouvrage^a. Consultez-les si les instructions présentées ici sont inopérantes dans votre cas.

a. <https://github.com/joelgrus/data-science-from-scratch/blob/master/INSTALL.md>

Vous pouvez télécharger le langage depuis [Python.org](https://www.python.org/)¹. Toutefois, si vous ne l'avez pas encore, je vous recommande plutôt d'installer la distribution Anaconda², car elle inclut la plupart des bibliothèques nécessaires en data science.

À l'heure où j'écrivais la première édition de ce livre, la version 2.7 de Python avait la préférence de la plupart des data scientists. Ladite édition s'appuyait, par conséquent, sur Python 2.7.

Cependant, ces dernières années, tous ou presque ont migré vers Python 3. Les versions récentes du langage présentent de nombreuses caractéristiques qui facilitent la rédaction d'un code propre. Ainsi, nous tirerons avantage de caractéristiques uniquement disponibles dans les versions 3.6 et ultérieures. En outre, nombre de bibliothèques utiles ne sont plus prises en charge depuis Python 2.7, motif supplémentaire de passer aux versions ultérieures.

Les environnements virtuels

À partir du chapitre suivant, nous utiliserons la bibliothèque `matplotlib` pour générer des tracés et des graphiques. Elle ne fait pas partie du noyau fondamental de Python. Vous devrez l'installer vous-même. Chaque projet de data science que vous entamerez nécessitera de combiner plusieurs bibliothèques externes, parfois à des versions spécifiques qui se distinguent des versions que vous utilisiez pour d'autres projets. Si vous ne possédiez qu'une seule installation Python, ces bibliothèques entreraient en conflit et vous occasionneraient toutes sortes de problèmes.

La solution standard consiste à utiliser des environnements virtuels. Ils constituent des environnements de test isolés Python de type sandbox. Ils conservent leurs propres versions des bibliothèques Python et, selon la configuration de l'environnement, du langage lui-même.

Dans cette section, je vais vous expliquer comment fonctionnent les environnements Anaconda. Si vous n'utilisez pas Anaconda, vous pouvez soit utiliser le module intégré `venv`³, soit installer `virtualenv`⁴ ; dans ces cas, il vous faudra suivre leurs instructions.

Pour créer un environnement virtuel (Anaconda), il vous suffit de procéder comme suit.

```
# créer un environnement Python 3.6 nommé "dsfs"
conda create -n dsfs python=3.6
```

Suivez les invites et vous obtiendrez l'environnement virtuel `dsfs`, avec les commentaires suivants :

```
# pour activer cet environnement, utiliser :
# > source activate dsfs
#
# pour désactiver un environnement actif, utiliser :
# > source deactivate
```

1. <https://www.python.org/>

2. <https://www.anaconda.com/products/individual>

3. <https://docs.python.org/3/library/venv.html>

4. <https://virtualenv.pypa.io/en/latest/>

Activez l'environnement comme indiqué. Votre invite de commande doit changer pour indiquer l'environnement actif. Sur mon ordinateur, elle se présente comme suit.

```
(dsfs) ip-10-0-0-198:~ joelg$
```

Tant que cet environnement est actif, toute bibliothèque que vous installerez sera disponible uniquement dans `dsfs`. Une fois l'étude du présent ouvrage terminée, pour revenir à vos propres projets, vous devrez créer votre propre environnement pour ces derniers.

À présent que vous disposez de votre environnement, il s'avère intéressant d'installer IPython⁵, à savoir un interpréteur de commandes – ou *shell* – Python offrant une fonctionnalité complète :

```
python -m pip install ipython
```

Remarque

Anaconda est livré avec `conda`, un gestionnaire de paquets. Vous pouvez aussi tout simplement utiliser `pip`, le gestionnaire de paquets standard de Python, ce que nous ferons.

Le reste du présent ouvrage considère que vous avez créé et activé un tel environnement virtuel Python 3.6 (vous pouvez, bien sûr, le nommer à votre guise). Les chapitres ultérieurs sont susceptibles de s'appuyer sur les bibliothèques que vous avez été antérieurement invité à installer.

Et pour faire montre de discipline, vous travaillerez systématiquement dans un environnement virtuel, sans jamais utiliser l'installation Python « de base ».

Les principes zen de Python

Python contient une description plutôt zen de ses principes de conception⁶, que vous pouvez obtenir à partir de l'interpréteur de commandes – ou *shell* – Python en tapant `"import this"`.

Parmi les principes qui ont fait couler le plus d'encre, on trouve celui-ci : il devrait y avoir une – et de préférence une seule – manière évidente de coder.

Le code écrit dans le respect de ladite manière « évidente » (qui ne l'est pas nécessairement aux yeux d'un débutant) est souvent qualifié de « pythonique ». Bien que ce livre ne soit pas dédié à ce langage, nous ne manquerons pas de comparer de temps en temps des solutions pythoniques et non pythoniques permettant de réaliser la même chose, en favorisant généralement les premières.

5. <http://ipython.org/>

6. <https://legacy.python.org/dev/peps/pep-0020/>

Plusieurs autres de ces principes portent sur l'esthétique :

- Beau vaut mieux que laid.
- Explicite vaut mieux qu'implicite.
- Simple vaut mieux que compliqué.

Tous ces principes représentent des idéaux que nous nous efforçons de véhiculer dans notre code.

La mise en page par les espaces

De nombreux langages utilisent les accolades pour délimiter des blocs de code. Python, lui, a recours à l'indentation :

```
# Le signe dièse marque le début d'un commentaire. En tant que langage,
# Python ignore les commentaires, mais ils sont utiles à toute personne
# amenée à lire le code.
for i in [1, 2, 3, 4, 5]:
    print(i) # première ligne du bloc "for i"
    for j in [1, 2, 3, 4, 5]:
        print(j) # première ligne du bloc "for j"
        print(i + j) # dernière ligne du bloc "for j"
    print(i) # dernière ligne du bloc "for i"
print("done looping")
```

Ce choix rend le code Python très lisible, mais il signifie aussi que vous devez faire très attention à votre mise en page.

Avertissement

En matière d'indentation, les programmeurs débattent souvent quant à l'usage des tabulations ou des espaces. Dans de nombreux langages, cette question a peu d'importance. Toutefois, Python considère les espaces et les tabulations comme deux modes d'indentation distincts et ne sera pas en mesure d'exécuter votre code si vous les mélangez. Lorsque vous rédigez en Python, vous devez systématiquement utiliser des espaces, jamais de tabulations. Si vous rédigez un code dans un éditeur, vous pouvez configurer ce dernier de sorte que la touche de tabulation du clavier insère uniquement des espaces.

Le caractère d'espacement est ignoré à l'intérieur des parenthèses et des crochets, ce qui est parfois utile pour les longues formules :

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12
+ 13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

Cela facilite également la lecture du code :

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
easier_to_read_list_of_lists = [[1, 2, 3],
                                [4, 5, 6],
                                [7, 8, 9]]
```

Vous pouvez aussi utiliser une barre oblique inverse pour indiquer qu'une instruction continue sur la ligne suivante, mais la chose est rare :

```
two_plus_three = 2 + \  
3
```

L'utilisation du caractère d'espacement rend difficile d'effectuer des copier-coller de code dans le *shell* Python ordinaire. Par exemple, si vous essayez de copier le code suivant :

```
for i in [1, 2, 3, 4, 5]:  
  
    # remarquez la ligne vide :  
    print(i)  
    # vous recevrez ce message :  
IndentationError: expected an indented block
```

En effet, pour l'interpréteur, la ligne vide signale la fin de la boucle `for`.

IPython possède une fonction magique `%paste` qui copie correctement tout ce qui se trouve dans le presse-papiers, y compris le caractère d'espacement. À elle seule, elle constitue une excellente raison d'utiliser IPython.

Les fonctions et modules

Les fonctions

Une fonction est une règle qui s'applique à des entrées pour retourner un résultat en sortie. En Python, nous définissons les fonctions avec le mot-clé `def` :

```
def double(x):  
    """C'est ici que vous placez un docstring facultatif pour expliquer  
    ce que fait cette fonction. Par exemple, cette fonction multiplie ses  
    entrées par 2."""  
    return x * 2
```

Les fonctions Python sont des objets de première classe, ce qui veut dire que nous pouvons les affecter à des variables et les passer à des fonctions comme n'importe quel autre argument :

```
def apply_to_one(f):  
    """appelle la fonction f avec 1 comme argument"""  
    return f(1)  
  
my_double = double # porte sur la fonction précédemment définie  
x = apply_to_one(my_double) # égale 2
```

On peut aussi facilement créer une petite fonction anonyme, dite `lambda` :

```
y = apply_to_one(lambda x: x + 4) # égale 5
```

Il est possible d'affecter une fonction `lambda` à une variable, même si les développeurs vous conseilleront pour la plupart d'utiliser `def` à la place :

```
another_double = lambda x: 2 * x # à éviter

def another_double(x):
    """à faire à la place"""
    return 2 * x
```

Les arguments (paramètres) d'une fonction peuvent aussi recevoir des valeurs par défaut. Vous spécifiez alors explicitement les arguments uniquement lorsque vous voulez leur donner une valeur différente de celle par défaut :

```
def my_print(message = "my default message"):
    print(message)

my_print("hello") # affiche 'hello'
my_print() # affiche 'my default message' (« mon message par défaut »)
```

Il est parfois utile de spécifier des arguments par leur nom :

```
def full_name(first = "what's-his-name", last = "Something"):
    return first + " " + last

full_name("Joel", "Grus") # "Joel Grus"
full_name("Joel") # "Joel Quelque-chose"
full_name(last="Grus") # "Quel est son nom ? Grus"
```

Nous allons créer beaucoup de fonctions.

Les modules

Certaines fonctions de Python ne sont pas chargées par défaut. C'est le cas de fonctions à la fois intégrées au langage et tierces à télécharger à part. Pour utiliser les secondes, vous devez importer les modules correspondants.

Une méthode consiste à importer le module proprement dit :

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Ici, `re` est le module qui contient des fonctions et des constantes pour travailler avec des expressions rationnelles. Avec ce type d'importation, vous devrez préfixer avec `re` les fonctions du module que vous utiliserez.

Si votre code présente déjà un `re` différent, il faudra utiliser un alias :

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

Vous pouvez aussi opter pour cette solution si votre module a un nom peu maniable ou si vous devez le taper très souvent. Par exemple, quand vous voulez visualiser des données avec `matplotlib`, il existe la convention suivante :

```
import matplotlib.pyplot as plt
plt.plot(...)
```

Si vous avez seulement besoin de quelques fonctions ou valeurs spécifiques d'un module, vous pouvez les importer explicitement et les utiliser sans les qualifier :

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

Si vous n'étiez pas très bon, vous pourriez être tenté d'importer l'intégralité du contenu d'un module dans votre espace de noms, mais cela risquerait d'écraser par inadvertance des variables déjà définies :

```
match = 10
from re import * # oh oh, re a une fonction du même nom
print(match) # "<function match at 0x10281e6a8>"
```

Néanmoins, ce n'est pas votre cas ; vous ne ferez donc jamais une chose pareille.

Les exceptions

En cas d'erreur, Python déclenche une exception. Si cette dernière n'est pas traitée, elle provoquera le plantage du programme. Vous pouvez gérer les exceptions avec `try` et `except` :

```
try:
    print(0 / 0)
except ZeroDivisionError:
    print "cannot divide by zero"
```

Alors que dans de nombreux langages les exceptions sont considérées comme des fautes, en Python il n'y a pas de honte à les utiliser pour rendre le code plus propre ; ce que nous nous permettrons à l'occasion.

Types complexes de données

Les chaînes de caractères

Les chaînes de caractères (*string* en anglais) sont délimitées par des guillemets simples ou doubles. Dans les deux cas, ceux-ci doivent être appariés :

```
single_quoted_string = 'data science'  
double_quoted_string = "data science"
```

Python utilise la barre oblique inverse pour coder les caractères spéciaux. Par exemple :

```
tab_string = "\t" # représente le caractère de tabulation  
len(tab_string) # vaut 1
```

Si vous avez besoin d'une barre oblique inverse en tant que caractère (par exemple, pour nommer un répertoire sous Windows ou dans une expression rationnelle), vous devez créer des chaînes de caractères brutes (*raw* en anglais) à l'aide de `r " "` :

```
not_tab_string = r "\t" # représente les caractères '\' et 't'  
len(not_tab_string) # vaut 2
```

Vous pouvez créer des chaînes de caractères sur plusieurs lignes à l'aide de trois paires de doubles guillemets emboîtées :

```
multi_line_string = """Ceci est la première ligne.  
Et ceci est la deuxième ligne.  
Et ceci est la troisième ligne."""
```

La fonction *f-string* est apparue avec Python 3.6. Elle fournit un moyen simple de remplacer des valeurs dans des chaînes. Par exemple, nous avons un prénom et un nom fournis séparément :

```
first_name = "Joel"  
last_name = "Grus"
```

Nous souhaitons les combiner en un nom complet (prénom et nom). Il existe plusieurs manières de construire une telle chaîne (*full_name*) :

```
full_name1 = first_name + " " + last_name # concaténation des chaînes  
full_name2 = "{0} {1}".format(first_name, last_name) # formatage de chaînes
```

Toutefois, la fonction `f-string` est bien plus simple :

```
full_name3 = f "{first_name} {last_name}"
```

Nous lui donnerons la préférence au fil de cet ouvrage.

Les listes

La liste est certainement la structure de données la plus fondamentale de Python. Une liste est simplement une collection ordonnée. Elle est assez similaire à ce que d'autres langages appellent un tableau, mais avec des fonctionnalités en plus.

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [integer_list, heterogeneous_list, []]

list_length = len(integer_list) # égale 3
list_sum = sum(integer_list) # égale 6
```

Vous pouvez initialiser ou récupérer le *n*-ième élément de la liste au moyen de crochets :

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

zero = x[0] # égale 0 ; les listes sont indexées à partir de 0
one = x[1] # égale 1
nine = x[-1] # égale 9, version 'pythonique' pour le dernier élément
eight = x[-2] # égale 8, version 'pythonique' pour l'avant-dernier élément
x[0] = -1 # maintenant x vaut [-1, 1, 2, 3, ..., 9]
```

Vous pouvez aussi utiliser des crochets et des symboles deux-points pour découper des listes en tranches : ainsi, la tranche `[i:j]` spécifie tous les éléments de *i* (inclus) à *j* (non inclus). Si vous ne précisez pas le début de la tranche, vous découperez la liste depuis son début. Si vous ne précisez pas la fin de la tranche, vous découperez la liste jusqu'à sa fin :

```
first_three = x[:3] # [-1, 1, 2]
three_to_end = x[3:] # [3, 4, ..., 9]
one_to_four = x[1:5] # [1, 2, 3, 4]
last_three = x[-3:] # [7, 8, 9]
without_first_and_last = x[1:-1] # [1, 2, ..., 8]
copy_of_x = x[:] # [-1, 1, 2, ..., 9]
```

De la même manière que les chaînes, vous pouvez découper d'autres éléments de type « séquentiel ».

Une tranche peut accepter un troisième argument pour indiquer son pas. Cet argument peut être négatif, auquel cas la chaîne est parcourue à l'envers :

```
every_third = x[::3] # [-1, 3, 6, 9]
five_to_three = x[5:2:-1] # [5, 4, 3]
```

Python dispose par ailleurs d'un opérateur pour vérifier l'appartenance d'un élément à une liste :

```
1 in [1, 2, 3] # Vrai (True)
0 in [1, 2, 3] # Faux (False)
```

Cette méthode de contrôle va examiner les éléments de la liste un par un. Vous ne devez donc l'employer que si vous savez que votre liste est vraiment toute petite (ou si vous ne vous intéressez pas à la durée du contrôle).

On peut aisément concaténer des listes : si vous souhaitez modifier une liste en place, vous pouvez recourir à une fonction d'extension – *extend* – pour ajouter des éléments issus d'une autre liste :

```
x = [1, 2, 3]
x.extend([4, 5, 6]) # x vaut à présent [1, 2, 3, 4, 5, 6]
```

Si vous ne voulez pas modifier *x*, vous pouvez utiliser l'addition de listes :

```
x = [1, 2, 3]
y = x + [4, 5, 6] # y vaut [1, 2, 3, 4, 5, 6] ; x est inchangé
```

Le plus souvent, on ajoute les éléments un par un à une liste :

```
x = [1, 2, 3]
x.append(0) # x vaut à présent [1, 2, 3, 0]
y = x[-1] # égale 0
z = len(x) # égale 4
```

Souvent, il est pratique de décompresser (*unpack* en anglais) des listes en affectant chaque élément à une variable si vous savez combien d'éléments la liste contient :

```
x, y = [1, 2] # maintenant x vaut 1, y vaut 2
```

Notez que vous risquez de déclencher une erreur *valueError* si vous n'avez pas le même nombre d'éléments des deux côtés.

Enfin, il est d'usage de marquer d'un tiret bas une valeur dont vous allez vous débarrasser :

```
_, y = [1, 2] # maintenant y vaut 2, peu importe le premier élément
```


Les tuples

Les tuples sont les cousins invariants des listes. Vous pouvez faire avec un tuple à peu près tout ce qu'il est possible de faire avec une liste, du moment que vous ne la modifiez pas. On décrit un tuple à l'aide de parenthèses (ou sans rien) au lieu de crochets :

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3 # my_list vaut désormais [1, 3]
try:
    my_tuple[1] = 3
except TypeError:
    print("cannot modify a tuple")
```

Les tuples sont un moyen pratique pour retourner des valeurs multiples depuis des fonctions :

```
def sum_and_product(x, y):
    return (x + y), (x * y)

sp = sum_and_product(2, 3) # sp vaut (5, 6)
s, p = sum_and_product(5, 10) # s vaut 15, p vaut 50
```

Les tuples (et les listes) peuvent aussi s'utiliser pour des affectations multiples :

```
x, y = 1, 2 # maintenant x vaut 1, y vaut 2
x, y = y, x # manière pythonique de permuter les variables ; maintenant x vaut 2,
            # y vaut 1
```

Les dictionnaires

Le dictionnaire est une autre structure de données fondamentale. Elle associe des valeurs (*values*) à des clés (*keys*) et vous permet de récupérer rapidement la valeur qui correspond à une clé donnée :

```
empty_dict = {} # pythonique
empty_dict2 = dict() # moins pythonique
grades = {"Joel": 80, "Tim": 95} # écriture littérale
```

Vous pouvez chercher la valeur d'une clé en utilisant des crochets :

```
joels_grade = grades["Joel"] # égale 80
```

Cependant, vous déclencherez une erreur `KeyError` si vous demandez une clé qui n'existe pas dans le dictionnaire :

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print("no grade for Kate!")
```

Vous pouvez vérifier l'existence d'une clé à l'aide de l'opérateur `in` :

```
joel_has_grade = "Joel" in grades # Vrai
kate_has_grade = "Kate" in grades # Faux
```

Ce contrôle d'appartenance est rapide, même dans le cas de dictionnaires volumineux. Les dictionnaires ont une méthode `get` qui retourne une valeur par défaut (au lieu de déclencher une exception) lorsque vous cherchez une clé qui n'existe pas :

```
joels_grade = grades.get("Joel", 0) # égale 80
kates_grade = grades.get("Kate", 0) # égale 0
no_ones_grade = grades.get("No one") # par défaut vaut None
```

On associe les paires clé-valeur en utilisant les mêmes crochets :

```
grades["Tim"] = 99 # remplace l'ancienne valeur
grades["Kate"] = 100 # ajoute une troisième entrée
num_students = len(grades) # égale 3
```

Comme nous l'avons vu au chapitre 1, les dictionnaires servent à représenter des données structurées :

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience",
                  "#awesome", "#yolo"]
}
```

Cela dit, nous étudierons sous peu une meilleure approche. Outre la recherche de clés spécifiques, il est possible de les rechercher toutes :

```
tweet_keys = tweet.keys() # liste des clés
tweet_values = tweet.values() # liste des valeurs
tweet_items = tweet.items() # liste des tuples (clé/valeur)
"user" in tweet_keys # Vrai mais non pythonique
"user" in tweet # méthode pythonique de recherche de clés
"joelgrus" in tweet_values # Vrai (lent mais seul moyen de contrôle)
```

Les clés de dictionnaire doivent être *hashables* ; notons, en particulier, qu'il n'est pas possible d'utiliser des listes comme clés. Si vous avez besoin d'une clé correspondant à plusieurs éléments, il faut utiliser un tuple ou imaginer une manière de transformer la clé en une chaîne de caractères.

Defaultdict

Imaginons que vous essayez de compter les mots dans un document. Une méthode simple consiste à créer un dictionnaire dont les clés sont les mots et les valeurs sont les nombres d'occurrences de ces mots. Tandis que vous passez sur chaque mot, vous pouvez incrémenter le compteur dès lors que le mot est déjà dans le dictionnaire. Dans la négative, vous l'y ajoutez :

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

Vous pouvez également utiliser la tactique « mieux vaut s'excuser que demander la permission » et vous contenter de gérer l'exception quand vous cherchez une clé manquante :

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

Une troisième méthode consiste à utiliser `get`, qui gère en douceur le cas des clés manquantes :

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Ces méthodes sont assez peu pratiques, ce qui justifie l'existence de `defaultdict`. Un `defaultdict` est comme un dictionnaire normal, à la différence que, lorsque vous cherchez une clé qu'il ne contient pas, il ajoute d'abord une valeur pour cette clé à l'aide d'une fonction sans argument que vous avez fournie lors de sa création. Pour utiliser des `defaultdict`, il faut les importer depuis le module `collections` :

```
from collections import defaultdict
word_counts = defaultdict(int) # int() produit 0
for word in document:
    word_counts[word] += 1
```

Les `defaultdict` sont également très utiles avec `list` ou `dict`, voire avec vos propres fonctions :

```
dd_list = defaultdict(list) # list() produit une liste vide
dd_list[2].append(1) # maintenant dd_list contient {2: [1]}

dd_dict = defaultdict(dict) # dict() produit un dict vide
dd_dict["Joel"]["City"] = "Seattle" # {"Joel": {"City": "Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1 # à présent dd_pair contient {2: [0, 1]}
```

Cette approche est très utile lorsque vous utilisez des dictionnaires afin de collecter des résultats selon une certaine clé sans vérifier à chaque fois si cette clé existe déjà.

Les compteurs

Un compteur (*counter*) transforme une séquence de valeurs en un objet de style `defaultdict(int)` qui fait correspondre des clés à des comptages :

```
from collections import Counter
c = Counter([0, 1, 2, 0]) # c vaut (à la base) {0: 2, 1: 1, 2: 1}
```

C'est là une manière très simple de résoudre le problème `word_counts` précédent :

```
# rappel : un document est une liste de mots
word_counts = Counter(document)
```

Une instance de compteur dispose d'une méthode `most_common` bien pratique :

```
# affiche les 10 mots les plus courants et leur nombre d'occurrences
for word, count in word_counts.most_common(10):
    print(word, count)
```

Les ensembles

L'ensemble (*set*) est une autre structure de données utile : il permet de représenter une collection d'éléments distincts.

Pour définir un ensemble, vous pouvez en énumérer les éléments entre accolades :

```
primes_below_10 = {2, 3, 5, 7}
```

Toutefois, ce procédé ne fonctionne pas avec les ensembles vides ; en effet, l'expression {} désigne déjà un dict vide. Dans ce cas, vous devrez utiliser `set()` :

```
s = set()
s.add(1) # s vaut maintenant {1}
s.add(2) # s vaut maintenant {1, 2}
s.add(2) # s vaut toujours {1, 2}
x = len(s) # égale 2
y = 2 in s # Vrai
z = 3 in s # Faux
```

Nous utiliserons cette structure de données principalement pour deux raisons. La première tient à ce que l'opérateur `in` est très rapide sur les ensembles.

Si nous disposons d'une grande collection d'éléments que nous voulons utiliser pour un test d'appartenance, un ensemble sera préférable à une liste :

```
stopwords_list = ["a", "an", "at"] + hundreds_of_other_words + ["yet", "you"]
"zip" in stopwords_list # faux, mais il faut contrôler chaque élément
stopwords_set = set(stopwords_list)
"zip" in stopwords_set # très rapide à contrôler
```

Une deuxième raison est qu'on identifie ainsi les éléments distincts d'une collection :

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list) # 6
item_set = set(item_list) # {1, 2, 3}
num_distinct_items = len(item_set) # 3
distinct_item_list = list(item_set) # [1, 2, 3]
```

Toutefois, nous utiliserons beaucoup moins les ensembles que les dictionnaires et les listes.

Les structures de contrôle

À l'instar de la plupart des langages de programmation, Python permet d'utiliser des branchements conditionnels via le mot-clé `if` :

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

Vous pouvez aussi utiliser la forme ternaire du branchement si-alors-sinon (if-then-else) sur une seule ligne, ce que nous ferons de temps en temps :

```
parity = "even" if x % 2 == 0 else "odd"
```

Python propose une boucle « tant que » (while) :

```
x = 0
while x < 10:
    print(f"{x} is less than 10")
    x += 1
```

Cependant, nous utiliserons plus souvent for et in :

```
# range(10) correspond aux chiffres 0 à 9
for x in range(10):
    print(f"{x} is less than 10")
```

Si vous avez besoin d'utiliser des logiques plus complexes, vous pouvez faire appel à continue et break :

```
for x in range(10):
    if x == 3:
        continue # va immédiatement à l'itération suivante
    if x == 5:
        break # quitte définitivement la boucle
    print(x) # affiche 0, 1, 2 et 4
```

Les éléments booléens

En Python, les éléments booléens fonctionnent comme dans la plupart des langages à ceci près qu'ils sont représentés avec une majuscule initiale :

```
one_is_less_than_two = 1 < 2 # égale True (Vrai)
true_equals_false = True == False # égale False (Faux)
```

Python utilise la valeur None pour indiquer une valeur qui n'existe pas. C'est l'équivalent de null dans d'autres langages :

```
x = None
assert x == None # n'est pas la manière pythonique d'identifier None
assert x is None # est la manière pythonique d'identifier None
```

Python vous permet d'utiliser n'importe quelle valeur quand il attend un élément booléen. Les termes suivants sont tous considérés comme `False` (faux) :

- `False`
- `None`
- `[]` (liste vide)
- `{}` (dictionnaire vide)
- `""` (chaîne vide)
- `set()` (ensemble vide)
- `0`
- `0.0`

Presque tout le reste est considéré comme `True` (vrai). Il est donc facile d'utiliser des instructions conditionnelles (`if`) pour tester si des listes, des chaînes de caractères ou des dictionnaires et autres sont vides. Cela peut aussi parfois induire des bogues délicats à résoudre si vous ne gardez pas ce comportement à l'esprit :

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

Il existe une manière plus simple (mais éventuellement plus troublante) de faire la même chose :

```
first_char = s and s[0]
```

En effet, `and` retourne sa deuxième valeur quand la première est vraie et la première quand elle ne l'est pas.

De la même façon, si `x` est un nombre ou éventuellement `None` :

```
safe_x = x or 0
```

est sans discussion un nombre, même si :

```
safe_x = x if x is not None else 0
```

est de toute évidence plus lisible.

Python propose une fonction `all` qui accepte une liste en entrée et retourne `True` si chaque élément est vrai et une fonction `any` qui retourne `True` si au moins un des éléments est vrai :

```
all([True, 1, {3}]) # True car tous les éléments sont vrais
all([True, 1, {}])  # False car {} est faux
any([True, 1, {3}]) # True car True est vrai
all([])             # True car aucun élément de la liste n'est faux
any([])             # False car aucun élément de la liste n'est vrai
```

Trier

Toute liste Python possède une méthode `sort` permettant un tri en place. Si vous ne voulez pas modifier votre liste, utilisez la fonction `sorted`, qui retourne une nouvelle liste :

```
x = [4, 1, 2, 3]
y = sorted(x) # y vaut [1, 2, 3, 4], x est inchangée
x.sort() # maintenant x vaut [1, 2, 3, 4]
```

Par défaut, `sort` et `sorted` trient une liste du plus petit au plus grand par comparaison naïve des éléments entre eux. Si vous voulez trier du plus grand au plus petit, spécifiez le paramètre `reverse=True`.

Au lieu de comparer les éléments eux-mêmes, vous pouvez comparer les résultats d'une fonction que vous indiquerez avec une clé :

```
# trie selon la valeur absolue, de la plus grande à la plus petite
x = sorted([-4, 1, -2, 3], key=abs, reverse=True) # vaut [-4, 3, -2, 1]
# trie les mots et les compteurs du nombre le plus grand au plus petit
wc = sorted(word_counts.items(),
            key=lambda word_and_count: word_and_count[1],
            reverse=True)
```

Les listes en compréhension

Il est fréquent de vouloir transformer une liste en une autre en modifiant des éléments ou en sélectionnant uniquement certains d'entre eux, ou les deux. La manière pythonique consiste alors à utiliser une liste en compréhension (*list comprehension*) :

```
even_numbers = [x for x in range(5) if x % 2 == 0] # [0, 2, 4]
squares = [x * x for x in range(5)] # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers] # [0, 4, 16]
```

De la même manière, les listes peuvent être converties en dictionnaires ou en sets :

```
dict_square_dict = {x: x * x for x in range(5)} # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
square_set = {x * x for x in [1, -1]} # {1}
```

Si vous n'avez pas besoin de récupérer la valeur dans la liste, il est d'usage d'utiliser un tiret bas en guise de variable :

```
zeros = [0 for _ in even_numbers] # présente la même longueur que even_numbers
```


Une liste en compréhension peut contenir plusieurs boucles `for` :

```
# 100 paires (0,0) (0,1) ... (9,8), (9,9):
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]
```

Certaines boucles `for` sont susceptibles de réutiliser les résultats des précédentes :

```
# uniquement des paires avec x < y :
increasing_pairs = [(x, y)
                    for x in range(10)
                    for y in range(x + 1, 10)]
```

Nous ferons un usage immodéré des listes en compréhension.

Les tests automatisés et instructions `assert`

Nous, data scientists, rédigeons des montagnes de code. Comment être certain que ce dernier est correct ? S'il est possible d'utiliser les types (voir plus loin), une autre méthode fait appel à des tests automatisés.

Des structures sophistiquées permettent de rédiger et d'exécuter des tests. Toutefois, dans le présent ouvrage, nous nous bornerons à utiliser des instructions `assert`. Avec elles, votre code lèvera une `AssertionError` si la condition que vous spécifiez se révèle fausse :

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 should equal 2 but didn't"
```

Comme vous le constatez dans le deuxième cas, vous avez la possibilité d'ajouter un message qui s'affichera si l'assertion échoue.

Formuler une assertion du type `1 + 1 == 2` n'a rien d'intéressant. Ce qui l'est davantage, c'est de formuler l'assertion que les fonctions que vous rédigez font ce qui en est attendu :

```
def smallest_item(xs):
    return min(xs)

assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

Au fil de ce livre, nous utiliserons `assert` de cette manière. Une pratique optimale, et que je recommande fortement, consiste à en faire un usage immodéré dans votre propre code.

Remarque

Si vous examinez le code qui accompagne cet ouvrage sur GitHub, vous constaterez qu'il contient un très grand nombre d'instructions `assert`. Ainsi, j'ai confiance dans le fait que le code que j'ai rédigé pour vous est correct.

Un autre usage moins courant consiste à formuler des assertions relatives à des entrées de fonctions :

```
def smallest_item(xs):  
    assert xs, ""une liste vide n'a pas de plus petit élément""  
    return min(xs)
```

La programmation orientée objet

Comme de nombreux langages, Python vous permet de définir des classes pour encapsuler des données et les fonctions associées. Nous les utiliserons parfois pour rendre le code plus propre et plus simple. Pour les expliquer, le plus simple consiste à construire un exemple commenté.

Ici, nous allons construire une classe représentant un « compteur manuel ». Cet outil s'utilise pour compter le nombre de personnes qui se sont présentées à l'entrée d'une conférence sur les « thèmes avancés en data science ».

Il mémorise le comptage, s'active pour être incrémenté, vous permet de lire le comptage (`read_count`) et peut être remis à zéro. En pratique réelle, les cylindres de l'outil vont de 9 999 à 0 000, mais ici nous ne nous en inquiéterons pas.

Pour définir une classe, vous utilisez le mot-clé `class` et un nom à syntaxe *PascalCase* :

```
class CountingClicker:  
    ""une classe peut/doit posséder un docstring""
```

Une classe contient zéro fonction membre ou plus. Par convention, chacune accepte un premier paramètre, `self`, qui désigne l'instance de classe en cours de traitement.

Normalement, une classe présente un constructeur, nommé `__init__`. Il accepte tous les paramètres dont vous avez besoin pour construire une instance de votre classe et exécute toute configuration nécessaire :

```
def __init__(self, count = 0):  
    self.count = count
```

Bien que le nom du constructeur soit bizarre, nous construisons les instances du compteur manuel en utilisant simplement le nom de classe :

```
clicker1 = CountingClicker() # initialisé à 0  
clicker2 = CountingClicker(100) # commence par count = 100  
clicker3 = CountingClicker(count=100) # plus explicite que le précédent
```

Remarquez que le nom de méthode `__init__` commence et se termine par deux tirets de soulignement. Ces méthodes « magiques » sont parfois appelées méthodes *dunder* (de l'anglais *double-underscore*) et représentent des comportements « spéciaux ».

Remarque

Les méthodes de classe dont les noms commencent par un tiret de soulignement sont par convention considérées comme « privées ». Les utilisateurs de la classe ne sont pas censés les appeler directement. Toutefois, Python n'empêchera pas les utilisateurs de le faire.

Autre méthode de ce type, `__repr__` reproduit la représentation en chaîne d'une instance de classe :

```
def __repr__(self):
    return f"CountingClicker(count={self.count})"
```

Et enfin, nous devons mettre en œuvre l'API publique de notre classe :

```
def click(self, num_times = 1):
    """cliquez sur le compteur un certain nombre de fois"""
    self.count += num_times

def read(self):
    return self.count

def reset(self):
    self.count = 0
```

Une fois celle-ci définie, utilisons une instruction `assert` pour rédiger quelques cas de test destinés à notre compteur manuel :

```
clicker = CountingClicker()
assert clicker.read() == 0, # le compteur devrait commencer à 0
clicker.click()
clicker.click()
assert clicker.read() == 2, # le compteur devrait être à 2
clicker.reset()
assert clicker.read() == 0, # le compteur devrait revenir à 0
```

Forts de tests de ce type, nous sommes confiants dans une exécution de notre code conforme à sa conception de départ. En outre, nous pourrions créer occasionnellement des sous-classes qui hériteront d'une partie de la fonctionnalité d'une classe parente. Par exemple, nous pourrions créer un compteur non réinitialisable en utilisant `CountingClicker` en tant que classe de base, puis en passant outre la méthode de réinitialisation :

```
# une sous-classe hérite de tout le comportement de sa classe parente
class NoResetClicker(CountingClicker):
```

```
# cette classe dispose de toutes les méthodes de CountingClicker, sauf la réinitialisation
def reset(self):
    pass

clicker2 = NoResetClicker()
assert clicker2.read() == 0
clicker2.click()
assert clicker2.read() == 1
clicker2.reset()
assert clicker2.read() == 1, # pas de réinitialisation
```

Les générateurs et itérateurs

Le bon côté d'une liste, c'est que vous pouvez en extraire des éléments spécifiques grâce à leur index. Cependant, une liste d'un milliard d'éléments risque d'occuper une place considérable en mémoire. Or, si vous traitez ces éléments un à un (et non en lot), il n'y a aucune raison de les conserver à portée de main. Ainsi, si vous n'avez besoin, finalement, que de quelques-uns des premiers éléments, conserver l'intégralité de la liste constituera un énorme gaspillage de ressources.

Le plus souvent, il suffira d'appliquer une itération à la collection en utilisant `for` et `in`. Un générateur permet de réaliser ces itérations sur des listes dont les valeurs ne sont produites qu'à la demande ; on parle alors d'évaluation paresseuse (par opposition à l'évaluation immédiate).

Pour créer un générateur, une solution consiste à utiliser des fonctions et l'opérateur `yield` :

```
def generate_range(n):
    i = 0
    while i < n:
        yield i # chaque appel à yield produit une valeur du générateur
        i += 1
```

La boucle suivante consomme les valeurs auxquelles `yield` est appliqué, une par une, jusqu'à épuisement :

```
for i in generate_range(10):
    print(f"i: {i}")
```

Remarque

En fait, `range` est elle-même paresseuse ; il n'est donc pas nécessaire de procéder ainsi.

Avec un générateur, vous pouvez même créer une séquence infinie :

```
def natural_numbers():
    """retourne 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

Toutefois, vous ne devriez pas l'utiliser dans une boucle sans mettre en place une logique d'interruption (`break`).

Conseil

Le revers de la médaille, c'est que vous ne pouvez réaliser une itération sur un générateur qu'une seule fois. Si vous devez repasser plusieurs fois en boucle sur un élément, il faudra recréer le générateur à chaque fois ou faire appel à une liste. Si la génération des valeurs est coûteuse, faire appel à une liste s'avère bénéfique.

Une autre solution pour créer des générateurs consiste à utiliser des expressions `for` entre parenthèses :

```
evens_below_20 = (i for i in generate_range(20) if i % 2 == 0)
```

Un tel générateur en compréhension sera inopérant tant que vous ne lui appliquez aucune itération (en utilisant `for` ou `next`). Nous pouvons faire appel à cette construction pour élaborer des *pipelines* (« tuyaux ») de traitement de données.

```
# aucun de ces calculs ne fait quoi que ce soit en l'absence d'itération
data = natural_numbers()
evens = (x for x in data if x % 2 == 0)
even_squares = (x ** 2 for x in evens)
even_squares_ending_in_six = (x for x in even_squares if x % 10 == 6)
# et ainsi de suite
```

Il n'est pas rare, lorsque vous itérez sur une liste ou un générateur, que vous souhaitiez obtenir non seulement les valeurs, mais aussi leurs index. Pour ce cas assez courant, Python met à disposition une fonction d'énumération. Celle-ci transforme les valeurs en paires (index, valeur) :

```
names = ["Alice", "Bob", "Charlie", "Debbie"]
# non pythonique
for i in range(len(names)):
    print(f"name {i} is {names[i]}")

# pas pythonique non plus
i = 0
for name in names:
```

```
print(f"name {i} is {names[i]}")
i += 1

# pythonique
for i, name in enumerate(names):
    print(f"name {i} is {name}")
```

Nous utiliserons beaucoup ce procédé.

Les valeurs aléatoires

Au cours de notre apprentissage de la data science, nous aurons souvent besoin de générer des nombres aléatoires, grâce au module `random` :

```
import random
random.seed(10) # pour obtenir le même résultat à chaque fois
# random.random() produit des nombres uniformément répartis entre 0 et 1
# c'est la fonction aléatoire que nous utiliserons le plus souvent
four_uniform_randoms = [random.random() for _ in range(4)]
# [0.5714025946899135,
# random.random() produit des nombres
# 0.4288890546751146,
# 0.5780913011344704,
# 0.2060982321395017]
```

Le module `random` produit en fait des nombres pseudo-aléatoires (c'est-à-dire déterministes) selon un état interne que vous pouvez initialiser avec la fonction `random.seed` si vous voulez des résultats reproductibles :

```
random.seed(10) # initialise la valeur de départ à 10
print(random.random()) # 0.57140259469
random.seed(10) # réinitialise la valeur de départ à 10
print(random.random()) # 0.57140259469 à nouveau
```

Nous utiliserons parfois `random.randrange`, qui prend soit un argument, soit deux, et retourne un élément choisi aléatoirement dans la plage `range()` correspondante :

```
random.randrange(10) # choix dans range(10) = [0, 1, ..., 9]
random.randrange(3, 6) # choix dans range(3, 6) = [3, 4, 5]
```

Il existe quelques autres méthodes bien pratiques parfois. Par exemple, `random.shuffle` réorganise aléatoirement les éléments d'une liste :

```
up_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(up_to_ten)
print(up_to_ten) # [7, 2, 6, 8, 9, 4, 10, 1, 3, 5] par exemple
```

Si vous devez choisir un élément au hasard dans une liste, utilisez `random.choice` :

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])
# "Bob" par exemple
```

Et si vous avez besoin de choisir au hasard un échantillon d'éléments sans remise (c'est-à-dire sans doublons), servez-vous de `random.sample` :

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)
# [16, 36, 10, 6, 25, 9] par exemple
```

Enfin, pour choisir un échantillon d'éléments avec remise (c'est-à-dire qui autorise les doublons), lancez plusieurs appels à `random.choice` :

```
four_with_replacement = [random.choice(range(10)) for _ in range(4)]
print(four_with_replacement) # [9, 4, 4, 2] par exemple
```

Les expressions rationnelles

Les expressions rationnelles permettent de chercher du texte. Elles sont incroyablement utiles, mais aussi extraordinairement complexes et il faudrait leur consacrer un livre entier. Nous donnerons des explications détaillées les rares fois où nous les rencontrerons. Voici quelques exemples d'utilisation en Python :

```
import re
re_examples = [ # tous sont vrais car
    not re.match("a", "cat"),
    # 'cat' ne commence pas par 'a'
    re.search("a", "cat"),
    # 'cat' contient un 'a'
    not re.search("c", "dog"),
    # 'dog' ne contient pas de 'c'
    3 == len(re.split("[ab]", "carbs")),
    # se coupe en a ou b en ['c', 'r', 's']
    "R-D-" == re.sub("[0-9]", "-", "R2D2"),
    # remplace les chiffres par des tirets
]
assert all(re_examples), "all the regex examples should be True"
```

Point important à remarquer, `re.match` s'assure que le début d'une chaîne correspond ou non à une expression rationnelle, tandis que `re.search` vérifie si toute partie d'une chaîne correspond ou non à une expression rationnelle. À un certain stade, vous confondrez ces deux éléments... et vous en souffrirez. La documentation officielle⁷ entre davantage dans les détails.

7. <https://docs.python.org/3/library/re.html>

Programmation fonctionnelle

Remarque

La première édition de cet ouvrage avait, à ce stade, présenté les fonctions Python suivantes : `partial`, `map`, `reduce` et `filter`. Au cours de mon cheminement vers l'illumination, je me suis rendu compte qu'il vaut mieux les éviter. Leur usage dans l'ouvrage a donc été remplacé par celui des listes en compréhension, des boucles `for` et autres constructions plus pythoniques.

zip et le déballage d'arguments

Nous aurons souvent besoin de la fonction `zip` pour agréger deux éléments itérables ou plus. Elle transforme ainsi plusieurs listes en une liste unique de tuples :

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
# zip est paresseuse ; vous devez donc procéder plus ou moins comme suit
[pair for pair in zip(list1, list2)] # is [('a', 1), ('b', 2), ('c', 3)]
```

Si les listes sont de longueurs différentes, `zip` s'arrête à la fin de la plus courte.

Vous pouvez aussi « déballer » une liste à l'aide d'une astuce bizarre :

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

L'astérisque effectue le déballage d'arguments. Cette technique utilise les éléments des paires comme des arguments distincts à réunir. Le résultat est le même que si vous aviez appelé :

```
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
```

Le déballage d'arguments peut être utilisé avec n'importe quelle fonction :

```
def add(a, b): return a + b

add(1, 2) # retourne 3
try:
    add([1, 2])
except TypeError:
    print("add expects two inputs")
add(*[1, 2]) # retourne 3
```

Cette approche est rarement avantageuse, mais c'est malgré tout une astuce intéressante.

args et kwargs

Supposons que vous vouliez créer une fonction d'ordre supérieur qui reçoit en entrée une certaine fonction `f`, puis retourne une nouvelle fonction qui, pour n'importe quelle entrée, retourne deux fois la valeur de `f` :

```
def doubler(f):
    # nous définissons une nouvelle fonction qui conserve une référence à f
    def g(x):
        return 2 * f(x)
    # et retourne cette nouvelle fonction
    return g
```

Parfois, cela fonctionne :

```
def f1(x):
    return x + 1

g = doubler(f1)
assert g(3) == 8, "(3 + 1) * 2 devrait être égal à 8"
assert g(-1) == 0, "(-1 + 1) * 2 devrait être égal à 0"
```

Cependant, la méthode ne marche plus avec les fonctions qui acceptent plus d'un argument :

```
def f2(x, y):
    return x + y

g = doubler(f2)
try:
    g(1, 2)
except TypeError:
    print("Telle que définie, g n'accepte qu'un seul argument.")
```

Nous avons besoin d'un moyen de spécifier une fonction qui accepte des arguments arbitraires. Pour ce faire, il faut recourir au déballage d'arguments et à un peu de « magie » :

```
def magic(*args, **kwargs):
    print("unnamed args:", args)
    print("keyword args:", kwargs)

magic(1, 2, key="word", key2="word2")
# affiche
# des arguments anonymes : (1, 2)
# des arguments mots-clés : {'key': 'word', 'key2': 'word2'}
```

En fait, quand nous définissons une telle fonction, `args` est un tuple de ses arguments anonymes et `kwargs` est un dict de ses arguments nommés. Le mécanisme peut être inversé si

vous voulez utiliser une liste (ou un tuple) et un dictionnaire pour fournir des arguments à une fonction :

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = {"z": 3}
assert other_way_magic(*x_y_list, **z_dict) == 6, "1 + 2 + 3 devrait être égal à 6"
```

Vous pouvez vous livrer à toutes sortes de bizarreries avec cette astuce ; nous l'utiliserons seulement pour produire des fonctions d'ordre supérieur dont les entrées peuvent accepter des arguments arbitraires :

```
def doubler_correct(f):
    """fonctionne quelle que soit la nature des entrées attendues"""
    def g(*args, **kwargs):
        """quels que soient les arguments fournis à g, les passe via f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
assert g(1, 2) == 6, "doubler devrait fonctionner à présent"
```

En règle générale, votre code sera plus correct et plus lisible si vous vous montrez explicite quant aux types d'arguments que vos fonctions requièrent ; aussi utiliserons-nous des `args` et des `kwargs` uniquement lorsque nous n'aurons pas d'autre choix.

Les annotations de type

Python est un langage à typage dynamique. En d'autres termes, il ne se soucie généralement pas des types des objets tant que nous les utilisons selon des modes valides :

```
def add(a, b):
    return a + b

assert add(10, 5) == 15, "+ est valide pour les nombres"
assert add([1, 2], [3]) == [1, 2, 3], "+ est valide pour les listes"
assert add("hi ", "there") == "hi there", \
    "+ est valide pour les chaînes"

try:
    add(10, "five")
except TypeError:
    print("Impossible d'ajouter un entier à une chaîne")
```

À l'inverse, dans un langage à typage statique, nos fonctions et objets devraient présenter des types spécifiques :

```
def add(a: int, b: int) -> int:
    return a + b
add(10, 5) # vous souhaitez que ceci soit correct
add("hi ", "there") # vous souhaitez que ceci soit incorrect
```

En fait, les versions récentes de Python offrent (en quelque sorte) cette fonctionnalité. La version précédente de la fonction `add` avec les annotations de type `int` est valide à partir de Python 3.6 !

Toutefois, ces annotations de type ne font rien à proprement parler. Vous pouvez toujours utiliser la fonction `add` pour ajouter des chaînes et l'appel à `add(10, "five")` lèvera toujours exactement la même `TypeError`.

Ceci étant, il existe encore (au moins) quatre bonnes raisons d'utiliser des annotations de type dans votre code Python :

- Les types constituent une forme importante de documentation. C'est doublement vrai dans un ouvrage qui utilise un code pour enseigner des concepts théoriques et mathématiques. Comparons les deux segments de fonction suivants :

```
def dot_product(x, y): ...

# imaginons que nous ayons déjà défini la fonction Vector
def dot_product(x: Vector, y: Vector) -> float: ...
```

Je trouve la deuxième version considérablement plus informative ; j'espère que vous aussi. À ce stade, je suis tellement habitué à utiliser les types pour fournir des indications que je trouve Python difficile à lire dès qu'il n'y en a plus.

- Il existe des outils externes (le plus répandu est `mypy`) qui lisent votre code, inspectent les annotations de type et vous informent des erreurs de type avant que vous n'exécutiez votre code. Par exemple, si vous le lancez sur un fichier contenant `add("hi ", "there")`, `mypy` retournera l'avertissement suivant :

```
error: Argument 1 to "add" has incompatible type "str"; expected "int"
```

À l'instar du test `assert`, cette méthode est efficace pour identifier des erreurs au sein de votre code avant même de l'exécuter. Dans cet ouvrage, la narration ne recourt à aucun contrôleur de ce type. Toutefois, en coulisses, j'en utilise un qui contribue à m'assurer que l'ouvrage proprement dit est correct.

- Le fait de réfléchir aux types au sein de votre code vous oblige à concevoir des fonctions et des interfaces plus propres :

```
from typing import Union
def secretly_ugly_function(value, operation): ...
def ugly_function(value: int,
                  operation: Union[str, int, float, bool]) -> int:
    ...
```

Nous avons ici une fonction dont le paramètre de fonctionnement peut être une chaîne (`str`), un entier (`int`), un nombre décimal (`float`) ou un élément booléen (`bool`). Il est très probable que cette fonction sera délicate et difficile à utiliser. Toutefois, la chose devient bien plus claire dès lors que les types sont rendus explicites. Ce faisant, nous sommes contraints d'employer une conception moins maladroite ; nos utilisateurs nous en remercieront.

- Grâce à l'utilisation de types, votre éditeur vous facilitera la tâche, notamment avec la saisie semi-automatique (figure 2–1), et vous épargnera d'agaçantes erreurs de type.

Figure 2–1

VSCoDe (votre éditeur fera probablement la même chose)

```
def f(xs: List[int]) -> None:
```

```
    xs.
```

```
    append
    clear
    copy
    count
```

Parfois, les gens insistent sur le fait que les indications via les types sont précieuses sur les grands projets, mais chronophages sur les petits. Toutefois, sachant que la saisie de telles indications ne nécessite aucun temps supplémentaire et permet à votre éditeur d'en gagner, je maintiens que les indications fournies via les types vous permettent bel et bien de rédiger votre code plus rapidement, même dans le cadre de projets de moindre envergure.

Pour toutes ces raisons, l'ensemble du code présenté dans le reste de cet ouvrage fera appel à des annotations de type. Je m'attends à ce que cela dérange certains lecteurs. Toutefois, je soupçonne que, à la fin de l'ouvrage, ils auront changé d'avis.

Comment rédiger des annotations de type

Comme nous l'avons vu, pour les types intégrés, tels que `int`, `bool` et `float`, il vous suffit d'utiliser le type lui-même en tant qu'annotation. Que se passe-t-il si vous avez (par exemple) une liste ?

```
def total(xs: list) -> float:
    return sum(total)
```

Ce n'est pas faux, mais le type n'est pas suffisamment spécifique. Il est clair que nous voulons réellement que `xs` soit une liste de décimaux (`float`) et non pas (par exemple) une liste de chaînes.

Le module `typing` (bibliothèque de types standards) fournit un certain nombre de types paramétrés que nous pouvons utiliser à cette fin précise :

```
from typing import List # notez le "L" majuscule
def total(xs: List[float]) -> float:
    return sum(total)
```

Jusqu'à maintenant, nous avons spécifié uniquement des annotations pour des paramètres de fonction et des types de retour. Pour les variables proprement dites, le type est généralement évident :

```
# voici comment annoter-typer des variables lorsque vous les définissez
# mais c'est superflu ; il est évident que x est ici un entier
x: int = 5
```

Toutefois, la chose est parfois moins évidente :

```
values = [] # quel est mon type ?
best_so_far = None # quel est mon type ?
```

Dans de tels cas, nous fournissons des indications via les types en ligne :

```
from typing import Optional, List
values: List[int] = []
best_so_far: Optional[float] = None # peut être soit un décimal, soit None
```

Le module `typing` contient de nombreux autres types ; nous n'en utiliserons que quelques-uns :

```
# les annotations de type dans cet extrait de code sont toutes superflues
from typing import Dict, Iterable, Tuple
# les clés sont des chaînes, les valeurs sont des entiers
counts: Dict[str, int] = {'data': 1, 'science': 2}
# les listes et les générateurs peuvent tous deux être itérés
if lazy:
    evens: Iterable[int] = (x for x in range(10) if x % 2 == 0)
else:
    evens = [0, 2, 4, 6, 8]
# les tuples spécifient un type pour chaque élément
triple: Tuple[int, float, int] = (10, 2.3, 5)
```

Enfin, sachant que Python dispose de fonctions de première classe, nous avons besoin d'un type pour les préciser elles aussi. En voici un exemple assez artificiel :

```
from typing import Callable
# L'indication par type explique que repeater est une fonction
# qui accepte deux arguments, une chaîne et un entier,
# et retourne une chaîne.
def twice(repeater: Callable[[str, int], str], s: str) -> str:
    return repeater(s, 2)

def comma_repeater(s: str, n: int) -> str:
    n_copies = [s for _ in range(n)]
    return ', '.join(n_copies)

assert twice(comma_repeater, "type hints") == "type hints, type hints"
```

Sachant que les annotations de type sont simplement des objets Python, nous pouvons les affecter à des variables pour faciliter leur référencement :

```
from typing import List
Number = int
Numbers = List[Number]
def total(xs: Numbers) -> Number:
    return sum(xs)
```

D'ici à la fin de cet ouvrage, la lecture et l'écriture d'annotations de type vous seront assez familières et j'espère que vous les utiliserez dans votre code.

Pour aller plus loin

Ainsi se termine le parcours du nouvel embauché. Essayez de ne rien détourner.

Voici quelques pistes pour aller plus loin dans votre découverte du langage :

- Il existe de nombreux tutoriels Python dans le monde. Celui du site officiel constitue plutôt un bon point de départ⁸.
- Vous pouvez opter pour le tutoriel officiel IPython⁹. N'hésitez pas à l'utiliser.
- La documentation `mypy` vous dira tout ce que vous voulez savoir sur les annotations et les contrôles de types de Python¹⁰.

8. <https://docs.python.org/3/tutorial/>

9. <https://ipython.readthedocs.io/en/stable/interactive/index.html>

10. <https://mypy.readthedocs.io/en/stable/>

3

Visualisation des données

Je suis convaincu que la visualisation est l'un des moyens les plus puissants pour atteindre ses objectifs personnels.

Harvey Mackay

Une partie importante de la boîte à outils du data scientist est dédiée à la visualisation des données. Bien qu'il soit très facile de créer des représentations visuelles, il est beaucoup plus difficile d'en créer qui soient efficaces.

La visualisation des données répond à deux attentes principales :

- explorer des données ;
- communiquer des données.

Dans ce chapitre, nous nous intéresserons avant tout au développement des compétences nécessaires pour commencer l'exploration de vos propres données et pour produire les représentations que nous utiliserons dans la suite du livre. Comme la plupart de nos thèmes de chapitre, la visualisation des données est un champ d'étude particulièrement riche qui mériterait d'y consacrer un livre entier. Cependant, nous allons essayer de vous apprendre à distinguer ce qui fait une bonne visualisation de ce qui ne fonctionne pas.

matplotlib

Il existe une grande variété d'outils de visualisation de données. Nous utiliserons la bibliothèque `matplotlib`¹, qui est largement répandue (même si elle commence à accuser son âge).

1. <https://matplotlib.org/>

Ce n'est pas forcément le bon choix si votre objectif consiste à produire des représentations interactives sophistiquées pour le Web. Toutefois, pour de simples graphiques à barres, des courbes et des nuages de points, elle est tout à fait suffisante.

`matplotlib` ne fait pas partie de la bibliothèque fondamentale de Python. Une fois votre environnement virtuel activé (chapitre 1), installez-la avec la commande suivante :

```
python -m pip install matplotlib
```

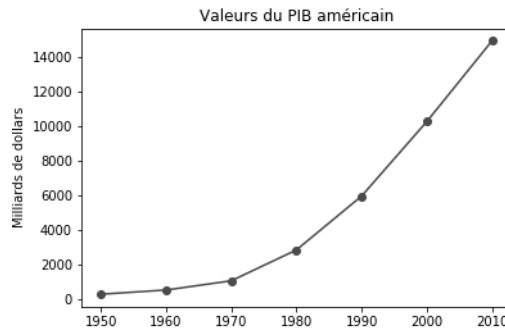
Nous allons nous servir du module `matplotlib.pyplot`. Dans son usage le plus basique, `pyplot` maintient un environnement interne au sein duquel vous pouvez construire une représentation pas à pas. Quand vous avez terminé, vous pouvez sauvegarder votre travail (avec `savefig`) ou l'afficher (avec `show`).

Par exemple, réaliser des diagrammes simples comme celui de la figure 3-1 est un jeu d'enfant :

```
from matplotlib import pyplot as plt
years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]
# trace une courbe ; les années sur l'axe des x, le PIB (GDP) sur l'axe des y
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')
# ajoute un titre
plt.title("valeurs de GDP")
# ajoute une étiquette sur l'axe des y
plt.ylabel("Milliards de dollars")
plt.show()
```

Figure 3-1

Une simple courbe.



Réaliser des diagrammes de qualité adaptés à la publication est plus difficile et sort du cadre de ce chapitre. Il existe de nombreuses solutions pour personnaliser vos graphiques en ajoutant des étiquettes sur les axes, des styles de ligne et des marqueurs de points. Sans essayer de traiter le sujet de manière exhaustive, nous allons nous concentrer sur quelques options seulement à travers plusieurs exemples.

Même si ce n'est pas le style que nous utiliserons le plus, `matplotlib` est capable de fournir des tracés complexes au sein de diagrammes, des mises en page très élaborées et des représentations interactives. Pour approfondir le sujet, reportez-vous à la documentation de cette bibliothèque.

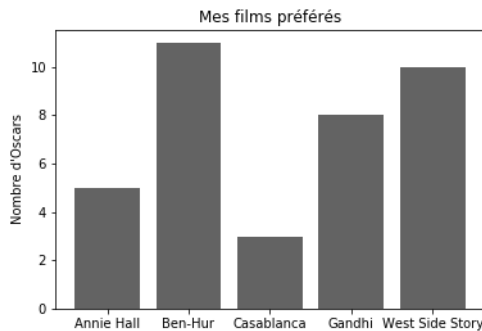
Les graphiques à barres

Un graphique à barres constitue un bon choix pour illustrer la variation d'une quantité au sein d'un ensemble d'éléments séparés. Par exemple, la figure 3-2 montre combien d'Oscars ont été décernés à une sélection de différents films :

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]
# dessine des barres avec :
# * en abscisse (x) [nb_oscars]
# * en ordonnées (y) [0, 1, 2, 3, 4] (index de films)
plt.bar(range(len(movies)), num_oscars)
plt.title("Mes films préférés") # ajoute un titre
plt.ylabel("Nombre d'Oscars") # étiquette de l'axe y
# étiquette de l'axe x avec les noms des films au centre des barres
plt.xticks(range(len(movies)), movies)
plt.show()
```

Figure 3-2

Un graphique simple à barres



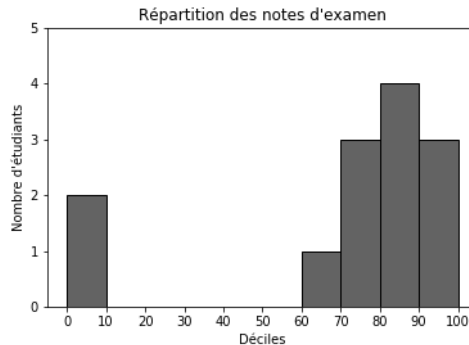
Le graphique à barres constitue aussi un bon choix pour représenter des histogrammes de valeurs numériques regroupées par catégories, afin d'explorer la répartition (distribution) des valeurs (figure 3-3).

```
from collections import Counter
grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]
# catégorise les notes par déciles, mais place 100 dans la catégorie des 90
histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)
plt.bar([x + 5 for x in histogram.keys()], # barres décalées de 5 à droite
        histogram.values(), # attribue à chaque barre sa hauteur correcte
        10, # largeur de 10
        edgecolor=(0, 0, 0)) # contours noirs pour chaque barre
```

```
plt.axis([-5, 105, 0, 5]) # axes x de -5 à 105 et y de 0 à 5
plt.xticks([10 * i for i in range(11)]) # étiquettes en x multiples de 10
plt.xlabel("Déciles")
plt.ylabel("Nombre d'étudiants")
plt.title("Répartition des notes d'examen")
plt.show()
```

Figure 3–3

Un graphique à barres utilisé
en tant qu'histogramme



Le troisième argument de `plt.bar` spécifie la largeur des barres. Ici, nous avons choisi une largeur de 10 pour remplir l'intégralité du décile. Nous avons en outre décalé les barres de 5 vers la droite, de sorte qu'elles soient centrées sur les valeurs médianes des déciles. Nous avons également ajouté un contour noir à chaque barre pour la distinguer visuellement.

L'appel à `plt.axis` indique que nous demandons une plage de valeurs de -5 à 105 pour les x (de sorte que les première et dernière barres disposent d'assez de place) et de 0 à 5 pour les y. L'appel à `plt.xticks` place les étiquettes de l'axe des x à 0, 10, 20... 100.

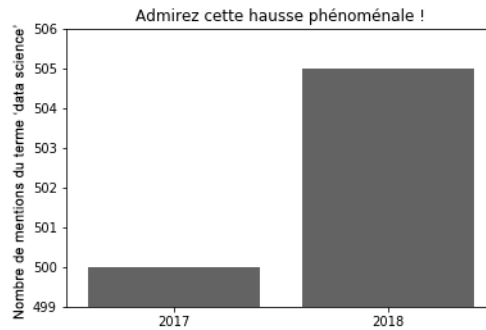
Utilisez judicieusement `plt.axis`. Quand vous créez un graphique à barres, il est déconseillé de faire démarrer l'axe des y autrement qu'à zéro, car c'est une manière facile de tromper les lecteurs (figure 3–4) :

```
mentions = [500, 505]
years = [2017, 2018]
plt.bar(years, mentions, 0.8)
plt.xticks(years)
plt.ylabel("Nombre de mentions du terme 'data science'")

# axe des y trompeur : affiche uniquement la partie au-dessus de 500
plt.axis([2016.5, 2018.5, 499, 506])
plt.title("Admirez cette hausse phénoménale !")
plt.show()
```

Figure 3–4

Un graphique à l'axe y trompeur

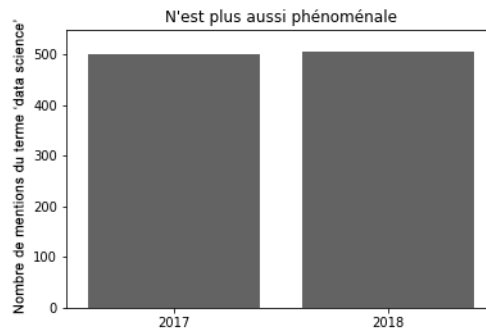


Sur la figure 3–5, nous utilisons des axes plus sérieux et l'impression produite n'est plus aussi forte :

```
plt.axis([2016.5, 2018.5, 0, 550])  
plt.title("N'est plus aussi phénoménale")  
plt.show()
```

Figure 3–5

Le même diagramme avec un axe des y plus honnête



Les courbes

Comme nous l'avons déjà vu, les courbes se tracent à l'aide de `plt.plot`. C'est un bon choix pour illustrer des tendances (figure 3–6) :

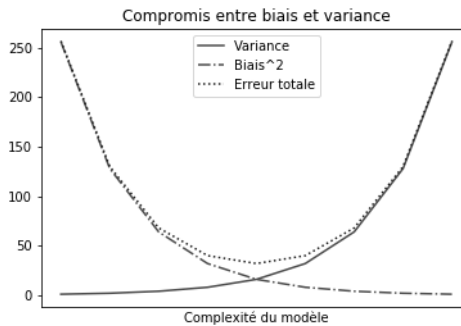
```
variance = [1, 2, 4, 8, 16, 32, 64, 128, 256]  
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]  
total_error = [x + y for x, y in zip(variance, bias_squared)]  
xs = [i for i, _ in enumerate(variance)]
```

```
# Nous pouvons multiplier les appels à plt.plot
# pour présenter plusieurs séries sur le même graphique.
# ligne verte continue :
plt.plot(xs, variance, 'g-', label='variance')
# ligne rouge alternant tirets et pointillés :
plt.plot(xs, bias_squared, 'r-.', label='bias^2')
# ligne bleue en pointillés :
plt.plot(xs, total_error, 'b:', label='total error')
# Parce que nous avons affecté des étiquettes à chaque série,
# nous avons une légende automatique :
# loc=9 signifie "top center" (centre haut).
plt.legend(loc=9)
plt.xlabel("Complexité du modèle")
plt.xticks([])

plt.title("Compromis entre biais et variance")
plt.show()
```

Figure 3–6

Plusieurs courbes
avec une légende



Les nuages de points

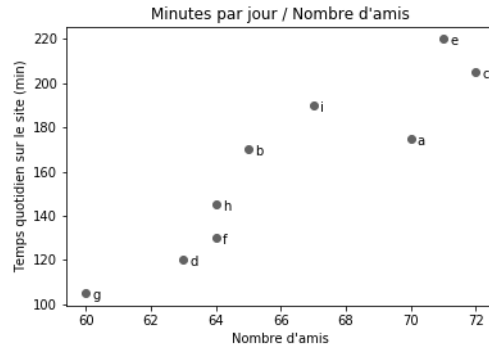
Le nuage de points est la solution privilégiée pour représenter la relation entre deux ensembles de données associés. Par exemple, la figure 3–7 illustre la relation entre le nombre d'amis des membres de DataSciencester et le nombre de minutes que ces derniers passent sur le site chaque jour :

```
friends = [70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
plt.scatter(friends, minutes)
# étiquette chaque point
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
                 # positionne l'étiquette près du point associé
                 xy=(friend_count, minute_count),
```

```
# avec un léger décalage
xytext=(5, -5),
textcoords='offset points')
plt.title("Minutes par jour / Nombre d'amis")
plt.xlabel("Nombre d'amis")
plt.ylabel("Temps quotidien sur le site (min)")
plt.show()
```

Figure 3-7

Un nuage de points pour représenter le nombre d'amis comparé au temps passé sur le site

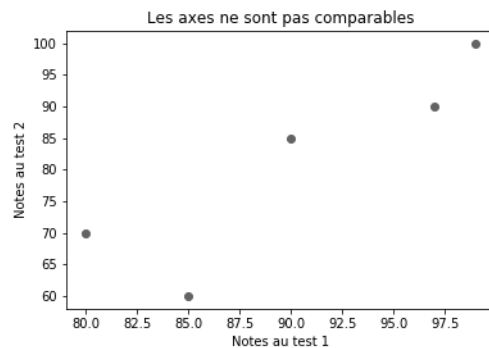


Lorsque vous représentez des variables comparables, vous risquez d'obtenir une image trompeuse si vous laissez `matplotlib` choisir l'échelle (figure 3-8).

```
test_1_grades = [99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]
plt.scatter(test_1_grades, test_2_grades)
plt.title("Les axes ne sont pas comparables")
plt.xlabel("Notes au test 1")
plt.ylabel("Notes au test 2")
plt.show()
```

Figure 3-8

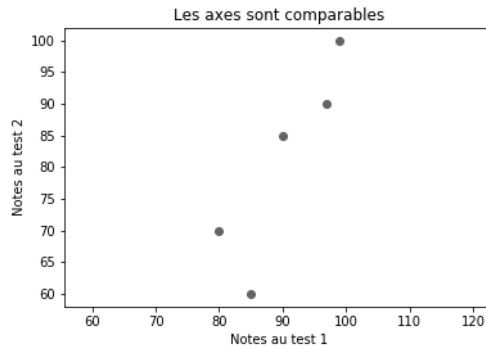
Un nuage de points avec des axes non comparables



Si nous insérons un appel à `plt.axis("equal")`, le diagramme montre bien mieux que la majorité de la variation provient du test 2 (figure 3–9).

Figure 3–9

Le même nuage de points avec, cette fois, des axes comparables



Vous en savez assez pour commencer à construire des représentations visuelles. Nous en apprendrons davantage tout au long de cet ouvrage.

Pour aller plus loin

- La galerie `matplotlib`² vous donnera une idée précise des différentes possibilités qu’offre cette bibliothèque graphique (et de leur exécution).
- `seaborn`³ se greffe sur `matplotlib` pour réaliser des représentations plus élaborées (et plus complexes).
- `Altair`⁴ est une bibliothèque Python plus récente dédiée à la création de visualisations déclaratives.
- `d3.js`⁵ est une bibliothèque JavaScript dédiée aux représentations interactives sophistiquées pour le Web. Bien qu’elle ne fasse pas partie de Python, elle reste très utilisée. Elle mérite vraiment que vous preniez le temps de vous familiariser avec elle.
- `Bokeh`⁶, quant à elle, est une bibliothèque qui apporte des effets 3D aux visualisations en Python.

2. <https://matplotlib.org/gallery.html>

3. <https://seaborn.pydata.org/>

4. <https://altair-viz.github.io/>

5. <https://d3js.org/>

6. <https://docs.bokeh.org/en/latest/>

4

Algèbre linéaire

Existe-t-il quelque chose de plus inutile ou de moins utile que l'algèbre ?

Billy Connolly

L'algèbre linéaire est la branche des mathématiques consacrée aux espaces vectoriels. Bien que je n'aie pas la prétention de vous enseigner l'algèbre linéaire en un court chapitre, il y a tellement de concepts et de techniques de la data science qui y font référence qu'il est de mon devoir d'essayer. Ce que nous apprendrons dans ce chapitre nous servira énormément tout au long du livre.

Les vecteurs

Sur un plan abstrait, les vecteurs sont des objets qui peuvent être ajoutés entre eux et/ou multipliés par des scalaires (autrement dit des nombres), pour former d'autres vecteurs.

Concrètement (pour nous), les vecteurs sont des points dans un espace de dimension finie. Même si vous n'avez pas l'habitude d'envisager vos données comme des vecteurs, ces derniers constituent pourtant une manière utile de représenter des données numériques.

Par exemple, si vous gérez la taille, le poids et l'âge d'un grand nombre de personnes, vous pouvez considérer vos données comme des vecteurs à trois dimensions [taille, poids, âge]. Si vous êtes enseignant et organisez quatre examens, vous pouvez considérer les notes des élèves comme des vecteurs à quatre dimensions [examen1, examen2, examen3, examen4].

La méthode la plus simple consiste à représenter les vecteurs par une liste de nombres. Par exemple, une liste de trois nombres correspond à un vecteur dans un espace à trois dimensions et vice versa.

Pour ce faire, nous utiliserons un *alias* de type qui stipule que `vector` est simplement une liste de nombres décimaux :

```
from typing import List
vector = List[float]

height_weight_age = [70, # en pouces, soit 167 centimètres
                     170, # en livres, soit 61 kilogrammes
                     40] # années

grades = [95, # examen 1
          80, # examen 2
          75, # examen 3
          62] # examen 4
```

En outre, nous voulons réaliser des opérations arithmétiques sur les vecteurs. Comme les listes Python ne sont pas des vecteurs (et n'offrent donc aucune possibilité d'arithmétique vectorielle), nous devons construire nos propres outils. Alors, allons-y !

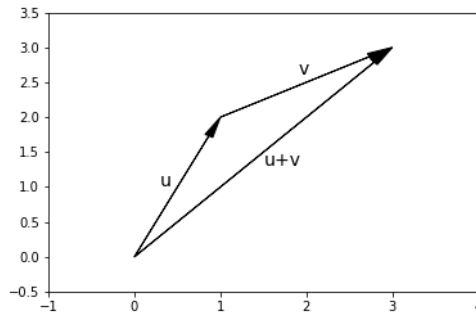
Pour commencer, nous aurons souvent besoin d'additionner deux vecteurs — *de même longueur* —, ce qui consiste à additionner leurs éléments.

Soit deux vecteurs de même longueur v et w ; leur somme est le vecteur dont le premier élément est $v[0] + w[0]$, le deuxième élément est $v[1] + w[1]$ et ainsi de suite.

Par exemple, l'addition des vecteurs $[1, 2]$ et $[2, 1]$ donne comme résultat $[1 + 2, 2 + 1]$ soit $[3, 3]$ (figure 4–1).

Figure 4–1

Additionner deux vecteurs



Il est facile de mettre en œuvre cette opération en zippant (`zip`) les vecteurs ensemble et en utilisant une liste en compréhension pour ajouter les éléments correspondants :

```
def add(v: Vector, w: Vector) -> Vector:
    """ajoute les éléments correspondants"""
    assert len(v) == len(w), "les vecteurs doivent être de même longueur"
    return [v_i + w_i for v_i, w_i in zip(v, w)]

assert add([1, 2, 3], [4, 5, 6]) == [5, 7, 9]
```

De même, il suffit de soustraire les éléments correspondants pour calculer la différence entre deux vecteurs :

```
def subtract(v: Vector, w: Vector) -> Vector:
    """soustrait les éléments correspondants"""
    assert len(v) == len(w), "les vecteurs doivent être de même longueur"
    return [v_i - w_i for v_i, w_i in zip(v, w)]

assert subtract([5, 7, 9], [4, 5, 6]) == [1, 2, 3]
```

Parfois, nous voulons faire la somme élément par élément d'une liste de vecteurs. Il s'agit de créer un vecteur dont le premier élément sera la somme de tous les premiers éléments, le deuxième élément la somme de tous les deuxièmes éléments et ainsi de suite.

```
def vector_sum(vectors: List[Vector]) -> Vector:
    """fait la somme de tous les éléments correspondants"""
    # vérifier que les vecteurs ne sont pas vides
    assert vectors, "Aucun vecteur fourni !"
    # vérifier que les vecteurs sont tous de la même taille
    num_elements = len(vectors[0])
    assert all(len(v) == num_elements for v in vectors), \
        "Tailles différentes !"
    # l'élément de rang i du résultat est la somme de chaque vecteur[i]
    return [sum(vector[i] for vector in vectors)
            for i in range(num_elements)]

assert vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]) == [16, 20]
```

Nous pouvons également être amenés à multiplier un vecteur par un scalaire, ce qui se fait simplement en multipliant chacun de ses éléments par ce nombre :

```
def scalar_multiply(c: float, v: Vector) -> Vector:
    """multiplie chaque élément par c"""
    return [c * v_i for v_i in v]

assert scalar_multiply(2, [1, 2, 3]) == [2, 4, 6]
```

Ceci nous permet de calculer élément par élément la moyenne d'une liste de vecteurs (de même taille) :

```
def vector_mean(vectors: List[Vector]) -> Vector:
    """calculé la moyenne élément par élément"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

assert vector_mean([[1, 2], [3, 4], [5, 6]]) == [3, 4]
```

Le produit scalaire (*dot product*) est un outil moins évident. Le produit scalaire de deux vecteurs est la somme de leurs produits, élément par élément :

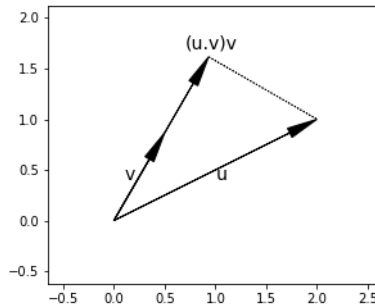
```
def dot(v: Vector, w: Vector) -> float:
    """calculé  $v_1 * w_1 + \dots + v_n * w_n$ """
    assert len(v) == len(w), "les vecteurs doivent être de même longueur"
    return sum(v_i * w_i for v_i, w_i in zip(v, w))

assert dot([1, 2, 3], [4, 5, 6]) == 32 # 1 * 4 + 2 * 5 + 3 * 6
```

Si w est un vecteur unité, le produit scalaire mesure de combien le vecteur v s'étend dans la direction w . Par exemple, si $w = [1, 0]$, alors $v \cdot w$ est le premier élément de v , tout simplement. Autrement dit, il s'agit de la longueur du vecteur obtenu si vous projetez v sur w (figure 4-2).

Figure 4-2

Le produit scalaire comme projection vectorielle



Il est alors facile de calculer la somme des carrés d'un vecteur :

```
def sum_of_squares(v: Vector) -> float:
    """retourne  $v_1 * v_1 + \dots + v_n * v_n$ """
    return dot(v, v)

assert sum_of_squares([1, 2, 3]) == 14 # 1 * 1 + 2 * 2 + 3 * 3
```

Elle sert pour calculer sa magnitude :

```
import math
def magnitude(v: Vector) -> float:
    """retourne la magnitude de v"""
    return math.sqrt(sum_of_squares(v)) # math.sqrt = racine carrée

assert magnitude([3, 4]) == 5
```

Nous avons maintenant tous les éléments nécessaires pour calculer la distance entre deux vecteurs, définie comme suit.

$$\sqrt{(v_1 - w_1)^2 + \dots + (v_n - w_n)^2}$$

```
def squared_distance(v: Vector, w: Vector) -> float:
    """calcule (v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(subtract(v, w))

def distance(v: Vector, w: Vector) -> float:
    """calcule la distance entre v et w"""
    return math.sqrt(squared_distance(v, w))
```

Ceci est plus clair si nous écrivons ce qui suit (équivalent) :

```
def distance(v: Vector, w: Vector) -> float:
    return magnitude(subtract(v, w))
```

Nous en savons à présent assez pour démarrer. Nous utiliserons largement ces fonctions tout au long du livre.

Remarque

Utiliser des listes comme vecteurs est parfait pour l'explication, mais pas pour les performances. Dans le code de production, vous utiliserez la bibliothèque `NumPy`. Elle contient une classe `array` très performante comportant toutes sortes d'opérations arithmétiques.

Les matrices

Une matrice est une collection de nombres à deux dimensions. Nous représenterons les matrices comme des listes de listes, toutes les listes internes ayant la même taille et représentant les lignes de la matrice. Si A est une matrice, alors $A[i][j]$ est l'élément de la i -ème ligne et de la j -ème colonne. Par convention mathématique, nous utiliserons souvent des majuscules pour représenter les matrices.

```
# un autre alias de type
Matrix = List[List[float]]
# A présente 2 lignes et 3 colonnes
A = [[1, 2, 3],
      [4, 5, 6]]
# B présente 3 lignes et 2 colonnes
B = [[1, 2],
      [3, 4],
      [5, 6]]
```

Remarque

En mathématiques, vous nommez généralement la première ligne de la matrice « ligne 1 » et sa première colonne « colonne 1 ».

En Python, les listes sont indexées à partir de zéro. Nous appellerons donc la première ligne d'une matrice « ligne 0 » et sa première colonne « colonne 0 ».

Étant donnée la convention de représentation en listes de listes, la matrice A possède `len(A)` lignes et `len(A[0])` colonnes, ce que nous appellerons sa « forme » :

```
from typing import Tuple
def shape(A: Matrix) -> Tuple[int, int]:
    """retourne (nombre de lignes de A, nombre de colonnes de A)"""
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0 # nombre d'éléments dans la première ligne
    return num_rows, num_cols

assert shape([[1, 2, 3], [4, 5, 6]]) == (2, 3) # 2 lignes, 3 colonnes
```

Si une matrice est composée de n lignes et k colonnes, nous la qualifierons de matrice $n \times k$. Nous pouvons (et parfois devons) considérer chaque ligne d'une matrice $n \times k$ comme un vecteur de longueur k et chaque colonne comme un vecteur de longueur n :

```
def get_row(A: Matrix, i: int) -> Vector:
    """retourne la ligne i de la matrice A (en tant que vecteur)"""
    return A[i]

def get_column(A: Matrix, j: int) -> Vector:
    """retourne la ligne j de la matrice A (en tant que vecteur)"""
    return [A_i[j] # élément j de la ligne A_i
            for A_i in A] # pour chaque ligne A_i
```

Nous voulons aussi être capables de créer une matrice selon sa forme et une fonction pour générer ses éléments. Nous pouvons le faire à partir d'une liste en compréhension imbriquée :

```
from typing import Callable
def make_matrix(num_rows: int,
               num_cols: int,
```

```

entry_fn: Callable[[int, int], float]) -> Matrix:
    """retourne une matrice num_rows x num_cols
    dont l'entrée de rang (i,j) est entry_fn(i, j)"""
    return [[entry_fn(i, j) # i étant donné, créer une liste
              for j in range(num_cols)] # [entry_fn(i, 0), ...]
            for i in range(num_rows)] # créer une liste pour chaque i

```

Avec cette fonction, vous pouvez créer une matrice identité 5×5 (avec des 1 sur la diagonale et des 0 ailleurs) :

```

def identity_matrix(n: int) -> Matrix:
    """retourne la matrice identité  $n \times n$ """
    return make_matrix(n, n, lambda i, j: 1 if i == j else 0)

assert identity_matrix(5) == [[1, 0, 0, 0, 0],
                              [0, 1, 0, 0, 0],
                              [0, 0, 1, 0, 0],
                              [0, 0, 0, 1, 0],
                              [0, 0, 0, 0, 1]]

```

Pour nous, les matrices sont importantes à plus d'un titre.

- Tout d'abord, nous pouvons utiliser une matrice pour représenter très simplement un jeu de données constitué de plusieurs vecteurs. Par exemple, si vous connaissez la taille, le poids et l'âge de 1 000 personnes, vous pouvez les ranger dans une matrice $1\,000 \times 3$:

```

data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
        ]

```

- Ensuite, comme nous le verrons plus loin, une matrice $n \times k$ nous permet de représenter une fonction linéaire qui fait correspondre des vecteurs de dimension k à des vecteurs de dimension n . Plusieurs de nos techniques et concepts feront appel à de telles fonctions.
- Enfin, les matrices servent à représenter des relations binaires. Au chapitre 1, nous avons représenté un réseau sous la forme d'une collection de paires (i, j) . Une autre représentation consiste à créer une matrice A telle que $A[i][j]$ vaille 1 si les nœuds i et j sont connectés, et 0 sinon.

Souvenez-vous que nous avons :

```

friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]

```

Nous pouvons aussi le représenter comme suit.

```
# utilisateur 0 1 2 3 4 5 6 7 8 9

friend_matrix = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # utilisateur 0
                 [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # utilisateur 1
                 [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # utilisateur 2
                 [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # utilisateur 3
                 [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # utilisateur 4
                 [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # utilisateur 5
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # utilisateur 6
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # utilisateur 7
                 [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # utilisateur 8
                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # utilisateur 9
```

S'il y a très peu de connexions, cette représentation s'avérera beaucoup moins efficace, car vous devrez stocker beaucoup de zéros. Cependant, avec la représentation matricielle, il est beaucoup plus rapide de vérifier si deux nœuds sont connectés. Il suffit pour cela d'établir un accès à la matrice, plutôt que d'inspecter (potentiellement) toutes les limites une à une :

```
assert friend_matrix[0][2] == 1, "0 et 2 sont amis"
assert friend_matrix[0][8] == 0, "0 et 8 ne sont pas amis"
```

De même, pour trouver les connexions d'un nœud donné, il suffit d'inspecter la colonne (ou la ligne) qui lui correspond :

```
# examiner une seule ligne suffit
friends_of_five = [i
                   for i, is_friend in enumerate(friend_matrix[5])
                   if is_friend]
```

Auparavant, avec un petit graphe, nous avions maintenu une liste de connexions pour chaque nœud afin d'accélérer le processus. Toutefois, pour un graphe plus important en évolution, l'approche serait sans doute trop coûteuse et trop difficile à maintenir.

Nous reviendrons sur les matrices tout au long de ce livre.

Pour aller plus loin

L'algèbre linéaire est très répandue chez les data scientists (souvent implicitement et parfois même chez des personnes qui n'y comprennent pas grand-chose). Lire un manuel de formation est alors recommandé. Il en existe plusieurs gratuitement disponibles en ligne (en anglais) :

- *Linear Algebra*¹, Jim Hefferon (Saint Michael's College) ;
- *Linear Algebra*², David Cherney, Tom Denton, Rohit Thomas et Andrew Waldron (UC Davis) ;
- *Linear Algebra Done Wrong*³, Sergei Treil (Brown University).

Vous trouverez tout ce que nous avons construit ici gratuitement dans `NumPy`⁴ (et bien d'autres choses encore, y compris un fonctionnement plus performant).

1. <http://joshua.smcvt.edu/linearalgebra/>

2. <https://www.math.ucdavis.edu/~linear/linear-guest.pdf>

3. https://www.math.brown.edu/~treil/papers/LADW/LADW_2017-09-04.pdf

4. <https://numpy.org/>

5

Statistique

Les faits sont têtus, mais les statistiques sont plus dociles.

Mark Twain

La statistique regroupe les mathématiques et les techniques qui permettent de comprendre les données. C'est un domaine très vaste et très riche, au point qu'il pourrait occuper tout un rayon, voire toute une pièce, d'une bibliothèque et non un simple chapitre dans un ouvrage. Notre discussion ne pourra donc pas explorer le sujet en profondeur. Ici, je vais vous en apprendre juste assez pour vous rendre redoutable, en espérant piquer suffisamment au vif votre curiosité pour vous inciter à approfondir le sujet par vous-même.

Décrire un unique jeu de données

Grâce à une combinaison de bouche à oreille et d'heureux hasards, DataSciencester compte désormais des dizaines de membres. Le responsable Appel aux dons vous demande de lui décrire combien d'amis de vos membres il peut inclure dans ses sollicitations. Grâce aux techniques du chapitre 1, vous pouvez facilement lui procurer ces données. Cependant, votre problème, c'est de savoir comment les décrire.

La description la plus évidente d'une donnée, c'est la donnée elle-même :

```
num_friends = [100, 49, 41, 40, 25, 21, 21, 19, 19, 18, 18, 16, 15, 15, 15, 15, 14,
14, 13, 13, 13, 13, 12, 12, 11, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
10, 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

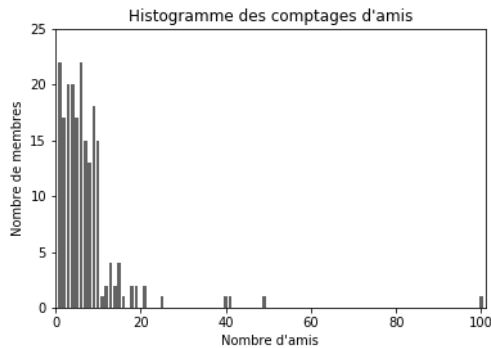
Pour un petit nombre de données, c'est sans doute la meilleure description. En revanche, pour un jeu de données plus vaste, c'est peu pratique et illisible. Imaginez-vous explorer une liste d'un million de nombres ! C'est cette raison qui nous pousse à faire appel aux statistiques pour communiquer des informations pertinentes sur nos données.

En première approche, vous pouvez construire un histogramme du nombre d'amis à l'aide de `Counter` et `plt.bar` (figure 5-1) :

```
from collections import Counter
import matplotlib.pyplot as plt
friend_counts = Counter(num_friends)
xs = range(101) # la plus grande valeur est 100
ys = [friend_counts[x] for x in xs] # la hauteur est le nombre d'amis
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogramme des comptages d'amis")
plt.xlabel("Nombre d'amis")
plt.ylabel("Nombre de membres")
plt.show()
```

Figure 5-1

Un histogramme des comptages d'amis



Malheureusement, ce diagramme est difficile à glisser dans les conversations. Vous allez donc commencer à générer des statistiques. Le plus simple des résultats statistiques est sans doute le *nombre de points de mesure* :

```
num_points = len(num_friends) # 204
```

Vous vous intéressez sans doute aux valeurs extrêmes :

```
largest_value = max(num_friends) # 100
smallest_value = min(num_friends) # 1
```

Elles ne sont que des cas particuliers de valeurs figurant à des positions spécifiques :

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0] # 1
second_smallest_value = sorted_values[1] # 1
second_largest_value = sorted_values[-2] # 49
```

Ce n'est que le début.

Tendances centrales

En général, nous savons plus ou moins où se situe le « centre » de nos données. Le plus souvent, nous utilisons la *moyenne*, qui est simplement la somme des données divisée par leur nombre :

```
def mean(xs: List[float]) -> float:
    return sum(xs) / len(xs)

print (mean(num_friends)) # 7.333333
```

Si vous disposez de deux points représentant les données, la moyenne est le point à mi-chemin entre les deux. À mesure que vous ajoutez des points, la moyenne se déplace, mais elle dépend toujours de la valeur de chaque donnée. Par exemple, si vous disposez de 10 points de données et augmentez la valeur de n'importe lequel d'entre eux de 1, vous augmentez la moyenne de 0,1.

Parfois, nous sommes intéressés par la *médiane*, qui est la valeur la plus centrale (si le nombre de points de données est impair) ou la moyenne des deux valeurs les plus centrales (si le nombre de points de données est pair).

Par exemple, si nous avons cinq points dans un vecteur *v* trié, la médiane est $v[5 // 2] = v[2]$. Et si nous avons six points, nous voulons la moyenne de $v[2]$ (le troisième point) et $v[3]$ (le quatrième point).

Remarquez que la médiane, à la différence de la moyenne, ne dépend pas intégralement de chaque valeur de vos données. Par exemple, si vous agrandissez le point le plus important (ou

si vous réduisez le point le plus petit), les points intermédiaires demeurent inchangés et donc la médiane aussi.

Nous allons rédiger différentes fonctions pour les cas pairs et impairs et les combiner :

```
# les tirets de soulignement indiquent qu'il s'agit de fonctions privées
# En effet, elles sont destinées à être appelées par notre fonction de
# médiane, mais non par d'autres utilisateurs de notre bibliothèque
# de statistiques.
def _median_odd(xs: List[float]) -> float:
    """len(xs) impaire, la médiane est l'élément du milieu"""
    return sorted(xs)[len(xs) // 2]

def _median_even(xs: List[float]) -> float:
    """len(xs) paire, la médiane est la moyenne des éléments du milieu"""
    sorted_xs = sorted(xs)
    hi_midpoint = len(xs) // 2
    return (sorted_xs[hi_midpoint - 1] + sorted_xs[hi_midpoint]) / 2

def median(v: List[float]) -> float:
    """trouve la valeur médiane de v"""
    return _median_even(v) if len(v) % 2 == 0 else _median_odd(v)

assert median([1, 10, 2, 9, 5]) == 5
assert median([1, 9, 2, 10]) == (2 + 9) / 2
```

Nous pouvons à présent calculer le nombre médian d'amis :

```
print(median(num_firends)) # 6
```

Manifestement, la moyenne est plus facile à calculer et elle varie de manière progressive en même temps que nos données. Si nous avons n données élémentaires et si une d'entre elles augmente d'une petite quantité e , alors nécessairement la moyenne va augmenter de e / n . Ceci rend la moyenne très souple pour toutes sortes d'astuces de calcul. En revanche, pour trouver la médiane, il faut trier nos données. Modifier une de nos données élémentaires d'une petite quantité e peut donc soit augmenter la médiane de e , soit d'une quantité inférieure à e , soit pas du tout (tout dépend des autres données).

Remarque

En fait, il existe des astuces pour calculer efficacement la médiane^a sans trier les données. Toutefois, elles sont hors du périmètre de cet ouvrage. Nous devons donc, pour notre part, trier nos données.

a. <https://fr.wikipedia.org/wiki/Quickselect>

D'un autre côté, la moyenne est très sensible aux valeurs extrêmes de nos données. Si notre utilisateur le plus populaire avait 200 amis (au lieu de 100), la moyenne augmenterait pour atteindre 7,82, mais la médiane ne bougerait pas. S'il s'avère que les valeurs extrêmes sont

vraisemblablement des données erronées (ou non représentatives du phénomène que nous essayons de comprendre), alors la moyenne peut parfois nous donner une image déformée. Par exemple, pour reprendre ce cas souvent cité, au milieu des années 1980, à l'université de Caroline du Nord, la spécialisation qui arrivait en tête pour le salaire moyen de première embauche était la géographie, à cause principalement de la star de la NBA Michael Jordan (un étudiant particulièrement peu représentatif).

Le *quantile* est une généralisation de la médiane. Il représente la valeur à partir de laquelle on se trouve en dessous d'un certain pourcentage des données. La médiane est la valeur pour laquelle 50 % des données se situent en dessous.

```
def quantile(xs: List[float], p: float) -> float:
    p_index = int(p * len(xs))
    return sorted(xs)[p_index]
    """retourne la valeur de percentile de rang p dans x"""

assert quantile(num_friends, 0.10) == 1
assert quantile(num_friends, 0.25) == 3
assert quantile(num_friends, 0.75) == 9
assert quantile(num_friends, 0.90) == 13
```

Il est moins fréquent de s'intéresser aux *modes* (ou dominantes), c'est-à-dire aux valeurs les plus fréquentes :

```
def mode(x: List[float]) -> List[float]:
    """retourne une liste ; peut concerner plus d'un mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.items()
            if count == max_count]

assert set(mode(num_friends)) == {1, 6}
```

La dispersion

La *dispersion* fait référence aux mesures qui décrivent la répartition de nos données. Il existe des statistiques pour lesquelles les valeurs proches de zéro signifient qu'il n'y a aucune dispersion et les valeurs élevées (quel que soit leur sens) signifient une grande dispersion. Par exemple, une mesure très simple est l'*étendue* (*range*), qui est tout simplement la différence entre les éléments les plus grands et les plus petits.

```
# "range" a déjà un sens en Python ; nous utiliserons donc un autre nom (data_range)
def data_range(xs: List[float]) -> float:
    return max(xs) - min(xs)

assert data_range(num_friendsnum_friends) == 99
```

L'étendue est de zéro quand le maximum et le minimum sont égaux, ce qui se produit seulement si les éléments sont tous les mêmes. Inversement, si l'étendue est grande, le maximum est largement supérieur au minimum et les données sont davantage éparpillées.

Comme la médiane, l'étendue ne repose pas réellement sur l'ensemble du jeu de données. Ainsi, un jeu de données dont les points valent tous 0 ou 100 a la même étendue qu'un autre dont les valeurs sont 0, 100 et beaucoup de 50. Pourtant, il semble que le premier jeu de données « devrait » être plus éparpillé.

Une mesure plus complexe de la dispersion est la *variance*. Elle se calcule comme suit.

```
from scratch.linear_algebra import sum_of_squares
# Supposant que les outils d'algèbre linéaire présentés dans le précédent
# chapitre ont été sauvegardés dans le module outils_al.
def de_mean(xs: List[float]) -> List[float]:
    """fait glisser xs en retranchant sa moyenne"""
    x_bar = mean(xs)
    return [x - x_bar for x in xs]

def variance(xs: List[float]) -> float:
    """presque la moyenne des carrés des écarts à la moyenne"""
    assert len(xs) >= 2, "la variance nécessite au moins deux éléments"
    n = len(xs)
    deviations = de_mean(xs)
    return sum_of_squares(deviations) / (n - 1)

assert 81.54 < variance(num_friends) < 81.55
```

Remarque

Il semble que ce soit presque la moyenne des carrés des écarts à la moyenne de l'échantillon, sauf que l'on divise par $n - 1$ au lieu de n . En réalité, quand on dispose d'un échantillon plus vaste, x_{bar} est seulement une estimation de la moyenne réelle, ce qui signifie qu'en moyenne $(x_i - x_{\text{bar}})^2$ est une sous-estimation du carré de la déviation de x_i à la moyenne, ce qui explique aussi que l'on divise par $n - 1$ au lieu de n .

a. https://en.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation

Quel que soit l'élément mesuré de nos données (par exemple, le nombre d'amis), toutes nos mesures de tendance centrale sont exprimées dans la même unité. L'étendue, elle aussi, est exprimée dans cette unité. Au contraire, la variance est exprimée dans l'unité originale au carré, ici, le nombre d'amis au carré. Comme cela ne signifie pas grand-chose dans ce cas, nous allons souvent nous intéresser plutôt à l'*écart-type*, qui est la racine carrée de la variance.

```
import math
def standard_deviation(xs: List[float]) -> float:
    """l'écart-type est la racine carrée de la variance"""
    return math.sqrt(variance(xs))

assert 9.02 < standard_deviation(num_friends) < 9.04
```

L'étendue et l'écart-type ont le même problème avec les valeurs extrêmes que celui déjà évoqué pour la moyenne. Dans notre exemple, si notre utilisateur le plus apprécié avait 200 amis, l'écart-type serait de 14,89, soit plus de 60 % de plus !

Une solution plus fiable consiste à calculer la différence entre les valeurs des 75^e et 25^e centiles :

```
def interquartile_range(xs: List[float]) -> float:
    """retourne la différence entre les 75e et 25e centiles"""
    return quantile(xs, 0.75) - quantile(xs, 0.25)

assert interquartile_range(num_friendsnum_friends) == 6
```

Le résultat est alors à peine affecté par un petit nombre de valeurs extrêmes.

La corrélation

La responsable Croissance chez DataSciencester a une théorie selon laquelle le temps que les gens passent sur le site dépend du nombre d'amis qu'ils y trouvent. Elle vous demande de la vérifier.

Après avoir épluché les journaux de trafic du site, vous avez obtenu une liste qui montre combien de minutes chaque utilisateur passe quotidiennement sur le site de DataSciencester. Vous l'avez triée pour que ses éléments correspondent à ceux de notre liste précédente, `num_friends`. Nous voudrions à présent explorer la relation entre ces deux résultats.

```
# durées quotidiennes en minutes
daily_minutes = [ 1.00, 68.77, 51.25, 52.08, 38.36, 44.54, 57.13, 51.40, 41.42,
31.22, 34.76, 54.01, 38.79, 47.59, 49.10, 27.66, 41.03, 36.73, 48.65, 28.12, 46.62,
35.57, 32.98, 35.00, 26.07, 23.77, 39.73, 40.57, 31.65, 31.21, 36.32, 20.45, 21.93,
26.02, 27.34, 23.49, 46.94, 30.50, 33.80, 24.23, 21.40, 27.94, 32.24, 40.57, 25.07,
19.42, 22.39, 18.42, 46.96, 23.72, 26.41, 26.97, 36.76, 40.32, 35.02, 29.47, 30.20,
31.00, 38.11, 38.18, 36.31, 21.03, 30.86, 36.07, 28.66, 29.08, 37.28, 15.28, 24.17,
22.31, 30.17, 25.53, 19.85, 35.37, 44.60, 17.23, 13.47, 26.33, 35.02, 32.09, 24.81,
19.33, 28.77, 24.26, 31.98, 25.73, 24.86, 16.28, 34.51, 15.23, 39.72, 40.80, 26.06,
35.76, 34.76, 16.13, 44.04, 18.03, 19.65, 32.62, 35.59, 39.43, 14.18, 35.24, 40.13,
41.82, 35.45, 36.07, 43.67, 24.61, 20.90, 21.90, 18.79, 27.61, 27.21, 26.61, 29.77,
20.59, 27.53, 13.82, 33.20, 25.00, 33.10, 36.65, 18.63, 14.87, 22.20, 36.81, 25.53,
24.62, 26.25, 18.21, 28.08, 19.42, 29.79, 32.80, 35.99, 28.32, 27.79, 35.88, 29.06,
36.28, 14.10, 36.63, 37.49, 26.90, 18.58, 38.48, 24.48, 18.95, 33.55, 14.24, 29.04,
32.51, 25.63, 22.22, 19.00, 32.73, 15.16, 13.90, 27.20, 32.01, 29.27, 33.00, 13.74,
20.42, 27.32, 18.23, 35.35, 28.48, 9.08, 24.62, 20.12, 35.26, 19.92, 31.02, 16.49,
12.16, 30.70, 31.22, 34.65, 13.13, 27.51, 33.20, 31.57, 14.10, 33.42, 17.44, 10.12,
24.42, 9.82, 23.39, 30.93, 15.03, 21.67, 31.09, 33.29, 22.61, 26.89, 23.48, 8.38,
27.81, 32.35, 23.84]

# durées quotidiennes en heures
daily_hours = [dm / 60 for dm in daily_hours]
```

Regardons tout d'abord la *covariance*. Alors que la variance mesure comment une variable isolée s'écarte de la moyenne, la covariance mesure comment deux variables s'écartent en tandem de leurs moyennes :

```
from scratch.linear_algebra import dot
def covariance(xs: List[float], ys: List[float]) -> float:
    assert len(xs) == len(ys), "xs et ys doivent être de même taille"
    return dot(de_mean(xs), de_mean(ys)) / (len(xs) - 1)

assert 22.42 < covariance(num_friendsnum_friends, daily_minutes) < 22.43
assert 22.42 / 60 < covariance(num_friendsnum_friends, daily_hours) < 22.43 / 60
```

N'oubliez pas que le produit scalaire (*dot*) ajoute les produits des paires d'éléments correspondants. Quand les éléments correspondants de *x* et *y* sont tous les deux au-dessus ou au-dessous de leurs moyennes respectives, un nombre positif est ajouté au total. Quand l'un est au-dessus et l'autre au-dessous, un nombre négatif est ajouté au total. Une covariance positive « élevée » signifie que *x* tend à être grand quand *y* est grand et petit quand *y* est petit. Une covariance négative « élevée » signifie au contraire que *x* et *y* varient en sens inverse. Une covariance proche de zéro signifie qu'il n'existe pas de relation entre les deux.

Cependant, ce nombre peut se montrer difficile à interpréter pour deux raisons :

- son unité est le produit des unités des entrées, ce qui parfois ne veut pas dire grand-chose (par exemple, amis-minutes par jour) ;
- si chaque utilisateur a deux fois plus d'amis (mais passe le même temps sur le site), la covariance va doubler, alors qu'en fait les variables sont liées de la même façon qu'avant. Autrement dit, il est difficile de définir une « grande » covariance.

Pour cette raison, il est plus fréquent d'examiner la *corrélation*, qui est le résultat de la division de l'écart-type des deux variables :

```
def correlation(xs: List[float], ys: List[float]) -> float:
    """mesure dans quelle proportion xs et ys varient en tandem par rapport à leurs moyennes"""
    stdev_x = standard_deviation(xs)
    stdev_y = standard_deviation(ys)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(xs, ys) / stdev_x / stdev_y
    else:
        return 0 # en l'absence de variation, la corrélation est à zéro

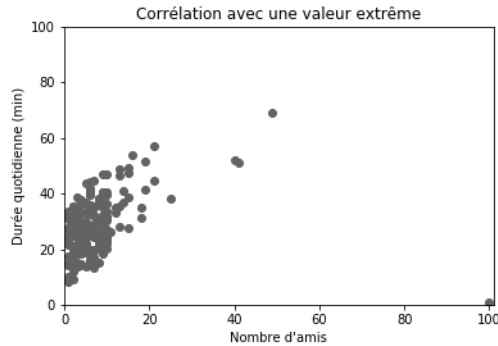
assert 0.24 < correlation(num_friendsnum_friends, daily_minutes) < 0.25
assert 0.24 < correlation(num_friendsnum_friends, daily_hours) < 0.25
```

La corrélation n'a pas d'unité et est toujours comprise entre -1 (anti-corrélation parfaite) et +1 (corrélation parfaite). Un nombre comme 0,25 représente une corrélation assez faible.

Cependant, nous avons complètement négligé de regarder nos données. Examinons la figure 5-2.

Figure 5-2

Corrélation en présence
d'une valeur extrême



La personne aux 100 amis (qui passe seulement une minute par jour sur le site) est une valeur extrême. Or, les corrélations peuvent se montrer très sensibles aux valeurs extrêmes. Que se passe-t-il si nous l'ignorons ?

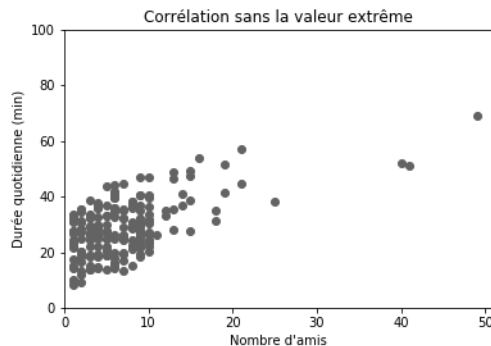
```
outlier = num_friendsnum_friends.index(100) # index de la valeur extrême
num_friendsnum_friends_good = [x
for i, x in enumerate(num_friendsnum_friends)
if i != outlier]
daily_minutes_good = [x
                        for i, x in enumerate(daily_minutes)
                        if i != outlier]
daily_hours_good = [dm / 60 for dm in daily_minutes_good]

assert 0.57 < correlation(num_friends_goodnum_friends, daily_minutes_good) < 0.58
assert 0.57 < correlation(num_friends_goodnum_friends, daily_hours_good) < 0.58
```

Sans la valeur extrême, la corrélation est beaucoup plus marquée (figure 5-3).

Figure 5-3

Corrélation après exclusion
de la valeur extrême



Vous menez votre petite enquête et vous découvrez que la valeur extrême correspond en fait à un compte de test interne que personne n'a pensé à annuler ! Donc, vous avez tout à fait le droit de l'exclure.

Le paradoxe de Simpson

Le paradoxe de Simpson correspond à une surprise classique rencontrée lorsqu'on analyse des données : les corrélations peuvent être trompeuses quand on ignore les variables parasites.

Par exemple, imaginons que vous puissiez identifier tous les membres de DataSciencester comme des data scientists de la côte Est ou de la côte Ouest. Vous décidez de chercher sur quelle côte les data scientists ont le plus d'amis :

Côte	Nombre de membres	Nombre moyen d'amis
Côte Ouest	101	8,2
Côte Est	103	6,5

Apparemment, les data scientists de la côte Ouest ont plus d'amis que ceux de la côte Est. Vos collègues avancent toutes sortes de théories pour expliquer cette situation : influence du soleil, du café, des produits bio, ou encore de l'ambiance tranquille du Pacifique.

En manipulant les données dans tous les sens, vous parvenez à une découverte troublante : si vous ne considérez que les titulaires d'un doctorat, les data scientists de la côte Est ont en moyenne plus d'amis. Et, si vous ne considérez que les non-titulaires d'un doctorat, ceux de la côte Est ont aussi en moyenne plus d'amis !

Côte	Diplôme	Nombre de membres	Nombre moyen d'amis
Côte Ouest	Doctorat	35	3,1
Côte Est	Doctorat	70	3,2
Côte Ouest	Pas de doctorat	66	10,9
Côte Est	Pas de doctorat	33	13,4

Dès que vous tenez compte du diplôme, la corrélation va dans la direction opposée ! Découper les données entre côte Est et côte Ouest masque le fait que la répartition des data scientists de la côte Est est déséquilibrée... en faveur des titulaires de doctorat.

Ce phénomène se produit assez régulièrement dans le monde réel. Le point clé tient à ce que la corrélation mesure la relation entre vos deux variables, toute chose étant égale par ailleurs. Si vos classes de données sont affectées de manière aléatoire, comme dans les règles de l'art de l'expérimentation, le « toute chose étant égale par ailleurs » n'est pas si mauvais comme hypothèse ; en revanche, quand il y a une logique sous-jacente, cela peut devenir une très mauvaise hypothèse.

La seule manière réelle d'éviter ce phénomène est de bien connaître vos données et de vous efforcer de vérifier l'existence ou non d'éventuels facteurs parasites. Évidemment, la chose n'est pas toujours possible. Si vous n'aviez pas connaissance du parcours universitaire de ces 200 data scientists, vous pourriez purement et simplement conclure qu'une plus grande sociabilité est inhérente à la côte Ouest.

Les autres chausse-trappes

Une corrélation à zéro indique qu'il n'existe pas de relation linéaire entre les deux variables. Cependant, il peut quand même exister d'autres types de relations.

Prenons un exemple :

```
x = [-2, -1, 0, 1, 2]
y = [ 2, 1, 0, 1, 2]
```

Dans ce cas, x et y ne présentent aucune corrélation. Pourtant, il y a certainement une relation, puisque chaque élément de y est égal à la valeur absolue de son correspondant dans x . Ce qu'ils n'ont pas, c'est une relation dans laquelle la comparaison de x_i et $\text{mean}(x)$ nous donne des informations sur la comparaison de y_i et $\text{mean}(y)$, soit le type de relation que cherchent les corrélations.

De plus, la corrélation ne vous dit rien de l'importance de la relation. Ainsi, les variables suivantes sont parfaitement corrélées, mais (selon ce que vous mesurez) il est tout à fait possible que cette relation ne présente aucun intérêt :

```
x = [-2, -1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

Corrélation et causalité

Vous avez sans doute déjà entendu dire que « corrélation n'est pas causalité », le plus souvent par quelqu'un confronté à des données qui défient des aspects de sa vision du monde qu'il n'a aucune envie de remettre en question. C'est pourtant un point important : si x et y sont fortement corrélés, cela peut signifier que x cause y , ou que y cause x , que chacun est la cause de l'autre, qu'un troisième facteur cause les deux... ou que tout cela ne signifie rien.

Considérons la relation entre `num_friends` et `daily_minutes`. Le fait d'avoir plus d'amis sur le site peut inciter (cause) les utilisateurs de DataSciencester à passer plus de temps en ligne. Ce sera le cas si chaque ami publie tous les jours un certain volume de contenu : ce qui veut dire que, plus vous avez d'amis (cause), plus vous passez de temps sur le site pour rester au courant de leur actualité.

Cependant, il se peut aussi que, plus vous passez de temps à argumenter sur les forums de DataSciencester, plus vous rencontrez des personnes qui partagent votre avis et qui deviennent des amis. Autrement dit, passer plus de temps sur le site a pour conséquence des utilisateurs qui ont davantage d'amis.

Troisième possibilité, les utilisateurs les plus passionnés de data science passent plus de temps sur le site (car ils trouvent les débats plus intéressants) et y cherchent plus activement des amis liés à ce domaine (parce qu'ils ne veulent fréquenter personne d'autre).

Pour se rassurer sur la causalité, il est possible d'organiser des essais *randomisés*. Si vous pouvez séparer aléatoirement vos utilisateurs en deux groupes à la démographie similaire et donner à un des groupes une expérience légèrement différente, vous aurez souvent l'intuition que ces expériences différentes sont la cause des résultats différents.

Par exemple, si vous n'avez pas peur d'être accusé de traiter vos utilisateurs comme des cobayes¹, vous pourriez choisir au hasard un sous-ensemble de ces personnes et leur présenter un contenu en provenance d'une partie de leurs amis seulement. Si ce sous-ensemble passait alors moins de temps sur le site, cela vous conforterait dans l'idée qu'avoir plus d'amis est bien la cause qui incite à passer plus de temps sur le site.

Pour aller plus loin

- `SciPy`², `pandas`³ et `StatsModels`⁴ apportent tous les trois un grand nombre de fonctions statistiques.
- La statistique est importante. Ou doit-on plutôt dire : les statistiques sont importantes ? Si vous voulez devenir un bon spécialiste en science des données, il est recommandé de lire des ouvrages sur le sujet. Beaucoup sont disponibles en ligne, notamment :
 - *Introductory Statistics*⁵, Douglas Shafer et Zhiyi Zhang (Saylor Foundation) ;
 - *OnlineStatBook*⁶, David Lane (Rice University) ;
 - *Introductory Statistics*⁷, OpenStax (OpenStax College).

1. <https://www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html>
2. <https://www.scipy.org/>
3. <https://pandas.pydata.org/>
4. <https://www.statsmodels.org/stable/index.html>
5. <https://open.umn.edu/opentextbooks/textbooks/introductory-statistics>
6. <http://onlinestatbook.com/>
7. <https://openstax.org/details/introductory-statistics>

6

Probabilités

Les lois de probabilité, si vraies en général, si fallacieuses en particulier.

Edward Gibbon

Il est difficile de pratiquer la data science sans comprendre, du moins dans leurs grandes lignes, les probabilités et les mathématiques qui vont avec. Comme nous avons traité de la statistique au chapitre précédent, nous allons faire un peu de magie et laisser de côté les aspects trop techniques.

Considérons les probabilités comme une manière de quantifier l'incertitude associée aux événements choisis au sein d'un univers d'événements. Plutôt que d'envisager les aspects techniques de cette déclaration, voyez les choses comme un lancement de dé. L'univers comprend tous les résultats possibles. Et chaque sous-ensemble de ces résultats est un événement (par exemple, « le un est sorti » ou « un nombre pair est sorti »).

Nous écrirons par convention $P(E)$ pour signifier « la probabilité de l'événement E ».

Nous utiliserons la théorie des probabilités pour construire des modèles. Nous utiliserons la théorie des probabilités pour évaluer des modèles. Nous utiliserons la théorie des probabilités absolument partout.

Si vous en avez envie, vous pouvez dissenter sur le sens philosophique profond de la théorie des probabilités (de préférence après quelques bières). Nous n'en ferons rien ici.

Dépendance et indépendance

En gros, nous dirons que deux événements E et F sont dépendants quand disposer d'informations sur la réalisation de E nous donne des informations sur la réalisation de F (et vice versa). Sinon, ils sont dits indépendants.

Par exemple, si nous lançons une pièce (non truquée) deux fois, savoir que le premier lancer a donné face ne nous dit absolument pas si le second lancer produira pile ou face ; ces événements sont indépendants. En revanche, savoir que le premier lancer a donné face nous donne certainement une information concernant la possibilité que les deux lancers donnent pile ; ces deux événements sont dépendants.

Sur un plan mathématique, nous dirons que deux événements E et F sont indépendants si la probabilité qu'ils se produisent tous les deux correspond au produit de la probabilité de chacun :

$$P(E, F) = P(E) P(F)$$

Dans l'exemple précédent, la probabilité de *premier lancer = face* est de $1/2$ et celle de *deux lancers = pile* est de $1/4$, mais la probabilité de *premier lancer = face et les deux lancers = pile* est de zéro.

Probabilité conditionnelle

Quand deux événements E et F sont indépendants, par définition nous avons :

$$P(E, F) = P(E) P(F)$$

S'ils ne sont pas nécessairement indépendants (et si la probabilité de F n'est pas 0), alors nous définissons la probabilité de E *conditionnelle selon* F telle que :

$$P(E|F) = P(E, F) / P(F)$$

Il faut interpréter cela comme la probabilité que E se produise, sachant que F s'est réalisé.

Nous écrivons souvent l'expression précédente sous la forme :

$$P(E, F) = P(E|F) P(F)$$

Quand E et F sont indépendants, vous pouvez vérifier le résultat :

$$P(E|F) = P(E)$$

Cette expression mathématique spécifie que savoir que F s'est produit ne nous donne aucune information supplémentaire pour savoir si E s'est réalisé également.

Un exemple classique, un peu délicat, consiste à considérer une famille avec deux enfants. Si nous supposons que :

- chaque enfant a autant de chances d'être une fille que d'être un garçon ;
- le sexe du deuxième enfant est indépendant du sexe du premier ; alors l'événement *aucune fille* a une probabilité de $1/4$, l'événement *un garçon, une fille* a la probabilité $1/2$ et l'événement *deux filles* a une probabilité de $1/4$.

Nous pouvons nous demander quelle est la probabilité que l'événement B *les deux enfants sont des filles* soit conditionné par l'événement G *l'aîné des enfants est une fille*. Utilisons la définition de la probabilité conditionnelle :

$$P(B|G) = P(B, G)/P(G) = P(B)/P(G) = 1/2$$

L'événement B et G (*les deux enfants sont des filles et l'aîné des enfants est une fille*) se réduit à l'événement B (une fois que vous savez que les deux enfants sont des filles, alors il est nécessairement vrai que l'aîné des enfants est une fille).

Il est très probable que ce résultat corresponde à votre intuition.

Nous pouvons nous demander quelle est la probabilité que l'événement *les deux enfants sont des filles* soit conditionné par l'événement L (*au moins un des enfants est une fille*). Surprise, la réponse est différente de la précédente !

Comme auparavant, l'événement B et L (*les deux enfants sont des filles et au moins un des enfants est une fille*) se réduit à l'événement B :

$$P(B|L) = P(B,L)/P(L) = P(B)/P(L) = 1/3$$

Comment est-ce possible ? En fait, si tout ce que vous savez c'est qu'au moins un des enfants est une fille, alors il est deux fois plus probable que la famille ait un garçon et une fille que deux filles.

Nous pouvons le vérifier en « générant » plusieurs familles :

```
import enum, random
# un Enum est un ensemble typé de valeurs énumérées
# nous pouvons l'utiliser pour rendre notre code plus lisible
class Kid(enum.Enum):
    BOY = 0 # garçon
    GIRL = 1 # fille

def random_kid() -> Kid:
    return random.choice([Kid.BOY, Kid.GIRL])

both_girls = 0
older_girl = 0
either_girl = 0
random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == Kid.GIRL:
        older_girl += 1
    if older == Kid.GIRL and younger == Kid.GIRL:
        both_girls += 1
    if older == Kid.GIRL or younger == Kid.GIRL:
        either_girl += 1

print("P(both | older):", both_girls / older_girl) # 0,514 ~ 1/2
print("P(both | either): ", both_girls / either_girl) # 0,342 ~ 1/3
```

Théorème de Bayes

Un des meilleurs amis de l'expert en data science est le théorème de Bayes. Il permet d'« inverser » les probabilités conditionnelles. Supposons que nous ayons besoin de la probabilité d'un certain événement E conditionné par un autre événement F , mais que nous n'ayons que des informations sur la probabilité de F conditionné par E . En utilisant deux fois la définition de la probabilité conditionnelle, nous déduisons que :

$$P(E|F) = P(E, F) / P(F) = P(F|E) P(E) / P(F)$$

L'événement F peut être divisé en deux événements mutuellement exclusifs F et E et F et $\neg E$. Si nous écrivons $\neg E$ pour *pas- E* (c'est-à-dire E ne se produit pas), alors on obtient la relation suivante :

$$P(F) = P(F, E) + P(F, \neg E)$$

On en déduit la formulation la plus fréquente du théorème de Bayes :

$$P(E|F) = P(F|E) P(E) / [P(F|E)P(E) + P(F|\neg E)P(\neg E)]$$

Ce théorème est souvent utilisé pour démontrer que les data scientists sont plus malins que les médecins. Imaginons une maladie qui affecte 1 personne sur 10 000 dans la population et qu'il existe un test de dépistage donnant le résultat correct (*infecté* si la personne est malade, *non infecté* si la personne est saine) 99 % du temps.

Que signifie un test positif ? Soit T_+ l'événement *votre test est positif* et D l'événement *vous êtes malade*. Le théorème de Bayes dit que la probabilité que vous ayez la maladie conditionnée au test positif est la suivante :

$$P(D|T_+) = P(T_+|D)P(D) / [P(T_+|D)P(D) + P(T_+|\neg D)P(\neg D)]$$

Nous savons que $P(T_+|D)$, la probabilité qu'un malade ait un test positif, est de 0,99. $P(D)$, la probabilité qu'une personne donnée soit malade, est de $1/10\,000 = 0,0001$. $P(T_+|\neg D)$, la probabilité qu'un non malade présente un test positif, est de 0,01. Et $P(\neg D)$, la probabilité qu'une personne donnée ne soit pas malade, est de 0,9999. Si vous reportez ces nombres dans l'énoncé du théorème de Bayes, vous obtenez :

$$P(D|T_+) = 0,98 \%$$

Cela veut dire que moins de 1 % des personnes ayant été testées positivement sont effectivement malades.

Remarque

Cela suppose que les personnes passent le test plus ou moins au hasard. Si seules les personnes qui présentent certains symptômes subissent le test, alors la condition portera sur l'événement *test positif et symptômes* et le nombre sera sans doute beaucoup plus élevé.

Une manière de voir plus intuitive consiste à imaginer une population de 1 million de personnes. Vous vous attendez à ce que 100 d'entre elles soient malades et que 99 de ces 100 personnes aient un test positif. D'un autre côté, vous vous attendez à ce que 999 900 d'entre elles

ne soient pas malades et que 9 999 de ces personnes aient un test positif. Cela signifie que vous vous attendez à ce que seulement 99 parmi (99 + 9999) personnes avec un test positif soient effectivement malades.

Variables aléatoires

Une variable aléatoire est une variable dont les valeurs possibles sont distribuées selon une loi de probabilité donnée. Une variable aléatoire très simple correspond à 1 si une pièce lancée indique face et 0 si elle indique pile. Une variable plus complexe, par exemple, mesurera le nombre de faces observées quand on lance la pièce 10 fois ou correspondra à une valeur prise dans une liste `range(10)` pour laquelle chaque nombre a la même chance de sortir.

La distribution associée donne les probabilités que la variable réalise chacune de ses valeurs possibles. La variable associée à la pièce vaut 0 avec la probabilité 0,5 et 1 avec la probabilité 0,5. La variable `range(10)` a une distribution qui affecte la probabilité 0,1 à chacun des nombres de 0 à 9.

Parfois, nous parlerons de *l'espérance* (la valeur attendue) d'une variable aléatoire, qui correspond à la moyenne de ses valeurs pondérée par leurs probabilités. La variable aléatoire associée à la pièce a une espérance de 0,5 ($= 0 \times 0,5 + 1 \times 0,5$) et la variable `range(10)` a une espérance de 4,5.

Les variables aléatoires peuvent être conditionnées par des événements. Revenons à notre exemple des deux enfants. Si x est la variable aléatoire représentant le nombre de filles, x est égale à 0 avec une probabilité de 1/4, à 1 avec une probabilité de 1/2 et à 2 avec une probabilité de 1/4.

Nous pouvons définir une nouvelle variable aléatoire y qui donne le nombre de filles conditionné par le fait qu'au moins un des enfants est une fille. Alors, y est égale à 1 avec la probabilité 2/3 et à 2 avec la probabilité 1/3. Et une variable z qui correspond au nombre de filles conditionné par le fait que l'aîné des enfants est une fille est égale à 1 avec la probabilité 1/2 et à 2 avec la probabilité 1/2.

La plupart du temps, nous utiliserons des variables aléatoires implicitement sans les mettre particulièrement en valeur. Cependant, vous les reconnaîtrez si vous leur prêtez attention.

Distributions continues

Un lancer de pièce de monnaie correspond à une distribution discrète qui associe une probabilité positive avec des résultats discrets (ou discontinus). Nous chercherons souvent des modèles de distribution correspondant à des résultats continus. Pour simplifier, nous supposerons que ces résultats sont toujours des nombres réels, ce qui n'est pas toujours le cas dans la vraie vie. Par exemple, la distribution uniforme affecte le même poids à tous les nombres entre 0 et 1.

Comme il existe une infinité de nombres entre 0 et 1, le poids affecté à chaque point pris isolément doit nécessairement être égal à zéro. Pour cette raison, nous représenterons une distribution

continue par une fonction de densité de probabilités (pdf), telle que la probabilité de voir une valeur dans un certain intervalle est égale à l'intégrale de la fonction de densité sur cet intervalle.

Remarque

Si vos souvenirs de calcul d'intégrales sont trop lointains, une manière simple de comprendre consiste à se dire que si une distribution présente une densité f de probabilités, alors la probabilité de voir une valeur entre x et $x + h$ est approximativement de $h \cdot f(x)$ si h est petit.

La densité de la distribution uniforme se résume donc comme suit :

```
def uniform_pdf(x: float) -> float:
    return 1 if 0 <= x < 1 else 0
```

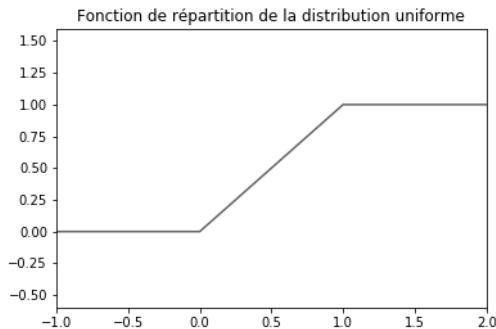
La probabilité qu'une variable aléatoire suivant cette distribution figure entre 0,2 et 0,3 est de 1/10, comme vous pouviez le deviner. La variable Python `random.random()` est une variable (pseudo-)aléatoire de densité uniforme.

Nous nous intéresserons davantage à la *fonction de distribution cumulative* (cdf), également appelée *fonction de répartition*. Elle donne la probabilité qu'une variable aléatoire soit inférieure ou égale à une valeur donnée. Il n'est pas difficile de créer la fonction de répartition de la distribution uniforme (figure 6-1) :

```
def uniform_cdf(x: float) -> float:
    """retourne la probabilité qu'une distribution aléatoire soit <= x"""
    if x == 0: return 0 # la distribution uniforme n'est jamais < 0
    elif x < 1: return x # par exemple, P(X <= 0.4) = 0.4
    else: return 1 # la distribution aléatoire uniforme est toujours < 1
```

Figure 6-1

La fonction de distribution cumulative de la distribution uniforme



Distribution normale

La distribution normale est la reine des distributions. C'est la classique courbe en cloche. Elle est complètement déterminée par deux paramètres : son espérance μ (mu) et son écart-type σ (sigma). L'espérance indique comment la cloche est centrée et l'écart-type indique la « largeur » de la cloche.

Voici sa fonction de densité :

$$f(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

```
import math
SQRT_TWO_PI = math.sqrt(2 * math.pi)

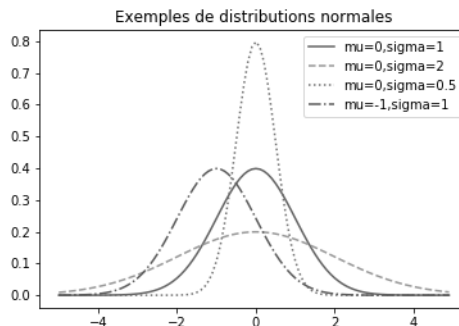
def normal_pdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (SQRT_TWO_PI * sigma))
```

Traçons quelques-unes de ces lois normales pour voir à quoi elles ressemblent (figure 6-2) :

```
import matplotlib.pyplot as plt
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs, [normal_pdf(x, sigma=1) for x in xs], '-', label='mu=0, sigma=1')
plt.plot(xs, [normal_pdf(x, sigma=2) for x in xs], '--', label='mu=0, sigma=2')
plt.plot(xs, [normal_pdf(x, sigma=0.5) for x in xs], ':', label='mu=0, sigma=0.5')
plt.plot(xs, [normal_pdf(x, mu=-1) for x in xs], '-.', label='mu=-1, sigma=1')
plt.legend()
plt.title("Exemples de distributions normales")
plt.show()
```

Figure 6-2

Exemples de distributions normales.



Quand $\mu = 0$ et $\sigma = 1$, on parle de *loi normale standard*. Si z est une variable aléatoire de loi normale standard, alors :

$$X = \sigma Z + \mu$$

est aussi normale, mais avec l'espérance μ et l'écart-type σ . Inversement, si x est une variable aléatoire normale d'espérance μ et d'écart-type σ , alors :

$$Z = (x - \mu) / \sigma$$

suit une loi normale standard.

La fonction de répartition de la loi normale ne peut pas être écrite sous forme « élémentaire », mais nous pouvons la coder en Python à l'aide de la fonction d'erreur `math.erf`¹ :

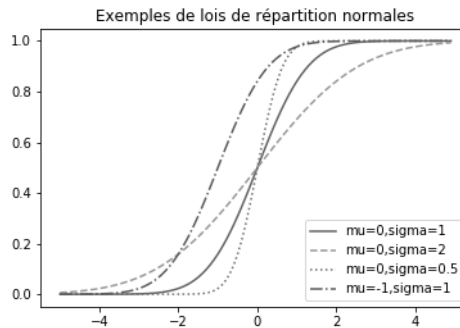
```
def normal_cdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Ici encore, nous avons représenté quelques exemples de répartitions (figure 6–3) :

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs, [normal_cdf(x, sigma=1) for x in xs], '-', label='mu=0, sigma=1')
plt.plot(xs, [normal_cdf(x, sigma=2) for x in xs], '--', label='mu=0, sigma=2')
plt.plot(xs, [normal_cdf(x, sigma=0.5) for x in xs], ':', label='mu=0, sigma=0.5')
plt.plot(xs, [normal_cdf(x, mu=-1) for x in xs], '-.', label='mu=-1, sigma=1')
plt.legend(loc=4) # en bas à droite
plt.title("Exemples de lois de répartition normales")
plt.show()
```

Figure 6–3

Exemples de lois de répartition normales



Parfois, nous aurons besoin d'inverser la fonction `normal_cdf` pour trouver la valeur correspondant à une probabilité spécifique. Il n'existe pas de manière simple de calculer l'inverse, mais `normal_cdf` est continue et strictement croissante, donc nous pourrons effectuer une recherche par dichotomie² (recherche binaire).

1. https://fr.wikipedia.org/wiki/Fonction_d'erreur

2. https://fr.wikipedia.org/wiki/Recherche_dichotomique

```
def inverse_normal_cdf(p: float,
                      mu: float = 0,
                      sigma: float = 1,
                      tolerance: float = 0.00001) -> float:
    """trouver l'inverse approximatif avec une recherche par dichotomie"""
    # si non standard, calculer le standard et redéfinir l'échelle
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z = -10.0 # normal_cdf(-10) est très proche de 0
    hi_z = 10.0 # normal_cdf(10) est très proche de 1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2 # considérer le point médian
        mid_p = normal_cdf(mid_z) # et la valeur de cdf à cet endroit
        if mid_p < p:
            low_z = mid_z # le point médian est encore trop bas
        else:
            hi_z = mid_z # le point médian est encore trop haut
    return mid_z
```

La fonction segmente l'intervalle en deux parties égales plusieurs fois jusqu'à le réduire à une valeur z suffisamment proche de la probabilité recherchée.

Théorème de la limite centrale

Une des raisons de l'intérêt porté à la distribution normale tient au théorème de la limite centrale : une variable aléatoire définie comme la moyenne d'un grand nombre de variables aléatoires indépendantes et de distribution identique suit approximativement une loi (distribution) normale.

En particulier, si $x_1 \dots x_n$ sont des variables aléatoires d'espérance μ et d'écart-type σ et si n est grand, alors :

$$\frac{1}{n}(x_1 + \dots + x_n)$$

est approximativement de distribution normale avec l'espérance μ et l'écart-type $\sigma / \sqrt{n}$. De même (et souvent plus intéressant) :

$$\frac{(x_1 + \dots + x_n) - \mu n}{\sigma \sqrt{n}}$$

suit approximativement une loi normale, d'espérance 0 et d'écart-type 1.

Une manière simple d'illustrer ce résultat consiste à examiner des variables aléatoires binomiales qui ont deux paramètres n et p . Une variable aléatoire binomiale (n, p) est simplement la somme de n variables aléatoires indépendantes de Bernoulli (p) , dont chacune vaut 1 avec une probabilité p et 0 avec une probabilité $1 - p$:

```
def bernoulli_trial(p: float) -> int:
    """retourne 1 avec une probabilité p et 0 avec une probabilité 1-p"""
    return 1 if random.random() < p else 0

def binomial(n: int, p: float) -> int:
    """retourne la somme des n essais Bernoulli (p)"""
    return sum(bernoulli_trial(p) for _ in range(n))
```

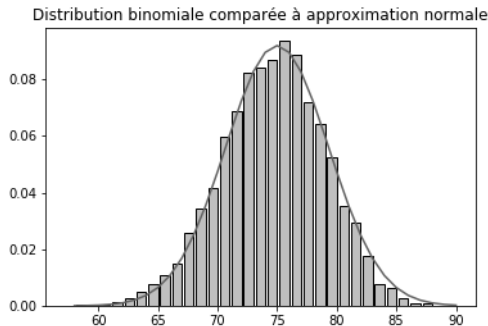
L'espérance de la variable de Bernoulli (p) est p et son écart-type est $\sqrt{p(1-p)}$. Le théorème de la limite centrale s'énonce ainsi : quand n est de plus en plus grand, une variable binomiale (n, p) est approximativement une variable aléatoire normale d'espérance $\mu = np$ et d'écart-type $\sigma = \sqrt{np(1-p)}$. Si nous représentons graphiquement les deux, vous pourrez facilement voir la ressemblance :

```
from collections import Counter
def binomial_histogram(p: float, n: int, num_points: int) -> None:
    """trace un histogramme d'une loi binomiale(n,p)"""
    data = [binomial(n, p) for _ in range(num_points)]
    # graphique à barres représentant les exemples de loi binomiale
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')
    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))
    # courbe représentant l'approximation de la loi normale
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs, ys)
    plt.title("Distribution binomiale comparée à approximation normale")
    plt.show()
```

Par exemple, quand vous appelez `make_hist(0.75, 100, 10000)`, vous obtenez le graphique de la figure 6-4.

Figure 6-4

Sortie graphique de `make_hist`
(`histogramme_loi_binomiale`)



La morale de cette approximation est que, si vous voulez connaître la probabilité qu'une pièce non truquée donne plus de 60 fois face pour 100 lancers, vous pouvez l'estimer comme la probabilité que `Normale(50, 5)` soit plus grand que 60, ce qui est plus facile à calculer que la fonction de répartition de `Binomiale(100, 0, 5)`. En réalité, dans la plupart des applications, vous disposerez probablement d'un logiciel statistique qui se fera un plaisir de calculer à votre place toutes les probabilités que vous voudrez.

Pour aller plus loin

- `scipy.stats`³ contient des fonctions de densité et de répartition pour les distributions de probabilités les plus répandues.
- Il vous est recommandé d'étudier un manuel de probabilités. Le meilleur que je connaisse est *Introduction to Probability*⁴, de Charles M. Grinstead et J. Laurie Snell (American Mathematical Society). Il est disponible en ligne (en anglais).

3. <https://docs.scipy.org/doc/scipy/reference/stats.html>

4. http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html

Hypothèses et inférences

C'est la marque d'une personne vraiment intelligente que d'être émue par les statistiques.

George Bernard Shaw

Qu'allons-nous faire de toutes ces statistiques et de cette théorie des probabilités ? La partie scientifique de la data science se préoccupe souvent de formuler et de tester des hypothèses concernant les données et les processus qui les ont générées.

Test statistique d'une d'hypothèse

Souvent, les data scientists veulent tester une hypothèse donnée pour savoir si elle a des chances de se vérifier. Pour nous, les hypothèses sont des affirmations que nous pouvons traduire en termes de statistiques sur les données ; des affirmations comme *cette pièce n'est pas truquée* ou *les data scientists préfèrent Python à R* ou encore *les gens seront plus enclins à quitter une page sans même la lire si nous faisons surgir une annonce publicitaire importune avec une minuscule touche de fermeture bien cachée*. En s'appuyant sur diverses suppositions, ces statistiques peuvent être vues comme l'observation de variables aléatoires qui suivent des lois (distributions) connues, ce qui nous permet d'affirmer que ces suppositions sont plus ou moins vraisemblables.

Dans le raisonnement classique, nous avons une hypothèse nulle H_0 , qui représente une position par défaut, et une autre hypothèse (de substitution) H_1 avec laquelle nous voulons la comparer. Nous utilisons les statistiques pour décider si H_0 est fausse ou non. Un exemple sera certainement plus parlant.

Exemple : lancer de pièce

Imaginons que nous voulons vérifier qu'une pièce n'est pas truquée. Émettons l'hypothèse qu'elle a une probabilité p de donner face. Notre hypothèse nulle stipule que la pièce n'est pas truquée ; en d'autres termes, $p = 0,5$. Nous allons tester cette hypothèse contre l'autre hypothèse : $p \neq 0,5$.

Notre test suppose donc de lancer la pièce n fois et de compter le nombre x de sorties face. Chaque lancer de pièce est un essai de Bernoulli, ce qui veut dire que x est une variable aléatoire Binomiale(n, p), que nous pouvons approcher par une distribution normale (chapitre 6) :

```
from typing import Tuple
import math
def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """trouver mu et sigma correspondant à Binomiale(n,p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

Chaque fois qu'une variable aléatoire suit une distribution normale, nous pouvons utiliser `normal_cdf` pour prévoir la probabilité que la valeur réalisée s'inscrive à l'intérieur (ou à l'extérieur) d'un intervalle donné :

```
from scratch.probability import normal_cdf # Supposant que les outils de probabilités
# présentés dans le précédent chapitre ont été sauvegardés dans le module outils_proba.

# normal_cdf correspond à la probabilité que la variable soit en dessous
# d'un seuil normal_probability_below = normal_cdf au-dessus du seuil
# si elle n'est pas sous le seuil.
def normal_probability_above(lo: float,
    mu: float = 0,
    sigma: float = 1) -> float:
    """la probabilité qu'un N(mu, sigma) soit supérieur à lo"""
    return 1 - normal_cdf(lo, mu, sigma)

def normal_probability_between(lo: float,
    hi: float,
    mu: float = 0,
    sigma: float = 1) -> float:
    """probabilité qu'un N(mu, sigma) soit dans l'intervalle (bas, haut)"""
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

def normal_probability_outside(lo: float,
    hi: float,
    mu: float = 0,
    sigma: float = 1) -> float:
    """probabilité qu'un N(mu, sigma) soit hors intervalle (bas, haut)"""
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

Nous pouvons aussi faire le raisonnement inverse : rechercher la zone excluant les extrémités de la distribution ou l'intervalle (symétrique) autour de l'espérance qui correspond à un certain niveau de vraisemblance. Par exemple, si nous voulons trouver un intervalle centré sur l'espérance et contenant 60 % de probabilité, nous devons rechercher les seuils à proximité desquels les extrémités inférieures et supérieures contiennent chacune 20 % de la probabilité :

```
from scratch.probability import inverse_normal_cdf
# Supposant que les outils de probabilités présentés dans le précédent
# chapitre ont été sauvegardés dans le module outils_proba.

def normal_upper_bound(probability: float,
mu: float = 0,
sigma: float = 1) -> float:
    """retourne la valeur de z pour laquelle P(Z <= z) = probabilité"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability: float,
mu: float = 0,
sigma: float = 1) -> float:
    """retourne la valeur de z pour laquelle P(Z >= z) = probabilité"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability: float,
mu: float = 0,
sigma: float = 1) -> Tuple[float, float]:
    """Retourne les limites symétriques (par rapport à la moyenne) qui contiennent la
    probabilité spécifiée."""
    tail_probability = (1 - probability) / 2
    # pour la limite supérieure, tail_probability doit être au-dessus
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)
    # pour la limite inférieure, tail_probability doit être en dessous
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)
    return lower_bound, upper_bound
```

En particulier, supposons que nous lançons la pièce $n = 1\,000$ fois. Si notre hypothèse de non-trucage est vraie, x devrait être distribuée approximativement selon une courbe normale d'espérance 500 et d'écart-type 15,8 :

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

Nous devons prendre une décision sur le sens que nous donnons à « significatif » afin de valider ou d'infirmer notre hypothèse : jusqu'à quel point sommes-nous prêts à accepter une erreur de type 1 (faux positif) qui nous amènerait à rejeter H_0 même si elle est vraie ? Pour des raisons qui se perdent dans la nuit des temps, ce seuil est souvent valorisé à 5 % ou 1 %. Choisissons 5 %.

Considérons le test qui rejette H_0 si x tombe à l'extérieur des limites données par :

```
# (469, 531)
lower_bound, upper_bound = normal_two_sided_bounds(0.95, mu_0, sigma_0)
```

En supposant que p soit réellement égale à 0,5 (c'est-à-dire que H_0 est vraie), il y a seulement 5 % de risque que nous observions un x qui se situe en dehors de cet intervalle, ce qui est exactement la valeur significative voulue. Autrement dit, si H_0 est vraie, alors ce test va donner le résultat correct environ 19 fois sur 20.

Souvent, nous nous intéressons à la puissance d'un test, c'est-à-dire à la probabilité de ne pas faire une erreur de type 2 (faux négatif), autrement dit de ne pas rejeter H_0 bien qu'elle soit fausse. Pour mesurer ce résultat, nous devons d'abord préciser ce que signifie exactement *H_0 est fausse*. Savoir seulement que p n'est pas égale à 0,5 ne vous renseigne pas vraiment sur la distribution de x . En particulier, examinons ce qui arrive si p est réellement égale à 0,55, ce qui veut dire que la pièce est légèrement biaisée en faveur de face. Dans ce cas, nous pouvons calculer la puissance du test comme suit :

```
# les limites à 95 % sur la base de l'hypothèse que p vaut 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)
# mu et sigma sur la base de p = 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)
# Une erreur de type 2 signifie que nous n'avons pas réussi à rejeter
# l'hypothèse nulle ce qui se produira si x est encore dans notre
# intervalle d'origine.
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability # 0.887
```

Imaginons à présent que l'hypothèse nulle correspond au fait que la pièce n'est pas biaisée en faveur de face, ou que $p \leq 0,5$. Dans ce cas, nous voulons qu'un test unilatéral rejette l'hypothèse nulle quand x est beaucoup plus grand que 500, mais non quand x est plus petit que 500. De fait, tester une valeur significative de 5 % revient à dire utiliser `normal_probability_below` pour trouver les seuils en dessous desquels se situent 95 % des probabilités :

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# vaut 526 (<531, car nous avons besoin d'une probabilité plus grande
# dans la partie supérieure).
type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1 - type_2_probability # 0.936
```

C'est un test beaucoup plus puissant, car il ne rejette plus H_0 quand x est en dessous de 469 (ce qui a très peu de chances de se produire si H_1 est vraie) mais, à la place, il rejette H_0 quand x est compris entre 526 et 531 (ce qui risque de se produire si H_1 est vraie).

p-values

Une autre manière d'envisager le test précédent consiste à utiliser les p-values. Au lieu de choisir des limites basées sur un seuil de probabilités, nous calculons la probabilité (en supposant que H_0 est vraie) de voir apparaître une valeur au moins aussi extrême que celle que nous avons véritablement observée.

Pour notre test bilatéral destiné à savoir si la pièce n'est pas truquée, calculons :

```
def two_sided_p_value(x: float, mu: float = 0, sigma: float = 1) -> float:
    """ Quelle est la probabilité de voir apparaître une valeur au moins aussi extrême
    que x (dans une direction ou l'autre) si nos valeurs sont issues d'un N(mu, sigma) ?
    """
    if x >= mu:
        # x est plus grand que l'espérance
        # la queue de distribution est donc tout ce qui est supérieur à x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # x est plus petit que l'espérance
        # la queue de distribution est donc tout ce qui est inférieur à x
        return 2 * normal_probability_below(x, mu, sigma)
```

Si nous avons vu 530 fois face, nous ferions le calcul suivant :

```
two_sided_p_value(529.5, mu_0, sigma_0) # 0.062
```

Remarque

Pourquoi utiliser 529,5 au lieu de 530 ? C'est ce qui s'appelle une correction de continuité^a. Elle prend en compte le fait que `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` est une meilleure estimation de la probabilité de voir face sortir 530 fois que ne l'est `normal_probability_between(530, 531, mu_0, sigma_0)`.

De même, `normal_probability_above(529.5, mu_0, sigma_0)` est une meilleure estimation de la probabilité de voir face sortir au moins 530 fois. Vous avez peut-être remarqué que nous avons déjà utilisé cette notation dans le code de la figure 6-4.

a. https://fr.wikipedia.org/wiki/Correction_de_continuité

Pour vous convaincre que c'est une estimation raisonnable, faisons une simulation :

```
import random
extreme_value_count = 0
for _ in range(1000):
    num_heads = sum(1 if random.random() < 0.5 else 0 # compteur de faces
                    for _ in range(1000)) # en 1 000 lancers
    if num_heads >= 530 or num_heads <= 470: # et compte le nombre de fois
        extreme_value_count += 1 # le tirage est extrême

# la valeur p-value était de 0.062 => ~62 valeurs extrêmes sur 1 000
assert 59 < extreme_value_count < 65, f"{extreme_value_count}"
```

Comme la p -value est supérieure à notre valeur significative de 5 %, nous ne rejetons pas l'hypothèse nulle. En revanche, si nous avions pris 532, la p -value serait alors :

```
two_sided_p_value(531.5, mu_0, sigma_0) # 0.0463
```

Ce qui est inférieur à 5 % et impliquerait que nous rejetons l'hypothèse nulle. C'est exactement le même test qu'auparavant, mais avec une approche différente des statistiques.

De même, nous devrions avoir :

```
upper_p_value = normal_probability_above  
lower_p_value = normal_probability_below
```

Pour notre test unilatéral, si nous avons vu sortir face 525 fois, nous ferions le calcul suivant :

```
upper_p_value(524.5, mu_0, sigma_0) # 0.061
```

Ce qui veut dire ne pas rejeter l'hypothèse nulle. Si nous avons obtenu face 527 fois, le calcul serait alors :

```
upper_p_value(526.5, mu_0, sigma_0) # 0.047
```

Et nous rejeterions l'hypothèse nulle.

Avertissement

Assurez-vous que vos données suivent approximativement une loi normale avant d'utiliser `normal_probability_above` pour calculer les p -values. Les annales de la mauvaise data science regorgent d'exemples de personnes déclarant que la chance d'observer un certain événement se produisant de manière aléatoire est de une sur un million, alors que ce qu'elles veulent dire en réalité est « si nous supposons que les données sont distribuées normalement, la chance de voir... », ce qui ne veut absolument rien dire si les données n'ont pas une distribution normale.

Il existe différents tests statistiques pour vérifier la normalité. Un bon début consiste à commencer par une simple représentation graphique des données.

Intervalle de confiance

Nous avons testé des hypothèses sur la probabilité p de voir sortir face, ce qui constitue un *paramètre* de la distribution inconnue de face. Dans ce genre de cas, il existe une troisième approche qui consiste à construire un *intervalle de confiance* autour de la valeur observée pour le paramètre.

Par exemple, on peut estimer la probabilité d'une pièce truquée en regardant la valeur moyenne des variables de Bernoulli qui correspondent à chaque lancer : 1 si face, 0 si pile. Si nous observons face 525 fois pour 1 000 lancers, nous estimerons que p vaut 0,525.

Quelle confiance pouvons-nous accorder à cette estimation ? En fait, si nous connaissions la valeur exacte de p , le théorème de la limite centrale (chapitre 6) nous dit que, en moyenne, ces variables de Bernoulli devraient approximativement suivre une loi normale d'espérance p et d'écart-type $\text{math.sqrt}(p * (1 - p) / 1\,000)$.

Ici cependant, nous ne connaissons pas p , donc nous utilisons une valeur approchée :

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
```

Si la chose n'est pas totalement justifiée, elle reste couramment pratiquée. Avec l'approximation de la loi normale, nous pouvons conclure que nous sommes « confiants à 95 % » que l'intervalle suivant contiendra le vrai paramètre p :

```
normal_two_sided_bounds(0.95, mu, sigma) # [0.4940, 0.5560]
```

Remarque

Cette affirmation concerne l'intervalle, mais non p . Vous devez la comprendre comme l'affirmation que, si vous répétiez l'expérience plusieurs fois, 95 % du temps la « vraie » valeur (qui est la même à chaque fois) se situerait dans l'intervalle de confiance observé (qui peut être différent à chaque fois).

Plus particulièrement, nous ne concluons pas que notre pièce est truquée, car 0,5 tombe dans notre intervalle de confiance.

Si nous avions observé face 540 fois, nous aurions :

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

Dans ce cas, la « pièce non truquée » ne se situe pas dans l'intervalle de confiance. L'hypothèse « pièce non truquée » ne réussit pas le test qu'elle était censée réussir dans 95 % des cas si elle était vraie.

P-hacking

Par définition, une procédure qui rejette à tort l'hypothèse nulle seulement 5 % du temps rejette l'hypothèse nulle à tort dans 5 % des cas :

```
from typing import List
def run_experiment() -> List[bool]:
    """lancer une pièce non truquée 1 000 fois, Vrai = face, Faux = pile"""
    return [random.random() < 0.5 for _ in range(1000)]
```

```
def reject_fairness(experiment: List[bool]) -> bool:
    """utilisation des niveaux de signification de 5 %"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
    for experiment in experiments
    if reject_fairness(experiment)])
assert num_rejections == 46
```

Cela veut dire que, si vous essayez de trouver des résultats « significatifs », vous le pourrez en général. Testez suffisamment d'hypothèses vis-à-vis de votre jeu de données et une d'entre elles apparaîtra certainement significative. Éliminez les extrêmes adéquats et vous pourrez probablement faire passer votre valeur p sous la barre des 0,05.

Remarque

Nous avons fait quelque chose de vaguement ressemblant précédemment sous l'intitulé « La corrélation » au chapitre 5.

Ce phénomène, parfois appelé *P-hacking*¹, est une conséquence de « l'inférence issue de la structure des p -values ». L'article de Jacob Cohen intitulé *The Earth Is Round*² (« La terre est ronde ») une bonne critique de cette approche.

Si vous voulez adopter une approche scientifique sérieuse, vous devez :

- formuler vos hypothèses avant de consulter vos données ;
- nettoyer vos données sans penser à vos hypothèses ;
- garder à l'esprit que les p -values ne remplacent pas le bon sens.

Une autre approche sera abordée plus loin dans ce chapitre : l'inférence bayésienne.

Exemple : effectuer un test A/B

Une de vos principales missions chez DataSciencester est « d'optimiser l'expérience client », ce qui est un euphémisme signifiant que vous devez inciter les internautes à cliquer sur les annonces publicitaires. Un de vos annonceurs a développé une nouvelle boisson énergisante qui cible les data scientists et le responsable Publicité veut que vous l'aidiez à choisir entre l'annonce A *Que? régal !* et l'annonce B *Moins de biais*.

1. <https://www.nature.com/news/scientific-method-statistical-errors-1.14700>

2. http://www.iro.umontreal.ca/~dift3913/cours/papers/cohen1994_The_earth_is_round.pdf

En bon scientifique (des données), vous décidez de lancer une expérience en présentant aux visiteurs du site une des deux annonces au hasard et en relevant le nombre de personnes qui cliquent dans chaque cas.

Si 990 visiteurs sur 1 000 ayant vu A cliquent sur cette annonce alors que seulement 10 sur 1 000 de ceux qui ont vu l'annonce B cliquent sur B, vous pouvez être sûr que A est meilleure que B. Toutefois, que conclure si les différences ne sont pas si marquées ? C'est là que nous allons faire intervenir l'inférence statistique.

Soit n_A le nombre de personnes qui ont vu l'annonce A et n_A le nombre de celles qui ont cliqué dessus. Nous pouvons envisager chaque visualisation d'annonce comme un essai de Bernoulli dans lequel p_A est la probabilité que quelqu'un clique sur A. Ensuite, (si n_A est grand, ce qui est le cas), nous savons que n_A / n_A suit approximativement une loi normale d'espérance p_A et d'écart-type $\sigma_A = \sqrt{p_A(1-p_A)/n_A}$. De même pour l'autre annonce, n_B / n_B est approximativement une variable aléatoire normale d'espérance p_B et d'écart-type $\sigma_B = \sqrt{p_B(1-p_B)/n_B}$. Nous exprimons cette approche par le code suivant :

```
def estimated_parameters(N: int, n: int) -> Tuple[float, float]:
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

Si nous considérons que les deux lois normales sont indépendantes (ce qui paraît raisonnable, car les essais de Bernoulli individuels doivent être indépendants), alors leur différence devrait également suivre une loi normale, d'espérance $p_B - p_A$ et d'écart-type $\sqrt{\sigma_A^2 + \sigma_B^2}$.

Remarque

En fait, nous trichons quand même un peu. Ce calcul ne fonctionne réellement ainsi que si vous connaissez les écarts-types. Ici, nous les estimons à partir des données, ce qui veut dire que nous devrions utiliser une *t-distribution* (loi de Student). Néanmoins, pour des ensembles de données suffisamment volumineux, les deux sont si proches qu'il n'y a pas beaucoup de différence.

Cela veut dire que nous pouvons tester l'hypothèse nulle selon laquelle p_A et p_B sont les mêmes (c'est-à-dire $p_A - p_B = 0$) en utilisant la statistique suivante :

```
def a_b_test_statistic(N_A: int, n_A: int, N_B: int, n_B: int) -> float:
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

Cela devrait correspondre approximativement à une distribution normale standard.

Par exemple, si le slogan A récolte 200 clics pour 1 000 vues, alors que le slogan B en récolte 180, la statistique vaut ainsi :

```
z = a_b_test_statistic(1000, 200, 1000, 180) # -1.14
```

La probabilité de voir une si grande différence si les espérances étaient égales est de :

```
two_sided_p_value(z) # 0.254
```

C'est suffisant pour vous empêcher de conclure qu'il y a une grande différence.

D'un autre côté, si le slogan B reçoit seulement 150 clics, alors :

```
z = a_b_test_statistic(1000, 200, 1000, 150) # -2.95
two_sided_p_value(z) # 0.003
```

Cela signifie qu'il y a seulement une probabilité de 0,003 de voir une grande différence dans le cas où les annonces ont la même efficacité.

Inférence bayésienne

La procédure que nous venons de voir nécessitait des décisions sur les probabilités dans nos tests : par exemple, « il n'y a que 0.3 % de chances d'observer des statistiques aussi extrêmes si notre hypothèse nulle est vraie ».

Une autre approche de l'inférence consiste à traiter les paramètres inconnus comme des variables aléatoires. L'analyste commence avec une *distribution a priori* des paramètres et utilise ensuite les données observées et le théorème de Bayes pour déduire une *distribution a posteriori* mise à jour. Plutôt que d'émettre des jugements de probabilité portant sur les tests, vous les faites porter sur les paramètres eux-mêmes.

Par exemple, quand le paramètre inconnu est une probabilité (comme dans notre exemple de lancer de pièce), nous utilisons souvent une distribution a priori bêta dont l'ensemble des valeurs est compris entre 0 et 1 :

```
def B(alpha: float, beta: float) -> float:
    """une constante de normalisation pour que la probabilité totale soit 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x: float, alpha: float, beta: float) -> float:
    if x <= 0 or x >= 1: # aucun poids en dehors de [0, 1]
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

De manière générale, la distribution est centrée sur :

```
alpha / (alpha + beta)
```

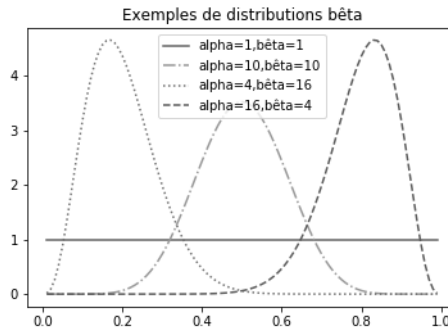
Plus alpha et beta sont grands et plus la distribution est « resserrée ».

Par exemple, si alpha et beta valent tous les deux 1, on retrouve une distribution uniforme (centrée sur 0,5 et très dispersée). Si alpha est beaucoup plus grand que beta, l'essentiel du

poids est proche de 1. Si α est beaucoup plus petit que β , l'essentiel du poids est proche de 0 (figure 7-1).

Figure 7-1

Exemples de distributions bêta



Soit une distribution a priori de p . Supposons que nous ne voulions pas préjuger de l'honnêteté de la pièce et que nous décidions qu' α et β sont tous deux égaux à 1. Nous pourrions aussi être persuadés que la pièce fait sortir face 55 % du temps et choisir $\alpha = 55$ et $\beta = 45$.

Puis nous lançons la pièce un certain nombre de fois et nous obtenons h fois face et t fois pile. Le théorème de Bayes (et d'autres considérations mathématiques trop indigestes pour être développées ici) nous indique que la distribution a posteriori de p est encore une distribution bêta, mais avec comme paramètres $\alpha + h$ et $\beta + t$.

Remarque

Le fait que la distribution a posteriori suive la loi bêta n'a rien d'une coïncidence. Le nombre de retombées côté face est donné par une distribution binomiale et la loi bêta est la conjuguée a priori^a de la distribution binomiale. En d'autres termes, quand vous mettez à jour une β a priori en utilisant des observations issues de la binomiale correspondante, vous retrouvez la β a posteriori.

a. https://www.johndcook.com/blog/conjugate_prior_diagram/

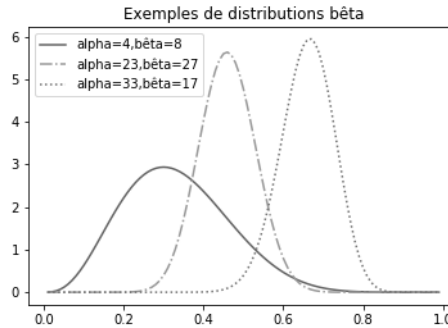
Supposons que vous lanciez la pièce 10 fois et que face sorte 3 fois seulement. Si vous avez commencé avec une distribution uniforme a priori (refusant d'admettre que la pièce puisse être truquée), votre distribution a posteriori sera une $\text{beta}(4, 8)$ centrée sur 0,33. Comme vous avez considéré que toutes les probabilités ont la même vraisemblance, votre meilleure prévision est assez proche de la réalité observée.

Si vous avez commencé avec une $\text{beta}(20, 20)$ (exprimant ainsi le fait que la pièce est globalement correcte), votre distribution a posteriori sera une $\text{beta}(23, 27)$ centrée sur 0,46, indiquant une conviction revue : peut-être que la pièce est un peu biaisée en faveur de pile.

Si vous avez commencé avec une $\text{beta}(30, 10)$ (exprimant votre conviction que la pièce est biaisée pour tomber sur face dans 75 % des lancers), votre distribution a posteriori sera une $\text{beta}(33, 17)$ centrée sur 0,66. Dans ce cas, la pièce est biaisée en faveur de pile, mais pas autant que vous le pensiez initialement.

Figure 7-2

Des lois a posteriori à partir
de différentes lois a priori



Si vous continuez à lancer la pièce encore et encore, la loi a priori aura de moins en moins d'importance jusqu'à ce que vous obteniez (presque) la même distribution a posteriori, quelle que soit l'hypothèse de départ.

Par exemple, quelle que soit votre conviction initiale sur le caractère biaisé de la pièce, il sera difficile de vous en tenir à votre opinion initiale après 1 000 retombées côté face sur 2 000 lancers... à moins d'être un original qui a choisi pour distribution a priori quelque chose du style `beta(1000000, 1)`.

Ce qui est intéressant, c'est que nous pouvons ainsi établir des probabilités au sujet des hypothèses : « à partir de la loi a priori et des données observées, il existe seulement 5 % de chances que la probabilité de retomber sur face soit comprise entre 49 et 51 % ». Sur le plan philosophique, c'est très différent de l'affirmation suivante : « si la pièce n'était pas truquée, nous pourrions prévoir d'observer des données aussi extrêmes seulement 5 % du temps ».

Le recours à l'inférence bayésienne pour tester des hypothèses est sujet à polémique, en partie à cause de ses développements mathématiques parfois très complexes et en partie à cause du caractère subjectif du choix de la distribution a priori. Nous ne l'utiliserons pas dans ce livre, mais il est important de connaître son existence.

Pour aller plus loin

- Nous n'avons fait qu'effleurer la surface de ce que vous devriez savoir de l'inférence statistique. Les ouvrages recommandés à la fin du chapitre 5 vous en apprendront davantage.
- Coursera propose par ailleurs un cours intitulé *Data Analysis and Statistical Inference*³ qui couvre plusieurs aspects de cette problématique.

3. <https://www.coursera.org/specializations/statistics>

8

Descente de gradient

Se vanter de qui on descend revient à se vanter de ce qu'on doit aux autres.

Sénèque

Souvent, en data science, nous essaierons de trouver le meilleur modèle pour une situation donnée. En général « meilleur » veut dire « qui minimise l'erreur du modèle » ou « qui maximise la vraisemblance des données ». En d'autres termes, ce modèle est la solution d'un problème d'optimisation.

Nous devons donc nous attacher à résoudre un certain nombre de problèmes d'optimisation et, plus particulièrement, à les résoudre à partir de rien. Nous utiliserons la méthode dite de *descente de gradient*, qui se prête particulièrement bien à une approche à partir de rien. Certes, ce n'est sans doute pas l'approche la plus excitante en soi, mais elle vous permettra de faire des choses tellement plus intéressantes tout au long du livre qu'il faut accepter d'en passer par là.

Idée cachée derrière la descente de gradient

Soit une fonction f qui accepte en entrée un vecteur de nombres réels et qui donne en sortie un unique nombre réel. Rappelez-vous la fonction `sum_of_squares` (chapitre 4) :

```
def sum_of_squares(v: Vecteur) -> float:
    """retourne  $v_1^2 + \dots + v_n^2$ """
    return dot(v, v)
```

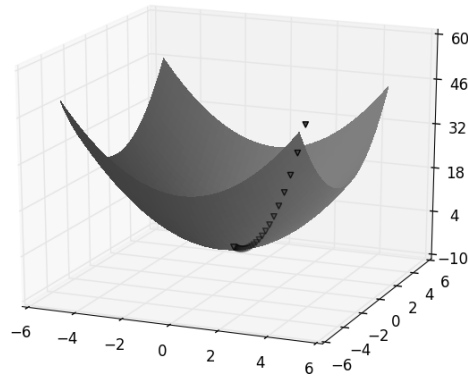
Nous aurons souvent à maximiser (ou minimiser) ce genre de fonction. En d'autres termes, nous devons trouver l'entrée v qui produit la plus grande (ou la plus petite) valeur possible.

Pour des fonctions comme les nôtres, le **gradient** (si vous avez oublié vos cours de calcul infinitésimal, c'est le vecteur des dérivées partielles) donne la direction vers laquelle la fonction augmente le plus rapidement.

Donc, une méthode pour maximiser une fonction consiste à prendre un point de départ au hasard, calculer le gradient, appliquer un petit déplacement dans la direction de ce dernier (c'est-à-dire la direction qui correspond à l'augmentation la plus importante), puis répéter le processus à partir de ce nouveau point de départ. De la même façon, il est possible de minimiser une fonction en effectuant de petits déplacements dans la direction opposée (figure 8-1).

Figure 8-1

Trouver un minimum à partir de la descente de gradient



Remarque

Si une fonction possède un unique minimum global, cette procédure le trouvera certainement. Si une fonction possède plusieurs minima locaux, la procédure pourrait conduire à en désigner un qui n'est pas le minimum global ; dans ce cas, il faudra recommencer à partir de différents points de départ. Si une fonction ne possède pas de minimum, la procédure peut se poursuivre sans fin.

Estimer le gradient

Si f est une fonction à une variable, sa dérivée au point x mesure comment $f(x)$ évolue quand on effectue un petit changement h sur x . La dérivée est définie comme la limite des quotients de différence quand h tend vers zéro :

```
from typing import Callable
def difference_quotient(f: Callable[[float], float],
    x: float,
    h: float) -> float:
    return (f(x + h) - f(x)) / h
```

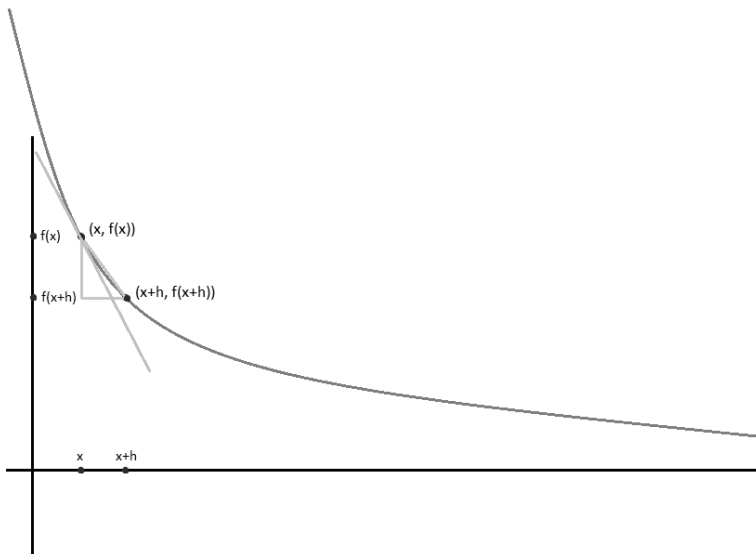
Remarque

De nombreux étudiants débutant en calcul infinitésimal se sont heurtés à la définition mathématique de la limite. Nous allons contourner cette difficulté en déclarant ici simplement que la limite signifie ce que vous pensez qu'elle signifie.

La dérivée est la pente de la tangente en $(x, f(x))$, alors que le quotient de différence représente la pente de la ligne presque-tangente qui passe par $(x + h, f(x + h))$. Quand h devient de plus en plus petit, la presque-tangente s'approche de plus en plus de la tangente (figure 8-2).

Figure 8-2

Approximation d'une dérivée avec un quotient de différence



Pour de nombreuses fonctions, il est facile de calculer les dérivées exactes. Prenons par exemple la fonction d'élévation au carré :

```
def square(x: float) -> float:
    return x * x
```

Elle a pour dérivée :

```
def derivative(x: float) -> float:
    return 2 * x
```

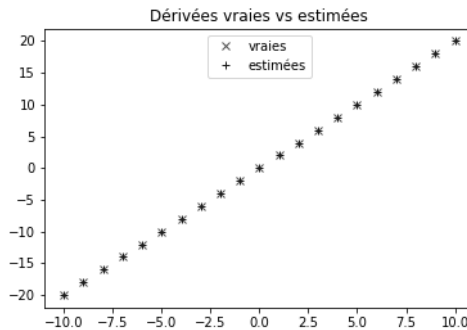
Vous le vérifierez facilement en calculant véritablement le quotient des différences et en prenant la limite. Pour ce faire, l'algèbre de vos années de lycée suffira.

Que se passe-t-il si vous ne pouvez pas (ou ne voulez pas) trouver le gradient ? Bien qu'il ne soit pas possible de manipuler des limites en Python, nous pouvons estimer les dérivées en évaluant le quotient des différences pour une quantité h très réduite (figure 8–3) :

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for x in xs]
# graphique pour prouver que ce sont fondamentalement les mêmes
import matplotlib.pyplot as plt
plt.title("Actual Derivatives vs. Estimates")
plt.plot(xs, actuals, 'rx', label='Actual') # x rouge
plt.plot(xs, estimates, 'b+', label='Estimate') # + bleu
plt.legend(loc=9)
plt.show()
```

Figure 8–3

Pertinence de l'estimation
par le quotient des différences



Quand une fonction f présente plusieurs variables, elle a plusieurs dérivées partielles, chacune indiquant comment f évolue quand nous procédons à de petites modifications sur une seule des variables d'entrée.

Nous calculons sa i -ème dérivée en la traitant comme une fonction à une seule variable de rang i , tandis que les autres sont fixes :

```
def partial_difference_quotient(f: Callable[[Vector], float],
                                v: Vector,
                                i: int,
                                h: float) -> float:
    """calcul le i-ème quotient partiel des différences de f en v"""
    # ajoute h pour ajuster le i-ème élément de v
    w = [v_j + (h if j == i else 0) # ajoute h pour ajuster le i-ème élément de v
          for j, v_j in enumerate(v)]
    return (f(w) - f(v)) / h
```


Cela permet ensuite d'estimer le gradient de la même façon :

```
def estimate_gradient(f: Callable[[Vector], float],
                     v: Vector,
                     h: float = 0.0001):
    return [partial_difference_quotient(f, v, i, h)
            for i in range(len(v))]
```

Remarque

L'inconvénient majeur de cette méthode d'estimation par les quotients des différences est son coût de calcul élevé. Si v est de longueur n , `estimate_gradient` doit estimer f pour $2n$ entrées différentes. Si vous vous livrez à des estimations de gradients à répétition, vous travaillez de manière excessive. Quelle que soit la tâche, nous faisons appel aux mathématiques pour calculer nos fonctions de gradient explicitement.

L'utilisation du gradient

Il est facile de vérifier que la fonction `sum_of_squares` est de valeur minimale quand son entrée v est un vecteur de zéros, mais imaginons que vous l'ignorez. Utilisons le gradient pour trouver le minimum dans le cas de vecteurs à trois dimensions. Nous prendrons un point de départ au hasard et nous effectuerons de très petits déplacements dans la direction opposée au gradient, jusqu'à ce que nous atteignons un point où le gradient est très faible :

```
import random
from scratch.linear_algebra import distance, add, scalar_multiply

def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
    """se déplacer de step_size dans la direction de gradient depuis v"""
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)

def sum_of_squares_gradient(v: Vector) -> Vector:
    return [2 * v_i for v_i in v]

# choisir un point de départ aléatoire
v = [random.uniform(-10, 10) for i in range(3)]
for epoch in range(1000):
    grad = sum_of_squares_gradient(v) # calculer le gradient en v
    v = gradient_step(v, grad, -0.01) # effectuer un pas de gradient négatif
    print(epoch, v)

assert distance(v, [0, 0, 0]) < 0.001 # v devrait être proche de 0
```

Exécutez ce code et vous verrez qu'il s'arrête toujours avec une valeur v très proche de $[0, 0, 0]$.

Plus vous imposez une tolérance faible, plus le programme se rapproche de cette valeur.

Choisir le bon pas

On comprend facilement le pourquoi de ce mouvement à l'opposé du gradient ; toutefois, la valeur à donner à ce déplacement est moins évidente. En fait, choisir le pas de déplacement relève davantage de l'art que de la science. Parmi les options favorites, citons :

- choisir un pas de déplacement fixe ;
- réduire progressivement la taille du pas au cours du temps ;
- ou, à chaque étape, choisir le pas qui minimise la valeur de la fonction objectif.

Cette dernière option semble parfaite, mais en pratique, elle représente des calculs coûteux. Pour faire simple, nous nous contenterons d'utiliser essentiellement une taille de pas fixe. Celle « qui marche » dépend du problème : trop petite et votre descente de gradient sera interminable, trop grande et vous ferez des pas de géant qui rendront probablement la fonction qui vous occupe plus grande, voir indéfinie. Vous devez donc procéder par expérimentation.

S'adapter à des modèles

Dans cet ouvrage, nous utiliserons une descente de gradient pour adapter des modèles paramétrés à des données. Dans le cas habituel, nous disposons d'un ensemble de données et d'un modèle (hypothétique). Ce dernier dépend (de manière différentiable) d'un ou plusieurs paramètres. Nous disposons en outre d'une fonction de perte qui mesure la qualité d'adaptation du modèle à nos données (le plus petit sera le mieux).

Si nous considérons nos données comme fixes, alors notre fonction de perte nous dira dans quelle mesure les paramètres d'un modèle particulier sont bons ou mauvais. En d'autres termes, nous pouvons utiliser une descente de gradient pour trouver des paramètres de modèle qui atténuent la perte autant que faire se peut. Considérons un exemple simple :

```
# x va de -50 à 49 ; y vaut toujours 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
```

Dans ce cas, nous connaissons les paramètres de la relation linéaire entre x et y . Imaginons à présent que nous souhaitions les identifier à partir des données. Nous utilisons alors une descente de gradient pour identifier la pente et l'ordonnée à l'origine qui minimisent l'erreur quadratique moyenne.

Nous commencerons avec une fonction qui détermine le gradient selon l'erreur issue d'un seul point de données :

```
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta
    predicted = slope * x + intercept # prévision du modèle
```

```

error = (predicted - y) # l'erreur est (prévue - réelle)
squared_error = error ** 2 # nous minimisons l'erreur quadratique
grad = [2 * error * x, 2 * error] # en utilisant son gradient
return grad

```

Réfléchissons à la signification du gradient. Imaginons que, pour une valeur x , notre prévision soit trop grande. Dans ce cas, l'erreur est positive. Le deuxième terme du gradient $-2 * \text{error}$ est positif, ce qui reflète le fait que de petites hausses de y_0 rendront la prévision (déjà trop grande) encore plus grande, ce qui aura pour conséquence de rendre l'erreur quadratique (pour ce x) encore plus élevée.

Le premier terme du gradient $-2 * \text{error} * x$ présente le même signe que x . Il va sans dire que si x est positif, de petites hausses sur la pente rendront à nouveau la prévision (et donc l'erreur) plus importante. Si x est négatif, en revanche, ces petites hausses rendront à nouveau la prévision (et donc l'erreur) plus petite.

Ce calcul concernait toutefois un seul point de données. Pour un ensemble complet de données, nous examinerons l'erreur quadratique moyenne. Et le gradient de l'erreur quadratique moyenne est simplement la moyenne des gradients pris individuellement.

Voici comment nous allons procéder.

- 1 Commencer avec une valeur aléatoire pour θ .
- 2 Calculer la moyenne des gradients.
- 3 Ajuster θ dans cette direction.
- 4 Répéter.

Après un grand nombre d'époques (d'étapes), qui désignent les différentes passes sur l'ensemble de données), nous devrions obtenir quelques éléments d'information ressemblant aux paramètres appropriés :

```

from scratch.linear_algebra import vector_mean
# commencer avec des valeurs aléatoires pour la pente et y0
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
learning_rate = 0.001
for epoch in range(5000):
    # calculer la moyenne des gradients
    grad = moyenne_vecteurs([gradient_linéaire(x, y, theta)
    for x, y in inputs])
    # faire un pas dans cette direction
    theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "la pente devrait être d'environ 20"
assert 4.9 < intercept < 5.1, "l'interception devrait être d'environ 5"

```

Mini-lot et descente du gradient stochastique

L'inconvénient avec la précédente approche tient à ce qu'il faut évaluer les gradients sur tout l'ensemble des données avant de pouvoir adopter un pas de gradient et d'actualiser nos paramètres. Dans le cas présent, tout allait bien car notre ensemble de données n'était constitué que de 100 paires. Le calcul du gradient réclamait donc peu de ressources.

En revanche, vos (vrais) modèles présenteront souvent des ensembles de données de grande envergure et des calculs de gradient plus gourmands en ressources. Dans de tels cas, vous devrez plus souvent faire appel à différents pas de gradient.

Pour ce faire, nous utilisons une technique appelée *descente de gradient par mini-lots*. Dans celle-ci, nous calculons le gradient (et adoptons un pas de gradient) en nous fondant sur un « mini-lot » échantillonné à partir d'un ensemble de données plus volumineux :

```
from typing import TypeVar, List, Iterator
T = TypeVar('T') # ceci nous permet de typer des fonctions " génériques "
def minibatches(dataset: List[T],
                batch_size: int,
                shuffle: bool = True) -> Iterator[List[T]]:
    """génère des mini-lots à partir de l'ensemble de données"""
    # débiter des index 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in range(0, len(dataset), batch_size)]
    if shuffle: random.shuffle(batch_starts) # mélange les lots
    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

Remarque

`TypeVar` nous permet de créer une fonction « générique ». Ce code explique que notre ensemble de données peut être une liste de n'importe quel type : chaînes, entiers, listes ou autre. En revanche, quel que soit le type, les résultats constitueront des lots de celui-ci.

Nous pouvons à présent résoudre notre problème en utilisant des mini-lots :

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta)
                             for x, y in batch])
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "la pente devrait être d'environ 20"
assert 4.9 < intercept < 5.1, "l'interception devrait être d'environ 5"
```

Dans une variante appelée la descente du gradient stochastique, vous pouvez adopter des pas de gradient en fonction d'un exemple d'entraînement à la fois :

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "la pente devrait être d'environ 20"
assert 4.9 < intercept < 5.1, "l'interception devrait être d'environ 5"
```

Pour ce problème, la descente du gradient stochastique identifie les paramètres optimaux dans un nombre beaucoup plus restreint d'époques. Néanmoins, la médaille a toujours son revers. S'appuyer sur des pas de gradients appliqués à des mini-lots (ou à des points de données uniques) vous permet d'en obtenir un plus grand nombre, mais le gradient d'un point unique peut s'inscrire dans une direction très différente de celui de tout l'ensemble des données.

En outre, dans le cas où nous n'utiliserions pas l'algèbre linéaire en partant de zéro, nous gagnerions probablement en performance à « vectoriser » nos calculs à l'échelle de lots, plutôt qu'à calculer le gradient d'un point à la fois.

Au fil du présent ouvrage, nous nous amuserons à déterminer des tailles de lots et de pas optimaux.

Remarque

La terminologie des différentes déclinaisons de la descente de gradient n'est pas uniforme. L'approche consistant à « calculer le gradient pour tout l'ensemble des données » est souvent appelée « descente de gradient en lots ». Certains emploieront « descente de gradient stochastique » pour désigner une version en mini-lots (dont la version en « un point à la fois » constitue un cas particulier).

Pour aller plus loin

- Poursuivez votre lecture ! Nous utiliserons la descente de gradient pour résoudre des problèmes tout au long de cet ouvrage.
- À ce stade, je suppose que vous êtes malade à l'idée que je vous recommande encore de lire des manuels. Si cela peut vous consoler, *Active Calculus 1.0*¹, de Matthew Boelkins, David Austin et Steven Schlicker (Grand Valley State University Libraries – en anglais), semble plus agréable que les manuels que j'ai utilisés pour mon apprentissage.
- Sebastian Ruder a publié sur son blog² un article sensationnel qui compare les nombreuses variantes de la descente de gradient.

1. <https://scholarworks.gvsu.edu/books/10/>

2. <https://ruder.io/optimizing-gradient-descent/index.html>

9

Collecte des données

*Il m'a fallu trois mois pour l'écrire, trois minutes pour le concevoir
et toute ma vie pour collecter les données.*

F. Scott Fitzgerald

Un expert en science des données a besoin de données. Il faut bien le reconnaître, vous allez passer l'essentiel de votre temps à acquérir, nettoyer et transformer des données. En un mot, vous pouvez toujours taper les données vous-même (ou confier cette tâche à vos assistants dévoués si vous en avez) mais, en général, ce n'est pas un usage efficace de votre temps. Dans ce chapitre, nous allons examiner différentes manières d'introduire les données dans Python avec les bons formats.

stdin et stdout

Si vous exécutez vos scripts Python en ligne de commande, vous pouvez gérer vos données à l'aide de `sys.stdin` et `sys.stdout`. Par exemple, voici un script qui lit des lignes de texte et restitue celles qui correspondent à une expression rationnelle :

```
# egrep.py
import sys, re
# sys.argv est la liste des arguments de ligne de commande
# sys.argv[0] est le nom du programme
# sys.argv[1] sera la regex spécifiée pour la ligne de commande
regex = sys.argv[1]
```

```
# pour chaque ligne passée dans le script
for line in sys.stdin:
    # si elle correspond à la regex, l'écrire dans stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

Et en voici un autre qui compte les lignes reçues et écrit le résultat :

```
# line_count.py
import sys
count = 0
for line in sys.stdin:
    count += 1

# l'impression est redirigée vers sys.stdout
print(count)
```

Vous pouvez alors utiliser ces scripts pour compter, dans un fichier, le nombre des lignes qui contiennent des nombres. Sous Windows, vous écrirez ce qui suit :

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

Et sous un système Unix, ce sera :

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

La barre verticale est le caractère *pipe*, qui signifie « utiliser le résultat de la commande de gauche comme entrée de celle de droite ». Ce caractère permet de construire des pipelines (des enchaînements) de traitement de données assez élaborés.

Remarque

Sous Windows, vous laisserez probablement de côté la partie Python de la commande :

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

Sous Unix, il faudra sans doute quelques étapes supplémentaires. D'abord vous placerez le « fourbi » initial sur la première ligne de votre script `#!/usr/bin/env python`. Ensuite, sur la ligne de commande, vous utiliserez `chmod x egrep.py++` pour rendre le fichier exécutable.

De même, voici un script pour compter les mots en entrée et afficher les plus courants en sortie :

```
# most_common_words.py
import sys
from collections import Counter
```



```
# passer le nombre de mots en premier argument
try:
    num_words = int(sys.argv[1])
except:
    print("usage: most_common_words.py num_words")
    sys.exit(1) # un code non zéro en sortie indique une erreur

counter = Counter(word.lower() # mots en minuscules
                  for line in sys.stdin
                  # séparer à partir des espaces
                  for word in line.strip().split()
                  if word) # sauter les mots (words) vides

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

Après exécution, vous vous retrouverez avec quelque chose comme ce qui suit (si vous utilisez Windows, utilisez `type` au lieu de `cat`) :

```
$ cat the_bible.txt | python most_common_words.py 10
36397 the
30031 and
20163 of
7154 to
6484 in
5856 that
5421 he
5226 his
5060 unto
4297 shall
```

Remarque

Si vous êtes un programmeur Unix expérimenté, vous connaissez certainement un grand nombre d'outils en ligne de commande (comme `egrep`) présents nativement dans votre système d'exploitation et qu'il convient d'utiliser, plutôt que de tout réinventer.

Lecture de fichiers

Il est possible de lire et d'écrire des fichiers nommés explicitement directement dans le code. Python facilite le travail avec des fichiers.

Fondamentaux des fichiers de type texte

La première étape consiste à obtenir un objet permettant de traiter le fichier à l'aide de `open` :

```
# 'r' (read-only) signifie lecture seule. Valeur par défaut.
file_for_reading = open('reading_file.txt', 'r')
file_for_reading2 = open('reading_file.txt')
# 'w' (write) signifie écrire. Détruit le fichier s'il existait déjà !
file_for_writing = open('writing_file.txt', 'w')
# 'a' (append) signifie ajouter à la fin du fichier existant
file_for_appending = open('appending_file.txt', 'a')
# n'oubliez pas de fermer les fichiers quand vous avez fini
file_for_writing.close()
```

Comme il est facile d'oublier de fermer un fichier, il est recommandé d'utiliser ces instructions dans un bloc `with`, à la fin duquel la fermeture sera automatique :

```
with open(filename) as f:
    data = function_that_gets_data_from(f)

# à ce stade, f a déjà été fermé ; n'essayez pas de l'utiliser
process(data)
```

Si vous voulez lire un fichier en entier, vous pouvez vous contenter d'itérations sur ses lignes dans une boucle `for` :

```
starts_with_hash = 0
with open('input.txt') as f:
    for line in f: # examine chaque ligne du fichier
        # utilise une regex pour voir si la ligne commence par '#'
        if re.match("^#", line):
            starts_with_hash += 1 # si oui, ajoute 1 au compteur
```

Chaque ligne retournée se termine par un caractère de fin de ligne (*newline*) que vous voudrez souvent supprimer avec `strip()` avant d'utiliser la ligne.

Par exemple, imaginons que vous ayez un fichier d'adresses de messagerie, une par ligne, et que vous vouliez générer un histogramme des domaines. Pour extraire correctement ces derniers, il existe des règles un peu subtiles (par exemple, la *Public Suffix List*¹) mais, en première approximation, vous pouvez simplement prendre la partie de l'adresse électronique qui se situe après le caractère `@` (ce qui donnera un mauvais résultat pour les adresses du type `joel@mail.datasciencester.com`, mais restera suffisant pour notre exemple).

1. <https://publicsuffix.org/>

```
def get_domain(email_address: str) -> str:
    """couper au niveau de '@' et retourner le dernier morceau"""
    return email_address.lower().split("@")[-1]

# deux tests
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencecenter.com') == 'm.datasciencecenter.com'
from collections import Counter
with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                             for line in f
                             if "@" in line)
```

Fichiers à délimiteur

L'hypothétique fichier d'adresses que nous venons de traiter comportait une adresse par ligne. La plupart du temps toutefois, nos fichiers comportent de nombreuses données sur chaque ligne et utilisent alors très souvent la virgule ou le caractère de tabulation comme *séparateur* entre les *champs*.

Les choses se compliquent quand certains champs contiennent eux-mêmes des virgules et des caractères de début de ligne (ce qui est inévitable). C'est pour cela que vouloir analyser la syntaxe du contenu soi-même constitue presque toujours une mauvaise idée. À la place, utilisez plutôt le module `csv` de Python (ou la bibliothèque `pandas`, voire d'autres bibliothèques conçues pour lire les fichiers à délimiteurs).

Avertissement

N'analysez jamais vous-même un fichier délimité par des virgules. Vous corrompez les cas limites !

Si votre fichier n'a pas d'en-tête (ce qui veut dire que vous voulez que chaque ligne soit une liste et impose de savoir ce que renferme chaque colonne), invoquez `csv.reader` pour parcourir les lignes, chacune d'entre elles devenant alors une liste épluchée. Par exemple, supposons un fichier répertoriant des cours d'actions, délimités par des tabulations :

```
6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB 64.5
6/19/2014 AAPL 91.86
6/19/2014 MSFT 41.51
6/19/2014 FB 64.34
```

Nous pouvons le gérer comme suit :

```
import csv
with open('tab_delimited_stock_prices.txt') as f:
    tab_reader = csv.reader(f, delimiter='\t')
    for row in tab_reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        # traitement des données...
```

Votre fichier aura souvent des en-têtes :

```
date:symbol:closing_price
6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB 64.5
6/19/2014 AAPL 91.86
6/19/2014 MSFT 41.51
6/19/2014 FB 64.34
```

Dans ce cas, vous pouvez soit sauter la ligne d'en-tête (avec un appel initial à `reader.next()`), soit récupérer chaque ligne en tant que `dict` (dont les en-têtes sont les clés) à l'aide de `csv.DictReader` :

```
with open('colon_delimited_stock_prices.txt') as f:
    colon_reader = csv.DictReader(f, delimiter=':')
    for dict_row in colon_reader:
        date = dict_row["date"]
        symbol = dict_row["symbol"]
        closing_price = float(dict_row["closing_price"])
        # traitement des données...
```

Même si votre fichier n'a pas d'en-tête, vous pouvez utiliser `DictReader` en lui passant les clés comme paramètres de nom de champ.

De façon similaire, on écrit des données délimitées à l'aide de `csv.writer` :

```
todays_prices = {'AAPL': 90.91, 'MSFT': 41.68, 'FB': 64.5 }
with open('comma_delimited_stock_prices.txt', 'w') as f:
    csv_writer = csv.writer(f, delimiter=',')
    for stock, price in todays_prices.items():
        csv_writer.writerow([stock, price])
```

`csv.writer` fera ce qu'il faut si vos champs eux-mêmes contiennent des virgules. Ce ne sera sans doute pas le cas d'un module écrit à la main. Par exemple, si vous essayez le code suivant.

```
results = ["test1", "success", "Monday"],
          ["test2", "success, kind of", "Tuesday"],
          ["test3", "failure, kind of", "Wednesday"],
          ["test4", "failure, utter", "Thursday"]
# à ne pas faire !
with open('bad_csv.txt', 'w') as f:
    for row in results:
        f.write(",".join(map(str, row))) # peut-être trop de virgules
        f.write("\n") # la ligne peut contenir aussi des caractères newline
```

Vous obtiendrez un fichier dont personne ne pourra rien tirer :

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

Ratissage du Web

Une autre manière de récupérer des données consiste à les obtenir à partir de pages web. Récupérer des pages web est assez facile ; en extraire des informations significatives et structurées l'est beaucoup moins.

Analyse syntaxique du HTML

Sur le Web, les pages sont écrites en HTML. Le texte est (idéalement) marqué au sein d'éléments et d'attributs :

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

Dans un monde parfait, où toutes les pages seraient sémantiquement correctes pour notre plus grand bonheur, nous serions capables d'extraire des données en utilisant des règles comme « trouver l'élément `<p>` dont l'identifiant est "subject" et retourner le texte qu'il contient ». Dans le monde réel, le HTML est généralement mal mis en forme et encore moins annoté. Nous avons donc besoin d'aide pour lui donner un sens.

Pour extraire des données du code HTML, nous utiliserons la bibliothèque `Beautiful Soup`², qui construit une arborescence à partir des différents éléments d'une page web fournissant une interface simple pour y accéder. Au moment où j'écris ce livre, la dernière version est la 4.6.0 ; c'est celle que nous allons utiliser. Nous nous servirons aussi de la bibliothèque de récupération d'URL `Requests`³ qui, en termes de requêtes HTTP, offre des fonctionnalités plus conviviales que tous les outils natifs de Python.

Le *parseur* par défaut de Python n'est pas très permissif ; cela veut dire qu'il n'accepte pas toujours un code HTML qui n'est pas parfaitement réalisé. Pour cette raison, nous utiliserons un autre parseur, qu'il faudra alors installer :

```
pip install html5lib
```

Assurez-vous que vous vous trouvez dans l'environnement virtuel approprié, puis installez les bibliothèques :

```
python -m pip install beautifulsoup4 requests html5lib
```

Nous devons passer le code HTML à la « moulinette » de la fonction `BeautifulSoup()`. Dans nos exemples, ce sera le résultat d'un appel à `requests.get` :

```
from bs4 import BeautifulSoup
import requests
# Je place le fichier HTML qui nous occupe sur GitHub. Pour adapter
# l'URL au format de l'ouvrage, je dois la répartir sur deux lignes.
# Rappelez-vous que les chaînes séparées par des caractères d'espace
# sont concaténées.
url = ("https://raw.githubusercontent.com/"
      "joelgrus/data/master/getting-data.html")
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

Ensuite, il suffit d'utiliser quelques méthodes simples.

Nous ferons usage des objets `Tags` qui correspondent aux balises structurant la page HTML.

Par exemple, pour trouver la première balise `<p>` (et son contenu), vous pouvez écrire :

```
first_paragraph = soup.find('p') # ou tout simplement soup.p
```

Vous obtenez le texte associé à une balise à partir de la propriété `text` :

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

2. <https://www.crummy.com/software/BeautifulSoup/>

3. <https://2.python-requests.org/en/master/>

Et vous pouvez extraire les attributs d'une balise en la traitant comme un dict :

```
# lève l'erreur KeyError si 'id' n'existe pas
first_paragraph_id = soup.p['id'] # retourne None si 'id' n'existe pas
first_paragraph_id2 = soup.p.get('id')
```

Vous pouvez aussi récupérer plusieurs balises en une seule fois :

```
all_paragraphs = soup.find_all('p') # ou simplement soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Et souvent, vous chercherez des balises affichant une classe particulière :

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                        if 'important' in p.get('class', [])]
```

Enfin, vous pouvez combiner ces procédés simples pour élaborer des stratégies plus complexes. Par exemple, si vous cherchez tous les éléments `<span>` contenus dans un élément `<div>`, procédez comme suit :

```
# Attention, retourne le même <span> plusieurs fois de suite
# s'il est présent dans plusieurs <div>.
# soyez plus malin dans ce cas
spans_inside_divs = [span
                     # pour chaque <div> sur la page
                     for div in soup('div')
                     # trouver chaque <span> à l'intérieur
                     for span in div('span')]
```

Ces quelques fonctions nous seront très utiles. Si vous avez besoin de requêtes plus complexes, ou si vous êtes simplement curieux, consultez la documentation⁴.

Il va de soi que les données importantes ne seront certainement pas identifiées comme telles avec `class="important"`. Vous devrez donc inspecter le code source HTML avec soin, vérifier votre logique de sélection et vous intéresser aux cas limites pour vous assurer que vos données sont correctes. Voyons un exemple concret.

Exemple : Congrès sous surveillance

Le vice-président chargé des politiques chez DataSciencester s'inquiète d'une potentielle réglementation du secteur de la science des données. Il vous demande de quantifier ce que le Congrès a à dire sur le sujet. Il veut plus particulièrement que vous identifiiez tous les repré-

4. <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

sentants de cette institution des États-Unis qui sont associés à des communiqués de presse relatifs aux données.

Au moment où nous publions ces lignes, une page contient des liens vers tous les sites web des représentants : <https://www.house.gov/representatives>.

Et si vous consultez la source, vous constatez que tous les liens vers ces sites web se présentent selon l'exemple suivant.

```
<td>
  <a href="https://jayapal.house.gov">Jayapal, Pramila</a>
</td>
```

Commençons par collecter toutes les URL liées depuis cette page :

```
from bs4 import BeautifulSoup
import requests
url = "https://www.house.gov/representatives"
text = requests.get(url).text
soup = BeautifulSoup(text, "html5lib")
all_urls = [a['href']
             for a in soup('a')
             if a.has_attr('href')]
print(len(all_urls)) # 965 pour moi, c'est beaucoup trop
```

Ce code retourne un bien trop grand nombre d'URL. Si vous les examinez, celles qui nous intéressent commencent par `http://` ou `https://`, présentent une sorte de nom et se terminent par `.house.gov` ou `.house.gov/`.

C'est un excellent cas pour appliquer une expression rationnelle :

```
import re
# doit commencer par http:// ou https://
# doit s'achever par .house.gov ou .house.gov/
regex = r"^https?://.*\.house\.gov/?$"
# rédigeons quelques tests !
assert re.match(regex, "http://joel.house.gov")
assert re.match(regex, "https://joel.house.gov")
assert re.match(regex, "http://joel.house.gov/")
assert re.match(regex, "https://joel.house.gov/")
assert not re.match(regex, "joel.house.gov")
assert not re.match(regex, "http://joel.house.com")
assert not re.match(regex, "https://joel.house.gov/biography")
# et mettons en application
good_urls = [url for url in all_urls if re.match(regex, url)]
print(len(good_urls)) # encore 862 pour moi
```


C'est encore beaucoup trop, sachant qu'il n'y a que 435 représentants. Si vous examinez la liste, vous constatez qu'elle présente beaucoup de doublons. Voyons comment nous en débarrasser :

```
good_urls = list(set(good_urls))
print(len(good_urls)) # seulement 431 pour moi
```

Il reste encore quelques sièges vides ou peut-être qu'un des représentants n'a pas de site web. Quel que soit le cas, c'est insuffisant. Les sites présentent pour la plupart un lien vers des communiqués de presse. Par exemple :

```
html = requests.get('https://jayapal.house.gov').text
soup = BeautifulSoup(html, 'html5lib')
# utiliser un ensemble car les liens peuvent se présenter plusieurs fois
links = {a['href'] for a in soup('a') if 'press releases' in a.text.lower()}
print(links) # {'/media/press-releases'}
```

Notez que le lien est relatif: nous devons mémoriser le site d'origine. Allons-y pour le ratissage :

```
from typing import Dict, Set
press_releases: Dict[str, Set[str]] = {}
for house_url in good_urls:
    html = requests.get(house_url).text
    soup = BeautifulSoup(html, 'html5lib')
    pr_links = {a['href'] for a in soup('a') if 'press releases'
                                                         in a.text.lower()}

    print(f"{house_url}: {pr_links}")
    press_releases[house_url] = pr_links
```

Remarque

Normalement, il n'est pas courtois de ratisser un site librement de cette manière. Les sites disposent pour la plupart d'un fichier `robots.txt` qui indique la fréquence à laquelle vous pouvez procéder à un tel ratissage (et les chemins que vous êtes censé éviter). Toutefois, sachant qu'il s'agit du Congrès, nous n'avons pas besoin de nous montrer particulièrement polis.

En faisant défiler ces éléments, vous constatez la présence d'un grand nombre de `/media/press-releases` et de `mediacenter/press-releases`, ainsi que diverses autres adresses. Une de ces URL est `https://jayapal.house.gov/media/press-releases`.

Rappelez-vous que notre objectif consiste à identifier les représentants dont des communiqués de presse mentionnent le mot *data* (données). Nous allons rédiger une fonction légèrement plus générale qui détermine si une page de communications de presse mentionne n'importe quel terme donné.

Si vous visitez le site et consultez la source, vous constaterez qu'à chaque communiqué de presse semble correspondre un fragment placé à l'intérieur d'une balise `<p>` ; nous utiliserons donc cette caractéristique dans notre première tentative :

```
def paragraph_mentions(text: str, keyword: str) -> bool:
    """retourne True si un <p> au sein du texte mentionne le mot-clé {keyword}"""
    soup = BeautifulSoup(text, 'html5lib')
    paragraphs = [p.get_text() for p in soup('p')]
    return any(keyword.lower() in paragraph.lower()
               for paragraph in paragraphs)
```

Rédigeons un test rapide pour ce code :

```
text = """<body><h1>Facebook</h1><p>Twitter</p>"""
# "twitter" est à l'intérieur d'un <p>
assert paragraph_mentions(text, "twitter")
# "facebook" est à l'extérieur d'un <p>
assert not paragraph_mentions(text, "facebook") # est à l'extérieur d'un <p>
```

Enfin, nous voilà prêts à identifier les représentants du Congrès qui nous intéressent et à en donner les noms à notre vice-président :

```
for house_url, pr_links in press_releases.items():
    for pr_link in pr_links:
        url = f"{house_url}/{pr_link}"
        text = requests.get(url).text
        if paragraph_mentions(text, 'data'):
            print(f"{house_url}")
            break # fini avec cette house_url
```

Lorsque j'exécute ce code, j'obtiens une liste d'environ 20 représentants. Vos résultats seront probablement différents.

Remarque

Si vous examinez les différentes pages de communication de presse (*press releases*), vous constaterez qu'elles sont, pour la plupart, paginées avec seulement 5 ou 10 communiqués chacune. En d'autres termes, nous n'avons extrait que les communiqués les plus récents pour chaque représentant. Une solution plus exhaustive consisterait à appliquer une itération aux pages et ainsi à récupérer le texte intégral de chaque communiqué.

Utilisation des API

De nombreux sites et services en ligne proposent des *interfaces de programmation d'application* (*Application Programming Interface – API*). Celles-ci vous permettent de passer des

requêtes explicites sur des données dans un format structuré. Il devient ainsi inutile d'aspirer l'intégralité du site web !

JSON et XML

Comme HTTP est un protocole de transfert de texte, les données que vous avez requises à l'aide d'une API web doivent être *sérialisées* dans un format de type chaîne de caractères. Souvent, la *sérialisation* se fait à l'aide de JSON (*JavaScript Object Notation*). Les objets JavaScript ressemblent beaucoup à des `dict` Python, ce qui facilite l'interprétation des chaînes de caractères :

```
{ "title" : "Data Science Book",  
  "author" : "Joel Grus",  
  "publicationYear" : 2019,  
  "topics" : [ "data", "science", "data science" ] }
```

Nous pouvons analyser JSON avec le module `json` de Python. En particulier, nous ferons appel à la fonction `loads` qui *déséréalise* une chaîne de caractères représentant un objet JSON en objet Python :

```
import json  
serialized = """{ "title" : "Data Science Book",  
  "author" : "Joel Grus",  
  "publicationYear" : 2019,  
  "topics" : [ "data", "science", "data science" ] }"""  
# analyse le JSON pour créer un dict Python  
deserialized = json.loads(serialized)  
assert deserialized["publicationYear"] == 2019  
assert "data science" in deserialized["topics"]
```

Parfois, le fournisseur d'API ne vous facilite pas la tâche et fournit une réponse au format XML :

```
<Book>  
  <Title>Data Science Book</Title>  
  <Author>Joel Grus</Author>  
  <PublicationYear>2014</PublicationYear>  
  <Topics>  
    <Topic>data</Topic>  
    <Topic>science</Topic>  
    <Topic>data science</Topic>  
  </Topics>  
</Book>
```

Vous pouvez utiliser `Beautiful Soup` pour récupérer les données à partir du XML. Pour en savoir plus, reportez-vous à la documentation.

Utiliser une API non authentifiée

Pour la plupart, les API exigent une authentification avant de pouvoir être utilisées. Nous ne contestons pas ce mode de fonctionnement, mais il génère des normes supplémentaires qui compliquent notre explication. Nous allons donc nous intéresser à l'API de GitHub⁵, qui permet de faire des choses simples sans authentification :

```
import requests, json
github_user = "joelgrus"
endpoint = f"https://api.github.com/users/{github_user}/repos"
repos = json.loads(requests.get(endpoint).text)
```

À ce stade, `repos` est une liste de `dict` Python dont chacun représente un répertoire de mon compte GitHub.

Utilisons cette liste pour déterminer quels mois et quels jours de la semaine j'ai le plus de chances de créer un répertoire. Le seul problème tient à ce que les dates dans la réponse sont des chaînes de caractères :

```
"created_at": "2013-07-05T02:02:28Z"
```

Comme, par défaut, Python ne fournit aucun analyseur de dates sérieux, il faut en installer un :

```
python -m pip install python-dateutil
```

Seule la fonction `dateutil.parser.parse` vous sera sans doute réellement utile :

```
from collections import Counter
from dateutil.parser import parse
dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

De même, nous pouvons trouver les langages de mes cinq derniers référentiels :

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["pushed_at"],
                             reverse=True)[:5]
last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

Normalement, nous ne travaillerons pas avec des API d'un degré d'abstraction si bas qu'il impose d'élaborer les requêtes et d'analyser les réponses nous-mêmes. Un des avantages de Python tient à ce que, pour la plupart des API qui vous intéressent, quelqu'un a déjà déve-

5. <https://developer.github.com/v3/>

loppé une bibliothèque de modules d'accès. Quand elles sont bien écrites, ces bibliothèques vous épargnent bien des erreurs au moment de mettre au point les détails d'accès. À l'inverse, si elles n'ont pas été bien écrites, ou si elles sont basées sur des versions obsolètes des API concernées, elles vous causeront bien des maux de tête.

Dans tous les cas, comme vous aurez besoin à l'occasion de lancer votre propre bibliothèque d'API (ou plus vraisemblablement de trouver pourquoi celle de quelqu'un d'autre ne marche pas), il est intéressant de connaître quelques détails.

Trouver des API

Si vous avez besoin de données d'un site en particulier, recherchez une section dudit site réservée aux développeurs ou aux API et lancez une recherche plus générale sur le Web avec la requête « API Python *nom_du_site* » pour trouver une bibliothèque.

Il existe pour Python des API pour Yelp, Instagram, Spotify et beaucoup d'autres.

Si vous cherchez une liste des API qui ont des encapsulateurs (*wrappers*) Python associés, vous en trouverez une intéressante chez Real Python sur GitHub⁶.

Et si vous n'avez toujours rien trouvé qui convienne à vos besoins, il reste l'extraction de site Web ; c'est l'ultime recours du data scientist.

Exemple : utilisation de l'API Twitter

Twitter constitue une formidable source de données à exploiter. Obtenir des actualités en temps réel, mesurer des réactions à des événements, trouver des liens relatifs à un sujet donné, vous pouvez l'utiliser pour beaucoup de choses du moment que vous arrivez à accéder aux données. Et pour cela, il faut se servir de son API.

Pour interagir avec les API de Twitter, vous devrez installer la bibliothèque `twython`⁷ (`python -m pip install twython`). Il existe quelques autres bibliothèques Python, mais c'est sans aucun doute celle qui m'a été la plus utile. N'hésitez pas à explorer aussi les autres !

L'accréditation

Pour avoir le droit d'utiliser des API Twitter, vous devez vous faire « accréditer » et, pour cela, il vous faut un compte Twitter, ce qui est de toute façon une bonne idée si vous voulez intégrer la communauté active et bienveillante des experts `#datascience` de Twitter.

6. <https://github.com/realpython/list-of-python-api-wrappers>

7. <https://github.com/ryanmcgrath/twython>

Avertissement

Comme avec toutes les consignes concernant des sites que je ne contrôle pas, il se peut que les lignes qui suivent ne soient plus entièrement d'actualité, mais j'espère qu'elles le resteront encore un peu. Elles ont pourtant déjà changé plusieurs fois depuis que j'ai écrit ce livre. Alors... bonne chance !

Voici la marche à suivre :

- 1 Ouvrez <https://developer.twitter.com/>.
- 2 Si ce n'est déjà fait, connectez-vous à Twitter avec votre nom d'utilisateur et votre mot de passe.
- 3 Cliquez sur le bouton adéquat pour postuler et solliciter un compte de développeur.
- 4 Demandez un accès au titre d'un usage personnel.
- 5 Remplissez le formulaire. Celui-ci vous invite à rédiger 300 mots (ils doivent y être !) sur votre motivation à obtenir un accès. Pour surmonter cet « obstacle », parlez donc du présent ouvrage et du plaisir qu'il vous apporte.
- 6 Patientez un certain temps (indéfini).
- 7 Si vous connaissez quelqu'un qui travaille chez Twitter, envoyez-lui un courrier électronique pour lui demander s'il peut accélérer le traitement de votre candidature. Sinon... patientez encore.
- 8 Une fois l'approbation obtenue, revenez sur developer.twitter.com, identifiez la section *Apps*, puis cliquez sur *Créer nouvelle app*.
- 9 Remplissez tous les champs requis (ici encore, si vous manquez de texte pour la description, parlez du livre et de son caractère édifiant).
- 10 Cliquez sur *Créer*.

À présent, votre app doit présenter un onglet dédié aux jetons et aux clés, avec une section pour clés d'API consommateur. Cette section répertorie une clé d'API et une clé secrète d'API. Notez-les ; vous en aurez besoin. En outre, ne les divulguez pas ! Considérez-les comme des mots de passe.

Avertissement

Vous ne devez ni partager les clés, ni les publier, ni les stocker dans votre référentiel public GitHub. Une solution simple consiste à les stocker dans un fichier `credentials.json` qui n'est pas contrôlé, pour les récupérer dans votre code au moyen de `json.loads`. Une autre consiste à les stocker dans des variables d'environnement et à utiliser `os.environ` pour les récupérer.

Utilisation de Twython

Lorsqu'il s'agit d'utiliser l'API Twitter, la partie la plus délicate consiste à vous authentifier. C'est d'ailleurs la partie la plus délicate pour beaucoup d'API. Leurs fournisseurs veulent la garantie que vous avez l'autorisation d'accéder à leurs données et que vous ne dépassez pas leurs limites d'usage. Ils veulent également savoir qui accède à leurs données.

L'authentification devient alors pénible. Un procédé simple, OAuth 2, suffit pour des recherches simples. Un autre, plus complexe, OAuth 1, est obligatoire lorsque vous souhaitez réaliser des actions (par exemple, envoyer des tweets) ou vous connecter au flux Twitter (c'est notre cas).

Nous sommes donc contraints de passer par le procédé compliqué, que nous allons tenter d'automatiser autant que faire se peut.

Tout d'abord, vous aurez besoin de votre clé d'API et de votre clé secrète (parfois connues respectivement sous les noms de clé consommateur et de code secret consommateur). Je récupère les miennes à partir de variables d'environnement, mais n'hésitez pas mettre les vôtres à la place si vous le souhaitez :

```
import os
# n'hésitez pas à connecter votre clé et votre code secret directement
CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY")
CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET")
```

Nous pouvons à présent instancier le client :

```
import webbrowser
from twython import Twython

# obtenir un client temporaire pour extraire une URL d'authentification
temp_client = Twython(CONSUMER_KEY, CONSUMER_SECRET)
temp_creds = temp_client.get_authentication_tokens()
url = temp_creds['auth_url']

# visiter cette URL pour autoriser l'application et obtenir un code PIN
print(f"go visit {url} and get the PIN code and paste it below")
webbrowser.open(url)
PIN_CODE = input("please enter the PIN code: ")

# utiliser le code PIN_CODE pour obtenir les jetons réels
auth_client = Twython(CONSUMER_KEY,
                      CONSUMER_SECRET,
                      temp_creds['oauth_token'],
                      temp_creds['oauth_token_secret'])
final_step = auth_client.get_authorized_tokens(PIN_CODE)
ACCESS_TOKEN = final_step['oauth_token']
ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']

# et les utiliser avec une nouvelle instance de Twython
twitter = Twython(CONSUMER_KEY,
                  CONSUMER_SECRET,
                  ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET)
```

Conseil

À ce stade, il serait intéressant d'enregistrer `ACCESS_TOKEN` et `ACCESS_TOKEN_SECRET` en lieu sûr. Ainsi, la prochaine fois, vous n'aurez pas à recommencer toute la petite plaisanterie.

Dès lors que vous disposez d'une instance de Twython authentifiée, vous pouvez lancer des recherches :

```
# recherche de tweets contenant la phrase "data science"
for status in twitter.search(q="data science")["statuses"]:
    user = status["user"]["screen_name"]
    text = status["text"]
    print(f"{user}: {text}\n")
```

Si vous exécutez ce code, vous devriez récupérer en retour des tweets tels que les suivants.

```
haithemnyc: Data scientists with the technical savvy & analytical chops to derive
meaning from big data are in demand. http://t.co/HsF9Q0dShP
```

```
RPubsRecent: Data Science http://t.co/6hCHUZ2PHM
```

```
spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization. So easy and Awesome.
```

Tout ceci n'est pas très intéressant, avant tout parce que l'API `Twitter Search` ne vous présente que les résultats récents qu'elle veut bien vous montrer. En data science, on veut le plus souvent beaucoup de tweets. C'est là que l'API `Streaming`⁸ est utile. Elle vous permet de vous connecter à (un échantillon) de l'intégralité de Twitter. Pour l'utiliser, vous devez vous authentifier à l'aide de votre jeton d'accès.

Afin d'avoir accès à l'API `Streaming` avec Twython, il faut définir une classe qui hérite de `TwythonStreamer` et qui écrase sa méthode `on_success` (et éventuellement sa méthode `on_error`) :

```
from twython import TwythonStreamer
# Ajouter des données à une variable globale n'est pas une très bonne
# pratique, mais cela rend l'exemple beaucoup plus facile.
tweets = []
class MyStreamer(TwythonStreamer):
    def on_success(self, data):
        """Que faire quand Twitter nous envoie des données ?
        Ici, les données seront un dict Python représentant un tweet."""
        # pour se limiter à la collecte de tweets en anglais
        if data.get('lang') == 'en':
            tweets.append(data)
            print(f"received tweet #{len(tweets)}")

        # arrêter quand il y a assez de tweets
        if len(tweets) >= 100:
            self.disconnect()

    def on_error(self, status_code, data):
        print(status_code, data)
        self.disconnect()
```

8. <https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data>

`MyStreamer` va se connecter au flux Twitter et attendre de recevoir des données. Chaque fois qu'il en reçoit (ici, un tweet est représenté comme un objet Python), il les passe à la méthode `on_success`, qui les ajoute à notre liste de tweets si la langue est l'anglais, puis déconnecte le *streamer* après en avoir collecté 100. Il ne reste plus qu'à l'initialiser et à le démarrer :

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
# commence à consommer les statuts publics qui contiennent le mot 'data'
stream.statuses.filter(track='data')
# Si, à la place, nous voulons commencer à consommer un échantillon de
# *tous* les statuts : stream.statuses.sample().
```

L'exécution se poursuit jusqu'à avoir collecté 100 tweets (ou jusqu'à la rencontre d'une erreur). Ensuite, c'est l'arrêt qui vous permet d'analyser ces tweets. Par exemple, vous pouvez trouver les hashtags les plus répandus :

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet['entities']["hashtags"])
print(top_hashtags.most_common(5))
```

Chaque tweet contient de nombreuses données. Vous pouvez tâtonner vous-même ou vous plonger dans la documentation de l'API Twitter⁹.

Remarque

En général, dans le cadre d'un projet réel, on ne se contente pas d'une liste en mémoire pour stocker les tweets. On les sauvegarde plutôt dans un fichier ou une base de données, pour en disposer en permanence.

Pour aller plus loin

- `pandas`¹⁰ est la bibliothèque de référence pour travailler avec des données (et les importer).
- `scrapy`¹¹ est une bibliothèque très riche pour construire des *scrapers* du Web plus complexes qui sont capables, par exemple, de suivre des liens inconnus.
- `kaggle`¹² héberge une collection volumineuse d'ensembles de données.

9. <https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object>

10. <https://pandas.pydata.org/>

11. <https://scrapy.org/>

12. <https://www.kaggle.com/datasets>

10

Travail sur les données

Souvent, les experts ont davantage d'informations que de jugeote.

Colin Powell

Le travail sur les données tient autant de l'art que de la science. Jusqu'ici, nous avons surtout parlé de science ; dans ce chapitre, nous allons aborder l'art.

Explorer les données

Après avoir identifié les questions auxquelles vous cherchez des réponses et mis la main sur quelques données, la tentation est grande d'entrer directement dans le vif du sujet et de construire des modèles pour en tirer des réponses. Vous devez résister à la tentation. En effet, votre première étape doit consister à explorer vos données.

Données à une dimension

Le cas le plus simple correspond à un ensemble de données à une seule dimension, à savoir, en fait, une collection de nombres. Par exemple, ce peut être la durée moyenne passée chaque jour sur le site, le nombre de fois qu'un tutoriel sur la data science a été vu parmi une collection de vidéos, ou encore le nombre de pages que compte chaque livre consacré à la data science dans votre bibliothèque.

La première étape consiste évidemment à réaliser quelques statistiques de synthèse. Vous voulez savoir de combien de données élémentaires (points de données) vous disposez, quelle est la plus petite, la plus grande, l'espérance et quel est l'écart-type.

Cependant, tout ceci ne vous aidera pas forcément à comprendre vos données. L'étape suivante passera donc avantageusement par la création d'un histogramme ; vous devrez répartir les données en tranches distinctes et compter combien figurent dans chaque tranche :

```
from typing import List, Dict
from collections import Counter
import math
import matplotlib.pyplot as plt

def bucketize(point: float, bucket_size: float) -> float:
    """dirige le point vers le prochain plus petit multiple de bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points: List[float], bucket_size: float) -> Dict[float, int]:
    """sépare les points et les compte dans chaque tranche"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points: List[float], bucket_size: float, title: str = ""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
```

Prenons par exemple les deux jeux de données suivants :

```
import random
from scratch.probability import inverse_normal_cdf
random.seed(0)
# distribution uniforme entre -100 et 100
uniform = [200 * random.random() - 100 for _ in range(10000)]
# distribution normale avec espérance 0 et écart-type 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

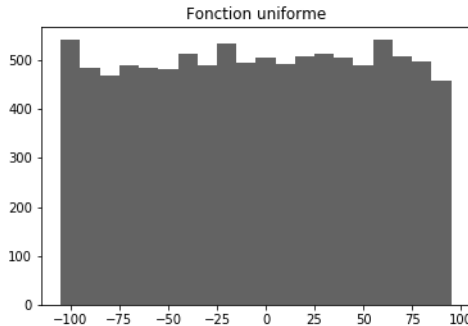
Dans les deux cas, l'espérance est proche de 0 et l'écart-type de 57. Pourtant, les deux distributions sont très différentes. Les figures 10–1 et 10–2 illustrent respectivement les distributions `uniform` et `normal` :

```
plot_histogram(uniform, 10, "Uniform Histogram")
(normal, 10, "Normal Histogram")
```

Dans ce cas, les deux distributions ont des valeurs minimales et maximales très différentes, mais même la connaissance de ce fait ne permet pas de comprendre en quoi elles diffèrent.

Figure 10–1

Histogramme de la distribution uniforme



Données à deux dimensions

Imaginons maintenant que votre jeu de données présente deux dimensions. Par exemple, en plus de la durée quotidienne en minutes de présence sur le site, vous connaissez le nombre d'années d'expérience par utilisateur. Bien sûr, vous voulez comprendre chaque dimension individuellement. Néanmoins, il est probable que vous vouliez surtout examiner la dispersion des données.

Voici par exemple un autre jeu de données fictif :

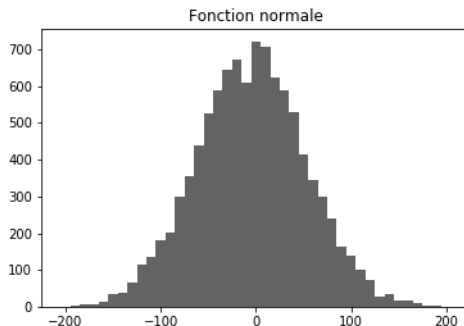
```
def random_normal() -> float:
    """retourne une distribution normale de paramètres aléatoires"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

Si vous deviez exécuter `plot_histogram` sur `ys1` et `ys2`, vous obtiendriez des graphiques très proches (en fait, les deux suivent une distribution normale de même espérance et de même écart-type).

Figure 10–2

Histogramme de loi normale

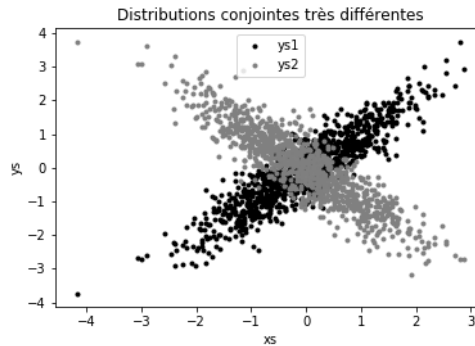


Pourtant, leurs distributions conjointes avec `xs` sont très différentes (figure 10–3) :

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Distributions conjointes très différentes")
plt.show()
```

Figure 10–3

Dispersion de deux `ys` différents



Cette différence est également très apparente si vous vous intéressez aux corrélations :

```
from scratch.statistics import correlation
print(correlation(xs, ys1)) # environ 0.9
print(correlation(xs, ys2)) # environ -0.9
```

Données à plusieurs dimensions

En présence de plusieurs dimensions, la question est de savoir comment celles-ci sont reliées entre elles. Une méthode simple consiste à examiner la matrice de corrélation dans laquelle la donnée à l'intersection de la ligne *i* et de la colonne *j* est la corrélation entre la *i*-ème et la *j*-ème dimension des données :

```
from scratch.linear_algebra import Matrix, Vector, make_matrix
def correlation_matrix(data: List[Vector]) -> Matrix:
    """Retourne la matrice carrée dont l'entrée de rang (i,j) est la
    corrélation entre les colonnes i et j des données."""
    def correlation_ij(i: int, j: int) -> float:
        return correlation(data[i], data[j])
    return make_matrix(len(data), len(data), correlation_ij)
```

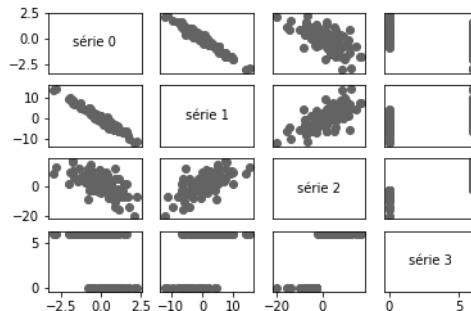
Pour une approche plus visuelle (si vous avez peu de dimensions), construisez une *matrice de nuages de points* (figure 10–4) pour illustrer les nuages correspondant aux paires de dimen-

sions. Pour ce faire, nous utilisons `plt.subplots` qui nous permet de créer des sous-fenêtres graphiques dans notre schéma. Nous fournissons le nombre de lignes et le nombre de colonnes et la fonction retourne un objet figure (que nous n'allons pas utiliser) et un tableau à deux dimensions des objets axes (que nous allons représenter) :

```
# corr_data est une liste de quatre vecteurs de dimension 100
num_vectors = len(corr_data)
fig, ax = plt.subplots(num_vectors, num_vectors)
for i in range(num_vectors):
    for j in range(num_vectors):
        # dispersion de column_j sur l'axe des x contre column_i sur l'axe des y
        if i != j: ax[i][j].scatter(corr_data[j], corr_data[i])
        # à moins que i == j, auquel cas montrer le nom de la série
        else: ax[i][j].annotate("série " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")
        # puis cache les labels des axes sauf à gauche et en bas
        if i < num_vectors - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)
# Corrige les étiquettes des axes en bas à droite et en haut à gauche,
# qui sont faux, car les diagrammes ne contiennent que du texte.
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())
plt.show()
```

Figure 10-4

Matrice de nuages de points



En examinant les nuages de points, vous pouvez voir que la série 1 est très négativement corrélée avec la série 0, que la série 2 est corrélée positivement avec la série 1 et que la série 3 ne prend que les valeurs 0 (correspondant aux petites valeurs de la série 2) et 6 (grandes valeurs).

C'est là un moyen rapide de comprendre grossièrement quelles variables sont corrélées (au lieu de passer des heures à paramétrer `matplotlib` pour les afficher exactement comme vous le souhaitez, ce qui n'est pas plus rapide).

Utiliser NamedTuple

Le recours aux `dict` constitue une manière courante de représenter des données :

```
import datetime
stock_price = {'closing_price': 102.06,
               'date': datetime.date(2014, 8, 29),
               'symbol': 'AAPL'}
```

Pourtant, cette solution est loin d'être idéale et ce pour plusieurs raisons. Cette représentation est quelque peu inefficace (un `dict` induit un certain coût). Ainsi, si le nombre de vos cours d'action est important, ceux-ci occuperont davantage de mémoire que nécessaire. Globalement, cet aspect a peu d'importance.

Plus important est l'accès aux éléments via une clé de `dict`, car cette technique favorise les erreurs. Le code suivant s'exécute sans erreur et pourtant il est incorrect.

```
# oups ! Erreur typographique
stock_price['cosing_price'] = 103.06
```

Enfin, si nous pouvons annoter les dictionnaires uniformes par type :

```
prices: Dict[datetime.date, float] = {}
```

Il n'existe aucun moyen secourable d'annoter des dictionnaires-en-tant-que-données qui présentent un grand nombre de types de valeur différents. Nous perdons ainsi la puissance des indications fournies via les types.

Comme solution de substitution, Python inclut une classe `namedtuple` qui se comporte comme un tuple mais avec des emplacements nommés :

```
from collections import namedtuple
StockPrice = namedtuple('StockPrice', ['symbol', 'date', 'closing_price'])
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)
assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
```

À l'instar du tuple standard, les tuples nommés sont constants ; vous ne pouvez donc pas modifier leurs valeurs une fois ceux-ci créés. Il se peut que nous soyons confrontés au « problème », mais globalement ce caractère non mutable est une bonne chose.

Remarquez que nous n'avons toujours pas résolu la question de l'annotation par type. Nous la résolvons en utilisant la variante typée, `NamedTuple` :

```
from typing import NamedTuple
class StockPrice(NamedTuple):
    symbol: str
```



```

date: datetime.date
closing_price: float

def is_high_tech(self) -> bool:
    """c'est une classe ; nous pouvons donc aussi ajouter des méthodes"""
    return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

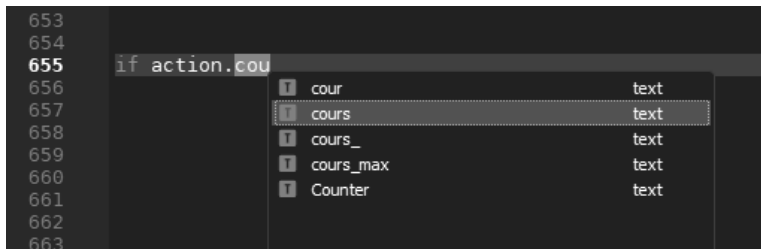
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()

```

Votre éditeur peut à présent vous aider (figure 10–5).

Figure 10–5
Un éditeur utile



Remarque

Peu de développeurs utilisent `NamedTuple` de cette manière. Et c'est bien dommage !

Dataclass

Les classes de données `dataclass` sont une sorte de version modifiable de `NamedTuple`. Je dis « une sorte » parce que les `NamedTuple` représentent leurs données de manière compacte, comme les tuples, tandis que les `dataclass` sont des classes Python normales qui génèrent automatiquement certaines méthodes pour vous.

Remarque

Les `dataclass` sont une nouveauté de Python 3.7. Si vous utilisez une version antérieure du langage, ce que cette section expose ne fonctionnera pas.

La syntaxe est très similaire à celle de `NamedTuple`. Toutefois, plutôt que d'hériter d'une classe de base, nous utilisons un décorateur :

```

from dataclasses import dataclass
@dataclass

```

```
class StockPrice2:
    symbol: str
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """c'est une classe ; nous pouvons donc aussi ajouter des méthodes"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price2 = StockPrice2('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price2.symbol == 'MSFT'
assert price2.closing_price == 106.03
assert price2.is_high_tech()
```

La grande différence tient à ce que nous pouvons modifier les valeurs d'une instance de `dataclass` :

```
# modification du cours
price2.closing_price /= 2
assert price2.closing_price == 51.03
```

Si nous avons essayé de modifier un champ de la version du `NamedTuple`, nous aurions levé une `AttributeError`.

En outre, cette approche nous expose au type d'erreurs que nous espérons éviter en renonçant à utiliser des `dict` :

```
# classe normale ; vous pouvez ajouter de nouveaux champs à l'envie !
price2.cosing_price = 75 # oups !
```

Nous n'utiliserons aucune `dataclass`, mais vous les rencontrerez probablement dans la pratique.

Nettoyer et transformer

Dans le monde réel, les données ne sont pas propres. Le plus souvent, un peu de travail est nécessaire avant de pouvoir les utiliser. Nous en avons vu des exemples au chapitre 9. Ainsi, il convient de convertir les chaînes de caractères en numériques à virgule flottante ou en entiers, avant de pouvoir les utiliser. Nous devons en outre nous enquérir de valeurs manquantes ou extrêmes ou de données erronées (non conformes).

C'est ce que nous avons fait jusqu'ici avant toute utilisation des données :

```
closing_price = float(row[2])
```

Toutefois, il y a probablement moins de risques d'erreurs à effectuer l'analyse dans le cadre d'une fonction que nous sommes en mesure de tester :

```
from dateutil.parser import parse
def parse_row(row: List[str]) -> StockPrice:
    symbol, date, closing_price = row
    return StockPrice(symbol=symbol,
                      date=parse(date).date(),
                      closing_price=float(closing_price))

# à présent, testons notre fonction
stock = parse_row(["MSFT", "2018-12-14", "106.03"])

assert stock.symbol == "MSFT"
assert stock.date == datetime.date(2018, 12, 14)
assert stock.closing_price == 106.03
```

Que se passe-t-il en cas de données non conformes, par exemple une valeur à virgule flottante qui ne représenterait pas un nombre ?

Nous préférons une réponse `None` à un plantage du programme.

```
from typing import Optional
import re
def try_parse_row(row: List[str]) -> Optional[StockPrice]:
    symbol, date_, closing_price_ = row
    # le symbole de l'action doit figurer entièrement en majuscules
    if not re.match(r"^[A-Z]+$", symbol):
        return None
    try:
        date = parse(date_).date()
    except ValueError:
        return None
    try:
        closing_price = float(closing_price_)
    except ValueError:
        return None
    return StockPrice(symbol, date, closing_price)

# doit retourner None pour les erreurs
assert try_parse_row(["MSFT0", "2018-12-14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12--14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12-14", "x"]) is None
# doit retourner la même chose qu'avant si les données sont correctes
assert try_parse_row(["MSFT", "2018-12-14", "106.03"]) == stock
```

Par exemple, si nous avons des cours d'action séparés par des virgules et contenant quelques données non conformes :

actions_virgules.txt

```
AAPL,6/20/2014,90.91
MSFT,6/20/2014,41.68
FB,6/20/3014,64.5
AAPL,6/19/2014,91.86
MSFT,6/19/2014,n/a
FB,6/19/2014,64.34
```

Nous sommes désormais en mesure de lire et de retourner uniquement les lignes valides :

```
import csv
data: List[StockPrice] = []
with open("comma_delimited_stock_prices.csv") as f:
    reader = csv.reader(f)
    for row in reader:
        maybe_stock = try_parse_row(row)
        if maybe_stock is None:
            print(f"skipping invalid row: {row}")
        else:
            data.append(maybe_stock)
```

Il nous appartient de décider ce qu'il convient de faire des lignes invalides. En général, il y a trois options : s'en débarrasser, revenir à la source et essayer de corriger les entrées non conformes ou absentes, ou ne rien faire et croiser les doigts. Si nous rencontrons une ligne non conforme parmi plusieurs millions, l'ignorer n'est pas un problème. En revanche, si la moitié de nos lignes présentent des données non conformes, il faudra y remédier.

En guise d'étape suivante, il est recommandé de vérifier la présence de valeurs extrêmes (section « Explorer les données ») ou de procéder à une investigation ad hoc. Par exemple, avez-vous remarqué qu'une des dates du fichier a pour année 3014 ? Ceci ne générera pas forcément une erreur, mais c'est une valeur manifestement erronée et vos résultats seront faussés si vous ne faites rien.

Dans la vraie vie, on se retrouve avec des virgules décimales manquantes, des zéros en trop, des coquilles et beaucoup d'autres problèmes qu'il est de votre devoir d'intercepter, peut-être pas officiellement, mais qui d'autre s'en chargera sinon ?

Manipuler les données

La manipulation des données est une compétence primordiale pour un expert en data science. Il s'agit plus d'une approche méthodologique que d'une technique spécifique. Nous nous contenterons ici de quelques exemples pour vous en donner une idée.

Supposons que nous travaillons sur des `dict` de cours d'actions :

```
data = [  
    StockPrice(symbol='MSFT',  
                date=datetime.date(2018, 12, 24),  
                closing_price=106.03),  
    # ...  
]
```

Interrogeons-nous sur ces données. Nous essayerons de repérer des motifs qui se répètent et nous prendrons du recul par rapport aux outils pour faciliter la manipulation.

Par exemple, supposons que nous cherchions le cours de clôture le plus élevé pour `AAPL`. Décomposons la recherche en étapes simples :

- 1 se limiter aux lignes `AAPL` ;
- 2 récupérer le cours de clôture `closing_price` de chaque ligne ;
- 3 trouver le maximum de ces cours.

Nous pouvons réaliser ces trois étapes d'un seul coup à l'aide d'une liste en compréhension :

```
max_aapl_price = max(stock_price.closing_price  
                     for stock_price in data  
                     if stock_price.symbol == "AAPL")
```

Plus généralement, nous pouvons chercher le cours de clôture le plus élevé pour chaque action de notre jeu de données. Voici une manière d'y parvenir :

- 1 créer un `dict` pour suivre les valeurs les plus élevées (nous utiliserons un `defaultdict` qui retourne `-inf` (moins infini) pour les valeurs manquantes, car toutes les valeurs seront forcément supérieures) ;
- 2 itérer sur nos données, en les actualisant.

Voici le code :

```
from collections import defaultdict  
max_prices: Dict[str, float] = defaultdict(lambda: float('-inf'))  
for sp in data:  
    symbol, closing_price = sp.symbol, sp.closing_price  
    if closing_price > max_prices[symbol]:  
        max_prices[symbol] = closing_price
```

Nous sommes à présent prêts à passer à des questions plus complexes, telles qu'identifier les variations les plus fortes et les plus faibles du cours sur un jour dans notre jeu de données. La variation en pourcentage s'exprime par `price_today / price_yesterday - 1`, ce qui veut dire qu'il faut trouver comment accéder aux cours d'hier et d'aujourd'hui et les associer.

Une méthode consiste à grouper les cours par symbole et ensuite, dans chaque groupe :

- 1 trier les cours par date ;
- 2 utiliser `zip` pour former des paires (précédent, courant) ;
- 3 transformer les paires en nouvelles lignes « variation en pourcentage ».

Pour commencer, regroupons les cours par symbole :

```
from typing import List
from collections import defaultdict
# collecter les cours par symbole
prices: Dict[str, List[StockPrice]] = defaultdict(list)
for sp in data:
    prices[sp.symbol].append(sp)
```

Les cours étant des tuples, ils sont triés selon leurs champs dans l'ordre : en premier par sigle, puis par date, puis par cours. En d'autres termes, si certains cours s'affichent tous avec le même sigle, `sorted` les trie par date (puis par cours, ce qui revient à ne rien faire puisque nous n'en avons qu'un par date) ; c'est ce que nous voulons.

```
# trier les cours par date
prices = {symbol: sorted(symbol_prices)
          for symbol, symbol_prices in prices.items()}
```

Nous nous en servons pour calculer une séquence de variations au jour le jour :

```
def pct_change(yesterday: StockPrice, today: StockPrice) -> float:
    return today.closing_price / yesterday.closing_price - 1

class DailyChange(NamedTuple):
    symbol: str
    date: datetime.date
    pct_change: float

def day_over_day_changes(prices: List[StockPrice]) -> List[DailyChange]:
    """considère que les cours pour chaque titre sont dans l'ordre"""
    return [DailyChange(symbol=today.symbol,
                        date=today.date,
                        pct_change=pct_change(yesterday, today))
            for yesterday, today in zip(prices, prices[1:])]
```

Puis, nous les collectons en totalité :

```
all_changes = [change
                for symbol_prices in prices.values()
                for change in day_over_day_changes(symbol_prices)]
```

À ce stade, il est facile de trouver le plus grand et le plus petit :

```
max_change = max(all_changes, key=lambda change: change.pct_change)
# consulter par exemple http://news.cnet.com/2100-1001-202143.html
assert max_change.symbol == 'AAPL'
assert max_change.date == datetime.date(1997, 8, 6)
assert 0.33 < max_change.pct_change < 0.34

min_change = min(all_changes, key=lambda change: change.pct_change)
# consulter par exemple http://money.cnn.com/2000/09/29/markets/techwrap/
assert min_change.symbol == 'AAPL'
assert min_change.date == datetime.date(2000, 9, 29)
assert -0.52 < min_change.pct_change < -0.51
```

Ce nouveau jeu de données `all_changes` peut nous servir à déterminer le meilleur mois pour investir dans les titres technologiques. Nous allons simplement examiner la variation quotidienne moyenne :

```
changes_by_month: List[DailyChange] = {month: [] for month in range(1, 13)}
for change in all_changes:
    changes_by_month[change.date.month].append(change)

avg_daily_change = {
    sum(change.pct_change
         for change in changes) / len(changes)
    for month, changes in changes_by_month.items()
}
# le meilleur mois est octobre
assert avg_daily_change[10] == max(avg_daily_change.values())
```

Nous nous livrerons à ce genre de manipulations tout au long du livre sans donner trop d'explications sur le sujet.

Changer d'échelle

De nombreuses techniques sont sensibles à l'échelle utilisée pour vos données. Par exemple, étant donné la taille et le poids de centaines de data scientists, supposez que vous essayiez de constituer des groupes (*clusters*) selon leurs mensurations.

Intuitivement, nous aimerions avoir des groupes qui représentent des points proches les uns des autres, ce qui veut dire qu'il faut mesurer d'une manière ou d'une autre la distance entre les points. Nous disposons déjà d'une fonction euclidienne `distance`. Une approche naturelle consiste à considérer les paires (taille, poids) comme des points dans un espace à deux dimensions. Prenons les personnes répertoriées dans le tableau 10-1.

Tableau 10–1 Taille et poids

Personne	Taille (pouces)	Taille (cm)	Poids (livres)
A	63	160	150
B	67	170.2	160
C	70	177.8	171

Si nous mesurons la taille en pouces, alors B a A pour voisin le plus proche :

```
from scratch.linear_algebra import distance
a_to_b = distance([63, 150], [67, 160]) # 10.77
a_to_c = distance([63, 150], [70, 171]) # 22.14
b_to_c = distance([67, 160], [70, 171]) # 11.40
```

Cependant, si nous mesurons la taille en centimètres, alors c'est désormais C qui est le plus proche de B :

```
a_to_b = distance([160, 150], [170.2, 160]) # 14.28
a_to_c = distance([160, 150], [177.8, 171]) # 27.53
b_to_c = distance([170.2, 160], [177.8, 171]) # 13.37
```

Évidemment, le fait qu'un simple changement d'unité change le résultat pose problème. Pour cette raison, quand les dimensions ne sont pas comparables entre elles, nous *redimensionnons nos données* de sorte que chaque dimension ait une espérance de 0 et un écart-type de 1. Cela permet de se débarrasser efficacement de la nature des unités en convertissant chaque dimension en « écart-type par rapport à la moyenne ».

Pour commencer, il faut calculer l'espérance et l'écart-type (*standard_deviation*) de chaque dimension :

```
from typing import Tuple
from scratch.linear_algebra import vector_mean
from scratch.statistics import standard_deviation
def scale(data: List[Vector]) -> Tuple[Vector, Vector]:
    """retourne l'espérance et l'écart-type de chaque colonne"""
    dim = len(data[0])
    means = vector_mean(data)
    stdevs = [standard_deviation([vector[i] for vector in data])
              for i in range(dim)]
    return means, stdevs

vectors = [[-3, -1, 1], [-1, 0, 1], [1, 1, 1]]
means, stdevs = scale(vectors)
assert means == [-1, 0, 1]
assert stdevs == [2, 1, 0]
```


Ensuite, nous allons les utiliser pour créer un nouvel ensemble de données :

```
def rescale(data: List[Vector]) -> List[Vector]:
    """Change l'échelle des entrées, de sorte que chaque position ait pour
       espérance 0 et pour écart-type 1.
       Ignore les positions dont l'écart-type est de 0."""
    dim = len(data[0])
    means, stdevs = scale(data)
    # faire une copie de chaque vecteur
    rescaled = [v[:] for v in data]
    for v in rescaled:
        for i in range(dim):
            if stdevs[i] > 0:
                v[i] = (v[i] - means[i]) / stdevs[i]
    return rescaled
```

Bien sûr, nous allons rédiger un test pour nous assurer que ce changement d'échelle produit bien les effets escomptés :

```
means, stdevs = scale(rescale(vectors))
assert means == [0, 0, 1]
assert stdevs == [1, 1, 0]
```

Comme toujours, vous devez faire preuve de discernement. Si vous exploitez un énorme jeu de données de tailles et de poids et si vous lui appliquez un filtre pour conserver seulement les personnes entre 69,5 pouces et 70,5 pouces, il est très probable (selon la question que vous étudiez) que la variation restante ne soit qu'un simple bruit et que vous ne vouliez pas traiter son écart-type sur un pied d'égalité avec ceux des autres dimensions.

En aparté : tqdm

Nous finissons souvent par effectuer des calculs qui prennent un temps considérable. Dans de telles situations, il est appréciable de savoir que les choses avancent et de connaître le temps d'attente prévu.

Une méthode consiste à utiliser la bibliothèque `tqdm`. Celle-ci génère des indicateurs de progression personnalisés. Sachant que nous en utiliserons certains dans le reste de l'ouvrage, prenons le temps d'apprendre leur fonctionnement.

Pour commencer, vous devez installer la bibliothèque `tqdm` :

```
python -m pip install tqdm
```

`tqdm` présente peu de caractéristiques à connaître. Tout d'abord, il faut savoir qu'un élément itérable incorporé dans `tqdm.tqdm` va produire une barre de progression :

```
import tqdm
for i in tqdm.tqdm(range(100)):
    # faire les choses lentement
    _ = [random.random() for _ in range(1000000)]
```

Cela produit une sortie présentant un aspect similaire à ce qui suit :

```
56%|██████████          | 56/100 [00:08<00:06, 6.49it/s]
```

Plus particulièrement, cet élément vous montre quelle fraction de votre boucle a été parcourue (bien qu'il ne le puisse pas si vous utilisez un générateur), le temps d'exécution déjà écoulé et le temps d'exécution restant attendu.

Dans cet exemple (où nous nous contentons d'incorporer un appel à `range`), vous pouvez vous contenter d'utiliser `tqdm.trange`.

Vous pouvez également définir la description de la barre de progression en cours d'exécution. Pour ce faire, vous devez capturer l'itérateur `tqdm` dans une instruction `with` :

```
from typing import List
def primes_up_to(n: int) -> List[int]:
    primes = [2]
    with tqdm.trange(3, n) as t:
        for i in t:
            # i est premier si aucun nombre premier plus petit ne le divise
            i_is_prime = not any(i % p == 0 for p in primes)
            if i_is_prime:
                primes.append(i)
            t.set_description(f"{len(primes)} primes")
    return primes

my_primes = primes_up_to(100_000)
```

Ce code ajoute une description semblable à ce qui suit, avec un compteur qui s'actualise à mesure que de nouveaux nombres premiers sont identifiés :

```
5116 primes: 50%|██████    | 49529/99997 [00:03<00:03, 15905.90it/s]
```

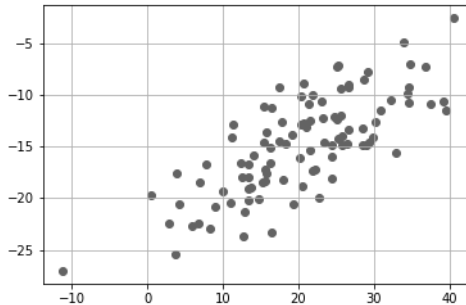
Utiliser `tqdm` rendra parfois votre code fragile : le rafraîchissement de l'affichage se montrera médiocre et la boucle pourra même se bloquer. Et si, accidentellement, vous incorporez une boucle `tqdm` dans une autre, il peut se produire des choses étranges. Il n'en reste pas moins que, en général, les avantages compensent ces inconvénients. Nous utiliserons donc `tqdm` dès lors que nous serons confrontés à des calculs à l'exécution lente.

Réduire la dimension

Parfois, les dimensions « effectives » (ou utiles) des données ne correspondent pas à celles dont nous disposons. Prenons l'exemple de la figure 10–6.

Figure 10–6

Données avec des axes
« non conformes »



L'essentiel de la variation des données semble exister pour une dimension unique qui ne correspond ni à l'axe des x , ni à celui des y .

Face à un tel cas, nous pouvons utiliser la technique dite « analyse en composantes principales » (*Principal Component Analysis – PCA*) pour extraire une ou plusieurs dimensions afin de capturer la plus grande partie de la variation des données.

Remarque

En pratique, on n'utilise pas cette technique sur un jeu de données comportant aussi peu de dimensions. Elle présente plus d'utilité quand votre jeu de données a de nombreuses dimensions et que vous cherchez un sous-ensemble qui capture l'essentiel de la variation. Malheureusement, il est difficile d'illustrer ce genre de cas sur une page de livre en deux dimensions.

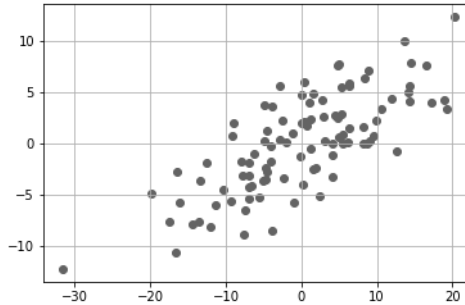
La première étape consiste à transformer les données de manière à ce que chaque dimension ait pour espérance zéro (si nous ne le faisons pas, nos méthodes risqueraient d'identifier l'espérance elle-même, plutôt que la variation de données) :

```
from scratch.linear_algebra import subtract
def de_mean(data: List[Vector]) -> List[Vector]:
    """Recentre les données pour qu'elles présentent une espérance de 0
    dans chaque dimension."""
    mean = vector_mean(data)
    return [subtract(vector, mean) for vector in data]
```

La figure 10–7 montre un exemple de données ainsi traitées (retrait d'espérance).

Figure 10–7

Les données après retrait
de l'espérance



Maintenant, étant donnée une matrice x après retrait d'espérance, nous pouvons demander quelle est la direction qui capte la plus grande variance de données.

Plus particulièrement, pour une direction donnée d (un vecteur de longueur 1), chaque ligne x de la matrice correspond à l'extension de la fonction $\text{dot}(x, d)$ dans la direction d . Et chaque vecteur w non nul détermine une direction si nous le redimensionnons pour obtenir une longueur 1 :

```
from scratch.linear_algebra import magnitude
def direction(w: Vector) -> Vector:
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

De fait, pour un vecteur non nul w , nous pouvons calculer la variance de nos données dans la direction déterminée par w :

```
from scratch.linear_algebra import dot
def directional_variance(data: List[Vector], w: Vector) -> float:
    """retourne la variance de x dans la direction de w"""
    w_dir = direction(w)
    return sum(dot(v, w_dir) ** 2 for v in data)
```

Nous voulons trouver la direction qui maximise cette variance. Pour ce faire, nous pouvons utiliser la descente de gradient, dès lors que nous avons une fonction gradient :

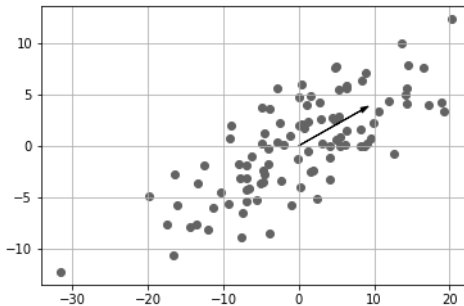
```
def directional_variance_gradient(data: List[Vector], w: Vector) -> Vector:
    """le gradient de variance directionnelle par rapport à w"""
    w_dir = direction(w)
    return [sum(2 * dot(v, w_dir) * v[i] for v in data)
            for i in range(len(w))]
```

À présent, la première composante principale dont nous disposons est tout simplement la direction qui maximise la fonction `directional_variance` :

```
from scratch.gradient_descent import gradient_step
def first_principal_component(data: List[Vector],
                             n: int = 100,
                             step_size: float = 0.1) -> Vector:
    # commencer avec un essai aléatoire
    guess = [1.0 for _ in data[0]]
    with tqdm.trange(n) as t:
        for _ in t:
            dv = directional_variance(data, guess)
            gradient = directional_variance_gradient(data, guess)
            guess = gradient_step(guess, gradient, step_size)
            t.set_description(f"dv: {dv:.3f}")
    return direction(guess)
```

Pour le jeu de données réduit (retrait de l'espérance), ce programme renvoie la direction $[0.924, 0.383]$, qui semble être le principal axe dans lequel nos données varient (figure 10–8).

Figure 10–8
Première composante principale



Une fois que nous avons trouvé la direction de la première composante principale, nous pouvons projeter nos données dessus pour trouver les valeurs de cette composante :

```
from scratch.linear_algebra import scalar_multiply
def project(v: Vector, w: Vector) -> Vector:
    """retourne la projection de v vers la direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

Si nous voulons trouver d'autres composantes, nous devons d'abord retirer les projections de ces données :

```
from scratch.linear_algebra import subtract
def remove_projection_from_vector(v: Vector, w: Vector) -> Vector:
```

```

"""projette v vers w et retranche le résultat de v"""
return subtract(v, project(v, w))

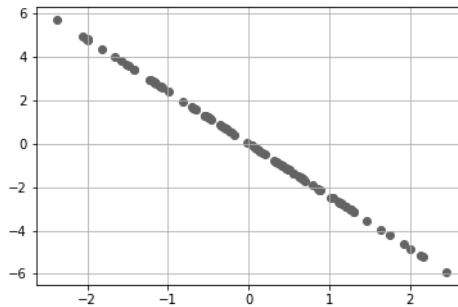
def first_principal_component(data: List[Vector],
                             w: Vector) -> List[Vector]:
    return [remove_projection_from_vector(v, w) for v in data]

```

Comme cet exemple est seulement à deux dimensions, une fois la première composante enlevée, il ne reste plus qu'une seule dimension (figure 10–9).

Figure 10–9

Les données après retrait de la première composante principale



À ce stade, nous trouverons la prochaine composante principale en répétant le processus sur le résultat de `remove_projection` (figure 10–10).

Avec un jeu de données ayant de nombreuses dimensions, nous pouvons trouver par itération autant de composantes que nous le souhaitons :

```

def pca(data: List[Vector], num_components: int) -> List[Vector]:
    components: List[Vector] = []
    for _ in range(num_components):
        component = first_principal_component(data)
        components.append(component)
        data = remove_projection(data, component)
    return components

```

Nous pouvons alors transformer les données en les transportant dans un espace de moindres dimensions grâce aux éléments :

```

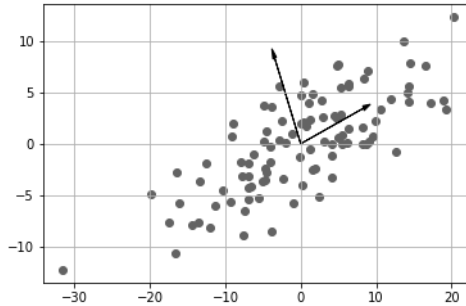
def transform_vector(v: Vector, components: List[Vector]) -> Vector:
    return [dot(v, w) for w in components]
def transform(data: List[Vector],
              components: List[Vector]) -> List[Vector]:
    return [transform_vector(v, components) for v in data]

```

Cette technique est utile pour plusieurs raisons. Premièrement, elle peut nous aider à nettoyer les données en éliminant les dimensions parasites et en consolidant celles qui sont hautement corrélées.

Figure 10–10

Les deux premières composantes principales



Deuxièmement, après avoir extrait de nos données une représentation à faible nombre de dimensions, nous pouvons utiliser une grande variété de techniques qui ne fonctionnent pas aussi bien sur un espace à nombreuses dimensions. Nous verrons des exemples de ces techniques au fil de l'ouvrage.

En contrepartie, si cette méthode contribue à construire de meilleurs modèles, elle peut aussi rendre ces derniers plus difficiles à interpréter. Il est facile de tirer des conclusions comme « chaque année d'expérience en plus se traduit par une augmentation moyenne de 10 000 \$ de salaire ». Il est beaucoup plus difficile de comprendre ce que cache l'affirmation « chaque augmentation de 0,1 dans la troisième composante principale ajoute 10 000 \$ de salaire en moyenne ».

Pour aller plus loin

- Comme indiqué à la fin du chapitre 9, `pandas`¹ est probablement l'outil Python de base pour nettoyer, transformer, manipuler et exploiter les données. Tous les exemples que nous avons exécutés à la main dans ce chapitre peuvent être considérablement simplifiés en faisant appel à `pandas`.

*Analyse de données en Python*² aux éditions Eyrolles (traduit de *Python for Data Analysis*, des éditions O'Reilly) est sans doute la meilleure ressource pour apprendre `pandas`.

- `scikit-learn` propose un grand nombre de fonctions de décomposition de matrice³, dont l'analyse en composantes principales.

1. <https://pandas.pydata.org/>

2. <https://www.editions-eyrolles.com/Livre/9782212141092/analyse-de-donnees-en-python>

3. <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.decomposition>

Apprentissage automatique

Je suis toujours prêt à apprendre, mais je n'aime pas toujours qu'on m'enseigne.

Winston Churchill

Beaucoup s'imaginent que la data science est surtout une affaire d'apprentissage automatique et que les experts passent leurs journées à construire, entraîner et ajuster des modèles. N'oublions pas que, généralement, les gens ne savent pas vraiment ce qu'est l'apprentissage automatique. En fait, la data science consiste surtout à transformer des problématiques métier en problématiques de données : collecter, comprendre, nettoyer, formater les données, ensuite l'apprentissage automatique est une préoccupation secondaire. Toutefois, c'est une préoccupation secondaire intéressante et essentielle que vous ne pouvez pas ignorer si vous voulez vous consacrer à la data science.

Modélisation

Avant d'aborder l'apprentissage automatique (*machine learning*), il faut commencer par parler des modèles.

Qu'est-ce qu'un modèle ? C'est simplement la spécification d'une relation mathématique (ou probabiliste) existant entre différentes variables.

Par exemple, si vous essayez de collecter de l'argent pour votre site de réseau social, vous pouvez construire un modèle économique (probablement une feuille de calcul) qui accepte en entrée *nombre d'utilisateurs*, *revenu des annonces par utilisateur* et *nombre d'employés* et en sortie *bénéfice annuel* pour les prochaines années. Un livre de recettes suit un modèle qui

associe des entrées *nombre de convives et degré d'appétit* à des *quantités d'ingrédients nécessaires*. Et, si vous avez regardé un jour du poker à la télévision, vous savez qu'on estime la probabilité de gain de chaque joueur en temps réel à partir d'un modèle qui prend en compte les cartes déjà dévoilées et la distribution de celles du sabot.

Le modèle économique est probablement fondé sur des relations mathématiques simples :

- bénéfice = recettes - dépenses ;
- revenu = nombre d'articles vendus $\times$ prix moyen ;
- etc.

Le modèle des recettes de cuisine est sans doute le résultat d'essais et d'erreurs : quelqu'un est allé dans une cuisine et a essayé différentes combinaisons d'ingrédients jusqu'à ce qu'il trouve ce qui lui plaît.

Le modèle du poker est basé sur la théorie des probabilités, les règles du jeu et quelques hypothèses raisonnables sur la manipulation aléatoire des cartes.

Qu'est-ce que l'apprentissage automatique ?

Si chacun a sa définition personnelle, nous parlerons ici d'*apprentissage automatique* pour faire référence à la création et à l'utilisation de modèles appris à partir des données. Dans d'autres contextes, on parle parfois de modélisation prédictive ou de *data mining*. Notre objectif consiste à utiliser les données existantes pour développer des modèles que nous utiliserons dans le but de prédire divers résultats pour de nouvelles données.

- Un courriel est-il ou non un spam ?
- Une transaction par carte bancaire est-elle frauduleuse ?
- Sur quelle annonce un acheteur a-t-il le plus de chance de cliquer ?
- Quelle est l'équipe de football qui va remporter le Super Bowl ?

Nous nous intéressons aux modèles supervisés (pour lesquels il existe un jeu de données d'apprentissage dans lequel les données sont étiquetées avec les bonnes réponses) et aux modèles non supervisés (pas de données étiquetées). Il existe d'autres types de modèles, que nous ne traiterons pas dans ce livre : semi-supervisés (seules quelques-unes des données sont étiquetées), en ligne (ajustement permanent aux données nouvellement arrivées) et de renforcement (après une série de prédictions, le modèle obtient un signal indiquant son efficacité).

Même dans la plus simple des situations, il existe des univers entiers de modèles susceptibles de décrire la relation qui vous intéresse. Dans la plupart des cas, nous choisirons une famille de modèles paramétrables et nous utiliserons les données pour apprendre les paramètres les mieux adaptés.

Par exemple, nous pourrions décider que la taille d'une personne est (à peu près) une fonction linéaire de son poids et utiliser les données pour déterminer quelle est cette fonction linéaire. Ou encore, nous pourrions décider qu'un arbre de décision constitue une bonne méthode pour diagnostiquer les maladies de nos patients et utiliser les données pour apprendre l'arbre

« optimal ». Dans ce livre, nous étudierons différentes familles de modèles à apprendre. Auparavant, nous devons mieux comprendre les principes de l'apprentissage automatique. Dans la suite de ce chapitre, nous présenterons quelques concepts de base avant de nous intéresser aux modèles eux-mêmes.

Surajustement et sous-ajustement

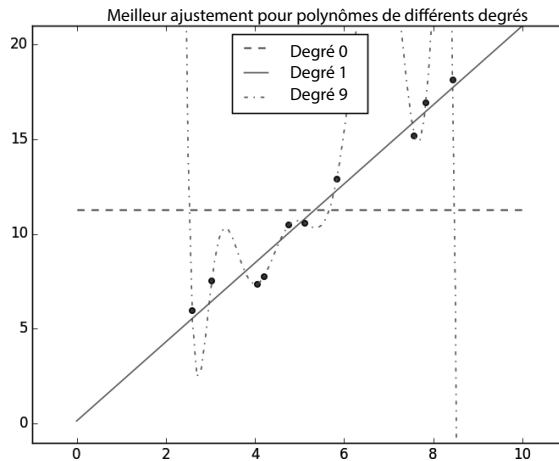
Un risque classique en apprentissage automatique consiste à être trop ajusté ou *surajusté*, à savoir produire un modèle performant sur les données d'apprentissage, mais qui se généralise mal en cas de données nouvelles. Cela revient à apprendre du « bruit » au niveau du modèle. En d'autres termes, cela revient à apprendre à identifier des entrées spécifiques non représentatives d'une règle générale, plutôt que les facteurs réellement prédictifs pour la sortie attendue.

Le risque opposé consiste à être *sous-ajusté* : produire un modèle aux performances médiocres même sur les données d'apprentissage, bien que normalement, face à un tel comportement, il est naturel de décider que votre modèle n'est pas assez efficace et d'en chercher un meilleur.

À la figure 11-1, j'ai fait correspondre trois polynômes à un échantillon de données (ne vous demandez pas comment j'ai fait, nous en parlerons plus loin).

Figure 11-1

Surajustement (*overfitting*) et sous-ajustement (*underfitting*)



La ligne horizontale montre le polynôme de degré 0 (c'est-à-dire une valeur constante) le mieux adapté. Il est très sous-adapté (sous-ajusté, donc) pour les données d'apprentissage. Le polynôme de degré 9 (c'est-à-dire à 10 paramètres) le mieux adapté passe exactement par chacune des données élémentaires d'apprentissage, mais il est « trop bien ajusté » : si nous ajoutons quelques nouvelles données matérialisées par des points, il est à peu près certain qu'il les manquera. La ligne de degré 1 dessine un équilibre plaisant : elle est proche de chaque point et, si ces données sont représentatives, elle sera aussi très proche des nouvelles données élémentaires.

Clairement, les modèles trop complexes sont trop ajustés et ne se généralisent pas efficacement dès lors qu'on sort des données d'apprentissage. Comment nous assurer que nos

modèles ne sont pas trop complexes ? L'approche la plus fondamentale suppose d'utiliser des données différentes pour élargir l'apprentissage du modèle et pour le tester.

La manière la plus simple consiste à diviser votre jeu de données, en en utilisant, par exemple, les deux tiers pour l'apprentissage et le tiers restant pour des mesures de performance :

```
import random
from typing import TypeVar, List, Tuple
X = TypeVar('X') # type générique pour représenter un point de données
def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
    """séparer les données selon les proportions [prob, 1 - prob]"""
    data = data[:] # faire une copie superficielle
    random.shuffle(data) # parce que shuffle modifie la liste
    cut = int(len(data) * prob) # utiliser prob pour trouver un seuil
    return données[:seuil], données[seuil:] # et diviser la liste mélangée (shuffle)

data = [n for n in range(1000)]
train, test = split_data(data, 0.75)
# les proportions doivent être correctes
assert len(train) == 750
assert len(test) == 250
# et les données d'origine doivent être préservées (dans un certain ordre)
assert sorted(train + test) == data
```

Souvent, nous aurons apparié des variables d'entrée et des variables en sortie. Dans ce cas, il faut s'assurer de placer les valeurs correspondantes ensemble, que ce soit dans les données d'apprentissage ou les données de test :

```
Y = TypeVar('Y') # type générique pour les variables de sortie
def train_test_split(xs: List[X], ys: List[Y],
                    test_pct: float) -> Tuple[List[X], List[X],
                                              List[Y], List[Y]]:
    """Créer un ensemble d'apprentissage et un de test en les associant
    aux valeurs de sortie correspondantes."""
    # générer les index et les séparer
    idxs = [i for i in range(len(xs))]
    train_idx, test_idx = split_data(idxs, 1 - test_pct)
    return ([xs[i] for i in train_idx], # x_train
            [xs[i] for i in test_idx], # x_test
            [ys[i] for i in train_idx], # y_train
            [ys[i] for i in test_idx]) # y_test
```

Comme toujours, nous voulons nous assurer du bon fonctionnement de notre code :

```
xs = [x for x in range(1000)] # xs sont 1 ... 1000
ys = [2 * x for x in xs] # chaque y_i vaut deux fois x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)
# vérifier que les proportions sont correctes
assert len(x_train) == len(y_train) == 750
```

```
assert len(x_test) == len(y_test) == 250
# vérifier que les données correspondantes sont correctement appariées
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))
```

Ensuite, vous pouvez écrire quelque chose comme ce qui suit :

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

Si le modèle est trop ajusté aux données d'apprentissage, on peut s'attendre à ce qu'il soit peu performant sur les données de test (complètement différentes des données d'apprentissage). Autrement dit, s'il est performant avec les données de test, vous pouvez considérer raisonnablement qu'il est adapté, mais pas trop ajusté.

Cependant, ce raisonnement est erroné dans certains cas...

Le premier cas se produit s'il y a des schémas communs dans les données de test et d'apprentissage, qui ne se généralisent pas à un jeu de données plus important.

Par exemple, imaginons que vos données concernent les activités d'utilisateurs, avec une ligne par utilisateur et par semaine. Dans ce genre de cas, les utilisateurs vont pour la plupart apparaître à la fois dans les données d'apprentissage et dans celles de test ; or, certains modèles sont susceptibles d'apprendre à identifier ces utilisateurs plutôt qu'à découvrir des relations impliquant des attributs. Certes, ce n'est pas un gros problème, mais cela m'est arrivé une fois.

Le problème est plus sérieux si vous utilisez la séparation test/apprentissage non seulement pour juger un modèle, mais aussi pour départager plusieurs modèles. Dans ce cas, bien que chaque modèle ne soit pas forcément surajusté, la décision de « choisir le modèle le plus performant sur le jeu de données de test » est un méta-apprentissage qui fait alors fonctionner le jeu de test comme un deuxième jeu d'apprentissage (évidemment, le modèle qui était le plus performant sur le jeu de données de test sera implicitement performant sur ce jeu).

Dans une telle situation, il faut séparer les données en trois parties : un jeu d'apprentissage pour construire les modèles, un jeu de validation pour sélectionner un modèle et enfin un jeu de test pour juger le modèle retenu à la fin de l'expérience.

Exactitude

Quand je ne pratique pas la data science, je m'intéresse à la médecine. Pendant mon temps libre, j'ai mis au point un test peu coûteux et non invasif destiné à prédire – avec une précision supérieure à 98 % – si un nouveau-né développera ou non une leucémie. Mon avocat m'ayant convaincu que ce test n'est pas brevetable, j'en partage les détails avec vous : mon test prédit la leucémie si et seulement si le bébé se prénomme Lucie (à l'oreille, ça ressemble un peu à « leucémie »).

Comme nous allons le voir, le test est fiable à 98 %. Il est aussi particulièrement stupide et constitue une bonne illustration de la raison pour laquelle on n'utilise pas l'exactitude (*accuracy* en anglais) pour mesurer la qualité (classification binaire) d'un modèle.

Imaginons que vous construisez un modèle de jugement binaire. Ce message est-il un spam ? Faut-il recruter ce candidat ? Ce passager de l'avion est-il un terroriste ?

Soit un jeu de données étiquetées et un modèle prédictif de ce type, chaque donnée élémentaire (point de données) tombe dans une des quatre catégories suivantes :

- Vrai positif
Ce message est un spam et nous l'avons identifié à juste titre comme tel.
- Faux positif (erreur de type 1)
Ce message n'est pas un spam, mais nous avons prédit que c'en était un.
- Faux négatif (erreur de type 2)
Ce message est un spam, mais nous avons prédit que ce n'en était pas un.
- Vrai négatif
Ce message n'est pas un spam et nous l'avons identifié à juste titre comme tel.

Nous représentons souvent ces résultats sous forme de compteurs dans une *matrice de confusion* (ou tableau de contingence).

	Spam	Non-spam
Prédit (reconnaît) un spam	Vrai positif	Faux positif
Prédit (reconnaît) un non-spam	Faux négatif	Vrai négatif

Voyons maintenant comment mon test de leucémie s'inscrit dans ce cadre. De nos jours, environ 5 bébés sur 1 000 se prénomment Lucie. La prévalence de la leucémie sur la durée de vie¹ est d'environ 1,4 %, soit 14 personnes sur 1 000.

Si nous pensons que ces deux facteurs sont indépendants et appliquons mon test *Lucie veut dire leucémie* à 1 million de personnes, nous nous attendons à une matrice de confusion revêtant la forme suivante :

	Leucémie	Absence de leucémie	Total
Lucie	70	4 930	5 000
Non Lucie	13 930	981 070	995 000
Total	14 000	986 000	1 000 000

1. <https://seer.cancer.gov/statfacts/html/leuks.html>

Nous pouvons utiliser ces résultats pour calculer différentes statistiques sur la performance du modèle. Par exemple, l'exactitude est définie comme la fraction de prédictions correctes :

```
def accuracy(tp: int, fp: int, fn: int, tn: int) -> float:
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

assert accuracy(70, 4930, 13930, 981070) == 0.98114
```

Voilà une valeur plutôt impressionnante. Pourtant, il est clair que ce n'est pas un bon test, ce qui veut dire que nous ne devrions pas accorder trop de confiance à l'exactitude brute.

Il est classique d'examiner la combinaison de ce qu'on appelle la précision et le rappel. La *précision* mesure à quel point nos prédictions positives sont exactes :

```
def precision(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fp)

assert precision(70, 4930, 13930, 981070) == 0.014
```

Le *rappel* (*recall*) mesure quelle fraction de vrais positifs notre modèle a identifiée :

```
def recall(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fn)

assert recall(70, 4930, 13930, 981070) == 0.005
```

Ces deux valeurs sont mauvaises, ce qui reflète la piètre qualité de notre modèle.

Parfois, la précision et le rappel se combinent dans le score f_1 , défini comme suit :

```
def f1_score(tp: int, fp: int, fn: int, tn: int) -> float:
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)
    return 2 * p * r / (p + r)
```

Il s'agit de la *moyenne harmonique*² de la précision et du rappel ; elle est nécessairement située entre les deux.

En général, le choix d'un modèle suppose d'arbitrer entre la précision et le rappel. À cet égard, un modèle qui prédit *oui* même quand il a à peine confiance aura sans doute un rappel élevé, mais une précision faible ; un modèle qui prédit *oui* uniquement quand il est extrêmement confiant aura un rappel faible et une précision élevée.

2. https://fr.wikipedia.org/wiki/Moyenne_harmonique

Vous pouvez aussi considérer la situation comme un arbitrage (un compromis) entre les faux positifs et les faux négatifs. Dire *oui* trop souvent génère beaucoup de faux positifs ; dire *non* trop souvent génère beaucoup de faux négatifs.

Imaginons qu'il existe 10 facteurs de risque pour la leucémie et que, plus vous cumulez ces facteurs, plus votre risque de développer une leucémie est important. Dans ce cas, on peut imaginer un continuum de tests : *prédit la leucémie si au moins un facteur de risque*, *prédit la leucémie si au moins deux facteurs de risque* et ainsi de suite. Dès lors que vous augmentez le seuil, vous augmentez la précision du test (des personnes avec davantage de facteurs de risques sont plus enclines à développer la maladie) et vous diminuez le rappel (car de moins en moins de malades potentiels vont atteindre le seuil). Dans ce genre de cas, choisir le seuil revient à trouver le bon compromis.

Le compromis entre le biais et la variance

Une autre manière d'envisager le problème du modèle trop ajusté consiste à le considérer comme un arbitrage entre le *biais* et la *variance*.

Les deux mesurent ce qui arriverait si vous recommenciez l'apprentissage de votre modèle plusieurs fois sur différents jeux de données (issus de la même population très nombreuse).

Par exemple, le modèle de degré 0 de la figure 11–1 fait de nombreuses erreurs pour à peu près n'importe quel jeu de données d'apprentissage (tirées de la même population), ce qui veut dire qu'il présente un biais important. Cependant, deux jeux de données d'apprentissage quelconques choisis au hasard devraient produire des résultats similaires (car ces jeux de données devraient présenter les mêmes valeurs moyennes). Nous dirons que la variance est faible. Un biais élevé et une variance faible sont caractéristiques d'un sous-ajustement.

À l'opposé, le modèle de degré 9 correspond très exactement au jeu de données d'apprentissage. Le biais est très faible, mais la variance est forte (il est probable que deux jeux de données d'apprentissage donneront des modèles très différents). C'est un cas typique d'une situation de surajustement.

Aborder les questions de modèle sous cet angle peut vous aider à comprendre pourquoi votre modèle ne fonctionne pas très bien.

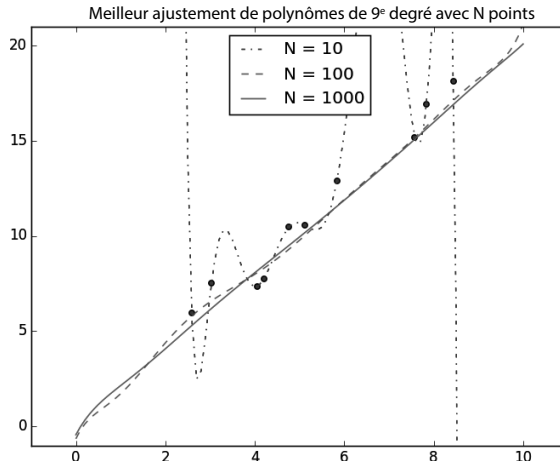
Si votre modèle présente un biais élevé (ce qui veut dire qu'il n'est pas performant avec vos données d'apprentissage), essayez d'ajouter davantage de variables (*features*). Dans le cas de la figure 11–1, passer du modèle de degré 0 à celui de degré 1 constituait une amélioration importante.

Si votre modèle a une variance élevée, alors, au contraire, enlevez des variables. Une autre solution consiste à obtenir davantage de données (si cela est possible).

À la figure 11–2, nous avons ajusté un polynôme de degré 9 à différentes tailles d'échantillons. Le modèle basé sur 10 données élémentaires s'étale partout, comme nous l'avons vu précédemment. Si nous prenons 100 données élémentaires d'apprentissage, alors au contraire il devient moins bien ajusté. Le modèle construit à partir de 1 000 données élémentaires est

assez similaire au modèle de degré 1. À complexité de modèle constante, plus vous avez de données, plus il est difficile de surajuster. D'un autre côté, la mise à disposition de davantage de données reste sans effet sur le biais. Si votre modèle n'utilise pas assez de variables pour capter les éléments réguliers dans les données, il est inutile d'ajouter encore plus de données.

Figure 11-2
Réduction de la variance
avec davantage de données



Extraction et sélection des variables

Comme nous l'avons déjà dit, si vos données ne présentent pas assez de variables, votre modèle est probablement sous-ajusté. À l'inverse, si vos données ont trop de variables, il est facile de surajuster. Cependant, que sont ces variables et d'où viennent-elles ?

Les *variables* sont des caractéristiques ; ce sont les entrées que nous fournissons à notre modèle.

Dans le cas le plus simple, les variables vous sont fournies ; point final. Si vous voulez prédire le salaire d'une personne à partir de ses années d'expérience, ces dernières sont alors la seule variable dont vous ayez besoin (bien que, comme nous l'avons vu précédemment, vous pouvez aussi envisager d'ajouter les années d'expérience au carré, au cube et ainsi de suite, pour voir si vous en tirez un meilleur modèle).

Tout devient plus intéressant quand vos données se compliquent. Imaginons que vous essayez de construire un filtre destiné à prédire si un message reçu est ou non un spam. Les modèles ne savent généralement pas traiter un courrier électronique brut, ce dernier n'étant qu'un ensemble d'éléments de texte. Il faut donc extraire des variables. Par exemple :

- Est-ce que le message contient le mot « viagra » ?
- Combien de fois la lettre « d » se présente-t-elle ?
- Quel est le domaine de l'expéditeur ?

La réponse à la première question est simplement un *oui* ou un *non*, que nous encoderons par 1 ou 0. La réponse à la deuxième est un nombre. Et la réponse à la troisième est un choix parmi un ensemble d'options indépendantes.

La plupart du temps, voire systématiquement, nous devrons extraire de nos données des variables correspondant à une de ces trois catégories. En outre, le type des variables utilisées va « contraindre » les types de modèles utilisables.

- Le classifieur naïf de Bayes (*Naive Bayes*), que nous développerons au chapitre 13, est adapté aux fonctions oui-ou-non.
- Les modèles de régression, que nous étudierons aux chapitres 14 à 16, font appel à des fonctions numériques (qui peuvent comporter des variables indicatrices acceptant uniquement les valeurs 0 et 1).
- Les arbres de décision, que nous aborderons au chapitre 17, peuvent traiter des données numériques ou des catégories.

Dans le cas du filtre antispam, nous avons cherché comment créer des variables mais, parfois, nous chercherons au contraire à en supprimer.

Par exemple, vos entrées peuvent être des vecteurs de plusieurs centaines de nombres. En fonction de la situation, il est parfois recommandé de les réduire à une poignée de dimensions importantes (chapitre 10) et de n'utiliser que ce petit nombre de variables. Ou il peut être préférable d'adopter une technique (comme la « régularisation » que nous aborderons plus loin) qui pénalise les modèles exploitant trop de variables.

Comment choisir les variables ? Ici, une combinaison d'expérience et d'expertise du domaine entre en jeu. Si vous recevez beaucoup de messages, vous avez probablement l'intuition que la présence de certains mots constitue un bon indicateur pour repérer les pourriels. Et vous avez sans doute aussi l'intuition que le nombre d'occurrences de la lettre « d » constitue vraisemblablement un indicateur peu intéressant à cette fin. Toutefois, en général vous devrez vous livrer à différents essais, ce qui fait toute la saveur de l'exercice.

Pour aller plus loin

- Poursuivez votre lecture ! Les chapitres suivants sont consacrés à différentes familles de modèles d'apprentissage automatique.
- Le cours Coursera *Machine Learning*³ est le MOOC d'origine sur le sujet. C'est le bon endroit pour comprendre en profondeur les fondements de l'apprentissage automatique.
- *The Elements of Statistical Learning*⁴, de Jerome H. Friedman, Robert Tibshirani et Trevor Hastie (Springer), est un ouvrage de référence qui peut être téléchargé gratuitement. Toutefois prenez garde, il est réservé aux forts en maths !

3. <https://www.coursera.org/learn/machine-learning>

4. <https://web.stanford.edu/~hastie/ElemStatLearn/>

12

Les k plus proches voisins

Si vous voulez embarrasser vos voisins, racontez la vérité sur leur cas.

Pietro Aretino

Imaginons que vous essayez de prédire pour qui je vais voter aux prochaines élections présidentielles. Si vous ne savez rien de moi (et si vous disposez des données qui vont bien), une méthode consiste à poser la même question à mes voisins. Habitant comme moi en centre-ville de Seattle, mes voisins prévoient toujours de voter pour le candidat démocrate, ce qui suggère que *candidat démocrate* est aussi une bonne prédiction me concernant.

Imaginons maintenant que vous en sachiez davantage sur moi que ma simple situation géographique : par exemple mon âge, ce que je gagne, combien j'ai d'enfants et ainsi de suite. Dans la mesure où mon comportement est influencé (ou caractérisé) par ces facteurs, analyser mes voisins les plus proches à l'aune de toutes ces dimensions donnera probablement une meilleure prédiction qu'analyser l'ensemble de mes voisins. Telle est l'idée qui sous-tend la classification par « plus proches voisins ».

Modèle

Le modèle des plus proches voisins est un des modèles prédictifs les plus simples. Il ne fait aucune hypothèse mathématique, ni appel à toute une artillerie. Il nécessite très peu de choses :

- une notion de distance ;
- et l'hypothèse que des points proches les uns des autres sont similaires.

Pour la plupart, les techniques examinées dans ce livre envisagent le jeu de données dans son ensemble pour y découvrir des motifs répétés. La méthode des plus proches voisins, au contraire, ignore volontairement de nombreuses informations ; en effet, la prédiction pour chaque nouveau point de données ne dépend que des quelques points les plus proches.

De plus, la méthode des plus proches voisins ne vous aidera sans doute pas à comprendre les ressorts du phénomène que vous étudiez. Prédire mes votes à partir de ceux de mes voisins ne vous apprend pas grand-chose sur les raisons de mon choix, là où d'autres modèles qui s'appuient sur mon revenu ou mon statut marital peuvent apporter ce genre d'informations.

Dans le cas général, nous disposons de quelques données élémentaires et d'un jeu d'étiquettes correspondantes. Ces dernières peuvent valoir `True` ou `False` ; dans ce cas, elles indiquent si chaque entrée vérifie une condition comme *Est-ce du spam ?*, *Est-ce toxique ?* ou *Est-ce un bon film ?* Elles peuvent aussi constituer des catégories, comme des classes de films (*G*, *PG*, *PG-13*, *R*, *NC-17*) ou les noms des candidats à l'élection présidentielle, ou encore des langages de programmation préférés.

Dans notre cas, les données élémentaires seront des vecteurs, ce qui veut dire que nous pourrions utiliser la fonction `distance` vue au chapitre 4.

Soit un nombre `k` de valeur 3 ou 5. Quand nous décidons de classer une nouvelle donnée élémentaire, nous recherchons les `k` données élémentaires étiquetées les plus proches et nous les faisons voter sur la nouvelle entrée.

Pour cela, il nous faut une fonction de comptage des votes. Une des possibilités se présente comme suit :

```
from typing import List
from collections import Counter
def raw_majority_vote(labels: List[str]) -> str:
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner

assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

Cette méthode ne produit pas de résultat particulièrement intelligent en cas d'égalité. Par exemple, si nous notons des films et que les plus proches sont notés *G*, *G*, *PG*, *PG* et *R*, alors *G* et *PG* ont chacun deux votes. Dans ce cas, nous avons plusieurs options :

- choisir un des vainqueurs au hasard ;
- pondérer les votes par la distance et déterminer le vainqueur pondéré ;
- ou réduire `k` jusqu'à ce qu'il ne reste plus qu'un seul vainqueur.

Nous choisissons la troisième option :

```
def majority_vote(labels: List[str]) -> str:
    """suppose que les labels sont triés du plus proche au plus éloigné"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
```

```

num_winners = len([count
                    for count in vote_counts.values()
                    if count == winner_count])
if num_winners == 1:
    # un seul gagnant bien défini ; on le retourne
    return winner
else:
    # recommencer en éliminant le plus éloigné
    return majority_vote(labels[:-1])

# égalité ; examiner les quatre premiers, puis 'b'
assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'

```

Cette méthode est certaine d'aboutir car, dans le pire des cas, nous irons jusqu'à ne garder qu'une seule étiquette... qui sera alors déclarée vainqueur. Avec cette fonction, la création du classificateur est aisée :

```

from typing import NamedTuple
from scratch.linear_algebra import Vector, distance
class LabeledPoint(NamedTuple):
    point: Vector
    label: str

def knn_classify(k: int,
                  labeled_points: List[LabeledPoint],
                  new_point: Vector) -> str:
    # trier les points de données étiquetés du plus proche au plus éloigné
    by_distance = sorted(labeled_points,
                        key=lambda lp: distance(lp.point, new_point))
    # chercher les labels pour le k le plus proche
    k_nearest_labels = [lp.label for lp in by_distance[:k]]
    # et les faire voter
    return majority_vote(k_nearest_labels)

```

Examinons son fonctionnement de plus près.

Exemple : jeu de données Iris

Le jeu de données Iris est un incontournable de l'apprentissage automatique. Connu sous le nom de Iris de Fisher, il contient un ensemble de mesures portant sur 150 fleurs représentant trois espèces d'iris. Pour chaque fleur, nous connaissons la longueur et la largeur de son pétale, celles de son sépale, ainsi que son espèce. Vous pouvez télécharger ce jeu de données depuis <https://archive.ics.uci.edu/ml/datasets/iris> :

```

import requests
données = requests.get("https://archive.ics.uci.edu/ml/machine-
                        "learning-databases/iris/iris.data")
with open('iris.dat', 'w') as f:
    f.write(data.text)

```

Les données sont comprises dans des champs séparés par des virgules, comme suit :

```
sepal_length, sepal_width, petal_length, petal_width, class
```

Par exemple, la première rangée présente l'aspect suivant :

```
5.1,3.5,1.4,0.2,Iris-setosa
```

Dans cette section, nous allons tenter de construire un modèle capable de prévoir l'espèce à partir des quatre premières mesures.

Pour commencer, chargeons et explorons les données. Notre fonction de voisins les plus proches a besoin d'un `LabeledPoint` ; nous représenterons donc nos données de la manière suivante :

```
from typing import Dict
import csv
from collections import defaultdict

def parse_iris_row(row: List[str]) -> LabeledPoint:
    """sepal_length, sepal_width, petal_length,
        petal_width, class"""
    measurements = [float(value) for value in row[:-1]]
    # La classe est, par exemple, "Iris-virginica" ;
    # nous voulons uniquement "virginica".
    label = li[-1].split("-")[-1]
    return LabeledPoint(measurements, label)

with open('iris.dat') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader]
                for li in lecteur
                if len(li)>0]

# En outre, nous grouperons les points uniquement par espèce/label
# de manière à les représenter graphiquement.
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

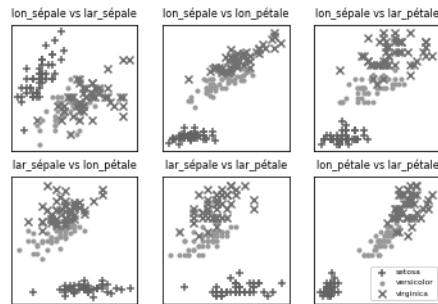
Pour visualiser leurs variations par espèce, nous avons besoin de tracer les mesures. Malheureusement, ces dernières sont quadridimensionnelles, ce qui rend leur tracé difficile. Une approche consiste à examiner les nuages de points pour chacune des six paires de mesures (figure 12-1). Je n'entrerai pas ici dans les détails, mais il s'agit là d'une excellente illustration de choses plus complexes que vous pouvez réaliser avec `matplotlib`. Il est donc conseillé de l'étudier :

```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
```

```
marks = ['+', '.', 'x'] # nous avons 3 espèces, donc 3 marqueurs
fig, ax = plt.subplots(2, 3)
for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
        ax[row][col].set_title(f'{metrics[i]} vs {metrics[j]}', fontsize=8)
        ax[row][col].set_xticks([])
        ax[row][col].set_yticks([])
        for marks, (species, points) in zip(marks,
                                             points_by_species.items()):
            xs = [point[i] for point in points]
            ys = [point[j] for point in points]
            ax[row][col].scatter(xs, ys, marker=mark, label=species)
ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

Figure 12–1

Nuages de points de données d'Iris



Vous constatez que les mesures se présentent comme des agrégats – des *clusters* – par espèce. Par exemple, en examinant les seules longueurs et largeurs de sépale, vous ne distingueriez probablement pas les espèces *versicolor* et *virginica*. En revanche, une fois les longueurs et largeurs de pétale ajoutées au mélange, vous serez probablement en mesure de prédire les espèces en vous fondant sur les voisins les plus proches.

Pour commencer, divisons les données en un ensemble de test et un ensemble d'apprentissage :

```
import random
from scratch.machine_learning import split_data
random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)

assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
```

L'ensemble d'apprentissage constituera les « voisins » que nous utiliserons pour classer les points dans l'ensemble de test. Il nous faut simplement choisir une valeur pour *k*, le nombre

de voisins qui vont voter. Si la valeur est trop faible (considérons $k = 1$), nous laissons trop d'influence aux valeurs extrêmes (outliers) ; si elle est trop élevée (considérons $k = 105$), nous prédisons les classes les plus courantes dans l'ensemble de données.

Dans une « vraie » application (et avec davantage de données), nous créerions un ensemble de validation séparé et l'utiliserions pour choisir k . Ici, nous nous contenterons d'utiliser $k = 5$:

```
from typing import Tuple
# suivre le nombre de fois que nous constatons (prédit, réel)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0
for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label
    if predicted == actual:
        num_correct += 1
    confusion_matrix[(predicted, actual)] += 1

pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

Sur cet ensemble simple de données, le modèle affiche des prédictions quasi-parfaites. Il prédit une seule fois *versicolor* pour *virginica*, mais pour le reste il est parfaitement exact.

Malédiction de la dimension

Le modèle des k plus proches voisins fonctionne difficilement lorsque les données sont représentées dans des espaces de grande dimension : c'est ce qu'on appelle « la malédiction de la dimension » (*curse of dimensionality*). En effet, dans les espaces qui affichent un grand nombre de dimensions, les points ont tendance à être éloignés les uns des autres. Une manière de se représenter le problème consiste à générer aléatoirement des paires de points dans le « cube unitaire » à d dimensions, pour diverses valeurs de d , puis à calculer les distances entre celles-ci.

À présent, la génération aléatoire de points devrait être une seconde nature :

```
def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
```

Il en est de même pour l'écriture de fonctions pour générer les distances :

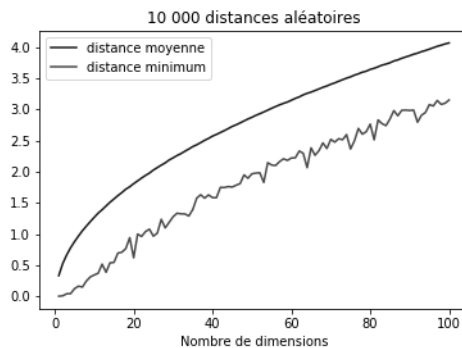
```
def random_distances(dim: int, num_pairs: int) -> List[float]:
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```


Pour chaque dimension de 1 à 100, nous allons calculer 10 000 distances et les utiliser pour déterminer la distance moyenne entre des points et la distance minimale entre des points dans chaque dimension (figure 12-2) :

```
import tqdm
dimensions = range(1, 101)
avg_distances = []
min_distances = []
random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
    distances = random_distances(dim, 10000) # 10 000 paires aléatoires
    avg_distances.append(sum(distances) / 10000) # suivre la moyenne
    min_distances.append(min(distances))         # suivre le minimum
```

Figure 12-2

La malédiction de la dimension

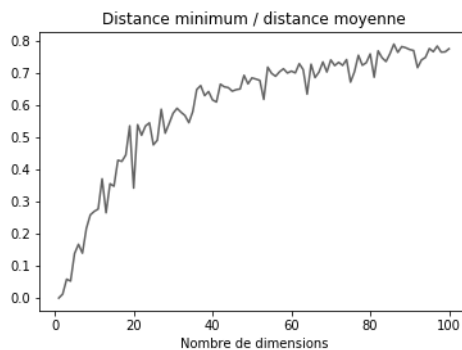


La distance moyenne entre les points augmente avec le nombre de dimensions. Néanmoins, c'est le ratio entre la distance la plus proche et la distance moyenne qui pose le plus gros problème (figure 12-3) :

```
min_avg_ratio = [min_dist / avg_dist
                 min_avg_ratio = [min_dist / avg_dist
```

Figure 12-3

Revoici la malédiction de la dimension



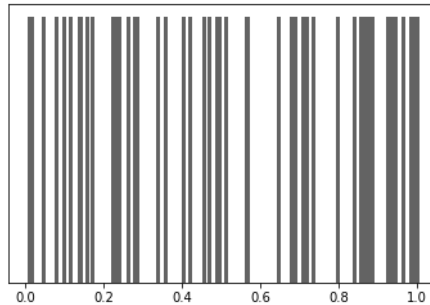
Avec les jeux de données faiblement dimensionnés, les points les plus proches ont tendance à être plus proches qu'en moyenne. Cependant, deux points sont proches uniquement s'ils sont proches dans chaque dimension. Toute dimension supplémentaire – bruit compris – est une occasion de plus pour chaque point de s'éloigner un peu de tous les autres. En présence d'un grand nombre de dimensions, il est probable que les points les plus proches ne seront pas beaucoup plus proches qu'en moyenne. Cela signifie que la proximité de deux points n'a pas beaucoup de signification (à moins que la structuration de vos données fasse que leur comportement soit le même que pour un nombre réduit de dimensions).

On peut aussi se représenter le problème en tenant compte de la faible densité des espaces fortement dimensionnés.

Si vous effectuez un tirage aléatoire de 50 nombres entre 0 et 1, vous obtiendrez sans doute un assez bon échantillon de l'intervalle d'une unité (figure 12-4).

Figure 12-4

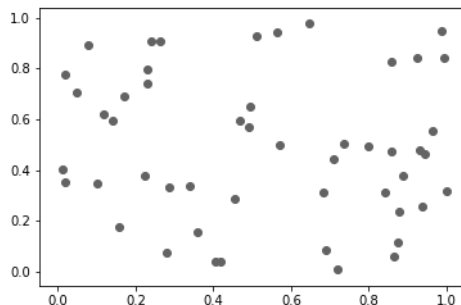
Cinquante points aléatoires dans une dimension



Si vous tirez 50 points dans le carré unitaire, vous obtiendrez une couverture moins dense (figure 12-5) ; et encore moins avec trois dimensions (figure 12-6).

Figure 12-5

Cinquante points aléatoires dans deux dimensions

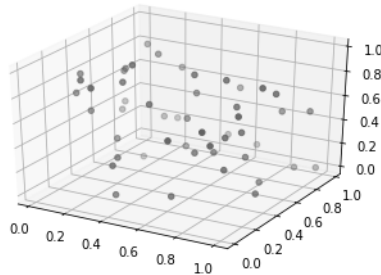


`matplotlib` n'a pas la capacité de représenter quatre dimensions, aussi nous arrêterons-nous là. Cependant, vous pouvez déjà constater qu'il commence à y avoir de grands espaces vides sans aucun point aux alentours. Avec davantage de dimensions, à moins que le nombre de données

n'augmente de manière exponentielle, ces régions vides représentent des régions éloignées de tous les points utiles à vos prédictions.

Figure 12-6

Cinquante points aléatoires
dans trois dimensions



En conclusion, si vous avez l'intention d'utiliser le modèle des plus proches voisins, il sera certainement préférable de procéder d'abord à une réduction du nombre de dimensions.

Pour aller plus loin

- `scikit-learn`¹ propose de nombreux modèles des plus proches voisins.

1. <https://scikit-learn.org/stable/modules/neighbors.html>

Classification naïve bayésienne

Il est bon que le cœur soit naïf et que l'esprit ne le soit pas.

Anatole France

Un réseau social ne sert à rien si on ne peut pas y développer des relations. DataSciencester propose une fonctionnalité prisée grâce à laquelle ses membres s'envoient des messages les uns aux autres. Pour la plupart, les utilisateurs sont des citoyens responsables qui n'envoient que des messages bienveillants et bien acceptés, mais un petit nombre de voyous persistent à inonder les autres de pourriels (*spam*) du style « comment devenir riche », « où acheter des médicaments sans ordonnance » et « où trouver des programmes de certification de data science ». Vos utilisateurs commencent à se plaindre, si bien que le responsable Messagerie vous a demandé d'utiliser la data science pour trouver un moyen de filtrer ces courriers indésirables.

Un filtre antispam particulièrement stupide

Soit un « univers » dans lequel on reçoit un message choisi au hasard parmi tous les messages possibles. Soit S l'événement *ce message est un spam* et B l'événement *ce message contient le terme bitcoin*. Le théorème de Bayes nous apprend que la probabilité que le message soit un spam sachant qu'il contient le mot « bitcoin » est la suivante :

$$P(S|B) = [P(B|S) P(S)] / [P(B|S) P(S) + P(B|\neg S) P(\neg S)]$$

Le numérateur représente la probabilité qu'un message soit un spam et qu'il contienne « bitcoin ». Le dénominateur représente seulement la probabilité qu'un message contienne

« bitcoin ». Vous pouvez ainsi envisager le résultat de ce calcul comme la proportion de messages qui sont du spam parmi tous ceux qui contiennent « bitcoin ».

Si nous avons une grande quantité de messages connus comme des spams et une grande quantité de messages connus comme n'en étant pas, il est facile d'estimer $P(B|S)$ et $P(B|?S)$. Si, en plus, nous supposons que tout message a la même probabilité d'être un spam ou un non-spam (de sorte que $P(S)=P(?S)=0.5$), alors :

$$P(S|B) = P(B|S) / [P(B|S) + P(B|?S)]$$

Par exemple, si le mot « bitcoin » se retrouve dans 50 % des spams mais dans seulement 1 % des non-spams, alors la probabilité qu'un message donné contenant « bitcoin » soit du spam s'exprime comme suit :

$$0.5 / (0.5 + 0.01) = 98 \%$$

Un filtre antispam plus élaboré

Imaginons que vous ayez un vocabulaire de plusieurs mots $m_1, \dots, m_n$. Pour respecter les conventions de la théorie des probabilités, nous écrivons x_i pour désigner l'événement *un message contient le mot m_i* . Imaginons également que, grâce à un processus non détaillé à ce stade, vous ayez pu déterminer une estimation de la probabilité $P(x_i|S)$ qu'un message de spam contienne le mot de rang i et une estimation similaire pour la probabilité $P(x_i|?S)$ qu'un message non-spam contienne le mot de rang i .

La clé de la classification naïve bayésienne consiste à faire l'hypothèse (énorme) que les présences ou les absences de ces mots sont indépendantes les unes des autres selon que le message est un spam ou pas. Intuitivement, cette hypothèse signifie que savoir qu'un certain message spam contient le mot « bitcoin » ne vous apporte aucune information sur le fait que ce même message contienne le mot « Rolex » :

$$P(x_1=x_1, \dots, x_n=x_n|S) = P(x_1=x_1|S) \times \dots \times P(x_n=x_n|S)$$

Il s'agit là d'une hypothèse extrême ; c'est la raison pour laquelle la technique est qualifiée de « naïve ». Si nous imaginons que notre vocabulaire se limite uniquement aux mots « bitcoin » et « Rolex » et qu'un message de spam contient *earn bitcoin* (« gagner des bitcoins ») et l'autre moitié *authentic Rolex* (« Rolex authentique »), alors la probabilité de Bayes naïf qu'un message de spam contienne à la fois les mots « bitcoin » et « Rolex » devient :

$$P(x_1=1, x_2=1|S) = P(x_1=1|S) P(x_2=1|S) = 0.5 \times 0.5 = 0.25$$

Nous avons ignoré volontairement l'information selon laquelle la présence des deux mots est simultanée. Malgré le caractère peu réaliste de cette hypothèse, ce modèle est souvent assez efficace. C'est pour cela qu'il est appliqué aux « vrais » filtres antispam.

Le raisonnement du théorème de Bayes auquel nous avons fait appel pour notre filtre *bitcoin* *seulement* nous dit aussi que nous pouvons calculer la probabilité qu'un message soit un spam à l'aide de l'équation suivante :

$$P(S|X=x) = P(X=x|S) / [P(X=x|S) + P(X=x|?S)]$$

L'hypothèse du classificateur naïf bayésien permet de calculer chacune des probabilités de la partie droite de l'équation simplement en multipliant les estimations unitaires des probabilités pour chaque mot du vocabulaire.

En pratique, on cherche à éviter de multiplier trop de probabilités à la fois pour éviter un problème appelé « sousassement » (ou dépassement de capacité inférieure), ou *underflow* en anglais : les ordinateurs gèrent mal les nombres à virgule flottante trop proches de zéro. En faisant appel à nos souvenirs d'algèbre, nous savons que $\log(ab) = \log a + \log b$ et que $\exp \log x = x$. Par conséquent, nous calculerons plutôt $p_1 \times \dots \times p_n$ sous la forme équivalente (cette formule est mieux adaptée à la virgule flottante) :

$$\exp (\log p_1 + ? + \log p_n)$$

À présent, il ne nous reste plus qu'à calculer les estimations de $P(x_i|S)$ et $P(x_i|?S)$, c'est-à-dire les probabilités qu'un spam (ou un non-spam) contienne le mot w_i . Si nous disposons d'une quantité raisonnable de messages d'apprentissage étiquetés comme spam ou non-spam, un premier essai évident consiste à estimer $P(x_i|S)$ simplement comme la fraction des messages de spam qui contiennent le mot w_i .

Cependant, cela pose un gros problème. Imaginons que, dans votre jeu d'apprentissage, le vocable « data » apparaisse seulement dans les messages non-spams. Nous estimerions alors $P(\text{data} | S) = 0$. Résultat, notre classificateur Bayes naïf affecterait systématiquement au spam la probabilité 0 pour tout message qui contient le mot « data », même si c'est un message comme *data on free bitcoin and authentic Rolex watches* (« données sur des bitcoins gratuits et des montres Rolex authentiques »). Pour éviter ce problème, il conviendra de procéder à un lissage.

En particulier, nous allons choisir un pseudo-compteur $-k-$ et estimer la probabilité de voir le i -ème mot dans un spam tel que :

$$P(x_i|S) = (k + \text{nombre de spams contenant } w_i) / (2k + \text{nombre total de spams})$$

Il en va de même pour $P(x_i|?S)$. En fait, quand nous calculons les probabilités de spam pour le i -ème mot, nous supposons que nous avons vu aussi k non-spams de plus qui contiennent le mot et k spams de plus qui ne le contiennent pas.

Par exemple, si « data » se rencontre dans 0/98 documents de spam et si k vaut 1, nous estimerons $P(\text{data}|S)$ à 1/100, ce qui permet à notre classificateur d'affecter aux messages contenant le mot « data » quelques probabilités de spam différentes de zéro.

La mise en œuvre

Nous avons maintenant toutes les pièces nécessaires pour construire notre classificateur. Commençons par créer une fonction simple pour découper les messages en mots distincts. Nous allons d'abord convertir les messages en lettres minuscules. Puis, avec `re.findall`, nous extrairons les « mots » composés de lettres, nombres et apostrophes. Enfin, nous utiliserons `set` pour récupérer les mots distincts.

```

from typing import Set
import re
def tokenize(text: str) -> Set[str]:
    text = text.lower() # convertir en minuscules
    all_words = re.findall("[a-z0-9']+", text) # extraire les mots
    return set(all_words) # supprimer les doublons

assert tokenize("Data Science is science") == {"data", "science", "is"}

```

Nous définissons également un type pour nos données d'apprentissage :

```

from typing import NamedTuple
class Message(NamedTuple):
    text: str
    is_spam: bool

```

Sachant que notre classificateur doit suivre les mots, comptages et étiquettes (*labels*) issus de nos données d'apprentissage, nous allons en faire une classe. Suivant la convention, nous désignerons les non-spams comme « hams ».

Le constructeur acceptera un seul paramètre ; le pseudo-compteur à utiliser lors du calcul de probabilités. Il initialise en outre un ensemble vide de mots, comptabilise la fréquence constatée pour chacun dans des spams et des hams, compte le nombre de messages auxquels s'applique son apprentissage :

```

from typing import List, Tuple, Dict, Iterable
import math
from collections import defaultdict
class NaiveBayesClassifieur:
    def __init__(self, k: float = 0.5) -> None:
        self.k = k # facteur de lissage
        self.tokens: Set[str] = set()
        self.token_spam_counts: Dict[str, int] = defaultdict(int)
        self.token_ham_counts: Dict[str, int] = defaultdict(int)
        self.spam_messages = self.ham_messages = 0

```

Ensuite, nous le dotons d'une méthode pour appliquer son apprentissage à un ensemble de messages. Tout d'abord, nous incrémentons les compteurs `spam_messages` et `ham_messages`. Ensuite, nous découpons le texte de chaque message et nous incrémentons le comptage `token_spam_counts` ou `token_ham_counts` selon le type de message :

```

def train(self, messages: Iterable[Message]) -> None:
    for message in messages:
        # compter les messages de chaque type
        if message.is_spam:
            self.spam_messages += 1
        else:
            self.ham_messages += 1

```



```
# compter les mots pour chaque type
for token in tokenize(message.text):
    self.tokens.add(token)
    if message.is_spam:
        self.token_spam_counts[token] += 1
    else:
        self.token_ham_counts[token] += 1
```

Finalement, nous voulons prédire $P(\text{spam} \mid \text{token})$. Comme nous l'avons vu auparavant, pour appliquer le théorème de Bayes, nous devons connaître $P(\text{token} \mid \text{spam})$ et $P(\text{token} \mid \text{ham})$ pour chaque mot dans le vocabulaire. Nous allons donc créer une fonction auxiliaire « privée » pour calculer ces éléments :

```
def _probabilities(self, token: str) -> Tuple[float, float]:
    """retourne P(token | spam) et P(token | ham)"""
    spam = self.token_spam_counts[token]
    ham = self.token_ham_counts[token]
    p_token_spam = (spam + self.k) / (self.spam_messages + 2 * self.k)
    p_token_ham = (ham + self.k) / (self.ham_messages + 2 * self.k)
    return p_token_spam, p_token_ham
```

Enfin, nous voici prêts à rédiger notre méthode de prédiction. Comme nous l'avons vu auparavant, plutôt que de multiplier de nombreuses petites probabilités, nous allons additionner les probabilités logarithmiques :

```
def predict(self, text: str) -> float:
    text_tokens = tokenize(text)
    log_prob_if_spam = log_prob_if_ham = 0.0
    # itérer sur chaque mot de notre vocabulaire
    for token in self.tokens:
        prob_if_spam, prob_if_ham = self._probabilities(token)
        # Si *token* apparaît dans le message,
        # ajouter le log de la probabilité de son occurrence.
        if token in text_tokens:
            log_prob_if_spam += math.log(prob_if_spam)
            log_prob_if_ham += math.log(prob_if_ham)
        # Sinon, ajouter le log de la probabilité de sa non-occurrence.
        # qui correspond à log(1-probabilité d'occurrence)
        else:
            log_prob_if_spam += math.log(1.0 - prob_if_spam)
            log_prob_if_ham += math.log(1.0 - prob_if_ham)

    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_ham = math.exp(log_prob_if_ham)
    return prob_if_spam / (prob_if_spam + prob_if_ham)
```

Nous disposons maintenant d'un classificateur.

Tester le modèle

Assurons-nous du bon fonctionnement de notre modèle en rédigeant quelques tests unitaires.

```
messages = [Message("spam rules", is_spam=True),
             Message("ham rules", is_spam=False),
             Message("hello ham", is_spam=False)]
model = NaiveBayesClassifier(k=0.5)
model.train(messages)
```

Tout d'abord, vérifions que les comptes sont bons :

```
assert model.tokens == {"spam", "ham", "rules", "hello"}
assert model.spam_messages == 1
assert model.ham_messages == 2
assert model.token_spam_counts == {"spam": 1, "rules": 1}
assert model.token_ham_counts == {"ham": 2, "rules": 1, "hello": 1}
```

À présent, faisons une prédiction. Nous allons en outre (laborieusement) passer en revue notre logique naïve de Bayes manuellement, et nous assurer que nous obtenons le même résultat :

```
text = "hello spam"
probs_if_spam = [
    (1 + 0.5) / (1 + 2 * 0.5), # "spam" (présent)
    1 - (0 + 0.5) / (1 + 2 * 0.5), # "ham" (non présent)
    1 - (1 + 0.5) / (1 + 2 * 0.5), # "rules" (non présent)
    (0 + 0.5) / (1 + 2 * 0.5)] # "hello" (présent)
]
probs_if_ham = [
    (0 + 0.5) / (2 + 2 * 0.5), # "spam" (présent)
    1 - (2 + 0.5) / (2 + 2 * 0.5), # "ham" (non présent)
    1 - (1 + 0.5) / (2 + 2 * 0.5), # "rules" (non présent)
    (1 + 0.5) / (2 + 2 * 0.5), )] # "hello" (présent)
]
p_if_spam = math.exp(sum(math.log(p) for p in probs_if_spam))
p_if_ham = math.exp(sum(math.log(p) for p in probs_if_ham))
# doit valoir environ 0.83
assert model.predict(text) == p_if_spam / (p_if_spam + p_if_ham)
```

Ce test est réussi. Aussi, il semble que notre modèle fasse ce que nous attendions de lui. Si vous examinez les probabilités réelles, les deux principaux « moteurs » sont le fait que notre message contienne `spam` (ce qui est le cas de notre unique spam d'apprentissage) et qu'il ne contienne pas `ham` (ce qui est le cas de nos deux hams d'apprentissage).

À présent, essayons cela sur des données réelles...

Utiliser le modèle

Le corpus public de SpamAssassin¹ constitue un bon jeu de données (bien qu'un peu ancien). Observons les fichiers préfixés par 20021010.

Voici un script qui les téléchargera et les décompactera dans le répertoire de votre choix (vous pouvez aussi le faire manuellement) :

```
from io import BytesIO
import requests # Pour télécharger les fichiers
import tarfile # qui sont au format .tar.bz2.
BASE_URL = "https://spamassassin.apache.org/old/publiccorpus"
FILES = ["20021010_easy_ham.tar.bz2",
         "20021010_hard_ham.tar.bz2",
         "20021010_spam.tar.bz2"]
# C'est ici que les données finissent,
# dans les sous-répertoires /spam, /easy_ham et /hard_ham.
# Changez ces destinations pour l'emplacement de votre choix.
OUTPUT_DIR = 'spam_data'
for filename in FILES:
    # utiliser des requêtes pour obtenir le contenu de chaque fichier
    content = requests.get(f"{BASE_URL}/{filename}").content
    # Regrouper les données pour les utiliser en tant que fichier
    fin = BytesIO(content)
    # extraire tous les fichiers vers le répertoire de sortie spécifié
    with tarfile.open(fileobj=fin, mode='r:bz2') as tf:
        tf.extractall(OUTPUT_DIR)
```

L'emplacement des fichiers peut changer (ce qui s'est produit entre la première et la deuxième édition de cet ouvrage). Dans ce cas, ajustez le script en conséquence.

Une fois les données téléchargées, vous devriez disposer de trois dossiers : `spam`, `easy_ham` et `hard_ham`. Ils contiennent de nombreux messages, chacun placé dans un fichier unique. Pour préserver une réelle simplicité, nous nous contenterons d'examiner les lignes d'objet de chaque message.

En passant les fichiers en revue, nous constatons qu'ils commencent tous par `Subject:` (objet: en français). Nous procéderons donc comme suit :

```
import glob, re
# modifiez le chemin d'accès avec la configuration de vos répertoires
path = 'spam_data/*/*'
data: List[Message] = []
# glob.glob retourne chaque nom de fichier qui correspond au masque de
# chemin fourni en entrée.
for filename in glob.glob(path):
    is_spam = "ham" not in filename
```

1. <https://spamassassin.apache.org/old/publiccorpus/>

```
# des caractères déchets sont présents dans les courriels
# L'instruction errors='ignore'
# les ignore au lieu de lever une exception.
with open(filename, errors='ignore') as email_file:
    for line in email_file:
        if line.startswith("Subject:"):
            subject = line.lstrip("Subject: ")
            data.append(Message(subject, is_spam))
            break # fini pour ce fichier
```

Nous pouvons désormais séparer les données en ensemble d'apprentissage et ensemble de test. Nous serons alors prêts à construire un classificateur :

```
import random
from scratch.machine_learning import split_data
random.seed(0) # juste pour avoir les mêmes réponses que moi
train_messages, test_messages = split_data(data, 0.75)
model = NaiveBayesClassifier()
model.train(train_messages)
```

Générons quelques prédictions et vérifions ce que fait notre modèle :

```
from collections import Counter
predictions = [(message, model.predict(message.text))
               for message in test_messages]
# Supposer que spam_probability>0.5 correspond à la prédiction de spam
# et compter les combinaisons de (actual is_spam, predicted is_spam).
confusion_matrix = Counter((message.is_spam, spam_probability > 0.5)
                           for message, spam_probability in predictions)
print(confusion_matrix)
```

Le résultat est de 87 vrais positifs (spam classé comme tel), 28 faux positifs (ham classé comme spam), 671 vrais négatifs (ham classé comme tel) et 39 faux négatifs (spam classé comme ham). Notre précision est donc de $87 / (87 + 28) = 76 \%$ et notre rappel est de $87 / (87 + 39) = 69 \%$. Ce n'est pas mal pour un modèle aussi simple. Nous pouvons probablement faire mieux si nous ne nous restreignons pas aux seules lignes d'objet.

Nous pouvons en outre inspecter la substantifique moelle du modèle pour déterminer quels mots sont les moins et les plus indicatifs d'un spam :

```
def p_spam_given_token(token: str, model: NaiveBayesClassifier) -> float:
    # Mieux vaut éviter d'appeler des méthodes privées, mais c'est pour
    # la bonne cause.
    prob_if_spam, prob_if_ham = model._probabilities(token)
    return prob_if_spam / (prob_if_spam + prob_if_ham)

words = sorted(model.tokens, key=lambda t: p_spam_given_token(t, model))
print("spammiest_words", words[-10:])
print("hammiest_words", words[:10])
```

Les mots les plus indésirables selon SpamAssassin sont notamment *sale*, *mortgage*, *money* et *rates* (« vente », « emprunt », « argent » et « taux » en français), tandis que les moins indésirables, à l'opposé, sont entre autres *spambayes*, *users*, *apt* et *perl* (« filtre antispam », « utilisateurs », *apt* et *perl*). D'après ce résultat, nous pouvons nous fier au fonctionnement de base de notre modèle.

Comment obtenir de meilleures performances ? Une solution évidente consiste à utiliser davantage de données pour l'apprentissage. Il existe aussi plusieurs manières d'améliorer le modèle. En voici quelques-unes que je vous invite à essayer.

- Utiliser le contenu du message et pas seulement la ligne contenant l'objet. Il convient alors de faire attention à la manière de traiter l'en-tête du message.
- Notre classificateur tient compte de chaque mot présent dans le jeu d'apprentissage, même lorsqu'il n'apparaît qu'une seule fois. Modifiez le classificateur selon un seuil minimal optionnel `min_count` et ignorez les mots dont le nombre d'occurrences est inférieur à ce minimum.
- Le système d'extraction des données (*tokeniser*) ignore la similitude entre les mots, par exemple *cheap* et *cheapest* (« pas cher » et « le moins cher »). Modifiez le classificateur en y intégrant une fonction facultative de racinisation (*stemming* en anglais) afin de convertir les mots en classes d'équivalence de mots. Voici un exemple d'une telle fonction très simple :

```
def drop_final_s(word):  
    return re.sub("s$", "", word)
```

Il est difficile de créer un racinisateur (*stemmer*) de qualité. Le plus souvent, on fait appel à celui de Martin Porter pour la langue anglaise ; le *Porter stemmer*.

- Bien que nos variables soient toutes de la forme *le message contient le mot w_i* , il n'y a aucune raison de nous limiter à ce cas. Dans notre mise en œuvre, nous pouvons ajouter des variables supplémentaires, telles que *le message contient un nombre*, en créant des unités lexicales fictives comme `contains:number` et en modifiant l'extraction de données afin qu'elle les émette quand cela est nécessaire.

Pour aller plus loin

- Pour approfondir vos connaissances sur les principes de construction des filtres à spam, vous pouvez consulter les articles de Paul Graham *A Plan for Spam*² et *Better Bayesian Filtering*³.
- `scikit-learn`⁴ contient un modèle appelé `BERNOULLINB`, qui met en œuvre l'algorithme naïf de Bayes que nous avons utilisé ici, ainsi que différentes variations de ce modèle.

2. <http://www.paulgraham.com/spam.html>

3. <http://www.paulgraham.com/better.html>

4. https://scikit-learn.org/stable/modules/naive_bayes.html

Régression linéaire simple

Comme la moralité, l'art consiste à savoir tirer une ligne quelque part.

G. K. Chesterton

Au chapitre 5, nous avons utilisé la corrélation pour mesurer la force de la relation linéaire entre deux variables. Pour la plupart des applications, il ne suffit pas de savoir qu'une relation linéaire existe. Il faut aussi en comprendre la nature. C'est là qu'intervient la régression linéaire simple.

Modèle

Comme vous vous en souvenez, nous recherchions la relation entre le nombre d'amis d'un utilisateur de DataSciencester et le temps qu'il passe sur le site chaque jour. Supposons que vous soyez convaincu qu'avoir plus d'amis conduit à passer plus de temps sur le site, plutôt qu'une autre des explications dont nous avons discuté.

La responsable Engagements vous demande de construire un modèle pour décrire cette relation. Comme vous avez trouvé une relation linéaire forte, le plus naturel est de construire un modèle linéaire. En particulier, vous faites l'hypothèse qu'il existe des constantes α et β telles que :

$$y_i = \beta x_i + \alpha + \epsilon_i$$

Dans cette expression, y_i est le nombre de minutes qu'un utilisateur i passe quotidiennement sur le site, x_i est le nombre de ses amis et ϵ est un terme d'erreur (on l'espère petit) pour représenter le fait qu'il existe d'autres facteurs non pris en compte par le modèle.

Supposons qu'on se fixe un α et un β donnés ; on peut alors faire des prédictions de manière simple avec la fonction suivante :

```
def predict(alpha: float, beta: float, x_i: float) -> float:
    return bêta * x_i + alpha
```

Toutefois, comment choisir α et β ? Chaque choix nous permet de prédire un résultat pour chaque entrée x_i . Comme nous connaissons le résultat réel y_i , nous pouvons calculer l'erreur pour chaque paire :

```
def erreur(alpha: float, beta: float, x_i: float, y_i: float) -> float:
    """L'erreur de la prédiction beta * x_i + alpha
    quand la valeur réelle est y_i."""
    return predict(alpha, beta, x_i) - y_i
```

Nous aimerions connaître l'erreur totale sur l'ensemble du jeu de données. Cependant, nous ne voulons pas simplement additionner les erreurs : si la prédiction pour x_1 est trop élevée et celle pour x_2 trop faible, les erreurs pourraient s'annuler.

Donc, à la place nous ajoutons les erreurs au carré :

```
from scratch.linear_algebra import Vector
def sum_of_sqerrors(alpha: float, beta: float,
                    x: Vector, y: Vector) -> float:
    return sum(error(alpha, beta, x_i, y_i) ** 2
                for x_i, y_i in zip(x, y))
```

La solution des *moindres carrés* consiste à choisir α et β pour rendre `sum_of_sqerrors` la plus petite possible.

En faisant appel au calcul (ou à une algèbre fastidieuse), on obtient les valeurs α et β qui minimisent l'erreur :

```
from typing import Tuple
from scratch.linear_algebra import Vector
from scratch.statistics import correlation, standard_deviation, mean
def least_squares_fit(x: Vector, y: Vector) -> Tuple[float, float]:
    """Soit deux vecteurs x et y,
    trouver les valeurs de moindres carrés de alpha et bêta."""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

Sans entrer dans les détails mathématiques, réfléchissons un peu : pourquoi pensez-vous qu'il puisse s'agir d'une solution raisonnable ? Le choix de α dit simplement que, lorsque nous connaissons la valeur moyenne de la variable indépendante x , nous pouvons prédire la valeur moyenne de la variable dépendante y .

Le choix de β signifie que, quand la variable d'entrée augmente de la valeur de l'écart type `standard_deviation(x)`, la prédiction augmente de `correlation(x,y) × standard_deviation(y)`. Dans le cas où x et y sont parfaitement corrélés, une augmentation d'un écart-type de x entraîne une augmentation d'un écart-type de y dans la prédiction.

S'ils sont parfaitement anti-corrélés, l'augmentation de x entraîne une diminution de la prédiction. Et quand la corrélation est égale à zéro, β vaut zéro, ce qui signifie que les modifications de x n'ont aucune incidence sur la prédiction.

Comme à l'accoutumée, rédigeons un test rapide pour ce code :

```
x = [i for i in range(-100, 110, 10)]
y = [3 * i - 5 for i in x]
# doit trouver que y = 3x - 5
assert least_squares_fit(x, y) == (-5, 3)
```

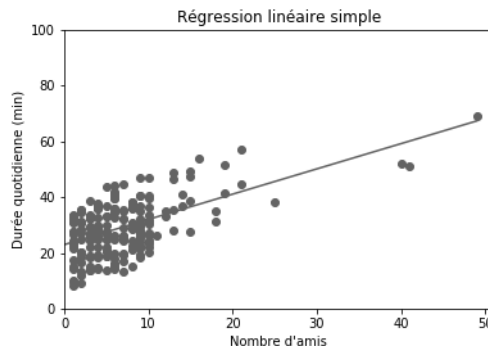
On peut facilement appliquer ces principes aux données du chapitre 5 (en excluant les valeurs extrêmes) :

```
from scratch.statistics import num_friends_good, daily_minutes_good
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
assert 22.9 < alpha < 23.0
assert 0.9 < beta < 0.905
```

Ceci nous donne les valeurs $\alpha = 22,95$ et $\beta = 0,904$. Notre modèle nous dit que nous nous attendons à ce qu'un utilisateur ayant n amis passe $22,95 + n \times 0,903$ minutes sur le site chaque jour. En d'autres termes, un utilisateur qui n'a aucun ami chez DataSciencester passera tout de même environ 23 minutes par jour sur le site. Et, pour chaque ami supplémentaire, il faut s'attendre à ce que l'utilisateur passe presque une minute de plus sur le site chaque jour.

Sur figure 14–1, nous avons dessiné la ligne de prédiction pour souligner à quel point le modèle correspond aux données observées.

Figure 14–1
Notre modèle linéaire simple



Il est évident qu'il nous faut une meilleure vérification de la bonne adéquation du modèle que celle obtenue par simple examen visuel du graphique. Une mesure classique est donnée par le

coefficient de détermination R^2 . Il mesure la fraction de la variation totale de la variable dépendante capturée par le modèle :

```
from scratch.statistics import de_mean
def total_sum_of_squares(y: Vector) -> float:
    """variation totale du carré des y_i à partir de leur espérance"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha: float, beta: float, x: Vector, y: Vector) -> float:
    """Fraction de la variation de y capturée par le modèle, qui est égale
    à 1 - la fraction de variation de y non capturée par le modèle."""
    return 1.0 - (sum_of_sqerrors(alpha, beta, x, y) / total_sum_of_squares(y))

rsq = r_squared(alpha, beta, num_friends_good, daily_minutes_good)
assert 0.328 < rsq < 0.330
```

À présent, rappelons-nous que nous avons choisi les α et β qui minimisent la somme des erreurs des prédictions au carré. Un autre modèle linéaire possible aurait pu être *toujours prédire moyenne(y)* (correspondant à $\alpha = \text{moyenne}(y)$ et $\beta = 0$), dont la somme des erreurs au carré est exactement égale à la somme totale des carrés ; son R^2 vaut zéro, ce qui indique que le modèle (à l'évidence dans ce cas) n'est pas plus performant que la simple prédiction de la moyenne.

Intuitivement, le modèle des moindres carrés doit être au moins aussi bon que le précédent, ce qui signifie que la somme des erreurs au carré est au plus égale à la somme totale des carrés et, donc, que le R^2 est au moins égal à 0. Et, comme la somme des erreurs au carré doit être au moins égale à 0, dans le meilleur des cas, le R^2 sera au maximum égal à 1.

Plus le nombre est élevé, meilleure est la correspondance entre notre modèle et les données. Ici, nous calculons un R^2 de 0,329 qui nous dit que notre modèle est à peine bon pour prédire les données et qu'il y a clairement d'autres facteurs en jeu.

Utiliser la descente de gradient

Si nous posons $\theta = [\alpha, \beta]$, nous pouvons résoudre notre problème par la descente de gradient :

```
import random
import tqdm
from scratch.gradient_descent import gradient_step
num_epochs = 10000
random.seed(0)
# choisir une valeur aléatoire pour démarrer
guess = [random.random(), random.random()]
learning_rate = 0.00001
with tqdm.trange(nb_étapes) as t:
    for _ in t:
```

```

alpha, beta = guess
# dérivée partielle de perte par rapport à alpha
grad_a = sum(2 * error(alpha, beta, x_i, y_i)
              for x_i, y_i in zip(num_friends_good, daily_minutes_good))
# dérivée partielle de perte par rapport à bêta
grad_b = sum(2 * error(alpha, beta, x_i, y_i) * x_i
              for x_i, y_i in zip(num_friends_good, daily_minutes_good))
# calculer la perte pour coller à la description tqdm
loss = sum_of_sqerrors(alpha, beta, num_friends_good, daily_minutes_good)
t.set_description(f"loss: {loss:.3f}")
# enfin actualiser la prévision
guess = gradient_step(guess, [grad_a, grad_b], -learning_rate)

# nous devrions obtenir pratiquement les mêmes résultats
alpha, beta = guess
assert 22.9 < alpha < 23.0
assert 0.9 < beta < 0.905

```

Si vous exécutez ce code, vous obtiendrez les mêmes valeurs pour α et β que lorsque nous utilisons la formule exacte.

Estimer le maximum de vraisemblance

Pourquoi choisir les moindres carrés ? Une des justifications réside dans l'estimation du *maximum de vraisemblance*. Imaginons un échantillon de données $v_1, \dots, v_n$ qui provient d'une distribution dépendante d'un paramètre inconnu θ :

$$p(v_1, \dots, v_n | \theta)$$

Si nous ignorons θ , nous pouvons l'approcher et le concevoir comme la vraisemblance de θ selon l'échantillon :

$$L(\theta | v_1, \dots, v_n)$$

Avec cette méthode, le θ le plus probable est la valeur qui maximise la fonction de vraisemblance ; c'est-à-dire, la valeur qui rend la donnée observée la plus probable. Dans le cas d'une distribution continue, pour laquelle nous disposons d'une fonction de distribution de probabilités plutôt que d'une fonction de masse probable, nous pouvons faire la même chose.

Revenons maintenant à la régression. Une hypothèse fréquente concernant le modèle de régression simple stipule que les erreurs de régression suivent une loi normale d'espérance 0 et d'écart-type (connu) σ . Dans ce cas, la fonction de vraisemblance de l'apparition d'une paire (x_i, y_i) devient :

$$L(\alpha, \beta | x_i, y_i, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(y_i - \alpha - \beta x_i)^2}{2 \sigma^2}\right)$$

La vraisemblance basée sur le jeu de données complet est le produit des vraisemblances individuelles, qui se trouve être la plus grande précisément quand α et β sont choisis tels qu'ils minimisent la somme des erreurs au carré. Donc, dans ce cas (et avec nos hypothèses), minimiser la somme des erreurs au carré équivaut à maximiser la vraisemblance des données observées.

Pour aller plus loin

Poursuivez vos lectures sur ce sujet avec la régression multiple présentée au chapitre 15 !

15

Régression multiple

Quand je considère un problème, je n'insère aucune variable qui n'aurait aucun effet sur lui.

Bill Parcells

Bien que la responsable soit très impressionnée par votre modèle de prédiction, elle pense que vous pouvez mieux faire. D'ailleurs, vous avez collecté d'autres données dans ce but : pour chacun de vos utilisateurs, vous savez maintenant combien d'heures il travaille par jour (*work hours*) et s'il a un doctorat (*PhD*). Vous aimeriez utiliser ces données supplémentaires pour améliorer votre modèle.

Donc, vous partez de l'hypothèse d'un modèle linéaire avec davantage de variables indépendantes :

Il est évident que le fait de posséder ou non un doctorat n'est pas une valeur numérique. Toutefois, nous pouvons créer (comme cela a été mentionné au chapitre 11) une variable factice (*dummy variable*) qui vaut 1 pour les utilisateurs ayant un doctorat et 0 pour les autres.

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \beta_3 \text{phd} + \epsilon$$

Modèle

Rappelez-vous, au chapitre 14, nous avions un modèle de la forme suivante :

$$y_i = \alpha + \beta x_i + \epsilon_i$$

Imaginons à présent que chaque entrée x_i n'est pas un nombre unique, mais un vecteur de k nombres, soit $x_{i1}, \dots, x_{ik}$. Le modèle de régression multiple fait l'hypothèse suivante :

$$y_i = \alpha + \beta_1 x_{i1} + \dots + \beta_k x_{ik} + \epsilon_i$$

En régression multiple, le vecteur de paramètres est en général désigné par β . Nous voulons introduire une constante ; nous ajouterons une colonne de 1 à nos données :

```
beta = [alpha, beta_1, ..., beta_k]
x_i = [1, x_i1, ..., x_ik]
```

Notre modèle devient alors le suivant :

```
from scratch.linear_algebra import dot, Vector
def predict(x: Vector, beta: Vector) -> float:
    """suppose que le premier élément de chaque x est 1"""
    return dot(x, beta)
```

Dans ce cas particulier, notre variable indépendante x sera une liste de vecteurs, chacun de la forme suivante :

```
[1, # constante
 49, # nombre d'amis
 4, # heures de travail par jour
 0] # n'a pas de doctorat
```

Autres hypothèses du modèle des moindres carrés

Deux hypothèses complémentaires sont nécessaires pour que ce modèle (et notre solution) ait un sens.

La première consiste à supposer que les colonnes de x sont *linéairement indépendantes* ; en d'autres termes, qu'il est impossible de décrire une d'entre elles comme la somme pondérée de certaines des autres. Si cette hypothèse est fausse, il est impossible d'estimer β . Pour visualiser un cas extrême, supposons que nous ayons un champ supplémentaire `num_acquaintances` dans nos données, exactement égal à `num_friends` pour chaque utilisateur.

Alors, quel que soit le β choisi, si nous ajoutons une quantité quelconque au coefficient `num_friends` et retranchons la même quantité au coefficient `num_acquaintances`, la prédiction du modèle reste inchangée. Cela signifie qu'il est impossible de trouver le coefficient de `num_friends`. En général, les violations de cette hypothèse ne sont pas aussi évidentes.

La deuxième hypothèse importante est que les colonnes de x sont toutes non corrélées avec les erreurs ϵ . Dans le cas contraire, nos estimations de β seraient systématiquement fausses.

Par exemple, au chapitre 14, nous avons construit un modèle qui prédit que chaque ami supplémentaire correspond à 0,90 minute de présence en plus sur le site chaque jour.

Imaginons qu'il soit également vrai que :

- les personnes qui travaillent le plus passent aussi moins de temps sur le site ;
- les personnes avec peu d'amis ont tendance à travailler plus longtemps.

Imaginons maintenant que le modèle « réel » soit le suivant :

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \epsilon$$

β_2 est négatif et les heures de travail et les amis sont corrélés positivement. Dans ce cas, quand nous minimisons les erreurs du modèle à une variable :

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \epsilon$$

Nous sous-estimons β_1 .

Pensez à ce qui arriverait si nous établissions des prédictions à partir du modèle à une variable avec la valeur « réelle » de β_1 (c'est-à-dire la valeur trouvée en minimisant les erreurs de ce que nous appelons le modèle « réel »). Les prédictions tendraient à être beaucoup trop importantes pour les utilisateurs qui travaillent beaucoup et un peu trop importantes pour ceux qui travaillent peu, car $\beta_2 < 0$ et nous avons « oublié » d'en tenir compte. Comme les heures de travail sont en corrélation positive avec le nombre d'amis, les prédictions tendent à être beaucoup trop importantes pour les utilisateurs qui ont beaucoup d'amis et seulement un peu trop importantes pour ceux qui ont peu d'amis.

Résultat, nous pouvons diminuer les erreurs (dans le modèle à une variable) en diminuant notre estimation de β_1 , ce qui signifie que la « minimisation » de l'erreur de β_1 est plus petite que la valeur « réelle ». En conclusion, dans ce cas, le modèle des moindres carrés à variable unique est biaisé dans le sens d'une sous-estimation de β_1 . De manière plus générale, chaque fois que les variables indépendantes sont corrélées avec les erreurs de cette façon, notre solution des moindres carrés donne une estimation biaisée de β_1 .

Ajuster le modèle

Comme nous l'avons fait précédemment avec le modèle linéaire simple, nous allons choisir β pour minimiser la somme des erreurs au carré. Il n'est pas facile de trouver une solution à la main ; aussi allons-nous faire appel à la descente de gradient. Commençons par minimiser la somme des erreurs au carré. La fonction d'erreur est pratiquement identique à celle utilisée au chapitre 14, à ceci près qu'au lieu d'attendre des paramètres `[alpha, beta]`, elle acceptera un vecteur de longueur arbitraire :

```
from typing import List
def error(x: Vector, y: float, beta: Vector) -> float:
    return predict(x, beta) - y

def squared_error(x: Vector, y: float, beta: Vector) -> float:
    return error(x, y, beta) ** 2

x = [1, 2, 3]
y = 30
beta = [4, 4, 4] # donc la prédiction = 4 + 8 + 12 = 24
assert error(x, y, beta) == -6
assert squared_error(x, y, beta) == 36
```

Si vous maîtrisez le calcul infinitésimal, vous pouvez calculer :

```
def sqerror_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    err = error(x, y, beta)
    return [2 * err * x_i for x_i in x]

assert sqerror_gradient(x, y, beta) == [-12, -24, -36]
```

Sinon, il va falloir me faire confiance.

À ce stade, nous sommes prêts à chercher le β optimal avec la descente du gradient : commençons par rédiger une fonction `least_squares_fit` capable d'accepter n'importe quel jeu de données :

```
import random
import tqdm
from scratch.linear_algebra import vector_mean
from scratch.gradient_descent import gradient_step
def least_squares_fit(xs: List[Vector],
                     ys: List[float],
                     learning_rate: float = 0.001,
                     num_steps: int = 1000,
                     taille_lot: int = 1) -> Vecteur:
    """Rechercher le bêta qui minimise la somme des erreurs au carré
    en considérant le modèle  $y = \text{dot}(x, \text{beta})$ ."""
    # commencer avec un essai aléatoire
    guess = [random.random() for _ in xs[0]]
    for _ in tqdm.trange(num_steps, desc="least squares fit"):
        for start in range(0, len(xs), batch_size):
            batch_xs = xs[start:start+batch_size]
            batch_ys = ys[start:start+batch_size]
            gradient = vector_mean([sqerror_gradient(x, y, guess)
                                   for x, y in zip(batch_xs, batch_ys)])
            guess = gradient_step(guess, gradient, -learning_rate)
    return guess
```

Nous pouvons appliquer ce code à nos données :

```
from scratch.statistics import daily_minutes_good
from scratch.gradient_descent import gradient_step
random.seed(0)
# j'ai utilisé un mode essai-erreur pour choisir num_iters et step_size
# cela devrait fonctionner un moment
learning_rate = 0.001
beta = least_squares_fit(inputs, daily_minutes_good, learning_rate, 5000, 25)

assert 30.50 < beta[0] < 30.70 # constante
assert 0.96 < beta[1] < 1.00 # nombre d'amis
assert -1.89 < beta[2] < -1.85 # heures de travail par jour
assert 0.91 < beta[3] < 0.94 # possède un doctorat
```


En pratique, personne n'estimerait une régression linéaire en recourant à une descente de gradient. On obtiendrait alors les coefficients exacts par des techniques algébriques linéaires qui sont au-delà de la portée du présent ouvrage. Si vous le faisiez, vous obtiendriez l'équation suivante :

$$\text{minutes} = 30.58 + 0.972 \text{ amis} - 1.87 \text{ heures de travail} + 0.923 \text{ doctorat}$$

Ce qui se rapproche beaucoup de ce que nous avons trouvé.

Interprétation du modèle

Vous pourriez interpréter les coefficients du modèle comme des estimations des impacts de chaque facteur... « toutes choses étant égales par ailleurs » : chaque ami supplémentaire représente une minute de plus passée sur le site chaque jour, chaque heure supplémentaire dans le temps de travail quotidien correspond à environ deux minutes de moins passées sur le site chaque jour et posséder un doctorat conduit à passer une minute de plus sur le site chaque jour.

En revanche, cela ne nous apprend rien (directement) sur les interactions entre les variables. Il est possible que le temps de travail ait un effet différent pour les personnes qui ont beaucoup d'amis et pour celles qui en ont peu. Ce modèle ne capture pas cette réalité. Pour traiter ce cas, une solution consiste à introduire une nouvelle variable qui sera égale au produit de *amis* et *heures travaillées*. Ainsi, le coefficient *heures travaillées* peut augmenter (ou diminuer) à mesure que le nombre d'amis augmente.

Il se peut aussi que plus vous avez d'amis, plus vous passez de temps sur le site, jusqu'à un certain point au-delà duquel c'est l'inverse (peut-être que, avec trop d'amis, l'expérience est simplement trop prenante). Nous pourrions essayer de capturer cette réalité dans notre modèle en ajoutant une nouvelle variable qui serait le carré du nombre d'amis.

Chaque fois que nous ajoutons une variable, nous devons nous poser la question suivante : son coefficient « a-t-il de l'importance ? » Il n'y a pas de limite au nombre de produits, logs, carrés et puissances plus élevées que nous pourrions ajouter.

Qualité de l'ajustement du modèle

Une fois de plus, nous allons examiner le R^2 , qui a désormais atteint 0,68 :

```
from scratch.simple_linear_regression import total_sum_of_squares
def multiple_r_squared(xs: List[Vector], ys: Vector,
                      beta: Vector) -> float:
    sum_of_squared_errors = sum(error(x, y, beta) ** 2
                                for x, y in zip(xs, ys))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(ys)

assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta) < 0.68
```

N'oubliez pas que le fait d'ajouter de nouvelles variables à une régression va obligatoirement augmenter le R^2 . Après tout, le modèle de régression simple n'est qu'un cas particulier du modèle de régression multiple pour lequel on aura établi les coefficients de *heures travaillées* et *doctorat* à 0. Le modèle de régression multiple optimal aura nécessairement, dans le pire des cas, une erreur aussi petite que celui-ci.

En conséquence, avec une régression multiple, nous devons aussi examiner les *erreurs standards* des coefficients, qui mesurent jusqu'à quel point nous sommes certains de nos estimations de chaque β_i . Il se peut que la régression, dans son ensemble, soit très bien ajustée à nos données. En revanche, si certaines des variables indépendantes sont corrélées (ou non pertinentes), leurs coefficients ne signifieront peut-être pas grand-chose.

La méthode classique pour mesurer ces erreurs commence par une nouvelle hypothèse : les erreurs ϵ_i sont des variables aléatoires normales indépendantes d'espérance 0 et d'écart-type commun (inconnu) σ . Dans ce cas, nous (ou plus probablement notre logiciel statistique) pouvons utiliser l'algèbre linéaire pour déterminer l'erreur standard de chaque coefficient. Plus celle-ci est grande, moins notre modèle est fiable pour ce coefficient. Malheureusement, nous ne sommes pas équipés pour effectuer ce calcul d'algèbre linéaire en partant de zéro.

Digression : l'amorce

Soit un échantillon de n données élémentaires générées suivant une distribution quelconque (qui nous est inconnue) :

```
data = get_sample(num_points=n)
```

Au chapitre 5, nous avons écrit une fonction pour calculer la médiane des données observées, que nous pouvons utiliser comme estimation de la médiane de la distribution.

Quelle confiance pouvons-nous accorder à notre estimation ? Si tous les points de données de l'échantillon sont très proches de 100, il est vraisemblable que la médiane sera elle aussi proche de 100. Si environ 50 % des points de données sont proches de 0 et l'autre moitié proche de 200, alors nous ne pouvons garantir la bonne estimation de la médiane avec autant de certitude.

Si nous pouvions avoir de nouveaux échantillons plusieurs fois, nous pourrions calculer la médiane de chacun d'eux et examiner la distribution de ces médianes. Malheureusement, la chose est en général impossible. À la place, nous pouvons « *amorcer* » (bootstrap) de nouveaux jeux de données en choisissant n points avec remplacement à partir de nos données.

Nous pouvons alors calculer les médianes de ces jeux de données de synthèse :

```
from typing import TypeVar, Callable
X = TypeVar('X') # type générique des données
Stat = TypeVar('Stat') # type générique pour "statistique"
def bootstrap_sample(data: List[X]) -> List[X]:
    """échantillon aléatoire de len(data) éléments avec remplacement"""
    return [random.choice(data) for _ in data]
```

```
def bootstrap_statistic(data: List[X],
                        stats_fn: Callable[[List[X]], Stat],
                        num_samples: int) -> List[Stat]:
    """Évalue stats_fn sur les échantillons d'amorçage
    à partir des données."""
    return [stats_fn(bootstrap_sample(data)) for _ in range(num_samples)]
```

Par exemple, soit les deux jeux de données suivants :

```
# 101 points tous très proches de 100
close_to_100 = [99.5 + random.random() for _ in range(101)]
# 101 points, 50 d'entre eux proches de 0, 50 d'entre eux proches de 200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

Les deux médianes sont très proches de 100. Toutefois, si vous examinez ce qui suit, vous verrez seulement des nombres réellement proches de 100 :

```
from scratch.statistics import median, standard_deviation
medians_close = bootstrap_statistic(close_to_100, median, 100)
```

Alors que si vous regardez le résultat suivant, vous verrez beaucoup de nombres proches de 0 et beaucoup d'autres proches de 200 :

```
medians_far = bootstrap_statistic(far_from_100, median, 100)
```

L'écart-type (`standard_deviation`) du premier jeu de médianes est proche de 0, alors que celui du second jeu est proche de 100 :

```
assert standard_deviation(medians_close) < 1
assert standard_deviation(medians_far) > 90
```

Ce cas extrême serait assez facile à comprendre en examinant les données une à une ; mais en général, ce n'est pas aussi évident.

Erreur standard des coefficients de régression

Nous pouvons adopter la même démarche pour estimer l'erreur standard de nos coefficients de régression. Nous prenons à plusieurs reprises un échantillon `bootstrap_sample` de nos données, à partir duquel nous estimons β . Si le coefficient correspondant à une des variables indépendantes (comme `num_friends`) varie peu dans les différents échantillons, alors on peut être certain que notre estimation est relativement ajustée. Au contraire, si le coefficient varie beaucoup, nous devons nous méfier de nos estimations.

La seule subtilité tient à ce que, avant échantillonnage, nous devons appliquer la fonction `zip` à nos données `x` et `y`, pour être sûrs que les valeurs correspondantes des variables indépendantes et dépendantes sont prélevées dans le même échantillon. En d'autres termes, `bootstrap_sample` retourne une liste de paires (x_i, y_i) que nous devons rassembler dans `x_sample` et `y_sample` :

```
from typing import Tuple
import datetime
def estimate_sample_beta(pairs: List[Tuple[Vector, float]]):
    x_sample = [x for x, _ in pairs]
    y_sample = [y for _, y in pairs]
    beta = least_squares_fit(x_sample, y_sample, learning_rate, 5000, 25)
    print("bootstrap sample", beta)
    return beta

random.seed(0) # pour obtenir les mêmes résultats que moi
# cela devrait prendre deux minutes
bootstrap_betas = bootstrap_statistic(list(zip(inputs, daily_minutes_good)),
                                     estimate_sample_beta, 100)
```

Après cela, nous pourrions estimer l'écart-type de chaque coefficient :

```
bootstrap_standard_errors = standard_deviation([beta[i] for beta in
bootstrap_betas])
for i in range(4)]
print(bootstrap_standard_errors)
# [1.272,      # constante, erreur mesurée = 1,19
#  0.103,      # nombre d'amis, erreur mesurée = 0.080
#  0.155,      # heures travaillées, erreur mesurée = 0.127
#  1.249]      # doctorat, erreur mesurée = 0.998
```

Nous obtiendrions probablement de meilleures estimations si nous collections plus de 100 échantillons et utilisons plus de 5 000 itérations pour estimer chaque β , mais nous n'avons pas toute la journée.

Nous pouvons utiliser cette approche pour tester des hypothèses comme *β_i est égal à zéro*. Avec cette hypothèse nulle $\beta_i = 0$ (et avec nos autres hypothèses sur la distribution de ϵ_i), le résultat statistique est le suivant :

$$t_j = \hat{\beta}_j / \hat{\sigma}_j$$

Ce résultat, qui représente notre estimation de β_j divisée par notre estimation de son erreur standard, suit une distribution *t* de Student avec $n-k$ degrés de liberté.

Si nous disposions d'une fonction `students_t_cdf`, nous pourrions calculer les valeurs-*p* pour chaque coefficient des moindres carrés. Cela indiquerait avec quelle vraisemblance on observerait une telle valeur si le coefficient réel valait zéro. Malheureusement, nous ne disposons pas d'une telle fonction (ce qui serait différent si nous partions de zéro).

Cependant, au fur et à mesure que les degrés de liberté deviennent plus grands, la distribution t se rapproche d'une loi normale standard. Dans une telle situation, quand n est beaucoup plus grand que k , nous pouvons utiliser `normal_cdf` en toute confiance :

```
from scratch.probability import normal_cdf
def p_value(beta_hat_j: float, sigma_hat_j: float) -> float:
    if beta_hat_j > 0:
        # Si le coefficient est positif, nous devons calculer deux fois la
        # probabilité de rencontrer une valeur encore plus grande.
        return 2 * (1 - fdc_normale(beta_hat_j / sigma_hat_j))
    else:
        # sinon, deux fois la probabilité de voir une valeur plus petite
        return 2 * fdc_normale(beta_hat_j / sigma_hat_j)

assert valeur_p(30.58, 1.27) < 0.001 # constante
assert valeur_p(0.972, 0.103) < 0.001 # nombre d'amis
assert valeur_p(-1.865, 0.155) < 0.001 # heures de travail
assert valeur_p(0.923, 1.249) > 0.4 # doctorat
```

Dans une situation différente, nous utiliserions probablement un logiciel statistique capable de calculer la distribution t ainsi que les erreurs standards exactes.

Alors que les coefficients affichent pour la plupart de toutes petites p -value (ce qui suggère qu'ils sont en fait non nuls), le coefficient concernant le doctorat n'est pas significativement différent de zéro, ce qui tend à prouver que ce coefficient est aléatoire plutôt que significatif.

Dans les scénarios de régression plus élaborés, on veut parfois tester des hypothèses plus complexes sur les données, telles que *au moins un des β_j est non nul* ou encore *β_1 égale β_2 et β_3 égale β_4* . Cela serait possible avec un test F , mais celui-ci dépasse malheureusement le cadre de ce livre.

Régulariser

En pratique, vous appliquerez souvent un modèle de régression linéaire à des jeux de données à grand nombre de variables. Par conséquent, vous rencontrerez quelques difficultés de plus. Tout d'abord, plus vous utiliserez de variables, plus vous risquerez de surajuster votre modèle par rapport au jeu d'apprentissage. Ensuite, plus vous aurez de coefficients non nuls, plus il sera difficile de leur donner un sens. Si l'objectif consiste à expliquer un phénomène, un modèle limité à trois facteurs est parfois plus utile qu'un modèle légèrement meilleur mais comptant des centaines de facteurs.

La *régularisation* est une méthode qui consiste à ajouter à l'erreur une pénalité qui augmente en même temps que β . Nous minimisons ainsi à la fois l'erreur et la pénalité. Et plus nous accordons d'importance à la pénalité, moins nous encourageons les coefficients élevés.

Par exemple, dans une régression d'arête (*ridge regression*), nous ajoutons une pénalité proportionnelle à la somme des carrés de β_i – sauf généralement à β_0 , la constante, que nous ne prenons pas en compte :

```
# alpha est un *hyperparamètre* contrôlant la sévérité de la pénalité
# Il est parfois appelé lambda, mais ce terme a déjà une signification
# en Python.
def ridge_penalty(beta: Vector, alpha: float) -> float:
    return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x: Vector,
                        y: float,
                        beta: Vector,
                        alpha: float) -> float:
    """estime l'erreur plus la pénalité d'arête sur beta"""
    return error(x, y, beta) ** 2 + ridge_penalty(beta, alpha)
```

Nous pouvons intégrer cette pénalité à la descente de gradient de la manière habituelle :

```
from scratch.linear_algebra import add
def ridge_penalty_gradient(beta: Vector, alpha: float) -> Vector:
    """gradient de la pénalité d'arête seulement"""
    return [0.] + [2 * alpha * beta_j for beta_j in beta[1:]]

def sqerror_ridge_gradient(x: Vector,
                          y: float,
                          beta: Vector,
                          alpha: float) -> Vector:
    """Le gradient correspondant à la i-ème erreur au carré
    incluant la pénalité d'arête."""
    return add(sqerror_gradient(x, y, beta),
               ridge_penalty_gradient(beta, alpha))
```

Il nous suffit alors de modifier la fonction `least_squares_fit` pour utiliser `sqerror_ridge_gradient` au lieu de `sqerror_gradient`.

Quand α est mis à zéro, il n'y plus aucune pénalité et nous obtenons le même résultat qu'auparavant :

```
random.seed(0)
beta_0 = least_squares_fit_ridge(inputs, daily_minutes_good, 0.0, # alpha
                                learning_rate, 5000, 25)
# [30.51, 0.97, -1.85, 0.91]
assert 5 < dot(beta_0[1:], beta_0[1:]) < 6
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0) < 0.69
```

À mesure que nous augmentons α , l'ajustement du modèle se dégrade, mais β diminue :

```
beta_0_1 = least_squares_fit_ridge(inputs, daily_minutes_good, 0.1, # alpha
                                  learning_rate, 5000, 25)
# [30.8, 0.95, -1.83, 0.54]
assert 4 < dot(beta_0_1[1:], beta_0_1[1:]) < 5
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_0_1) < 0.69

beta_1 = least_squares_fit_ridge(inputs, daily_minutes_good, 1, # alpha
                                 learning_rate, 5000, 25)
# [30.6, 0.90, -1.68, 0.10]
assert 3 < dot(beta_1[1:], beta_1[1:]) < 4
assert 0.67 < multiple_r_squared(inputs, daily_minutes_good, beta_1) < 0.69

beta_10 = least_squares_fit_ridge(inputs, daily_minutes_good, 10, # alpha
                                  learning_rate, 5000, 25)
# [28.3, 0.67, -0.90, -0.01]
assert 1 < dot(beta_10[1:], beta_10[1:]) < 2
assert 0.5 < multiple_r_squared(inputs, daily_minutes_good, beta_10) < 0.6
```

En particulier, le coefficient concernant le doctorat s'estompe à mesure que nous augmentons la pénalité ; ce qui est en accord avec notre résultat précédent, selon lequel il n'est pas significativement différent de zéro.

Remarque

En général, on adapte l'échelle des données avant d'utiliser cette méthode. Après tout, si vous échangez des années d'expérience contre des siècles d'expérience, le coefficient des moindres carrés augmente d'un facteur 100 et se trouve donc brusquement pénalisé davantage, bien que le modèle reste le même.

Une autre méthode est la *régression du Lasso*, qui utilise la pénalité suivante :

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

Alors que la pénalité d'arête tend à minimiser les valeurs des coefficients au global, celle du Lasso tend à forcer les coefficients à se rapprocher de zéro. Elle est donc très adaptée pour les modèles d'apprentissage peu denses. Malheureusement, elle n'est pas exploitable dans le cadre d'une descente du gradient ; ce qui signifie que nous ne pourrions pas la résoudre à partir de rien.

Pour aller plus loin

- La régression fait l'objet d'une théorie riche et complète. C'est un autre domaine que vous devriez explorer en lisant un manuel ou au moins des articles sur Wikipédia.
- `scikit-learn` dispose d'un module `linear_model`¹ qui fournit un modèle `LinearRegression` similaire au nôtre, des régressions d'arête et du Lasso, ainsi que d'autres types de régularisation.
- `statsmodels`² est un autre module Python qui contient (entre autres trésors) des modèles de régression linéaire.

1. https://scikit-learn.org/stable/modules/linear_model.html

2. <https://www.statsmodels.org/stable/index.html>

16

Régression logistique

Beaucoup pensent que la ligne est ténue entre le génie et la folie. Je ne pense pas qu'il s'agisse d'une ligne ténue ; je pense qu'il s'agit d'un gouffre béant.

Bill Bailey

Au chapitre 1, nous nous sommes intéressés brièvement à la question de prédire quels utilisateurs de DataSciencester souscrivaient un compte payant premium. Nous allons revenir sur cette question dans ce chapitre.

Problème

Nous disposons d'un jeu de données anonymisées concernant 200 utilisateurs avec, pour chacun, le salaire, le nombre d'années d'expérience et l'information sur le choix ou non d'un compte premium (figure 16–1). Comme pour toutes les variables de type catégorie, nous représenterons la variable de dépendance sous la forme de 0 (pas de compte premium) ou de 1 (compte premium).

Comme d'habitude, nos données sont dans une matrice dont chaque ligne est une liste [experience, salary, paid_account]. Convertissons-les au format qui nous intéresse :

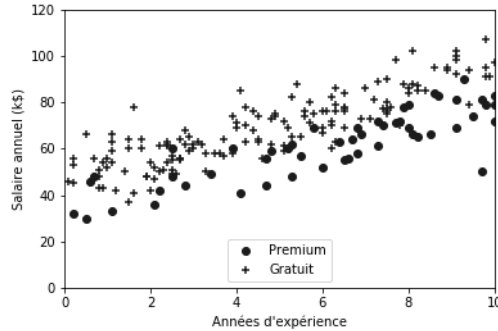
```
# chaque élément est [1, expérience, salaire]
xs = [[1.0] + row[:2] for row in data]
# chaque élément est premium
ys = [row[2] for row in data]
```

Une première tentative évidente serait d'utiliser une régression linéaire afin de trouver le meilleur modèle :

$$\text{paid account} = \beta + \beta_1 \text{experience} + \beta_2 \text{salary} + \varepsilon$$

Figure 16–1

Utilisateurs payants et non payants

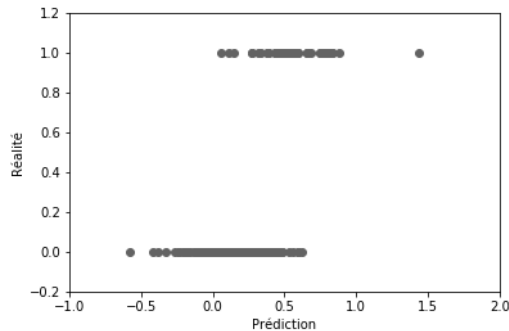


Rien ne nous empêche de choisir cette modélisation. Les résultats sont présentés en figure 16–2.

```
from matplotlib import pyplot as plt
from scratch.working_with_data import rescale
from scratch.multiple_regression import least_squares_fit, predict
from scratch.gradient_descent import gradient_step
learning_rate = 0.001
rescaled_xs = rescale(xs)
beta = least_squares_fit(rescaled_xs, ys, learning_rate, 1000, 1)
# [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_xs]
plt.scatter(predictions, ys)
plt.xlabel("Prédiction")
plt.ylabel("Réalité")
plt.show()
```

Figure 16–2

Utilisation de la régression linéaire pour prédire les comptes premium



Cependant, cette méthode apporte son lot de problèmes immédiats.

- Nous aimerions que nos prédictions se présentent sous la forme 0 ou 1 pour marquer l'appartenance à une classe. C'est parfait si elles figurent entre 0 et 1, car nous pouvons les interpréter comme des probabilités : ainsi, un résultat de 0,25 signifie 25 % de chance d'être membre payant. Toutefois, les résultats du modèle linéaire peuvent être constitués de grands nombres positifs ou négatifs, qui ne sont pas évidents à interpréter. En effet, dans notre cas, beaucoup de prédictions étaient négatives.
- Le modèle de régression linéaire suppose que les erreurs ne sont pas corrélées avec les colonnes de x . Dans notre cas, le coefficient pour la variable `experience` est de 0,43, indiquant que davantage d'expérience conduit à une plus grande probabilité de souscription d'un compte premium. Notre modèle donne donc comme résultats des valeurs très élevées pour les personnes très expérimentées. Or, nous savons que les valeurs effectives ne dépassent pas 1, ce qui signifie que, nécessairement, les très grands résultats (et donc les très grandes valeurs pour l'expérience) correspondent à de très grandes valeurs négatives du terme d'erreur. Comme c'est le cas, notre estimation de β est biaisée.

Ce que nous aimerions, à la place, c'est que les grandes valeurs positives de `dot(x_i, beta)` correspondent à des probabilités proches de 1 et que les grandes valeurs négatives correspondent à des probabilités proches de 0. Il est possible d'y parvenir, à condition d'appliquer une autre fonction au résultat.

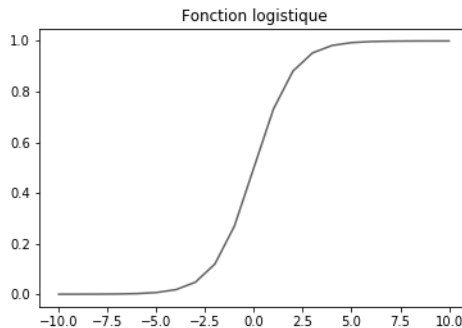
Fonction logistique

Dans le cas de la régression logistique, nous faisons appel à la fonction logistique (figure 16–3) :

```
def logistic(x: float) -> float:  
    return 1.0 / (1 + math.exp(-x))
```

Figure 16–3

La fonction logistique



Quand ses entrées sont élevées et positives, elle tend vers 1. Et quand elles sont élevées et négatives, elle tend vers 0. De plus, sa dérivée possède la propriété intéressante de pouvoir s'écrire sous la forme suivante :

```
def logistic_prime(x: float) -> float:
    y = logistic(x)
    return y * (1 - y)
```

Nous l'utiliserons bientôt pour ajuster un modèle dans lequel f est la fonction logistique :

$$y_i = f(x_i\beta) + \varepsilon_i$$

Vous vous souvenez probablement que la régression linéaire ajuste le modèle en minimisant la somme des carrés des erreurs, ce qui équivaut à choisir le β qui maximise la vraisemblance des données.

Ici, les deux ne sont pas équivalents, donc nous utiliserons la descente du gradient pour maximiser directement la vraisemblance. En d'autres termes, nous devons calculer la fonction de vraisemblance et son gradient.

Pour un β donné, notre modèle dit que chaque y_i devrait être égal à 1 avec la probabilité $f(x_i\beta)$ et à 0 avec la probabilité $1-f(x_i\beta)$.

En particulier, la densité de probabilité de y_i peut s'écrire comme suit :

$$p(y_i|x_i, \beta) = f(x_i\beta)^{y_i} (1 - f(x_i\beta))^{1-y_i}$$

En effet, si y_i est égal à 0, elle vaut :

$$1 - f(x_i\beta)$$

Et, si y_i est égal à 1, elle vaut :

$$f(x_i\beta)$$

Il s'avère qu'il est plus facile de maximiser le logarithme de la fonction de vraisemblance :

$$\log L(\beta|x_i, y_i) = y_i \log f(x_i\beta) + (1 - y_i) \log(1 - f(x_i\beta))$$

Comme $\log$ est une fonction strictement croissante, tout formule qui maximise le $\log$ de la vraisemblance maximise aussi la fonction de vraisemblance et vice versa. Sachant qu'une descente de gradient minimise les choses, nous travaillerons effectivement avec le $\log$ de vraisemblance négatif. En effet, maximiser la vraisemblance revient à minimiser le négatif :

```
import math
from scratch.linear_algebra import Vector, dot
def _negative_log_likelihood(x: Vector, y: float, beta: Vector) -> float:
    """le log de vraisemblance négatif d'un point de données particulier"""
    if y == 1:
        return -math.log(logistic(dot(x, beta)))
    else:
        return -math.log(1 - logistic(dot(x, beta)))
```

Si nous supposons que des points de données distincts sont indépendants les uns des autres, la fonction de vraisemblance globale est simplement le produit des fonctions de vraisemblance individuelles. Cela signifie que le `log` de la vraisemblance globale est la somme des `log`s des fonctions de vraisemblance individuelles :

```
from typing import List
def negative_log_likelihood(xs: List[Vector],
                           ys: List[float],
                           beta: Vector) -> float:
    return sum(_negative_log_likelihood(x, y, beta)
               for x, y in zip(xs, ys))
```

Et maintenant, un peu de calcul pour obtenir le gradient :

```
from scratch.linear_algebra import vector_sum
def _negative_log_partial_j(x: Vector, y: float,
                           beta: Vector, j: int) -> float:
    """La dérivé partielle de rang j pour un point de données distinct.
    Ici i est l'index du point de données."""
    return -(y - logistic(dot(x, beta))) * x[j]

def _negative_log_gradient(x: Vector, y: float, beta: Vector) -> Vector:
    """le gradient d'un point de données distinct"""
    return [_negative_log_partial_j(x, y, beta, j)
            for j in range(len(beta))]

def negative_log_gradient(xs: List[Vector],
                          ys: List[float],
                          beta: Vector) -> Vector:
    return vector_sum([_negative_log_gradient(x, y, beta)
                       for x, y in zip(xs, ys)])
```

Nous avons désormais toutes les pièces nécessaires.

Appliquer le modèle

Séparons nos données en jeu d'apprentissage et jeu de test :

```
from scratch.machine_learning import train_test_split
import random
import tqdm
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_xs, ys, 0.33)
learning_rate = 0.01
# choisir un point de départ aléatoire
beta = [random.random() for _ in range(3)]
with tqdm.trange(5000) as t:
    for étape in t:
        gradient = negative_log_gradient(x_train, y_train, beta)
```

```
beta = gradient_step(beta, gradient, -learning_rate)
loss = negative_log_likelihood(x_train, y_train, beta)
t.set_description(f"loss: {loss:.3f} beta: {beta}")
```

Après quoi nous constatons que `beta` vaut approximativement :

```
# [-2.0, 4.7, -4.5]
```

Ce sont là les coefficients pour les données après changement d'échelle (*rescaled*), mais nous pouvons revenir aux données d'origine :

```
from scratch.working_with_data import scale
means, stdevs = scale(xs)
beta_unscaled = [(beta[0]
                  - beta[1] * means[1] / stdevs[1]
                  - beta[2] * means[2] / stdevs[2]),
                 beta[1] / stdevs[1],
                 beta[2] / stdevs[2]]
# [8.9, 1.6, -0.000288]
```

Malheureusement, ces coefficients ne sont pas aussi aisés à interpréter que ceux de la régression linéaire classique. Toutes choses étant égales par ailleurs, une année d'expérience de plus ajoute 1,6 en entrée de la fonction logistique et 10 000 \$ de salaire de plus enlèvent 2,88.

Cependant, l'impact sur le résultat dépend aussi des autres entrées. Si `dot(beta, x_i)` est déjà grand (correspondant à une probabilité proche de 1), l'augmenter, même de manière significative, n'aura pas beaucoup d'impact sur la probabilité. S'il est proche de 0, une toute petite augmentation risque de considérablement augmenter la probabilité.

Tout ce que nous pouvons affirmer, toutes choses étant égales par ailleurs, c'est que les personnes les plus expérimentées auront probablement plus souvent un compte payant et les personnes aux salaires les plus élevés auront probablement moins souvent un compte payant. C'était déjà plus ou moins apparent sur le graphique des données.

Bien ajuster le modèle

Nous n'avons pas encore utilisé les données de test que nous avons mises de côté. Que se passe-t-il si nous faisons une prédiction de compte payant dès que la probabilité dépasse 0,5 ?

```
true_positives = false_positives = true_negatives = false_negatives = 0
for x_i, y_i in zip(x_test, y_test):
    prediction = logistic(dot(beta, x_i))
    if y_i == 1 and prediction >= 0.5: # TP : payant prédit payant
        true_positives += 1
    elif y_i == 1: # FN : payant prédit gratuit
        false_negatives += 1
```

```

elif prediction >= 0.5: # FP : gratuit prédit payant
    false_positives += 1
else: # TN : gratuit prédit gratuit
    true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)

```

Nous obtenons une précision de 75 % (quand nous prédisons un compte payant, nous avons raison dans 75 % des cas) et un rappel de 80 % (quand un utilisateur a un compte payant, nous prédisons un compte payant dans 80 % des cas).

Nous pouvons aussi représenter l'adéquation des prédictions avec la réalité (figure 16–4), ce qui confirme que la performance du modèle est bonne :

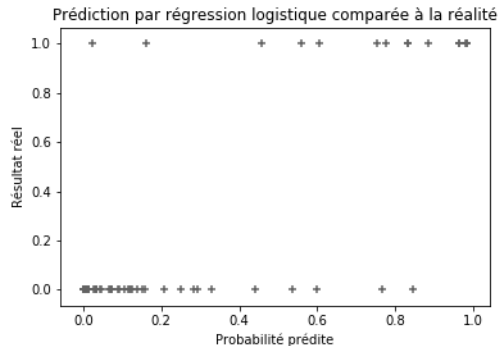
```

predictions = [logistic(dot(beta, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test, marker='+')
plt.xlabel("Probabilité prédite")
plt.ylabel("Résultat réel")
plt.title("Prédiction par régression logistique comparée à la réalité")
plt.show()

```

Figure 16–4

Prédiction par régression logistique comparée à la réalité

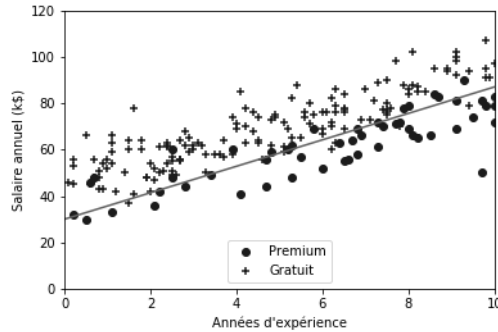


Séparateurs à vaste marge

Le jeu de points pour lequel $\text{dot}(\text{beta}, x_i)$ vaut 0 marque la limite entre nos classes. Nous pouvons réaliser un graphique pour suivre exactement le comportement de notre modèle (figure 16–5).

Figure 16-5

Utilisateurs payants et non payants
avec limite de décision.

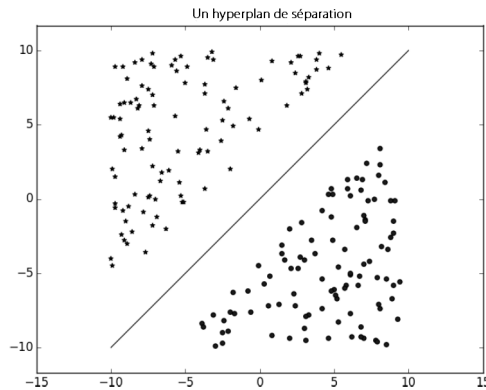


Cette limite correspond à un hyperplan qui divise l'espace des paramètres en deux demi-espaces correspondant aux prédictions *premium* et *non premium*. Nous découvrons qu'il s'agit d'un effet secondaire de l'identification du modèle logistique le plus probable.

Une autre approche de la classification consiste à simplement identifier l'hyperplan qui sépare « le mieux » les classes dans les données d'apprentissage. C'est l'idée qui sous-tend le *séparateur à vaste marge* (ou machine à vecteurs de support). Ce dernier identifie l'hyperplan qui maximise la distance au point le plus proche dans chaque classe (figure 16-6).

Figure 16-6

Un hyperplan de séparation

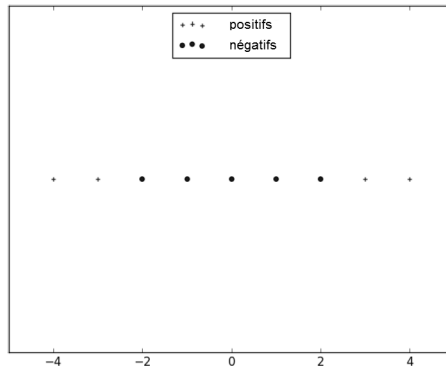


La recherche de l'hyperplan est un problème d'optimisation qui requiert des techniques trop avancées pour nous. Il se peut par ailleurs qu'un tel hyperplan n'existe pas. Dans notre jeu de données *qui paye ?*, il n'existe aucune ligne séparant parfaitement les comptes premium des comptes gratuits.

Il est parfois possible de contourner le problème en transformant les données dans un espace de plus grandes dimensions. Par exemple, prenons le jeu de données à une dimension présenté en figure 16-7.

Figure 16–7

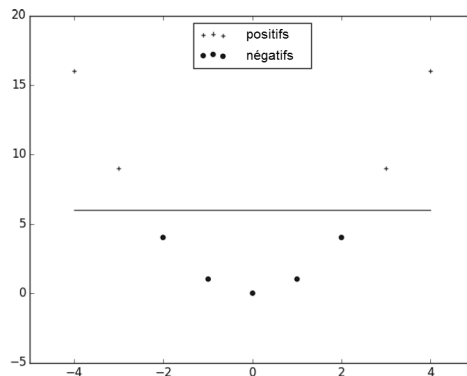
Un jeu de données à une dimension non séparable



Il est évident qu'il n'existe aucun hyperplan séparant les exemples positifs des exemples négatifs. Cependant, voyez ce qui se passe lorsqu'on projette ce jeu de données dans deux dimensions en faisant correspondre le point x au vecteur (x, x^2) . Tout d'un coup, il devient possible de trouver un hyperplan qui sépare les données (figure 16–8).

Figure 16–8

Le jeu de données devient séparable dans un espace de plus grandes dimensions.



On appelle cela l'astuce du noyau (*kernel trick* en anglais) car, plutôt que de projeter réellement les points dans un espace de plus grandes dimensions (ce qui peut se révéler très coûteux s'il y a beaucoup de points et si la correspondance est difficile), nous préférons utiliser une fonction « noyau » pour calculer les produits scalaires dans l'espace de grandes dimensions et ensuite les utiliser pour trouver un hyperplan. Il est difficile (et sans doute pas très futé) de faire appel aux séparateurs à vaste marge sans s'appuyer sur un logiciel d'optimisation spécialisé écrit par des personnes suffisamment expertes ; aussi, nous arrêterons notre traitement à ce stade.

Pour aller plus loin

- `scikit-learn` propose des modules pour la régression logistique¹ et les séparateurs à vaste marge (SVM)².
- `LIBSVM`³ est la mise en œuvre de SVM utilisée par `scikit-learn`. Son site web propose une documentation utile et variée sur cette famille d'algorithmes.

1. https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
2. <https://scikit-learn.org/stable/modules/svm.html>
3. <https://www.csie.ntu.edu.tw/~cjlin/libsvm/>

Arbres de décision

Un arbre est un mystère incompréhensible.

Jim Woodring

Le responsable Talents de DataSciencester a interrogé de nombreux candidats à l'embauche postulant sur le site, avec des résultats mitigés. Il a collecté des jeux de données reprenant plusieurs attributs (qualitatifs) des candidats ainsi que l'opinion sur la prestation du candidat en entretien. Il vous demande de construire un modèle pour identifier les candidats qui réaliseront un bon entretien et lui éviter de perdre son temps à tous les recevoir.

C'est là un joli cas d'école pour utiliser un arbre de décision, autre outil de modélisation prédictive du data scientist.

Qu'est-ce qu'un arbre de décision ?

Un *arbre de décision* (*decision tree*) est une structure arborescente qui représente différents choix possibles (ou cheminements possibles) et un résultat pour chacun d'eux.

Si vous déjà joué à des jeux du type « devine à quoi je pense »¹, vous connaissez déjà les arbres de décision. En voici un exemple :

- « Je pense à un animal.
- Est-ce qu'il a plus de cinq pattes ?
- Non.

1. https://en.wikipedia.org/wiki/Twenty_Questions

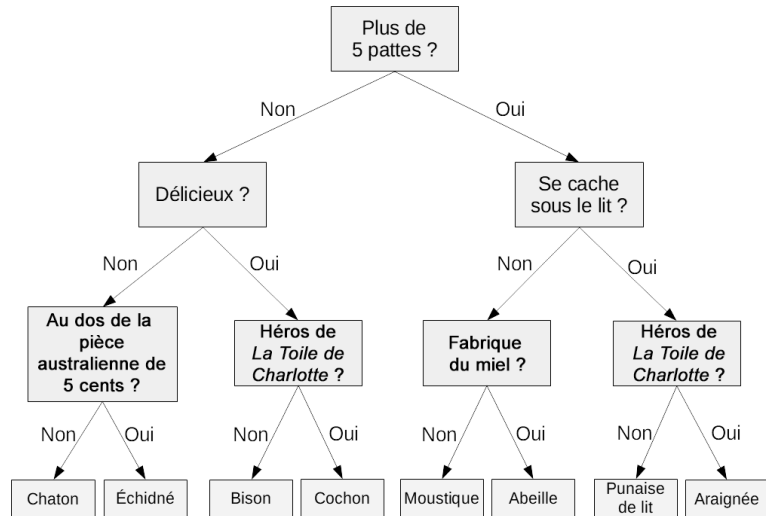
- Est-ce qu'il se mange ?
- Non.
- Est-ce qu'il apparaît sur le dos de la pièce australienne de 5 cents ?
- Oui.
- Est-ce un échidné ?
- Oui, c'est ça ! »

Voici le cheminement suivi dans un arbre de décision idiosyncratique (et pas vraiment exhaustif) du jeu (figure 17–1).

pas plus de 5 pattes -> ne se mange pas -> sur la pièce de 5 cents -> échidné !

Figure 17–1

Un arbre de décision
« devine à quel animal je pense »



Les arbres de décision ont de nombreux atouts. Ils sont faciles à comprendre et à interpréter et le processus pour arriver à une prédiction est complètement transparent. À la différence des autres modèles que nous avons examinés jusque-là, les arbres de décision peuvent manipuler des attributs de type numérique (nombre de pattes) ou catégoriel (se mange ou ne se mange pas). Ils peuvent même classer les données pour lesquelles il manque un attribut.

En revanche, trouver un arbre de décision « optimal » pour un jeu de données d'apprentissage constitue un problème particulièrement difficile (au sens de la complexité théorique). Nous réglerons cette question en essayant de construire un arbre acceptable, plutôt qu'un arbre optimal, bien qu'il s'agisse, malgré tout, d'une tâche considérable pour de grands jeux de données. Plus important encore, on peut très vite (et cela est fortement déconseillé) se retrouver avec un arbre de décision surajusté vis-à-vis des données d'apprentissage et qui ne se généralise pas bien aux données nouvelles. Voyons comment procéder pour éviter cela.

Pour la plupart, les experts divisent les arbres de décision en arbres de classification (avec comme résultats finaux des catégories) et en arbres de régression (produisant des résultats numériques). Dans ce chapitre, nous nous intéresserons aux arbres de classification. Nous utiliserons l'algorithme ID3 pour construire un arbre de décision à partir d'un ensemble de données étiquetées, ce qui devrait nous aider à comprendre le mécanisme. Et, pour faciliter les choses, nous nous limiterons à des problèmes à réponses binaires, tels que *dois-je embaucher ce candidat ?*, *dois-je présenter la publicité A ou B à ce visiteur de mon site web ?* ou encore *la nourriture que j'ai trouvée dans le réfrigérateur au bureau va-t-elle me rendre malade ?*

Entropie

Pour construire un arbre de décision, nous devons sélectionner les questions à poser et décider de l'ordre dans lequel nous les présenterons. Chaque étape comprend alors des possibilités que nous avons éliminées et d'autres que nous avons conservées. Après avoir appris qu'un animal ne peut avoir plus de cinq pattes, nous avons éliminé la possibilité que ce soit une sauterelle, mais pas celle que ce soit un canard. Chaque question sépare les possibilités restantes en fonction de la réponse apportée.

Dans l'absolu, nous aimerions choisir des questions dont les réponses fournissent beaucoup d'informations sur ce que notre arbre devrait prédire. S'il existe une question simple pour une réponse binaire *oui/non* pour laquelle *oui* correspond toujours à des résultats vrais et *non* à des résultats faux (ou vice versa), alors cette question est à retenir. À l'inverse, une question *oui/non* pour laquelle aucune des réponses ne vous donne beaucoup de nouvelles informations pour orienter la prédiction n'a sans doute pas beaucoup d'intérêt.

Cette notion de « quantité d'information apportée » se mesure avec l'*entropie*. Vous avez probablement déjà entendu ce terme dans le sens de désordre. Nous l'utiliserons pour représenter l'incertitude attachée aux données.

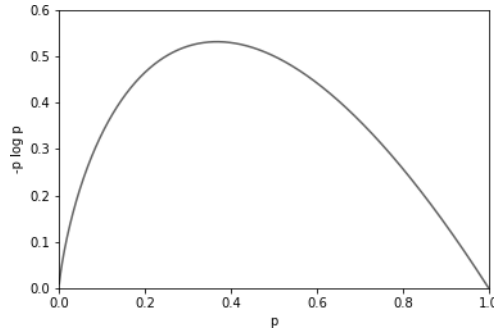
Soit un ensemble S de données dans lequel chaque membre est étiqueté comme appartenant à une des classes de la liste finie de classes $c_1, \dots, c_n$. Si toutes les données élémentaires appartiennent à une seule classe, alors il n'y a plus vraiment d'incertitude, ce qui revient à dire que l'entropie est faible. Si les données élémentaires sont distribuées régulièrement entre les classes, elles contiennent beaucoup d'incertitude, ce qui revient à dire que leur entropie est élevée.

En termes mathématiques, si p_i est la proportion des données étiquetées comme appartenant à la classe c_i , nous définirons l'entropie comme suit :

$$H(S) = -p_1 \log_2 p_1 - \dots - p_n \log_2 p_n$$

Avec pour convention standard que $0 \log_2 0 = 0$.

Sans trop entrer dans les détails, il suffit de garder en tête que chaque terme $-p_i \log_2 p_i$ est positif et proche de zéro précisément quand p_i est proche soit de 0, soit de 1 (figure 17-2).

Figure 17-2Graphique de $-p \log_2 p$ 

Cela veut dire que l'entropie est faible quand chaque p_i est proche de 0 ou 1 (autrement dit, quand les données s'inscrivent pour la plupart dans une classe unique) et élevée quand de nombreuses occurrences de p_i ne sont pas proches de 0 (c'est-à-dire que les données sont réparties dans plusieurs classes). C'est précisément le comportement que nous recherchons.

On peut aisément résumer tout cela dans une fonction :

```
from typing import List
import math
def entropy(class_probabilities: List[float]) -> float:
    """Soit une liste de probabilités de répartition de classes,
    calculer l'entropie."""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p > 0) # ignore les probabilités à zéro

assert entropy([1.0]) == 0
assert entropy([0.5, 0.5]) == 1
assert 0.81 < entropy([0.25, 0.75]) < 0.82
```

Nos données sont des paires (entrée, étiquette), ce qui signifie qu'il est nécessaire que nous calculions les probabilités de répartition des classes nous-mêmes. Vous remarquerez que nous ne cherchons pas à savoir quelle étiquette (`label`) est associée à quelle probabilité, mais que nous nous intéressons uniquement à la nature (`value`) des probabilités :

```
from typing import Any
from collections import Counter
def class_probabilities(labels: List[Any]) -> List[float]:
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labels: List[Any]) -> float:
    return entropy(class_probabilities(labels))

assert data_entropy(['a']) == 0
assert data_entropy([True, False]) == 1
assert data_entropy([3, 4, 4, 4]) == entropy([0.25, 0.75])
```

Entropie d'une partition

Jusqu'à présent, nous avons déterminé l'entropie (pensez « incertitude ») d'un seul jeu de données étiquetées. Maintenant, chaque étape de l'arbre de décision nécessite de poser une question dont la réponse sépare les données dans un ou (espérons-le) plusieurs sous-ensembles. Par exemple, notre question *plus de cinq pattes ?* sépare les animaux en deux groupes : ceux qui ont plus de cinq pattes (comme les araignées) et ceux qui en ont moins (comme les échidnés).

De même, nous aimerions connaître l'entropie qui résulte du partitionnement d'un jeu de données d'une manière spécifique. Nous voulons une partition à faible entropie si elle sépare les données en sous-ensembles qui présentent, eux-mêmes, une faible entropie (donc qui sont hautement certains), à entropie élevée si elle contient des sous-ensembles (de grande taille) à forte entropie (donc hautement incertains).

Par exemple, ma question à propos de la pièce australienne (bien que j'aie eu de la chance) était assez stupide, car elle séparait les animaux restants à ce stade en $S_1 = \{\text{échidné}\}$ et $S_2 = \{\text{tout le reste}\}$ avec S_2 à la fois de grande dimension et de forte entropie. S_1 n'a pas d'entropie, mais il représente une toute petite fraction des « classes restantes ». Mathématiquement, si nous séparons les données S en sous-ensembles $S_1, \dots, S_m$ contenant des proportions $q_1, \dots, q_m$ des données et si nous calculons l'entropie de la partition sous la forme d'une somme pondérée, nous obtenons alors la relation suivante :

$$H = q_1 H(S_1) + \dots + q_m H(S_m)$$

Nous pouvons la mettre en œuvre comme suit :

```
def partition_entropy(subsets: List[List[Any]]) -> float:
    """Renvoie l'entropie de cette partition des données dans
       des sous-ensembles."""
    total_count = sum(len(subset) for subset in subsets)
    return sum(data_entropy(subset) * len(subset) / total_count
               for subset in subsets)
```

Remarque

Un des problèmes de cette méthode vient du fait que, si on partitionne les données selon un attribut comportant beaucoup de valeurs différentes, nous serons conduits à une entropie très basse à cause du surajustement. Par exemple, vous travaillez dans une banque et vous souhaitez construire un arbre de décision pour prédire lesquels de vos clients risquent de ne pas pouvoir rembourser leur crédit immobilier. Vous disposez pour cela d'un jeu de données historiques comme données d'apprentissage. Imaginons de plus que ce jeu de données d'apprentissage contient le numéro de sécurité sociale (NSS) de chaque membre. Si vous partitionnez vos données selon le NSS, vous obtiendrez des sous-ensembles d'une seule personne, dont chacun aura nécessairement une entropie de 0. Un modèle de type NSS sera certainement impossible à généraliser au-delà du jeu de données d'apprentissage. Pour cette raison, lorsque vous construisez des arbres de décision, vous devriez sans doute éviter autant que possible (ou alors les regrouper si c'est pertinent) les attributs qui présentent un grand nombre de valeurs possibles.

Construire un arbre de décision

Le responsable vous fournit les données recueillies en entretien. Conformément à vos spécifications, ces données sont des `NamedTuple` des attributs pertinents pour chaque candidat : niveau (`level`), langage privilégié (`lang`), degré d'activité sur Twitter (`tweets`), s'il possède ou non un doctorat et si l'entretien s'est bien déroulé (`did_well`) :

```
from typing import NamedTuple, Optional
class Candidate(NamedTuple):
    level: str
    lang: str
    tweets: bool
    phd: bool
    did_well: Optional[bool] = None # autorise des données non étiquetées

# level lang tweets phd did_well
inputs = [Candidate('Senior', 'Java', False, False, False),
           Candidate('Senior', 'Java', False, True, False),
           Candidate('Mid', 'Python', False, False, True),
           Candidate('Junior', 'Python', False, False, True),
           Candidate('Junior', 'R', True, False, True),
           Candidate('Junior', 'R', True, True, False),
           Candidate('Mid', 'R', True, True, True),
           Candidate('Senior', 'Python', False, False, False),
           Candidate('Senior', 'R', True, False, True),
           Candidate('Junior', 'Python', True, False, True),
           Candidate('Senior', 'Python', True, True, True),
           Candidate('Mid', 'Python', False, True, True),
           Candidate('Mid', 'Java', True, False, True),
           Candidate('Junior', 'Python', False, True, False)]
```

Notre arbre se compose de nœuds décisionnels (qui posent une question et nous orientent différemment selon la réponse) et de nœuds terminaux ou nœuds feuilles ou feuilles (qui nous fournissent une prédiction). L'algorithme ID3 va nous permettre de le construire facilement. Il opère de la manière suivante. Soit des données avec étiquettes et une liste d'attributs à leur affecter :

- si les données ont toutes la même étiquette, créer un nœud terminal pour prédire cette étiquette et arrêter ;
- si la liste d'attributs est vide (il n'y a plus de questions à poser), créer un nœud terminal pour prédire l'étiquette la plus courante et arrêter ;
- sinon, essayer de séparer les données en fonction de chacun des attributs ;
- choisir la partition à plus faible entropie ;
- ajouter un nœud décisionnel en s'appuyant sur l'attribut choisi ;
- répéter sur tous les sous-ensembles résultant du partitionnement à l'aide des attributs restants.

Cet algorithme est dit « glouton » car, à chaque étape, il choisit la meilleure option immédiate dans le contexte considéré. Pour un jeu de données particulier, il existe peut-être un meilleur arbre avec une première question pire en apparence. Si c'est le cas, cet algorithme ne va pas la trouver. Néanmoins, il est assez facile à comprendre et à mettre en œuvre, ce qui en fait un bon point de départ pour explorer les arbres de décision.

Effectuons manuellement ces étapes sur le jeu de données issu des entretiens. Le jeu de données contient des étiquettes `False` et `True` et nous disposons de quatre attributs pour la séparation. Notre première étape va consister à trouver la partition avec l'entropie la plus faible. Nous commencerons par écrire une fonction de partitionnement :

```
from typing import Dict, TypeVar
from collections import defaultdict
T = TypeVar('T') # type générique des entrées
def partition_by(inputs: List[T], attribute: str) -> Dict[Any, List[T]]:
    """partitionner les entrées en listes selon l'attribut spécifié"""
    partitions: Dict[Any, List[T]] = defaultdict(list)
    for input in inputs:
        key = getattr(input, attribute) # valeur de l'attribut spécifié
        partitions[key].append(input) # ajouter à la partition correcte
    return partitions
```

Il nous faut aussi une autre fonction qui l'utilisera pour calculer l'entropie :

```
def partition_entropy_by(inputs: List[Any],
                        attribute: str,
                        label_attribute: str) -> float:
    """calculer l'entropie correspondant à la partition donnée"""
    # les partitions sont constituées de nos entrées
    partitions = partition_by(inputs, attribute)
    # mais partition_entropy a seulement besoin des étiquettes
    labels = [[getattr(input, label_attribute) for input in partition]
               for partition in partitions.values()]
    return partition_entropy(labels)
```

Il nous suffira alors de trouver la partition avec la plus faible entropie pour le jeu de données complet :

```
for key in ['level', 'lang', 'tweets', 'phd']:
    print(key, partition_entropy_by(inputs, key, 'did_well'))

assert 0.69 < partition_entropy_by(inputs, 'level', 'did_well') < 0.70
assert 0.86 < partition_entropy_by(inputs, 'lang', 'did_well') < 0.87
assert 0.78 < partition_entropy_by(inputs, 'tweets', 'did_well') < 0.79
assert 0.89 < partition_entropy_by(inputs, 'phd', 'did_well') < 0.90
```

L'entropie la plus faible est trouvée lorsque la séparation se fait sur l'attribut `level`, de sorte qu'il faut créer un sous-arbre pour chaque valeur de niveau (`level`) possible. Tout candidat de

niveau intermédiaire (`Mid`) est étiqueté `True`, ce qui veut dire que le sous-arbre `Mid` est un simple nœud terminal qui prédit `True`. Pour les candidats `Senior`, nous avons à la fois des valeurs `True` et `False`, donc il faut encore séparer les données :

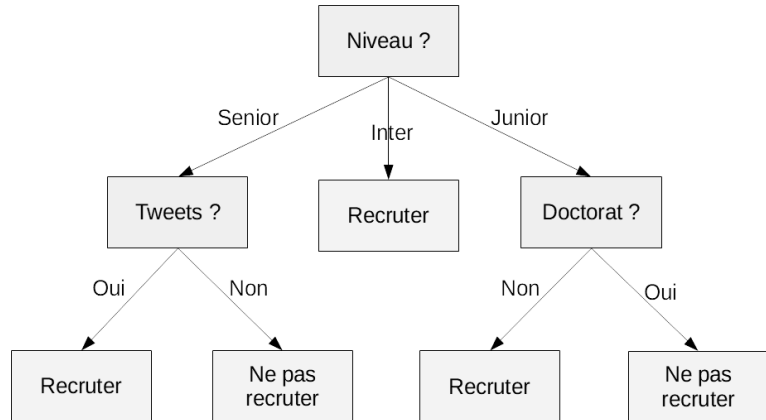
```
senior_inputs = [input for input in inputs if input.level == 'Senior']
assert 0.4 == partition_entropy_by(senior_inputs, 'lang', 'did_well')
assert 0.0 == partition_entropy_by(senior_inputs, 'tweets', 'did_well')
assert 0.95 < partition_entropy_by(senior_inputs, 'phd', 'did_well') < 0.96
```

Cela nous montre que notre prochaine séparation se fera sur `tweets` avec pour conséquence une partition à entropie 0. Pour les candidats de niveau `Senior`, l'affectation à l'attribut `tweets` de `Oui` donne toujours pour résultat `True` ; l'affectation à l'attribut `tweets` de `Non` conduit toujours à `False`.

Pour finir, si nous appliquons la même méthode aux candidats `Junior`, nous devons effectuer la partition sur l'attribut `phd`, après quoi nous trouvons que l'absence de doctorat résulte toujours en `True` et la détention d'un doctorat résulte toujours en `False`.

Figure 17-3

L'arbre de décision du recrutement



Synthèse

Maintenant que nous avons vu comment fonctionne l'algorithme, nous aimerions le mettre en œuvre de manière plus générale. Nous devons donc décider comment nous voulons représenter les arbres. Nous utiliserons ici la représentation la plus légère possible. Un arbre est constitué d'éléments de l'un des deux types suivants :

- **Leaf** (feuille, qui prédit une valeur unique) ;
- **Division** (qui contient un attribut à diviser, des sous-arbres pour des valeurs spécifiques de cet attribut et, éventuellement, une valeur par défaut à utiliser si nous identifions une valeur inconnue).

```

from typing import NamedTuple, Union, Any

class Leaf(NamedTuple):
    value: Any

class Split(NamedTuple):
    attribute: str
    subtrees: dict
    default_value: Any = None

DecisionTree = Union[Leaf, Split]

```

Avec cette représentation, notre modèle d'arbre de recrutement ressemble à ceci :

```

hiring_tree = Split('level', { # tout d'abord, tenir compte de "level"
    'Junior': Split('phd', { # si level == "Junior", voir ensuite "phd"
        False: Leaf(True), # si "phd" correspond à False, prédire True
        True: Leaf(False) # si "phd" correspond à True, prédire False
    }),
    'Mid': Leaf(True), # si 'level'=='Mid', prédire True
    'Senior': Split('tweets', { # si 'level'=='Senior', examiner 'tweets'
        False: Leaf(False), # si 'tweets'==False, prédire False
        True: Leaf(True)} # si 'tweets'==True, prédire True
    })

```

Il nous reste à traiter les cas particuliers des entrées qui ont un attribut manquant ou une valeur inattendue. Que doit faire notre arbre de recrutement s'il rencontre un candidat de niveau stagiaire (`Intern`) ? Nous gérerons ce cas en renseignant l'attribut `default_value` au moyen de l'étiquette la plus courante.

Avec une telle représentation, nous pouvons classer les entrées de la manière suivante :

```

def classify(tree: DecisionTree, input: Any) -> Any:
    """classer l'entrée à l'aide de l'arbre de décision"""
    # si c'est un nœud terminal, retourner la valeur
    if isinstance(tree, Leaf):
        return tree.value
    # Sinon cet arbre se compose d'un attribut à considérer pour aiguiller
    # la décision (diviser)
    # et d'un dictionnaire dont les clés sont des valeurs de cet attribut
    # et dont les valeurs sont les sous-arbres à envisager ensuite.
    subtree_key = getattr(input, tree.attribute)
    if subtree_key not in tree.subtrees: # pas de branche correspondante
        return tree.default_value # retourne la valeur par défaut
    subtree = tree.subtrees[subtree_key] # choisir la branche adaptée
    return classify(subtree, input) # et l'utiliser pour classer l'entrée

```

Il ne nous reste plus qu'à construire la représentation arborescente de nos données d'apprentissage :

```
def build_tree_id3(inputs: List[Any],
                  split_attributes: List[str],
                  target_attribute: str) -> DecisionTree:
    # compter les labels cible
    label_counts = Counter(getattr(input, target_attribute)
                           for input in inputs)
    most_common_label = label_counts.most_common(1)[0][0]
    # en présence d'une unique valeur, la prédire
    if len(label_counts) == 1:
        return Leaf(most_common_label)
    # s'il ne reste plus d'attributs, retourner l'étiquette majoritaire
    if not split_attributes:
        return Leaf(most_common_label)
    # sinon, diviser selon le meilleur attribut

    def split_entropy(attribute: str) -> float:
        """fonction d'aide pour rechercher le meilleur attribut"""
        return partition_entropy_by(inputs, attribute, target_attribute)

    best_attribute = min(split_attributes, key=split_entropy)
    partitions = partition_by(inputs, best_attribute)
    new_attributes = [a for a in split_attributes
                      if a != best_attribute]
    # construire les branches de manière récursive
    subtrees = {attribute_value : build_tree_id3(subset,
                                                  new_attributes,
                                                  target_attribute)
                 for attribute_value, subset in partitions.items()}
    return Split(best_attribute, subtrees, default_value=most_common_label)
```

Dans cet arbre, chaque nœud terminal contient soit uniquement des entrées `True`, soit uniquement des entrées `False`. Cela signifie que l'arbre a réalisé une prédiction parfaite sur les données d'apprentissage. Nous pouvons aussi l'appliquer à de nouvelles données qui ne figuraient pas dans le jeu d'apprentissage :

```
tree = build_tree_id3(inputs,
                      ['level', 'lang', 'tweets', 'phd'],
                      'did_well')
# devrait prédire True
assert classify(tree, Candidate("Junior", "Java", True, False))
# devrait prédire False
assert not classify(tree, Candidate("Junior", "Java", True, True))
```

Testons des données nouvelles avec des valeurs inattendues :

```
# devrait prédire True
assert classify(tree, Candidate("Intern", "Java", True, True))
```

Remarque

Notre objectif principal étant de montrer comment construire un arbre de décision, nous avons utilisé le jeu de données complet. Comme toujours, si nous voulions réellement créer un modèle performant pour un cas réel, nous aurions collecté davantage de données en les séparant en sous-ensembles d'apprentissage, de validation et de test.

Forêts aléatoires

Étant donné que les arbres décisionnels peuvent s'ajuster très finement à vos données d'apprentissage, il n'est pas surprenant qu'ils présentent une tendance au surajustement. Pour éviter cela, faisons appel à la technique des *forêts aléatoires*, ou *forêts d'arbres décisionnels*. Cette technique consiste à construire plusieurs arbres décisionnels et à les combiner. S'il s'agit d'arbres de classification, nous les laissons voter ; s'il s'agit d'arbres de régression, nous moyennons leurs prédictions.

Notre procédé d'élaboration des arbres était déterministe. Comment donc créer des arbres aléatoires ? Une solution consiste à mobiliser les *techniques d'amorce* (ou *bootstrap*) (chapitre 15). Plutôt que d'entraîner chaque arbre sur toutes les entrées de l'ensemble d'apprentissage, nous le faisons sur les résultats de `bootstrap_sample(inputs)`. Comme chaque arbre est construit à partir de données différentes, il sera différent des autres. Avantage collatéral, il est possible, sans tomber dans le surajustement, d'utiliser les données non échantillonnées pour tester chaque arbre ; cela signifie que vous pouvez vous permettre d'utiliser toutes vos données en tant que jeu d'apprentissage si vous mesurez intelligemment la performance. Cette technique est connue sous le nom d'*ensachage* (*bagging*) ou d'*agrégation d'amorce* (*bootstrap aggregating*).

Une autre source de hasard provient du changement d'utilisation de la fonction `best_attribute` pour séparer les données. Plutôt que d'examiner tous les attributs restants, nous en choisissons d'abord un sous-ensemble aléatoire et nous réalisons la séparation sur le meilleur :

```
# s'il existe trop peu de candidats pour la division, les examiner tous
if len(split_candidates) <= self.num_split_candidates:
    sampled_split_candidates = split_candidates
# sinon prendre un échantillon au hasard
else:
    sampled_split_candidates = random.sample(split_candidates,
self.num_split_candidates)

# choisir maintenant le meilleur attribut parmi ces candidats seulement
best_attribute = min(sampled_split_candidates, key=split_entropy)
partitions = partition_by(inputs, best_attribute)
```

Il s'agit là d'un exemple d'une technique plus générale appelée *apprentissage ensembliste* (*ensemble learning*) et, suivant laquelle nous combinons plusieurs modèles d'apprentissage faibles (*weak learners* – modèles à biais élevé et variance faible) afin de produire un modèle général puissant.

Pour aller plus loin

- `scikit-learn` propose de nombreux modèles d'arbres de décision². Il propose aussi un module `ensemble`³ qui inclut le modèle `RandomForestClassifier` ainsi que d'autres méthodes ensemblistes.
- `XGBoost`⁴ est une bibliothèque d'arbres de décision d'apprentissage dopés par une approche en gradient. Ces arbres ont tendance à l'emporter lors des compétitions d'apprentissage automatique de type Kaggle.
- Dans ce chapitre, nous n'avons fait que survoler les arbres de décision et leurs algorithmes. Wikipédia constitue un bon point de départ pour explorer plus largement ce sujet⁵.

2. <https://scikit-learn.org/stable/modules/tree.html>

3. <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble>

4. <https://xgboost.ai/>

5. [https://fr.wikipedia.org/wiki/Arbre_de_d%C3%A9cision_\(apprentissage\)](https://fr.wikipedia.org/wiki/Arbre_de_d%C3%A9cision_(apprentissage))

Réseaux neuronaux

J'aime le non-sens, ça réveille les neurones.

Dr. Seuss

Un *réseau neuronal artificiel* (ou simplement réseau de neurones) est un modèle prédictif qui reproduit le fonctionnement du cerveau. Le cerveau est considéré comme un ensemble de neurones connectés les uns aux autres. Chaque neurone examine les sorties des autres neurones, qui deviennent ses entrées, effectue un calcul, puis se déclenche (si le calcul excède un certain seuil) ou non (dans le cas contraire).

De la même manière, les réseaux neuronaux artificiels se composent de neurones artificiels qui réalisent le même genre de calculs sur leurs entrées. Les réseaux neuronaux sont capables de résoudre de nombreux problèmes tels que la reconnaissance de l'écriture manuscrite ou la reconnaissance faciale. Ils sont largement utilisés en *apprentissage profond* (*deep learning*), un des champs d'étude les plus prometteurs de la data science. Cependant, les réseaux neuronaux sont pour la plupart des « boîtes noires ». L'inspection de leurs caractéristiques ne permet pas de comprendre comment ils résolvent un problème. Et les grands réseaux neuronaux sont parfois difficiles à entraîner. Pour la plupart des problèmes que vous rencontrerez en tant que data scientist, ils ne constitueront donc probablement pas le bon choix. Un jour, quand vous vous attellerez à la construction d'une intelligence artificielle qui mettra en avant la « singularité » (qui permettra de faire éclore l'*Humanité 2.0* de Ray Kurzweil), ce sera peut-être différent.

Perceptron

Le plus simple des réseaux neuronaux est sans conteste le *perceptron*, qui correspond à un neurone unique à n entrées binaires. Il calcule une somme pondérée de ses entrées et « se déclenche » si la somme vaut zéro ou davantage :

```
from scratch.linear_algebra import Vector, dot
def step_function(x : float) -> float :
    return 1.0 if x >= 0 else 0.0

def perceptron_output(weights : Vector,
                      bias : float, x : Vector) -> float :
    """retourne 1 si le perceptron se déclenche, 0 sinon"""
    calcul = dot(weights, x) + bias
    return step_function(calcul)
```

Le perceptron fait simplement la distinction entre les demi-espaces séparés par l'hyperplan des points x pour lesquels :

```
dot(weights, x) + bias == 0
```

Lorsqu'on choisit correctement les pondérations, les perceptrons savent résoudre un grand nombre de problèmes simples (figure 18-1). Par exemple, nous pouvons créer une *porte logique* *and* (qui renvoie 1 si les deux entrées valent 1, mais 0 si une des entrées vaut 0) :

```
and_weights = [2., 2]
and_bias = -3.
assert perceptron_output(and_weights, and_bias, [1, 1]) == 1
assert perceptron_output(and_weights, and_bias, [0, 1]) == 0
assert perceptron_output(and_weights, and_bias, [1, 0]) == 0
assert perceptron_output(and_weights, and_bias, [0, 0]) == 0
```

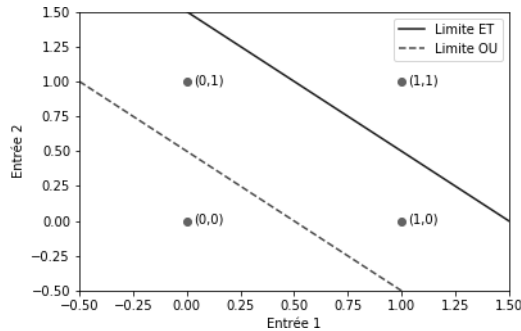
Si les deux entrées valent 1, le calcul revient à $2 + 2 - 3 = 1$ et la sortie vaut 1. Si seule une des deux entrées est égale à 1, le calcul revient à $2 + 0 - 3 = -1$ et la sortie vaut 0. Si les deux entrées sont égales à 0, le calcul revient à -3 et la sortie vaut 0.

Selon un raisonnement similaire, nous pouvons créer une *porte logique* *or* :

```
or_weights = [2., 2]
or_bias = -1.
assert perceptron_output(or_weights, or_bias, [1, 1]) == 1
assert perceptron_output(or_weights, or_bias, [0, 1]) == 1
assert perceptron_output(or_weights, or_bias, [1, 0]) == 1
assert perceptron_output(or_weights, or_bias, [0, 0]) == 0
```


Figure 18–1

Espace de décision pour un perceptron à deux entrées



Et nous pouvons de même créer une *porte* not (à une entrée, qui convertit 0 en 1 et 1 en 0) :

```
not_weights = [-2.]
not_bias = 1.
assert perceptron_output(not_weights, not_bias, [0]) == 1
assert perceptron_output(not_weights, not_bias, [1]) == 0
```

Cependant, il existe des problèmes qu'il est tout simplement impossible de résoudre par un perceptron unique. Par exemple, malgré tous vos efforts, aucun perceptron ne saura construire une *porte logique* xor (ou exclusif), qui envoie 1 si seule une des entrées vaut 1 et 0 dans le cas contraire. À ce stade, il vous faudra des réseaux neuronaux plus complexes.

Évidemment, il n'est pas indispensable de passer par un neurone pour créer une porte logique :

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

Comme les vrais neurones, les neurones artificiels deviennent plus intéressants dès que vous les connectez entre eux.

Réseaux neuronaux à propagation vers l'avant

La topologie du cerveau est extrêmement compliquée. Il est courant de la simuler par un *réseau neuronal idéal à propagation vers l'avant* (*feed-forward*). Il consiste en des couches discrètes de neurones, chacune connectée à la suivante. On parle alors de couche d'entrée (qui reçoit les entrées et les transmet vers l'avant sans les changer), d'au moins une couche « cachée » (recevant les sorties de la couche précédente, effectuant des calculs et transmettant le résultat à la couche suivante) et d'une couche de sortie (qui produit les sorties finales).

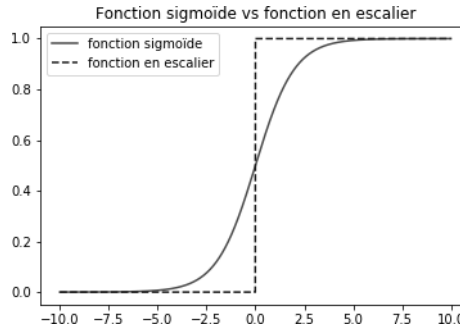
Comme dans le perceptron, chaque neurone (autre qu'entrée) présente un *poids* associé à chacune de ses entrées et un *biais*. Pour faciliter la représentation, nous ajouterons le biais à la fin de notre vecteur de poids et donnerons à chaque neurone un biais d'entrée toujours égal à 1.

Pour chaque neurone, nous calculerons la somme des produits de ses entrées et de ses poids, comme avec le perceptron. Cependant, plutôt que d'appliquer `step_function` (fonction en escalier) à cette somme pour générer une sortie, nous en retournerons une approximation lissée. Nous utiliserons la fonction sigmoïde (figure 18–2) :

```
import math
def sigmoid(t: float) -> float:
    return 1 / (1 + math.exp(-t))
```

Figure 18–2

La fonction sigmoïde



Pourquoi utiliser une sigmoïde plutôt que `step_function`, qui est plus simple ? Afin d'entraîner un réseau neuronal, nous devons faire appel au calcul et, pour cela, il nous faut des fonctions lissées.

La fonction en escalier n'est pas uniformément continue ; la sigmoïde en est une bonne approximation.

Remarque

Vous vous rappelez sans doute la sigmoïde du chapitre 16, lorsqu'elle était appelée fonction logistique. Techniquement, « sigmoïde » fait référence à la forme de la fonction et « logistique » se réfère à la fonction proprement dite. En pratique, les deux termes sont interchangeables.

Nous pouvons alors calculer la sortie de cette manière :

```
def neuron_output(weights: vector, inputs: vector) -> float:
    # le poids (weights) inclut le terme de biais, l'entrée (inputs) inclut un 1
    return sigmoid(dot(weights, inputs))
```

Étant donnée cette fonction, nous représentons un neurone simplement comme un vecteur de poids dont la longueur est supérieure de un au nombre d'entrées (à cause du biais de pondération). Ensuite, nous représentons un réseau neuronal comme une liste de couches (autres que couche d'entrée), dont chacune est une liste des neurones qu'elle contient.

Cela revient à dire que nous représentons un réseau neuronal comme une liste (couches) de listes (neurones) de vecteurs (poids). Ainsi, l'utilisation du réseau neuronal devient plutôt simple :

```
from typing import List
def feed_forward(neural_network: List[List[Vector]],
                 input_vector: Vector) -> List[Vector]:
    """Fournit le vecteur d'entrée via le réseau neuronal.
    Renvoie les sorties de toutes les couches (pas uniquement
    de la dernière)."""
    outputs: List[Vector] = []
    for layer in neural_network:
        input_with_bias = input_vector + [1] # ajouter une constante
        output = [neuron_output(neuron, input_with_bias) # calculer la sortie
                  for neuron in layer] # pour chaque neurone
        outputs.append(output) # ajouter les résultats
        # l'entrée de la couche suivante est la sortie de cette couche-ci
        input_vector = output
    return outputs
```

Il devient alors facile de construire la porte `xor`, qui était impossible à créer avec un simple perceptron. Il suffit pour cela de grossir l'échelle des poids, de telle sorte que les résultats neuronaux (`neuron_outputs`) soient très proches de 0 ou de 1 :

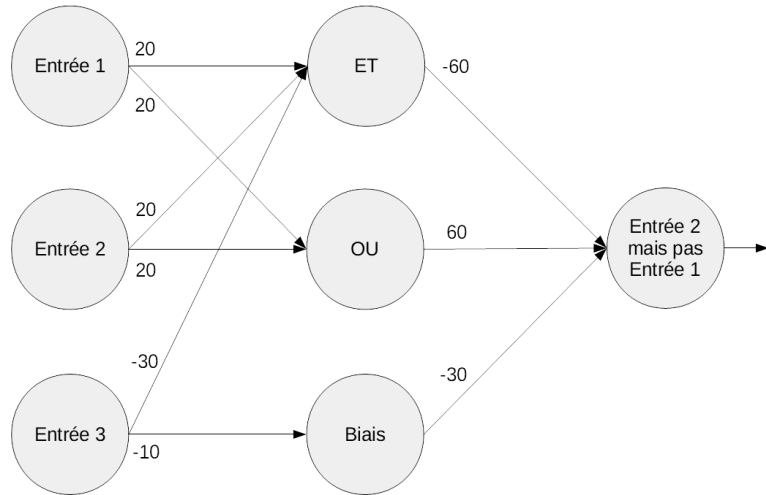
```
xor_network = [# couche cachée
               [[20., 20, -30], # neurone et
                [20., 20, -10]], # neurone ou
               # couche de sortie
               [[-60., 60, -30]] # neurone "2e entrée mais pas 1re"
# la propagation vers l'avant produit les sorties de toutes les couches
# Le [-1] obtient la sortie finale
# et le [0] la valeur sortant du vecteur résultant.
assert 0.000 < feed_forward(xor_network, [0, 0])[-1][0] < 0.001
assert 0.999 < feed_forward(xor_network, [1, 0])[-1][0] < 1.000
assert 0.999 < feed_forward(xor_network, [0, 1])[-1][0] < 1.000
assert 0.000 < feed_forward(xor_network, [1, 1])[-1][0] < 0.001
```

Pour une entrée donnée (qui consiste en un vecteur à deux dimensions), la couche cachée produit un vecteur à deux dimensions constitué du *and* des deux valeurs d'entrée et du *or* des deux valeurs d'entrée.

La couche de sortie accepte un vecteur à deux dimensions et calcule *deuxième élément mais non premier élément*. Le résultat est un réseau qui exécute *ou mais pas et*, ce qui correspond précisément à un opérateur `xor` (figure 18–3).

Figure 18-3

Un réseau neuronal
pour opérateur xor



Une manière évocatrice d'envisager la chose consiste à considérer que la couche cachée calcule des caractéristiques des données d'entrée (dans le cas présent ET et OU) et que la couche de sortie combine ces caractéristiques de sorte à générer la sortie souhaitée.

Rétropropagation

Les réseaux neuronaux ne sont généralement pas créés à la main. D'une part, nous les utilisons pour résoudre des problèmes particulièrement complexes : un problème de reconnaissance d'image peut impliquer des centaines ou des milliers de neurones. Et d'autre part, nous ne pouvons le plus souvent pas « déduire » ce que devraient être les neurones.

À la place, nous utilisons (comme d'habitude) des données pour entraîner un réseau neuronal. L'outil le plus répandu, pour cette tâche, est l'algorithme de *rétropropagation*, qui fait appel à la descente de gradient ou à une de ses variantes. Imaginons un jeu de données d'apprentissage composé de vecteurs d'entrée et des vecteurs de sortie correspondants. Par exemple, dans notre `xor_network` précédent, le vecteur d'entrée `[1, 0]` correspondait à la sortie cible `[1]`. Imaginons que notre réseau comporte un ensemble de poids. Nous allons ajuster les poids à l'aide de l'algorithme suivant :

- 1 Exécuter `feed_forward` sur un vecteur d'entrée pour produire les sorties de tous les neurones du réseau.
- 2 Nous connaissons la sortie cible, aussi nous pouvons calculer une perte qui correspond à la somme des erreurs au carré.
- 3 Calculer le gradient de cette perte sous la forme d'une fonction des poids du neurone de sortie.
- 4 « Propager » les gradients et erreurs vers l'arrière pour calculer les gradients par rapport aux poids des neurones cachés.
- 5 Effectuer un pas de descente de gradient.

D'habitude, on exécute cet algorithme plusieurs fois sur notre jeu d'apprentissage complet, jusqu'à ce que le réseau converge.

Pour commencer, rédigeons la fonction de calcul des gradients :

```
def sqerror_gradients(network: List[List[Vector]],
                      input_vector: Vector,
                      target_vector: Vector) -> List[List[Vector]]:
    """Soit un réseau neuronal, un vecteur d'entrée et un de sortie.
       Faire une prédiction et calculer le gradient de la perte
       des erreurs au carré par rapport aux poids du neurone."""
    # passe vers l'avant
    hidden_outputs, outputs = feed_forward(network, input_vector)
    # Gradients par rapport aux sorties de pré-activation
    # du neurone de sortie.
    output_deltas = [output * (1 - output) * (output - target)
                     for output, target in zip(outputs, target_vector)]
    # gradients par rapport aux poids du neurone de sortie
    output_grads = [[output_deltas[i] * hidden_output
                     for hidden_output in hidden_outputs + [1]]
                    for i, output_neuron in enumerate(network[-1])]
    # Gradients par rapport aux sorties de pré-activation
    # des neurones caches.
    hidden_deltas = [hidden_output * (1 - hidden_output) *
                     dot(output_deltas,
                         [n[i] for n in network[-1]])
                     for i, hidden_output in enumerate(hidden_outputs)]
    # gradients par rapport aux poids des neurones cachés
    hidden_grads = [[hidden_deltas[i] * input
                     for input in input_vector + [1]]
                    for i, hidden_neuron in enumerate(network[0])]
    return [hidden_grads, output_grads]
```

Le processus mathématique qui sous-tend les calculs précédents n'est pas très difficile. En revanche, il implique quelques opérations fastidieuses et une attention particulière aux détails. Je vous le laisse à titre d'exercice personnel.

Forts de cette capacité à calculer des gradients, nous pouvons à présent entraîner des réseaux neuronaux. Essayons d'entraîner le réseau xor que nous avons conçu manuellement plus haut.

Commençons par générer les données d'entraînement et par initialiser notre réseau neuronal avec des poids aléatoires :

```
import random
random.seed(0)
# données d'entraînement
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]]
# commencer avec des poids aléatoire
network = [ # couche cachée : 2 entrées -> 2 sorties
            [random.random() for _ in range(2 + 1)], # 1er neurone caché
```

```
[random.random() for _ in range(2 + 1)], # 2e neurone caché
      # couche de sortie : 2 entrées -> 1 sortie
[[random.random() for _ in range(2 + 1)] # neurone de sortie
```

Comme d'habitude, nous pouvons l'entraîner en utilisant une descente de gradient. À la différence des exemples précédents, nous disposons ici de plusieurs vecteurs paramétriques dont chacun possède son propre gradient. Cela signifie que nous devons appeler `gradient_step` pour chacun d'eux.

```
from scratch.gradient_descent import gradient_step
import tqdm
learning_rate = 1.0
for epoch in tqdm.trange(20000, desc="réseau neuronal pour xor"):
    for x, y in zip(xs, ys):
        gradients = sqerror_gradients(network, x, y)
        # accepter un pas de gradient pour chaque neurone dans chaque couche
        network = [[gradient_step(neuron, grad, -learning_rate)
                     for neuron, grad in zip(layer, layer_grad)]
                   for layer, layer_grad in zip(network, gradients)]

# s'assurer qu'il a appris xor
assert feed_forward(network, [0, 0])[-1][0] < 0.01
assert feed_forward(network, [0, 1])[-1][0] > 0.99
assert feed_forward(network, [1, 0])[-1][0] > 0.99
assert feed_forward(network, [1, 1])[-1][0] < 0.01
```

Dans mon cas, le réseau résultant présente des poids comme suit :

```
[
    [7, 7, -3], # couche cachée
    [5, 5, -8]], # calculer or
    [5, 5, -8]], # calculer and
    # couche de sortie
    [[11, -12, -5]] # calculer " premier mais non deuxième "
]
```

Conceptuellement, c'est assez similaire à notre réseau sur mesure précédent.

Exemple : Fizz Buzz

Le responsable Ingénierie souhaite utiliser Fizz Buzz, le classique défi de programmation, comme test à l'embauche des candidats techniques :

Imprimez les nombres de 1 à 100 sauf si ceux-ci sont :

- divisibles par 3, auquel cas imprimez fizz ;
- divisibles par 5, auquel cas imprimez buzz ;
- divisibles par 15, auquel cas imprimez fizzbuzz.

Selon lui, la capacité à réussir ce test prouve des compétences extrêmes en programmation. De votre côté, vous considérez ce problème si simple qu'un réseau neuronal pourrait le résoudre.

Les réseaux neuronaux acceptent des vecteurs en tant qu'entrées et produisent des vecteurs en sortie. Énoncé ainsi, le problème de programmation consistera à transformer un entier en chaîne. Le premier défi sera donc d'obtenir un moyen de refondre l'énoncé sous forme de problème vectoriel.

Quant aux sorties, il n'y a rien de bien difficile. Il en existe essentiellement quatre classes ; nous pourrions donc coder la sortie sous forme d'un vecteur constitué de quatre 0 et 1 :

```
def fizz_buzz_encode(x: int) -> Vector:
    if x % 15 == 0:
        return [0, 0, 0, 1]
    elif x % 5 == 0:
        return [0, 0, 1, 0]
    elif x % 3 == 0:
        return [0, 1, 0, 0]
    else:
        return [1, 0, 0, 0]
assert fizz_buzz_encode(2) == [1, 0, 0, 0]
assert fizz_buzz_encode(6) == [0, 1, 0, 0]
assert fizz_buzz_encode(10) == [0, 0, 1, 0]
assert fizz_buzz_encode(30) == [0, 0, 0, 1]
```

Nous utiliserons ce code pour générer nos vecteurs cibles. Les vecteurs d'entrée sont moins évidents. Vous ne voulez pas vous contenter d'utiliser un vecteur à une dimension contenant le nombre en entrée et ce pour deux raisons. Si une seule entrée capte une « intensité », le fait que 2 est deux fois plus que 1 et que 4 est deux fois plus encore n'est pas pertinent dans le cadre de ce problème. En outre, avec une seule entrée, la couche cachée n'est pas en mesure de traiter des caractéristiques très intéressantes ; autrement dit, elle ne permettrait pas de résoudre le problème.

Il s'avère qu'une approche fonctionne plutôt bien : convertir chaque nombre en sa représentation binaire (constituée de 1 et de 0). Pas de panique, ce n'était pas évident pour moi non plus ; du moins pas au début.

```
def binary_encode(x: int) -> Vector:
    binary: List[float] = []
    for i in range(10):
        binary.append(x % 2)
        x = x // 2
    return binary
#
# 1 2 4 8 16 32 64 128 256 512
assert binary_encode(0) == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
assert binary_encode(1) == [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
assert binary_encode(10) == [0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
assert binary_encode(101) == [1, 0, 1, 0, 0, 0, 1, 1, 0, 0]
assert binary_encode(999) == [1, 1, 1, 0, 0, 1, 1, 1, 1, 1]
```

Sachant que l'objectif consiste à construire les résultats pour les nombres de 1 à 100, fonder un entraînement sur ces nombres reviendrait à tricher. Aussi fonderons-nous l'entraînement sur les nombres de 101 à 1 023 (qui est le plus grand nombre que nous pouvons représenter avec 10 chiffres binaires) :

```
xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]
```

Ensuite, nous allons créer un réseau neuronal avec des poids initiaux aléatoires. Il disposera de 10 neurones d'entrée (sachant que nous représentons nos entrées sous forme de vecteurs à 10 dimensions) et de 4 neurones de sortie (nous représentons nos cibles sous forme de vecteurs à 4 dimensions). Nous lui fournirons 25 unités cachées, mais utiliserons une variable afin de faciliter les changements :

```
NUM_HIDDEN = 25
network = [ # couche cachée : 10 entrées -> sorties NUM_HIDDEN
    [[random.random() for _ in range(10 + 1)]
     for _ in range(NUM_HIDDEN)],
    # couche de sortie : entrées NUM_HIDDEN -> 4 sorties
    [[random.random() for _ in range(NUM_HIDDEN + 1)]
     for _ in range(4)]]
```

Et voilà. Nous sommes à présent prêts pour l'entraînement. Sachant qu'il s'agit d'un problème réclamant plus d'attention (beaucoup d'éléments peuvent prêter à confusion), nous allons surveiller étroitement le processus d'entraînement.

Nous suivrons et imprimerons particulièrement la somme des erreurs au carré pour chaque epoch (étape). Nous devons nous assurer que ces valeurs diminuent bel et bien :

```
from scratch.linear_algebra import squared_distance
learning_rate = 1.0
with tqdm.trange(500) as t:
    for epoch in t:
        epoch_loss = 0.0
        for x, y in zip(xs, ys):
            predicted = feed_forward(network, x)[-1]
            epoch_loss += squared_distance(predicted, y)
            gradients = sqerror_gradients(network, x, y)
            # Accepter un pas de gradient
            # pour chaque neurone dans chaque couche.
            network = [[gradient_step(neuron, grad, -learning_rate)
                        for neuron, grad in zip(layer, layer_grad)]
                       for layer, layer_grad in zip(network, gradients)]
        t.set_description(f"fizz buzz (loss: {epoch_loss:.2f})")
```

Cet entraînement va prendre du temps. Toutefois, la perte devrait finir par se réduire au minimum. Enfin, nous voilà prêts à résoudre le problème initial. Il ne nous reste qu'un point à

gérer. Notre réseau va produire un vecteur à 4 dimensions, mais nous ne voulons qu'une seule prédiction. Pour y remédier, nous accepterons `argmax`, l'index de la valeur la plus élevée :

```
def argmax(xs: list) -> int:
    """renvoyer l'index de la valeur la plus élevée"""
    return max(range(len(xs)), key=lambda i: xs[i])

assert argmax([0, -1]) == 0 # items[0] est le plus élevé
assert argmax([-1, 0]) == 1 # items[1] est le plus élevé
assert argmax([-1, 10, 5, 20, -3]) == 3 # items[3] est le plus élevé
```

Nous pouvons enfin résoudre le défi Fizz Buzz :

```
num_correct = 0
for n in range(1, 101):
    x = binary_encode(n)
    predicted = argmax(feed_forward(network, x)[-1])
    actual = argmax(fizz_buzz_encode(n))
    labels = [str(n), "fizz", "buzz", "fizzbuzz"]
    print(n, labels[predicted], labels[actual])
    if predicted == actual:
        num_correct += 1
print(num_correct, "/", 100)
```

Dans mon cas, le réseau entraîné se comporte correctement à 96/100, un score qui se situe bien au-dessus du seuil de recrutement du responsable Ingénierie. Confronté à l'évidence, il reconsidère les choses et change de défi de recrutement au profit de l'inversion d'un arbre binaire.

Pour aller plus loin

- Poursuivez votre lecture : le chapitre 19 aborde ces thèmes plus en détails.
- Ma publication de blog intitulée *Fizz Buzz in Tensorflow*¹ est assez intéressante.

1. <https://joelgrus.com/2016/05/23/fizz-buzz-in-tensorflow/>

19

Apprentissage profond

Un savoir médiocre est une chose dangereuse ; il faut boire à pleine coupe à la fontaine d'Hypocrène.

Alexander Pope

L'*apprentissage profond* (*deep learning*) désigne à l'origine l'application de réseaux neuronaux dotés de plus d'une couche cachée. En pratique, le terme englobe désormais une gamme étendue d'architectures neuronales (incluant les réseaux neuronaux « simples » que nous avons développés au chapitre 18).

Dans ce chapitre, nous allons capitaliser sur notre travail précédent et examiner une gamme plus étendue de réseaux neuronaux. Pour ce faire, nous allons introduire un certain nombre d'abstractions qui vont nous permettre d'appréhender les réseaux neuronaux de manière plus générale.

Tenseur

Jusqu'ici, nous faisons une distinction entre vecteurs (tableaux à une dimension) et matrices (tableaux à deux dimensions). En commençant à travailler avec des réseaux neuronaux plus complexes, nous avons aussi besoin de tableaux affichant davantage de dimensions.

Dans nombre de bibliothèques de réseaux neuronaux, les tableaux à n dimensions sont connus sous le nom de *tenseurs*. C'est d'ailleurs ainsi que nous les appellerons.

Remarque

Un certain pédantisme mathématique justifie de ne pas désigner les tableaux à n dimensions par le terme « tenseurs » ; si vous êtes dans ce cas, votre objection est notée.

Si j'écrivais un livre entier consacré à l'apprentissage profond, je mettrais en œuvre une classe `Tensor` exhaustive qui surchargerait les opérateurs arithmétiques de Python et saurait gérer tout un panel d'autres opérations. Une telle mise en œuvre occuperait un chapitre entier à elle seule. Ici, nous allons tricher et dire qu'un tenseur est simplement une liste. C'est d'ailleurs vrai dans un sens : tous nos vecteurs ou matrices, et éléments analogues fortement dimensionnés, sont des listes. Toutefois, ce n'est absolument pas le cas dans l'autre sens : en majorité, les listes Python ne sont pas des tableaux à n dimensions au sens où nous l'entendons.

Remarque

Dans l'idéal, vous écririez quelque chose comme ce qui suit :

```
# in Tensor est soit un float, soit une liste de tenseurs
```

```
Tensor = Union[float, List[Tensor]]
```

Toutefois, Python ne vous laisse pas définir de tels types récurifs. Et, quand bien même, cette définition resterait incorrecte car elle autorise de mauvais tenseurs, comme le suivant dont les rangées présentent des tailles différentes, ce qui en fait autre chose qu'un tableau à n dimensions.

```
[[1.0, 2.0],  
 [3.0]]
```

Donc, nous allons tout simplement tricher :

```
Tensor = list
```

Et nous rédigerons une fonction d'aide pour rechercher la forme (*shape*) d'un tenseur :

```
from typing import List
def shape(tensor: Tensor) -> List[int]:
    sizes: List[int] = []
    while isinstance(tensor, list):
        sizes.append(len(tensor))
        tensor = tensor[0]
    return sizes

assert shape([1, 2, 3]) == [3]
assert shape([[1, 2], [3, 4], [5, 6]]) == [3, 2]
```

Sachant que les tenseurs peuvent présenter n'importe quel nombre de dimensions, nous devons en général les exploiter de manière récursive. Nous ferons une opération dans le cas à une dimension et appliquerons une récursivité dans le cas plus dimensionné :

```
def is_1d(tensor: Tensor) -> bool:
    """Si tensor[0] est une liste, c'est un tenseur d'ordre supérieur.
    Sinon, le tenseur est à 1 dimension (c'est-à-dire, un vecteur)."""
    return not isinstance(tensor[0], list)

assert is_1d([1, 2, 3])
assert not is_1d([[1, 2], [3, 4]])
```

Nous pouvons l'utiliser pour rédiger une fonction récursive `tensor_sum` :

```
def tensor_sum(tensor: Tensor) -> float:
    """fait la somme de toutes les valeurs présentes dans le tenseur"""
    if is_1d(tensor):
        # une simple liste de float ; utiliser la somme Python
        return sum(tensor) # une simple liste de float ; utiliser la somme Python
    else:
        # appeler somme_tenseur sur chaque ligne et additionner ces résultats
        return sum(tensor_sum(tensor_i) # appeler tensor_sum sur chaque ligne
                    for tensor_i in tensor) # et faire la somme de ces résultats

assert tensor_sum([1, 2, 3]) == 6
assert tensor_sum([[1, 2], [3, 4]]) == 10
```

Si vous n'avez pas l'habitude de raisonner de manière récursive, vous devez méditer sur ce thème jusqu'à ce qu'il fasse sens. En effet, nous utiliserons la même logique tout au long du chapitre. Toutefois, nous allons créer deux fonctions d'aide pour éviter la contrainte de réécrire cette logique à chaque fois. La première applique une fonction élémentaire à un tenseur unique :

```
from typing import Callable
def tensor_apply(f: Callable[[float], float], tensor: Tensor) -> Tensor:
    """appliquer f aux éléments"""
    if is_1d(tensor):
        return [f(x) for x in tensor]
    else:
        return [tensor_apply(f, tensor_i) for tensor_i in tensor]

assert tensor_apply(lambda x: x + 1, [1, 2, 3]) == [2, 3, 4]
assert tensor_apply(lambda x: 2 * x, [[1, 2], [3, 4]]) == [[2, 4], [6, 8]]
```

Servons-nous de ce code pour rédiger une fonction qui crée un tenseur nul doté de la même forme que celle d'un tenseur donné :

```
def zeros_like(tensor: Tensor) -> Tensor:
    return tensor_apply(lambda _: 0.0, tensor)

assert zeros_like([1, 2, 3]) == [0, 0, 0]
assert zeros_like([[1, 2], [3, 4]]) == [[0, 0], [0, 0]]
```

Nous devons en outre appliquer une fonction à des éléments correspondants à partir de deux tenseurs (dont il vaudrait mieux qu'ils aient exactement la même forme, même si nous ne vérifierons pas la chose) :

```
def tensor_combine(f: Callable[[float, float], float],
                  t1: Tensor,
                  t2: Tensor) -> Tensor:
    """applique f aux éléments correspondants de t1 et t2"""
```

```

if is_1d(t1):
    return [f(x, y) for x, y in zip(t1, t2)]
else:
    return [tensor_combine(f, t1_i, t2_i)
            for t1_i, t2_i in zip(t1, t2)]
import operator
assert tensor_combine(operator.add, [1, 2, 3], [4, 5, 6]) == [5, 7, 9]
assert tensor_combine(operator.mul, [1, 2, 3], [4, 5, 6]) == [4, 10, 18]

```

Abstraction de couche

Au chapitre précédent, nous avons construit un réseau neuronal simple : nous avons empilé deux couches de neurones, dont chacune calculait `sigmoid(dot(weights, inputs))`.

Bien qu'il s'agisse sans doute là d'une représentation idéalisée de ce que fait un vrai neurone, en pratique nous apprécierons de pouvoir élargir un peu le champ d'applications. Ainsi, peut-être aimerions-nous que les neurones se souviennent de quelque chose à propos de leurs entrées précédentes et qu'ils utilisent une fonction d'activation autre que sigmoïde ? En outre, nous aimerions souvent utiliser plus de deux couches (notre fonction `feed_forward` gérait en fait n'importe quel nombre de couches, à la différence de nos calculs de gradient).

Dans ce chapitre, nous allons élaborer une mécanique pour mettre en œuvre une telle variété de réseaux neuronaux. Notre abstraction fondamentale sera la *couche* (*layer*) ; un élément qui sait comment appliquer une forme fonctionnelle à ses entrées et comment rétropropager des gradients.

Une manière d'appréhender les réseaux neuronaux que nous avons construits au chapitre 18 consiste à les considérer comme une couche « linéaire » suivie d'une couche « sigmoïde », puis d'une autre couche « linéaire » et d'une autre couche « sigmoïde ». Si nous ne les avons pas distingués en ces termes, le faire nous permet d'expérimenter des structures plus générales :

```

from typing import Iterable, Tuple
class Layer:
    """Nos réseaux neuronaux sont constitués de couches.
    Chacune d'elles sait comment appliquer certains calculs
    à ses entrées vers "l'avant"
    et propager des gradients "vers l'arrière"."""
    def forward(self, input):
        """Notez l'absence de types.
        Nous n'allons pas nous montrer trop contraignants
        quant à ce que les types de couches d'entrée peuvent accepter,
        ni quant aux types de sortie qu'elles peuvent retourner."""
        raise NotImplementedError

```

```

def backward(self, gradient):
    """Nous n'allons pas nous montrer trop contraignants non plus
    quant à l'aspect du gradient. Il est du ressort de l'utilisateur
    de s'assurer qu'il fait preuve de bon sens."""
    raise NotImplementedError

def params(self) -> Iterable[Tensor]:
    """Retourne les paramètres de cette couche.
    La mise en œuvre par défaut ne retourne rien.
    Ainsi, si votre couche ne dispose d'aucun paramètre,
    vous n'êtes pas contraints de mettre cette logique en œuvre."""
    return ()

def grads(self) -> Iterable[Tensor]:
    """retourne les gradients, dans le même ordre que params()"""
    return ()

```

Les méthodes vers l'avant et l'arrière devront être mises en œuvre dans nos sous-classes concrètes. Une fois le réseau neuronal construit, nous le formerons à utiliser une descente de gradient ; en d'autres termes, à actualiser chaque paramètre présent dans le réseau en utilisant son gradient. De même, nous insisterons sur le fait que chaque couche soit en mesure de nous fournir ses paramètres et gradients.

Certaines couches (par exemple, celles qui appliquent une fonction sigmoïde à chacune de leurs entrées) ne présentent aucun paramètre à actualiser. Nous fournissons donc une mise en œuvre par défaut pour gérer ce cas.

Examinons une telle couche :

```

from scratch.neural_networks import sigmoid
class Sigmoid(Layer):
    def forward(self, input: Tensor) -> Tensor:
        """Applique une fonction sigmoïde à chaque élément du tenseur
        d'entrée et enregistre les résultats pour les utiliser en
        rétro-propagation."""
        self.sigmoids = tensor_apply(sigmoid, input)
        return self.sigmoids
    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda sig, grad: sig * (1 - sig) * grad,
                               self.sigmoids, gradient)

```

Deux choses sont à noter dans cette situation. Premièrement, pendant la passe vers l'avant, nous avons enregistré les fonctions sigmoïdes calculées de manière à pouvoir les réutiliser dans la passe vers l'arrière. Nos couches nécessiteront typiquement ce type de procédé.

Deuxièmement, vous pouvez vous demander d'où vient le $\text{sig} * (1 - \text{sig}) * \text{grad}$. Il s'agit simplement de la règle de chaîne issue du calcul et cela correspond à $\text{output} * (1 - \text{output}) * (\text{output} - \text{target})$ de nos précédents réseaux neuronaux.

Enfin, vous constatez que nous pouvons utiliser les fonctions `tensor_apply` et `tensor_combine`. Pour la plupart, nos couches utiliseront ces fonctions de manière similaire.

Couche linéaire

L'autre élément dont nous aurons besoin pour dupliquer les réseaux neuronaux du chapitre 18 est une couche « linéaire » qui représente la partie `dot(weights, inputs)` des neurones.

Cette couche présente des paramètres que nous souhaitons initialiser avec des valeurs aléatoires.

Il s'avère que les valeurs initiales des paramètres peuvent faire une énorme différence en termes de rapidité d'apprentissage du réseau (voire en termes d'apprentissage tout court). Si les poids sont trop élevés, ils sont susceptibles de produire des sorties volumineuses dans une plage où la fonction d'activation affiche des gradients proches de zéro. Or, des parties du réseau qui présentent des gradients nuls ne peuvent évidemment rien apprendre via une descente de gradient.

En conséquence, nous mettrons en œuvre trois schémas différents de génération aléatoire de nos tenseurs de poids. Le premier consiste à choisir chaque valeur à partir d'une distribution uniforme aléatoire sur $[0, 1]$ – à savoir en tant que `random.random`. Le deuxième (et par défaut) consiste à choisir chaque valeur aléatoirement à partir d'une distribution normale standard. Et la troisième consiste à utiliser la technique de Xavier, dans laquelle chaque poids est initialisé avec un tirage aléatoire à partir d'une distribution normale d'espérance 0 et de variance $2 / (\text{num\_inputs} + \text{num\_outputs})$. Il s'avère que cette approche est souvent bien adaptée à des poids de réseau neuronal. Nous les mettrons donc en œuvre avec une fonction `random_uniform` et une fonction `random_normal` :

```
import random
from scratch.probability import inverse_normal_cdf
def random_uniform(*dims: int) -> Tensor:
    if len(dims) == 1:
        return [random.random() for _ in range(dims[0])]
    else:
        return [random_uniform(*dims[1:]) for _ in range(dims[0])]

def random_normal(*dims: int,
                  mean: float = 0,0,
                  variance: float = 1.0) -> Tensor:
    if len(dims) == 1:
        return [mean + variance * inverse_normal_cdf(random.random())
                for _ in range(dims[0])]
    else:
        return [random_normal(*dims[1:], mean=mean, variance=variance)
                for _ in range(dims[0])]

assert shape(random_uniform(2, 3, 4)) == [2, 3, 4]
assert shape(random_normal(5, 6, mean=10)) == [5, 6]
```


Puis nous incorporons l'ensemble dans une fonction `random_tensor` :

```
def random_tensor(*dims: int, init: str = 'normal') -> Tensor:
    if init == 'normal':
        return random_normal(*dims)
    elif init == 'uniform':
        return random_uniform(*dims)
    elif init == 'xavier':
        variance = len(dims) / sum(dims)
        return random_normal(*dims, variance=variance)
    else:
        raise ValueError(f"unknown init: {init}")
```

À présent, nous pouvons définir notre couche linéaire. Nous devons l'initialiser avec la dimension des entrées (qui nous informe sur le nombre de poids dont chaque neurone a besoin), la dimension des sorties (qui nous informe sur le nombre de neurones dont nous devrions disposer) et le schéma d'initialisation souhaité :

```
from scratch.linear_algebra import dot
class Linear(Layer):
    def __init__(self,
                  input_dim: int,
                  output_dim: int,
                  init: str = 'xavier') -> None:
        """Une couche de neurones output_dim,
        chacun doté de poids input_dim
        (et d'un biais)."""
        self.input_dim = input_dim
        self.output_dim = output_dim
        # self.w[o] est le poids pour l'autre neurone
        self.w = random_tensor(output_dim, input_dim, init=init)
        # self.b[o] est le terme de biais pour l'autre neurone
        self.b = random_tensor(output_dim, init=init)
```

Remarque

Au cas où vous vous interrogerez sur l'importance des schémas d'initialisation, sachez que je n'ai pas pu « entraîner » certains des réseaux de ce chapitre avec des initialisations autres que celles que je vous ai indiquées.

La méthode vers l'avant est facile à mettre en œuvre. Nous obtenons une sortie par neurone, que nous attachons dans un vecteur. Et la sortie de chaque neurone est le produit scalaire de son poids avec l'entrée, plus son biais :

```
def forward(self, input: Tensor) -> Tensor:
    # enregistre l'entrée à utiliser dans la passe vers l'arrière
    self.input = input
    # retourne le vecteur des sorties de neurone
    return [dot(input, self.w[o]) + self.b[o]
            for o in range(self.output_dim)]
```

La méthode vers l'arrière est moins évidente mais, si vous vous y connaissez en calcul, ce n'est pas difficile :

```
def backward(self, gradient: Tensor) -> Tensor:
    # Chaque b[o] est ajouté à output[o], ce qui signifie
    # que le gradient de b est le même que celui de sortie.
    self.b_grad = gradient
    # chaque w[o][i] multiplie input[i] et est ajouté à output[o]
    # son gradient est input[i] * gradient[o]
    self.w_grad = [[self.input[i] * gradient[o]
                     for i in range(self.input_dim)]
                    for o in range(self.output_dim)]
    # Chaque input[i] multiplie chaque w[o][i] et est ajoutée à chaque
    # output[o]. Son gradient est la somme de w[o][i] * gradient[o]
    # à l'échelle de toutes les sorties.
    return [sum(self.w[o][i] * gradient[o]
                for o in range(self.output_dim))
            for i in range(self.input_dim)]
```

Remarque

Dans une « vraie » bibliothèque de tenseurs, ces opérations (ainsi que de nombreuses autres) se présenteraient sous forme de matrices ou de multiplications de tenseurs, ce que ces bibliothèques sont conçues pour réaliser très rapidement. La nôtre est très lente.

Enfin, nous aurons besoin ici de mettre en œuvre des paramètres et des gradients (`params` et `grads`). Nous disposons de deux paramètres et de deux gradients correspondants :

```
def params(self) -> Iterable[Tensor]:
    return [self.w, self.b]

def grads(self) -> Iterable[Tensor]:
    return [self.w_grad, self.b_grad]
```

Réseaux neuronaux en tant que séquences de couches

Il est recommandé de considérer les réseaux neuronaux comme des séquences de couches ; aussi, trouvons un moyen de combiner plusieurs couches en une seule. Le réseau neuronal résultant est lui-même une couche et il met en œuvre des méthodes `Layer` de manière évidente :

```
from typing import List
class Sequential(Layer):
    """Une couche consistant en une séquence d'autres couches.
    Vous devez vous assurer que la sortie de chacune
    fait sens comme entrée de la suivante."""
```

```

def __init__(self, layers: List[Layer]) -> None:
    self.layers = layers

def forward(self, input):
    """passe l'entrée vers l'avant à travers les couches dans l'ordre"""
    for layer in self.layers:
        input = layer.forward(input)
    return input

def backward(self, gradient):
    """rétropropage le gradient à travers les couches"""
    for layer in reversed(self.layers):
        gradient = layer.backward(gradient)
    return gradient

def params(self) -> Iterable[Tensor]:
    """retourne les paramètres de chaque couche"""
    return (param
            for layer in self.layers
            for param in layer.params())

def grads(self) -> Iterable[Tensor]:
    """retourne les gradients de chaque couche"""
    return (grad
            for layer in self.layers
            for grad in layer.grads())

```

Représentons par exemple le réseau neuronal que nous utilisons pour le xor :

```

xor_net = Sequential([
    Linear(input_dim=2, output_dim=2),
    Sigmoid(),
    Linear(input_dim=2, output_dim=1),
    Sigmoid()
])

```

Nous avons encore besoin d'un peu de mécanique supplémentaire pour le former.

Perte et optimisation

Nous avons précédemment rédigé des fonctions individuelles de perte et de gradient pour nos modèles. Ici, nous voulons essayer différentes fonctions de perte. Donc, nous allons introduire une nouvelle abstraction `Loss` (« perte ») qui encapsule à la fois le calcul de perte et celui du gradient :

```

class Loss:
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        """Nos prédictions sont-elles bonnes ?
        (Les nombres les plus élevés sont les pires.)"""
        raise NotImplementedError

```

```
def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
    """Comment la perte évolue-t-elle quand les prédictions changent ?"""
    raise NotImplementedError
```

Nous avons déjà travaillé à de nombreuses reprises avec la perte qui consiste en la somme des erreurs au carré. Aussi devrait-il nous être facile de mettre cette mécanique en œuvre. Seul point délicat, nous devons recourir à `tensor_combine` :

```
class SSE(Loss):
    """fonction Loss qui calcule la somme des erreurs au carré"""
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        # calculer le tenseur des différences au carré
        squared_errors = tensor_combine(
            lambda predicted, actual: (predicted - actual) ** 2,
            predicted,
            actual)

        # et les additionner
        return tensor_sum(squared_errors)

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        return tensor_combine(
            lambda predicted, actual: 2 * (predicted - actual),
            predicted,
            actual)
```

Le dernier élément à déterminer est la descente de gradient. Au fil du livre, vous avez établi toutes les descentes de gradient manuellement en recourant à une boucle d'apprentissage qui implique un code comme suit :

```
theta = gradient_step(theta, grad, -learning_rate)
```

Ici, ce procédé ne fonctionnera pas. La première raison tient à ce que nos réseaux neuronaux présentent de nombreux paramètres et que nous devons tous les actualiser. La seconde tient à ce que nous aimerions pouvoir utiliser des variantes plus intelligentes de descente de gradient, sans en réécrire le code à chaque fois.

Ajoutons une abstraction d'optimiseur, (`Optimizer`), dont la descente de gradient constituera une instance spécifique :

```
class Optimizer:
    """Un optimiseur actualise les poids d'une couche (en place)
    en utilisant des informations connues soit de la couche,
    soit de l'optimiseur (soit des deux)."""
    def step(self, layer: Layer) -> None:
        raise NotImplementedError
```

Après cela, il est facile de mettre en œuvre une descente de gradient, ici encore en utilisant `tensor_combine` :

```
class GradientDescent(Optimizer):
    def __init__(self, learning_rate: float = 0.1) -> None:
        self.lr = learning_rate

    def step(self, layer: Layer) -> None:
        for param, grad in zip(layer.params(), layer.grads()):
            # actualiser param en utilisant un pas de gradient
            param[:] = tensor_combine(
                lambda param, grad: param - grad * self.lr,
                param,
                grad)
```

La seule chose susceptible de surprendre, c'est *l'affectation de tranche* (*slice assignment*), qui reflète le fait que réaffecter une liste ne change pas sa valeur d'origine. En d'autres termes, si vous venez de coder `param = tensor_combine(...)`, vous redéfinissez la variable locale `param`, mais vous n'affectez pas le tenseur d'origine stocké dans la couche. Cependant, si vous l'affectez à la tranche `[:]`, l'opération change en fait les valeurs présentes dans la liste.

Voici un exemple simple pour illustrer la chose :

```
tensor = [[1, 2], [3, 4]]
for row in tensor:
    row = [0, 0]
# une affectation ne change pas la liste
assert tensor == [[1, 2], [3, 4]]
for row in tensor:
    row[:] = [0, 0]
# mais une affectation de tranche, si
assert tensor == [[0, 0], [0, 0]]
```

Si vous manquez quelque peu d'expérience en Python, ce comportement peut s'avérer surprenant. Aussi, méditez sur le sujet et exercez-vous jusqu'à ce que les choses prennent un sens.

Pour démontrer la valeur de cette abstraction, mettons en œuvre un autre optimiseur qui utilise `Momentum`. L'idée est de ne pas réagir excessivement à chaque nouveau gradient. Nous maintenons donc une moyenne mobile des gradients que nous avons vus, en l'actualisant avec chaque nouveau gradient et en progressant d'un pas dans la direction de la moyenne :

```
class Momentum(Optimizer):
    def __init__(self,
        learning_rate: float,
        momentum: float = 0.9) -> None:
        self.lr = learning_rate
        self.mo = momentum
        self.updates: List[Tensor] = [] # moyenne mobile
```

```
def step(self, layer: Layer) -> None:
    # commencer avec uniquement des zéros
    if not self.updates:
        self.updates = [zeros_like(grad) for grad in layer.grads()]
    for update, param, grad in zip(self.updates,
                                   layer.params(),
                                   layer.grads()):

        # appliquer momentum
        update[:] = tensor_combine(
            lambda u, g: self.mo * u + (1 - self.mo) * g,
            update,
            grad)

        # ensuite progresser d'un pas de gradient
        param[:] = tensor_combine(
            lambda p, u: p - self.lr * u,
            param,
            update)
```

Sachant que nous avons utilisé une abstraction, nous pouvons facilement basculer entre nos différents optimiseurs.

Exemple : l'opérateur xor revisité

Voyons à quel point il est facile d'utiliser notre nouvelle infrastructure pour entraîner un réseau capable de calculer un opérateur `xor` (ou exclusif). Commençons par recréer les données d'entraînement :

```
# données d'entraînement
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]
```

Ensuite, nous définissons le réseau, même si à présent nous pouvons laisser de côté la dernière couche sigmoïde :

```
random.seed(0)
net = Sequential([
    Linear(input_dim=2, output_dim=2),
    Sigmoid(),
    Linear(input_dim=2, output_dim=1)
])
```

Nous pouvons à présent rédiger une simple boucle d'entraînement à l'aide des abstractions `optimizer` et `Loss`, grâce auxquelles il nous sera facile de tester des versions différentes :

```
import tqdm
optimizer = GradientDescent(learning_rate=0.1)
loss = SSE()
```

```

with tqdm.trange(3000) as t:
    for epoch in t:
        epoch_loss = 0.0
        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)
            optimizer.step(net)
        t.set_description(f"xor loss {epoch_loss:.3f}")

```

Ce code devrait « apprendre rapidement » et vous devriez voir la perte diminuer. Nous pouvons maintenant inspecter les poids :

```

for param in net.params():
    print(param)

```

Pour mon réseau, j'obtiens grosso modo :

```

hidden1 = -2.6 * x1 + -2.7 * x2 + 0.2 # non ou
hidden2 = 2.1 * x1 + 2.1 * x2 - 3.4   # et
output = -3.1 * h1 + -2.6 * h2 + 1.8 # non ou

```

Ainsi, `hidden1` s'active si aucune entrée n'est égale à 1, `hidden2` s'active si les deux entrées valent 1 et `output` sortie s'active si aucune sortie cachée n'est égale à 1 (en d'autres termes, si le cas n'est pas celui où aucune entrée ne vaut 1 et si ce n'est pas non plus celui où les deux entrées sont égales à 1).

Remarquez que ce réseau a appris des caractéristiques autres que celles dont nous avons fait l'apprentissage au chapitre 18, mais parvient tout de même à faire la même chose.

Autres fonctions d'activation

La fonction sigmoïde est tombée en disgrâce pour deux raisons. La première tient à ce que `sigmoid(0)` est égal à 1/2, ce qui signifie qu'un neurone dont la somme des entrées est de 0 présente une sortie positive. La seconde est que son gradient est très proche de 0 pour les très grandes et très petites entrées ; en d'autres termes, ses gradients peuvent être « saturés » et ses poids bloqués.

Substitut plébiscité, `tanh` (« tangente hyperbolique ») est une autre fonction de forme sigmoïde qui va de -1 à 1 et génère 0 en sortie si son entrée est 0. La dérivée de `tanh(x)` est simplement `1 - tanh(x)2`, ce qui facilite l'écriture de la couche :

```

import math
def tanh(x: float) -> float:
    # pour x très grand ou très petit, tanh vaut (essentiellement) 1 ou -1
    # nous nous en assurons car math.exp(1000) lève une erreur

```

```

    if x < -100: return -1
    elif x > 100: return 1
    em2x = math.exp(-2 * x)
    return (1 - em2x) / (1 + em2x)

class Tanh(Layer):
    def forward(self, input: Tensor) -> Tensor:
        # enregistre la sortie pour les passes vers l'arrière
        self.tanh = tensor_apply(tanh, input)
        return self.tanh

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda tanh, grad: (1 - tanh ** 2) * grad,
                               self.tanh,
                               gradient)

```

Dans des réseaux de plus grande envergure, ReLU (*Rectified Linear Unit*) constitue un substitut répandu : 0 pour les entrées négatives et l'identité pour les entrées positives :

```

class Relu(Layer):
    def forward(self, input: Tensor) -> Tensor:
        self.input = input
        return tensor_apply(lambda x: max(x, 0), input)

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda x, grad: grad if x > 0 else 0,
                               self.input,
                               gradient)

```

Il en existe beaucoup d'autres. Je vous encourage à les expérimenter vous-mêmes dans vos réseaux.

Exemple : Fizz Buzz revisité

Nous pouvons à présent utiliser notre infrastructure d'apprentissage profond sur l'exemple de Fizz Buzz (chapitre 18). Mettons les données en place :

```

from scratch.neural_networks import binary_encode, fizz_buzz_encode, argmax
xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]

```

Et créons le réseau :

```

NUM_HIDDEN = 25
random.seed(0)

```



```
net = Sequential([
    Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
    Tanh(),
    Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform'),
    Sigmoid()
])
```

Nous allons suivre notre précision sur le jeu d'apprentissage :

```
def fizzbuzz_accuracy(low: int, hi: int, net: Layer) -> float:
    num_correct = 0
    for n in range(low, hi):
        x = binary_encode(n)
        predicted = argmax(net.forward(x))
        actual = argmax(fizz_buzz_encode(n))
        if predicted == actual:
            num_correct += 1
    return num_correct / (hi - low)

optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SSE()

with tqdm.trange(1000) as t:
    for epoch in t:
        epoch_loss = 0.0
        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)
            optimizer.step(net)
        accuracy = fizzbuzz_accuracy(101, 1024, net)
        t.set_description(f"fb loss: {epoch_loss:.2f}
                           acc: {accuracy:.2f}")
# à présent, contrôlons les résultats sur l'ensemble de test
print("test results", fizzbuzz_accuracy(1, 101, net))
```

Après 1 000 itérations d'entraînement, le modèle atteint une précision de 90 % sur l'ensemble de test ; si vous continuez à l'entraîner, il devrait faire mieux. Je ne pense pas qu'il soit possible de l'entraîner pour atteindre une précision de 100 % avec seulement 25 unités cachées, mais c'est tout à fait possible si vous montez à 50 unités.

Fonctions softmax et entropie croisée

Le réseau neuronal que nous avons utilisé dans la section précédente s'achevait dans une couche sigmoïde ; en d'autres termes, sa sortie était un vecteur de nombres entre 0 et 1. Plus particulièrement, il pouvait générer en sortie un vecteur qui était entièrement constitué de 0

ou entièrement constitué de 1. Cependant, lorsque nous nous attaquons à des problèmes de classification, nous avons besoin de produire un 1 pour la classe correcte et un 0 pour toutes les autres. En général, nos prédictions ne seront pas aussi parfaites, mais nous voudrions au moins prédire une répartition de probabilités réelle sur les classes.

Par exemple, si nous disposons de deux classes et si notre modèle génère $[0, 0]$, il est difficile d'en tirer quelque chose de pertinent. Comment peut-il considérer que la sortie appartient à l'une des deux classes ?

En revanche, si notre modèle génère $[0.4, 0.6]$, nous pouvons l'interpréter comme une prédiction d'une probabilité de 0,4 que notre entrée appartienne à la première classe et de 0,6 qu'elle appartienne à la deuxième classe.

Pour ce faire, nous renonçons généralement à la couche `sigmoid` et utilisons en lieu et place la fonction `softmax` (exponentielle normalisée), qui convertit un vecteur de nombres réels en un vecteur de probabilités. Nous calculons $\exp(x)$ pour chaque nombre dans le vecteur, ce qui débouche sur un vecteur de nombres positifs. Ensuite, nous divisons simplement chacun de ces nombres positifs par la somme, ce qui nous donne un ensemble de nombres positifs qui s'ajoutent à 1 –, à savoir un vecteur de probabilités.

Si jamais nous essayons de calculer, par exemple, $\exp(1000)$, nous obtenons une erreur Python. Aussi, avant d'accepter `exp`, nous soustrayons la valeur la plus élevée. Il en résulte alors les mêmes probabilités ; le calcul est juste plus sûr en Python :

```
def softmax(tensor: Tensor) -> Tensor:
    """softmax sur la dernière dimension"""
    if is_1d(tensor):
        # Soustraire la valeur la plus élevée
        # pour garantir une stabilité numérique.
        largest = max(tensor)
        exps = [math.exp(x - largest) for x in tensor]
        sum_of_exps = sum(exps) # il s'agit du « poids » total
        return [exp_i / sum_of_exps # La probabilité est la fraction
                for exp_i in exps] # du poids total.
    else:
        return [softmax(tensor_i) for tensor_i in tensor]
```

Dès lors que notre réseau produit des probabilités, nous utilisons souvent une fonction de perte différente appelée *entropie croisée* (ou parfois « log-vraisemblance négative »).

Vous vous rappelez sans doute qu'à la section « Estimation du maximum de vraisemblance » (chapitre 14), nous avons justifié l'utilisation des moindres carrés dans une régression linéaire au prétexte que (selon certaines hypothèses) les coefficients des moindres carrés maximisaient la vraisemblance des données observées.

Nous pouvons ici procéder de manière similaire : si les sorties de notre réseau sont des probabilités, la perte d'entropie croisée représente la log-vraisemblance (et donc la vraisemblance) des données d'apprentissage.

En général, on n'inclut pas la fonction `softmax` en tant que partie intégrante du réseau neuronal proprement dit. En effet, il s'avère que, si `softmax` est intégrée à votre fonction de perte mais non à votre réseau proprement dit, les gradients de la perte par rapport aux sorties du réseau sont très faciles à calculer.

```
class SoftmaxCrossEntropy(Loss):
    """La log-vraisemblance négative des valeurs observées, selon le modèle
    de réseau neuronal.
    Si nous choisissons des poids pour la minimiser, notre modèle
    maximisera la vraisemblance des données observées."""
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        # appliquer softmax pour obtenir des probabilités
        probabilities = softmax(predicted)
        # Il s'agira de log p_i pour la classe réelle i
        # et de 0 pour les autres.
        # ajouter une petite quantité à p pour ne pas accepter log(0)
        likelihoods = tensor_combine(
            lambda p, act: math.log(p + 1e-30) * act,
            probabilities,
            actual)
        # puis additionner les négatifs
        return -tensor_sum(likelihoods)

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        probabilities = softmax(predicted)
        # n'est-ce pas là une équation plaisante ?
        return tensor_combine(lambda p, actual: p - actual,
                               probabilities,
                               actual)
```

Si à présent j'entraîne le même réseau Fizz Buzz en utilisant une perte `SoftmaxCrossEntropy`, je constate qu'il s'entraîne typiquement plus vite (à savoir, avec beaucoup moins d'étapes). Il est probablement beaucoup plus facile de trouver des poids qui s'appliquent à une distribution donnée avec `softmax` qu'avec `sigmoid`.

Ainsi, si j'ai besoin de prédire une classe 0 (un vecteur avec un 1 en première position et des 0 dans les autres) dans le cas `linear + sigmoid`, il faut que la première sortie soit un nombre positif élevé et que les sorties restantes soient des nombres négatifs élevés. Dans le cas de `softmax`, j'ai juste besoin que la première sortie soit plus grande que les autres. En clair, il existe de nombreuses autres manières pour que le deuxième cas se produise, ce qui laisse entendre qu'il devrait être plus facile de trouver des poids adéquats :

```
random.seed(0)
net = Sequential([
    Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
    Tanh(),
    Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform') ])
# aucune couche sigmoid finale pour le moment
```

```

optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SoftmaxCrossEntropy()
with tqdm.trange(100) as t:
    for epoch in t:
        epoch_loss = 0.0
        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)
            optimizer.step(net)
        accuracy = fizzbuzz_accuracy(101, 1024, net)
        t.set_description(f"fb loss: {epoch_loss:.3f}    "
                        "acc: {accuracy:.2f}")
# contrôlons les résultats sur l'ensemble de test
print("test results", fizzbuzz_accuracy(1, 101, net))

```

Abandon

À l'instar de la majorité des modèles d'apprentissage automatique, les réseaux neuronaux sont enclins au surajustement vis-à-vis de leurs données d'entraînement. Nous avons vu des moyens d'améliorer les choses ; par exemple, nous avons pénalisé les poids importants, ce qui a contribué à empêcher ledit surajustement (voir la régularisation, chapitre 15).

L'*abandon* (appelé aussi « décrochage ») est un moyen répandu de régulariser les réseaux neuronaux. En phase d'entraînement, nous désactivons aléatoirement chaque neurone (à savoir, remplaçons sa sortie par 0) selon une certaine probabilité fixe. En d'autres termes, le réseau ne peut pas apprendre à dépendre d'un neurone donné, ce qui semble contribuer à résoudre le problème de surajustement.

En phase d'évaluation, nous ne souhaitons abandonner aucun neurone. Donc, une couche `Dropout` (« abandon ») devra savoir s'il s'agit ou non d'entraînement. En outre, en phase d'entraînement, une couche `Dropout` passe uniquement une certaine fraction aléatoire de son entrée. Pour rendre sa sortie comparable en cours d'évaluation, nous diminuerons l'échelle des sorties (uniformément) en utilisant cette même fraction :

```

class Dropout(Layer):
    def __init__(self, p: float) -> None:
        self.p = p
        self.train = True

    def forward(self, input: Tensor) -> Tensor:
        if self.train:
            # Créer un masque de 0 et de 1 ayant la forme de l'entrée
            # en utilisant la probabilité spécifiée.
            self.mask = tensor_apply(
                lambda _: 0 if random.random() < self.p else 1,
                input)

```

```

        # multiplier le masque pour abandonner les entrées
        return tensor_combine(operator.mul, input, self.mask)
    else:
        # En phase d'évaluation, diminuer uniformément
        # l'échelle des sorties.
        return tensor_apply(lambda x: x * (1 - self.p), input)

def backward(self, gradient: Tensor) -> Tensor:
    if self.train:
        # propager uniquement les gradients où mask == 1
        return tensor_combine(operator.mul, gradient, self.mask)
    else:
        raise RuntimeError(
            "ne pas appeler backward hors mode d'entraînement"

```

Exemple : MNIST

MNIST¹ est un ensemble de données de chiffres manuscrits particulièrement utilisé pour enseigner l'apprentissage profond.

Il est mis à disposition dans un format binaire quelque peu délicat. Aussi installerons-nous la bibliothèque `mnist` pour l'exploiter. Eh oui, techniquement, cette partie ne part pas « de zéro ».

```
python -m pip install mnist
```

Nous pouvons alors charger les données :

```

import mnist
# téléchargement des données ; modifiez l'emplacement selon vos souhaits
# Oui, c'est bien une fonction à zéro argument ;
# c'est ce qu'attend la bibliothèque.
# Oui, j'affecte bien une lambda à une variable ;
# faites ce que je dis, pas ce que je fais.
mnist.temporary_dir = lambda: '/tmp'
# Chacune de ces fonctions télécharge d'abord les données, puis retourne
# un tableau numpy.
# nous appelons .tolist() car nos "tenseurs" sont de simples listes
train_images = mnist.train_images().tolist()
train_labels = mnist.train_labels().tolist()

assert shape(train_images) == [60000, 28, 28]
assert shape(train_labels) == [60000]

```

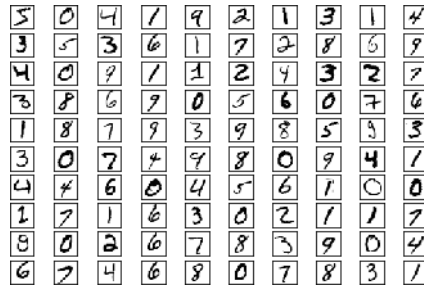
1. <http://yann.lecun.com/exdb/mnist/>

Traçons les 100 premières images d'entraînement pour voir de quoi elles ont l'air (figure 19–1) :

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(10, 10)
for i in range(10): for j in range(10):
    # tracer chaque image en noir et blanc et masquer les axes
    ax[i][j].imshow(train_images[10 * i + j], cmap='Greys')
    ax[i][j].xaxis.set_visible(False)
    ax[i][j].yaxis.set_visible(False)
plt.show()
```

Figure 19–1

Des images MNIST



Remarque

Ma première tentative de présentation des images a débouché sur des nombres jaunes sur fond noir. Il faut ajouter `cmap='Greys'` pour obtenir des images en noir et blanc. J'ai trouvé la solution sur Stack Overflow. En tant que scientifique des données, vous deviendrez vite adepte de ce site web.

Nous devons également charger les images de test :

```
test_images = mnist.test_images().tolist()
test_labels = mnist.test_labels().tolist()

assert shape(test_images) == [10000, 28, 28]
assert shape(test_labels) == [10000]
```

Chaque image mesure 28×28 pixels, mais nos couches linéaires ne gèrent que des entrées à une dimension. Nous devons donc les aplatir (et également diviser par 256 pour les inscrire entre 0 et 1). En outre, notre réseau neuronal s'entraînera mieux si nos entrées ont une moyenne nulle ; nous soustrairons donc la valeur moyenne :

```
# calculer la valeur moyenne des pixels
avg = tensor_sum(train_images) / 60000 / 28 / 28
# recentrer, redimensionner et aplatir
train_images = [(pixel - avg) / 256
                 for row in image
                 for pixel in row]
for image in train_images]
```

```

test_images = [(pixel - avg) / 256
                for row in image
                for pixel in row]
                for image in test_images]

assert shape(train_images) == [60000, 784], "Les images doivent être aplaties"
assert shape(test_images) == [10000, 784], "Les images doivent être aplaties"

# après centrage, le pixel moyen doit être très proche de 0
assert -0.0001 < tensor_sum(train_images) < 0.0001

```

Nous voulons également appliquer un encodage *un parmi n* (`one_hot_encode`) aux cibles, car nous avons 10 sorties :

```

def one_hot_encode(i: int, num_labels: int = 10) -> List[float]:
    return [1.0 if j == i else 0.0 for j in range(num_labels)]

assert one_hot_encode(3) == [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
assert one_hot_encode(2, num_labels=5) == [0, 0, 1, 0, 0]

train_labels = [one_hot_encode(label) for label in train_labels]
test_labels = [one_hot_encode(label) for label in test_labels]

assert shape(train_labels) == [60000, 10]
assert shape(test_labels) == [10000, 10]

```

L'une des forces de nos abstractions tient à ce que nous pouvons utiliser la même boucle apprentissage/évaluation avec une variété de modèles. Commençons donc par rédiger ce code. Nous allons lui passer notre modèle, les données, une fonction de perte et (si nous sommes en apprentissage) un optimiseur.

Il fera une passe sur nos données, suivra les performances (si nous passons un optimiseur) et mettra nos paramètres à jour :

```

import tqdm
def loop(model: Layer,
        images: List[Tensor],
        labels: List[Tensor],
        loss: Loss,
        optimizer: Optimizer = None) -> None:
    correct = 0 # suit le nombre de prédictions correctes
    total_loss = 0.0 # suit la perte totale
    with tqdm.trange(len(images)) as t:
        for i in t:
            predicted = model.forward(images[i]) # prédiction
            if argmax(predicted) == argmax(labels[i]): # Recherche le
                correct += 1 # caractère correct.
            total_loss += loss.loss(predicted, labels[i]) # calcule la perte
            # En phase d'entraînement, rétropropage le gradient
            # et actualise les poids.

```

```

if optimizer is not None:
    gradient = loss.gradient(predicted, labels[i])
    model.backward(gradient)
    optimizer.step(model)
# actualise nos éléments de mesure dans la barre de progression
avg_loss = total_loss / (i + 1)
acc = correct / (i + 1)
t.set_description(f"mnist loss: {avg_loss:.3f}    "
                  acc: {acc:.3f}")

```

À la base, nous pouvons utiliser notre bibliothèque d'apprentissage profond pour entraîner un module de régression logistique (à classes multiples), ce qui équivaut à une couche linéaire unique suivie d'une fonction `softmax`. Par essence, ce modèle recherche uniquement 10 fonctions linéaires, de sorte que, si l'entrée représente, disons, un 5, alors la cinquième fonction linéaire produit la sortie la plus élevée.

Une passe sur nos 60 000 exemples d'entraînement devrait suffire à l'apprentissage du modèle :

```

random.seed(0)
# la régression logistique est une couche linéaire suivie d'un softmax
model = Linear(784, 10)
loss = SoftmaxCrossEntropy()
# cet optimiseur semble fonctionner
optimizer = Momentum(learning_rate=0.01, momentum=0.99)
# entraîner sur les données d'entraînement
loop(model, train_images, train_labels, loss, optimizer)
# tester sur les données de test
# l'absence d'optimiseur revient à une simple évaluation
loop(model, test_images, test_labels, loss)

```

Nous obtenons une précision d'environ 89 %. Voyons si nous pouvons faire mieux avec un réseau neuronal profond. Nous allons utiliser deux couches cachées, respectivement de 30 et de 10 neurones, et notre activation `Tanh` :

```

random.seed(0)
# nommés pour pouvoir activer et désactiver leur entraînement
dropout1 = Dropout(0.1)
dropout2 = Dropout(0.1)
model = Sequential([
    Linear(784, 30), # couche cachée 1 : taille 30
    dropout1,
    Tanh(),
    Linear(30, 10), # couche cachée 2 : taille 10
    dropout2,
    Tanh(),
    Linear(10, 10) ]) # couche de sortie : taille 10

```


Et nous pouvons tout simplement utiliser la même boucle d'entraînement !

```
optimizer = Momentum(learning_rate=0.01, momentum=0.99)
loss = SoftmaxCrossEntropy()
# Activer l'abandon et l'entraînement
# (cela prend moins de 20 minutes sur mon ordinateur portable !).
dropout1.train = dropout2.train = True
loop(model, train_images, train_labels, loss, optimizer)
# désactiver l'abandon et évaluer
dropout1.train = dropout2.train = False
loop(model, test_images, test_labels, loss)
```

Notre modèle profond affiche une précision de plus de 92 % sur l'ensemble de test, ce qui constitue une belle amélioration par rapport au modèle logistique simple.

Le site web MNIST décrit une variété de modèles qui surclassent ces exemples. Nombre d'entre eux peuvent être mis en œuvre au moyen de la mécanique que nous avons élaborée jusqu'à maintenant. Toutefois, l'entraînement de tels modèles s'avérerait extrêmement long dans notre infrastructure de listes-en-tant-que-tenseurs. Certains des meilleurs modèles font appel à des couches dites *convolutives* ; si elles sont importantes, elles dépassent malheureusement largement le cadre de ce livre d'introduction à la data science.

Enregistrement et chargement des modèles

L'entraînement de ces modèles prend du temps. Aussi avons-nous intérêt à les enregistrer pour éviter de devoir les entraîner à chaque fois. Heureusement, le module `json` facilite la sérialisation des poids de modèle dans un fichier.

Pour l'enregistrement, nous utilisons `Layer.params` afin de collecter les poids, de les placer dans une liste et d'utiliser `json.dump` pour enregistrer cette liste dans un fichier :

```
import json
def save_weights(model: Layer, filename: str) -> None:
    weights = list(model.params())
    with open(filename, 'w') as f:
        json.dump(weights, f)
```

Le « rechargement » des poids ne constitue qu'une petite charge de travail supplémentaire. Nous récupérons la liste avec `json.load` et une affectation de tranche pour définir les poids de notre modèle.

Cela signifie, en particulier, que nous devons instancier le modèle nous-mêmes pour ensuite charger les poids. Une autre approche consisterait à également enregistrer une certaine représentation de l'architecture du modèle et à l'utiliser pour instancier le modèle. L'idée n'est pas fabuleuse car elle nécessite beaucoup plus de code ainsi que des modifications de toutes nos couches. Nous nous en tiendrons donc à la méthode la plus simple.

Avant de charger les poids, mieux vaut nous assurer qu'ils présentent les mêmes formes que les paramètres de modèle dans lesquels nous les chargeons. Cette étape est une protection contre, par exemple, une tentative de charger les poids d'un réseau neuronal profond dans un réseau moins profond ou contre tout problème similaire.

```
def load_weights(model: Layer, filename: str) -> None:
    with open(filename) as f:
        weights = json.load(f)
        # recherche de cohérence
        assert all(shape(param) == shape(weight)
                    for param, weight in zip(model.params(), weights))
        # ensuite, chargement en utilisant une affectation de tranche
        for param, weight in zip(model.params(), weights):
            param[:] = weight
```

Remarque

JSON stocke vos données sous forme de texte, une représentation particulièrement inefficace. Dans de « vraies » applications, vous utiliseriez probablement la bibliothèque `pickle`, qui sérialise les éléments dans un format binaire plus efficace. Ici, j'ai décidé de préserver la simplicité et la lisibilité humaine.

Vous pouvez télécharger les poids des différents réseaux que nous avons testés à partir du référentiel GitHub² de cet ouvrage.

Pour aller plus loin

L'apprentissage profond est un sujet particulièrement prisé en ce moment. Dans ce chapitre, nous n'avons fait que le survoler. Il existe des ouvrages et des articles de blog aussi nombreux qu'excellents (et d'autres tout aussi nombreux... et mauvais) abordant la quasi-totalité des aspects de l'apprentissage profond susceptibles de vous intéresser.

- Le manuel canonique *Deep Learning*³, de Ian Goodfellow, Yoshua Bengio et Aaron Courville (MIT Press), est disponible gratuitement en ligne (en anglais). C'est une excellente ressource mais qui implique un peu de mathématiques.
- L'ouvrage *Deep Learning with Python*⁴ (Manning) de François Chollet constitue une formidable présentation de la bibliothèque `Keras`, sur laquelle repose en quelque sorte le « schéma » de notre bibliothèque d'apprentissage profond (en anglais également).
- J'utilise moi-même essentiellement `PyTorch`⁵. Son site web regorge de documentation et de didacticiels.

2. <https://github.com/joelgrus/data-science-from-scratch>

3. <https://www.deeplearningbook.org/>

4. <https://www.manning.com/books/deep-learning-with-python>

5. <https://pytorch.org/>

20

Partitionnement de données

*Quand de tels groupes nous avions
Qui furieux nous rendaient, mais pas fous.*

Robert Herrick

Les algorithmes de ce livre font pour la plupart de l'apprentissage supervisé : ils commencent avec un ensemble de données étiquetées qu'ils utilisent comme base pour calculer des prédictions sur des données nouvelles non étiquetées. Le partitionnement de données (*clustering*) est quant à lui un exemple d'*apprentissage non supervisé* : les données à étudier sont totalement dépourvues d'étiquettes (ou présentent des étiquettes que nous ignorons).

Idée sous-jacente

Lorsque vous examinez des données, il est probable que, d'une manière ou d'une autre, vous y voyiez des groupes (*clusters*). Un ensemble de données indiquant où vivent les millionnaires présenterait probablement des groupes de lieux, comme Beverly Hills ou Manhattan. Un ensemble de données sur le temps de travail hebdomadaire présenterait probablement un groupe autour de 40. Si l'étude provient d'un état dont la loi prévoit des avantages particuliers pour ceux qui travaillent au moins 20 heures par semaine, elle révélerait probablement un autre groupe autour de 19. Un ensemble de données démographiques sur les électeurs inscrits donnerait lieu à des groupes variés (« mères de footballeurs en herbe », « retraités qui s'ennuient », « génération du millénaire au chômage ») considérés comme pertinents pour leur étude par les instituts de sondage et les analystes politiques.

À la différence des cas que nous avons examinés jusqu'ici, il n'existe en général pas de partitionnement « correct » de ces groupes. Un autre schéma de partitionnement pourrait tout aussi bien regrouper une partie de « génération du millénaire au chômage » en « étudiants diplômés » et l'autre en « adolescents qui squattent chez leurs parents ». Aucune solution n'est meilleure qu'une autre. Au contraire, chacune est sans doute la meilleure si on tient compte de sa propre métrique de partitionnement.

En outre, les groupes ne vont pas s'étiqueter tout seuls. Nous devons les qualifier en examinant les données sous-jacentes.

Modèle

Dans notre exemple, chaque entrée sera un vecteur dans un espace à d dimensions (que nous représenterons comme d'habitude par une liste de nombres). Notre but sera d'identifier des groupes d'entrées similaires et (parfois) de trouver une valeur représentative de chaque groupe.

Par exemple, chaque entrée pourrait être un vecteur numérique qui représente d'une certaine façon le titre d'un article de blog. Dans ce cas, le but serait de trouver des groupes d'articles similaires, par exemple pour mieux appréhender les sujets traités par nos utilisateurs sur leurs blogs respectifs. Ou encore, imaginons que vous disposiez d'une image avec des milliers de couleurs (fondées sur le rouge/vert/bleu) et que vous ayez besoin d'une impression écran limitée à 10 couleurs. Un partitionnement pourra alors vous aider à choisir les 10 couleurs qui vont minimiser « l'erreur couleur » d'ensemble.

Une des méthodes de partitionnement les plus simples est celle des *k-moyennes* (*k-means* en anglais) : le nombre k de groupes est choisi d'avance et on partitionne les entrées en ensembles $s_1, \dots, s_k$ de telle sorte qu'on minimise la somme totale des distances au carré entre chaque point et la moyenne de son groupe désigné.

Il existe plusieurs manières d'affecter n points à k groupes. Autrement dit, trouver un partitionnement optimal est particulièrement difficile. Faisons appel à un algorithme itératif qui permet en général d'en trouver un.

- 1 Commencer avec un jeu de k -moyennes, qui sont des points d'un espace à d dimensions.
- 2 Affecter chaque point à la moyenne dont il est le plus proche.
- 3 Si aucune affectation de point n'a changé, arrêter et conserver ces groupes.
- 4 Si certaines affectations de points ont changé, recalculer les moyennes puis recommencer à l'étape 2.

En utilisant la fonction `vector_mean` (chapitre 4), il est assez facile de créer une classe qui réalise cette opération.

Pour commencer, nous allons créer une fonction d'aide qui mesure le nombre des coordonnées par lesquelles deux vecteurs diffèrent. Nous l'utiliserons pour suivre la progression de notre entraînement.

```

from scratch.linear_algebra import Vector
def num_differences(v1: Vector, v2: Vector) -> int:
    assert len(v1) == len(v2)
    return len([x1 for x1, x2 in zip(v1, v2) if x1 != x2])

assert num_differences([1, 2, 3], [2, 1, 3]) == 2
assert num_differences([1, 2], [1, 2]) == 0

```

Nous avons aussi besoin d'une fonction qui, étant donnés certains vecteurs et leurs affectations à des groupes, calcule la moyenne des groupes. C'est le cas potentiel lorsqu'aucun point n'est affecté à un groupe. Nous ne pouvons pas accepter la moyenne d'une collection vide. Donc, dans ce cas, nous devons simplement prélever aléatoirement un des points pour l'utiliser comme la « moyenne » de ce groupe :

```

from typing import List
from scratch.linear_algebra import vector_mean
def cluster_means(k: int,
                  inputs: List[Vector],
                  assignments: List[int]) -> List[Vector]:
    # clusters[i] contient les entrées dont l'affectation est i
    clusters = [[] for i in range(k)]
    for input, assignment in zip(inputs, assignments):
        clusters[assignment].append(input)
    # si un groupe est vide, utiliser simplement un point aléatoire.
    return [vector_mean(cluster) if cluster else random.choice(inputs)
            for cluster in clusters]

```

Nous sommes à présent prêts à coder notre générateur de groupes, notre *partitionneur*. Comme d'habitude, nous utiliserons `tqdm` pour suivre notre progression. Toutefois, ici, nous ignorons le nombre d'itérations nécessaires. Nous utiliserons donc `itertools.count`, qui crée un élément itérable infini, et retournerons son résultat à la fin :

```

import itertools
import random
import tqdm
from scratch.linear_algebra import squared_distance
class KMeans:
    def __init__(self, k: int) -> None:
        self.k = k # nombre de groupes
        self.means = None

    def classify(self, input: Vector) -> int:
        """retourne l'index du groupe le plus proche de la sortie"""
        return min(range(self.k),
                    key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs: List[Vector]) -> None:
        # commencer avec des affectations aléatoires

```

```

assignments = [random.randrange(self.k) for _ in inputs]
with tqdm.tqdm(itertools.count()) as t:
    for _ in t:
        # calculer des moyennes et trouver de nouvelles affectations
        new_assignments = [self.classify(input) for input in inputs]
        # Contrôler le nombre d'affectations changées
        # et si nous avons terminé.
        num_changed = num_differences(assignments, new_assignments)
        if num_changed == 0:
            return
        # Sinon, conserver les nouvelles affectations
        # et calculer de nouvelles moyennes.
        assignments = new_assignments
        self.means = cluster_means(self.k, inputs, assignments)
        t.set_description(f"changed: {num_changed} / {len(inputs)}")

```

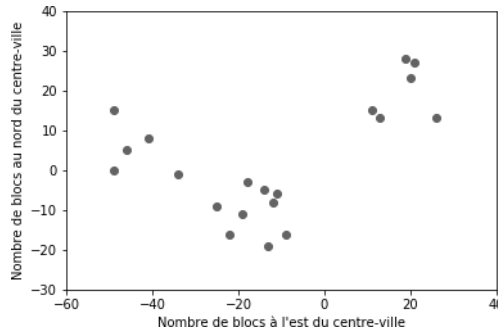
Examinons son fonctionnement de plus près.

Exemple : rencontres

Pour fêter la croissance de DataSciencester, la responsable Récompenses veut organiser plusieurs rencontres « en vrai » pour les utilisateurs de votre ville, avec bière, pizza et T-shirt DataSciencester. Vous savez où habitent vos utilisateurs (figure 20–1) et elle aimerait donc choisir des lieux de rencontre qui soient pratiques pour tout le monde.

Figure 20–1

Emplacement des utilisateurs
de votre ville



Selon la manière dont vous regardez les emplacements, vous pouvez probablement identifier deux ou trois groupes. Cela fonctionne bien visuellement parce qu'il n'y a que deux dimensions ; avec davantage, ce serait nettement plus difficile à l'œil nu.

Imaginons qu'on dispose d'un budget pour organiser trois rencontres. Vous démarrez votre ordinateur et vous essayez ce qui suit :

```

random.seed(12) # pour obtenir les mêmes résultats que moi
clusterer = KMeans(k=3)
clusterer.train(inputs)

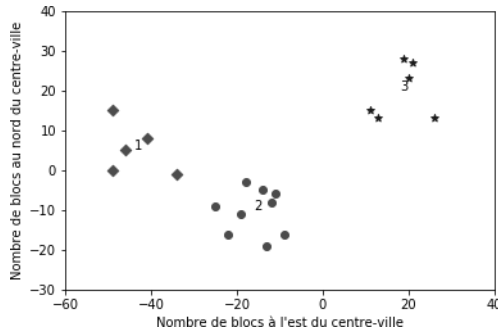
```

```
means = sorted(clusterer.means) # tri du test unitaire
assert len(means) == 3
# contrôler que les moyennes sont proches de ce qui est attendu
assert squared_distance(means[0], [-44, 5]) < 1
assert squared_distance(means[1], [-16, -10]) < 1
assert squared_distance(means[2], [18, 20]) < 1
```

Vous trouvez trois groupes centrés sur $[-44, 5]$, $[-16, -10]$ et $[18, 20]$. Vous cherchez maintenant des lieux de rencontre à proximité (figure 20–2).

Figure 20–2

Emplacement des utilisateurs :
trois groupes



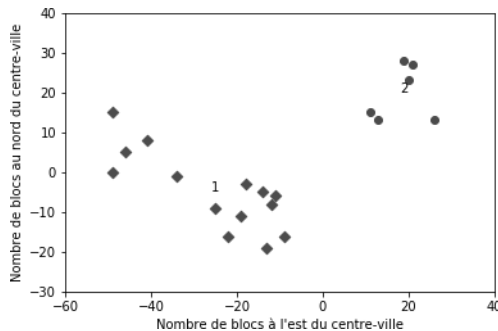
Vous faites part de votre proposition à la responsable, qui vous informe que son budget ne permet plus désormais que deux rencontres. « Aucun problème ! », répondez-vous :

```
random.seed(0)
clusterer = KMeans(k=2)
clusterer.train(inputs)
means = sorted(clusterer.means)
assert len(means) == 2
assert squared_distance(means[0], [-26, -5]) < 1
assert squared_distance(means[1], [18, 20]) < 1
```

Comme le montre la figure 20–3, une rencontre peut toujours se tenir près de $[18, 20]$ mais l'autre devra avoir lieu près de $[-26, -5]$.

Figure 20–3

Emplacement des utilisateurs :
deux groupes



Choisir k

Dans l'exemple précédent, le choix de k était déterminé par des facteurs externes hors de notre contrôle. En général, ce n'est pas le cas. Il existe de nombreuses manières de choisir k . Une solution assez facile à comprendre consiste à représenter graphiquement la somme des carrés des erreurs (entre chaque point et la moyenne de son groupe) comme une fonction de k et à rechercher le point où le graphique « s'infléchit » :

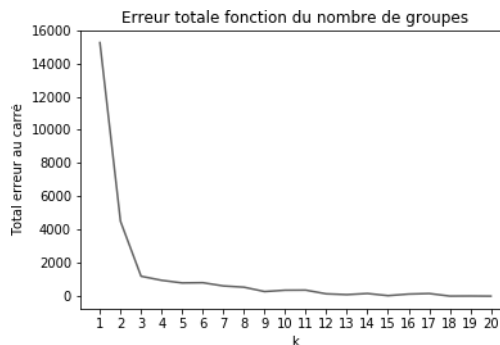
```
from matplotlib import pyplot as plt
def squared_clustering_errors(inputs: List[Vector], k: int) -> float:
    """Cherche la somme totale des carrés des erreurs des k-moyennes
    partitionnant les entrées."""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = [clusterer.classify(input) for input in inputs]
    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))
```

Appliquons à notre exemple précédent :

```
# tracer à présent la courbe de 1 à len(inputs) groupes
ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]
plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("Total erreur au carré")
plt.title("Erreur totale contre nombre de groupes")
plt.show()
```

D'après la figure 20–4, cette méthode confirme notre première impression visuelle : 3 est le « bon » nombre de groupes.

Figure 20–4
Choisir une valeur de k



Exemple : partitionnement de couleurs

Le responsable Style a conçu des autocollants DataSciencester très décoratifs à distribuer lors des rencontres. Malheureusement, votre imprimante ne peut pas imprimer plus de cinq couleurs par document. Et, comme le responsable Création est en congé sabbatique, le responsable Style vous demande s'il est possible de partir de son modèle et de n'en garder que cinq couleurs.

Les images informatisées peuvent être ramenées à des tableaux de pixels à deux dimensions, chaque pixel étant lui-même un vecteur à trois dimensions (rouge, vert, bleu) indiquant sa couleur.

La création d'une version en cinq couleurs signifie alors :

- 1 choisir cinq couleurs ;
- 2 affecter une de ces couleurs à chaque pixel.

C'est une tâche toute trouvée pour le partitionnement par les k -moyennes. Celui-ci peut, en effet, répartir les pixels en cinq groupes dans un espace rouge-vert-bleu. Si nous recolorons ensuite les pixels de chaque groupe par la couleur moyenne qui leur est associée, nous aurons terminé.

Pour commencer, il nous faut un moyen de charger une image dans Python. `matplotlib` est tout indiqué, mais nous devons tout d'abord installer la bibliothèque `pillow` :

```
python -m pip install pillow
```

Il nous suffit ensuite d'utiliser `matplotlib.image.imread` :

```
image_path = r"girl_with_book.jpg"
import matplotlib.image as mpimg
img = mpimg.imread(image_path) / 256 # redimensionner entre 0 et 1
```

Derrière `img` se cache un tableau `NumPy`. Dans notre cas, nous pouvons le traiter comme une liste de listes de listes.

`img[i][j]` est le pixel de la ligne `i` et de la colonne `j` ; chaque pixel est une liste `[red, green, blue]` de nombres entre 0 et 1 indiquant sa couleur¹ :

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

En particulier, nous obtenons une liste aplatie de tous les pixels, comme suit :

```
# .tolist() convertit un tableau NumPy en liste Python
pixels = [pixel.tolist() for row in img for pixel in row]
```

1. https://fr.wikipedia.org/wiki/Rouge_vert_bleu

Utilisons-la pour alimenter notre partitionneur :

```
clusterer = KMeans(5)
clusterer.train(pixels) # peut prendre un certain temps
```

Une fois l'opération terminée, il ne nous reste plus qu'à construire une nouvelle image de même format :

```
def recolor(pixel: Vector) -> Vector:
    cluster = clusterer.classify(pixel) # index du groupe le plus proche
    return clusterer.means[cluster] # moyenne du groupe le plus proche

new_img = [[recolor(pixel) for pixel in row]
            for row in img]
```

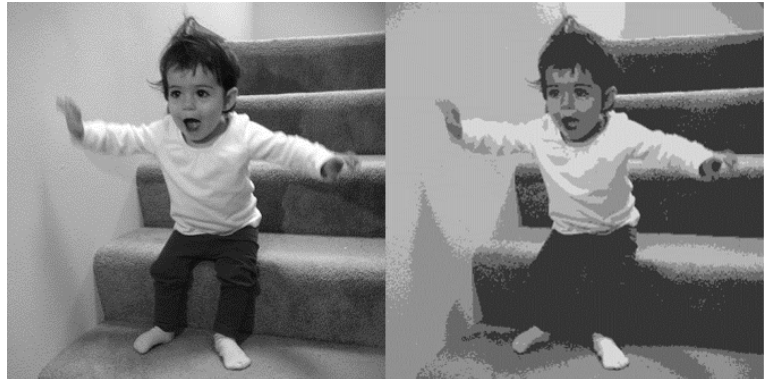
Enfin, on l'affiche à l'aide de `plt.imshow()` :

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

Il est difficile de présenter un résultat en couleurs dans un livre en noir et blanc, mais la figure 20-5 montre les versions en nuances de gris d'une image avec ses couleurs d'origine, puis retravaillée pour réduire les couleurs au nombre de cinq.

Figure 20-5

L'image originale et sa version recolorée en cinq couleurs



Partitionnement hiérarchique de bas en haut

Une autre méthode de partitionnement consiste à « faire croître » des groupes de bas en haut. Pour cela, il faut procéder comme suit.

- 1 Faire de chaque entrée son propre groupe d'un élément.
- 2 Tant qu'il reste plusieurs groupes, trouver les deux groupes les plus proches et les fusionner.

À la fin, il nous reste un groupe géant qui contient toutes les entrées. Si nous conservons la trace de l'ordre des fusions, nous pouvons recréer n'importe quel nombre de groupes en éclatant ces dernières. Par exemple, si nous voulons trois groupes, il nous suffit de défaire les deux dernières fusions.

Nous utiliserons une représentation très simple. Nos valeurs seront dans des groupes de type `Leaf` que nous représenterons comme des tuples à un élément :

```
from typing import NamedTuple, Union
class Leaf(NamedTuple):
    value: Vector

leaf1 = Leaf([10, 20])
leaf2 = Leaf([30, -15])
```

Nous les utiliserons pour faire croître les groupes fusionnés, que nous représenterons comme des `NamedTuple` :

```
class Merged(NamedTuple):
    children: tuple
    order: int

merged = Merged((leaf1, leaf2), order=1)
Cluster = Union[Leaf, Merged]
```

Remarque

Voici un autre cas où les annotations de type de Python nous ont abandonnés. Vous souhaitez taper l'indication `Merged.children` sous la forme `Tuple[Cluster, Cluster]` mais `mypy` n'autorise pas les types récursifs de ce genre.

Nous reparlerons de l'ordre de fusion par la suite. Créons d'abord une fonction d'aide qui retourne récursivement toutes les valeurs présentes dans un groupe (éventuellement fusionné) :

```
def get_values(cluster: Cluster) -> List[Vector]:
    if isinstance(cluster, Leaf):
        return [cluster.value]
    else:
        return [value
                for child in cluster.children
                for value in get_values(child)]

assert get_values(merged) == [[10, 20], [30, -15]]
```

Pour fusionner les groupes les plus proches, nous devons préciser la notion de distance entre groupes. Nous utiliserons la distance minimale entre les éléments, ce qui aura pour effet de fusionner les deux groupes les plus proches au point qu'ils se touchent (mais cela produit par-

fois des groupes volumineux en forme de chaînes peu proches les uns des autres). Si nous souhaitons obtenir des formes plutôt sphériques et compactes, nous utiliserons plutôt la distance maximale, car elle fusionne les deux groupes qui tiennent dans le plus petit espace sphérique. Les deux choix sont courants, comme l'est aussi le choix de la distance moyenne :

```
from typing import Callable
from scratch.linear_algebra import distance
def cluster_distance(cluster1: Cluster,
                    cluster2: Cluster,
                    distance_agg: Callable = min) -> float:
    """Calculer toutes les paires de distances entre cluster1 et cluster2
    et appliquer _distance_agg_ à la liste résultante."""
    return distance_agg([distance(v1, v2)
                        for v1 in get_values(cluster1)
                        for v2 in get_values(cluster2)])
```

Nous utiliserons la partie consacrée à l'ordre de tri pour suivre l'ordre selon lequel a été réalisée la fusion. Les nombres les plus petits représentent les fusions les plus tardives. En d'autres termes, pour inverser la fusion des groupes, on part de l'ordre de fusion le plus bas pour aller vers le plus élevé. Comme les groupes feuilles ne proviennent pas d'une fusion, nous leur affecterons l'infini, valeur la plus élevée possible. Et, sachant que nous ne disposons d'aucune propriété `.order`, nous créerons une fonction d'aide :

```
def get_merge_order(cluster: Cluster) -> float:
    if isinstance(cluster, Leaf):
        return float('inf') # ne résulte pas d'une fusion
    else:
        return cluster.order
```

De même, sachant que les groupes feuilles n'ont pas d'enfants, nous créerons et ajouterons une fonction d'aide à cet effet :

```
from typing import Tuple
def get_children(cluster: Cluster):
    if isinstance(cluster, Leaf):
        raise TypeError("Une feuille n'a pas d'enfant")
    else:
        return cluster.children
```

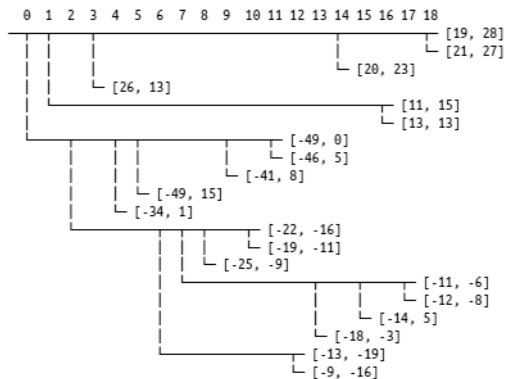
Nous sommes maintenant prêts à créer notre algorithme de partitionnement :

```
def bottom_up_cluster(inputs: List[Vector],
                     distance_agg: Callable = min) -> Cluster:
    # commencer avec toutes les feuilles
    clusters: List[Cluster] = [Leaf(input) for input in inputs]

    def pair_distance(pair: Tuple[Cluster, Cluster]) -> float:
        return cluster_distance(pair[0], pair[1], distance_agg)

    # tant qu'il reste plus d'un groupe...
    while len(clusters) > 1:
        # rechercher les deux groupes les plus proches
        c1, c2 = min(((cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]),
                     key=pair_distance)
        # les retirer de la liste des groupes
        clusters = [c for c in clusters if c != c1 and c != c2]
        # les fusionner en utilisant merge_order = nbre de groupes restants
        merged_cluster = Merged((c1, c2), order=len(clusters))
        # et ajouter leur fusion
        clusters.append(merged_cluster)
    # quand il ne reste plus qu'un groupe, le retourner
    return clusters[0]
```

Son utilisation est très simple. Il en résulte un groupe dont la représentation se présente comme suit :



Les nombres du haut indiquent « l'ordre de fusion ». Comme nous avons 20 entrées, 19 mouvements de fusion sont nécessaires pour arriver à ce groupe unique. La première fusion a créé le groupe 18 par combinaison des feuilles [19, 28] et [21, 27] et la dernière a créé le groupe 0.

Pour obtenir uniquement deux groupes, il faut effectuer une division au niveau de la première fourche (0), générant ainsi un groupe de six points et un autre avec le reste. Pour trois groupes, il faut poursuivre jusqu'à la deuxième fourche (1) et diviser les six points en deux groupes ([19, 28], [21, 27], [20, 23], [26, 13]) et ([11, 15], [13, 13]). Et ainsi de suite.

Cependant, généralement nous ne voulons pas nous abîmer les yeux sur des représentations graphiques de ce genre. Écrivons plutôt une fonction qui génère n'importe quel nombre de groupes en exécutant un nombre approprié de « dé-fusions » :

```
def generate_clusters(base_cluster: Cluster,
                     num_clusters: int) -> List[Cluster]:
    # commencer avec une liste contenant seulement le groupe de base
    clusters = [base_cluster]
    # tant que nous n'avons pas encore assez de groupes...
    while len(clusters) < num_clusters:
        # choisir le dernier groupe fusionné
        next_cluster = min(clusters, key=get_merge_order)
        # le retirer de la liste
        clusters = [c for c in clusters if c != next_cluster]
        # ajouter ses enfants à la liste (c'est-à-dire, inverser la fusion)
        clusters.extend(get_children(next_cluster))
    # une fois que nous avons assez de groupes...
    return clusters
```

Ainsi, par exemple, si nous voulons générer trois groupes, il nous suffit d'invoquer la commande suivante :

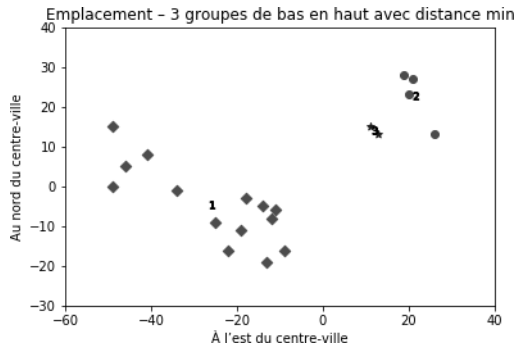
```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

Le résultat est facile à représenter graphiquement (figure 20–6) :

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D', 'O', '*'],
                                     ['r', 'g', 'b']):
    xs, ys = zip(*cluster) # astuce magique pour dézipper
    plt.scatter(xs, ys, color=color, marker=marker)
    # associer un nombre à la moyenne du groupe
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')
plt.title("Emplacement - 3 clusters de bas en haut avec distance min")
plt.xlabel("À l'est du centre-ville")
plt.ylabel("Au nord du centre-ville")
plt.show()
```

Figure 20–6

Trois groupes construits de bas en haut avec la méthode de la distance minimale

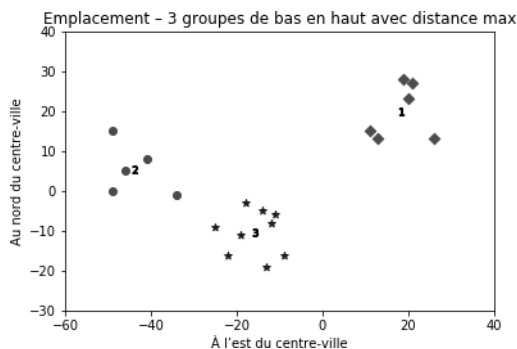


Les résultats sont très différents de ceux obtenus avec les *k*-moyennes. En effet, comme nous l'avons déjà expliqué, l'utilisation du minimum dans `cluster_distance` tend à donner des groupes en forme de chaînes. Si nous utilisons le maximum (qui donne des groupes serrés), le résultat est le même que celui des *k*-moyennes avec $k = 3$ (figure 20–7).

La mise en œuvre du partitionnement de bas en haut (`bottom_up_clustering`) est relativement simple, mais aussi remarquablement inefficace. En particulier, elle recalcule la distance entre chaque paire d'entrées à chaque étape. Il serait mieux de précalculer ces distances entre chaque paire d'entrées et d'exécuter ensuite une boucle à l'intérieur de `cluster_distance`. Une mise en œuvre réellement efficace devrait en outre mémoriser les valeurs de distance de l'étape précédente.

Figure 20–7

Trois groupes construits de bas en haut avec la méthode de la distance maximale



Pour aller plus loin

- `scikit-learn` propose un module entier `sklearn.cluster`² qui contient plusieurs algorithmes de partitionnement, y compris `K-Means` et l'algorithme hiérarchique de `Ward` (qui utilise un critère différent du nôtre pour fusionner les groupes).
- `Scipy`³ contient deux modèles de partitionnement : `scipy.cluster.vq` (méthode des `k-moyennes`) et `scipy.cluster.hierarchy` (qui propose divers algorithmes de partitionnement hiérarchique).

2. <https://scikit-learn.org/stable/modules/clustering.html>

3. <https://www.scipy.org/>

21

Traitement automatique du langage naturel

Ils sont allés à un grand festin de langues et ils ont volé les restes.

William Shakespeare

Le traitement du langage naturel (TLN) est un ensemble de techniques informatiques appliquées au langage.

Il s'agit d'un champ d'étude très vaste. Ici, nous allons nous intéresser seulement à quelques techniques plus ou moins simples.

Nuages de mots

Au chapitre 1, nous avons calculé des nombres de mots marquant les intérêts des utilisateurs. Une méthode pour visualiser les mots et leur nombre d'occurrences consiste à dessiner des nuages de mots dans lesquels ces derniers sont affichés dans une taille proportionnelle à leur fréquence.

En général, les data scientists ne sont pas très friands de cette représentation, en grande partie parce que la position des mots ne signifie rien d'autre que « j'ai trouvé une place suffisante pour écrire ce mot ».

Si jamais vous êtes amené à créer un nuage de mots, essayez de réfléchir au sens que vous pourriez donner aux axes. Par exemple, imaginons que, pour chaque collection de mots à la

mode relatifs à la data science, vous avez deux nombres entre 0 et 100 : le premier représente la fréquence du mot dans les offres d'emploi, le second sa fréquence dans les CV :

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
        ("R", 50, 40), ("machine learning", 80, 20),
        ("statistics", 20, 60), ("data science", 60, 70),
        ("analytics", 90, 3), ("team player", 85, 85),
        ("dynamic", 2, 90), ("synergies", 70, 0),
        ("actionable insights", 40, 30), ("think out of the box", 45, 10),
        ("self-starter", 30, 50),
        ("customer focus", 65, 15), ("thought leadership", 35, 35)]
```

La méthode consiste à organiser les mots sur une page pour donner une image « artistique » (figure 21-1).

Figure 21–1

Un nuage de mots à la mode



Tout cela est bien joli, mais cela ne nous apprend pas grand-chose. Une méthode plus intéressante consiste à répartir les mots de sorte que la position horizontale reflète la popularité dans les offres et la position verticale celle dans les CV (figure 21-2) :

```
from matplotlib import pyplot as plt
def text_size(total: int) -> float:
    """égale 8 si total vaut 0, 28 si total égale 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
```

```
plt.xlabel("Popularité dans les offres")
plt.ylabel("Popularité dans les CV")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

Figure 21–2

Un nuage de mots plus significatif
(mais moins artistique)



Modèles n-grammes

La responsable Marketing et SEO de DataSciencester veut créer des milliers de pages web sur le thème de la data science afin que le site bénéficie d'un meilleur classement dans les résultats de recherche sur les termes relatifs à ce sujet. Vous essayez de lui expliquer que les algorithmes des moteurs de recherche sont assez intelligents pour que cette approche ne donne rien, mais elle refuse de vous écouter.

Évidemment, elle n'a aucune intention d'écrire des milliers de pages web ni de payer une armée de spécialistes de la stratégie de contenu pour le faire. À la place, elle vous demande si vous pouvez générer ce genre de pages automatiquement, à l'aide d'un programme. Pour cela, vous devrez trouver un moyen de modéliser le langage.

Une méthode consiste à utiliser un corpus de documents pour entraîner un modèle statistique du langage. Dans notre cas, nous commencerons avec l'essai *What is data science ?*¹ de Mike Loukides (O'Reilly, non traduit).

Comme au chapitre 9, nous utiliserons les modules `Requests` et `Beautiful Soup` pour récupérer les données. Plusieurs problèmes méritent qu'on s'y attarde.

Tout d'abord, les apostrophes dans le texte sont représentées par le caractère Unicode `u"\u2019"`. Nous allons créer une fonction support pour les remplacer par des apostrophes normales :

```
def fix_unicode(text: str) -> str:
    return text.replace(u"\u2019", "'")
```

1. <https://www.oreilly.com/radar/what-is-data-science/>

Le second problème est que, une fois que nous avons récupéré le texte de la page web, il faut le diviser en séquence de mots et de points (pour pouvoir signifier les endroits où se terminent les phrases). Pour ce faire, nous utilisons `re.findall` :

```
import re
from bs4 import BeautifulSoup
import requests
url = "https://www.oreilly.com/ideas/what-is-data-science"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
# trouver les div de l'article
content = soup.find("div", "main-post-radar-content")
regex = r"[\w']+|[\.]" # correspond à un mot ou un point
document = []
for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

Nous pourrions certainement (et devrions) nettoyer encore un peu les données. Notamment, nous avons parfois découpé les phrases sur la base de points qui se trouvent en milieu de phrase (par exemple *Web 2.0*). Ceci dit, nous allons travailler avec ce document tel qu'il se présente.

Maintenant que nous avons le texte sous forme d'une séquence de mots, nous pouvons modéliser le langage de la manière suivante : à partir d'un mot de départ (comme *book*), nous regardons tous les mots qui le suivent dans le document source. Nous choisissons un de ces mots au hasard pour être le mot suivant et nous répétons ce processus jusqu'à ce que nous rencontrions un point, qui spécifie la fin de la phrase. Nous appelons cela un *modèle bigramme*, car il est complètement déterminé par les fréquences des bigrammes (les paires de mots) dans les données d'origine.

Que dire du mot de départ ? Nous pouvons simplement le prendre au hasard parmi les mots qui suivent un point. Pour commencer, faisons un précalcul des transitions. N'oublions pas que `zip` s'arrête lorsque n'importe laquelle de ses entrées est terminée. Donc, `zip(document, document[1:])` nous fournit précisément les paires d'éléments consécutifs du document :

```
from collections import defaultdict
transitions = defaultdict(list)
for prev, current in zip(document, document[1:]):
    transitions[prev].append(current)
```

Nous sommes prêts à générer des phrases :

```
def generate_using_bigrams() -> str:
    current = "." # signifie que le mot suivant commence une phrase
    result = []
    while True:
        next_word_candidates = transitions[current] # bigrammes (current,_)
        current = random.choice(next_word_candidates) # en choisir un
        result.append(current) # l'ajouter à la fin des résultats
        if current == ".":
            return " ".join(result) # si "." c'est fini
```

Les phrases produites sont du charabia, mais c'est le genre de charabia que vous pourriez publier sur votre site pour avoir l'air d'un expert. Voici un exemple :

If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.

Cela correspond en français à quelque chose comme :

Si vous savez peut-être qui sont vous voulez données trier les données flux web amis quelqu'un sur les sujets tendances que les données dans Hadoop est la science des données exige un livre démontre pourquoi les visualisations sont, mais nous faisons des corrélations massives dans de nombreux lecteurs de disques commerciaux en langage Python et crée plus traitables connexions sous forme de prise utilisent ensuite et l'utilise pour résoudre une donnée.

Nous pouvons rendre les phrases moins incompréhensibles en examinant les trigrammes, ou triplets de mots consécutifs. Plus généralement, vous pouvez utiliser des n -grammes constitués de n mots consécutifs, mais trois sont déjà suffisants pour nous. Les transitions vont dépendre des deux mots précédents :

```
trigram_transitions = defaultdict(list)
starts = []
for prev, current, next in zip(document, document[1:], document[2:]):
    if prev == ".": # si le mot précédent est un point
        starts.append(current) # alors ceci est un mot de départ
    trigram_transitions[(prev, current)].append(next)
```

Remarquez que nous devons à présent suivre les mots de départ séparément. Nous pouvons générer des phrases d'une manière très similaire :

```
def generate_using_trigrams() -> str:
    current = random.choice(starts) # choisir un mot de départ aléatoirement
    prev = "." # et le faire précéder d'un '.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)
        prev, current = current, next_word
        result.append(current)
        if current == ".":
            return " ".join(result)
```

Les phrases produites sont quelque peu meilleures :

In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That's not a question we could even have asked a few years there has been instrumented.

(Avec le recul MapReduce est comme une épidémie et, si tel est le cas, cela nous donne de nouvelles perspectives sur la façon dont les économies fonctionnent. Ce n'est pas une question que nous aurions pu poser il y a quelques années qui a été instrumentée.)

Bien sûr, le résultat semble meilleur car, à chaque étape, le processus de génération a moins de choix. Souvent, il n'a même qu'une seule possibilité. Cela veut dire que, fréquemment, les phrases générées (au moins les longues phrases) sont présentes telles qu'elles figurent dans l'original. Disposer d'une grande quantité de données est un avantage, de même que collecter des *n*-grammes de sources variées sur le même thème.

Grammaires

Une approche différente de la modélisation de la langue consiste à utiliser des grammaires, autrement dit des règles pour générer des phrases acceptables. À l'école primaire, vous avez sans doute appris les différentes composantes du discours et comment les combiner. Par exemple, si votre instituteur était vraiment très mauvais, vous diriez qu'une phrase consiste obligatoirement en un nom suivi d'un verbe. Si vous disposez d'une liste de noms et de verbes, vous pouvez générer des phrases en respectant cette règle.

Définissons une grammaire un peu plus complexe :

```
from typing import List, Dict
# taper un alias pour référencer des grammaires par la suite
Grammar = Dict[str, List[str]]
grammar = {"_S" : ["_NP _VP"],
          "_NP" : ["_N",
                  "_A _NP _P _A _N"],
          "_VP" : ["_V",
                  "_V _NP"],
          "_N" : ["data science", "python", "regression"],
          "_A" : ["big", "linear", "logistic"],
          "_P" : ["about", "near"],
          "_V" : ["learns", "trains", "tests", "is"]}
```

J'ai imaginé la convention suivant laquelle les noms commençant par des tirets bas sont des règles qui doivent être développées, tandis que les autres noms sont des terminaisons qui n'ont pas besoin d'être travaillées davantage.

Ainsi, par exemple, la phrase "_S" (*sentence*) est la règle qui produit une règle de syntagme nominal "_NP" (*noun phrase*) suivie par la règle de syntagme verbal "_VP" (*verb phrase*).

Le syntagme verbal peut produire la règle "_V" (verbe) soit seule, soit suivie d'un syntagme nominal.

Notez que la règle "_NP" se contient elle-même dans un de ses produits. La récursivité permet à des grammaires finies comme celle-ci de produire une infinité de phrases différentes.

Comment générer des phrases à partir de cette grammaire ? Commençons par une liste contenant la règle de phrase ["_S"]. Développons de manière répétitive chaque règle en la remplaçant par une autre choisie au hasard dans ses productions. Nous nous arrêterons lorsque nous serons arrivés à une liste qui ne contient que des terminaisons.

Par exemple, notre progression pourrait ressembler à ce qui suit :

```
['_S']
['_NP', '_VP']
['_N', '_VP']
['Python', '_VP']
['Python', '_V', '_NP']
['Python', 'trains', '_NP']
['Python', 'trains', '_A', '_NP', '_P', '_A', '_N']
['Python', 'trains', 'logistic', '_NP', '_P', '_A', '_N']
['Python', 'trains', 'logistic', '_N', '_P', '_A', '_N']
['Python', 'trains', 'logistic', 'data science', '_P', '_A', '_N']
['Python', 'trains', 'logistic', 'data science', 'about', '_A', '_N']
['Python', 'trains', 'logistic', 'data science', 'about', 'logistic', '_N']
['Python', 'trains', 'logistic', 'data science', 'about', 'logistic', 'Python']
```

Comment mettre en œuvre ce mécanisme ? Pour commencer, nous allons créer une fonction support simple pour identifier les terminaisons :

```
def is_terminal(token: str) -> bool:
    return token[0] != "_"
```

Ensuite, nous devons écrire une fonction pour transformer une liste d'unités lexicales (*token*) en une phrase. Cherchons la première unité non terminale. Si nous n'en trouvons aucune, c'est que nous sommes en présence d'une phrase complète : nous avons terminé.

Si nous trouvons une règle non terminale, nous choisissons au hasard une de ses productions. Si cette production est une terminaison (c'est-à-dire un mot), nous remplaçons l'unité par ce mot. Sinon, c'est une séquence d'unités non terminales séparées par des espaces que nous devons séparer et ensuite raccorder avec les unités en cours. Dans tous les cas, nous répéterons le processus sur le nouvel ensemble d'unités :

```
def expand(grammar: Grammar, tokens: List[str]) -> List[str]:
    for i, token in enumerate(tokens):
        # s'il s'agit d'une unité terminale, l'ignorer
        if is_terminal(token): continue
        # sinon, l'unité est non terminale,
        # donc nous devons choisir son remplaçant au hasard.
        replacement = random.choice(grammar[token])
        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            # le remplacement peut être, par exemple, "_NP _VP"
            # nous devons donc le diviser sur les espaces
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]
        # appel récursif sur la nouvelle liste d'éléments lexicaux
        return expand(grammar, tokens)
    # nous n'avons plus que des terminaisons donc nous avons terminé
    return tokens
```

Nous pouvons commencer à créer des phrases :

```
def generate_sentence(grammar: Grammar) -> List[str]:  
    return expand(grammar, ["_S"])
```

Essayez de changer la grammaire : ajoutez davantage de mots et de règles et insérez vos propres composantes discursives, jusqu'à ce que vous soyez prêt à générer autant de pages web que votre société le souhaite.

En réalité, les grammaires sont plus intéressantes lorsqu'on les utilise dans l'autre sens, c'est-à-dire pour analyser une phrase et identifier ses sujets et verbes, donc pour comprendre son sens.

Utiliser la data science pour générer des textes est assez amusant ; mais l'utiliser pour comprendre un texte est carrément magique. Vous trouverez à la section « Pour aller plus loin » une liste de bibliothèques utiles.

Aparté : échantillonnage de Gibbs

Il est facile de générer des échantillons à partir de certaines distributions. On peut, par exemple, obtenir des variables aléatoires de distribution uniforme :

```
random.random()
```

Et des variables aléatoires de distribution normale :

```
inverse_normal_cdf(random.random())
```

Toutefois, certaines distributions sont plus difficiles à échantillonner. *L'échantillonnage de Gibbs* est une technique de génération d'échantillons à partir de distributions multidimensionnelles. Elle s'avère utile lorsqu'on ne connaît que quelques-unes des distributions conditionnelles.

Prenons l'exemple d'un lancer de deux dés. Soit x la valeur du premier et y la somme des deux. Imaginons que vous vouliez générer de nombreuses paires (x, y) . Dans ce cas, il est facile de générer les échantillons directement :

```
from typing import Tuple  
import random  
def roll_a_die() -> int:  
    return random.choice([1, 2, 3, 4, 5, 6])  
  
def direct_sample() -> Tuple[int, int]:  
    d1 = roll_a_die()  
    d2 = roll_a_die()  
    return d1, d1 + d2
```


Imaginons maintenant que vous ne connaissiez que les distributions conditionnelles. La distribution de y sachant x est facile : si vous connaissez la valeur de x , y donnera, avec une probabilité égale, $x + 1$, $x + 2$, $x + 3$, $x + 4$, $x + 5$ ou $x + 6$:

```
def random_y_given_x(x: int) -> int:
    """probabilité égale de donner x + 1, x + 2, ... , x + 6"""
    return x + roll_a_die()
```

C'est en revanche plus compliqué dans l'autre sens. Par exemple, si vous savez que y vaut 2, alors nécessairement x vaut 1 (car la seule manière d'obtenir une somme égale à 2 avec deux dés est que chacun affiche 1). Si vous savez que y est égal à 3, alors x a autant de chances d'être égal à 1 qu'à 2. De même, si y est égal à 11, alors x doit être égal à 5 ou à 6 :

```
def random_x_given_y(y: int) -> int:
    if y <= 7:
        # Si le total est 7 ou moins, le premier dé a la même probabilité
        # de valoir 1, 2, ..., (total - 1).
        return random.randrange(1, y)
    else:
        # Si le total vaut plus de 7, le premier dé a la même probabilité
        # de valoir (total - 6), (total - 5), ..., 6.
        return random.randrange(y - 6, 7)
```

Comment réaliser l'échantillonnage de Gibbs ? Nous commençons avec une valeur quelconque (valide) pour x et y et nous répétons en alternance le remplacement de x par une valeur aléatoire sachant y et le remplacement de y par une valeur aléatoire sachant x . Après un certain nombre d'itérations, les valeurs résultantes de x et y représentent un échantillon de la distribution jointe non conditionnelle :

```
def gibbs_sample(num_iters: int = 100) -> Tuple[int, int]:
    x, y = 1, 2 # sans réelle importance
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

Vérifions que le résultat est similaire à celui de l'échantillonnage direct :

```
def comparer_distributions(num_samples: int = 1000) -> Dict[int, List[int]]:
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

Nous utiliserons cette technique dans la section suivante.

Modélisation thématique

Quand nous avons mis en place la recommandation « des experts que vous connaissez peut-être » au chapitre 1, nous avons simplement recherché des correspondances exactes parmi les centres d'intérêt des utilisateurs.

Pour comprendre les motivations de nos utilisateurs, il serait plus subtil d'essayer d'identifier les thèmes sous-jacents. La technique appelée *allocation de Dirichlet latente* (*Latent Dirichlet Analysis* – LDA) est couramment utilisée pour identifier des thèmes communs dans un ensemble de documents. Nous l'appliquerons aux documents qui représentent les centres d'intérêt de chaque utilisateur.

La LDA présente des similarités avec le classificateur de Bayes naïf que nous avons construit au chapitre 13 : comme lui, elle suppose un modèle probabiliste pour les documents. Avant de discuter des détails mathématiques les plus croustillants, voyons ce que suppose le modèle dans notre cas :

- il existe un nombre K fixe de thèmes ;
- il existe une variable aléatoire qui affecte à chaque thème une distribution probabiliste selon les mots. Vous pouvez considérer cette distribution comme la probabilité de rencontrer un mot m pour le thème donné k ;
- il existe une autre variable aléatoire qui affecte à chaque document une distribution probabiliste selon les thèmes. Vous pouvez considérer cette distribution comme le mélange de thèmes dans un document d ;
- chaque mot dans un document a été généré en choisissant d'abord un thème au hasard (à partir de la distribution de probabilité des thèmes du document) et en choisissant au hasard un mot (à partir de la distribution de probabilité des mots du thème).

En particulier, nous avons une collection de documents dont chacun est représenté sous la forme d'une liste de mots. Et nous avons une collection correspondante `documents_topics`, qui affecte un thème (ici, un nombre entre 0 et $K-1$) à chaque mot de chaque document.

Ainsi, le cinquième mot du quatrième document est `documents[3][4]` et le thème à partir duquel le mot a été choisi est `document_topics[3][4]`.

Cela définit la distribution de chaque document par rapport aux thèmes de manière explicite et la distribution de chaque thème par rapport aux mots de manière implicite.

Nous pouvons estimer la probabilité que le thème 1 produise un certain mot en comparant le nombre de fois qu'il produit ce mot avec le nombre de fois où il produit n'importe quel mot. De manière similaire, quand nous avons construit un filtre antispam au chapitre 13, nous avons comparé le nombre d'occurrences de chaque mot dans les spams avec le nombre total de mots apparaissant dans lesdits spams.

Bien que ces thèmes ne soient que des numéros, nous pouvons leur donner des noms plus évocateurs en examinant les mots auxquels ils accordent le plus de poids. Il nous suffit pour cela de générer `document_topics`. C'est là que l'échantillonnage de Gibbs entre en jeu.

Nous commençons par affecter à chaque mot de chaque document un thème complètement au hasard. Puis nous parcourons chaque document mot par mot. À chaque mot de chaque document, nous attribuons un poids pour chaque thème. Ces poids dépendent de la distribution (actuelle) des thèmes dans le document et de la distribution (actuelle) des mots pour ce thème. Nous utilisons ensuite ces poids pour échantillonner un nouveau thème pour ce mot. Si nous réitérons ce processus plusieurs fois, nous obtiendrons au final un échantillon représentatif combinant la distribution thème-mot et la distribution document-thème.

Pour commencer, il nous faut une fonction pour choisir au hasard un index fondé sur un jeu de poids arbitraires :

```
def sample_from(weights: List[float]) -> int:
    """retourne i avec la probabilité weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random() # uniforme entre 0 et total
    for i, w in enumerate(weights):
        rnd -= w # retourne le plus petit i tel que
        if rnd <= 0: return i # weights[0] + ... + weights[i] >= rnd
```

Par exemple, si vous lui donnez les poids [1, 1, 3], alors une fois sur cinq il retournera 0, une fois sur cinq il retournera 1 et trois fois sur cinq il retournera 2. Rédigeons un test :

```
from collections import Counter
# tirer 1 000 fois et compter
draws = Counter(sample_from([0.1, 0.1, 0.8])
                 for _ in range(1000))
assert 10 < draws[0] < 190 # doit être ~10 % ; test vraiment peu fiable
assert 10 < draws[1] < 190 # doit être ~10 % ; test vraiment peu fiable
assert 650 < draws[2] < 950 # doit être ~80 % ; test vraiment peu fiable
assert draws[0] + draws[1] + draws[2] == 1000
```

Nos documents sont les centres d'intérêt de nos utilisateurs, ils ressemblent à ceci :

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

Nous allons y rechercher $K = 4$ thèmes. Pour calculer les poids des échantillons, nous devons assurer le suivi de plusieurs compteurs. Créons tout d'abord les structures de données pour les accueillir.

- Nombre de fois que chaque thème est affecté à chaque document :

```
# liste de compteurs (Counter) ; un pour chaque document
document_topic_counts = [Counter() for _ in documents]
```

- Nombre de fois que chaque mot est affecté à chaque thème :

```
# liste de compteurs (Counter) ; un pour chaque thème
topic_word_counts = [Counter() for _ in range(K)]
```

- Nombre total des mots affectés à chaque thème :

```
# liste de nombres ; un pour chaque thème
topic_counts = [0 for _ in range(K)]
```

- Nombre total de mots contenus dans chaque document :

```
# liste de nombres ; un pour chaque document
document_lengths = [len(document) for document in documents]
```

- Nombre de mots distincts :

```
distinct_words = set(word for document in documents
                      for word in document)
w = len(distinct_words)
```

- Nombre de documents :

```
D = len(documents)
```

Une fois que nous avons renseigné ces compteurs, nous pouvons, par exemple, chercher le nombre des mots de `documents[3]` qui sont associés avec le thème 1 :

```
document_topic_counts[3][1]
```

Nous pouvons aussi chercher le nombre de fois où `nlp` est associé avec le thème 2 :

```
topic_word_counts[2]["nlp"]
```

Nous sommes enfin prêts à définir nos fonctions de probabilités conditionnelles. Comme au chapitre 13, chacune a un terme de lissage qui garantit que chaque thème a une chance non

nulle d'être choisi dans un document et que chaque mot a une chance non nulle d'être choisi pour un thème :

```
def p_topic_given_document(topic: int, d: int, alpha: float = 0,1) -> float:
    """Proportion des mots du document d
       qui sont affectés au thème (plus lissage)."""
    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word: str, topic: int, beta: float = 0,1) -> float:
    """Proportion des mots affectés à thème
       qui valent mot (plus lissage)."""
    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + W * beta))
```

Nous utiliserons ces fonctions pour calculer les poids destinés à mettre à jour les thèmes :

```
def topic_weight(d: int, word: str, k: int) -> float:
    """Soit un document et un mot de ce document,
       retourner le poids pour le thème de rang k."""
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

def choose_new_topic(d: int, word: str) -> int:
    return sample_from([topic_weight(d, word, k)
                       for k in range(K)])
```

De solides raisons mathématiques expliquent pourquoi `topic_weight` (poids_thème) est défini de cette façon, mais les détailler nous entraînerait trop loin. Heureusement, il est assez intuitif de penser que, pour un mot et son document, la probabilité de choix d'un thème quelconque dépend à la fois de la probabilité de ce thème dans le document et de celle du mot par rapport au thème.

C'est toute la mécanique dont nous avons besoin. Commençons par affecter chaque mot à un thème au hasard en mettant à jour nos différents compteurs :

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]
for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Notre but est de trouver un échantillon en jointure de la distribution thèmes-mots et de la distribution thèmes-documents. Pour cela, nous utiliserons une forme d'échantillonnage de Gibbs qui fait appel aux probabilités conditionnelles définies précédemment :

```
import tqdm
for iter in tqdm.trange(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):
            # Enlever ce mot-thème des compteurs
            # pour ne pas influencer les poids.
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1
            # choisir un nouveau thème en s'appuyant sur les poids
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic
            # et maintenant l'ajouter de nouveau aux compteurs
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

Que sont les thèmes ? Ce sont seulement des numéros 0, 1, 2 et 3. Si nous voulons leur donner des noms, il faudra le faire nous-mêmes. Voyons les cinq mots qui ont le poids le plus élevé pour chaque thème (tableau 21-1) :

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0:
            print(k, word, count)
```

Tableau 21-1 Les mots les plus courants dans chaque thème

Thème 0	Thème 1	Thème 2	Thème 3
Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probability	Cassandra	apprentissage automatique
artificial intelligence	pandas	NoSQL	neural networks

Sur la base de ce tableau, je donnerais sans doute les noms suivants aux thèmes :

```
topic_names = ["Big Data and programming languages",
               "python and statistics",
               "databases",
               "machine learning"]
```

À ce stade, nous pouvons voir comment le modèle affecte des thèmes à chaque centre d'intérêt des utilisateurs :

```
for document, topic_counts in zip(documents, document_topic_counts):
    print()
    print(document)
    for topic, count in topic_counts.most_common():
        if count > 0:
            print(topic_names[topic], count)
```

Cela donne :

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4
databases 3

['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5

['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5
machine learning 1
```

Et ainsi de suite. Étant donné les opérateurs « et » (*and*) nécessaires dans certains noms de thèmes, il est possible que nous devions utiliser davantage de thèmes, même s'il est plus probable qu'il n'y ait pas assez de données pour un apprentissage plus fin.

Vecteurs de mots

De très nombreuses avancées en TLN impliquent l'apprentissage profond. Dans le reste de ce chapitre, nous allons examiner deux cas faisant appel à la mécanique que nous avons développée au chapitre 19.

Une innovation importante implique une représentation des mots en tant que *vecteurs de mots* faiblement dimensionnés. Ces vecteurs peuvent être comparés, ajoutés les uns aux autres, fournis à des modèles d'apprentissage automatique ou manipulés de toute autre manière qui vous intéresse. Ils affichent généralement des propriétés intéressantes. Par exemple, des mots similaires ont tendance à présenter des vecteurs similaires. Typiquement, le vecteur de mot pour « grand » (*big*) est généralement assez proche de celui pour « gros » (*large*). Ainsi, un modèle fonctionnant avec des vecteurs de mots peut (dans une certaine mesure) gérer des propriétés comme la synonymie.

Les vecteurs font souvent aussi montre de propriétés arithmétiques plaisantes. Par exemple, dans certains modèles de ce type, si vous prenez le vecteur du mot « roi » (*king*), le soustrayez à celui du mot « homme » (*man*), puis ajoutez celui de « femme » (*woman*), vous obtiendrez un vecteur

très proche de celui de « reine » (*queen*). Il peut s'avérer intéressant de réfléchir à la signification de cet « apprentissage » des vecteurs de mots, même si nous n'y consacrerons pas notre temps ici.

Obtenir de tels vecteurs pour un lexique important est une entreprise difficile. Aussi, nous les apprendrons à partir d'un corpus de textes. Il existe deux schémas différents. Toutefois, à un degré d'abstraction supérieur, la tâche ressemble à ce qui suit.

- 1 Obtenir un ensemble de textes.
- 2 Créer un jeu de données où l'objectif consiste à prédire un mot selon des mots proches (ou, inversement, à prédire des mots proches selon un mot donné).
- 3 Entraîner un réseau neuronal à être efficace dans cette tâche.
- 4 Récupérer les états internes du réseau neuronal entraînés en tant que vecteurs de mots.

Plus particulièrement – sachant que la tâche consiste à prédire un mot selon des mots proches – des mots qui interviennent dans des contextes similaires devraient présenter des états internes similaires et donc des vecteurs similaires.

Ici, nous calculerons la « *similarité cosinus* », qui consiste en un nombre compris entre -1 et 1 mesurant dans quelle mesure deux vecteurs pointent dans la même direction :

```
from scratch.linear_algebra import dot, vector
import math
def cosine_similarity(v1: vector, v2: vector) -> float:
    return dot(v1, v2) / math.sqrt(
        dot(v1, v1) * dot(v2, v2))

assert cosine_similarity([1., 1, 1], [2., 2, 2]) == 1, "même direction"
assert cosine_similarity([-1., -1], [2., 2]) == -1, "directions opposées"
assert cosine_similarity([1., 0], [0., 1]) == 0, "directions orthogonales"
```

Procédons à l'apprentissage de quelques vecteurs de mots pour comprendre ce fonctionnement.

Pour commencer, nous devons installer un jeu de données factice. Les vecteurs généralement utilisés sont dérivés d'un entraînement sur des millions, voire des milliards de mots. Sachant que notre bibliothèque factice ne peut en gérer autant, nous allons créer un jeu de données artificiel auquel nous apporterons une certaine structure :

```
colors = ["red", "green", "blue", "yellow", "black", ""]
nouns = ["bed", "car", "boat", "cat"]
verbs = ["is", "was", "seems"]
adverbs = ["very", "quite", "extremely", ""]
adjectives = ["slow", "fast", "soft", "hard"]

def make_sentence() -> str:
    return " ".join([
        "The",
        random.choice(colors),
        random.choice(nouns),
        random.choice(verbs),
        random.choice(adverbs),
```



```

        random.choice(adjectives),
        ".")
NUM_SENTENCES = 50
random.seed(0)
sentences = [make_sentence() for _ in range(NUM_SENTENCES)]

```

Ce code va générer de nombreuses phrases de structure similaire, mais avec des mots différents ; par exemple, *The green boat seems quite slow* (« le bateau vert semble assez lent »). Sachant cette configuration, les couleurs figureront essentiellement dans des contextes « similaires », tout comme les noms et ainsi de suite. Donc, si nous affectons correctement les vecteurs de mots, les couleurs devraient bénéficier de vecteurs similaires et ainsi de suite.

Remarque

Dans la pratique, vous disposerez probablement d'un corpus de millions de phrases. Vous aurez alors un contexte « suffisant » issu de phrases telles quelles. Ici, avec seulement 50 phrases, nous devons les rendre quelque peu artificielles.

Comme nous l'avons vu auparavant (chapitre 19), nous voulons appliquer un encodage one-hot (« un parmi n ») à nos mots, ce qui signifie que nous avons besoin de les convertir en ID. Nous définirons donc une classe `Vocabulary` pour suivre cette mise en correspondance (ce mappage) mot-ID :

```

from scratch.deep_learning import Tensor
class Vocabulary:
    def __init__(self, words: List[str] = None) -> None:
        self.w2i: Dict[str, int] = {} # mappage word -> word_id
        self.i2w: Dict[int, str] = {} # mappage word_id -> word
        for word in (words or []):
            # Si des mots sont fournis,
            self.add(word)           # les ajouter.

    @property
    def size(self) -> int:
        """nombre de mots du vocabulaire"""
        return len(self.w2i)

    def add(self, word: str) -> None:
        if word not in self.w2i:
            # si les mots sont nouveaux
            word_id = len(self.w2i) # rechercher l'ID suivant
            self.w2i[word] = word_id # ajouter à la correspondance word -> word_id map
            self.i2w[word_id] = word # ajouter au word_id -> word map.

    def get_id(self, word: str) -> int:
        """retourner l'ID du mot (ou None)"""
        return self.w2i.get(word)

    def get_word(self, word_id: int) -> str:
        """retourner le mot ayant l'ID donné (ou None)"""
        return self.i2w.get(word_id)

```

```
def one_hot_encode(self, word: str) -> Tensor:
    word_id = self.get_id(word)
    assert word_id is not None, f"unknown word {word}"
    return [1.0 if i == word_id else 0.0 for i in range(self.size)]
```

Voici toutes les opérations que nous pouvons effectuer manuellement ; il est toutefois pratique d'en disposer dans une classe. Nous devrions probablement la tester :

```
vocab = Vocabulary(["a", "b", "c"])
assert vocab.size == 3, "vocab comprend 3 mots"
assert vocab.get_id("b") == 1, "b doit présenter word_id 1"
assert vocab.one_hot_encode("b") == [0, 1, 0]
assert vocab.get_id("z") is None, "z ne figure pas dans vocab"
assert vocab.get_word(2) == "c", "word_id 2 est c"
vocab.add("z")
assert vocab.size == 4, "vocab comprend à présent 4 mots"
assert vocab.get_id("z") == 3, "z devrait à présent présenter id 3"
assert vocab.one_hot_encode("z") == [0, 0, 0, 1]
```

Nous devrions également élaborer des fonctions d'aide simples pour enregistrer et charger un vocabulaire, tout comme nous l'avons fait pour nos modèles d'apprentissage profond :

```
import json
def save_vocab(vocab: Vocabulary, filename: str) -> None:
    with open(filename, 'w') as f:
        json.dump(vocab.w2i, f) # uniquement besoin d'enregistrer w2i

def load_vocab(filename: str) -> Vocabulary:
    vocab = Vocabulary()
    with open(filename) as f:
        # charger w2i et générer i2w à partir de celui-ci
        vocab.w2i = json.load(f)
        vocab.i2w = {id: word for word, id in vocab.w2i.items()}
    return vocab
```

Nous allons utiliser un modèle de vecteur de mot appelé *skip-gram*. Il accepte un mot en entrée et génère des probabilités quant à l'éventualité que d'autres mots soient proches de lui. Nous alimenterons notre modèle au moyen de paires d'entraînement (*word* « mot », *nearby_word* « mot_proche ») et nous nous efforcerons de minimiser la perte `SoftmaxCrossEntropy`.

Remarque

Autre modèle commun, *continuous bag-of-words* (CBOW) accepte les mots proches en tant qu'entrées et tente de prédire le mot d'origine.

Concevons notre réseau neuronal. En son cœur, nous placerons une couche d'incorporation (*embedding layer*), qui accepte un ID de mot en entrée et retourne un vecteur de mot. Sous le capot, il nous suffira d'utiliser une table de consultation.

Nous transmettrons alors le vecteur de mot à une couche `Linear` présentant le même nombre de sorties qu'il y a de mots dans notre vocabulaire. Comme auparavant, nous utiliserons `softmax` pour convertir ces sorties en probabilités sur nos mots proches. À mesure que nous utiliserons la descente de gradient pour entraîner le modèle, nous actualiserons les vecteurs dans la table de consultation. Une fois l'entraînement terminé, la table de consultation nous fournira nos vecteurs de mots.

Créons à présent la couche d'incorporation (*embedding*). En pratique, mieux vaut incorporer des éléments autres que des mots. Nous construirons donc une couche `Embedding` plus générale. Nous élaborerons ultérieurement une sous-classe `TextEmbedding` spécifiquement dédiée à des vecteurs de mots.

Dans son constructeur, nous allons fournir le nombre et la dimension de nos vecteurs, ce qui permettra à la couche de créer les incorporations (qui constitueront initialement des normales aléatoires standards) :

```
from typing import Iterable
from scratch.deep_learning import Layer, Tensor, random_tensor, \
    zeros_like

class Embedding(Layer):
    def __init__(self, num_embeddings: int, embedding_dim: int) -> None:
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        # Un vecteur de taille embedding_dim
        # pour chaque incorporation souhaitée.
        self.embeddings = random_tensor(num_embeddings, embedding_dim)
        self.grad = zeros_like(self.embeddings)
        # enregistrer le dernier ID entré
        self.last_input_id = None
```

Dans notre cas, nous n'incorporerons qu'un mot à la fois. Toutefois, dans d'autres modèles, mieux vaudra incorporer une séquence de mots et obtenir en retour une séquence de vecteurs (par exemple, si nous voulions entraîner le modèle CBOW). Aussi, un substitut conceptuel consisterait à accepter des séquences d'ID de mots. Nous nous en tiendrons au « un à la fois » par souci de simplicité.

```
def forward(self, input_id: int) -> Tensor:
    """Sélectionner le vecteur d'incorporation correspondant
    à l'ID d'entrée."""
    self.input_id = input_id # mémoriser pour la rétropropagation
    return self.embeddings[input_id]
```

Pour la passe vers l'arrière, nous obtiendrons un gradient correspondant au vecteur d'incorporation choisi et nous aurons besoin de construire le gradient correspondant pour `self.embeddings`, à savoir zéro pour chaque incorporation autre que celle choisie :

```
def backward(self, gradient: Tensor) -> None:
    # Mise à zéro du gradient correspondant à la dernière entrée
```

```
# bien moins coûteuse que de créer un nouveau tenseur de zéros
# à chaque fois.
if self.last_input_id is not None:
    zero_row = [0 for _ in range(self.embedding_dim)]
    self.grad[self.last_input_id] = zero_row
self.last_input_id = self.input_id
self.grad[self.input_id] = gradient
```

Sachant que nous disposons de paramètres et de gradients, nous devons surcharger les méthodes correspondantes :

```
def params(self) -> Iterable[Tensor]:
    return [self.embeddings]

def grads(self) -> Iterable[Tensor]:
    return [self.grad]
```

Nous souhaitons disposer d'une sous-classe spécifique pour les vecteurs de mots. Dans ce cas, le nombre de nos incorporations est déterminé par notre vocabulaire ; aussi nous transmettrons plutôt celui-ci :

```
class TextEmbedding(Embedding):
    def __init__(self, vocab: Vocabulary, embedding_dim: int) -> None:
        # appeler le constructeur de la superclasse
        super().__init__(vocab.size, embedding_dim)
        # et y rattacher le vocabulaire
        self.vocab = vocab
```

Les autres méthodes intégrées fonctionneront toutes telles quelles, mais nous ajouterons deux méthodes spécifiques pour fonctionner avec du texte. Par exemple, nous aimerions pouvoir récupérer le vecteur d'un mot donné. Ceci ne fait pas partie de l'interface `Layer`, mais nous restons toujours libres d'ajouter quelques méthodes en plus pour des couches spécifiques selon les besoins.

```
def __getitem__(self, word: str) -> Tensor:
    word_id = self.vocab.get_id(word)
    if word_id is not None:
        return self.embeddings[word_id]
    else:
        return None
```

Cette méthode *dunder* (*double-underscore*) nous permet de récupérer des vecteurs de mots en utilisant une indexation :

```
word_vector = embedding["black"]
```

Nous aimerions également que la couche d'incorporation nous révèle les mots les plus proches d'un mot donné :

```
def closest(self,
            word: str, n: int = 5) -> List[Tuple[float, str]]:
    """Retourne les n mots les plus proches
       selon une similarité cosinus."""
    vector = self[word]
    # Calculer des paires (similarity, other_word)
    # puis trier les plus similaires en premier.
    scores = [(cosine_similarity(vector, self.embeddings[i]), other_word)
              for other_word, i in self.vocab.w2i.items()]
    scores.sort(reverse=True)
    return scores[:n]
```

Notre couche d'incorporation se contente de générer en sortie des vecteurs avec lesquels nous alimentons une couche `Linear`.

À présent, nous sommes prêts à assembler nos données d'entraînement. Pour chaque mot entré, nous choisirons comme cibles les deux mots placés à sa gauche et les deux placés à sa droite.

Commençons par mettre les phrases en minuscules et à les diviser en mots :

```
import re
# cette regex n'est pas fantastique, mais fonctionne sur nos données
tokenized_sentences = [re.findall("[a-z]+|[.]", sentence.lower())
                       for sentence in sentences]
```

Construisons un vocabulaire :

```
# Créer un vocabulaire (à savoir, une correspondance word -> word_id)
# à partir de notre texte.
vocab = Vocabulary(word
                  for sentence_words in tokenized_sentences
                  for word in sentence_words)
```

Nous pouvons à présent créer des données d'entraînement :

```
from scratch.deep_learning import Tensor, one_hot_encode
inputs: List[int] = []
targets: List[Tensor] = []
for sentence in tokenized_sentences:
    for i, word in enumerate(sentence): # Pour chaque mot
        for j in [i - 2, i - 1, i + 1, i + 2]: # accepter les emplacements proches
            if 0 <= j < len(sentence): # qui ne sont pas hors limites
                nearby_word = sentence[j] # et les récupérer.
                # ajouter une entrée qui constitue l'ID_mot d'origine
                inputs.append(vocab.get_id(word))
                # ajouter une cible qui est le mot proche encodé un_parmi-n
                targets.append(vocab.one_hot_encode(nearby_word))
```

Avec la mécanique que nous avons construite, il est à présent facile de créer notre modèle :

```
from scratch.deep_learning import Sequential, Linear
random.seed(0)
EMBEDDING_DIM = 5 # semble une bonne taille
# définir la couche d'incorporation séparément, afin de la référencer
embedding = TextEmbedding(vocab=vocab, embedding_dim=EMBEDDING_DIM)
model = Sequential([
    # Soit un mot (en tant que vecteur de word_ids),
    # examiner son incorporation.
    embedding,
    # Et utiliser une couche linéaire pour calculer
    # des scores pour les "mots proches".
    Linear(input_dim=EMBEDDING_DIM,
           output_dim=vocab.size)])
```

En recourant à la mécanique du chapitre 19, il est facile d'entraîner notre modèle :

```
from scratch.deep_learning import SoftmaxCrossEntropy, Momentum,\
                                GradientDescent

loss = SoftmaxCrossEntropy()
optimizer = GradientDescent(learning_rate=0.01)
for epoch in range(100):
    epoch_loss = 0.0
    for input, target in zip(inputs, targets):
        predicted = model.forward(input)
        epoch_loss += loss.loss(predicted, target)
        gradient = loss.gradient(predicted, target)
        model.backward(gradient)
        optimizer.step(model)
    print(epoch, epoch_loss) # Imprimer la perte
    print(embedding.closest("black")) # et aussi quelques plus proches
    print(embedding.closest("slow")) # afin de voir ce qui a été
    print(embedding.closest("car")) # appris.
```

En examinant cet entraînement, vous constatez que les couleurs se rapprochent l'une de l'autre, de même que les adjectifs et les noms.

Une fois le modèle entraîné, il est amusant d'explorer les mots les plus similaires :

```
pairs = [(cosine_similarity(embedding[w1], embedding[w2]), w1, w2)
          for w1 in vocab.w2i
          for w2 in vocab.w2i
          if w1 < w2]
pairs.sort(reverse=True)
print(pairs[:5])
```

Dans mon cas, cela débouche sur ce qui suit :

```
[(0.9980283554864815, 'boat', 'car'),
 (0.9975147744587706, 'bed', 'cat'),
 (0.9953153441218054, 'seems', 'was'),
 (0.9927107440377975, 'extremely', 'quite'),
 (0.9836183658415987, 'bed', 'car')]
```

De toute évidence, *bed* (« lit ») et *cat* (« chat ») ne sont pas réellement similaires mais, dans nos phrases d'entraînement, ils en donnent l'impression. Et c'est cet aspect que le modèle capture.

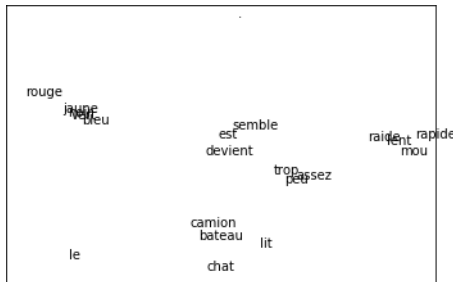
Nous pouvons également extraire les deux premières composantes principales et les représenter graphiquement :

```
from scratch.working_with_data import pca, transform
import matplotlib.pyplot as plt
# Extraire les deux premières composantes principales
# et transformer les vecteurs de mots.
components = pca(embedding.embeddings, 2)
transformed = transform(embedding.embeddings, components)
# Disperser les points (et les mettre en blanc
# pour qu'ils deviennent "invisibles").
fig, ax = plt.subplots()
ax.scatter(*zip(*transformed), marker='.', color='w')
# ajouter des annotations pour chaque mot à son emplacement transformé
for word, idx in vocab.w2i.items():
    ax.annotate(word, transformed[idx])
# masquer les axes
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.show()
```

On constate que les mots similaires sont effectivement regroupés (figure 21–3) :

Figure 21–3

Des vecteurs de mots



Si le sujet vous intéresse, les vecteurs de mots CBOW ne sont pas difficiles à entraîner. Cela implique seulement un peu de travail. Tout d'abord, vous devrez modifier la couche `Embedding` afin qu'elle accepte en entrée une liste d'ID et génère en sortie une liste de vecteurs d'incorporation. Ensuite, vous devrez créer une nouvelle couche (`Sum` ?) qui accepte une liste de vecteurs et retourne leur somme.

Chaque mot représente un exemple d'entraînement dans lequel l'entrée correspond aux ID des mots environnants et la cible à l'encodage one-hot du mot proprement dit.

La couche `Embedding` modifiée transforme les mots environnants en une liste de vecteurs, la nouvelle couche `Sum` réduit la liste de vecteurs à un vecteur unique. Une couche `Linear` peut alors produire des scores auxquels `softmax` peut s'appliquer pour obtenir une distribution représentant « les mots les plus probables compte tenu de ce contexte ».

J'ai trouvé le modèle CBOW plus difficile à entraîner que le modèle skip-gram, mais je vous encourage à essayer par vous-même.

Réseaux neuronaux récurrents

Les vecteurs de mots que nous avons développés dans la section précédente sont souvent utilisés en tant qu'entrées pour les réseaux neuronaux. Mettre cette mécanique en œuvre pose un problème : les phrases varient en longueur. Vous pouvez considérer une phrase de 3 mots comme un tenseur `[3, embedding_dim]` et une phrase de 10 mots comme un tenseur `[10, embedding_dim]`. Pour, disons, les transmettre à une couche `Linear`, nous devons donc intervenir sur cette première dimension de longueur variable.

Une possibilité consiste à utiliser une couche `Sum` (ou une variante qui récupère la moyenne) ; toutefois, l'ordre des mots dans une phrase est généralement important pour son sens. Pour prendre un exemple courant, « un chien mord un homme » et « un homme mord un chien » sont deux récits très différents !

Une autre manière de gérer cette situation consiste à utiliser des *réseaux neuronaux récurrents*, ou RNN (*Recurrent Neural Network*). Ces réseaux disposent d'un état masqué (*hidden*) qu'ils conservent entre les entrées. Dans le cas le plus simple, chaque entrée se combine avec l'état masqué courant pour produire une sortie. Celle-ci est alors utilisée en tant que nouvel état masqué. Ainsi, ces réseaux « se rappellent » (dans un sens) les entrées qu'ils ont vues et construisent une sortie finale qui dépend de toutes les entrées et de leur ordre.

Nous allons créer ce qui s'apparente à la couche RNN la plus simple possible. Elle acceptera une entrée unique (correspondant, par exemple, à un mot unique dans une phrase, ou à un caractère unique dans un mot) et maintiendra son état masqué entre les appels.

Rappelez-vous que notre couche `Linear` présentait un poids `w` et un biais `b`. Elle acceptait une entrée vectorielle et produisait un vecteur différent en sortie en recourant à la logique suivante :

```
output[o] = dot(w[o], input) + b[o]
```

Ici, nous souhaitons incorporer notre état masqué. Nous disposons donc de deux ensembles de poids – l'un à appliquer à l'entrée et l'autre à l'état masqué précédent :

```
output[o] = dot(w[o], input) + dot(u[o], hidden) + b[o]
```


Ensuite, nous utiliserons le vecteur de sortie en tant que nouvelle valeur de `hidden`. Si la modification n'est pas énorme, elle va, en revanche, permettre à nos réseaux de réaliser des choses extraordinaires.

```
from scratch.deep_learning import tensor_apply, tanh
class SimpleRnn(Layer):
    """sans doute la couche récurrente la plus simple possible"""
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.w = random_tensor(hidden_dim, input_dim, init='xavier')
        self.u = random_tensor(hidden_dim, hidden_dim, init='xavier')
        self.b = random_tensor(hidden_dim)
        self.reset_hidden_state()

    def reset_hidden_state(self) -> None:
        self.hidden = [0 for _ in range(self.hidden_dim)]
```

Vous constatez que nous commençons l'état masqué sous la forme d'un vecteur de 0 et que nous fournissons une fonction que les utilisateurs du réseau peuvent appeler pour réinitialiser l'état masqué.

Sachant cette configuration, la fonction `forward` est raisonnablement simple (elle l'est du moins si vous vous rappelez et comprenez le fonctionnement de notre couche `Linear`) :

```
def forward(self, input: Tensor) -> Tensor:
    self.input = input # enregistrer à la fois une entrée et un état
    # état masqué précédent à utiliser en rétropropagation
    self.prev_hidden = self.hidden
    a = [(dot(self.w[h], input) + # poids @ input
           dot(self.u[h], self.hidden) + # poids @ hidden
           self.b[h]) # biais
          for h in range(self.hidden_dim)]
    # appliquer une activation tanh
    self.hidden = tensor_apply(tanh, a)
    return self.hidden # et retourner le résultat
```

La passe vers l'arrière est similaire à celle de notre couche `Linear`, à ceci près qu'elle doit calculer un ensemble de gradients supplémentaires pour les poids `u` :

```
def backward(self, gradient: Tensor):
    # rétropropager via la tanh
    a_grad = [gradient[h] * (1 - self.hidden[h] ** 2)
               for h in range(self.hidden_dim)]
    # b présente le même gradient que a
    self.b_grad = a_grad
    # Chaque w[h][i] est multiplié par input[i] et ajouté à a[h],
    # ainsi, chaque w_grad[h][i] = a_grad[h] * input[i].
```

```

self.w_grad = [[a_grad[h] * self.input[i]
                 for i in range(self.input_dim)]
                for h in range(self.hidden_dim)]
# Chaque u[h][h2] est multipliée par hidden[h2] et ajoutée à a[h],]
# ainsi, chaque u_grad[h][h2] = a_grad[h] * prev_hidden[h2].
self.u_grad = [[a_grad[h] * self.prev_hidden[h2]
                 for h2 in range(self.hidden_dim)]
                for h in range(self.hidden_dim)]
# Chaque input[i] est multipliée par chaque w[h][i]
# et ajoutée à a[h],].
# ainsi, chaque input_grad[i] = sum(a_grad[h] * w[h][i])
return [sum(a_grad[h] * self.w[h][i]
            for h in range(self.hidden_dim))
        for i in range(self.input_dim)]

```

Et enfin, nous devons surcharger les méthodes `params` et `grads` :

```

def params(self) -> Iterable[Tensor]:
    return [self.w, self.u, self.b]

def grads(self) -> Iterable[Tensor]:
    return [self.w_grad, self.u_grad, self.b_grad]

```

Avertissement

Ce RNN est si simple que vous ne devriez probablement pas l'utiliser en pratique réelle.

Notre `SimpleRnn` présente deux caractéristiques indésirables. L'une tient à ce que l'état masqué entier est utilisé pour actualiser l'entrée à chaque fois que vous l'appellez. L'autre tient à ce que l'état masqué entier est écrasé à chaque fois que vous l'appellez. Ces deux caractéristiques rendent le modèle difficile à entraîner, tout particulièrement lorsqu'il s'agit d'assimiler des dépendances de grande portée.

C'est pour cette raison que pratiquement personne n'utilise ce type de RNN simple. Au lieu de cela, les développeurs font appel à des variantes plus sophistiquées telles que LSTM (*Long Short-Term Memory*) ou GRU (*Gated Recurrent Unit*). Ces variantes disposent de bien plus de paramètres et utilisent des « portes » (*gates* en anglais) paramétrées pour actualiser (et exploiter) uniquement une partie de l'état à chaque étape.

Ces variantes ne présentent pas de difficultés particulières. Toutefois, elles nécessitent un volume de code beaucoup plus important, ce qui (à mon avis) ne serait pas pour autant passionnant à lire. Le code dédié à ce chapitre, disponible sur GitHub, inclut une mise en œuvre LSTM. Je vous encourage à le consulter. Il n'en reste pas moins fastidieux et nous n'en discuterons pas plus avant ici.

Autre singularité de notre mise en œuvre, elle n'effectue qu'une seule « étape » à la fois et nous oblige à réinitialiser manuellement l'état masqué. Une mise en œuvre d'un RNN plus « pratique » pourrait accepter des séquences d'entrées, définir son état masqué à 0 au début de

chaque séquence et produire des séquences de sorties. La nôtre saurait sans aucun doute être modifiée pour se comporter de la sorte, mais là encore, la modification nécessiterait davantage de code et de sophistication pour un faible gain en compréhension.

Exemple : utilisation d'un RNN de niveau caractère

Fraîchement recruté, le nouveau responsable de l'image de marque n'est donc pas à l'origine du nom DataSciencester. Et, selon lui, un nom plus adapté pourrait mener à davantage de réussite pour l'entreprise. Il vous demande de recourir à la data science pour suggérer des substituts.

Une application sympathique des RNN consiste à leur faire utiliser des caractères (plutôt que des mots) en guise d'entrées, à les entraîner à apprendre les schémas de langage subtils d'un jeu de données, puis à générer des instances fictives à partir de ce dernier.

Par exemple, vous pouvez entraîner un RNN sur les noms de groupes musicaux alternatifs, utiliser le modèle ainsi entraîné pour générer de nouveaux noms de groupes alternatifs fictifs, puis sélectionner manuellement les plus drôles et les partager sur Twitter. Hilarité garantie !

Ayant vu ce truc suffisamment souvent pour ne plus le considérer comme un procédé intelligent, vous décidez de lui donner tout de même sa chance.

Après quelques recherches, vous constatez que *YCombinator* a publié une liste² de ses 102 meilleures start-up. Cette liste pourrait constituer un bon point de départ. En inspectant la page, vous constatez que les noms des entreprises se trouvent tous à l'intérieur des balises `<b class="h4">`. Il est donc facile d'utiliser des compétences en ratissage du Web pour les extraire :

```
from bs4 import BeautifulSoup
import requests
url = "https://www.ycombinator.com/topcompanies/"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
# nous obtenons les entreprises deux fois
# nous utilisons un ensemble en compréhension pour les dédupliquer
companies = list({b.text
                    for b in soup("b")
                    if "h4" in b.get("class", ())})

assert len(companies) == 101
```

Comme toujours, la page peut changer (ou disparaître), auquel cas ce code ne fonctionnera pas et il vous faudra utiliser vos toutes nouvelles compétences en data science pour le rectifier ou simplement récupérer la liste sur le site GitHub de l'ouvrage.

Revenons à notre stratégie... Étant donné le caractère courant et un état masqué représentant tous les caractères que nous avons vus jusqu'ici, nous allons entraîner un modèle à prédire le prochain caractère d'un nom.

2. <https://www.ycombinator.com/topcompanies/>

Comme à l'accoutumée, nous allons en fait prédire une distribution de probabilités sur des caractères et entraîner notre modèle à minimiser la perte `SoftmaxCrossEntropy`.

Une fois notre modèle entraîné, nous nous en servons pour générer des probabilités, échantillonner aléatoirement un caractère selon ces probabilités, puis fournir ce caractère en tant qu'entrée suivante. Ce processus nous permettra de générer des noms d'entreprise en utilisant les poids appris.

Pour commencer, nous devons construire un élément `vocabulary` à partir des caractères présents dans les noms :

```
vocab = Vocabulary([c for company in companies
                    for c in company])
```

En outre, nous utiliserons des *tokens* spéciaux pour signifier le début et la fin d'un nom d'entreprise. Le modèle saura ainsi apprendre par quels caractères commencer et finir un nom d'entreprise.

Nous utiliserons simplement les caractères *regex* pour le début et la fin ; ils n'apparaissent heureusement pas dans notre liste d'entreprises :

```
START = "^"
STOP = "$"
```

Nous devons également les ajouter au vocabulaire.

```
vocab.add(START)
vocab.add(STOP)
```

Dans le cas de notre modèle, nous appliquerons un encodage *one-hot* à chaque caractère, transmettrons ce dernier via deux `SimpleRnns`, puis utiliserons une couche `Linear` pour générer les scores de chaque caractère suivant possible :

```
HIDDEN_DIM = 32 # vous devez essayer des tailles différentes !
rnn1 = SimpleRnn(input_dim=vocab.size, hidden_dim=HIDDEN_DIM)
rnn2 = SimpleRnn(input_dim=HIDDEN_DIM, hidden_dim=HIDDEN_DIM)
linear = Linear(input_dim=HIDDEN_DIM, output_dim=vocab.size)
model = Sequential([rnn1, rnn2, linear])
```

Imaginons pour le moment que nous avons entraîné ce modèle. Rédigeons la fonction qui l'utilise pour générer de nouveaux noms d'entreprise. Utilisons à cette fin la fonction `sample_from` issue de la section « Modélisation thématique » :

```
from scratch.deep_learning import softmax
def generate(seed: str = START, max_len: int = 50) -> str:
    rnn1.reset_hidden_state() # réinitialiser les deux états masqués
    rnn2.reset_hidden_state()
    output = [seed] # commencer la sortie avec la graine spécifiée
```

```
# Poursuivre jusqu'à ce que nous produisions le caractère STOP
# ou que nous atteignons la longueur maximale.
while output[-1] != STOP and len(output) < max_len:
    # utiliser le dernier caractère en tant qu'entrée
    input = vocab.one_hot_encode(output[-1])
    # générer des scores en utilisant le modèle
    predicted = model.forward(input)
    # convertir en probabilités et extraire un ID de caractère aléatoire
    probabilities = softmax(predicted)
    next_char_id = sample_from(probabilities)
    # ajouter le caractère correspondant à notre sortie
    output.append(vocab.get_word(next_char_id))
# se débarrasser des caractères START et END et retourner le mot
return ''.join(output[1:-1])
```

Enfin, nous voici prêts à entraîner notre RNN de niveau caractère. Et cela va prendre un certain temps !

```
loss = SoftmaxCrossEntropy()
optimizer = Momentum(learning_rate=0.01, momentum=0.9)
for epoch in range(300):
    # entraîner chaque étape dans un ordre différent
    random.shuffle(companies)
    epoch_loss = 0 # suivre la perte
    for company in tqdm.tqdm(companies):
        rnn1.reset_hidden_state() # réinitialiser les deux états masqués
        rnn2.reset_hidden_state()
        company = START + company + STOP # ajouter les caractères START et STOP
        # Le reste consiste simplement en notre boucle d'entraînement
        # habituelle, à ceci près que les entrées et la cible
        # sont les caractères encodés one-hot précédents et suivants.
        for prev, next in zip(company, company[1:]):
            input = vocab.one_hot_encode(prev)
            target = vocab.one_hot_encode(next)
            predicted = model.forward(input)
            epoch_loss += loss.loss(predicted, target)
            gradient = loss.gradient(predicted, target)
            model.backward(gradient)
            optimizer.step(model)
    # pour chaque epoch, imprimer la perte et générer aussi un nom
    print(epoch, epoch_loss, generate())
    # baisser le rythme d'apprentissage pour les 100 dernières étapes
    # il n'y a aucune raison théorique mais cela semble fonctionner
    if epoch == 200:
        optimizer.lr *= 0.1
```

Une fois entraîné, le modèle génère des noms réels issus de la liste (ce qui n'est pas surprenant puisqu'il dispose d'une bonne capacité, mais non d'un grand nombre de données d'entraînement), ainsi que des noms qui sont légèrement différents de ceux d'entraînement (Scribe, Loinbare, Pozium), d'autres qui semblent authentiquement fruits d'une création (Benuus,

Clepto, Equite, Vivest) et des noms qui tiennent à la fois du déchet et du mot (SFitready, Sint ocanelp, GlyOx, Doorboronelhav).

Malheureusement, comme la plupart des sorties de RNN de niveau caractère, ces résultats sont seulement à moitié intéressants et le responsable de l'image de marque se retrouve dans l'incapacité de les exploiter.

Si je porte la dimension masquée à 64, j'obtiens bien plus de noms issus directement de la liste ; si je le ramène à 8, j'obtiens essentiellement des déchets.

Pour aller plus loin

- NLTK³ est une bibliothèque répandue d'outils TLN pour Python. Elle dispose de son propre livre complet, disponible en ligne⁴ gratuitement (en anglais).
- gensim⁵ est une bibliothèque Python destinée à la modélisation thématique. C'est un bien meilleur support que notre modèle de toutes pièces.
- spacy⁶ est une bibliothèque de traitement du langage naturel de classe industrielle en Python. Elle est également très répandue.
- Andrej Karpathy a publié son célèbre billet de blog intitulé *The Unreasonable Effectiveness of Recurrent Neural Networks*⁷ (en anglais). C'est une lecture particulièrement précieuse.
- Mon travail au quotidien consiste, entre autres, à construire AllenNLP⁸, une bibliothèque Python dédiée à la recherche TLN (du moins, c'est ce que je faisais à l'heure de la mise sous presse du présent ouvrage). La bibliothèque dépasse la portée de ce livre, mais elle reste intéressante pour vous et dispose d'une démonstration interactive attrayante de nombreux modèles TLN dernier cri.

3. <http://www.nltk.org/>

4. <http://www.nltk.org/book/>

5. <https://radimrehurek.com/gensim/>

6. <https://spacy.io/>

7. <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

8. <https://allennlp.org/>

Analyse des réseaux

Vos connexions à tout ce qui vous entoure définissent littéralement qui vous êtes.

Aaron O'Connell

De nombreux problèmes de données peuvent être étudiés fructueusement en les abordant en termes de *réseaux* constitués de *nœuds* et d'*arêtes* de jonction.

Par exemple, vos amis sur Facebook sont les nœuds d'un réseau dont les arêtes sont les relations d'amitié. Le Web, lui-même, pour lequel chaque page est un nœud et chaque hyperlien de page à page est une arête, est un autre exemple moins évident.

L'amitié sur Facebook est réciproque : si je suis ami avec vous sur ce site, alors vous êtes forcément ami avec moi. Dans ce cas, on dit que les *arêtes* ne sont *pas orientées*. Les hyperliens, eux, le sont : ainsi, mon site contient un hyperlien vers la Maison-Blanche, mais (pour des raisons que j'ignore) whitehouse.gov refuse de mettre un lien vers mon site. On parle alors d'*arêtes orientées*. Nous allons examiner ces deux types de réseaux.

Centralité internœud

Au chapitre 1, nous avons identifié les connecteurs clés du réseau DataSciencester en comptant le nombre d'amis de chaque utilisateur. Désormais, nous sommes suffisamment bien équipés en mécanique pour étudier d'autres approches. Nous allons utiliser le même réseau, mais avec des éléments `NamedTuples` pour les données.

Souvenez-vous que ce réseau (figure 22–1) comprend des utilisateurs :

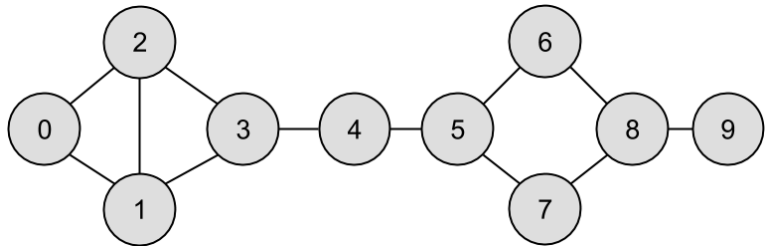
```
from typing import NamedTuple
class User(NamedTuple):
    id: int
    name: str

users = [User(0, "Hero"), User(1, "Dunn"),
          User(2, "Sue"), User(3, "Chi"),
          User(4, "Thor"), User(5, "Clive"),
          User(6, "Hicks"), User(7, "Devin"),
          User(8, "Kate"), User(9, "Klein")]
```

Ils sont liés par des liens d'amitié :

```
friend_pairs = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
                (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Figure 22–1
Le réseau DataSciencester



Les liens d'amitié sont plus faciles à exploiter sous forme de dict :

```
from typing import Dict, List
# nouveau type pour les amitiés
Friendships = Dict[int, List[int]]

friendships: Friendships = {user.id: [] for user in users}
for i, j in friend_pairs:
    friendships[i].append(j)
    friendships[j].append(i)

assert friendships[4] == [3, 5]
assert friendships[8] == [6, 7, 9]
```

Nous n'étions pas satisfaits de la notion de *centralité de degré*, qui n'était pas en phase avec notre intuition pour déterminer les connecteurs clés du réseau.

Une autre mesure possible est la *centralité internœud*, qui identifie les personnes qui sont fréquemment sur les chemins les plus courts entre des paires d'autres personnes. La centralité internœud d'un nœud i se calcule en ajoutant, pour chaque paire (j, k) , la proportion des plus courts chemins entre le nœud j et le nœud k qui passent par i .

Par exemple, pour déterminer la centralité internœud de `Thor`, nous devons calculer tous les chemins les plus courts entre toutes les paires d'utilisateurs qui ne sont pas `Thor`. Ensuite, il faudra compter combien de ces chemins les plus courts passent par `Thor`. Par exemple, le seul chemin le plus court entre `Chi` (id 3) et `Clive` (id 5) passe par `Thor`, alors que ce n'est le cas d'aucun des deux chemins les plus courts entre `Hero` (id 0) et `Chi` (id 3).

Dans un premier temps, nous devons donc calculer tous les chemins les plus courts entre toutes les paires d'utilisateurs. Il existe pour cela des algorithmes très élaborés et très efficaces. Cependant (comme dans la plupart des cas de ce livre), nous préférons utiliser un algorithme moins efficace, mais plus facile à comprendre.

Cet algorithme (une mise en œuvre de l'algorithme de parcours en profondeur) est un des plus compliqués de ce livre et va donc réclamer toute votre attention.

- 1 Notre but est de construire une fonction qui part d'un utilisateur `from_user` et recherche tous les chemins les plus courts vers chaque autre utilisateur.
- 2 Nous représenterons un chemin sous la forme d'une liste d'identifiants d'utilisateurs. Comme chaque chemin part de `from_user`, nous n'incluons pas son identifiant dans la liste. Cela signifie que la longueur de la liste représentant le chemin correspondra à la longueur du chemin lui-même.
- 3 Nous maintiendrons un dictionnaire `shortest_paths_to` dont chaque clé est l'identifiant d'un utilisateur et la valeur est une liste des chemins qui mènent vers ce dernier. S'il existe un unique chemin le plus court, la liste ne contiendra que ce chemin. S'il existe plusieurs chemins les plus courts, la liste les contiendra tous.
- 4 Nous aurons aussi une file d'attente `frontier` qui contient les utilisateurs dans l'ordre dans lequel nous souhaitons les explorer. Nous les stockerons sous forme de paires (`prev_user`, `user`) pour déterminer comment nous sommes parvenus à chacun d'entre eux. Nous initialiserons la file d'attente avec tous les voisins de `from_user`.

Remarque

Nous n'avons pas encore parlé des files d'attente, qui sont des structures de données optimisées pour les opérations « ajouter à la fin » et « retirer à partir du début ». En Python, elles sont mises en œuvre sous la forme de `collections.deque`. Il s'agit, en fait, d'une file d'attente à double extrémité.

- 5 Tandis que nous explorons le graphe, à chaque fois que nous trouvons de nouveaux voisins pour lesquels nous n'avons pas encore identifié de chemins les plus courts, nous les ajoutons à la fin de la file d'attente pour les explorer plus tard, l'utilisateur actuel tenant lieu de `prev_user`.
- 6 Si nous retirons de la file d'attente un utilisateur que nous n'avons pas encore rencontré, cela signifie que nous avons trouvé un ou plusieurs chemins les plus courts menant à lui : ces chemins correspondent à chacun des chemins les plus courts vers `prev_user` avec l'ajout d'une étape supplémentaire.
- 7 Si nous retirons de la file d'attente un utilisateur que nous avons déjà rencontré auparavant, soit nous avons trouvé un autre chemin le plus court (et dans ce cas nous devons l'ajouter), soit nous avons trouvé un chemin plus long (auquel cas il ne faut pas en tenir compte).

- 8 S'il ne reste aucun utilisateur dans la file d'attente, cela veut dire que nous avons exploré tout le graphe (ou du moins les parties qu'il est possible d'atteindre à partir de l'utilisateur de départ) et donc que nous avons fini. Nous pouvons réunir le tout dans une fonction (complexe) :

```
from collections import deque
Path = List[int]
def shortest_paths_from(from_user_id: int,
                        friendships: Friendships) -> Dict[int, List[Path]]:
    # Dictionnaire de tous les chemins les plus courts vers
    # l'utilisateur référencé en clé.
    shortest_paths_to: Dict[int, List[Path]] = {from_user_id: [[]]}
    # La file d'attente de (previous user, next user) à vérifier
    # commence avec toutes les paires (from_user, friend_of_from_user).
    frontier = deque((from_user_id, friend_id)
                     for friend_id in friendships[from_user_id])
    # continuer jusqu'à vider la file d'attente
    while frontier:
        # retirer la paire qui suit dans la file d'attente
        prev_user_id, user_id = frontier.popleft()
        # Vue la manière dont nous ajoutons des éléments à la file d'attente,
        # nous connaissons forcément quelques plus courts chemins menant
        # à prev_user.
        paths_to_prev_user = shortest_paths_to[prev_user_id]
        new_paths_to_user = [path + [user_id]
                             for path in paths_to_prev_user]
        # il est possible que nous connaissions déjà un chemin le plus court
        old_paths_to_user = shortest_paths_to.get(user_id, [])
        # chemin le plus court parmi ceux que nous avons déjà vus
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')
        # ne garder que les chemins pas trop longs et nouveaux
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]
        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user
        # ajouter les voisins jusqu'ici inconnus à la file d'attente
        frontier.extend((user_id, friend_id)
                        for friend_id in friendships[user_id]
                        if friend_id not in shortest_paths_to)
    return shortest_paths_to
```

À présent, calculons tous les chemins les plus courts :

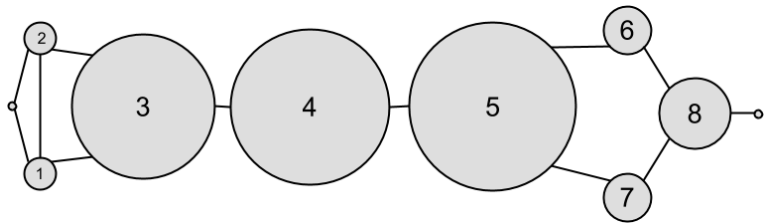
```
# pour chaque from_user, pour chaque to_user, lister les plus courts chemins
shortest_paths = {user.id: shortest_paths_from(user.id, friendships)
                  for user in users}
```

Et nous voilà prêts à déterminer la centralité internœud. Pour chaque paire de nœuds i et j , nous connaissons les n plus courts chemins de i à j . Et pour chacun de ces chemins, nous ajoutons seulement $1 / n$ à la centralité de chaque nœud sur le chemin :

```
betweenness centrality = {user.id: 0.0 for user in users}
for source in users:
    for target_id, paths in shortest_paths[source.id].items():
        if source.id < target_id: # ne pas compter en double
            num_paths = len(paths) # nombre des chemins les plus courts
            contrib = 1 / num_paths # contribution à la centralité
            for path in paths:
                for between_id in path:
                    if between_id not in [source.id, target_id]:
                        betweenness centrality[between_id] += contrib
```

Comme le montre la figure 22–2, les utilisateurs 0 et 9 ont une centralité de 0 (aucun d'entre eux n'est sur un plus court chemin entre les autres utilisateurs), alors que 3, 4 et 5 présentent tous une centralité élevée (car tous les trois se trouvent sur plusieurs plus courts chemins).

Figure 22–2
Le réseau DataSciencester
dimensionné par centralité
internœud



Remarque

En général, les valeurs de centralité n'ont pas de sens en elles-mêmes. Ce qui nous intéresse, c'est la comparaison des valeurs de chaque nœud par rapport aux autres.

Nous pouvons observer une autre mesure ; la *centralité de proximité*. Pour cela, nous calculons l'éloignement de chaque utilisateur, c'est-à-dire la somme des longueurs de leurs plus courts chemins vers chacun des autres utilisateurs. Comme nous avons déjà calculé les plus courts chemins entre chaque paire de nœuds, il est facile d'ajouter leurs longueurs. En cas de chemins les plus courts multiples, ceux-ci affichent tous la même longueur, il suffit alors de prendre le premier.

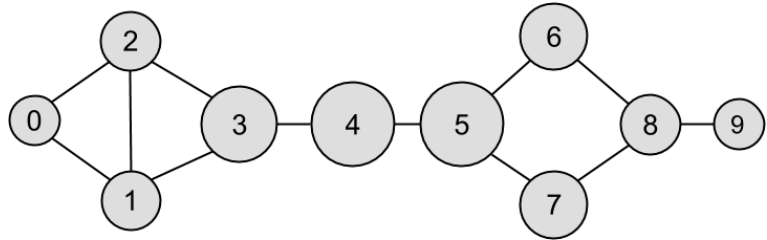
```
def farness(user_id: int) -> float:
    """Somme des longueurs des plus courts chemins
    vers chaque autre utilisateur."""
    return sum(len(paths[0])
               for paths in shortest_paths[user_id].values())
```

Il ne manque plus grand-chose pour calculer la centralité de proximité (figure 22–3) :

```
closeness centrality = {user.id: 1 / farness(user.id)
                        for user in users}
```

Figure 22–3

Le réseau DataSciencester dimensionné par centralité de proximité



Ici, les variations sont beaucoup plus réduites, car même les nœuds les plus centraux restent très éloignés des nœuds périphériques.

Comme nous l'avons vu, calculer les plus courts chemins est assez pénible. Pour cette raison, la centralité internœud et la centralité de proximité sont rarement utilisées avec les grands réseaux. Le calcul moins intuitif (mais généralement plus facile à effectuer) de la centralité de vecteur propre est plus fréquent.

Centralité de vecteur propre

Avant d'examiner la *centralité de vecteur propre*, nous devons d'abord parler des vecteurs propres et, avant cela, de la multiplication matricielle.

Multiplication matricielle

Si A est une matrice $n \times m$ et B une matrice $m \times k$ (remarquez que la deuxième dimension de A est identique à la première de B), leur produit AB est la matrice $n \times k$ dont l'entrée de rang (i, j) est :

$$A_{i1}B_{1j} + A_{i2}B_{2j} + \dots + A_{im}B_{mj}$$

Cela correspond au produit scalaire de la rangée i de A (vue comme un vecteur) par la colonne j de B (également vue comme un vecteur).

Mettons cette mécanique en œuvre en utilisant la fonction `make_matrix` du chapitre 4 :

```
from scratch.linear_algebra import Matrix, make_matrix, shape
def matrix_times_matrix(m1: Matrix, m2: Matrix) -> Matrix:
    nr1, nc1 = shape(m1)
    nr2, nc2 = shape(m2)
    assert nc1 == nr2, "nbre de colonnes de m1 doit être égal au nbre de lignes de m2"
```

```
def entry_fn(i: int, j: int) -> float:
    """produit dot de la rangée i de m1 par la colonne j de m2"""
    return sum(m1[i][k] * m2[k][j] for k in range(nc1))

return make_matrix(nr1, nc2, entry_fn)
```

Si nous considérons un vecteur de dimension m comme une matrice $(m, 1)$, nous pouvons le multiplier par une matrice (n, m) pour obtenir une matrice $(n, 1)$, que nous pouvons alors considérer comme un vecteur de dimension n .

En d'autres termes, une autre manière de considérer une matrice (n, m) consiste en une façon de transformer des vecteurs de dimension m en vecteurs de dimension n :

```
from scratch.linear_algebra import Vector, dot
def matrix_times_vector(m: Matrix, v: Vector) -> Vector:
    nr, nc = shape(m)
    n = len(v)
    assert nc == n, "nbre de colonnes de m doit être égal à nbre d'éléments de v"
    return [dot(row, v) for row in m] # sortie de longueur nr
```

Quand A est une matrice carrée, cette opération fait correspondre des vecteurs de dimension n à d'autres vecteurs de dimension n . Il est possible que, pour certaines matrices A et certains vecteurs v , nous obtenions v multiplié par un scalaire. Cela revient à dire que le résultat peut être un vecteur qui pointe dans la même direction que v . Dans un tel cas (et quand, en plus, v n'est pas un vecteur de zéros), v est appelé *vecteur propre* de A . Et le multiplicateur est appelé *valeur propre* (eigenvalue).

Une méthode possible pour trouver un vecteur propre de A consiste à récupérer un vecteur initial v , à lui appliquer `matrix_times_vector`, puis à changer l'échelle du résultat pour qu'il soit de longueur 1 et à répéter jusqu'à la convergence du processus :

```
from typing import Tuple
import random
from scratch.linear_algebra import magnitude, distance
def find_eigenvector(m: Matrix,
                    tolerance: float = 0.00001) -> Tuple[Vector, float]:
    guess = [random.random() for _ in m]
    while True:
        result = matrix_times_vector(m, guess) # transformer guess
        norm = magnitude(result) # calculer la norme
        next_guess = [x / norm for x in result] # changer l'échelle
        if distance(guess, next_guess) < tolerance:
            # convergence : retourner (eigenvector, eigenvalue)
            return next_guess, norm
        guess = next_guess
```

Par construction, la valeur `guess` retournée est un vecteur tel que, si vous lui appliquez `matrix_times_vector` et changez l'échelle pour obtenir la longueur 1, alors vous récupérez en retour un vecteur très proche. Cela signifie que c'est un vecteur propre.

Toutes les matrices de nombres réels n'ont pas forcément des vecteurs propres et des valeurs propres. Prenons un exemple :

```
rotate = [[ 0, 1],
          [-1, 0]]
```

Cette matrice fait tourner les vecteurs de 90° dans le sens des aiguilles d'une montre, ce qui veut dire que le seul vecteur auquel elle fait correspondre ce même vecteur multiple d'un scalaire est un vecteur de zéros. Si vous tentez `find_eigenvector(rotate)`, le programme ne s'arrêtera jamais. Même les matrices présentant des vecteurs propres peuvent parfois être bloquées indéfiniment. Voici un exemple :

```
flip = [[0, 1],
        [1, 0]]
```

Cette matrice fait correspondre à tout vecteur $[x, y]$ le vecteur $[y, x]$. Cela implique que, par exemple, $[1, 1]$ est un vecteur propre de valeur propre 1. Cependant, si vous commencez avec un vecteur aléatoire aux coordonnées non égales, `find_eigenvector` va simplement faire alterner les coordonnées pour toujours. Les bibliothèques qui ne partent pas de rien, comme NumPy, utilisent des méthodes différentes qui fonctionneraient dans ce cas. Cependant, quand `find_eigenvector` retourne un résultat, cela signifie que le vecteur en question est bien un vecteur propre.

Centralité

En quoi tout ceci nous aide-t-il à comprendre le réseau DataSciencester ? Pour commencer, nous devons représenter les connexions dans notre réseau comme une matrice d'adjacence (`adjacency_matrix`), dont l'entrée (i, j) est soit 1 (si les utilisateurs i et j sont amis), soit 0 (s'ils ne le sont pas) :

```
def entry_fn(i: int, j: int):
    return 1 if (i, j) in friend_pairs or (j, i) in friend_pairs else 0

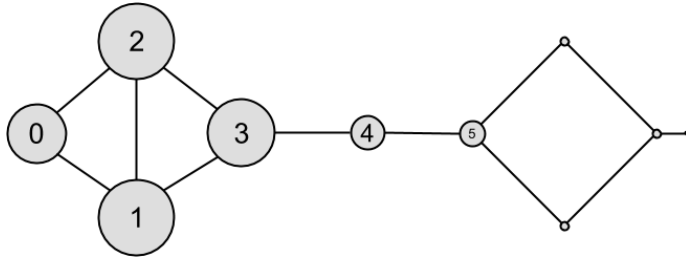
n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

La centralité de vecteur propre pour chaque utilisateur est donc l'entrée correspondant à l'utilisateur dans le vecteur retourné par `find_eigenvector` (figure 22–4).

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

Figure 22–4

Le réseau DataSciencester dimensionné par centralité de vecteur propre

**Remarque**

Pour des raisons techniques qui dépassent largement le cadre de cet ouvrage, toute matrice d'adjacence non nulle présente nécessairement un vecteur propre, dont toutes les valeurs sont non négatives. Et, heureusement pour nous, pour cette `adjacency_matrix`, notre fonction `find_eigenvector` le trouve.

Les utilisateurs avec une centralité de vecteur propre élevée devraient être ceux qui affichent beaucoup de connexions et des connexions avec des personnes qui ont elles-mêmes une centralité élevée.

Ici, les utilisateurs 1 et 2 sont les plus centraux, car ils ont tous les deux trois connexions à des personnes qui, elles-mêmes, affichent une forte centralité. Dès qu'on s'éloigne d'eux, les centralités décroissent fortement.

Avec un réseau si petit, la centralité de vecteur propre fait preuve d'un comportement assez erratique. Si vous essayez d'ajouter ou de supprimer des liens, vous verrez que de petits changements dans le réseau affectent fortement les valeurs de centralité. Avec un réseau de plus grande taille, les choses seraient différentes.

Nous n'avons toujours pas justifié le fait qu'un vecteur propre puisse conduire à une notion raisonnable de la centralité. Être un vecteur propre signifie que si vous calculez :

```
matrix_times_vector(adjacency_matrix, eigenvector_centralities)
```

Le résultat est un multiple de `eigenvector_centralities`.

Si vous observez le fonctionnement de la multiplication de matrices, vous verrez que `matrix_times_vector` produit un vecteur dont l'élément de rang `i` est :

```
dot(adjacency_matrix[i], eigenvector_centralities)
```

C'est précisément la somme des centralités de vecteur propre des utilisateurs connectés à l'utilisateur `i`.

En d'autres termes, les centralités de vecteur propre sont des nombres, un par utilisateur, tels que chaque valeur associée à un utilisateur est un multiple constant de la somme des valeurs de ses voisins. Dans ce cas, la centralité signifie qu'un utilisateur est connecté à des personnes qui sont, elles-mêmes, centrales. Plus vous êtes connecté directement avec des personnes de

centralité élevée, plus vous êtes, vous-même, central. C'est évidemment une définition circulaire : les vecteurs propres constituent le moyen de rompre cette circularité.

Une autre interprétation consiste à réfléchir à la manière dont `find_eigenvector` fonctionne ici. La fonction commence par affecter une centralité aléatoire à chaque nœud. Ensuite, elle répète les deux étapes suivantes jusqu'à la convergence du processus.

- 1 Donner à chaque nœud un nouveau score de centralité égal à la somme des (anciens) scores de ses voisins.
- 2 Changer d'échelle pour que le vecteur de centralités soit de taille 1.

Bien que les principes mathématiques qui se cachent derrière tout ceci puissent sembler opaques de prime abord, le calcul lui-même est assez aisé (à la différence de la centralité internœud, par exemple) et il est assez facile à appliquer, même à des graphes de très grande taille (tout du moins si vous utilisez une vraie bibliothèque d'algèbre linéaire).

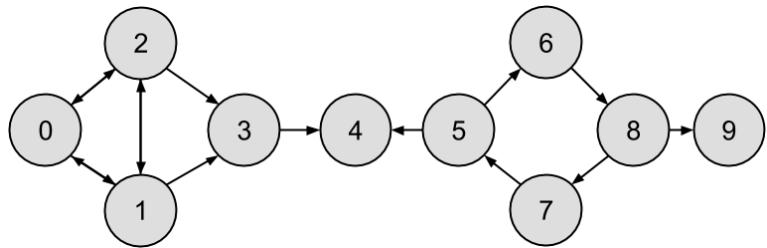
Graphes orientés et PageRank

DataSciencester n'attire pas beaucoup, de sorte que la responsable Recettes envisage de passer d'un modèle d'amis à un modèle de recommandations. Il s'avère que personne ne s'intéresse particulièrement aux amis des experts en data science, alors que les chasseurs de têtes, pour leur part, aimeraient beaucoup savoir quels sont les data scientists respectés par leurs pairs.

Dans ce nouveau modèle, nous suivrons les recommandations (source, cible) qui ne représentent plus une relation de réciprocité, mais le fait que la source recommande la cible comme un scientifique de valeur (figure 22-5).

Figure 22-5

Le réseau DataSciencester de recommandations



Nous devons tenir compte de cette asymétrie :

```
endorsements = [(0, 1), (1, 0), (0, 2), (2, 0), (1, 2),
                 (2, 1), (1, 3), (2, 3), (3, 4), (5, 4),
                 (5, 6), (7, 5), (6, 8), (8, 7), (8, 9)]
```

Ensuite, il est facile de trouver les data scientists les plus recommandés (`most_endorsed`) et vendre cette information aux chasseurs de têtes :

```
from collections import Counter
endorsement_counts = Counter(target for source, target in endorsements)
```


Cependant, le nombre de recommandations est un indicateur assez facile à truquer. Il suffit simplement de créer des comptes fictifs qui vous recommanderont ou de vous mettre d'accord avec vos amis pour vous recommander mutuellement (comme les utilisateurs 0, 1 et 2 semblent l'avoir fait).

Il existe un meilleur indicateur : prendre en compte l'identité de celui qui vous a recommandé. Les recommandations émanant de personnes qui ont elles-mêmes beaucoup de recommandations devraient compter davantage, d'une manière ou d'une autre, que celles émanant de personnes peu recommandées. C'est le cœur de l'algorithme de PageRank utilisé par Google pour classer les sites web à partir d'autres sites qui pointent sur eux et ainsi de suite... (si tout ceci vous fait penser à la centralité de vecteur propre, alors vous êtes sur la bonne voie).

En voici une version simplifiée.

- 1 Il y a un total de 1.0 (ou 100 %) de PageRank dans le réseau.
- 2 Initialement, ce PageRank est réparti équitablement entre tous les nœuds.
- 3 À chaque étape, une large fraction de PageRank de chaque nœud est répartie uniformément entre les liens sortants.
- 4 À chaque étape, le PageRank restant de chaque nœud est réparti uniformément entre tous les nœuds.

```
import tqdm
def page_rank(users: List[User],
              endorsements: List[Tuple[int, int]],
              damping: float = 0,85,
              num_iters: int = 100) -> Dict[int, float]:
    # calculer le nombre de personnes que chaque personne recommande
    outgoing_counts = Counter(target for source, target in endorsements)
    # initialement, distribuer uniformément le PageRank
    num_users = len(users)
    pr = {user.id : 1 / num_users for user in users}
    # petite fraction du PageRank que reçoit chaque nœud à chaque itération
    base_pr = (1 - damping) / num_users
    for iter in tqdm.trange(num_iters):
        # débiter avec pr_base
        next_pr = {user.id : base_pr for user in users} # débiter avec base_pr
        for source, target in endorsements:
            # ajouter une fraction réduite du pr source à la cible
            next_pr[target] += damping * pr[source] / outgoing_counts[source]
        pr = next_pr
    return pr
```

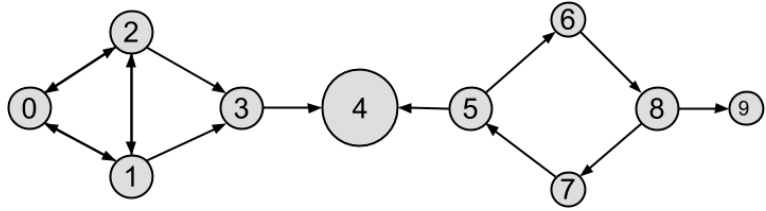
Calculons le classement des pages :

```
pr = page_rank(users, endorsements)
# le classement de page de Thor (user_id 4) est le plus élevé
assert pr[4] > max(page_rank
                  for user_id, page_rank in pr.items()
                  if user_id != 4)
```

Le PageRank (figure 22-6) identifie l'utilisateur 4 (Thor) comme l'expert de rang le plus élevé.

Figure 22-6

Le réseau DataSciencester dimensionné par PageRank



Bien qu'il ait moins de recommandations (2) que les utilisateurs 0, 1 et 2, ses recommandations véhiculent leurs rangs propres. De plus, les deux recommandations ne concernent que lui, ce qui veut dire qu'elles n'ont pas besoin de partager leur rang avec d'autres utilisateurs.

Pour aller plus loin

- Il existe beaucoup d'autres notions de centralité¹ en plus de celles que nous avons utilisées (bien que nous ayons abordé ici les plus populaires d'entre elles).
- `NetworkX`² est une bibliothèque Python dédiée à l'analyse de réseau. Elle contient des fonctions de calcul de centralité et de représentation de graphes.
- `Gephi`³ est un outil de représentation de réseaux basé sur une interface graphique utilisateur. On l'adore ou on le déteste.

1. <https://fr.wikipedia.org/wiki/Centralité>

2. <http://networkx.github.io/>

3. <https://gephi.org/>

Systèmes de recommandation

Ô nature, nature, pourquoi es-tu si peu honnête, toi qui envoies dans le monde des hommes avec ces fausses recommandations ?

Henry Fielding

Produire des recommandations est un autre problème classique qu'on rencontre en gestion des données. Netflix vous recommande des films que vous pourriez avoir envie de voir. Amazon vous recommande des produits que vous pourriez avoir envie d'acheter. Twitter vous recommande des utilisateurs que vous pourriez avoir envie de suivre. Dans ce chapitre, nous allons explorer diverses méthodes de recommandation à partir des données.

En particulier, nous reviendrons sur les données concernant les centres d'intérêt de nos utilisateurs.

Nous réfléchirons aussi à la possibilité de recommander de nouveaux sujets à un utilisateur à partir de ses centres d'intérêt actuels.

```
users_interests = [  
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],  
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],  
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],  
    ["R", "Python", "statistics", "regression", "probability"],  
    ["machine learning", "regression", "decision trees", "libsvm"],  
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],  
    ["statistics", "probability", "mathematics", "theory"],  
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],  
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],  
    ["Hadoop", "Java", "MapReduce", "Big Data"],  
    ["statistics", "R", "statsmodels"],
```

```
["C++", "deep learning", "artificial intelligence", "probability"],
["pandas", "R", "python"],
["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
["libsvm", "regression", "support vector machines"]]
```

Méthode manuelle

Avant Internet, pour chercher des recommandations de lecture, vous alliez à la bibliothèque, où on vous suggérait un livre correspondant à vos goûts ou ressemblant aux livres que vous aimiez.

Étant donné le nombre limité de centres d'intérêt des utilisateurs de DataSciencester, il devrait vous être facile, en une après-midi, de fournir des recommandations personnalisées à chacun d'entre eux. Toutefois, cette méthode n'est plus gérable dès lors que le nombre d'utilisateurs augmente. En outre, elle est limitée par vos connaissances personnelles et votre imagination (attention, je n'ai pas dit que vous aviez des connaissances et une imagination bornées). Réfléchissons plutôt au moyen d'exploiter des données.

Recommander ce qui est populaire

Une méthode simple consiste à recommander ce qui a du succès :

```
from collections import Counter
popular_interests = Counter(interest
                             for user_interests in users_interests
                             for interest in user_interests)

[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 #...
]
```

Une fois ce calcul effectué, nous pouvons suggérer à un utilisateur les sujets les plus populaires qui ne figurent pas encore parmi ses centres d'intérêt :

```
from typing import List, Tuple
def most_popular_new_interests(
    user_interests: List[str],
    max_results: int = 5) -> List[Tuple[str, int]]:
    suggestions = [(interest, frequency)
                    for interest, frequency in popular_interests.most_common()
                    if interest not in user_interests]
    return suggestions[:max_results]
```



```
assert unique_interests[:6] == ['Big Data', 'C++', 'Cassandra',
                                'HBase', 'Hadoop', 'Haskell',
                                # ...
                                ]
```

Puis, nous produirons un « vecteur des sujets » en bouclant sur la liste `unique_interests` et en mettant 1 si l'utilisateur a déclaré ce centre d'intérêt, sinon 0 :

```
def make_user_interest_vector(user_interests: List[str]) -> List[int]:
    """Soit une liste de centres d'intérêt, produire un vecteur dont le
       i-ème élément est 1 si unique_interests[i] est dans la liste, 0 sinon."""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

Ensuite, nous pouvons construire une liste des centres d'intérêt d'utilisateur :

```
user_interest_vectors = [make_user_interest_vector(user_interests)
                        for user_interests in users_interests]
```

À présent, `user_interest_vectors[i][j]` vaut 1 si l'utilisateur *i* a déclaré un intérêt *j*, sinon 0.

Comme notre jeu de données est réduit, il n'y a aucun problème pour calculer les similarités par paires d'utilisateurs :

```
from scratch.nlp import cosine_similarity
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_vectors]
                     for interest_vector_i in user_interest_vectors]
```

Après ce calcul, `user_similarities[i][j]` nous donne la similarité entre les utilisateurs *i* et *j*.

```
# les utilisateurs 0 et 9 aiment Hadoop, Java et les Big Data
assert 0.56 < user_similarities[0][9] < 0.58, "plusieurs intérêts partagés"
# les utilisateurs 0 et 8 partagent un seul intérêt : les Big Data
assert 0.18 < user_similarities[0][8] < 0.20, "un seul intérêt partagé"
```

En particulier, `user_similarities[i]` est le vecteur des similarités de l'utilisateur *i* avec chaque autre utilisateur. Cela nous permet d'écrire une fonction de recherche des utilisateurs qui ressemblent le plus à un utilisateur donné. Nous nous assurons de n'inclure ni l'utilisateur proprement dit, ni aucun utilisateur ayant une similarité nulle. Nous trions ensuite les résultats par ordre décroissant :

```
def most_similar_users_to(user_id: int) -> List[Tuple[int, float]]:
    pairs = [(other_user_id, similarity)
              for other_user_id, similarity in
              enumerate(user_similarities[user_id])
              if user_id != other_user_id and similarity > 0] # non nulle.
    # Trouver d'autres utilisateurs présentant une similarité
```

```

return sorted(pairs,                # Les trier
               key=lambda pair: pair[-1], # en commençant par
               reverse=True)          # les plus similaires.

```

Par exemple, si nous appelons `most_similar_users_to(0)`, nous obtenons le résultat suivant :

```

[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]

```

Comment utiliser ces résultats pour suggérer de nouveaux centres d'intérêt à un utilisateur ? Pour chaque sujet, nous pouvons ajouter les similarités des autres utilisateurs intéressés par le centre d'intérêt concerné :

```

from collections import defaultdict
def user_based_suggestions(user_id: int,
                           include_current_interests: bool = False):
    # faire la somme des similarités
    suggestions: Dict[str, float] = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity
    # les convertir en une liste triée
    suggestions = sorted(suggestions.items(),
                        key=lambda pair: pair[-1], # poids
                        reverse=True)
    # et exclure (éventuellement) les centres d'intérêt déjà existants
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]

```

Si nous appelons `user_based_suggestions(0)`, les premiers centres d'intérêt suggérés sont les suivants :

```

[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 #...
]

```

Ces suggestions semblent plutôt raisonnables pour une personne qui déclare être intéressée par les Big Data et les bases de données associées. Les poids n'ont pas de sens intrinsèque ; ils servent uniquement pour trier les sujets.

Cette méthode ne fonctionne pas bien si le nombre de sujets possibles est très élevé. Ainsi, souvenez-vous de la malédiction de la dimension évoquée au chapitre 12 : dans les espaces vectoriels qui ont de nombreuses dimensions, les vecteurs sont pour la plupart très éloignés les uns des autres (et pointent également dans des directions très différentes). En clair, s'il y a un grand nombre de centres d'intérêt possibles, les « utilisateurs les plus similaires » à un utilisateur donné pourront très bien ne pas être similaires du tout.

Imaginons un site comme *Amazon.com*, sur lequel j'ai acheté des milliers d'articles ces vingt dernières années. Vous pouvez essayer d'identifier des utilisateurs qui me ressemblent à partir de leurs typologies d'achat, mais il est vraisemblable qu'il n'existe personne dans le monde dont l'historique d'achats ressemble, même vaguement, au mien. Qui que soit mon « acheteur similaire », il ne me ressemble sans doute pas du tout et ses achats feraient certainement de bien pauvres recommandations.

Filtrage collaboratif sur la base des articles

Une autre méthode de recommandation consiste à calculer les similarités directement à partir des centres d'intérêt. Nous pouvons générer des suggestions pour chaque utilisateur en rassemblant les intérêts similaires aux siens.

Pour commencer, transposons notre matrice utilisateur-sujet pour que les lignes correspondent aux centres d'intérêt et les colonnes aux utilisateurs :

```
interest_user_matrix = [[user_interest_vector[j]
                        for user_interest_vector in user_interest_vectors]
                        for j, _ in enumerate(unique_interests)]
```

La ligne *j* de la matrice *interest_user_matrix* est la colonne *j* de *user_interest_matrix*. Donc, elle contient 1 pour chaque utilisateur déclarant ce centre d'intérêt et 0 pour chaque utilisateur qui l'ignore complètement. Par exemple, *interest_user_matrix[0]* (Big Data) vaut :

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

Nous pouvons utiliser encore une fois la similarité cosinus. Si exactement les mêmes utilisateurs s'intéressent à deux sujets, alors leur similarité sera à 1. Si aucun des utilisateurs ne s'intéresse aux deux sujets, alors leur similarité sera nulle (0) :

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                        for user_vector_j in interest_user_matrix]
                        for user_vector_i in interest_user_matrix]
```


Par exemple, nous découvrirons les sujets les plus similaires à Big Data (sujet 0) avec le code suivant :

```
def most_similar_interests_to(interest_id: int):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
              for other_interest_id, similarity in enumerate(similarities)
              if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
                  key=lambda pair: pair[-1],
                  reverse=True)
```

Ce code suggère les intérêts similaires suivants :

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

Désormais, nous pouvons créer des recommandations pour un utilisateur en faisant la somme des similarités des centres d'intérêt similaires aux siens :

```
def item_based_suggestions(user_id: int,
                           include_current_interests: bool = False):
    # additionner les centres d'intérêt similaires
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_vectors[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity
    # les trier par poids
    suggestions = sorted(suggestions.items(),
                        key=lambda pair: pair[-1],
                        reverse=True)
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

Pour l'utilisateur 0, le code génère les recommandations suivantes, qui semblent raisonnables :

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('C++', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('Python', 0.2886751345948129),
 ('R', 0.2886751345948129)]
```

Factorisation de matrice

Comme nous l'avons vu, nous pouvons représenter les préférences de nos utilisateurs sous la forme d'une matrice `[num_users, num_items]` constituée de 1 (éléments qui sont appréciés) et de 0 (éléments qui ne le sont pas).

Dans certains cas, vous obtenez en fait des notations numériques ; par exemple, lorsque vous rédigez un avis sur Amazon, vous attribuez un score à l'article concerné allant de 1 à 5 étoiles. Il est possible de représenter ces scores par des valeurs numériques dans une matrice `[num_users, num_items]` (en mettant de côté, pour le moment, le traitement des articles non notés).

Dans cette section, nous allons considérer que nous disposons de données de notation de ce type et que nous essayons d'apprendre un modèle capable de prédire la note d'un utilisateur pour un article donné.

Une approche du problème consiste à considérer que chaque utilisateur présente un « type » latent – représenté sous la forme d'un vecteur de valeurs numériques – et qu'il en va de même pour chaque article.

Si les types d'utilisateur sont représentés sous la forme d'une matrice `[num_users, dim]` et si la transposition des types d'article est représentée sous la forme d'une matrice `[dim, num_items]`, leur produit est une matrice `[num_users, num_items]`. De même, une manière de construire un tel modèle consiste à « factoriser » la matrice de préférences en un produit d'une matrice d'utilisateurs et d'une matrice d'articles.

Remarque

Cette idée de types latents vous rappellera peut-être les incorporations de mots que nous avons élaborées au chapitre 21. Gardez cette idée à l'esprit.

Plutôt que de travailler avec le jeu de données de 10 utilisateurs que nous avons constitué, nous allons exploiter le jeu de données MovieLens 100k. Il contient les notes de 0 à 5 attribuées par de nombreux utilisateurs à de nombreux films. Chaque utilisateur a noté uniquement un petit sous-ensemble des films. Nous allons utiliser cette configuration pour tenter de construire un système capable de prédire la note de n'importe quelle paire (utilisateur, film). Nous allons entraîner ce système à effectuer des prédictions efficaces sur les films que chaque utilisateur a notés, en espérant alors qu'il saura « généraliser » son approche à des films qu'un utilisateur n'a pas notés.

Pour commencer, récupérons le jeu de données. Vous pouvez le télécharger depuis <http://files.grouplens.org/datasets/movielens/ml-100k.zip>.

Décompressez-le et extrayez les fichiers ; nous n'en utiliserons que deux :

```
# ceci pointe vers le répertoire courant
# le modifier si vos fichiers se trouvent à un autre emplacement
MOVIES = "u.item" # id_movie_id|title|...
RATINGS = "u.data" #user_id, movie_id, rating, timestamp
```

Comme c'est souvent le cas, nous allons introduire un `NamedTuple` pour faciliter le travail :

```
from typing import NamedTuple
class Rating(NamedTuple):
    user_id: str
    movie_id: str
    rating: float
```

Remarque

L'ID de film et celui de l'utilisateur sont en fait des entiers. Toutefois, ils ne sont pas consécutifs. En d'autres termes, si nous les avons exploités comme des entiers, nous aurions obtenu un grand nombre de dimensions gâchées (à moins de tout renuméroter). Aussi, par souci de simplicité, nous allons les traiter comme des chaînes.

À présent, procédons à la lecture et à l'exploration des données. Le fichier de films utilise une délimitation par le caractère pipe | et présente de nombreuses colonnes. Nous ne nous intéresserons qu'aux deux premières : ID et titre :

```
import csv
# codage évitant de lever une erreur UnicodeDecodeError
# consulter : https://stackoverflow.com/a/53136168/1076346
with open(MOVIES, encoding="iso-8859-1") as f:
    reader = csv.reader(f, delimiter="|")
    movies = {movie_id: title for movie_id, title, *_ in reader}
```

Le fichier des notes utilise une délimitation par retraits et contient quatre colonnes : `user_id` (ID utilisateur), `movie_id` (ID film), `rating` (note de 1 à 5) et `timestamp` (horodatage). Nous n'aurons pas besoin de ce dernier :

```
# créer une liste de notes
with open(RATINGS, encoding="iso-8859-1") as f:
    reader = csv.reader(f, delimiter="\t")
    ratings = [Rating(user_id, movie_id, float(rating))
               for user_id, movie_id, rating, _ in reader]

# 1 682 films notés par 943 utilisateurs
assert len(movies) == 1682
assert len(list({rating.user_id for rating in ratings})) == 943
```

Ces données autorisent un grand nombre d'analyses exploratoires intéressantes. Par exemple, vous pouvez vous intéresser aux notes moyennes des films de la saga *Star Wars* (le jeu de données date de 1998 ; il précède *La Menace fantôme* d'un an) :

```
import re
# structure de données pour accumuler des notes par movie_id
star_wars_ratings = {movie_id: []
                     for movie_id, title in movies.items()
                     if re.search("Star Wars|Empire Strikes|Jedi",
                                   title)}

# itérer sur les notations, en accumulant celles de Star Wars
for rating in ratings:
    if rating.movie_id in star_wars_ratings:
        star_wars_ratings[rating.movie_id].append(rating.rating)
# calculer la note moyenne de chaque film
avg_ratings = [(sum(title_ratings) / len(title_ratings), movie_id)
               for movie_id, title_ratings in star_wars_ratings.items()]
# puis les afficher dans l'ordre
for avg_rating, movie_id in sorted(avg_ratings, reverse=True):
    print(f"{avg_rating:.2f} {movies[movie_id]}")
```

Ils affichent tous des notes assez élevées :

```
4.36 Star Wars (1977)
4.20 Empire Strikes Back, The (1980)
4.01 Return of the Jedi (1983)
```

À présent, essayons d'obtenir un modèle capable de prédire ces scores. Commençons par diviser les notes en ensembles d'apprentissage, de validation et de test :

```
import random
random.seed(0)
random.shuffle(ratings)
split1 = int(len(ratings) * 0.7)
```

```
split2 = int(len(ratings) * 0.85)
train = ratings[:split1] # 70 % des données
validation = ratings[split1:split2] # 15 % des données
test = ratings[split2:] # 15 % des données
```

Il est toujours bénéfique de disposer d'un modèle de base simple et de s'assurer que le nôtre est plus performant. Voici un modèle de base simple susceptible de « prédire la note moyenne ». Nous utiliserons pour élément de mesure l'erreur au carré. Voyons comment cette base se comporte sur notre ensemble de test :

```
avg_rating = sum(rating.rating for rating in train) / len(train)
baseline_error = sum((rating.rating - avg_rating) ** 2
                     for rating in test) / len(test)

# voici ce que nous espérons pouvoir surclasser
assert 1.26 < baseline_error < 1.27
```

Étant donné nos incorporations, les notes prédites sont calculées comme le produit des matrices des incorporations utilisateur et des incorporations film. Pour un utilisateur et un film donnés, cette valeur est simplement le produit scalaire (*dot*) des incorporations correspondantes.

Aussi, commençons par créer les incorporations. Nous les représenterons sous forme d'éléments *dict*, dans lesquels les clés sont des ID et les valeurs sont des vecteurs. Nous pourrions ainsi facilement extraire l'incorporation pour un ID donné :

```
from scratch.deep_learning import random_tensor
EMBEDDING_DIM = 2
# rechercher des ID uniques
user_ids = {rating.user_id for rating in ratings}
movie_ids = {rating.movie_id for rating in ratings}
# créer ensuite un vecteur aléatoire par ID
user_vectors = {user_id: random_tensor(EMBEDDING_DIM)
                for user_id in user_ids}
movie_vectors = {movie_id: random_tensor(EMBEDDING_DIM)
                 for movie_id in movie_ids}
```

À ce stade, nous devrions être plutôt experts dans l'élaboration de boucles d'entraînement :

```
from typing import List
import tqdm
from scratch.linear_algebra import dot
def loop(dataset: List[Rating],
         learning_rate: float = None) -> None:
    with tqdm.tqdm(dataset) as t:
        loss = 0.0
        for i, rating in enumerate(t):
            movie_vector = movie_vectors[rating.movie_id]
            user_vector = user_vectors[rating.user_id]
```

```

predicted = dot(user_vector, movie_vector)
error = predicted - rating.rating
loss += error ** 2
if learning_rate is not None:
    # prédit = m_0 * u_0 + ... + m_k * u_k
    # Chaque u_j fait entrer une sortie de coefficient m_j
    # et chaque m_j fait entrer une sortie de coefficient u_j.
    user_gradient = [error * m_j for m_j in movie_vector]
    movie_gradient = [error * u_j for u_j in user_vector]
    # accepter des pas de gradient
    for j in range(EMBEDDING_DIM):
        user_vector[j] -= learning_rate * user_gradient[j]
        movie_vector[j] -= learning_rate * movie_gradient[j]
t.set_description(f"avg loss: {loss / (i + 1)}")

```

Nous pouvons à présent entraîner notre modèle (à savoir, trouver les incorporations optimales). Dans mon cas, il a fonctionné au mieux si je diminuais un peu le taux d'apprentissage pour chaque étape (epoch) :

```

learning_rate = 0.05
for epoch in range(20):
    learning_rate *= 0.9
    print(epoch, learning_rate)
    loop(train, learning_rate=learning_rate)
    loop(validation)
    loop(test)

```

Ce modèle est plutôt apte à surajuster l'ensemble d'entraînement. J'ai obtenu les meilleurs résultats avec `EMBEDDING_DIM=2`, qui m'a décroché une perte moyenne sur l'ensemble de test d'environ 0,89.

Remarque

Si vous souhaitez des incorporations plus fortement dimensionnées, vous pouvez tenter une régularisation (chapitre 15). Plus particulièrement, à chaque actualisation de gradient, vous pourriez rétrécir les poids en tendant vers 0. Je ne suis pas parvenu à obtenir de meilleurs résultats de cette manière.

À présent, inspectons les vecteurs appris. Il n'existe aucune raison de s'attendre à ce que les deux composantes soient particulièrement pertinentes ; aussi utiliserons-nous l'analyse des composantes principales :

```

from scratch.working_with_data import pca, transform
original_vectors = [vector for vector in movie_vectors.values()]
components = pca(original_vectors, 2)

```

Transformons nos vecteurs pour représenter les composantes principales et joindre les ID de film et les notes moyennes :

```
ratings_by_movie = defaultdict(list)
for rating in ratings:
    ratings_by_movie[rating.movie_id].append(rating.rating)
vectors = [(movie_id,
            sum(ratings_by_movie[movie_id]) / len(ratings_by_movie[movie_id]),
            movies[movie_id],
            vector)
            for movie_id, vector in zip(movie_vectors.keys(),
                                       transform(original_vectors, components))]
```

```
# Afficher les 25 meilleurs et les 25 plus mauvais
# par première composante principale.
print(sorted(vectors, key=lambda v: v[-1][0])[:25])
print(sorted(vectors, key=lambda v: v[-1][0])[-25:])
```

Les 25 meilleurs sont tous très bien notés, tandis que les 25 plus mauvais obtiennent essentiellement des notes faibles (ou ne sont pas notés dans les données d'entraînement), ce qui laisse penser que la première composante principale capture essentiellement la réponse à la question : « Quelle est la qualité de ce film ? »

Il m'est difficile de tirer un sens de la deuxième composante ; et, en effet, les incorporations à deux dimensions se sont comportées à peine mieux que celles à une dimension, laissant ainsi entendre que la deuxième composante capture quelque chose de vraisemblablement très subtil. Il est probable qu'un des jeux de données les plus volumineux de MovieLens recèlera bien davantage de choses.

Pour aller plus loin

- `surprise`¹ est une bibliothèque Python pour « construire et analyser des systèmes de recommandation ». Elle semble relativement populaire et actualisée.
- Le Netflix Prize² était un concours assez célèbre destiné à construire un meilleur système de recommandations de films pour les abonnés à Netflix.

1. <http://surpriselib.com/>

2. <https://www.netflixprize.com/>

Bases de données et SQL

La mémoire est le meilleur ami de l'homme et son pire ennemi.

Gilbert Parker

Les données dont vous avez besoin sont souvent stockées dans des bases de données. Ces dernières sont des systèmes conçus pour stocker et rechercher les données efficacement. Elles sont en majorité de type relationnel (SGBDR) comme PostgreSQL, MySQL et SQL Server, qui stockent les données dans des tables et que l'on interroge à l'aide de requêtes SQL (*Structured Query Language*), un langage déclaratif de manipulation de données.

SQL est un élément essentiel de la boîte à outils du data scientist. Dans ce chapitre, nous allons créer `NotQuiteABase`, une mise en œuvre Python de quelque chose qui n'est pas tout à fait une base de données. Nous aborderons aussi des fondamentaux de SQL en présentant leur application à notre « presque-base-de-données », ce qui vous aidera à comprendre leur fonctionnement. Il s'agit de la méthode « de A à Z » la plus poussée que j'ai pu élaborer pour vous aider à comprendre cette mécanique. J'espère sincèrement que la résolution des problèmes dans notre « presque-base-de-données » vous montrera comment les appréhender avec SQL.

CREATE TABLE et INSERT

Une base de données relationnelle est une collection de tables (et de relations entre celles-ci). Et une table est simplement une collection de lignes, un peu à la manière des matrices. Cependant, une table est associée à un schéma fixe qui donne les noms des colonnes et leurs types.

Prenons, par exemple, un jeu de données `users` contenant, pour chaque utilisateur, son identifiant, son nom et le nombre de ses amis (soit respectivement `user_id`, `name` et `num_friends`) :

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

En SQL, voici comment nous allons créer cette table :

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

Vous noterez que nous avons spécifié que l'identifiant et le nombre d'amis doivent être des entiers, que `user_id` ne peut pas afficher de valeur `NULL` (qui indique une valeur absente, un peu comme notre `None`) et que le nom doit être une chaîne de caractères de longueur 200 ou moins. Nous utiliserons des types Python de manière similaire.

Remarque

SQL est à peu près totalement indifférent à la casse et à l'indentation. Le choix des majuscules et l'indentation des exemples relèvent de mes préférences stylistiques. Si vous commencez à apprendre SQL, vous verrez d'autres exemples utilisant d'autres conventions.

Pour insérer des lignes, vous pouvez utiliser l'instruction `INSERT` :

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

Remarquez que les instructions SQL se terminent par un point-virgule et que les chaînes de caractères sont encadrées par des guillemets simples.

Dans `NotQuiteABase`, vous créez une table (classe `Table`) en spécifiant un schéma similaire. Pour insérer une ligne dans la table, vous utiliserez ensuite la méthode `insert`. Cette instruction prend en entrée une liste de valeurs de ligne qui doivent être classées dans le même ordre que les noms des colonnes de la table.

Techniquement parlant, chaque ligne est stockée dans un `dict`, associant des valeurs aux noms des colonnes. Une vraie base de données n'utiliserait jamais une représentation qui gaspille autant d'espace, mais ce choix rend `NotQuiteABase` beaucoup plus facile à utiliser.

Nous allons mettre en œuvre une table sous la forme d'une classe géante, que nous élaborerons en prenant une méthode à la fois. Commençons par préciser certains imports et alias de type :

```
from typing import Tuple, Sequence, List, Any, Callable, Dict, Iterator
from collections import defaultdict
```

```
# quelques alias que nous utiliserons plus tard
Row = Dict[str, Any] # une ligne de base de données
whereClause = Callable[[Row], bool] # prédicat pour une ligne unique
havingClause = Callable[[List[Row]], bool] # prédicat sur plusieurs lignes
```

Commençons par le constructeur. Pour créer une table `NotQuiteABase`, nous devons passer une liste de noms de colonne et une liste de types de colonne, tout comme nous le ferions pour créer une table dans une base de données SQL :

```
class Table:
    def __init__(self, columns: List[str], types: List[type]) -> None:
        assert len(columns) == len(types), "# de colonnes doit == # de types"
        self.columns = columns # noms de colonnes
        self.types = types # types de données de colonnes
        self.rows: List[Row] = [] # aucune donnée pour l'instant
```

Nous allons ajouter une méthode d'aide pour obtenir le type d'une colonne :

```
def col2type(self, col: str) -> type:
    idx = self.columns.index(col) # Rechercher l'index de la colonne
    return self.types[idx] # et retourner son type.
```

Et nous ajoutons une méthode `insert` qui vérifie que les valeurs que nous insérons sont valides. Vous devez, en particulier, fournir le nombre approprié de valeurs et chacune d'elles doit être du type adéquat (ou `None`) :

```
def insert(self, values: list) -> None:
    # contrôler le nombre de valeurs
    if len(values) != len(self.types):
        raise ValueError(f"Vous devez fournir {len(self.types)} values")
    # contrôler les types de valeurs
    for value, typ3 in zip(values, self.types):
        if not isinstance(value, typ3) and value is not None:
            raise TypeError(f"Type attendu {typ3} but got {value}")
    # ajouter le dict correspondant sous forme de ligne
    self.rows.append(dict(zip(self.columns, values)))
```

Dans une vraie base de données SQL, vous spécifieriez explicitement si une colonne donnée, quelle qu'elle soit, est ou non autorisée à ne pas contenir de valeurs (`None`). Pour nous simplifier la vie, nous partirons du principe que toutes les colonnes le peuvent.

Nous allons également définir quelques méthodes *dunder* pour traiter une table sous la forme d'une `List[Row]` que nous utiliserons essentiellement pour tester notre code :

```
def __getitem__(self, idx: int) -> Row:
    return self.rows[idx]
```

```
def __iter__(self) -> Iterator[Row]:
    return iter(self.rows)

def __len__(self) -> int:
    return len(self.rows)
```

Enfin, nous allons spécifier une méthode pour obtenir un affichage élégant de notre table :

```
def __repr__(self):
    """affichage élégant de la table : colonnes, puis lignes"""
    rows = "\n".join(str(row) for row in self.rows)
    return f"{self.columns}\n{rows}"
```

Nous pouvons à présent créer notre table `users` :

```
# le constructeur a besoin de noms et de types de colonne
users = Table(['user_id', 'name', 'num_friends'], [int, str, int])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

Si vous lancez à présent `print(users)`, vous obtiendrez ce qui suit :

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

L'API de type liste facilite l'écriture des tests :

```
assert len(users) == 11
assert users[1]['name'] == 'Dunn'
```

Nous avons bien d'autres fonctionnalités à ajouter.

UPDATE

Il peut vous arriver de devoir mettre à jour les données qui sont déjà dans une table. Par exemple, ajoutons un nouvel ami de Dunn (en SQL) :

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

Les points importants à prendre en compte pour cette opération sont les suivants :

- Quelle est la table à mettre à jour ?
- Quelles sont les lignes à actualiser ?
- Quels sont les champs concernés ?
- Quelles sont les nouvelles valeurs ?

Nous allons ajouter une méthode similaire de mise à jour (`update`) à `NotQuiteABase`. Son premier argument sera un `dict` dont les clés sont les colonnes à mettre à jour et dont les valeurs sont les nouvelles valeurs pour ces champs. Son deuxième argument (facultatif) sera un prédicat qui retourne `True` pour les lignes à mettre à jour et `False` sinon :

```
def update(self,
              updates: Dict[str, Any],
              predicate: whereClause = lambda row: True):
    # Tout d'abord, s'assurer que les mises à jour affichent
    # des types et des noms valides.
    for column, new_value in updates.items():
        if column not in self.columns:
            raise ValueError(f"colonne invalide : {column}")
        typ3 = self.col2type(column)
        if not isinstance(new_value, typ3) and new_value is not None:
            raise TypeError(f"type attendu {typ3}, mais obtenu {new_value}")
    # à présent mettre à jour
    for row in self.rows:
        if predicate(row):
            for column, new_value in updates.items():
                row[column] = new_value
```

Ensuite, il nous suffit simplement de procéder comme suit :

```
assert users[1]['num_friends'] == 2 # valeur d'origine
users.update({'num_friends' : 3}, # définir num_friends = 3
             lambda row: row['user_id'] == 1) # si user_id == 1
assert users[1]['num_friends'] == 3 # valeur mise à jour
```

DELETE

Il existe deux manières de supprimer des lignes dans une table en SQL. La méthode dangereuse supprime toutes les lignes de la table :

```
DELETE FROM users;
```

La méthode moins dangereuse ajoute une clause `WHERE` pour supprimer uniquement les lignes qui vérifient une certaine condition :

```
DELETE FROM users WHERE user_id = 1;
```

On peut facilement ajouter cette fonctionnalité à notre classe `Table` :

```
def delete(self, predicate: whereClause = lambda row: True) -> None:
    """supprimer toutes les lignes correspondant au prédicat"""
    self.rows = [row for row in self.rows if not predicate(row)]
```

Si vous fournissez une fonction prédicat (à savoir, une clause `WHERE`), seules les lignes qui vérifient la condition exposée seront supprimées. Toutefois, si vous ne fournissez pas de prédicat, le prédicat par défaut retournera systématiquement `True` et toutes les lignes seront supprimées.

```
# nous n'exécuterons pas réellement ce code
# supprimer les lignes avec id_u == 1
users.delete(lambda row: row["user_id"] == 1)
# supprimer toutes les lignes
users.delete()
```

SELECT

En général, personne n'examine directement les tables SQL. Au lieu de cela, on les interroge avec une instruction `SELECT` :

```
SELECT * FROM users; # récupérer l'intégralité du contenu
SELECT * FROM users LIMIT 2; # les deux premières lignes
SELECT user_id FROM users; # une colonne spécifique
SELECT user_id FROM users WHERE name = 'Dunn'; # une ligne spécifique
```

Les instructions `SELECT` permettent également de calculer des champs :

```
SELECT LENGTH(name) AS name_length FROM users;
```

Nous allons fournir à notre classe `Table` une méthode `select` qui retournera une nouvelle `Table`. Cette méthode acceptera deux arguments optionnels :

- `keep_columns` spécifiera le nom des colonnes à conserver dans le résultat. Si vous ne le fournissez pas, le résultat contiendra toutes les colonnes ;
- `additional_columns` constituera un dictionnaire dont les clés sont les noms des nouvelles colonnes et dont les valeurs sont des fonctions spécifiant comment en calculer les valeurs. Les fonctions devront disposer de types de retour annotés qui serviront à déterminer les types des nouvelles colonnes.

Si vous ne fournissez aucun des deux, vous récupérerez une simple copie de la table :

```
def select(self,
            keep_columns: List[str] = None,
            additional_columns: Dict[str, Callable] = None) -> 'Table':
    if keep_columns is None: # si aucune colonne n'est spécifiée
        keep_columns = self.columns # retourner toutes les colonnes
    if additional_columns is None:
        additional_columns = {}
    # noms et types des nouvelles colonnes
    new_columns = keep_columns + list(additional_columns.keys())
    keep_types = [self.col2type(col) for col in keep_columns]
    # voici comment obtenir le type de retour d'une annotation de type
    # le code plantera si calcul ne présente pas de type de retour
    add_types = [calculation.__annotations__['return']
                 for calculation in additional_columns.values()]
    # créer une table pour les résultats
    new_table = Table(new_columns, keep_types + add_types)
    for row in self.rows:
        new_row = [row[column] for column in keep_columns]
        for column_name, calculation in additional_columns.items():
            new_row.append(calculation(row))
        new_table.insert(new_row)
    return new_table
```

Remarque

Vous vous rappelez qu'au chapitre 2, je disais que les annotations de type ne servaient en fait à rien ? Eh bien, voici un contre-exemple ! Néanmoins, voyez la procédure compliquée que nous devons mettre en œuvre pour les obtenir.

Notre instruction `select` retourne une nouvelle `Table`, là où l'instruction `SELECT SQL` se contente de produire une sorte d'ensemble de résultats transitoires (à moins que vous n'insériez explicitement les résultats dans une table). Nous allons également avoir besoin des méthodes `where` et `limit`. Elles sont assez simples l'une comme l'autre :

```
def where(self, predicate: WhereClause = lambda row: True) -> 'Table':
    """retourner uniquement les lignes qui satisfont le prédicat"""
    where_table = Table(self.columns, self.types)
```

```

    for row in self.rows:
        if predicate(row):
            values = [row[column] for column in self.columns]
            where_table.insert(values)
    return where_table

def limit(self, num_rows: int) -> 'Table':
    """retourner uniquement les premières lignes 'num_rows'"""
    limit_table = Table(self.columns, self.types)
    for i, row in enumerate(self.rows):
        if i >= num_rows:
            break
        values = [row[column] for column in self.columns]
        limit_table.insert(values)
    return limit_table

```

Maintenant, il est facile de construire des équivalents `NotQuiteABase` des instructions `SQL` précédentes :

```

# SELECT * FROM users;
all_users = users.select()
assert len(all_users) == 11

# SELECT * FROM users LIMIT 2;
two_users = users.limit(2)
assert len(two_users) == 2

# SELECT user_id FROM users;
just_ids = users.select(keep_columns=["user_id"])
assert just_ids.columns == ['user_id']

# SELECT user_id FROM users WHERE name = 'Dunn';
dunn_ids = (users
            .where(lambda row: row["name"] == "Dunn")
            .select(keep_columns=["user_id"]))
assert len(dunn_ids) == 1
assert dunn_ids[0] == {"user_id": 1}

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row) -> int: return len(row["name"])
name_lengths = users.select(keep_columns=[],
                           additional_columns = {"name_length": name_length})
assert name_lengths[0]['name_length'] == len("Hero")

```

Notez que, pour permettre une lecture fluide des requêtes occupant plusieurs lignes, nous devons placer l'intégralité de celles-ci entre parenthèses.

GROUP BY

Autre opération souvent utilisée en SQL, `GROUP BY` sert à regrouper des lignes de valeurs identiques dans des colonnes spécifiées et à produire des valeurs d'agrégat telles que `MIN`, `MAX`, `COUNT` (nombre d'éléments) et `SUM` (somme).

Par exemple, si vous cherchez le nombre d'utilisateurs et le plus petit identifiant `user_id` pour chaque longueur de nom possible, voici comment procéder :

```
SELECT LENGTH(name) as name_length,  
       MIN(user_id) AS min_user_id,  
       COUNT(*) AS num_users  
FROM users  
GROUP BY LENGTH(name);
```

Chaque champ mentionné dans le `SELECT` doit soit figurer dans la clause `GROUP BY` (comme `name_length`), soit être agrégé d'une manière ou d'une autre (comme le sont `min_user_id` et `num_users`).

SQL prend également en charge une clause `HAVING` qui se comporte comme une clause `WHERE`, à ceci près que le filtre est appliqué aux agrégats (alors qu'un filtre `WHERE` filtre les lignes avant agrégation).

Cherchons le nombre moyen d'amis des utilisateurs dont le nom commence par une lettre donnée, tout en demandant les résultats des lettres dont la moyenne est supérieure à 1 (je sais, certains de ces exemples sont un peu tirés par les cheveux).

```
SELECT SUBSTR(name, 1, 1) AS first_letter,  
       AVG(num_friends) AS avg_num_friends  
FROM users  
GROUP BY SUBSTR(name, 1, 1)  
HAVING AVG(num_friends) > 1;
```

Remarque

Les fonctions de traitement des chaînes de caractères sont différentes selon les mises en œuvre SQL ; certaines bases de données utilisent parfois `SUBSTRING` ou autre chose.

Vous pouvez aussi calculer des agrégats globaux. Dans ce cas, il est possible de laisser tomber le `GROUP BY` :

```
SELECT SUM(user_id) as user_id_sum  
FROM users  
WHERE user_id > 1;
```

Nous allons ajouter cette fonctionnalité à notre `NotQuiteABase` via la méthode `group_by`. Elle acceptera en entrée les noms des colonnes à regrouper, un dictionnaire des fonctions d'agrégation à exécuter sur chaque groupe et un prédicat facultatif `having` qui opérera sur plusieurs lignes.

Elle réalisera les étapes suivantes.

- 1 Créer un `defaultdict` qui fait correspondre les tuples (des valeurs de `group-by`) aux lignes (contenant les valeurs de `group-by`). N'oubliez pas que vous ne pouvez pas utiliser des listes comme clés : vous devez utiliser des tuples.
- 2 Boucler sur les lignes de la table en remplissant le `defaultdict`.
- 3 Créer une nouvelle table avec les colonnes correctes en sortie.
- 4 Boucler sur le `defaultdict` et remplir la table de sortie en appliquant le filtre `having` le cas échéant.

```
def group_by(self,
              group_by_columns: List[str],
              aggregates: Dict[str, Callable],
              having: HavingClause = lambda group: True) -> 'Table':
    grouped_rows = defaultdict(list)
    # remplir les groupes
    for row in self.rows:
        key = tuple(row[column] for column in group_by_columns)
        grouped_rows[key].append(row)
    # table de résultats constituée de colonnes groupées et d'agrégats
    new_columns = group_by_columns + list(aggregates.keys())
    group_by_types = [self.col2type(col) for col in group_by_columns]
    aggregate_types = [agg.__annotations__['return']
                       for agg in aggregates.values()]
    result_table = Table(new_columns, group_by_types + aggregate_types)
    for key, rows in grouped_rows.items():
        if having(rows):
            new_row = list(key)
            for aggregate_name, aggregate_fn in aggregates.items():
                new_row.append(aggregate_fn(rows))
            result_table.insert(new_row)
    return result_table
```

Remarque

Une véritable base de données procéderait très certainement d'une manière plus efficace.

Une fois encore, voyons comment nous pourrions faire la même chose que les instructions SQL précédentes. Les indicateurs de longueur de nom (`name_length`) sont les suivants :

```
def min_user_id(rows) -> int:
    return min(row["user_id"] for row in rows)

def length(rows) -> int:
    return len(rows)
```

```
stats_by_length = (users
    .select(additional_columns={"name_length" : name_length})
    .group_by(group_by_columns=["name_length"],
        aggregates={"min_user_id" : min_user_id,
            "num_users" : length}))
```

Voici les indicateurs de première lettre :

```
def first_letter_of_name(row: Row) -> str:
    return row["name"][0] if row["name"] else ""

def average_num_friends(rows: List[Row]) -> float:
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows: List[Row]) -> bool:
    return average_num_friends(rows) > 1

avg_friends_by_letter = (users
    .select(additional_columns={'first_letter' :
        first_letter_of_name})
    .group_by(group_by_columns=['first_letter'],
        aggregates={"avg_num_friends" : average_num_friends},
        having=enough_friends))
```

La somme des `user_id` est donc la suivante :

```
def sum_user_ids(rows: List[Row]) -> int:
    return sum(row["user_id"] for row in rows)

user_id_sum = (users
    .where(lambda row: row["user_id"] > 1)
    .group_by(group_by_columns=[],
        aggregates={"user_id_sum" : sum_user_ids}))
```

ORDER BY

Le plus souvent, vous souhaitez trier les résultats de vos calculs. Par exemple, on veut connaître les noms de nos deux premiers utilisateurs (par ordre alphabétique) :

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

Pour cela, il nous suffit de doter notre table d'une méthode `order_by` qui accepte une fonction de tri :

```
def order_by(self, order: Callable[[Row], Any]) -> 'Table':
    new_table = self.select() # faire une copie
    new_table.rows.sort(key=order)
    return new_table
```

Nous pouvons l'utiliser comme suit :

```
friendliest_letters = (
    avg_friends_by_letter
    .order_by(lambda row: -row["avg_num_friends"])
    .limit(4))
```

L'instruction `ORDER BY` du langage `SQL` vous permet de spécifier un ordre `ASC` (croissant) ou `DESC` (décroissant) pour chaque champ. Ici, nous devons intégrer cet aspect dans notre fonction `ordre`.

JOIN

Les tables de bases de données relationnelles sont souvent normalisées, ce qui veut dire qu'elles sont organisées de façon à limiter les redondances. Par exemple, quand nous travaillons sur les centres d'intérêt des utilisateurs avec Python, nous pouvons donner une liste de thèmes pour chaque utilisateur.

Les tables `SQL`, elles, ne peuvent pas contenir de listes. La solution classique consiste à créer une deuxième table, `user_interests`, qui contient les relations 1 - n (1-à-plusieurs) entre les `user_id` et les centres d'intérêt. En `SQL`, cela s'écrit comme suit :

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

Voici comment cela se traduit dans `NotQuiteABase` :

```
user_interests = Table(['user_id', 'interest'], [int, str])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

Remarque

Cette méthode de stockage contient encore de nombreuses redondances ; en effet, le centre d'intérêt SQL est stocké à deux emplacements distincts. Dans une vraie base de données, on stockerait `user_id` et `interest_id` dans la table `user_interests`, puis on créerait une troisième table, `interests`, pour faire correspondre `interest_id` à `interest`, de façon à ne stocker les noms des centres d'intérêt qu'une seule fois. Ici, cela ne ferait que rendre nos exemples plus compliqués que nécessaire.

Pour analyser nos données lorsqu'elles sont disséminées dans différentes tables, il nous faut créer des jointures. Une jointure – instruction `JOIN` – combine les lignes de la table « de gauche » avec les lignes correspondantes de la table « de droite ». La façon de « faire correspondre » les données repose sur la clause de jointure.

Par exemple, pour trouver les utilisateurs intéressés par le langage SQL, votre requête sera la suivante :

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

Dans cet exemple, l'instruction `JOIN` spécifie que, pour chaque ligne de la table `users`, nous devons l'associer avec chaque ligne de `user_interests` contenant le même `user_id`.

Notez que nous devons spécifier les tables à joindre et aussi les colonnes concernées. À cette fin, une instruction `INNER JOIN` retourne les combinaisons de lignes (et seulement les combinaisons de lignes) qui correspondent suivant les critères de jointure spécifiés.

Il existe par ailleurs une instruction `LEFT JOIN` qui retourne non seulement les combinaisons de lignes correspondantes, mais aussi une ligne pour chaque ligne de la table de gauche sans correspondance dans celle de droite (dans ce cas, les champs sans correspondance à droite affichent une valeur `NULL`).

Avec un `LEFT JOIN`, on peut aisément compter le nombre de centres d'intérêt de chaque utilisateur :

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

`LEFT JOIN` garantit que tous les utilisateurs sans aucun centre d'intérêt auront encore des lignes dans l'ensemble de données jointes (avec des valeurs `NULL` pour les champs issus de `user_interests`). Et `COUNT` ne compte que les valeurs non `NULL`.

La méthode `join()` de `NotQuiteABase` sera plus restrictive : elle effectuera simplement la jointure de deux tables sur l'ensemble des colonnes qu'elles ont en commun. Même ainsi, la fonction n'est pas facile à écrire :

```
def join(self,
        other_table: 'Table',
        left_join: bool = False) -> 'Table':
    join_on_columns = [c for c in self.columns    # Colonnes dans
                       if c in other_table.columns] # les deux tables.
    join_on_columns = [c for c in self.columns    # Colonnes uniquement
                       if c not in join_on_columns] # dans la table de droite.
    # Toutes les colonnes issues de la table de gauche
    # + additional_columns de la table de droite.
    new_columns = self.columns + additional_columns
    new_types = self.types + [other_table.col2type(col)
                              for col in additional_columns]
    join_table = Table(new_columns, new_types)
    for row in self.rows:
        def is_join(other_row):
            return all(other_row[c] == row[c] for c in join_on_columns)

        other_rows = other_table.where(is_join).rows
        # garder chaque ligne correspondante
        for other_row in other_rows:
            join_table.insert([row[c] for c in self.columns] +
                             [other_row[c] for c in additional_columns])
        # aucune correspondance de ligne et LEFT JOIN => None
        if left_join and not other_rows:
            join_table.insert([row[c] for c in self.columns] +
                             [None for c in additional_columns])
    return join_table
```

Nous trouverons les utilisateurs intéressés par le SQL avec les instructions suivantes :

```
sql_users = (users
             .join(user_interests)
             .where(lambda row: row["interest"] == "SQL")
             .select(keep_columns=["name"]))
)
```

Nous pourrions également obtenir le nombre de centre d'intérêt :

```
def count_interests(rows: List[Row]) -> int:
    """compte combien de lignes ont des centres d'intérêt non None"""
    return len([row for row in rows if row["interest"] is not None])

user_interest_counts = (users
                       .join(user_interests, left_join=True)
                       .group_by(group_by_columns=["user_id"],
                                aggregates={"num_interests" : count_interests })))
```

SQL propose aussi une instruction `RIGHT JOIN`, qui conserve les lignes de la table de droite sans correspondance, et une instruction `FULL OUTER JOIN`, qui conserve les lignes des deux tables qui ne présentent aucune correspondance. Nous ne les utiliserons pas.

Sous-requêtes

Avec SQL, vous pouvez exécuter une instruction `SELECT` (et `JOIN`) à partir des résultats de requêtes, comme si ces derniers étaient eux-mêmes des tables. Ainsi, si vous cherchez le plus petit `user_id` de tous ceux qui s'intéressent au sujet SQL, vous pouvez exécuter une sous-requête. On peut réaliser le même calcul avec une instruction `JOIN`, mais cela n'illustrerait pas efficacement le cas des sous-requêtes.

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = 'SQL') sql_interests;
```

Étant donné la façon dont a été conçue `NotQuiteABase`, l'opération est « offerte », nos résultats de requêtes étant déjà des tables :

```
likes_sql_user_ids = (user_interests
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns=['user_id']))
likes_sql_user_ids.group_by(group_by_columns=[],
    aggregates={ "min_user_id" : min_user_id })
```

Index

Pour chercher des lignes qui contiennent une valeur particulière (par exemple, dont le nom est Hero), `NotQuiteABase` doit inspecter toute la table. Si cette dernière contient un grand nombre de lignes, cette recherche risque de prendre du temps.

Notre algorithme `join` sera tout aussi inefficace. Pour chaque ligne de la table de gauche, il inspectera chaque ligne de la table de droite à la recherche d'une correspondance. Avec deux grandes tables, ce processus risque de durer éternellement.

En outre, vous aurez peut-être des contraintes à appliquer sur certaines colonnes. Par exemple, dans votre table `users`, vous ne voulez sans doute pas autoriser deux utilisateurs différents à avoir le même `user_id`.

Les *index* sont la réponse à tous ces problèmes. Si la table `user_interests` a un index défini sur `user_id`, un algorithme de jointure intelligent peut trouver des correspondances directement, sans balayer toute la table. Et, si la table `users` a un index « unique » sur l'identifiant `user_id`, vous provoquerez une erreur en essayant d'insérer un doublon.

Chaque table de la base de données peut avoir un ou plusieurs index ; ainsi, vous accélérez la recherche de lignes par colonnes clés, joignez efficacement des tables et imposez facilement des contraintes d'unicité sur des colonnes ou des combinaisons de colonnes.

Bien concevoir et utiliser les index relève de la magie noire (et varie selon la spécificité de la base de données). Toutefois, si vous comptez exploiter intensivement vos bases de données, l'investissement de départ est largement justifié.

Optimisation des requêtes

Souvenez-vous de la requête qui vous permettait de trouver tous les utilisateurs s'intéressant à SQL :

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

Avec `NotQuiteABase`, il existe (au moins) deux façons différentes de l'écrire. Vous pouvez filtrer la table `user_interests` avant de faire la jointure :

```
(user_interests
 .where(lambda row: row["interest"] == "SQL")
 .join(users)
 .select(["name"]))
```

Ou filtrer le résultat de la jointure :

```
(user_interests
 .join(users)
 .where(lambda row: row["interest"] == "SQL")
 .select(["name"]))
```

Dans les deux cas, le résultat est le même. Cependant, filtrer avant jointure est très certainement plus efficace. En effet, dans ce cas, la jointure opère sur un nombre réduit de lignes.

En général, en SQL, on ne se préoccupe pas de ce genre de choses : on « déclare » les résultats voulus et on laisse le moteur de requête se débrouiller (et utiliser les index efficacement).

NoSQL

Une tendance actuelle consiste à se tourner vers des bases de données non relationnelles, dites NoSQL, qui ne représentent pas les données dans des tables. Par exemple, `MongoDB`, un

moteur de bases de données très utilisé, se présente sans schéma et ses éléments sont des documents JSON plus ou moins complexes, plutôt que des lignes.

Il existe également d'autres types de bases de données :

- qui stockent les données dans des colonnes plutôt que dans des lignes (parfaitement adaptées quand elles contiennent un grand nombre de colonnes, mais que les requêtes n'en utilisent que quelques-unes) ;
- qui procèdent à des stockages optimisés par clé-valeur pour récupérer des valeurs uniques (complexes) par leurs clés ;
- pour stocker et parcourir des graphes ;
- optimisées pour s'exécuter à l'échelle de plusieurs centres de données ;
- conçues pour s'exécuter en mémoire ;
- pour stocker des séries temporelles ;
- et des centaines d'autres.

La mode de demain n'existe sans doute pas encore aujourd'hui et je ne peux pas vous en dire beaucoup plus sur le phénomène NoSQL, à part le fait qu'il existe.

Pour aller plus loin

- Si vous souhaitez télécharger un moteur de bases de données relationnelles pour vous exercer, `SQLite`¹ est rapide et peu encombrant, alors que `MySQL`² et `PostgreSQL`³ sont plus volumineux et mieux dotés en fonctionnalités. Tous sont gratuits et très bien documentés.
- Si vous voulez explorer NoSQL, `MongoDB`⁴ est très simple pour commencer, ce qui est à la fois une bénédiction et une malédiction. Sa documentation est, elle aussi, excellente.
- Enfin l'article de Wikipédia sur `NoSQL`⁵ contient certainement des liens vers des bases de données qui n'existaient pas encore au moment de l'écriture de ce livre.

1. <https://www.sqlite.org/index.html>

2. <https://www.mysql.com/>

3. <https://www.postgresql.org/>

4. <https://www.mongodb.com/>

5. <https://fr.wikipedia.org/wiki/NoSQL>

25

MapReduce

Le futur est déjà arrivé, mais il n'a pas encore été distribué partout.

William Gibson

MapReduce est un modèle de programmation destiné aux traitements parallèles sur de grands jeux de données. Bien qu'il soit très puissant, ses principes de base sont relativement simples. Imaginons une collection d'éléments à traiter ; par exemple, journaux de sites web, textes de différents livres ou fichiers d'images. Voici les étapes que suivra une version de base de l'algorithme MapReduce :

Tout ceci est toutefois un peu abstrait. Voyons maintenant un exemple. Il existe peu de règles absolues en data science ; toutefois, l'une d'elles dit que votre premier exemple en MapReduce doit compter des mots.

- 1 Utiliser une fonction de correspondance (`mapper`) pour transformer chaque élément en zéro ou plus paire(s) clé-valeur.

Avertissement

Souvent on parle de fonction `map`, mais il existe déjà une fonction Python qui porte ce nom. Il faudra donc éviter la confusion.

- 2 Collecter toutes les paires ayant des clés identiques.
- 3 Utiliser une fonction de réduction (`reducer`) sur chaque collection de valeurs groupées pour produire des valeurs pour la clé correspondante.

Remarque

MapReduce appartient en quelque sorte au passé, au point que j'ai envisagé de retirer ce chapitre de la deuxième édition. J'ai pourtant considéré qu'il s'agissait d'un thème intéressant et j'ai fini par le laisser.

Exemple : compteur de mots

DataSciencester atteint des millions d'utilisateurs ! C'est sans doute parfait pour la sécurité de votre emploi, mais cela complique singulièrement la moindre tâche d'analyse de données.

Par exemple, votre responsable Contenu veut savoir de quoi parlent les experts lorsqu'ils mettent leurs statuts à jour. En première approche, vous pouvez compter les mots qui apparaissent afin de préparer un rapport sur les mots les plus fréquents.

Quand vous n'aviez que quelques centaines d'utilisateurs, l'opération était facile :

```
from typing import List
from collections import Counter
def tokenize(document: str) -> List[str]:
    """diviser au niveau des espaces"""
    return document.split()

def word_count_old(documents: List[str]):
    """compter les mots sans utiliser MapReduce"""
    return Counter(word
                    for document in documents
                    for word in tokenize(document))
```

Aujourd'hui toutefois, avec des millions d'utilisateurs, le contenu du jeu de documents (les mises à jour des statuts) est devenu trop volumineux pour votre ordinateur. Alors que, si vous arrivez à le gérer dans un modèle MapReduce, vous pourrez faire appel à une infrastructure Big Data mise en œuvre par vos informaticiens.

Tout d'abord, il vous faut une fonction pour transformer un document en séquence de paires clé-valeur. Nous voulons regrouper nos résultats par mots ; en d'autres termes, les clés doivent être les mots. Et, pour chaque mot, nous allons émettre la valeur 1 pour indiquer que cette paire correspond à une occurrence du mot :

```
from typing import Iterator, Tuple
def wc_mapper(document: str) -> Iterator[Tuple[str, int]]:
    """pour chaque mot du document, émettre (mot, 1)"""
    for word in tokenize(document):
        yield (word, 1)
```

Sans nous attarder sur la « mécanique » de l'étape 2 pour le moment, imaginons que vous avez collecté, pour un mot donné, une liste des compteurs correspondants que nous avons

émis. Ainsi, pour produire le comptage global pour ce mot, nous aurons simplement besoin des instructions suivantes :

```
from typing import Iterable
def wc_reducer(word: str,
               counts: Iterable[int]) -> Iterator[Tuple[str, int]]:
    """faire la somme des compteurs pour un mot"""
    yield (word, sum(counts))
```

Revenons maintenant à l'étape 2 : vous devez collecter les résultats de `wc_mapper` pour alimenter `wc_reducer`. Réfléchissons à la manière d'opérer sur un seul ordinateur :

```
from collections import defaultdict
def word_count(documents: List[str]) -> List[Tuple[str, int]]:
    """compter les mots dans les documents à l'aide de MapReduce"""
    collector = defaultdict(list) # lieu de stockage des valeurs groupées
    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)
    return [output
            for word, counts in collector.items()
            for output in wc_reducer(word, counts)]
```

Imaginons maintenant que nous ayons trois documents ["data science", "Big Data", "science fiction"].

`wc_mapper` appliqué au premier document retourne les deux paires ("data", 1) et ("science", 1). Après avoir parcouru les trois documents, nous obtenons dans `collector` :

```
{"data" : [1, 1],
 "science" : [1, 1],
 "big" : [1],
 "fiction" : [1]}
```

Enfin, `wc_reducer` produit le compteur pour chaque mot :

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

Pourquoi MapReduce ?

Grâce à MapReduce, il est possible de *distribuer les calculs* en déplaçant le traitement vers les données. Imaginons que vous vouliez compter des mots dans des milliards de documents.

Notre méthode d'origine (sans MapReduce) suppose que la machine a accès à l'ensemble des documents. Cela signifie que tous les documents doivent être soit présents sur la machine,

soit transférés sur cette dernière au cours du processus. Et, encore plus important, cela signifie que la machine ne peut traiter qu'un seul document à la fois.

Remarque

En réalité, elle peut en traiter plusieurs à la fois si elle dispose de plusieurs cœurs et si le code est réécrit pour en tirer parti. Néanmoins, même ainsi, tous les documents doivent encore parvenir à la machine.

Imaginons maintenant que les milliards de documents sont distribués entre 100 machines. Avec la bonne infrastructure (et en faisant abstraction des menus détails), nous pouvons appliquer la démarche suivante :

- sur chaque machine, exécuter le `mapper` sur les documents et produire beaucoup de paires clé-valeur ;
- distribuer ces paires à un certain nombre de machines « de réduction » en vous assurant que les paires correspondant à une clé donnée finissent toutes sur les mêmes machines ;
- sur chacune des machines de réduction, grouper les paires par clé, puis exécuter le `reducer` sur chaque ensemble de valeurs ;
- retourner chaque paire clé-résultat.

Ce qui est étonnant, avec cette solution, c'est qu'elle peut s'étendre horizontalement. Si nous doublons le nombre de machines, alors (en ignorant certains coûts fixes d'exécution du système MapReduce) nos calculs devraient être approximativement deux fois plus rapides. En effet, chaque machine `mapper` doit réaliser seulement la moitié du travail réalisé précédemment et (en supposant qu'il y ait assez de clés distinctes pour continuer à distribuer équitablement le travail du `reducer`) la même chose s'applique aux machines `reducer`.

MapReduce vu plus généralement

Si vous prenez la peine d'y réfléchir, vous vous rendrez compte que le code spécifique au comptage des mots de l'exemple précédent est contenu dans les fonctions `wc_mapper` et `wc_reducer`. Cela veut dire que, au prix de quelques modifications, nous disposons d'un cadre beaucoup plus général (qui s'exécute toujours sur une seule machine).

Nous pourrions utiliser des types génériques pour intégralement annoter par type notre fonction `map_reduce`. Cependant, l'opération déboucherait sur une sorte de galimatias pédagogique. Aussi, dans ce chapitre, nous nous montrerons beaucoup plus « décontractés » quant à ces annotations de type :

```
from typing import Callable, Iterable, Any, Tuple
# une paire clé-valeur est juste un 2-tuple
KV = Tuple[Any, Any]
# Un Mapper est une fonction qui retourne un élément itérable
# de paires clé-valeur.
```

```
Mapper = Callable[..., Iterable[KV]]
# Un Reducer est une fonction qui accepte une clé et un élément itérable
# de valeurs et retourne une paire clé-valeur.
Reducer = Callable[[Any, Iterable], KV]
```

À présent, nous pouvons rédiger une fonction `map_reduce` générale :

```
def map_reduce(inputs: Iterable,
               mapper: Mapper,
               reducer: Reducer) -> List[KV]:
    """Exécuter MapReduce sur les entrées en utilisant le Mapper
    et le Reducer."""
    collector = defaultdict(list)
    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)
    return [output
            for key, values in collector.items()
            for output in reducer(key, values)]
```

Il est alors simple de compter les mots :

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

Ce code nous fournit la flexibilité nécessaire pour résoudre une grande variété de problèmes. Avant de poursuivre, remarquez que `wc_reducer` est une simple somme des valeurs correspondant à chaque clé. Ce type d'agrégation est suffisamment courant pour justifier de le traiter avec une abstraction :

```
def values_reducer(values_fn: Callable) -> Reducer:
    """retourner un Reducer qui applique fn_val à ses valeurs"""
    def reduce(key, values: Iterable) -> KV:
        return (key, values_fn(values))
    return reduce
```

À partir de cette abstraction, il est facile de créer divers reducers :

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))

assert sum_reducer("key", [1, 2, 3, 3]) == ("key", 9)
assert min_reducer("key", [1, 2, 3, 3]) == ("key", 1)
assert max_reducer("key", [1, 2, 3, 3]) == ("key", 3)
assert count_distinct_reducer("key", [1, 2, 3, 3]) == ("key", 3)
```

Et ainsi de suite.

Exemple : analyser les mises à jour de statuts

Le responsable Contenu a été impressionné par les comptes de mots et vous demande ce que vous pouvez apprendre d'autres des mises à jour des statuts. Vous parvenez à extraire un ensemble de données lié à ces actualisations, comme suit :

```
status_updates = [{"id": 2,
                  "username": "joelgrus",
                  "text": "Dois-je écrire une deuxième édition"
                        " de mon ouvrage sur la data science ?"},
                  {"created_at": datetime.datetime(2018, 2, 21, 11, 47, 0),
                  "liked_by": ["data_guy", "data_gal", "mike"]},
                  # ...
                ]
```

Disons que nous avons besoin de déterminer le jour de la semaine au cours duquel les gens parlent le plus de data science. Pour ce faire, nous allons simplement compter le nombre de mises à jour relatives à ce sujet que compte chaque jour de la semaine. En d'autres termes, nous devons faire un regroupement par jour de la semaine ; ce sera notre clé. Et si nous émettons une valeur de 1 pour chaque mise à jour qui contient `data science`, il nous suffit ensuite de calculer la somme de toutes les occurrences :

```
def data_science_day_mapper(status_update: dict) -> Iterable:
    """donne (day_of_week, 1) si status_update contient "data science""""
    if "data science" in status_update["text"].lower():
        day_of_week = status_update["created_at"].weekday()
        yield (day_of_week, 1)

data_science_days = map_reduce(status_updates,
                               data_science_day_mapper,
                               sum_reducer)
```

Voyons à présent un exemple un peu plus compliqué. Imaginons que vous ayez besoin, pour chaque utilisateur, de trouver le mot le plus usité dans ses mises à jour de statut. La fonction `mapper` permet trois approches à cette fin :

- mettre le nom de l'utilisateur en clé et les mots et les compteurs en valeurs ;
- mettre le mot en clé et les noms d'utilisateurs et les compteurs en valeurs ;
- mettre le nom d'utilisateur et le mot dans la clé et les compteurs en valeurs.

À la réflexion, nous allons regrouper nos résultats par nom d'utilisateur, car nous voulons considérer séparément les mots de chaque utilisateur. De plus, nous ne souhaitons pas les grouper par mot, car notre fonction `reducer` doit voir tous les mots pour chaque personne afin de trouver le plus populaire. La première option est donc la bonne :

```
def words_per_user_mapper(status_update: dict):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
```



```

        yield (user, (word, 1))

def most_popular_word_reducer(user: str,
                              words_and_counts: Iterable[KV]):
    """Soit une séquence de paires (mot, compteur),
    retourner le mot affichant le compte total le plus élevé."""
    word_counts = Counter()
    for word, count in words_and_counts:
        word_counts[word] += count
    word, count = word_counts.most_common(1)[0]
    yield (user, (word, count))

user_words = map_reduce(status_updates,
                        words_per_user_mapper,
                        most_popular_word_reducer)

```

Nous pourrions aussi chercher le nombre de personnes ayant aimé (*liker*) la publication pour chaque utilisateur :

```

def liker_mapper(status_update: dict):
    user = status_update["username"]
    for liker in status_update["liked_by"]:
        yield (user, liker)

distinct_likers_per_user = map_reduce(status_updates,
                                      liker_mapper,
                                      count_distinct_reducer)

```

Exemple : multiplication matricielle

Nous avons déjà présenté la multiplication matricielle (chapitre 22) : soit une matrice A $[n, m]$ et une matrice B $[m, k]$, le résultat de leur multiplication est une matrice C $[n, k]$. L'élément de C en ligne i et colonne j est donné par la formule suivante :

```

C[i][j] = sum(A[i][x] * B[x][j] for x in range(m))

```

Cette approche fonctionne si nous représentons nos matrices sous forme de listes de listes, comme nous l'avons fait auparavant. En revanche, les très grandes matrices sont souvent peu denses, c'est-à-dire que leurs éléments sont pour la plupart nuls. Avec des matrices peu denses de grande envergure, une représentation sous forme de liste de listes peut constituer un gâchis d'espace. Une représentation plus compacte stockera uniquement les emplacements des valeurs non nulles :

```

from typing import NamedTuple
class Entry(NamedTuple):
    name: str

```

```
i: int
j: int
value: float
```

Par exemple, une matrice 1 milliard $\times$ 1 milliard possède un trillion d'entrées, ce qui ne va pas être facile à stocker sur un ordinateur. S'il n'y a que quelques entrées non nulles par ligne, cette représentation alternative est plus compacte de plusieurs ordres de grandeur.

Avec ce type de représentation, nous pouvons utiliser `MapReduce` pour calculer des produits matriciels de manière distribuée.

Pour comprendre notre algorithme, notez que chaque élément `A[i][j]` est seulement utilisé pour calculer les éléments de `C` dans la ligne `i` et que chaque élément `B[i][j]` est utilisé exclusivement pour calculer les éléments de `C` en colonne `j`. Notre but est que chaque sortie de notre `reducer` corresponde à une entrée unique de `C` ; notre `mapper` doit donc émettre des clés identifiant une entrée unique de `C`. Cela nous amène à suggérer les fonctions suivantes :

```
def matrix_multiply_mapper(num_rows_a: int, num_cols_b: int) -> Mapper:
    # C[x][y] = A[x][0] * B[0][y] + ... + A[x][m] * B[m][y]
    # donc un élément A[i][j] entre dans chaque C[i][y]
    # de coefficient B[j][y]
    # et un élément B[i][j] entre dans chaque C[x][j]
    # de coefficient A[x][i].

    def mapper(entry: Entry):
        if entry.name == "A":
            for y in range(num_cols_b):
                key = (entry.i, y) # élément de C
                value = (entry.j, entry.value) # entrée dans la somme
                yield (key, value)
        else:
            for x in range(num_rows_a):
                key = (x, entry.j) # élément de C
                value = (entry.i, entry.value) # entrée dans la somme
                yield (key, value)
    return mapper
```

Puis :

```
def matrix_multiply_reducer(key: Tuple[int, int],
                           indexed_values: Iterable[Tuple[int, int]]):
    results_by_index = defaultdict(list)
    for index, value in indexed_values:
        results_by_index[index].append(value)
    # Multiplier les valeurs des positions ayant deux valeurs
    # (une de A et une de B) puis les additionner.
    sumproduct = sum(values[0] * values[1]
                     for values in results_by_index.values()
                     if len(values) == 2)
```

```
if sumproduct != 0.0:  
    yield (key, sumproduct)
```

Prenons par exemple les deux matrices suivantes :

```
A = [[3, 2, 0],  
      [0, 0, 0]]  
B = [[4, -1, 0],  
      [10, 0, 0],  
      [0, 0, 0]]
```

Vous pouvez les réécrire sous forme de tuples :

```
entries = [Entry("A", 0, 0, 3),  
           Entry("A", 0, 1, 2),  
           Entry("B", 0, 0, 4),  
           Entry("B", 0, 1, -1),  
           Entry("B", 1, 0, 10)]  
mapper = matrix_multiply_mapper(num_rows_a=2, num_cols_b=3)  
reducer = matrix_multiply_reducer  
# le produit doit être [[32, -3, 0], [0, 0, 0]]  
# il doit donc présenter deux entrées  
assert (set(map_reduce(entries, mapper, reducer)) == {(0, 1), -3}, {(0, 0), 32})
```

Cette représentation n'est pas très intéressante lorsque vos matrices sont petites mais, si vous avez des millions de lignes et de colonnes à traiter, elle vous sera d'une grande aide.

Aparté : les fonctions de combinaison

Vous avez probablement remarqué que plusieurs de nos `mapper` semblent inclure beaucoup d'informations supplémentaires. Par exemple, s'agissant de compter les mots, plutôt que d'émettre `(word, 1)` et de faire la somme pour toutes les valeurs, nous aurions pu émettre `(word, None)` et prendre seulement la longueur.

Une des raisons qui nous en a empêchés tient à ce que, dans un environnement distribué, nous voulons parfois utiliser des fonctions de combinaison pour réduire la quantité de données à transférer de machine à machine. Si une de nos machines `mapper` voit 500 fois le mot `data`, nous pouvons lui dire de combiner les 500 instances de `(data, 1)` dans une seule `(data, 500)` avant de transmettre le résultat à la machine `reducer`. Ainsi, nous aurons beaucoup moins de données en mouvement, ce qui peut accélérer substantiellement notre algorithme.

Notre mise en œuvre du `reducer` lui permettra de traiter ces données combinées correctement, ce qui ne serait pas le cas si nous avions utilisé `len`.

Pour aller plus loin

- MapReduce a perdu en popularité par rapport à l'époque où j'écrivais la première édition de cet ouvrage. Il n'est sans doute pas nécessaire d'y consacrer beaucoup de votre temps.
- Cela étant, le système MapReduce le plus largement usité est Hadoop¹. Il existe différentes distributions commerciales et non commerciales, ainsi qu'un immense écosystème d'outils liés à Hadoop.
- Amazon.com propose un service Elastic MapReduce² qu'il sera probablement plus facile d'utiliser plutôt que de mettre en œuvre votre propre *cluster*.
- Les tâches Hadoop ont en général un temps de latence important, ce qui les disqualifie pour les analyses en temps réel. Pour de telles charges de travail, Spark³ constitue un choix compatible avec MapReduce très répandu.

1. <http://hadoop.apache.org/>

2. <https://aws.amazon.com/fr/emr/>

3. <http://spark.apache.org/>

Éthique des données

La bouffe d'abord, puis l'éthique.

Bertolt Brecht

Qu'est-ce que l'éthique des données ?

L'usage des données entraîne leur mésusage. Si la chose s'est toujours plutôt vérifiée, l'idée s'est récemment réifiée sous forme « d'éthique des données » apparaissant de manière quelque peu dominante dans l'actualité.

Par exemple, lors des élections américaines de 2016, l'entreprise Cambridge Analytica avait accédé de manière illicite à des données Facebook et avait utilisé celles-ci pour un ciblage publicitaire politique¹.

En 2018, une voiture autonome testée par Uber avait renversé et tué un piéton (une « conductrice de sécurité » était présente dans le véhicule, mais ne faisait visiblement pas attention à cet instant)².

Des algorithmes sont utilisés pour prédire le risque de voir des criminels récidiver et pour fixer leur peine en conséquence. Est-ce plus ou moins juste qu'un jugement humain³ ?

1. https://fr.wikipedia.org/wiki/Scandale_Facebook-Cambridge_Analytica

2. <https://www.nytimes.com/2018/05/24/technology/uber-autonomous-car-ntsb-investigation.html>

3. <https://www.themarshallproject.org/2015/08/04/the-new-science-of-sentencing>

Certaines compagnies aériennes attribuent à des familles des sièges séparés, les obligeant ainsi à payer un supplément pour s'asseoir ensemble⁴. Le data scientist doit-il s'interposer pour empêcher ces comportements ? Nombre d'entre eux considèrent que oui.

« L'éthique des données » entend fournir des réponses à ces questions, ou tout au moins un cadre pour y faire face. Je ne serai pas arrogant au point de vous dire comment réfléchir à ces sujets (et « ces sujets » évoluent rapidement). Aussi, dans ce chapitre, nous ferons juste un tour rapide de certaines des questions les plus pertinentes ; tour rapide qui (je l'espère), saura inspirer vos futures réflexions sur lesdites questions. Hélas, je ne suis pas suffisamment calé en philosophie pour aborder l'éthique de A à Z.

Mais qu'est-ce que l'éthique des données ?

Bien. Commençons par la définition de cette éthique des données. Si vous prenez la moyenne des définitions du mot *éthique*, vous obtiendrez quelque chose comme « cadre pour réfléchir à ce qu'est un comportement "juste" ou "injuste" ». L'éthique des données est donc un cadre de réflexion quant à la manière de se comporter justement ou injustement avec les données.

Certains parlent (éventuellement implicitement) de l'éthique des données comme s'il s'agissait de commandements quant à ce que vous pouvez ou ne pouvez pas faire. Parmi ces personnes, certaines constituent une « frange dure » qui travaille à élaborer des manifestes ou encore des engagements obligatoires sur lesquels ils espèrent vous faire prêter serment. D'autres encore sont en campagne pour faire de l'éthique une part obligatoire des programmes pédagogiques de la data science ; d'où ce chapitre qui me sert à miser sur leur réussite éventuelle.

Remarque

Curieusement, peu de données laissent entendre que des cours d'éthique conduisent à un comportement éthique, auquel cas cette campagne est en soi vraisemblablement non éthique^a !

a. <https://www.washingtonpost.com/news/on-leadership/wp/2014/01/13/can-you-teach-businessmen-to-be-ethical/>

D'autres personnes (dont votre serviteur) considèrent que les gens raisonnables sont souvent en désaccord sur des questions subtiles de bien et de mal. Et c'est là la part importante de l'éthique des données : s'impliquer pour envisager les conséquences éthiques de son propre comportement. Cette attitude requiert une compréhension de ces choses dont nombre de partisans de l'éthique des données prônent la désapprobation, même si cette compréhension n'implique pas nécessairement d'être d'accord avec eux.

4. <https://twitter.com/RStatsGanesh/status/1066161967105216512>

Dois-je me soucier de l'éthique des données ?

Quel que soit votre métier, vous devez faire preuve d'éthique. Si votre travail implique des données, vous êtes libre de fixer votre seuil de tolérance en la matière. En revanche, vous devez faire preuve de la même éthique dans tous les aspects de votre profession, données comprises.

Les métiers des secteurs technologiques sont sans doute spécifiques, au sens où la technologie monte en puissance. Ainsi, les décisions prises par les personnes qui travaillent dans ces domaines (qu'ils soient liés ou non aux données) présentent potentiellement des effets de plus grande portée.

Un minuscule changement apporté à un algorithme d'exploration des actualités peut constituer, pour un article publié, la différence entre des millions de lecteurs... et l'absence de lecteur.

Un unique algorithme défectueux utilisé dans tout le pays pour gérer les libérations conditionnelles affecte des millions de personnes, là où une commission, même biaisée, n'affecte que les quelques personnes qui passent devant elle.

Alors oui, en règle générale, vous devez vous soucier des effets de votre travail sur le monde. Et plus les effets de votre travail sont importants, plus vous devez vous inquiéter de ce genre de choses.

Malheureusement, certains discours autour de l'éthique des données impliquent que des personnes tentent de vous forcer à accepter leurs conclusions en la matière. Et le fait que vous vous souciez des mêmes choses que ces personnes ne tient qu'à vous.

Élaborer de mauvais produits de données

Certaines questions d'éthique des données sont la conséquence de l'élaboration de mauvais produits.

Par exemple, Microsoft a lancé un *chat bot* baptisé Tay⁵. Celui-ci répétait comme un perroquet des éléments qui lui parvenaient par tweets. Le Tout-Internet ne tarda pas à découvrir qu'il pouvait accéder à Tay pour tweeter toutes sortes de choses choquantes. Il est peu probable que quelqu'un chez Microsoft ait débattu du caractère éthique de lancer un *bot* « raciste ». Plus probable est l'hypothèse que l'outil ait été construit sans considérer la manière dont il pouvait être exploité de manière abusive. L'exemple est sans doute extrême, mais convenons au moins du fait que vous devez réfléchir à la manière dont ce que vous élaboriez peut être exploité à des fins préjudiciables.

Autre exemple, Google Photos utilisait à un moment donné un algorithme de reconnaissance d'image qui parfois classait les photos de personnes de couleur dans une catégorie « gorilles »⁶. Ici encore, il est très peu probable qu'une personne chez Google ait explicitement décidé d'incorporer cette fonction (sans parler de se préoccuper de son caractère

5. [https://fr.wikipedia.org/wiki/Tay_\(intelligence_artificielle\)](https://fr.wikipedia.org/wiki/Tay_(intelligence_artificielle))

6. <https://www.theverge.com/2018/1/12/16882408/google-racist-gorillas-photo-recognition-algorithm-ai>

« éthique »). Le problème vient sans doute d’une combinaison de mauvaises données d’entraînement, de modèle imprécis et du caractère injurieux brutal de l’erreur (si le modèle avait occasionnellement classé des boîtes aux lettres dans une catégorie « camions de pompiers », personne ne s’en serait soucié).

Dans ce cas, la solution est moins évidente ; comment vous assurer que votre modèle entraîné n’émettra pas de prédictions insultantes de quelque façon que ce soit ? Bien sûr, vous devez entraîner (et tester) votre modèle sur une plage diversifiée d’entrées, mais comment être sûr qu’une entrée particulière, quelque part, n’entraînera pas un comportement de votre modèle qui vous mettra dans l’embarras ? Le problème est difficile à gérer (Google semble l’avoir « résolu » en refusant simplement de prédire « gorille »).

Compromis entre précision et équité

Imaginons que vous construisiez un modèle qui prédit la propension d’un public à effectuer certaines actions. Vous avez fait du bon travail (tableau 26–1).

Tableau 26–1 Du bon travail

Prédiction	Personnes	Actions	%
Improbable	125	25	20 %
Probable	125	75	60 %

Parmi les personnes pour lesquelles vous avez prédit qu’il est improbable qu’elles passent à l’acte, seules 20 % d’entre elles correspondent et 60 % non. Il n’y a rien d’effrayant en apparence.

À présent, imaginons que les personnes soient divisées en deux groupes A et B. Certains de vos collègues s’inquiètent du fait que votre modèle puisse être injuste à l’égard d’un des groupes. Si le modèle ne tient pas compte de l’adhésion à un groupe, il ne prend pas moins en compte différents autres facteurs qui se corrélient avec ladite adhésion selon des modes complexes.

En effet, lorsque nous répartissons les prédictions par groupe, des statistiques surprenantes se font jour (tableau 26–2).

Tableau 26–2 Des statistiques surprenantes

Groupe	Prédiction	Personnes	Actions	%
A	Improbable	100	20	20 %
A	Probable	25	15	60 %
B	Improbable	25	5	20 %
B	Probable	100	60	60 %

Votre modèle est-il injuste ? Les data scientists de votre équipe avancent différents arguments :

- **Argument 1**

Votre modèle classe 80 % du groupe A sous improbable, mais 80 % du groupe B sous probable. Ce data scientist se plaint que le modèle traite les deux groupes de manière

injuste au sens où il génère des prédictions considérablement différentes à l'échelle des deux groupes.

- **Argument 2**

Quelle que soit l'adhésion à un groupe, si vous prédisez *improbable*, vous obtenez 20 % de chance de passer à l'acte et si vous prédisez *probable*, vous obtenez 60 % de chance de passer à l'acte. Ce data scientist insiste sur le fait que le modèle est « imprécis » au sens où ses prédictions semblent signifier la même chose, quel que soit le groupe d'appartenance.

- **Argument 3**

40 / 125 = 32 % du groupe B ont été faussement étiquetés *probable*, tandis que seuls 10 / 125 = 8 % du groupe A ont été faussement étiquetés *probable*. Ce data scientist (qui considère une prédiction *probable* comme une mauvaise chose) insiste sur le fait que le modèle stigmatise (est injuste avec) le groupe B.

- **Argument 4**

20 / 125 = 16 % du groupe A ont été faussement étiquetés *improbable*, tandis que seuls 5 / 125 = 4 % du groupe B ont été faussement étiquetés *improbable*. Ce data scientist (qui considère une prédiction *improbable* comme une mauvaise chose) insiste sur le fait que le modèle stigmatise (est injuste avec) le groupe A.

Lequel de vos collègues est dans le vrai ? Et d'ailleurs, l'un d'eux l'est-il ? Peut-être ; tout dépend du contexte.

Vous donnerez éventuellement raison à l'un si les deux groupes sont *Hommes* et *Femmes*, à l'autre si les deux groupes sont *Utilisateurs de R* et *Utilisateurs de Python*, ou peut-être pas s'il s'avère que les utilisateurs de Python ont tendance à être des hommes et ceux de R des femmes.

Peut-être encore aurez-vous un penchant particulier si le modèle est destiné à prédire si un utilisateur DataSciencester postulera ou non pour un poste via la page de candidature du site, ou un autre penchant si le modèle prédit si un utilisateur réussira ou non un entretien.

Votre opinion peut éventuellement dépendre du modèle proprement dit, des caractéristiques dont il tient compte et des données sur lesquelles il a été entraîné.

Quel que soit le cas, l'idée ici est d'insister sur le fait qu'il peut y avoir compromis entre *précision* et *équité* – bien sûr, selon votre manière propre de définir ces concepts – et que ce compromis ne présente pas systématiquement de solutions « correctes » évidentes.

Collaborer

Les officiels gouvernementaux d'un pays totalitaire (selon vos standards) ont finalement décidé de permettre à leurs citoyens d'adhérer à DataSciencester. Ils insistent toutefois sur le fait que les utilisateurs de leur pays ne seront pas autorisés à débattre de l'apprentissage profond. En outre, ils veulent que vous leur rapportiez les noms de tous les utilisateurs qui ne feraient même qu'essayer de rechercher des informations sur ce sujet.

Les scientifiques des données de ce pays sont-ils mieux lotis avec un accès à un DataSciencester restreint (et surveillé) que vous seriez à même de fournir ? Ou bien les restrictions proposées sont-elles si exécrables qu'il vaudrait mieux ne donner aucun accès ?

Interpréter

Le responsable RH de DataSciencester vous demande de développer un modèle prédisant les employés les plus susceptibles de quitter l'entreprise, ceci afin de pouvoir intervenir en tentant d'améliorer leur sort (le taux d'attrition est une composante importante du classement des « 10 meilleures entreprises en termes de conditions de travail » que révèlent certains magazines et dans lequel votre P.-D.G. aspire à figurer).

Vous avez collecté un assortiment de données d'historique et vous envisagez trois modèles :

- un arbre de décision ;
- un réseau neuronal ;
- un « expert en rétention » au coûté élevé.

Un de vos collègues insiste sur le fait que vous devriez simplement utiliser le modèle le plus performant.

Un deuxième argue que vous ne devriez pas utiliser le modèle du réseau neuronal, puisque seuls les deux autres expliquent leurs prédictions, permettant au service RH d'instituer des changements de grande ampleur (par opposition à des interventions ponctuelles).

Enfin, un troisième explique que, si « l'expert » peut proposer une explication pour ses prédictions, rien ne justifie de la prendre pour argent comptant et de considérer qu'il dépeint les vraies raisons qui conduisent à ses prédictions.

Comme pour nos autres exemples, il n'existe ici aucun choix meilleur et absolu. Dans certaines circonstances (éventuellement pour des raisons juridiques ou si vos prédictions induisent un certain bouleversement de l'activité), vous préférerez opter pour un modèle qui fonctionne moins bien, mais dont les prédictions peuvent être expliquées. Dans d'autres, vous vous en tiendrez au modèle qui fournit les meilleures prédictions. Et dans d'autres encore, peut-être n'existe-t-il aucun modèle interprétable qui se comporte correctement.

Recommander

Comme nous l'avons vu au chapitre 23, recommander des biens ou des services à des gens constitue une application répandue de la data science. Lorsqu'un utilisateur regarde une vidéo sur YouTube, ce site lui recommande des vidéos à suivre susceptibles de l'intéresser.

YouTube gagne de l'argent via la publicité. Le réseau a (vraisemblablement) intérêt à vous recommander des vidéos que vous pourriez apprécier afin de prolonger votre visionnage et ainsi vous présenter davantage de publicités. Toutefois, il s'avère que certaines personnes

aiment regarder des vidéos relatives à des théories du complot, vidéos qui tendent alors à figurer dans les recommandations.

Remarque

Au moment où j'écris ce chapitre, si vous tapez « Saturn » dans YouTube, le troisième résultat qui s'affiche est le suivant : « *Something is happening on Saturn... Are THEY hiding it?* » (Il se passe quelque chose sur Saturne... Nous le cache-t-on ?) Notons qu'il s'agit de la version anglaise de YouTube et que la saisie de recherche s'effectue aux États-Unis. Le résultat vous donne une idée du type de vidéos dont je parle.

YouTube a-t-il obligation de ne pas recommander de vidéos sur le thème de la conspiration, même si beaucoup de gens veulent les visionner ?

Exemple différent, si vous effectuez une recherche sur google.com (ou bing.com), le moteur de recherche va suggérer des solutions en complétant le début de votre saisie (mode *autocomplete*). Ces suggestions reposent (au moins en partie) sur les recherches d'autres internautes ; c'est le cas particulièrement si ces autres internautes effectuent des recherches inconvenantes susceptibles de se refléter dans vos suggestions.

Un moteur de recherche doit-il filtrer positivement les suggestions qui ne sont pas à son goût ? Google (pour quelque raison) semble avoir tendance à ne pas suggérer de réponses liées à la religion. Par exemple, si vous tapez « mitt romney m » dans Bing, la première suggestion est « mitt romney mormon » (ce à quoi je m'attendais). En revanche, Google refuse de fournir cette suggestion. Notons ici encore qu'il s'agit de conditions de recherche en anglais aux États-Unis – le principe reste toutefois le même quel que soit le contexte géographique/linguistique.

En effet, Google filtre explicitement les suggestions en complément de saisie qu'il considère « insultantes ou dénigrantes »⁷. La manière dont l'algorithme décide ce qui l'est reste vague. Et pourtant, parfois, la vérité est offensante. Est-il éthique de « protéger » un internaute de ces suggestions ? Est-il éthique de faire le contraire ? Ou n'est-ce pas du tout une question d'éthique ?

Données biaisées

Dans la section « Vecteurs de mots » (chapitre 21), nous avons utilisé un corpus de documents pour l'apprentissage des incorporations des vecteurs de mots. Ces vecteurs ont été conçus pour présenter une similarité de distribution. En d'autres termes, les mots qui figurent dans des contextes similaires doivent présenter des vecteurs similaires. Plus particulièrement, toute inclinaison, ou biais, présent dans les données d'entraînement se reflétera dans les vecteurs de mots proprement dits.

7. <https://blog.google/products/search/google-search-autocomplete/>

Par exemple, si nos documents concernent tous le fait que les utilisateurs de R sont moralement réprouvés et que ceux de Python sont des parangons de vertu, alors il est très probable que le modèle apprendra des associations de ce type pour les mots « Python » et « R ».

À titre plus général, les vecteurs de mots reposent sur une combinaison d'articles de Google News, d'ouvrages, de Wikipédia et de pages web ratissées. En d'autres termes, ces vecteurs apprendront les schémas de distribution présents dans ces sources, quelles qu'elles soient.

Par exemple, si les articles d'actualité relatifs aux ingénieurs logiciels portent en majorité sur des hommes, alors le vecteur appris pour le mot « logiciel » pourrait se trouver plus proche de vecteurs d'autres mots « masculins » que de vecteurs de mots « féminins ».

À ce stade, toute application aval élaborée utilisant ces vecteurs peut également refléter cette proximité. Selon l'application, cette situation peut ou non poser problème. Dans l'affirmative, différentes techniques vous permettent de « supprimer » des biais spécifiques, même si vous ne parviendrez probablement pas à tous les éliminer. Toutefois, ce sont des techniques à connaître.

De la même manière, comme pour l'exemple de Google Photos de la section « Élaborer de mauvais produits de données » de ce chapitre, si vous entraînez un modèle sur des données non représentatives, il est fort probable qu'il se comporte médiocrement en situation réelle, voire de manière insultante ou embarrassante.

En suivant différentes orientations, il est également possible que vos algorithmes puissent codifier des biais tangibles, qui existent bel et bien dans la réalité. Par exemple, votre modèle portant sur la libération conditionnelle peut s'avérer parfait lorsqu'il s'agit de prédire quels criminels libérés seront à nouveau arrêtés. En revanche, si ces nouvelles arrestations sont elles-mêmes le fruit de processus biaisés du monde réel, votre modèle est susceptible de perpétuer ce biais.

Protéger les données

Vous savez beaucoup de choses sur les utilisateurs de DataSciencester. Vous connaissez les technologies qui ont leurs faveurs, leurs amis data scientists, leur lieu de travail, leur rémunération, le temps qu'ils consacrent au site, les publications professionnelles sur lesquelles ils cliquent et ainsi de suite.

Le responsable Monétisation veut vendre ces données à des publicitaires. Ceux-ci sont particulièrement désireux de commercialiser leurs diverses solutions Big Data à vos utilisateurs. Le responsable Data Science veut partager ces données avec des chercheurs universitaires, plus enclins, pour leur part, à publier des articles concernant les profils de ceux qui embrassent la carrière de data scientist. Le responsable Gestion électorale envisage de fournir ces données à des chargés de campagnes politiques en majorité désireux de recruter leurs propres organismes de data science. Quant au responsable Affaires gouvernementales, il souhaiterait exploiter ces données pour répondre à des questions posées par les autorités.

Grâce à l'anticipation du responsable Contrats, vos utilisateurs ont accepté des conditions générales qui vous garantissent le droit de faire pratiquement ce que bon vous semble avec leurs données.

Toutefois (comme vous vous y attendez désormais), certains des spécialistes de votre équipe soulèvent différentes objections quant à ces usages. L'un d'eux considère qu'il n'est pas éthique de remettre des données à des publicitaires. Une autre s'inquiète du fait que les universitaires ne puissent pas garantir une protection responsable des données. Un troisième pense que l'entreprise doit se tenir à l'écart de la politique, tandis qu'un dernier insiste sur le fait qu'on ne peut pas faire confiance à la police et que collaborer avec les autorités serait susceptible de porter préjudice à des personnes innocentes.

Le point de vue de l'un de ces scientifiques des données est-il justifié ?

En résumé

Les éléments dont il faut se soucier sont nombreux ! Et il en existe une infinité que nous n'avons pas même mentionnés et bien plus encore à venir que nous ne connaissons pas encore.

Pour aller plus loin

- Les gens qui émettent des considérations importantes sur l'éthique des données ne manquent pas. Une recherche sur Twitter (ou sur votre site d'actualités favori) constituera probablement la meilleure manière de découvrir la controverse la plus en vogue en matière d'éthique des données.
- Si vous souhaitez quelque chose de légèrement plus applicable, Mike Loukides, Hilary Mason et DJ Patil ont écrit un ebook condensé, *Ethics and Data Science*⁸, sur la mise en pratique de l'éthique des données. J'ai l'honneur de le recommander au nom de Mike Loukides, car c'est lui qui a accepté de publier la première édition (en anglais) de cet ouvrage en 2014. Exercice : est-ce éthique de mon point de vue ?

8. <https://www.oreilly.com/library/view/ethics-and-data/9781492043898/>

En avant pour la data science

Et maintenant, encore une fois, je priai ma hideuse créature d'aller et de prospérer.

Mary Shelley

Qu'allez-vous faire à présent ? Si je ne vous ai pas totalement dégoûté de la data science, il vous reste encore beaucoup à apprendre.

IPython

Nous avons mentionné IPython dans cet ouvrage. C'est un interpréteur de commandes, beaucoup plus riche en fonctionnalités que le shell standard de Python : il ajoute des « fonctions magiques » pour (entre autres choses) copier-coller facilement du code (ce qui est normalement assez compliqué en présence de lignes blanches et de mise en page avec espaces) et exécuter des scripts depuis le shell.

La maîtrise d'IPython¹ vous facilitera considérablement la vie, même si vous n'en apprenez qu'un tout petit bout.

1. <http://ipython.org/>

Remarque

Dans la première édition, je recommandais aussi d'apprendre IPython (à présent Jupyter) Notebook. Il s'agit d'un environnement informatique qui vous permet de combiner texte, code Python interactif et représentations graphiques.

Depuis, je suis devenu sceptique quant à ce type d'environnement^a, car j'ai constaté que cela perturbait les débutants et encourageait les mauvaises pratiques de codage (j'ai aussi d'autres raisons d'être sceptique). De nombreuses personnes autres que moi vous encouragerons certainement à utiliser cet environnement ; rappelez-vous simplement que je suis la voix dissidente.

a. <https://twitter.com/joelgrus/status/1033035196428378113>

Mathématiques

Tout au long de ce livre, nous avons abordé l'algèbre linéaire (chapitre 4), les statistiques (chapitre 5), les probabilités (chapitre 6), ainsi que divers aspects de l'apprentissage automatique.

Pour devenir un bon expert en data science, vous devrez en savoir beaucoup plus sur ces sujets. Je vous encourage donc vivement à les étudier tous en profondeur à l'aide des ouvrages recommandés à chaque fin de chapitre, de vos livres préférés ou en suivant des cours en ligne ou en présentiel.

Ne pas partir de rien

Mettre en œuvre des fonctions de A à Z est une excellente technique pour comprendre leur fonctionnement. Cependant, en général, ce n'est ni très bon pour les performances (sauf, bien sûr, si les performances font partie de vos préoccupations initiales), ni facile, ni rapide à prototyper, ni commode pour gérer les erreurs.

En pratique, vous utiliserez le plus souvent des bibliothèques bien conçues qui mettent en œuvre les éléments fondamentaux. Ma proposition d'origine pour cet ouvrage impliquait une deuxième partie – l'apprentissage des bibliothèques – à laquelle l'éditeur s'est heureusement opposé. Depuis la parution de la première édition, Jake VanderPlas a écrit l'ouvrage *Python Data Science Handbook*² (O'Reilly, en anglais), qui constitue une bonne introduction aux bibliothèques pertinentes et un bon complément de lecture au présent livre.

2. <http://shop.oreilly.com/product/0636920034919.do>

NumPy

`NumPy`³ (pour *Numeric Python*) propose des solutions pour effectuer de « vrais » calculs scientifiques. Il dispose de tableaux plus performants que nos vecteurs-listes, de matrices plus performantes que nos listes de listes et offre beaucoup de fonctions numériques prêtes à l'emploi.

`NumPy` est un bloc d'assemblage qui sert de socle à de nombreuses bibliothèques, ce qui le rend particulièrement intéressant.

pandas

`pandas`⁴ propose des structures de données supplémentaires pour travailler avec des jeux de données en Python. Son concept de base est le `DataFrame`, similaire à la classe `Table` de `NotQuiteABase` (chapitre 24), mais avec davantage de fonctionnalités et de meilleures performances.

Si vous envisagez d'utiliser Python pour triturer, couper, regrouper et manipuler des jeux de données, `pandas` est l'outil indispensable.

scikit-learn

`scikit-learn`⁵ est certainement la bibliothèque la plus populaire pour l'apprentissage automatique en Python. Elle contient tous les modèles que nous avons mis en œuvre et bien d'autres encore. Dans la vraie vie, vous ne construirez jamais un arbre de décision à partir de rien : vous déléguerez à `scikit-learn` tout le gros œuvre. De même, vous n'écrirez jamais un algorithme d'optimisation à la main : vous appellerez `scikit-learn` pour trouver directement un algorithme efficace. Sa documentation contient de très nombreux exemples⁶ de ce que peut faire cette bibliothèque (et, plus généralement, de ce que l'apprentissage automatique peut faire).

Représentations graphiques

Les graphiques `matplotlib` que nous avons créés étaient propres et fonctionnels, mais pas particulièrement esthétiques (et pas du tout interactifs). Si vous voulez approfondir vos connaissances en représentation graphique des données, plusieurs possibilités s'offrent à vous.

La première consiste à explorer `matplotlib` de manière plus approfondie, car nous n'avons vu que quelques-unes de ses possibilités. Son site web contient de nombreux exemples⁷ et une galerie⁸ regroupe ceux qui sont les plus intéressants. Si vous voulez créer des représentations statistiques (par exemple, pour un ouvrage imprimé), c'est sans doute l'étape suivante pour vous.

3. <https://numpy.org/>

4. <https://pandas.pydata.org/>

5. <https://scikit-learn.org/stable/>

6. https://scikit-learn.org/stable/auto_examples/

7. <https://matplotlib.org/examples/>

8. <http://matplotlib.org/gallery.html>

Vous devriez aussi explorer `seaborn`⁹, une bibliothèque qui rend `matplotlib` encore plus attractive (entre autres choses).

Pour créer des représentations interactives à partager sur le Web, le choix le plus évident est sans conteste `D3.js`¹⁰, une bibliothèque JavaScript pour créer des *Data Driven Documents* (les fameux 3D : des documents dirigés par les données). Si vous ne connaissez pas très bien JavaScript, vous pouvez choisir des exemples de la `D3 gallery`¹¹ et les adapter à vos traitements de données (les bons data scientists copient depuis la galerie ; les excellents data scientists la pillent).

Même si `D3` ne vous intéresse pas, faire défiler sa galerie donne des idées incroyables en matière de représentation des données.

Enfin, `Bokeh`¹² est un projet qui offre des fonctionnalités de style `D3` à Python.

R

Même s'il est possible d'ignorer totalement `R`¹³, de nombreux experts et de nombreux projets de data science l'utilisent. Il est donc intéressant de se familiariser avec cet outil :

- en partie pour comprendre les exemples et le code abordés sur le blog de ceux qui travaillent avec `R` ;
- en partie pour mieux apprécier l'élégance de Python par comparaison ;
- et en partie pour vous aider à devenir un participant mieux informé sur le débat éternel et passionné qui oppose les partisans de `R` et ceux de Python.

Apprentissage profond

Il n'est nul besoin de pratiquer l'apprentissage profond pour devenir data scientist. Toutefois, vous ne deviendrez pas un expert en vogue sans passer par cette discipline.

Les deux infrastructures d'apprentissage profond les plus répandues pour Python sont `TensorFlow`¹⁴ (Google) et `PyTorch`¹⁵ (Facebook). Internet regorge de tutoriels dédiés à ces deux environnements. Le meilleur y côtoie le pire.

`TensorFlow` est le plus ancien et le plus largement utilisé, mais `PyTorch` reste (à mon avis) plus facile d'emploi et (en particulier) bien plus convivial pour les débutants. Je préfère (et recommande) `PyTorch`. Toutefois, comme on dit, personne ne s'est jamais fait licencier pour avoir choisi `TensorFlow`.

9. <https://seaborn.pydata.org/>

10. <https://d3js.org/>

11. <https://github.com/d3/d3/wiki/Gallery>

12. <https://docs.bokeh.org/en/latest/>

13. <https://www.r-project.org/>

14. <https://www.tensorflow.org/>

15. <https://pytorch.org/>

Trouver des données

Si la data science devient partie intégrante de votre travail, les données vous seront sans doute fournies dans le cadre de vos projets (mais pas toujours). Toutefois, que faire si vous vous intéressez à ce domaine à titre personnel ? Les données sont partout ; voici quelques points de départ :

- [data.gov](https://www.data.gov/)¹⁶ est le portail ouvert du gouvernement américain. Si vous vous intéressez à ce que fait le gouvernement (c'est-à-dire, à peu près à tout en ce moment), c'est votre point de départ.
- Reddit propose quelques forums, [r/datasets](https://www.reddit.com/r/datasets/)¹⁷ et [r/data](https://www.reddit.com/r/data/)¹⁸, où vous pourrez découvrir et demander des données.
- [Amazon.com](https://registry.opendata.aws/)¹⁹ offre une collection de jeux de données publics pour vous encourager à effectuer des analyses avec leurs produits (mais vous êtes libre d'analyser avec ce que vous voulez).
- Robb Seaton a une liste assez originale de données soigneusement organisées disponibles sur son blog²⁰.
- Enfin, Kaggle²¹ est un site qui héberge des compétitions en data science. Je n'ai jamais réussi à me lancer (je n'ai pas l'esprit de compétition dans ce domaine). À vous de voir. Kaggle héberge un grand nombre de jeux de données.
- Google dispose d'un tout nouvel outil de recherche de jeux de données, [Dataset Search](https://datasetsearch.research.google.com/)²². Il vous permet (devinez quoi) de chercher des jeux de données.

Pratiquer la data science

Explorer des catalogues de données, c'est parfait, mais les meilleurs projets (et produits) sont ceux qui vous passionnent. En voici quelques-uns que j'ai personnellement réalisés.

Hacker News

Hacker News²³ est un agrégateur d'actualités et un site de discussion autour des actualités technologiques. Il rassemble de nombreux articles dont la plupart ne m'intéressent pas.

16. <https://www.data.gov/>

17. <https://www.reddit.com/r/datasets/>

18. <https://www.reddit.com/r/data/>

19. <https://registry.opendata.aws/>

20. <https://rs.io/100-interesting-data-sets-for-statistics/>

21. <https://www.kaggle.com/>

22. <https://datasetsearch.research.google.com/>

23. <https://news.ycombinator.com/news>

Il y a quelques années, j'avais décidé de mettre en place un classificateur d'articles sur l'actualité d'Hacker²⁴ pour prédire si un article donné pourrait m'intéresser ou non. Mon initiative n'avait pas été très bien perçue par les utilisateurs du site, qui ne pouvaient pas s'imaginer que quelqu'un puisse ne pas être intéressé par tous les articles.

Pour mener à bien mon projet, j'ai étiqueté manuellement un grand nombre d'articles (pour créer un jeu de données d'apprentissage), puis j'ai choisi les caractéristiques des articles (comme les mots dans le titre ou les domaines des liens) et, enfin, j'ai entraîné un classificateur naïf de Bayes sur le modèle de notre filtre antispam.

Pour des raisons que l'histoire ne retiendra pas, j'avais programmé en Ruby. Puissent mes erreurs vous servir de leçon.

Camions de pompiers

J'habite sur une grande artère au centre de Seattle, à mi-chemin entre une caserne de pompiers et la plupart des incendies de la ville (du moins, c'est mon impression). Au fil des années, j'ai donc développé un certain intérêt pour les pompiers de Seattle.

Heureusement (du point de vue des données), ils ont un site – Real-Time 911²⁵ – qui recense toutes les alertes incendie et les véhicules dépêchés pour celles-ci.

Pour le plaisir, j'ai donc récupéré des années de données et je les ai utilisées pour réaliser une analyse de type réseau social sur les camions de pompiers²⁶. Entre autres choses, j'ai dû pour cela inventer la notion de centralité d'un camion de pompiers, que j'ai appelée Truck-Rank.

T-shirts

J'ai une fille en bas âge et c'est un regret permanent pour moi depuis sa naissance de voir combien les T-shirts « pour filles » sont tristes, comparés à ceux beaucoup plus rigolos « pour garçons ».

En particulier pour les très jeunes enfants, il est évident qu'il y a une grande différence entre les T-shirts proposés pour les fillettes et les garçonnetts. Un jour, je me suis demandé si je pouvais entraîner un modèle à reconnaître les différences... Eh bien, oui²⁷ !

Pour cela, j'ai téléchargé des centaines d'images de T-shirts, je les ai compressées pour en uniformiser la taille, je les ai transformées en vecteurs de pixels de couleur et j'ai construit un classificateur par régression logistique.

Une première méthode a pris en compte seulement les couleurs présentes dans chaque vêtement. Une autre a cherché les 10 composantes principales des images vectorielles et a classé

24. <https://github.com/joelgrus/hackernews>

25. <http://www2.seattle.gov/fire/realtime911/getDatePubTab.asp>

26. <https://github.com/joelgrus/fire>

27. <https://github.com/joelgrus/shirts>

chaque T-shirt à partir de ses projections dans l'espace à 10 dimensions engendré par les vecteurs propres (figure 27-1).

Figure 27-1

Vecteurs propres correspondant à la première composante principale



Des tweets sur le globe

Pendant de nombreuses années, j'ai voulu élaborer un « globe tournant ». À l'occasion des élections américaines de 2016, j'ai développé une petite application web²⁸. Elle « écoutait » les tweets géolocalisés correspondant à une recherche donnée (j'ai utilisé « Trump » car ce mot figurait dans de nombreux tweets à l'époque), puis les affichait en faisant tourner un globe vers leur emplacement d'origine.

Ce projet de données fut entièrement réalisé en JavaScript. Peut-être qu'il vous faudra donc apprendre un peu JavaScript.

Et vous ?

Qu'est-ce qui vous intéresse ? Quelles sont les questions qui vous empêchent de dormir ? Cherchez un jeu de données (ou explorez quelques sites Internet) et pratiquez la data science.

Partager vos trouvailles !

Écrivez-moi à joelgrus@gmail.com ou rejoignez-moi sur Twitter @joelgrus.

28. <https://github.com/joelgrus/twitter-globe>

Index

Symboles

- `__init__`
 - constructeur 34
- `__repr__`
 - méthode, représentation 35
- `%paste`
 - copier-coller 19

A

- abandon 252
- abstraction de couche 238
- accréditation
 - API Twitter 125
- action
 - cours de l', exemple 140
- addition
 - éléments, liste 24
- affectation de tranche 245
- agrégation d'amorce 221
- aléatoire
 - valeur 38
- algèbre linéaire 55
- algorithme
 - éthique des données 359
- all
 - fonction 31
- allocation de Dirichlet
 - latente 282
- Altair 54
- amorce 194
 - arbre aléatoire 221
- Anaconda
 - questionnaire, conda 17
 - Python, distribution 16
- analyse des réseaux 303

- analyse en composantes
 - principales 147
- annotation de type 42
 - rédaction 44
- antispam
 - filtre 173
- any
 - fonction 31
- API 122, 124
 - recherche de 125
- apprentissage
 - ensemble 167
- apprentissage automatique 153
 - vecteur de mot 287
- apprentissage profond 223, 235, 370
- apprentissage supervisé 259
- arbre
 - étapes, partition 215
- arbre de classification 213, 221
- arbre de décision 211
 - construction 216
 - nœud feuille, feuille 216
 - représentation 218
- arbre de régression 213, 221
- args 41
- argument
 - arbitraire 41
 - déballage 40
 - fonction 20
- array
 - classe, NumPy 59
- article
 - scores, exemple 322
- assert
 - tests automatisés 33

- assertion
 - instruction assert 33
- AssertionError 33
- astuce du noyau 209
- attribut
 - manipulation, arbre de décision 212
 - manquant, default_value 219

B

- bagging 221
- barre oblique inverse
 - caractères spéciaux 22
 - syntaxe, Python 19
- base de données 329
 - table, index 343
- Bayes
 - théorème 173
 - théorème, application 177
 - théorème, probabilités 80
- Beautiful Soup 118, 123
- Bernoulli
 - variable de 86
- biais
 - données, éthique des 363
 - et variance, compromis 160
 - réseau neuronal 225
- bibliothèque
 - externe, combinaison 16
 - matplotlib 47
 - tqdm 145
 - types, module typing 45
- Big Data
 - MapReduce 348
- bigramme
 - modèle 276

- binaire
 - représentation des nombres 231
- bitcoin
 - filtre antispam, exemple 173
- Bokeh 54
- booléen
 - syntaxe 30
- bootstrap 194, 221
- bootstrap aggregating 221
- boucle
 - complexe, continue et break 30
 - for et in 30
 - while 30
- boucle d'entraînement 325
- branchement conditionnel
 - if 29
- break
 - boucle complexe 30
- C**
- caractère
 - RNN 299
- caractère d'espacement
 - syntaxe Python 18
- carré
 - élévation au, fonction 103
 - moindre 184
- causalité
 - essai randomisé 76
 - et corrélation 75
- centralité 310
 - vecteur propre 311
- centralité de degré
 - réseau 304
- centralité de proximité
 - réseau 307
- centralité de vecteur propre
 - réseau 308
- centralité internœud
 - analyse des réseaux 303
 - chemins entre utilisateurs 307
- centre d'intérêt
 - similarité 319
 - utilisateur, recommandations 317
- chaîne
 - valeur, f-string 22
- chaîne de caractères 22
- chemin
 - entre utilisateurs, fonction 305
- classe 34
 - entropie 213
- classification
 - naïve bayésienne 173
- classifieur naïf de Bayes 162, 175
- clé
 - dictionnaire 25
- clé consommateur
 - API 127
- clé secrète
 - API 127
- cluster 259
- clustering 259
- code
 - exemples de VI
 - tests automatisés 33
 - utilisation, droits VI
- coefficient de
 - détermination 186
- coefficient de régression
 - erreur standard 195
- collaboration
 - éthique des données 361
- collecte
 - données 111
- collection
 - éléments distincts, identification 29
- colonne
 - matrice 60
- combinaison
 - fonctions 355
- compte
 - payant ou gratuit, détermination 12
- compte premium
 - exemple 201
 - interprétation 206
- compteur 28
 - spam_messages, ham_messages 176
 - suivi des mots/thèmes 284
- compteur manuel
 - exemple de code 34
- conception
 - principes de, Python 17
- conda
 - Anaconda 17
- connecteur clé 3
- connexion
 - entre utilisateurs, dict 5
- constructeur
 - classe 34
- continue
 - boucle complexe 30
- copier-coller
 - %paste 19
- corrélation 71
 - données parasites 74
 - et causalité 75
 - facteurs parasites 75
 - matrice de 134
- couche
 - abstraction de 238
 - réseau neuronal 225
 - réseau neuronal, représentation 227
 - séquence de, réseau neuronal 242
- couche d'incorporation
 - réseau neuronal 290
- couche linéaire 240
- couleurs
 - partitionnement en groupes, exemple 265

- Counter 28
 - nombre d'amis 66
- courbe 51
 - échelle 54
 - en cloche, distribution normale 83
 - tracé, matplotlib 48
- courriel
 - analyse et filtrage, modèle 179
- CREATE TABLE SQL 329
- crochets
 - liste 23
- csv
 - module, fichier à délimiteur 115
- csv.reader
 - lignes, parcourir 115
- D**
- D3.js 54
- data science
 - définition 2
- data scientist
 - définition 2
- dataclass 137
- DataSciencester 3
- dé
 - lancer, échantillon 280
- décision
 - arbre de 211
 - nœuds, arbre 216
- décompression
 - liste 24
- décorateur
 - classe, NamedTuple 137
- deep learning 223, 235
- defaultdict 27
- DELETE SQL 334
- délimiteur
 - fichiers à 115
- dépendance
 - probabilités 77
- dérivée
 - gradient 102
- dérivée partielle 104
- descente de gradient 101, 186
 - modèle, ajustement 191
 - modèles, adaptation aux 106
 - par mini-lots 108
 - rétropropagation, réseau neuronal 228
 - variance 148
 - vecteurs de mots 291
 - vraisemblance 204
- descente du gradient stochastique 109
- détermination
 - coefficient 186
- diagramme 48
 - échelle 54
- dichotomie
 - recherche 84
- dict 25
 - création, exemple 5
 - defaultdict 28
 - extraction de données 4
- dictionnaire 25
 - defaultdict 27
 - méthode get 26
- dimension
 - données à deux 133
 - et distance 169
 - extraction, PCA 147
 - malédiction de la 168
 - multiple 134
 - multiple, tenseur 235
 - réduction 147
 - unique, données 131
 - vecteur, multiplication matricielle 309
- dispersion 69
 - données 133
 - variance, statistique 70
- distance
 - entre groupes de données 267
 - et dimension 169
 - fonction de génération 168
- distribution
 - a posteriori 98
 - a priori 98
 - discontinue, probabilités 81
 - documents/thèmes 282
 - échantillonnage 280
 - étendue 98
 - lancer de dés 281
 - normale 83
 - poids des mots, thème 283
 - probabiliste, variable aléatoire 282
 - t, loi Student 196
 - thèmes-mots 286
 - uniforme 81, 98
- document
 - thème sous-jacent 282
- dominante
 - valeurs fréquentes 69
- données
 - classification, arbre de décision 212
 - collecte 111
 - d'apprentissage 157
 - échelle 143
 - exploration 131
 - facteurs parasites 74
 - filtre, exemple du spam 161
 - manipulation 140
 - nettoyage 138
 - non conformes 139
 - partitionnement 259
 - statistique 66
 - transformation 138
 - visualisation 47
- dot product 58
- dunder 35
 - méthode 292

E

- écart-type
 - distribution normale 83
 - statistique 70
- échantillon
 - amorçage 194
 - choix aléatoire,
 - random.sample 39
 - collecter, API Streaming 128
 - génératrice, distribution 280
- échantillonnage de Gibbs 280
- échelle 206
 - données 143
 - données, écart-type 144
 - données, espérance 144
 - données, exemple des mensurations 143
- encodage one-hot 289
- ensachage 221
- ensemble 28
- en-tête
 - ligne d', fichier 116
- entraînement
 - modèle, phrases et mots 294
 - réseau neuronal 232
- entropie 213
 - calcul 217
 - surajustement 215
- entropie croisée 249
- environnement virtuel
 - Python, création 16
- epoch 107
- erreur
 - pénalité, régularisation 197
 - ValueError 24
- erreur quadratique 107
- erreur standard
 - coefficient de régression 195
- espace
 - indentation 18
- espace vectoriel 55
- espérance
 - distribution normale 83
 - probabilité 81

état masqué

- réseau neuronal
 - récurrent 296

étendue

- dispersion 69

éthique des données 357, 360

exactitude

- modèle de données 157

except

- exception 21

expérience

- et salaire, exemple de code 9

expression

- mathématique,
 - probabilités 78
- rationnelle 39

extend

- liste, concaténation 24

extraction

- de données, dict 4

F

Facebook

- réseau d'amitiés 303

factorisation de matrice 322

False 31

faux négatif 92, 158

faux positif 158

feed-forward 225

feuille

- nœud feuille, arbre de décision 216

fichier

- à délimiteur 115

- lecture 113

- texte, lecture 114

filter 40

filtrage collaboratif

- utilisateurs,
 - recommandation 317

filtre

- antispam 173

- données, exemple du spam 161

Fizz Buzz

- exemple, réseau neuronal 230

- exemple, revisité 248

fonction

- anonyme, lambda 20

- argument 20

- élaboration de A à Z 368

- lissage, réseau neuronal 226

- privée 68

- Python 19

- Python, modules 20

fonction de densité de

- probabilités 82

fonction de distribution

- cumulative 82

fonction logistique 203

fonction sigmoïde 226, 247

for

- boucle 30

forêt aléatoire 221

forme

- matrice 60

fusion

- groupes de données 268

G

générateur 36

- itération 37

- parenthèses, expression 37

get 27

- méthode 26

Gibbs

- échantillonnage de 280

GitHub

- Python, installation 15

Google Photos

- éthique des données 359

gradient 102

- descente de 186

- erreur quadratique

- moyenne 107

- prévision 107

grammaire 278

- phrase 278

- graphe orienté 312
- graphique
 - matplotlib 369
- graphique à barres 49
 - plt.axis 50
 - plt.bar 50
 - plt.xticks 50
- GROUP BY
 - SQL 337
- groupe
 - de données 259
 - de données, distance 267
 - de données, fusion 268
 - de données,
 - représentation 270
- guillemet
 - simple/double, chaîne 22
- H**
- hachable
 - clé, dictionnaire 27
- hachage 27
- Hacker News 371
- HAVING
 - SQL, clause 337
- histogramme
 - données, dimension
 - unique 132
 - nombre d'amis 66
- HTML
 - analyse syntaxique 117
 - balise 118
 - données, extraction 118
- hyperplan 208
- hypothèse 89
 - décision 91
- hypothèse nulle 89
 - test 97
- I**
- id
 - relations 4
- if
 - branchement
 - conditionnel 29
 - contrôle 29
 - if-then-else
 - branchement
 - conditionnel 30
 - import this
 - description de Python 17
 - in
 - boucle 30
 - ensemble 29
 - incertitude
 - entropie 213
 - probabilités 77
 - indentation
 - espaces et tabulations 18
 - syntaxe, Python 18
 - indépendance
 - probabilités 77
 - index
 - base de données, table 343
 - indicateur
 - de progression, tqdm 145
 - inférence 89
 - inférence bayésienne 98
 - INSERT
 - SQL 329
 - installation
 - Python, GitHub 15
 - instruction
 - SQL, syntaxe 330
 - intérêt
 - centres d', exemple de code 9
 - commun, exemple
 - d'identification 12
 - commun, exemple de
 - comptage 8
 - interface de programmation
 - d'application 122
 - interprétation
 - modèle 193
 - intervalle
 - normal_cd 90
 - intervalle de confiance 94
 - IPython 367
 - installation 17
- Iris
 - jeu de données, exemple 165
- itérateur 36
- itération
 - exemple, amis 7
 - générateur 36
- J**
- JavaScript 123
- jeu de données
 - amorçage 194
- JOIN
 - SQL 340
- jointure
 - table, SQL 341
- JSON 123, 258
 - sérialisation 123
- Jupyter 368
- K**
- key 25
- KeyError 26
- k-moyennes
 - choisir k 264
 - partitionnement,
 - couleurs 265
- kwargs 41
- L**
- lambda
 - fonction anonyme 20
- langage
 - modélisation 275
 - naturel, traitement 273
- langage naturel
 - grammaire 278
- Lasso
 - régression 199
- LDA 282
- Leaf
 - groupes de type 267
- lexique
 - vecteurs de mots 288
- lien
 - d'amitié, réseau 304

- ligne
 - comptage, itération 114
 - fin de, newline 114
 - matrice 60
 - non conforme,
 - traitement 140
 - SQL, regroupement 337
- limite centrale
 - théorème 85
- Linear
 - couche, vecteur de mot 291
- list
 - defaultdict 28
- list comprehension 32
- liste 23
 - agrégation, zip 40
 - appartenance d'un
 - élément 24
 - choix aléatoire,
 - random.choice 39
 - crochets 23
 - décompression 24
 - découpage en tranches 23
 - extend, concaténation 24
 - matrice, représentation 60
 - ordre aléatoire,
 - random.shuffle 38
 - réseau neuronal,
 - représentation 227
 - tri 32
 - vecteurs 58
- liste en compréhension 32
 - boucle for 33
- log
 - vraisemblance 205
- log-vraisemblance négative 250
- loi bêta 99
- M**
- machine learning 153
- magnitude
 - vecteur 59
- maladie
 - test positif, probabilités 81
- manipulation des données 140
- map 40
- mapper
 - MapReduce 347
- MapReduce 347
 - multiplication
 - matricielle 354
- mathématiques 368
- matplotlib 21, 54
 - données, visualisation 47
 - échelle 53
 - installation 48
 - nuage de points 53
 - tracé quadridimensionnel 166
 - tracés, bibliothèque 16
- matplotlib.pyplot
 - module 48
- matrice 59
 - de nuage de points 134
 - factorisation 322
 - jeu de données,
 - représentation 61
 - multiplication 309, 353
 - multiplication de 311
 - utilisateur-intérêt,
 - recommandations 320
- matrice de corrélation
 - dimensions de données 134
- maximum de vraisemblance 187
- médiane
 - statistique 67
- méthode dunder 292
- mini-lot
 - descente de gradient par 108
- mise à jour
 - table, SQL 333
- mise en page
 - syntaxe, Python 18
- MNIST
 - exemple 253
- modèle 153
 - améliorations possibles 181
 - biais et variance,
 - compromis 160
 - complexité et
 - optimisation 156
 - des k plus proches voisins,
 - dimension 168
 - des moindres carrés,
 - ajustement 191
 - descente de gradient 101
 - descente de gradient,
 - adaptation 106
 - enregistrement et
 - chargement 257
 - entraînement, phrases et
 - mots 294
 - interprétation 193
 - linéaire 183
 - linéaire simple 185
 - moindres carrés 186
 - prédiction de notes,
 - films 325
 - prédiction, performances 186
 - supervisé 154
 - surajustement 197, 326
 - surajustement et
 - sous-ajustement 155
 - variables, nombre et
 - ajustement 160
- modèle bigramme 276
- modèle des moindres carrés 190
- modèle linéaire
 - relation linéaire forte 183
- modèle n-grammes 275
- modélisation thématique 282
- module
 - fonctions Python 20
- moindres carrés 184
- momentum
 - optimiseur 245
- MongoDB
 - NoSQL 344
- most_common
 - compteur 28
- mot
 - classe Vocabulary 289

compteur de,
 MapReduce 348
 modèle d'entraînement 294
 modèle n-grammes 277
 nuage de 273
 ordre dans une phrase 296
 phrase, terminaison 279
 position, fréquence 273
 similarité, regroupement 295
 vecteur de 287
 MovieLens 100k
 jeu de données 323
 moyenne
 erreur quadratique,
 gradient 107
 statistique 67
 valeur extrême 68
 moyenne harmonique 159
 multiplication matricielle 308,
 353
 mypy 43

N

NamedTuple 136
 nettoyage
 données 138
 neurone 223
 biais 225
 méthode vers l'arrière 242
 poids 225
 sortie, méthode vers
 l'avant 241
 newline
 fin de ligne 114
 n-grammes
 modèle 275
 noms d'entreprise
 langage naturel, exemple
 des 300
 None
 données non conformes,
 réponse 139
 normal_cd
 intervalle 90

normal_cdf
 probabilités 84
 NoSQL 344
 note
 films, exemple 324
 NotQuiteABase
 exemple, base de
 données 329
 nœud
 centralité entre nœuds,
 calcul 304
 nœud décisionnel
 arbre de décision 216
 nœud feuille 216
 nuage de mots 273
 nuage de points 52
 matrice de 134
 NULL
 valeur 330
 NumPy 369
 array, classe 59

O

objet
 programmation 34
 opérateur 246
 xor, réseau neuronal 227
 optimisation 243
 optimiseur
 abstraction 244
 Optimizer 244
 ORDER BY
 SQL 339
 ou exclusif
 porte logique, revisité 246

P

PageRank 312
 paire
 clé-valeur 26
 clé-valeur, MapReduce 347
 paire d'utilisateurs
 réseau, liens entre 305
 pandas 369

paramètre
 espace et hyperplan 208
 parenthèse
 expression, générateur 37
 tuple, syntaxe 25
 partial 40
 partition
 arbre, entropie 215
 partitionnement 259
 couleurs, k-moyennes 265
 k-moyennes, méthode 260
 partitionnement hiérarchique
 de bas en haut 266
 partitionneur
 groupes de données 261
 pas
 de déplacement, choix 106
 PCA 147
 pénalité
 régularisation 197
 perceptron 224
 perte 243
 fonction de 106
 p-hacking 95
 phrase
 analyse 280
 grammaire 278
 modélisation 278
 terminaison 279
 unité lexicale 279
 pièce
 lancer, hypothèse 90
 lancer, intervalle de
 confiance 94
 lancer, loi a priori/
 posteriori 100
 lancer, variable aléatoire 81
 pipe
 caractère 112
 plt.axis
 graphique à barres 50
 plt.axis("equal")
 diagramme, échelle 54

- plt.bar
 - graphiques à barres 50
 - nombre d'amis 66
 - plt.plot
 - courbe 51
 - plt.xticks
 - graphiques à barres 50
 - plus proches voisins
 - classification 163
 - poids
 - aléatoire, réseau neuronal 229
 - distribution 283
 - mot 283
 - réseau neuronal 225
 - tenseur, génération aléatoire 240
 - thème 283, 285
 - point
 - génération aléatoire 168
 - porte logique
 - and 224
 - not 225
 - or 224
 - or exclusif, revisité 246
 - xor 225
 - xor, construction 227
 - pourriel 173
 - précision
 - et équité, éthique des données 360
 - et rappel, combinaison 159
 - test, exemple de la leucémie 158
 - prédiction
 - exemple des votes 164
 - modèle, performances 186
 - modèles 154
 - première classe
 - fonctions Python 19
 - prévision
 - gradient 107
 - principe
 - de conception, Python 17
 - probabilité 77
 - conditionnelle 78
 - échantillonnage de Gibbs 281
 - échantillons, lancer de dés 281
 - lancer de dés 281
 - souppassement (underflow) 175
 - test bilatéral 93
 - théorème de Bayes 80
 - produit scalaire 58, 72
 - programmation orientée objet 34
 - propagation vers l'avant
 - réseau neuronal 225
 - protection
 - des données, éthique des 364
 - publicité
 - slogan, choix 96
 - p-value 93
 - inférence, P-hacking 96
 - pythonique 17
- Q**
- quantile 69
- R**
- R 370
 - random
 - module, aléatoire 38
 - random.choice
 - choix aléatoire, liste 39
 - random.random()
 - variable aléatoire 82
 - random.sample
 - échantillon, choix aléatoire 39
 - random.seed
 - fonction, aléatoire 38
 - random.shuffle
 - liste, ordre aléatoire 38
 - range
 - itération 36
- rappel
 - et précision, combinaison 159
 - ratissage web
 - noms d'entreprise 299
 - re
 - module 21
 - re.match
 - expression rationnelle 39
 - re.search
 - expression rationnelle 39
 - recommandation
 - entre utilisateurs, réciprocité 312
 - éthique des données 362
 - graphe orienté 313
 - matrices, calcul 320
 - méthode manuelle 316
 - PageRank 314
 - système de 315
 - récurtivité 236
 - reduce 40
 - réduction
 - dimension 147
 - regex
 - caractères, début et fin de mot 300
 - régression
 - coefficient, erreur standard 195
 - simple, modèle de 187
 - régression d'arête 198
 - régression du Lasso 199
 - régression linéaire
 - recherche de modèle 202
 - vraisemblance 204
 - régression linéaire simple 183
 - régression logistique 201
 - régression multiple 189
 - modèle, ajustement du R^2 194
 - régularisation 197
 - relation
 - entre données, modèle 153

- ReLU 248
 - rencontres
 - groupes, exemple 262
 - représentation
 - données, NamedTuple et dict 136
 - requête
 - SQL, optimisation 344
 - réseau
 - analyse 303
 - centralité de degré 304
 - centralité de proximité 307
 - centralité de vecteur
 - propre 308
 - centralité internœud 303
 - chemins entre
 - utilisateurs 305
 - d'utilisateurs,
 - recommandations 313
 - réseau neuronal 223
 - comme séquence de
 - couches 242
 - couche d'incorporation 290
 - entraînement,
 - rétropropagation 228
 - Fizz Buzz, exemple 231
 - propagation vers l'avant 225
 - représentation, couches et listes 227
 - vecteurs de mots 288
 - réseau neuronal récurrent 296
 - SimpleRNN, exemple 298
 - rétropropagation 228
 - RNN 296
 - niveau caractère 299
 - robots.txt
 - fichier, collecte 121
 - Rolex
 - filtre antispam, exemple 174
- S**
- salaire
 - moyen, exemple de code 9
 - sandbox
 - environnement virtuel,
 - Python 16
 - scalaire 55
 - multiplication, vecteur 57
 - produit 58
 - scikit-learn 369
 - score
 - articles, exemple 322
 - script
 - comptage de mots,
 - exemple 112
 - seaborn 54
 - SELECT
 - SQL 334
 - séparateur à vaste marge 207
 - sérialisation
 - données, JSON 123
 - set 28
 - SGBDR 329
 - sigmoïde
 - fonction 247
 - similarité
 - cosinus 288
 - entre utilisateurs,
 - vecteurs 317
 - similarité cosinus 288
 - recommandations 320
 - Simpson
 - paradoxe 74
 - skip-gram
 - vecteur de mot 290
 - slice assignment 245
 - softmax 249
 - solution
 - moindres carrés 184
 - sort
 - tri, fonction 32
 - sorted
 - données, tri 142
 - tri, fonction 32
 - source de données
 - trouver 371
 - sous-ajustement
 - modèle 155
 - sous-classe
 - héritage 35
 - sous-requête
 - SQL 343
 - spam 173
 - SpamAssassin
 - jeu de données,
 - installation 179
 - SQL 329
 - syntaxe 330
 - Stack Overflow 254
 - Star Wars
 - prédiction de notation,
 - exemple 324
 - statistique 65
 - hypothèse 89
 - valeurs extrêmes 67
 - statut
 - mise à jour, analyse 352
 - stdin
 - données, gestion 111
 - stdout
 - données, gestion 111
 - Streaming
 - API, collecte
 - d'échantillons 128
 - string 22
 - structure de contrôle 29
 - Structured Query Language 329
 - Student
 - loi 196
 - surajustement
 - abandon 252
 - arbre de décision 212
 - arbre et forêt 221
 - entropie 215
 - modèle 155, 197, 326
 - syntaxe
 - Python, indentation 18
 - sys.stdin
 - données, gestion 111

sys.stdout
 données, gestion 111
système de
 recommandation 315

T

table
 base de données 329
 base de données, index 343
 SQL, création et
 insertion 329
 SQL, interrogation 334
 SQL, jointure 340
 SQL, mise à jour 333
 SQL, suppression 334

tabulation
 fichier à délimiteur 115
 indentation 18

tangente
 dérivée, gradient 103

Tay
 éthique des données 359

tendance centrale
 statistique 67

tenseur 235
 de poids, génération
 aléatoire 240

terminaison
 phrase 279

test
 bilatéral 93
 ensemble 167
 positif, probabilités 81
 précision, exemple de la
 leucémie 158
 puissance, calcul 92
 p-value 93

thème
 centres d'intérêt,
 utilisateurs 287
 distribution, documents 282
 distribution, mots 282
 distribution, utilisateurs 282
 poids 285

sous-jacent, document et
 utilisateur 282

TLN 273

tqdm 145

traitement du langage
 naturel 273

tranche
 affectation de 245
 données, histogramme 132
 liste, découpage en 23

transformation
 données 138

tri 32
 groupes de données 268
 SQL, ORDER BY 339

True 31

try
 exception 21
tuple 25
 agrégation, zip 40

tutoriel
 Python 15

Twitter
 API, exemple 125

Twython 125

typage
 dynamique 42
 statique 43

type
 bibliothèque de, module
 typing 45

TypeError 43

typing
 module, bibliothèque de
 types 45

U

unité
 lexicale, phrase 279

unpack 24

UPDATE
 SQL 333

URL
 collecte, exemple du
 Congrès 120

utilisateur
 centres d'intérêt,
 recommandations 317
 thèmes, centres d'intérêt 287

V

valeur
 aléatoire, module random 38
 extrême, statistique 67
 multiple, tuple 25
 répartition, graphique 49

valeur aléatoire 38

valeur extrême
 corrélation 73
 étendue, écart-type 71
 influence, apprentissage 168
 moyenne 68

valeur propre
 multiplication
 matricielle 309

value 25

ValueError
 erreur 24

variable
 aléatoire, binomiale 90
 aléatoire, probabilités 81
 appariement, modèle 156
 corrélation 75
 modèle, ajustement 160
 modèle, extraction et
 sélection 161

variable aléatoire
 binomiale 85
 distribution probabiliste 282
 LDA 282

variable factice 189

variance
 dispersion, statistique 70
 et biais, compromis 160
 vecteur, calcul 148

vecteur 55
 addition 56
 de mot 287
 différence entre deux, 57

- distance entre deux 59
- liste 58
- magnitude 58
- multiplication
 - matricielle 309
- multiplication, scalaire 57
- opérations sur 56
- projection 58
- propriétés arithmétiques,
 - mots 287
- régression multiple 190
- similarité cosinus 288
- similarité entre
 - utilisateurs 317
- somme 57
- variance, calcul 148
- vecteur de mot
 - couche Linear 291
 - indexation 292
 - réseau neuronal 288
 - skip-gram 290
- vecteur des sujets 318
- vecteur propre 308, 310
 - centralité 311
- Vector 56
- venv
 - module 16

- virtualenv
 - module 16
- vocabulaire
 - enregistrement/chargement,
 - fonction 290
- voisin
 - ensemble
 - d'apprentissage 167
 - plus proche,
 - classification 163
- vote
 - prédiction, exemple 164
- vrai négatif 158
- vrai positif 158
- vraisemblance
 - estimation du maximum
 - de 187
 - log 204
 - maximisation, descente de
 - gradient 204
 - régression linéaire 204

W

- Web
 - ratissage 117
 - ratissage, noms
 - d'entreprise 299

WHERE

- clause SQL 334
- SQL, clause 337
- while
 - boucle 30
- word_counts 28

X

- XML 123
- xor
 - porte logique 225
 - porte logique,
 - construction 227
 - réseau neuronal 243

Y

- yield
 - itération 36
- YouTube
 - recommandations, éthique
 - des données 362

Z

- zip
 - agrégation 40
 - vecteur, addition 57