Joel Grus

Data science par la pratique

Fondamentaux avec Python

EYROLLES

Préparez-vous aux métiers du futur !

Résumé

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 la discipline sans comprendre la data science. 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 coeur de la data science et à acquérir 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.

- 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, le Bayes naïf, 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.

Sommaire

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

Biographie auteur

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.

Joel Grus

Data science par la pratique

EYROLLES

This book was posted by AlenMiler on AvaxHome!
https://avxhm.se/blogs/AlenMiler

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
de Joel Grus, ISBN : 9781491901427,
publié par O'Reilly Media.

All Rights Reserved.

Attention : pour lire les exemples de lignes de code, réduisez la police de votre support au maximum.

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.
© 2015 by Joel Grus / O'Reilly Media pour l'édition en langue anglaise
© Groupe Eyrolles, 2017, pour la présente édition, ISBN : 978-2-212-11868-1
© Traduction française : Dominique Durand-Fleischer / Relecture technique : Yvan Aillet

Avant-propos

« Expert en data science » a été désigné comme « métier le plus sexy du XXIe siècle » par une journaliste qui n'avait sans doute jamais visité une caserne de pompiers. Mais il est vrai que la data science est un sujet brûlant 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.

Mais qu'est-ce que la data science ou science des données ? Comment former des data scientists si nous ne savons pas répondre à cette question. D'après un diagramme de Venn devenu célèbre chez les spécialistes, 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 rien que le traitement approfondi de « l'expertise fonctionnelle » 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 la data science.

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 pour mettre au point *votre* propre méthode. Tout le code et toutes les données de ce livre sont disponibles sur GitHub pour vous aider à démarrer.

La meilleure façon d'apprendre les mathématiques, c'est de faire des mathématiques. Ceci n'est pas un livre de mathématiques et la plupart du temps nous ne ferons pas « des maths ».

Cependant, vous ne pouvez pas réellement pratiquer la data science sans comprendre un peu les probabilités, les statistiques et l'algèbre linéaire. Cela signifie que toutes les fois que ce sera nécessaire, 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 faire toucher du doigt 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 comme mineur.)

Partir de rien

Il existe pléthore de bibliothèques, frameworks, modules et boîtes à outils dédiés aux data sciences pour mettre en place les algorithmes et les techniques de data science les plus courants (et aussi les moins courants d'ailleurs). Si vous devenez data scientist, vous pénétrerez dans l'intimité de NumPy, scikit-learn, pandas, et toute une panoplie d'autres bibliothèques. Parfaites pour la data science, elles sont aussi une bonne manière de commencer en data science sans réellement comprendre la data science.

Dans cet ouvrage, nous partirons de zéro pour appréhender la data science. 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.

Tout au long 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.) Quelques-uns suggèrent Java ou Scala. Cependant, à mon avis, le choix de Python saute aux yeux.

Python possède plusieurs fonctionnalités qui le rendent particulièrement bien adapté pour apprendre (et pratiquer) 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 existe d'autres langages que je trouve plus agréables, mieux conçus ou simplement plus amusants à utiliser. Et pourtant, presque toutes les fois que je commence un projet de data science, je reviens finalement à Python. Chaque fois que j'ai besoin de faire un prototype rapide mais fonctionnel, je reviens à Python. Et chaque fois que je veux démontrer des concepts de data science de manière claire et limpide, je

finis par utiliser Python. Ce livre fait donc appel à Python.

L'objectif de cet ouvrage n'est pas de vous apprendre Python. (Même s'il est certain qu'en lisant ce livre vous apprendrez aussi un peu Python.) Je vais vous accompagner dans un cours accéléré pendant un chapitre pour souligner les fonctionnalités les plus importantes pour nous, mais si vous ignorez tout de la programmation 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 voie : entrer dans les détails quand il est crucial ou particulièrement instructif d'entrer dans les détails, 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 data scientists que 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 faire de la data science. Tout ce qu'il faut c'est un esprit avide de découverte, de la motivation, et ce livre… D'où ce livre.

Utilisation des exemples de code

Du matériel supplémentaire (exemples de code, exercices…) est disponible, en anglais, en téléchargement à l'adresse https://github.com/joelgrus/data-science-from-scratch.

Ce livre a pour objet de vous aider dans votre travail. En règle générale, vous pouvez utiliser librement le code présenté dans ces pages dans vos programmes et votre documentation. Il n'est donc pas nécessaire de nous contacter pour obtenir un accord, à moins que vous ne souhaitiez reproduire une portion significative du code. Par exemple, l'écriture d'un programme qui reprend plusieurs fragments de code tirés de ce livre ne nécessite pas d'autorisation, mais la vente ou la distribution d'un CD d'exemples tirés de cet ouvrage en exige une. Par ailleurs, si vous répondez à une question en citant ce livre et ses exemples de codes, nul besoin de demander une autorisation, mais pour incorporer une quantité significative de code tiré de ce livre dans la documentation de vos produits, il en faut une.

L'auteur apprécie d'être reconnu en tant que tel mais sans en faire une condition impérative. Cette reconnaissance prend généralement la forme suivante : titre, auteur, éditeur et numéro ISBN. Par exemple cet ouvrage, initialement publié en anglais par O'Reilly sous le titre *Data Science from Scratch* (Copyright 2015 Joel Grus, 978-1-4919-0142-7) a été traduit en français pour les éditions Eyrolles par Dominique Durand-Fleischer.

Si vous pensez que votre usage des exemples de code ne correspond pas à l'usage équitable ni à la demande d'autorisation évoqués ci-dessus, veuillez nous contacter à l'adresse : ahabian@eyrolles.com.

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 mon éditrice, Marie Beaugureau, qui m'a guidé tout au long du processus de publication et qui a hissé ce livre à un état bien meilleur que tout ce que j'aurais pu faire moi-même.

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 la data science sans l'influence de Dave Hsu, Igor Tatarinov, John Rauser, et le reste du gang Farecast (à une époque si lointaine qu'on ne parlait pas encore de data science !). Les organisateurs de Coursera 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 des explications obscures, et 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, et Wendy Grus m'ont tous donné des conseils infiniment avisés. 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 foule 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 (en passant) 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.

Table des matières

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

CHAPITRE 2
Cours accéléré de Python
- **Les fondamentaux**
  - Installer Python
  - Les principes Zen de Python
  - La mise en page par les espaces
  - Les modules
  - L'arithmétique
  - Les fonctions
  - Les chaînes de caractères
  - Les exceptions
  - Les listes
  - Les tuples
  - Les dictionnaires
    - *Les defaultdict*
    - *Les compteurs*
  - Les ensembles

Les structures de contrôle
Vrai/faux
Quelques fonctionnalités avancées de Python
Trier
Les list comprehensions
Générateurs et itérateurs
Les valeurs aléatoires
Les expressions rationnelles
La programmation orientée objet
Les outils fonctionnels
enumerate
zip et le déballage d'arguments
args et kwargs
Bienvenue chez DataSciencester !
Pour aller plus loin

CHAPITRE 3
Visualisation des données
matplotlib
Les diagrammes en bâtons
Les courbes
Les nuages de points
Pour aller plus loin

CHAPITRE 4
Algèbre linéaire
Les vecteurs
Les matrices
Pour aller plus loin

CHAPITRE 5
Statistique
Décrire un unique jeu de données
Les tendances centrales
La dispersion
La corrélation
Le paradoxe de Simpson
Les autres chausse-trappes

Corrélation et causalité
Pour aller plus loin

Chapitre 6
Probabilités
Dépendance et indépendance
La probabilité conditionnelle
Le théorème de Bayes
Les variables aléatoires
Les distributions continues
La distribution normale
Le théorème de la limite centrale
Pour aller plus loin

Chapitre 7
Hypothèse et inférence
Le test statistique d'une d'hypothèse
Exemple : le lancer de pièce
p-values
L'intervalle de confiance
P-hacking
Exemple : effectuer un test A/B
L'inférence bayésienne
Pour aller plus loin

Chapitre 8
Descente de gradient
L'idée qui se cache derrière la descente de gradient
Estimer le gradient
L'utilisation du gradient
Choisir le bon pas
Synthèse
La descente du gradient stochastique
Pour aller plus loin

Chapitre 9
Collecte des données

stdin et stdout
La lecture de fichiers
Les fondamentaux des fichiers texte
Les fichiers à délimiteur
Le ratissage du Web
L'analyse syntaxique du HTML
Exemple : les ouvrages consacrés aux données
L'utilisation des API
JSON (et XML)
L'utilisation d'une API non authentifiée
À la recherche des API
Exemple : l'utilisation de l'API Twitter
L'accréditation
L'utilisation de Twython
Pour aller plus loin

CHAPITRE 10
Travail sur les données
L'exploration des données
Explorer des données à une dimension
Deux dimensions
Plusieurs dimensions
Nettoyage et transformation
La manipulation des données
Le changement d'échelle
La réduction de dimensionnalité
Pour aller plus loin

CHAPITRE 11
Apprentissage automatique
La modélisation
Qu'est-ce que l'apprentissage automatique ?
Surajustement et sous-ajustement
Exactitude
Le compromis entre le biais et la variance
Extraction et sélection des variables
Pour aller plus loin

CHAPITRE 12
k plus proches voisins
- **Le modèle**
- **Exemple : langages préférés**
- **La malédiction de la dimension**
- **Pour aller plus loin**

CHAPITRE 13
Classification naïve bayésienne
- **Un filtre antispam particulièrement stupide**
- **Un filtre antispam plus élaboré**
- **La mise en œuvre**
- **Tester le modèle**
- **Pour aller plus loin**

CHAPITRE 14
Régression linéaire simple
- **Le modèle**
- **L'utilisation de la descente de gradient**
- **L'estimation du maximum de vraisemblance**
- **Pour aller plus loin**

CHAPITRE 15
Régression linéaire multiple
- **Le modèle**
- **Les autres hypothèses du modèle des moindres carrés**
- **Ajuster le modèle**
- **L'interprétation du modèle**
- **Le bon ajustement du modèle**
- **Digression : l'amorce *(Bootstrap)***
- **Les erreurs standards des coefficients de régression**
- **La régularisation**
- **Pour aller plus loin**

CHAPITRE 16
Régression logistique
- **Le problème**

La fonction logistique
L'application du modèle
Le bon ajustement du modèle
Les machines à vecteurs support
Pour aller plus loin

CHAPITRE 17
Arbres de décision
Qu'est-ce qu'un arbre de décision ?
L'entropie
L'entropie d'une partition
La construction d'un arbre de décision
Synthèse
Les forêts aléatoires *(Random Forests)*
Pour aller plus loin

CHAPITRE 18
Réseaux neuronaux
Le perceptron
Les réseaux neuronaux à propagation vers l'avant *(Feed-Forward)*
La rétropropagation
Exemple : déjouer un CAPTCHA
Pour aller plus loin

CHAPITRE 19
Clustering
L'idée
Le modèle
Exemple : rencontres
Choisir k
Exemple : partitionnement de couleurs
Le partitionnement hiérarchique de bas en haut
Pour aller plus loin

CHAPITRE 20
Traitement automatique du langage naturel
Les nuages de mots

Les modèles n-grammes
Les grammaires
Aparté : l'échantillonnage de Gibbs
La modélisation thématique
Pour aller plus loin

CHAPITRE 21
Analyse des réseaux
La centralité internœud
La centralité de vecteur propre
La multiplication matricielle
La centralité
Graphes orientés et PageRank
Pour aller plus loin

CHAPITRE 22
Systèmes de recommandation
La méthode manuelle
Recommander ce qui est populaire
Le filtrage collaboratif sur la base des utilisateurs
Le filtrage collaboratif sur la base des articles
Pour aller plus loin

CHAPITRE 23
Base de données et SQL
CREATE TABLE et INSERT
UPDATE
DELETE
SELECT
GROUP BY
ORDER BY
JOIN
Les sous-requêtes
Les index
L'optimisation des requêtes
NoSQL
Pour aller plus loin

CHAPITRE 24
MapReduce
- **Exemple : un compteur de mots**
- **Pourquoi MapReduce ?**
- **MapReduce vu plus généralement**
- **Exemple : analyser les mises à jour de statuts**
- **Exemple : la multiplication matricielle**
- **Aparté : les combiners**
- **Pour aller plus loin**

CHAPITRE 25
En avant pour la data science
- **IPython**
- **Les mathématiques**
- **Ne pas partir de rien**
  - NumPy
  - pandas
  - scikit-learn
  - Les représentations graphiques
  - R
- **Trouver des données**
- **Faites de la data science**
  - Hacker News
  - Camions de pompiers
  - T-shirts
  - Et vous ?

Index

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 par 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 de la donnée » portent des podomètres gonflés aux stéroïdes pour enregistrer en permanence leur rythme cardiaque, leurs gestes, ce qu’ils mangent et comment ils dorment. 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 mèmes et les cocktails ; et trop de statistiques officielles (certaines d’entre elles presque vraies !) publiées par trop de gouvernements pour alimenter la réflexion.

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 confondent avec des ingénieurs en informatique. Certains sont des experts de l’apprentissage automatique, et d’autres seraient incapables de programmer l’itinéraire de sortie du jardin d’enfants. 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 de la data science, vous trouverez des spécialistes auxquels elle ne s’applique absolument pas !

Mais 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 ramassis de données inextricables. Le monde d’aujourd’hui est rempli de gens qui tentent de donner du sens aux données.

Par exemple, le site de rencontre OkCupid pose à ses membres des tonnes de questions pour trouver le partenaire qui leur correspond le mieux. Mais il analyse aussi ces résultats pour imaginer des questions innocentes qui vous permettront de connaître vos chances de coucher avec elle ou lui dès le premier rendez-vous.

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

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 prédictif, comme savoir quelles clientes sont enceintes pour mieux mettre en valeur les articles de puériculture auprès d’elles.

En 2012, Obama a employé pour sa campagne électorale des douzaines de data scientists pour explorer les bases de données d’électeurs dans le but de trouver ceux qui devaient être particulièrement soignés afin d’orienter les appels de levées de fonds vers certains donateurs et de concentrer les efforts là où ils seraient le plus utiles. Il est généralement admis que tous ces efforts ont joué

un rôle non négligeable dans la réélection du président, ce qui permet d'affirmer que les futures campagnes seront de plus en plus pilotées par les données, avec pour résultat une course aux armements sans fin pour la collecte et l'analyse des données.

Rassurez-vous, certains data scientists mettent de temps en temps leurs compétences au service de la bonne cause, pour rendre les gouvernements plus efficaces, aider les sans-abris ou améliorer la santé publique. Mais votre carrière ne s'en portera pas plus mal si vous préférez imaginer comment inciter les gens à cliquer sur des publicités.

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 de la data science.

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 pas 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 rencontrez 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 a une très forte culture hostile à ce qui « ne vient pas de chez nous », nous construirons nos propres outils à partir de zéro. 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 ! (Vous pouvez venir en jeans le vendredi, et les toilettes sont à droite au fond du couloir.)

À la recherche des connecteurs clés

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

En particulier, il veut que vous identifiiez les « connecteurs clés » parmi les experts de data science. Pour ce faire, il vous remet le contenu complet de tout le réseau DataSciencester. (Dans la vraie vie, il est rare que quelqu’un vous remette ainsi les données qui vous sont nécessaires. Le chapitre 9 traite de la collecte des données.)

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) :

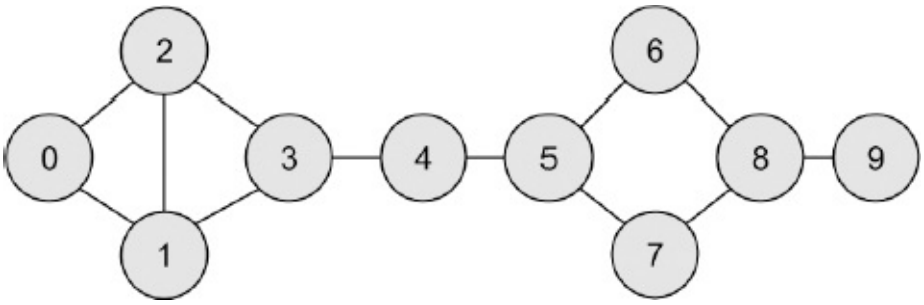
```
users = [
{ "id": 0, "name": "Hero" },
{ "id": 1, "name": "Dunn" },
{ "id": 2, "name": "Sue" },
{ "id": 3, "name": "Chi" },
{ "id": 4, "name": "Thor" },
{ "id": 5, "name": "Clive" },
{ "id": 6, "name": "Hicks" },
{ "id": 7, "name": "Devin" },
{ "id": 8, "name": "Kate" },
{ "id": 9, "name": "Klein"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

Il vous remet aussi les « relations », représentées par une liste de paires d'`id` :

```
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)]
```

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

Figure 1–1
Le réseau DataSciencester



Comme nous représentons nos utilisateurs comme des dict, il est facile de les enrichir de données supplémentaires.

> **Note**
> 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.

Par exemple, si nous voulons ajouter une liste d'amis à chaque utilisateur, il faut d'abord assigner à l'attribut `friends` de chaque utilisateur une liste vide :

```
for user in users:
    user["friends"] = []
```

Et ensuite, il faut remplir la liste à partir des données de l'objet `friendships` :

```
for i, j in friendships:
    # possible parce que user[i] est l'utilisateur dont l'id est i
    users[i]["friends"].append(users[j])  # ajoute j comme ami de i
    users[j]["friends"].append(users[i])  # ajoute i comme ami de j
```

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

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

```
def number_of_friends(user):
    """combien d'amis l'utilisateur _user_ a-t-il ?"""
    return len(user["friends"])  # longueur de la liste  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 :

```
from __future__ import division  # la division entière est bancale
num_users = len(users)           # longueur de la liste des utilisateurs
avg_connections = total_connections / num_users  # 2.4
```

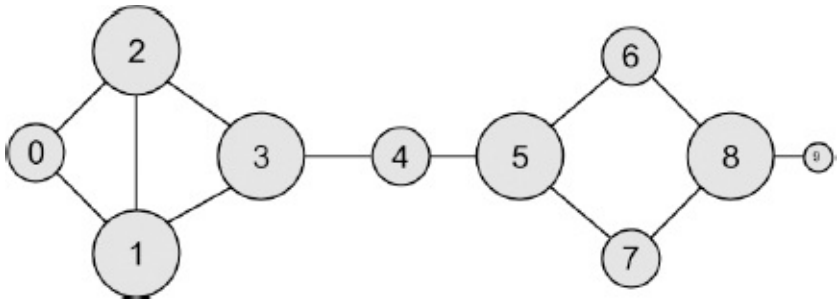
Il est tout aussi facile de trouver les personnes les plus connectées : ce sont les personnes qui ont le plus grand nombre d'amis.

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]
sorted(num_friends_by_id,  # la trier
  key=lambda (user_id, num_friends): num_friends,  # 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, ce que nous avons calculé est le degré de centralité du réseau (figure 1-2).

Figure 1–2
Le réseau DataSciencester dimensionné par degré



L'avantage est que c'est très facile à calculer, mais le résultat n'est pas toujours celui qui est 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. Dans le chapitre 21, 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 entre vos adhérents. Elle vous demande donc de concevoir un système pour suggérer « des experts que vous connaissez peut-être ».

Votre première idée est de suggérer qu'un utilisateur peut connaître les amis de ses amis. C'est facile à calculer : pour chacun des amis d'un utilisateur, il suffit de naviguer parmi les amis de cette personne et collecter les résultats :

```
def friends_of_friend_ids_bad(user):
  # "foaf" est l'abréviation de  "friend of a friend", autrement dit ami d'un ami
  return [foaf["id"]
      for friend in user["friends"]       # pour chaque ami de l'utilisateur
      for for foaf in friend["friends"]]  # obtenir chacun de leurs amis _their_ friends
```

Quand nous appelons cette fonction sur `users[0]` (Hero), on récupère :

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

Ce qui 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 [friend["id"] for friend in users[0]["friends"]]  # [1, 2]
print [friend["id"] for friend in users[1]["friends"]]  # [0, 2, 3]
print [friend["id"] for friend in users[2]["friends"]]  # [0, 1, 3]
```

Savoir que des personnes sont des amis d'amis de plusieurs manières semble être une information intéressante, donc nous devrions 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 not_the_same(user, other_user):
  """deux utilisateurs ne sont pas le même s'ils ont des identifiants différents"""
  return user["id"] != other_user["id"]

def not_friends(user, other_user):
  """l'autre utilisateur n'est pas un ami s'il n'est pas dans  user["friends"];
  C'est-à-dire, s'il n'est pas le même que tous ceux présents dans  user["friends"]"""
  return all(not_the_same(friend, other_user)
      for friend in user["friends"])

def friends_of_friend_ids(user):
  return Counter(foaf["id"]
    for friend in user["friends"]               # pour chacun de mes amis
    for foaf in friend["friends"]               # compter *leurs* amis
    if not_the_same(user, foaf)                 # ceux qui ne sont pas moi
                and not_friends(user, foaf))  # et ne sont pas mes amis

print friends_of_friend_ids(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 l'aspect « domaines 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 l'apprentissage automatique.

Il est facile de construire une fonction qui trouve les utilisateurs partageant un sujet donné :

```
def data_scientists_who_like(target_interest):
    return [user_id
        for user_id, user_interest in interests
        if user_interest == target_interest]
```

Ça marche, 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

# les clés sont les centres d'intérêt, les valeurs sont les listes d'identifiants user_id
# avec 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 :

```
# les clés sont les user_ids, les valeurs sont les listes de centres d'intérêt
# pour cet identifiant d'utilisateur
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 les autres utilisateurs avec ce centre d'intérêt ;
- compter combien de fois nous voyons les autres utilisateurs.

```
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 pourrons alors utiliser ces résultats pour construire une fonctionnalité plus riche des « 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 22.

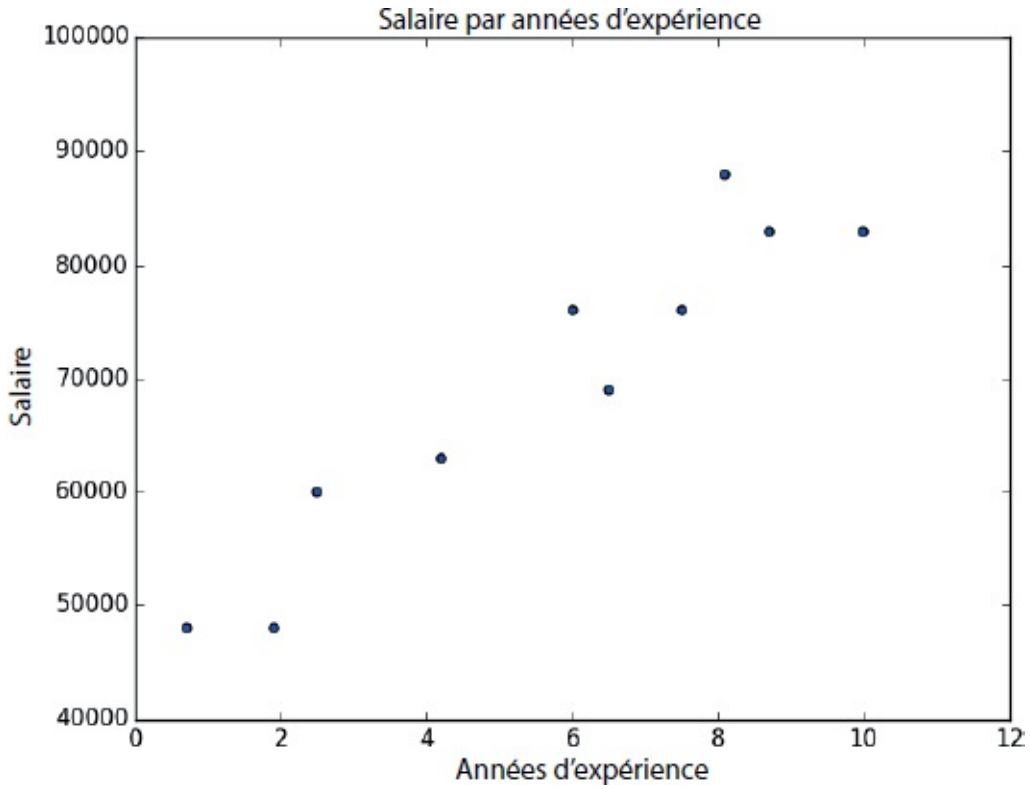
Salaires et expérience

Vous êtes prêt à sortir pour le 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 sont des données sensibles, mais il a réussi 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 les données sur un graphique (nous verrons comment procéder au chapitre 3). Vous pouvez découvrir le résultat obtenu sur la figure 1-3.

Figure 1–3
Salaire par années d'expérience



Il est évident que les personnes les plus expérimentées ont tendance à gagner davantage. Comment illustrer cette réalité ? Votre première idée est de regarder le salaire moyen par nombre d'années d'ancienneté :

```
# les clés sont les années, les valeurs sont les listes 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)

# les clés sont les années, chaque 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"
```

ensuite de regrouper les salaires par catégories :

```
# les clés sont les catégories de postes, les valeurs sont les listes 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, de calculer le salaire moyen de chaque catégorie :

```
# les clés sont les catégories de poste, les valeurs sont 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.iteritems()
}
```

ce qui 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. »

Mais nous avons choisi les catégories de manière totalement arbitraire. Nous préférerions 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 les 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 les années d'expérience et les comptes payants :

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6   unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10  paid
```

Les utilisateurs qui ont très peu ou beaucoup d'années d'expérience ont tendance à payer, alors que les utilisateurs avec une durée d'expérience moyenne ne paient pas.

Donc, 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 » et 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 maths), 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 savoir quels sujets intéressent le plus les utilisateurs, afin de

pouvoir 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 serait de compter les mots :

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

Soit sous forme de code :

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

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

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

Ce qui vous donne le résultat prévu (à moins que vous ne vous attendiez à séparer en deux mots « scikit-learn », auquel cas vous n'obtiendrez pas le résultat attendu) :

```
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 20.

À suivre

Ce premier jour s'est plutôt bien passé ! 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* la journée d'intégration des nouveaux employés. 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 les plus importantes pour ce qui nous concerne (dont certaines sont rarement mises en avant dans les tutoriels Python).

Les fondamentaux

Installer Python

Vous pouvez télécharger Python à partir de `python.org`. Mais si vous ne l'avez pas encore, je vous recommande plutôt d'installer la distribution Anaconda, car elle inclut la plupart des bibliothèques nécessaires en data science.

À l'heure où j'écris ce livre, la dernière version de Python est la 3.4. Toutefois, chez Data-Sciencester nous en sommes restés à la bonne vieille Python 2.7. Python 3 n'est pas compatible avec son prédécesseur Python 2 et de nombreuses bibliothèques importantes ne fonctionnent qu'avec la 2.7. La communauté data science est encore très attachée à la 2.7, ce qui veut dire que nous allons faire de même. Vérifiez que c'est bien cette version que vous installez.

Si vous n'installez pas Anaconda, vérifiez que vous avez installé pip, un gestionnaire de paquetage Python qui vous permet d'installer facilement des paquetages tiers (dont certains vous seront nécessaires). Il est préférable de télécharger aussi IPython, un shell Python nettement plus convivial que celui proposé par défaut.

(Si vous avez installé Anaconda, normalement pip et IPython sont inclus.) Il suffit de lancer :

```
pip install ipython
```

et de chercher sur la Toile des solutions aux messages d'erreurs plus ou moins sibyllins que vous ne manquerez pas d'obtenir en retour.

Les principes Zen de Python

Python contient une description plutôt zen des principes de sa conception que vous pouvez obtenir à partir de l'interpréteur Python en tapant `import this`.

Un des principes qui ont fait couler le plus d'encre est celui-ci : il devrait y avoir une – et de préférence une seule – manière évidente de coder.

Le code écrit dans le respect de la 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 Python, nous ne manquerons pas de

comparer de temps en temps des solutions pythoniques et non pythoniques permettant de réaliser la même chose. En général, nous préférerons les solutions pythoniques.

La mise en page par les espaces

De nombreux langages utilisent les parenthèses pour délimiter des blocs de code. Python utilise l'indentation :

```
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. 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)
```

et pour faciliter 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 c'est rare :

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

Une des conséquences de l'utilisation du caractère d'espacement pour la mise en page est la difficulté d'effectuer des copier-coller de code dans le shell Python. Par exemple, si vous essayez de copier, dans le shell Python ordinaire, le code :

```
for i in [1, 2, 3, 4, 5]:

```

```
    # remarquez la ligne blanche
    print i
```

vous recevrez ce message :

```
IndentationError: expected an indented block
```

car, pour l'interpréteur, la ligne blanche signale la fin de la boucle `for`.

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

Les modules

Certaines fonctionnalités de Python ne sont pas chargées par défaut. C'est le cas de toutes les fonctionnalités incluses à la fois dans le langage et comme fonctionnalités tierces à télécharger à part. Pour utiliser ces dernières, vous devez importer les modules correspondants.

Une méthode consiste à importer le module lui-même :

```
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'import, vous devrez préfixer ces fonctions présentes dans le module par `re.` pour y accéder.

Si vous avez déjà un `re` différent dans votre code il faudra utiliser un alias :

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

Vous pouvez aussi choisir 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
```

Si vous avez 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'ensemble des contenus d'un module dans votre espace de nom, mais cela risquerait d'écraser par inadvertance des variables que vous aviez déjà définies :

```
match = 10
from re import *  # oh oh, re a une fonction match
print match       # "<fonction re.match>"
```

Cependant, comme ce n'est pas votre cas, vous ne ferez jamais une telle chose.

L'arithmétique

Par défaut, Python 2.7 utilise la division entière, c'est-à-dire que 5 / 2 égale 2. Ce n'est presque jamais ce que vous voulez, donc nous commencerons toujours nos fichiers par :

```
from __future__ import division
```

et alors 5 / 2 égale 2,5. Tous les exemples de ce livre utilisent ce nouveau style de division. Dans les rares cas où nous aurons besoin d'une division entière, il suffira d'utiliser une double barre oblique 5 // 2.

Les fonctions

Une fonction est une règle qui est appliquée à des entrées pour retourner un résultat en sortie. En Python, nous définissons les fonctions par `def` :

```
def double(x):
  """c'est ici que vous placez un docstring facultatif pour expliquer ce que fait cette
  function. Par exemple, cette function 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 assigner 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 précédement définie
x = apply_to_one(my_double)    # égale 2
```

On peut aussi facilement créer de petites fonctions anonymes dites

« lambdas » :

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

Vous pouvez assigner des lambdas à des variables, même si la plupart des développeurs vous conseilleront d'utiliser `def` à la place :

```
another_double = lambda x: 2 * x     # à ne pas faire
def another_double(x): return 2 * x  # à faire à la place
```

Les arguments d'une fonction peuvent aussi recevoir des valeurs par défaut, vous spécifiez alors explicitement les arguments uniquement quand vous voulez leur donner une valeur différente de celle par défaut :

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


my_print("hello")  # imprime 'hello'
my_print()         # imprime 'my default message'(mes messages par défaut)
```

Et il est parfois utile de spécifier des arguments par leur nom :

```
def subtract(a=0, b=0):
  return a - b
subtract(10, 5) # retourne 5
subtract(0, 5)  # retourne -5
subtract(b=5)   # même chose que précédemment
```

Nous allons créer beaucoup de fonctions.

Les chaînes de caractères

Les chaînes de caractères *(strings)* peuvent être délimitées par des guillemets simples ou doubles (mais ils 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 comme 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 triples [doubles] guillemets :

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

Les exceptions

En cas d'erreur, Python déclenche une exception. Si l'exception n'est pas traitée, elle va provoquer 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 vues 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.

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 $n^{ème}$ élément de la liste au moyen de crochets :

```
x = range(10)  # est la liste [0, 1, ..., 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 pour « découper » des listes :

```
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]
```

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, ce qui veut dire que vous ne devriez pas l'employer sauf 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 :

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

Et 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 d'une liste un par un :

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

Et souvent, il est pratique de déballer des listes (*unpack* en anglais) en assignant 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
```

À noter 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 == 2, peu importe le premier élément
```

Les tuples

Les tuples sont les cousins immutables 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)       # égale (5, 6)
s, p = sum_and_product(5, 10)    # s vaut 15, p is 50
```

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

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

Les dictionnaires

Le dictionnaire est une autre structure de données fondamentale qui 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
```

Mais vous déclencherez une erreur `KeyError` si vous demandez une clé qui n'est 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 `in` :

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

Les dictionnaires possèdent une méthode `get` qui retourne une valeur par défaut (au lieu de déclencher une exception) quand vous cherchez une clé qui n'est pas dans le dictionnaire :

```
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 assigne des paires clé-valeur *(key-value)* 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
```

On utilisera souvent les dictionnaires pour représenter de manière simple des données structurées :

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

En plus de pouvoir chercher une clé spécifique, on peut les balayer toutes :

```
tweet_keys   = tweet.keys()     # liste de clés
tweet_values = tweet.values()   # liste de valeurs
tweet_items  = tweet.items()    # liste de tuples(clé-valeur)
```

```
"user" in tweet_keys            # Vrai mais lent (in dans une liste)
"user" in tweet    # plus pythonique, utilise la version dict du in
"joelgrus" in tweet_values   # Vrai
```

Les clés de dictionnaires doivent être immutables ; en particulier, on ne peut pas 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.

Les 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 le nombre d'occurrences de ces mots. Comme vous passez sur chaque mot, vous pouvez incrémenter le compteur s'il est déjà dans le dictionnaire et l'ajouter au dictionnaire sinon :

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

Vous pouvez aussi utiliser la tactique « mieux vaut s'excuser après coup plutôt que demander la permission avant » 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
```

Chacune de ces méthodes est assez peu pratique, ce qui explique pourquoi defaultdict existe. Un defaultdict est comme un dictionnaire normal à la différence que, quand vous cherchez une valeur 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 initialisés avec `list` ou `dict` ou même 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                    # maintenant dd_pair contient {2: [0,1]}
```

Cela sera très utile quand vous utiliserez des dictionnaires afin de « collecter » des résultats selon une certaine clé et lorsque vous ne voulez pas 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 compteurs. Nous l'utiliserons avant tout pour créer des histogrammes :

```
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 :

```
word_counts = Counter(document)
```

Une instance de compteur possède une méthode `most_common` bien pratique :

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

Les ensembles

L’ensemble *(set)* est encore une autre structure de données qui permet de représenter une collection d’éléments distincts :

```
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  # égale Vrai
z = 3 in s  # égale Faux
```

Nous utiliserons des ensembles principalement pour deux raisons. La première est que le `in` est une opération très rapide sur les ensembles. Si nous avons une grande collection d’éléments que nous voulons utiliser pour faire un test d’appartenance, un ensemble est mieux adapté qu’une liste :

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list  # Faux, mais il faut tester chaque élément

stopwords_set = set(stopwords_list)
"zip" in stopwords_set   # très rapide à vérifier
```

La seconde raison est qu’on trouve 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 dict et les listes.

Les structures de contrôle

Comme avec la plupart des langages de programmation, il est possible d’utiliser des branchements conditionnels via le mot-clé `if` :

```
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 ligne, ce que nous ferons de temps en temps :

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

Python possède une boucle tant que (`While`) :

```
x = 0
while x < 10:
  print x, "is less than 10"
  x += 1
```

mais nous utiliserons plus souvent `for` et `in` :

```
for x in range(10):
  print 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
```

Ce code va afficher `0`, `1`, `2` et `4`.

Vrai/faux

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

```
one_is_less_than_two = 1 < 2       # égale vrai
true_equals_false = True == False # égale 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
print x == None  # imprime True, mais n'est pas pythonique
print x is None  # imprime True, et est pythonique
```

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

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

Presque tout le reste est considéré comme `True` (vrai). Il est donc facile d'utiliser des instructions conditionnelles avec `if` pour tester si des listes, des chaînes de caractères ou des dictionnaires sont vides. Cela peut aussi parfois induire des bugs difficiles à résoudre si vous n'avez pas ce comportement bien 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 de faire la même chose :

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

car `and` retourne sa deuxième valeur quand la première est « vraie », 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.

Python possède une fonction `all` qui prend en entrée une liste 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 }])  # Vrai
all([True, 1, {}])     # Faux, {} est plutôt faux
any([True, 1, {}])     # Vrai, True est plutôt vrai
all([])                # Vrai, pas d'éléments plutôt faux dans la liste
any([])                # Faux, pas d'éléments plutôt vrais dans la liste
```

Quelques fonctionnalités avancées de Python

Les fonctionnalités plus avancées de Python que nous allons examiner maintenant nous seront utiles pour manipuler des données.

Trier

Toute liste Python possède une méthode de tri (`sort`) pour la trier sur place. Si vous ne voulez pas modifier votre liste, vous pouvez utiliser la fonction `sorted`, qui retourne une nouvelle liste :

```
x = [4,1,2,3]
y = sorted(x)  # vaut [1,2,3,4], x est inchangé
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 la clé :

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

Les list comprehensions

Il est fréquent de vouloir transformer une liste en une autre liste en sélectionnant certains éléments, ou en transformant des éléments, ou les deux. La manière pythonique de faire est d'utiliser une *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, des listes peuvent être converties en dictionnaires ou en sets :

```
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 les éléments de la liste, il est d'usage d'utiliser un tiret bas :

```
zeroes = [0 for _ in even_numbers]  # a la même longueur que even_numbers
```

Une list comprehension peut contenir plusieurs boucles `for` :

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

et d'autres boucles `for` peuvent réutiliser les résultats des précédentes :

```
increasing_pairs = [(x, y)                       # uniquement les paires avec x < y,
                    for x in range(10)           # plage de valeurs (lo, hi) égale
                    for y in range(x + 1, 10)]   # [lo, lo + 1, ..., hi - 1]
```

Nous ferons un usage immodéré des list comprehensions.

Générateurs et itérateurs

Le problème avec les listes, c'est qu'elles peuvent devenir gigantesques, `range(1000000)` par exemple crée une liste d'un million d'éléments. Si vous devez les traiter un par un, ce sera très inefficace (ou cela provoquera un dépassement de mémoire). Si vous n'avez besoin que des premières valeurs, les calculer toutes relève d'un énorme gaspillage.

Un générateur permet de réaliser des itérations (pour nous, surtout des boucles `for`) dont les valeurs ne sont produites qu'à la demande (on parle alors d'évaluation paresseuse, en opposition à l'évaluation immédiate).

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

```
def lazy_range(n):
  """une version paresseuse de range"""
  i = 0
  while i < n:
      yield i
      i += 1
```

La boucle suivante consomme les valeurs de `yield` une par une jusqu'à épuisement :

```
for i in lazy_range(10):
  do_something_with(i)
```

(En fait, Python est fourni avec une fonction implémentant `lazy_range` appelée `xrange`, et en Python 3 `range` est elle-même paresseuse.) Cela veut dire que 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.

Astuce

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 quelque chose, il faudra recréer le générateur à chaque fois ou faire appel à une liste.

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

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i % 2 == 0)
```

N'oubliez pas que chaque dict possède une méthode `items()` qui retourne la liste de ses paires clé-valeur. Le plus souvent, nous utiliserons la méthode `iteritems()` qui renvoie de manière paresseuse les paires clé-valeurs une par une quand nous exécutons la boucle.

Les valeurs aléatoires

Au cours de notre apprentissage de la science des données, nous aurons souvent besoin de générer des nombres aléatoires, ce que nous pouvons réaliser avec le module `random` :

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

# [0.8444218515250481,  # random.random() produit des nombres
#  0.7579544029403025,   # uniformément distribués entre 0 et 1
#  0.420571580830845,    # c'est la fonction aléatoire que nous
#  0.25891675029296335]  # utiliserons le plus souvent
```

Le module `random` produit en fait des nombres pseudo-aléatoires (c'est-à-dire déterministes) basés sur 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)        # remet la valeur de départ à 10
print random.random()  # 0.57140259469 encore une fois
```

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

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

Il existe quelques autres méthodes bien pratiques parfois. `random.shuffle` réorganise au hasard les éléments d'une liste :

```
up_to_ten = range(10)
random.shuffle(up_to_ten)
print up_to_ten
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0]  (vos résultats seront sans doute différents)
```

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

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

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

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

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

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

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

print all([                                 # tous sont vrais car
  not re.match("a", "cat"),                 # * * 'cat' ne commence pas par 'a'
  re.search("a", "cat"),                    # * 'cat' possède 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
  ]) # affiche True
```

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 est de construire un exemple commenté.

Imaginons que vous ne disposez pas de set dans Python. Nous voudrions alors créer notre propre classe `Set`.

Quel devrait être le comportement de notre classe ? Soit une instance donnée de `Set` ; nous devons être capables d'y ajouter des éléments, de retirer des éléments et de vérifier s'il contient une certaine valeur. Pour cela, nous allons créer des fonctions membres, ce qui veut dire que nous y accéderons en utilisant une instance d'objet `Set` suivie d'un point :

```
# par convention, on attribue aux classes de noms en PascalCase
class Set:
  # voici les fonctions membres
  # chacune admet un paramètre "self" (une autre convention)
  # qui fait référence à l'objet Set particulier en cours d'utilisation
  def __init__(self, values=None):
    """Ceci est le constructeur.
    Il est appelé quand vous créez un nouveau set.
    Voilà comment l'utiliser"""

    s1 = Set()           # set vide
    s2 = Set([1,2,2,3])  # initialise avec les valeurs
    self.dict = {}       # chaque instance de set a sa propre propriété  dict
                         # qui est celle utilisée pour suivre l'appartenance
    if values is not None:
      for value in values:
        self.add(value)

  def __repr__(self):
    """ceci est la représentation d'un objet set comme chaîne de caractères
    si vous la tapez dans le prompt Python ou si vous la passez à str()"""
    return "Set: " + str(self.dict.keys())
```

```
  # nous représenterons l'appartenance comme une clé dans self.dict avec la valeur True
  def add(self, value):
    self.dict[value] = True

  # la valeur est dans le Set si c'est une clé du dictionnaire
  def contains(self, value):
    return value in self.dict

  def remove(self, value):
    del self.dict[value]
```

Que nous pourrons utiliser comme suit :

```
s = Set([1,2,3])
s.add(4)
print s.contains(4)  # Vrai
s.remove(3)
print s.contains(3)  # Faux
```

Les outils fonctionnels

Quand vous faites appel à des fonctions, vous voulez parfois les appliquer partiellement *(curryfication)* pour créer de nouvelles fonctions. Par exemple, imaginons une fonction à deux variables :

```
def exp(base, power):
  return base ** power
```

Nous voulons créer à partir de là une fonction à une variable `two_to_the` dont l'entrée est une puissance et dont la sortie est le résultat de `exp(2, power)`.

Évidemment, il est toujours possible d'utiliser `def`, mais c'est parfois peu pratique :

```
def two_to_the(power):
  return exp(2, power)
```

Une méthode différente consiste à utiliser `functools.partial` :

```
from functools import partial
two_to_the = partial(exp, 2)   # est maintenant une fonction à une variable
print two_to_the(3)            # 8
```

Vous pouvez aussi utiliser `partial` pour ajouter des arguments ultérieurs si vous spécifiez leurs noms :

```
square_of = partial(exp, power=2)
print square_of(3)             # 9
```

L'affaire se complique si vous curryfiez les arguments au milieu de la fonction, donc nous éviterons de jouer à ce jeu-là.

De temps en temps, nous utiliserons `map`, `reduce` et `filter`, qui sont autant d'alternatives fonctionnelles aux list comprehensions.

```
def double(x):
  return 2 * x


xs = [1, 2, 3, 4]
twice_xs = [double(x) for x in xs]   # [2, 4, 6, 8]
twice_xs = map(double, xs)           # comme ci-dessus
list_doubler = partial(map, double)  # *function* qui double une liste
twice_xs = list_doubler(xs)          # de nouveau [2, 4, 6, 8]
```

Vous pouvez utiliser `map` avec des fonctions à plusieurs arguments si vous passez en paramètres plusieurs listes :

```
def multiply(x, y): return x * y


products = map(multiply, [1, 2], [4, 5])  # [1 * 4, 2 * 5] = [4, 10]
```

De même, `filter` effectue le travail de la clause `if` d'une list comprehension :

```
def is_even(x):
  """Vrai si x est pair, Faux si x est impair"""
  return x % 2 == 0
x_evens = [x for x in xs if is_even(x)]  # [2, 4]
x_evens = filter(is_even, xs)            # comme ci-dessus
list_evener = partial(filter, is_even)   # *function* qui filtre une liste
x_evens = list_evener(xs)                # de nouveau [2, 4]
```

Enfin, `reduce` combine les deux premiers éléments d'une liste, puis ce résultat avec le troisième, le résultat avec le quatrième, et ainsi de suite pour produire un résultat unique :

```
x_product = reduce(multiply, xs)          # = 1 * 2 * 3 * 4 = 24
list_product = partial(reduce, multiply)  # **function* qui réduit une liste
x_product = list_product(xs)              # de nouveau = 24
```

enumerate

Assez souvent, vous voudrez exécuter une itération sur une liste et utiliser à la fois ses éléments et leur index :

```
# non pythonique
for i in range(len(documents)):
  document = documents[i]
  do_something(i, document)
```

```
# non pythonique (bis)
i = 0
for document in documents:
  do_something(i, document)
  i += 1
```

La solution pythonique est `enumerate`, qui produit des tuples (index, élément) :

```
for i, document in enumerate(documents):
  do_something(i, document)
```

De même, si nous voulons seulement les index :

```
for i in range(len(documents)): do_something(i)    # non pythonique
for i, _ in enumerate(documents): do_something(i)  # pythonique
```

Nous adopterons très souvent cette solution.

zip et le déballage d'arguments

Nous aurons souvent besoin de réunir une ou plusieurs listes. `zip` transforme plusieurs listes en une liste unique de tuples des éléments correspondants :

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)  # vaut [('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, ce qui utilise les éléments des paires comme des arguments séparés à réunir. La fin est la même que si vous aviez appelé :

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

qui renvoie `[('a','b','c'), ('1','2','3')]`.

Vous pouvez utiliser le déballage d'arguments avec n'importe quelle fonction :

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

add(1, 2)     # retourne 3
add([1, 2])   # TypeError!
add(*[1, 2])  # retourne 3
```

Il est rare que ce soit avantageux, mais c'est malgré tout une astuce intéressante.

args et kwargs

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

```
def doubler(f):
  def g(x):
    return 2 * f(x)
  return g
```

Parfois ça marche :

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

g = doubler(f1)
print g(3)   # 8 (== ( 3 + 1) * 2)
print g(-1)  # 0 (== (-1 + 1) * 2)
```

Cependant, la méthode s'arrête de fonctionner avec les fonctions qui acceptent plus d'un argument :

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

g = doubler(f2)
print g(1, 2)  # TypeError : g() accepte exactement 1 argument (2 ont été fournis)
```

Nous avons besoin d'un moyen de spécifier une fonction qui accepte des arguments arbitraires. Et, cela est possible avec le déballage d'arguments et un peu de magie :

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

magic(1, 2, key="word", key2="word2")

# affiche
# args sans noms : (1, 2)
# args mots-clés : {'key2': 'word2', 'key': 'word'}
```

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 dict pour apporter 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 }
print other_way_magic(*x_y_list, **z_dict)  # 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):
  """va bien quelle que soit la nature des entrées attendues par f"""
  def g(*args, **kwargs):
    """quels que soient les arguments fournis à g, les passe par l'intermédiaire de f"""
    return 2 * f(*args, **kwargs)
  return g

g = doubler_correct(f2)
print g(1, 2)  # 6
```

Bienvenue chez DataSciencester !

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

Pour aller plus loin

- Il existe de nombreux tutoriels Python dans le monde. Celui du site officiel n'est pas une mauvaise idée pour démarrer.
- Le tutoriel officiel IPython est loin d'être aussi bon. Vous en apprendrez davantage en suivant leurs vidéos et leurs présentations. Sinon, sachez que *L'analyse de données en Python*, de Wes McKinney (chez Eyrolles), contient un très bon chapitre sur IPython.

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 consacrée à la visualisation des données. Bien qu'il soit très facile de créer des représentations visuelles, il est beaucoup plus difficile d'en créer de bonnes.

La visualisation des données répond à deux attentes principales :

- explorer les données ;
- communiquer les 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 sujets, 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 marche pas.

matplotlib

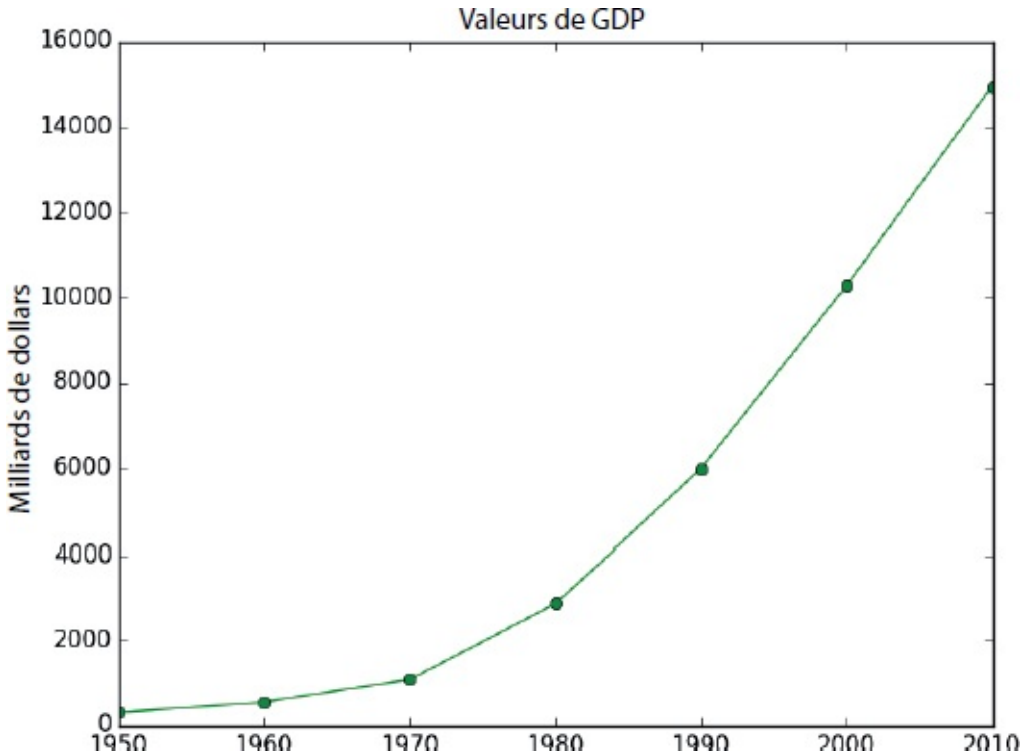
Il existe une grande variété d'outils de visualisation. Nous utiliserons la bibliothèque matplotlib, qui est largement répandue (même si elle commence à accuser son âge). Ce n'est pas forcément le bon choix si votre objectif est de produire des représentations interactives sophistiquées pour le web, mais pour de simples diagrammes en bâtons, des courbes et des nuages de points, elle est tout à fait suffisante.

En particulier, nous ferons appel au 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 fini, 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, gdp sur l'axe des y
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# ajoute un titre
plt.title("Valeurs de GPD")

# ajoute un label 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 labels sur les axes, des styles de ligne et des marqueurs de points. Plutôt que d'essayer de traiter le sujet de manière exhaustive, nous allons nous concentrer sur quelques options seulement à travers plusieurs exemples.

Note

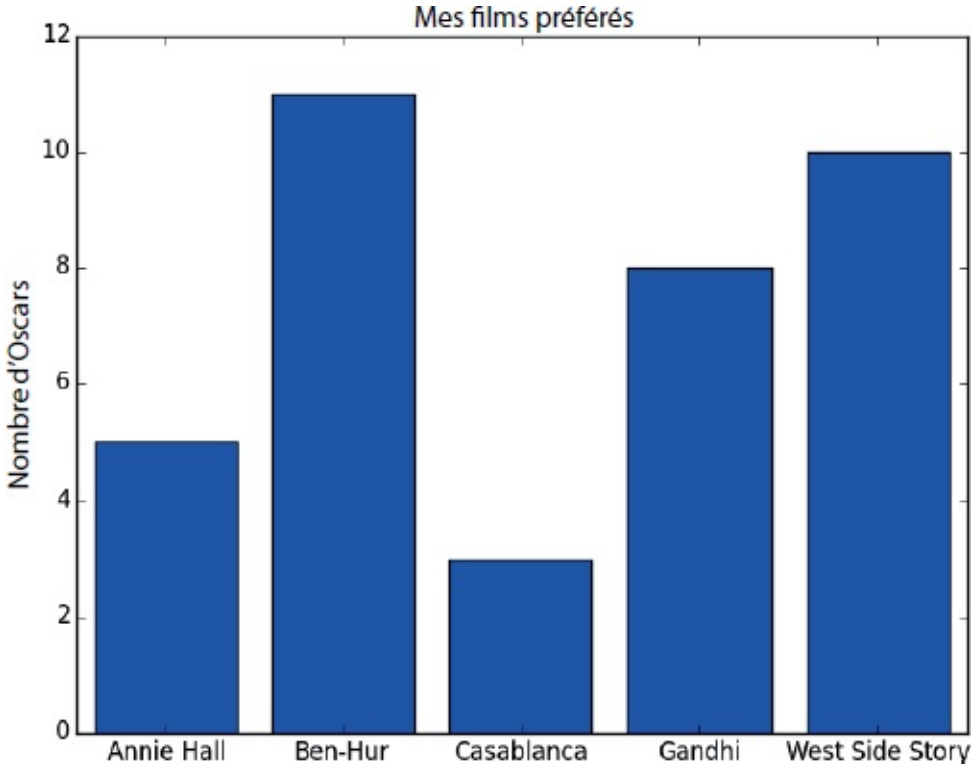
Même si ce n'est pas le style que nous utiliserons le plus, matplotlib peut fournir des diagrammes complexes à l'intérieur des diagrammes, des mises en page très élaborées et des représentations interactives. Pour approfondir le sujet, vous pouvez vous reportez à la documentation de matplotlib.

Les diagrammes en bâtons

Un diagramme en bâtons est 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 attribués à une sélection de différents films :

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"] num_oscars
= [5, 11, 3, 8, 10]

# par défaut les bâtons sont larges de 0,8 donc nous ajouterons 0,1 aux coordonnées
# à gauche pour centrer chaque bâton
xs = [i + 0.1 for i, _ in enumerate(movies)]

# dessine des bâtons avec les coordonnées de gauche sur l'axe x [xs],
# hauteurs [num_oscars]
plt.bar(xs, num_oscars)

plt.ylabel("Nombre d'Oscars")
plt.title("Mes films préférés")

# labellise l'axe des x avec le nom des films centrés sur le bâton
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

Figure 3–2
Un diagramme simple en bâtons

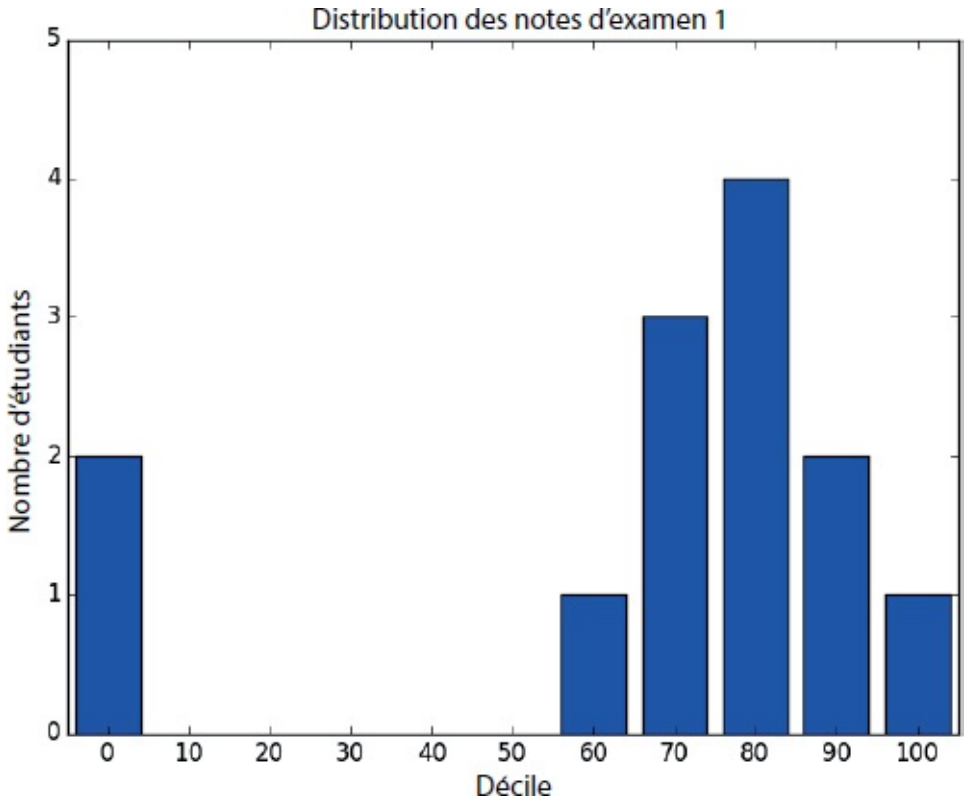


Le diagramme en bâtons est aussi un bon choix pour représenter des histogrammes de valeurs numériques regroupées par catégories afin d'explorer la distribution des valeurs, comme sur la figure 3-3 :

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # décale chaque bâton à gauche de la valeur 4
        histogram.values(),                # donne à chaque bâton sa hauteur correcte
        8)                                 # donne à chaque bâton une largeur de 8

plt.axis([-5, 105, 0, 5])                  # x-axis from -5 to 105,
                                           # axe des y de 0 à 5

plt.xticks([10 * i for i in range(11)])    # labels sur l'axe des x à 0, 10, ..., 100
plt.xlabel("Décile")
plt.ylabel("Nombre d'étudiants")
plt.title("Distribution des notes d'examen 1")
plt.show()
```

Le troisième argument de `plt.bar` spécifie la largeur des bâtons. Ici, nous avons choisi une largeur de 8 (ce qui laisse un petit espace entre les bâtons, car nos catégories ont une largeur de 10). Et nous avons fait glisser le bâton à gauche de 4, de sorte que le bâton « 80 » soit centré sur 80, ses côtés gauche et droit étant à 76 et 84 respectivement.

Figure 3–3
Un diagramme en bâtons utilisé pour un histogramme



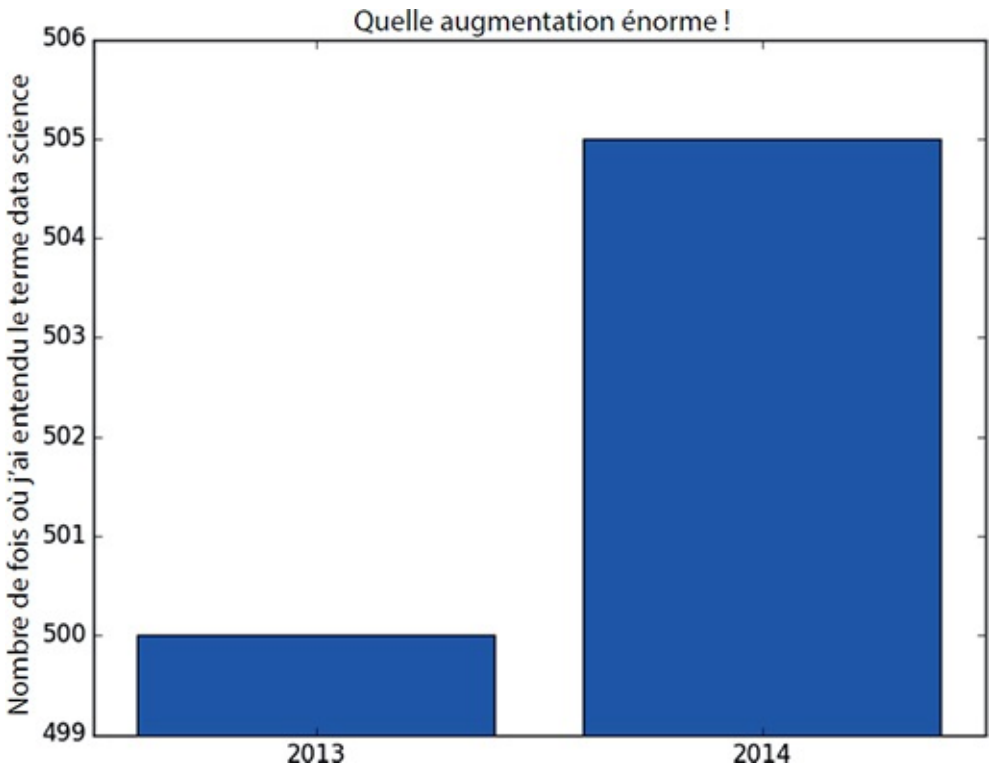
L'appel de `plt.axis` indique que nous demandons une plage de valeurs de -5 à 105 pour les x (de sorte que les bâtons « 0 » et « 100 » soient complets) et que l'axe des y soit gradué de 0 à 5. L'appel à `plt.xticks` place les labels de l'axe des x à 0, 10, 20… 100.

Utilisez judicieusement `plt.axis()`. Quand vous créez un diagramme en bâtons, il n'est pas recommandé 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 = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("Nombre de fois où j'ai entendu le terme 'data science'")

# sinon, matplotlib va labelliser l'axe des x  0, 1

# et ajouter un +2.013e3 dans le coin (vilain matplotlib !)
plt.ticklabel_format(useOffset=False)

# l'axe des y trompeur ne montre que la partie au-dessus de 500
plt.axis([2012.5,2014.5,499,506])
```

```
plt.title("Quelle augmentation énorme !")
plt.show()
```

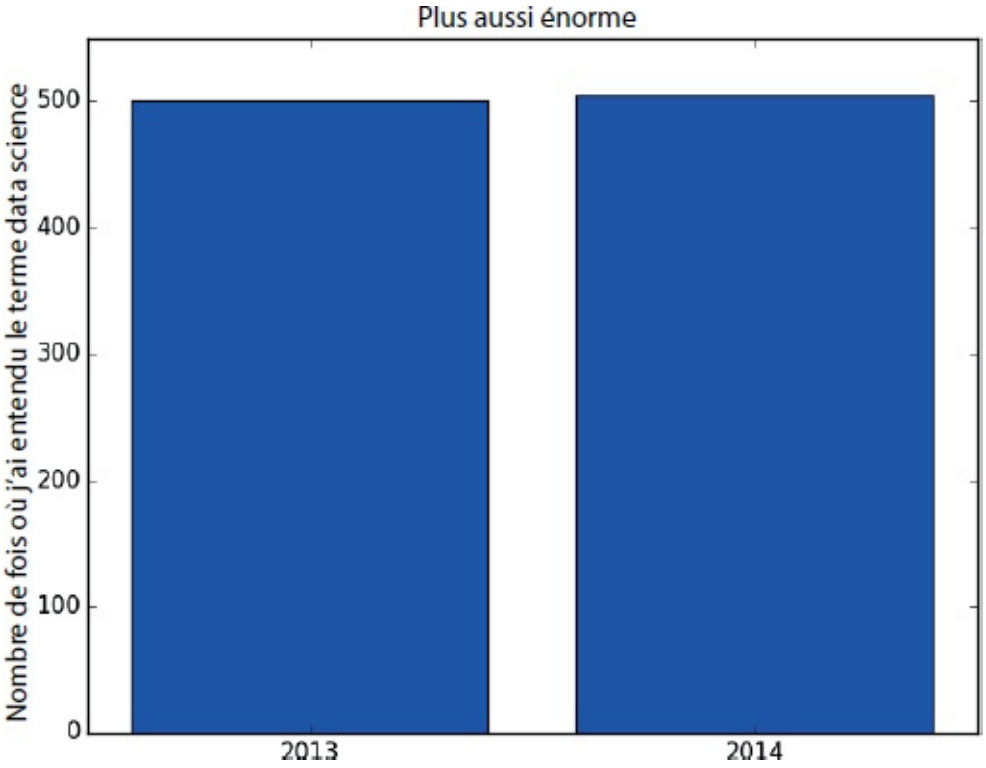
Figure 3–4
Un diagramme avec un axe des y trompeur



La figure 3-5 rétablit des axes plus sérieux et l'impression produite n'est plus aussi forte :

```
plt.axis([2012.5,2014.5,0,550])
plt.title("Plus si énorme")
plt.show()
```

Figure 3–5
Le même diagramme avec un axe des y plus honnête

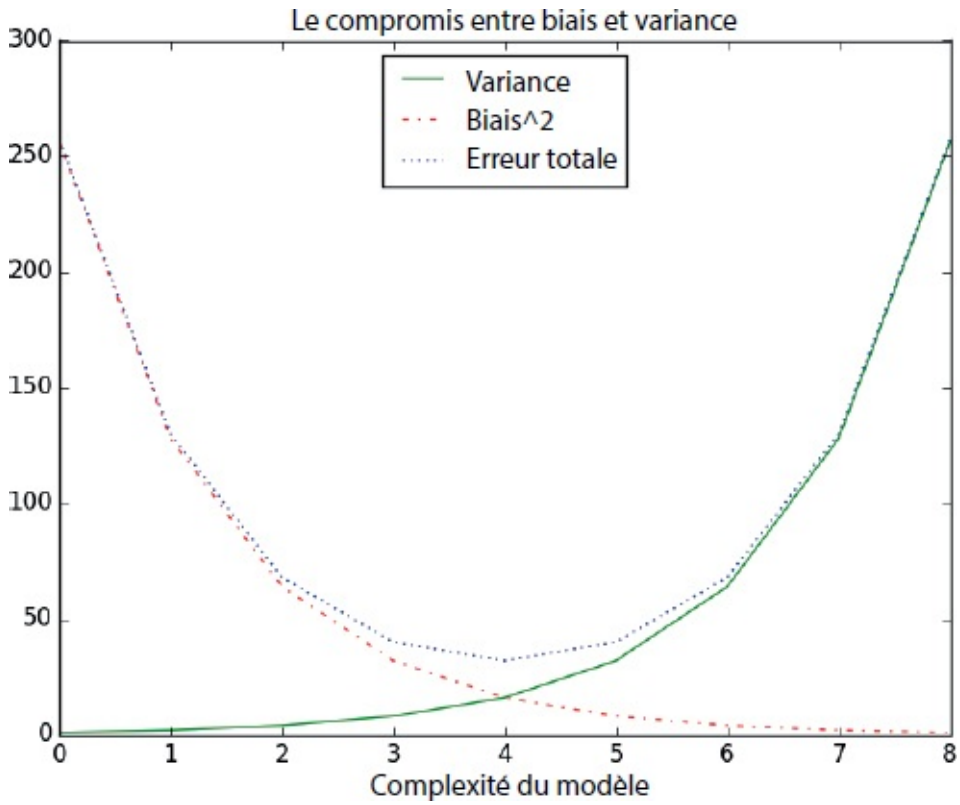

Plus aussi énorme
Nombre de fois où j'ai entendu le terme data science
0
100
200
300
400
500
2013
2014

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, comme sur la 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 montrer plusieurs séries sur le même graphique
plt.plot(xs, variance, 'g-', label='variance')       # ligne verte continue
plt.plot(xs, bias_squared, 'r-.', label='bias^2')    # ligne rouge pointillée avec tirets
plt.plot(xs, total_error, 'b:', label='total error') # ligne bleue pointillée

# parce que nous avons assigné des labels pour chaque série
# nous avons une légende gratuite
# loc=9 veut dire "top center" (centré en haut)
plt.legend(loc=9)
plt.xlabel("Complexité du modèle")
plt.title("Le compromis entre biais et variance")
plt.show()
```

Figure 3–6
Plusieurs courbes avec une légende

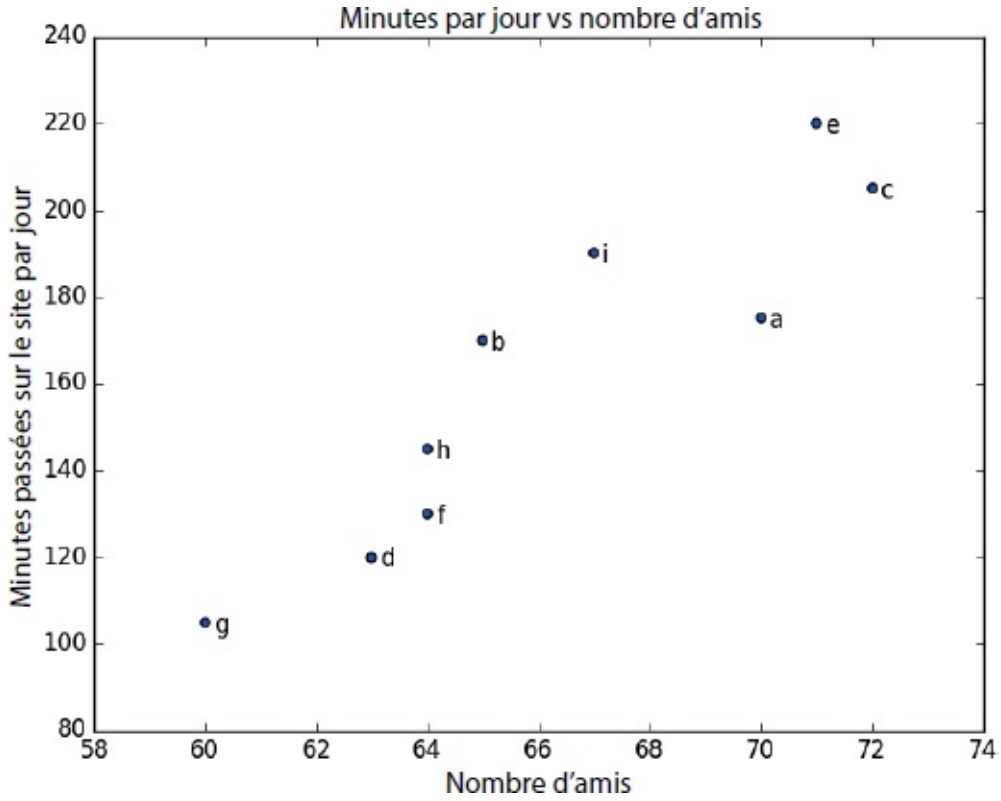


Les nuages de points

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 de vos utilisateurs et le nombre de minutes qu'ils 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)

# labellise chaque point
for label, friend_count, minute_count in zip(labels, friends, minutes):
  plt.annotate(label,
    xy=(friend_count, minute_count),  # place le label près du point
    xytext=(5, -5),                   # mais légèrement décalé
    textcoords='offset points')

plt.title("Minutes par jour vs nombre d'amis")

plt.xlabel("Nombre d'amis")
plt.ylabel("Minutes passées sur le site par jour")
plt.show()
```

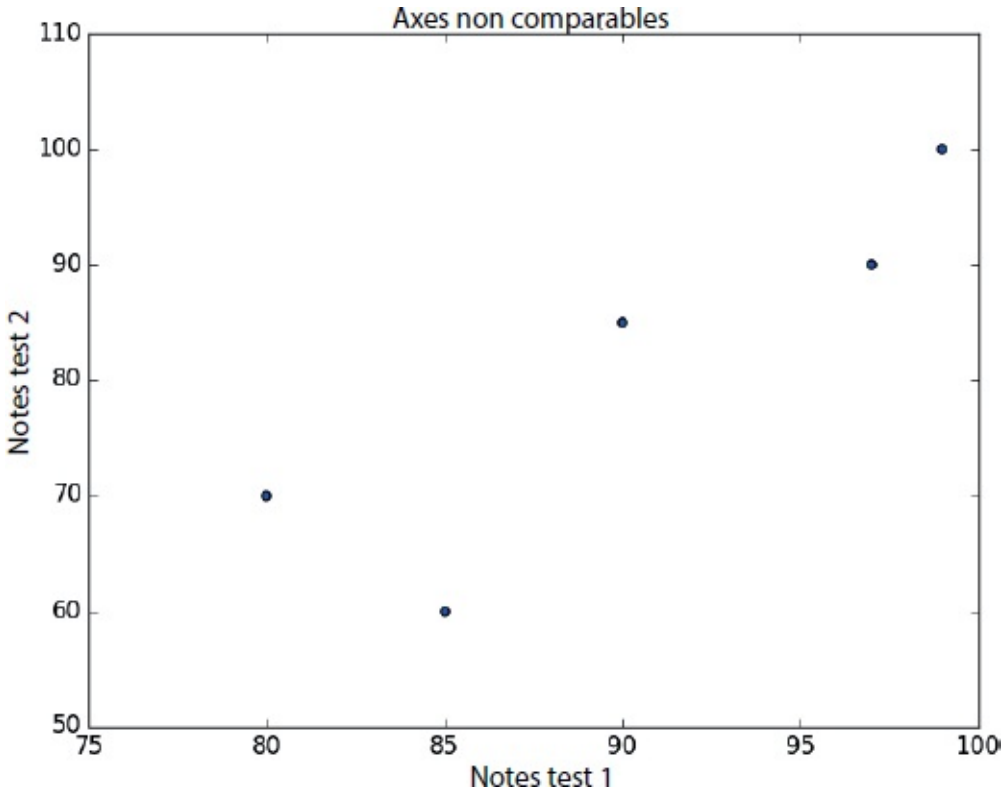
Figure 3–7
Un nuage de points pour représenter le nombre d'amis comparé au temps passé sur le site



Si vous représentez des variables comparables, vous risquez d'obtenir une image trompeuse si vous laissez matplotlib choisir l'échelle, comme sur la 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("Axes non comparables")
plt.xlabel("Notes test 1")
plt.ylabel("Notes test 2")
plt.show()
```

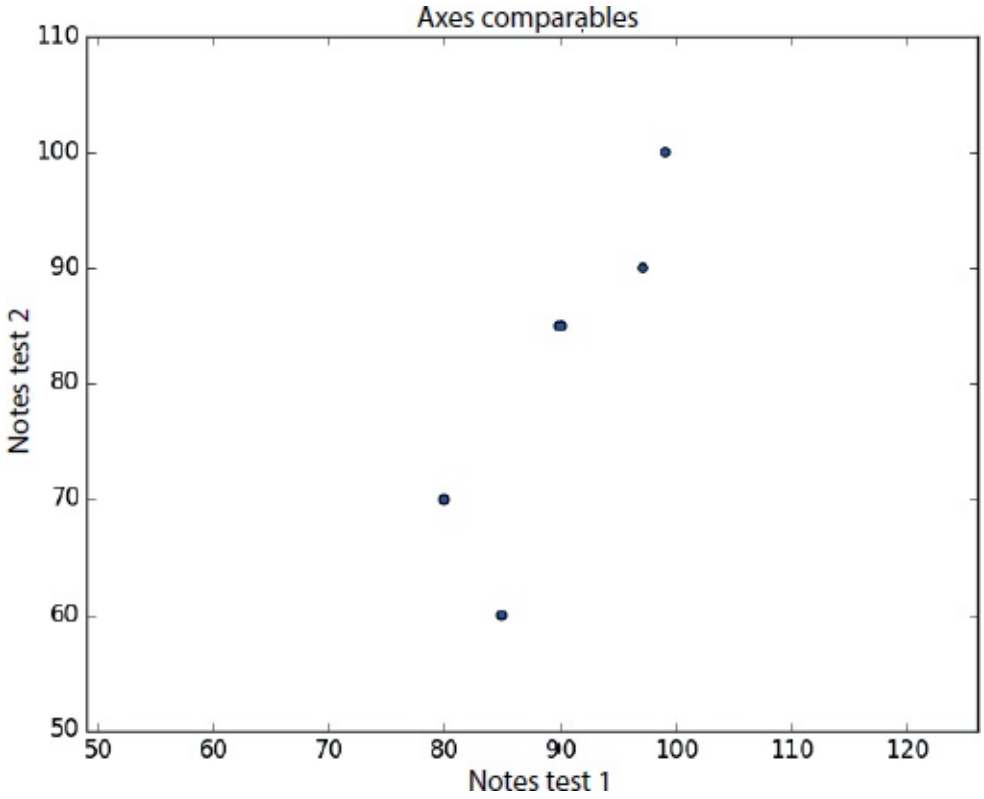
Figure 3–8
Un nuage de points avec des axes non comparables



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

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

Figure 3–9
Le même nuage de points avec des axes comparables


Axes comparables
110
100
90
80
70
60
50
Notes test 2
50
60
70
80
90
100
110
120
Notes test 1

Pour aller plus loin

- Seaborn s’ajoute à matplotlib et vous permet de réaliser des représentations plus jolies (et plus complexes).
- D3.js est par ailleurs une bibliothèque JavaScript dédiée aux représentations interactives pour le Web. Bien qu’elle ne fasse pas partie de Python, elle est à la fois à la mode et très utilisée. Elle mérite vraiment que vous preniez le temps de vous familiariser avec elle.
- Bokeh, quant à elle, est une nouvelle bibliothèque qui apporte des effets 3D à Python.
- Enfin, ggplot est une version Python de la célèbre bibliothèque R ggplot2, largement utilisée pour créer des diagrammes et des graphiques en qualité « prêt à publier ». C’est sans doute passionnant si vous êtes déjà un utilisateur enthousiaste de ggplot2, mais probablement un peu mystérieux si tel n’est pas le cas.

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'ai 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 (pour former d'autres vecteurs) et qui peuvent être multipliés par des scalaires (autrement dit des nombres) pour former de nouveaux vecteurs.

Concrètement (pour nous), les vecteurs sont des points dans un espace fini. Même si vous n'avez pas l'habitude d'envisager vos données comme des vecteurs, ces derniers sont pourtant une bonne manière 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 et âge). Et si vous êtes enseignant et que vous organisez quatre examens, vous pouvez considérer les notes des élèves comme des vecteurs à quatre dimensions (`exam1`, `exam2`, `exam3`, `exam4`).

La méthode la plus simple consiste à représenter les vecteurs comme une liste de nombres. Une liste de trois nombres correspond à un vecteur dans un espace à trois dimensions et vice versa :

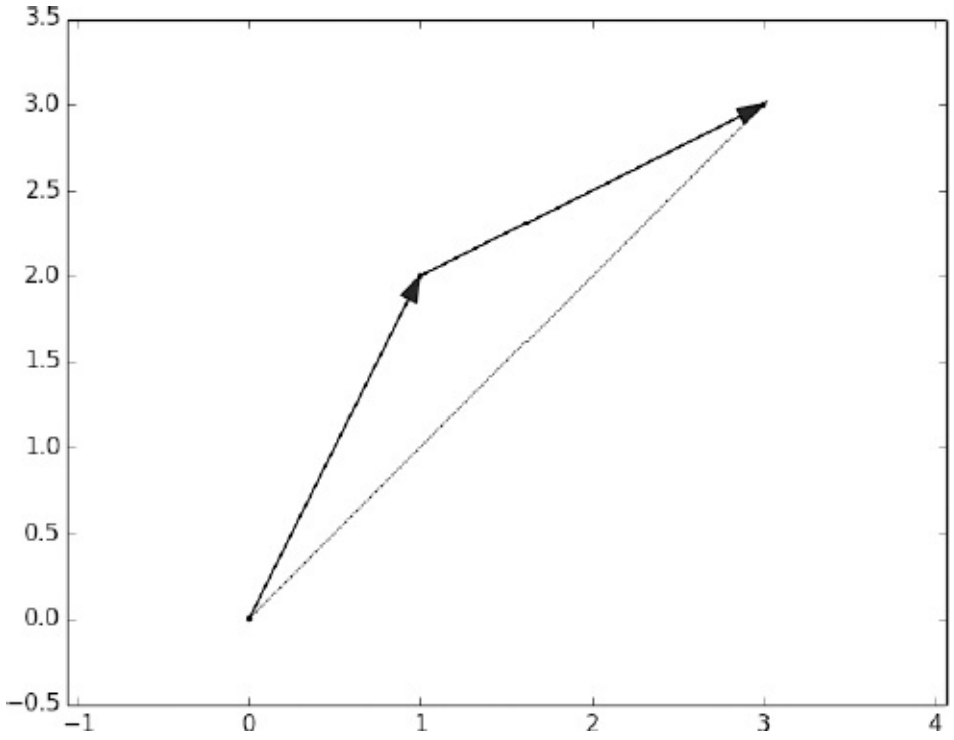
```
height_weight_age = [70, # pouces,
                     170, # livres,
                     40 ] # années

grades = [95,            # exam1
          80,            # exam2

          75,            # exam3
          62 ]           # exam4
```

Un des problèmes, avec cette méthode, est que 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 nous-mêmes nos outils arithmétiques. Alors en avant !

Pour commencer, nous aurons souvent besoin d'additionner deux vecteurs. Les vecteurs s'ajoutent en additionnant 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. (S'ils ne sont pas de même longueur, vous ne pouvez pas les ajouter.)

Par exemple, l'addition des vecteurs `[1, 2]` et `[2, 1]` donne comme résultat `[1 + 2, 2 + 1]`, soit `[3,3]`, comme le montre la figure 4-1.

Figure 4–1
Additionner deux vecteurs



Il est facile d'implémenter cette opération en zippant les vecteurs ensemble et en utilisant une list comprehension pour ajouter les éléments correspondants :

```
def vector_add(v, w):
  """ajoute les éléments correspondants"""
  return [v_i + w_i
          for v_i, w_i in zip(v, w)]
```

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

```
def vector_subtract(v, w):
  """soustrait les éléments correspondants"""
  return [v_i - w_i
          for v_i, w_i in zip(v, w)]
```

Quelquefois, nous voulons faire la somme élément par élément d'une liste de vecteurs. Il s'agit de créer un nouveau 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. Le plus facile consiste à ajouter un vecteur à la fois :

```
def vector_sum(vectors):
```

```
    """fait la somme de tous les éléments correspondants"""
    result = vectors[0]                    # commence avec le premier vecteur
    for vector in vectors[1:]:             # puis effectue une boucle sur les suivants
        result = vector_add(result, vector)  # et les ajoute au résultat
    return result
```

En y réfléchissant bien, nous sommes en train de faire un `reduce` de la liste de vecteurs à l'aide de `vector_add`, ce qui veut dire qu'il est possible de tout réécrire plus proprement avec des fonctions d'ordre supérieur :

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

ou même :

```
vector_sum = partial(reduce, vector_add)
```

Toutefois, cette dernière ligne est sans doute plus élégante que réellement utile.

Nous pouvons également avoir besoin de multiplier un vecteur par un scalaire, ce qui se fait simplement en multipliant chaque élément du vecteur par ce nombre :

```
def scalar_multiply(c, v):
    """c est un nombre, v est un vecteur"""
    return [c * v_i for v_i in v]
```

ce qui nous permet de calculer élément par élément la moyenne d'une liste de vecteurs (de même dimension) :

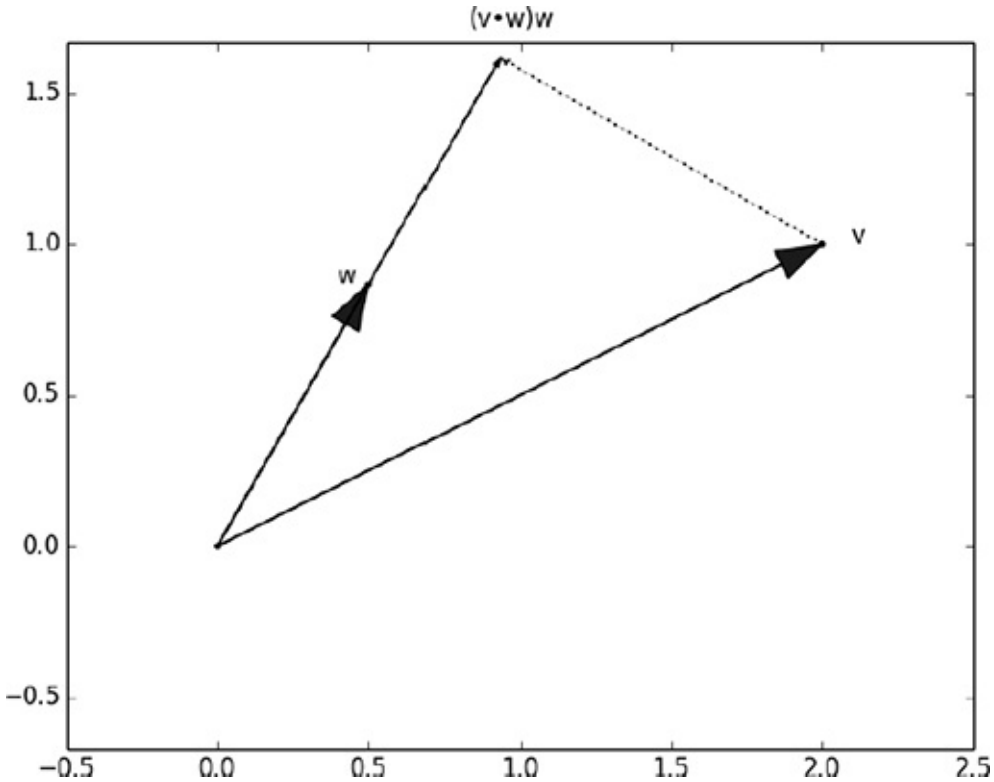
```
def vector_mean(vectors):
    """calcule le vecteur dont le i^ème élément est la moyenne des ièmes éléments
    des vecteurs d'entrée"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

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

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```

Si `v` a une magnitude de `1`, le produit scalaire mesure de combien le vecteur `v` s'étend dans la direction `w`. Par exemple, si `w = [1, 0]` alors `dot(v, w)` est le premier élément de `v`, tout simplement. Autrement dit, 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):
  """v_1 * v_1 + ... + v_n * v_n"""
  return dot(v, v)
```

que nous pouvons utiliser pour calculer sa longueur (magnitude) :

```
import math

def magnitude(v):
  return math.sqrt(sum_of_squares(v))  # math.sqrt est la fonction racine carrée
```

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

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

```
def squared_distance(v, w):
  """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
  return sum_of_squares(vector_subtract(v, w))
```

```
def distance(v, w):
  return math.sqrt(squared_distance(v, w))
```

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

```
def distance(v, w):
  return magnitude(vector_subtract(v, w))
```

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

Note

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, qui 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, chaque liste interne ayant la même taille et représentant une ligne de la matrice. Si `A` est une matrice, `A[i][j]` est l'élément de la $i^{ème}$ ligne et de la $j^{ème}$ colonne. Par convention mathématique, nous utiliserons des majuscules pour représenter les matrices. Par exemple :

```
A = [[1, 2, 3],  # A a 2 lignes et 3 colonnes
     [4, 5, 6]]

B = [[1, 2],     # B a 3 lignes and 2 colonnes
     [3, 4],
     [5, 6]]
```

Note

En mathématiques, on appelle normalement la première ligne de la matrice « ligne 1 » et la première colonne « colonne 1 ». Mais comme nous représentons les matrices au moyen de listes Python, dont l'indice démarre à zéro, nous appellerons la première ligne de la matrice « ligne 0 » et la première colonne « colonne 0 ».

Étant donnée la convention de représentation des listes de listes, la matrice `A` a `len(A)` lignes et `len(A[0])` colonnes, ce que nous appelons sa forme *(shape)* :

```
def shape(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
```

Si une matrice a *n* lignes et *k* colonnes nous la qualifierons de matrice $n \times k$. Nous pouvons 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, i):
  return A[i]               # A[i] est déjà la i^ème ligne

def get_column(A, j):
  return [A_i[j]            # j^ème élément de la ligne A_i
          for A_i in A]     # pour chaque ligne A_i
```

Nous voulons aussi être capables de créer une matrice de forme donnée avec une fonction pour générer ses éléments. Nous pouvons le faire à partir d'une list comprehension imbriquée :

```
def make_matrix(num_rows, num_cols, entry_fn):
  """retourne une matrice num_rows x num_cols matrix dont l'entrée de rang (i,j) est
l'entrée _fn(i, j)"""
    return [[entry_fn(i, j)            # soit i, 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 d'identité 5 × 5 (avec des 1 sur la diagonale et des 0 ailleurs) :

```
def is_diagonal(i, j):
  """des 1 sur la 'diagonale', des 0 partout ailleurs"""
  return 1 if i == j else 0

identity_matrix = make_matrix(5, 5, is_diagonal)

# [[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, en considérant chaque vecteur comme une ligne de la matrice. 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 × 3 :

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

Ensuite, comme nous le verrons plus loin, nous pouvons utiliser une matrice $n \times k$ pour 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 peuvent servir à représenter des relations binaires. Au chapitre 1 nous avons représenté les limites d'un réseau comme une collection de paires `(i, j)`. Une autre représentation consiste à créer une matrice `A` telle que `A[i][j]` est 1 si les nœuds i et j sont connectés, et 0 sinon.

Souvenez-vous que nous avions :

```
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 par :

```
  # utilisateur 0 1 2 3 4 5 6 7 8 9
  #
friendships = [[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, ce sera une représentation 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 de faire un accès à la matrice au lieu d’inspecter (potentiellement) toutes les limites une par une :

```
friendships[0][2] == 1  # Vrai, 0 et 2 sont amis
friendships[0][8] == 1  # Faux, 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 :

```
friends_of_five = [i                                          # il faut seulement
                   for i, is_friend in enumerate(friendships[5])  # regarder
                   if is_friend]                              # une ligne
```

Auparavant, nous avions maintenu une liste de connexions pour chaque nœud afin d’accélérer le processus, mais pour un large graphe en évolution, ce serait sans doute trop coûteux 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). Ce ne serait pas une mauvaise idée de lire un manuel de formation. Il en existe plusieurs en ligne qui sont gratuits (en anglais) :
  - *Linear Algebra*, d'UC Davis ;
  - *Linear Algebra*, du Saint Michael's College ;
  - et si vous êtes plus courageux, *Linear Algebra Done Wrong* est une introduction plus avancée.
- Vous trouverez tout ce que nous avons construit ici disponible gratuitement dans NumPy. (Et encore beaucoup d'autres choses.)

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, qui pourrait plutôt occuper toute une étagère, voire une pièce entière d'une bibliothèque, au lieu d'un simple chapitre dans un livre. Notre discussion ne pourra pas explorer le sujet en profondeur. Ici, je vais vous en apprendre juste assez pour vous rendre redoutable, en espérant piquer suffisamment 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 parmi vos membres, combien de leurs amis peuvent être inclus dans ses sollicitations.

Grâce aux techniques du chapitre 1, vous pouvez facilement lui procurer ces données. Mais votre problème, c'est de savoir comment les décrire.

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

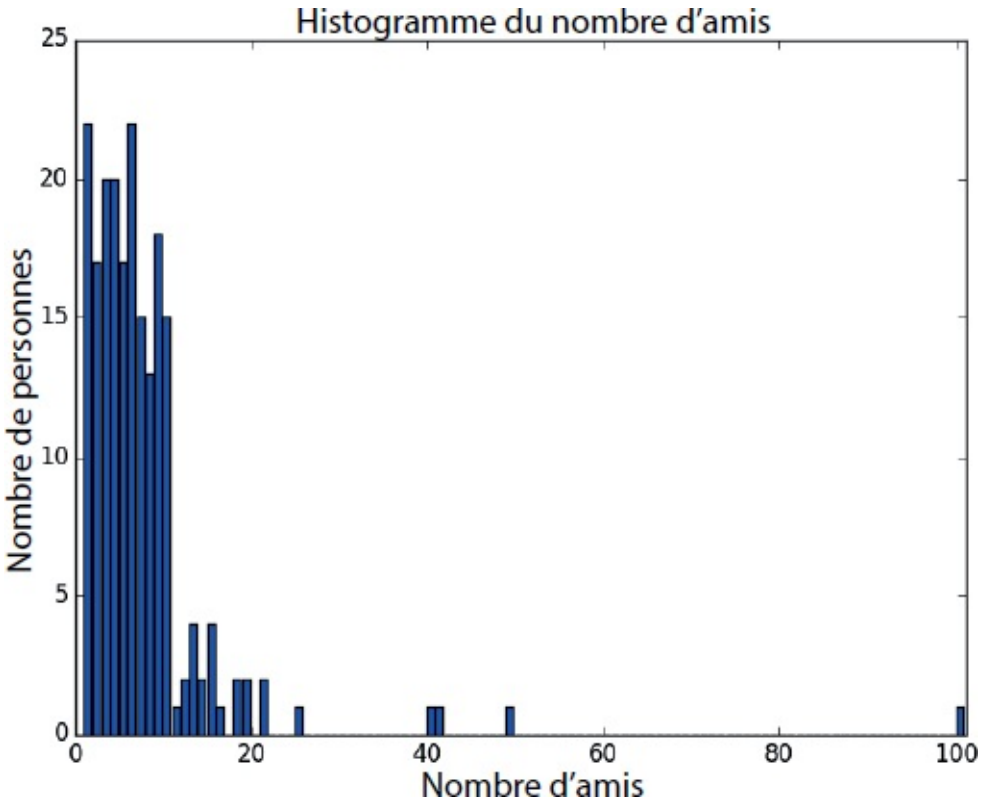
```
num_friends = [100, 49, 41, 40, 25,
               # ... et beaucoup d'autres
              ]
```

Pour un petit nombre de données, c'est sans doute la meilleure description. Mais 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 distiller les 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) :

```
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 seulement le nombre d'amis
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogramme du nombre d'amis")
plt.xlabel("#Nombre d'amis")
plt.ylabel("#Nombre de personnes")
plt.show()
```

Figure 5–1
Un histogramme du nombre 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
```

qui ne sont que des cas particuliers de valeurs figurant dans 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
```

Mais ce n'est que le début.

Les 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 le nombre de données :

```
# ceci n'est pas correct si vous ne le faites pas à partir de la future division import
def mean(x):
    return sum(x) / len(x)

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. Quand vous ajoutez d'autres points, la moyenne se déplace, mais elle dépend toujours de la valeur de chaque point.

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

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

Remarquez que la médiane, à la différence de la moyenne, ne dépend pas de chaque valeur de vos données. Par exemple, si vous agrandissez le point le plus grand (ou si vous réduisez le point le plus petit), les points intermédiaires demeurent inchangés et donc la médiane aussi.

La fonction médiane est cependant un peu plus compliquée en grande partie à cause du cas « pair » :

```
def median(v):
  """trouver la valeur 'la plus au milieu' de v"""
  n = len(v)
  sorted_v = sorted(v)
  midpoint = n // 2
  if n % 2 == 1:
    # si impair, retourner la valeur du milieu
    return sorted_v[midpoint]
  else:
    # si pair, retourner la moyenne des valeurs du milieu
    lo = midpoint - 1
    hi = midpoint
    return (sorted_v[lo] + sorted_v[hi]) / 2

  median(num_friends) # 6.0
```

Manifestement, la moyenne est plus facile à calculer et elle varie en même temps que nos données. Si nous avons *n* données élémentaires et qu'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.) Alors que pour trouver la médiane, il faut trier nos données. Modifier une de nos données élémentaires d'une petite quantité *e* peut soit augmenter la médiane de *e*, soit d'une quantité inférieure à *e*, soit pas du tout (tout dépend des autres données).

> **Note**
> En fait, il existe des astuces cachées pour calculer efficacement la médiane sans trier les données, mais elles sont hors du périmètre de cet ouvrage, donc nous devons trier nos données.

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 à 7,82, mais la médiane ne bougerait pas. S'il est vrai que les valeurs extrêmes sont vraisemblablement des données erronées (ou non représentatives du phénomène que nous essayons de comprendre), alors parfois la moyenne nous donne une image déformée. Par exemple, reprenons 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 représente la valeur pour laquelle 50 % des données sont situées en dessous.)

```
def quantile(x, p):
  """retourne le percentile p^ème dans x"""
  p_index = int(p * len(x))
  return sorted(x)[p_index]


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

Il est moins fréquent de s'intéresser au mode (ou dominante), c'est-à-dire aux valeurs les plus fréquentes :

```
def mode(x):
  """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.iteritems()
          if count == max_count]
```

```
mode(num_friends)  # 1 et 6
```

Mais le plus souvent, nous nous limiterons à la moyenne.

La dispersion

La dispersion fait référence aux mesures qui décrivent comment sont réparties 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'envergure *(range)*, qui est tout simplement la différence entre les éléments les plus grands et les plus petits :

```
# "range" a déjà un sens en Python et nous utiliserons donc un autre nom
def data_range(x):
  return max(x) - min(x)

data_range(num_friends) # 99
```

La portée est de zéro précisément quand le maximum et le minimum sont égaux, ce qui peut se produire seulement si les éléments de `x` sont tous les mêmes, c'est-à-dire quand les données ne sont pas du tout dispersées. Inversement, si la portée est grande, le maximum est largement supérieur au minimum et les données sont davantage éparpillées.

Comme la médiane, la portée ne dépend pas réellement de l'ensemble du jeu de données. Un jeu de données dont les points sont tous 0 ou 100 a la même portée qu'un jeu de données dont les valeurs sont 0, 100 et beaucoup de 50. Mais il semble que le premier jeu de données « devrait » être plus éparpillé.

Une mesure plus complexe de la dispersion est la *variance*, qui se calcule comme suit :

```
def de_mean(x):
     """fait glisser x en retranchant sa moyenne  (de sorte que le résultat donne la
moyenne
0)"""
    x_bar = mean(x)
    return [x_i - x_bar for x_i in x]
def variance(x):
    """suppose que x a au moins deux éléments"""
    n = len(x)
    deviations = de_mean(x)
    return sum_of_squares(deviations) / (n - 1)

variance(num_friends) # 81.54
```

Note

Il semble que ce soit presque le carré médian de la déviation de la moyenne, sauf qu'on divise par *n-1* au lieu de *n*. En réalité, quand on a un échantillon d'une population plus vaste, *x_bar* est seulement une estimation de la moyenne réelle, ce qui signifie qu'en moyenne *(x_ - x_bar) ** 2* est une sous-estimation du carré de la déviation de la moyenne, ce qui explique qu'on divise par *n-1* au lieu de *n*. Pour plus de détails, se reporter à Wikipédia.

Quel que soit l'élément mesuré de nos données (par exemple, « le nombre amis »), toutes nos mesures de tendance centrale sont exprimées dans la même unité. L'envergure elle aussi est exprimée dans cette unité. Au contraire, la variance est dans l'unité originale au carré (« 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 :

```
def standard_deviation(x):
    return math.sqrt(variance(x))


standard_deviation(num_friends)  # 9.03
```

L'envergure 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 $75^{ème}$ et $25^{ème}$ centiles.

```
def interquartile_range(x):
    return quantile(x, 0.75) - quantile(x, 0.25)


interquartile_range(num_friends)  # 6
```

Le résultat est 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 ont sur le site. Elle vous demande de la vérifier.

Après avoir épluché les journaux de trafic du site, vous avez obtenu une liste de durées quotidiennes `daily_minutes` qui montre combien chaque utilisateur passe de minutes sur le site de DataSciencester et vous l'avez triée pour que les éléments correspondent aux éléments de votre liste précédente `num_friends` portant sur les amis. Nous voudrions explorer la relation entre ces deux résultats.

Regardons tout d'abord la covariance qui est associée à la variance. 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 :

```
def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n - 1)

covariance(num_friends, daily_minutes) # 22.43
```

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 de leurs moyennes ou en dessous, un nombre positif est ajouté au total. Quand l'un est au-dessus et l'autre en 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 l'opposé : `x` tend à être petit quand `y` est grand et inversement. 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 (par exemple, amis-minutes par jour), ce qui parfois ne veut pas dire grand-chose (qu'est-ce qu'un « ami-minute par jour » ?).
- Si chaque utilisateur a deux fois plus d'amis (mais passe le même nombre de minutes sur le site), la covariance va doubler. Alors qu'en fait les

variables sont sans doute liées de la même façon qu'avant. Autrement dit, il est difficile de définir une « grande » covariance.

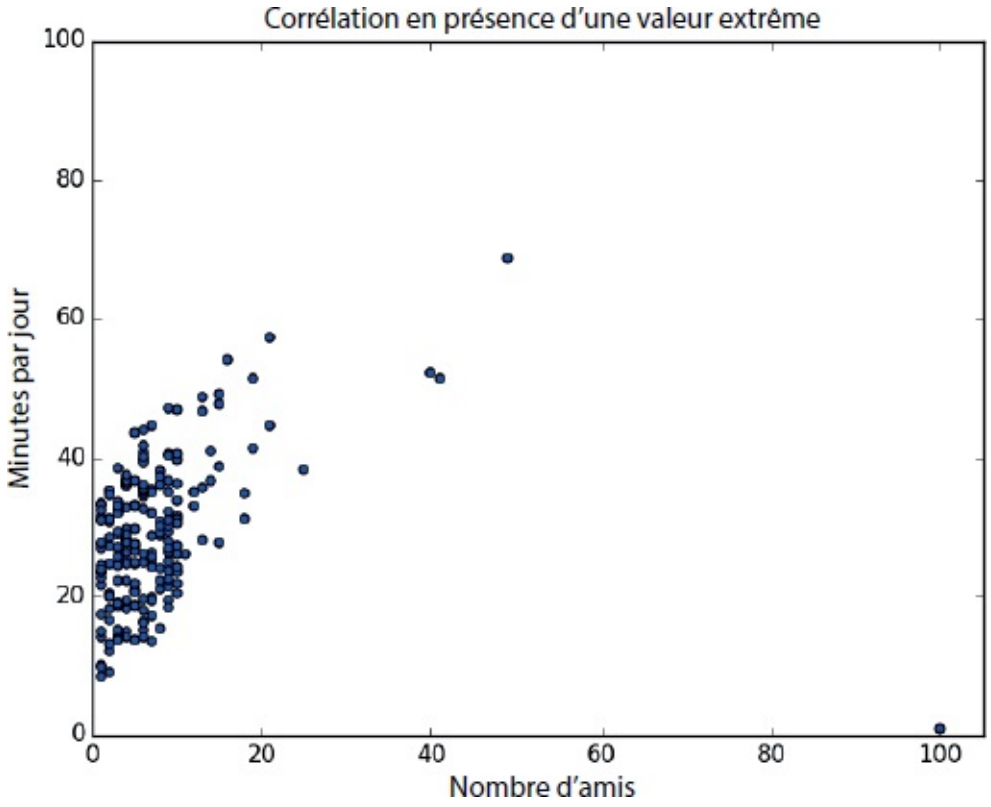
Pour cette raison, il est plus fréquent d'examiner la corrélation, qui est le résultat de la division de l'écart-type des deux variables :

```
def correlation(x, y):
  stdev_x = standard_deviation(x)
  stdev_y = standard_deviation(y)
  if stdev_x > 0 and stdev_y > 0:
      return covariance(x, y) / stdev_x / stdev_y
  else:
      return 0  # en l'absence de variation, la corrélation est à zéro

correlation(num_friends, daily_minutes) # 0.25
```

La corrélation n'a pas d'unité et elle 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



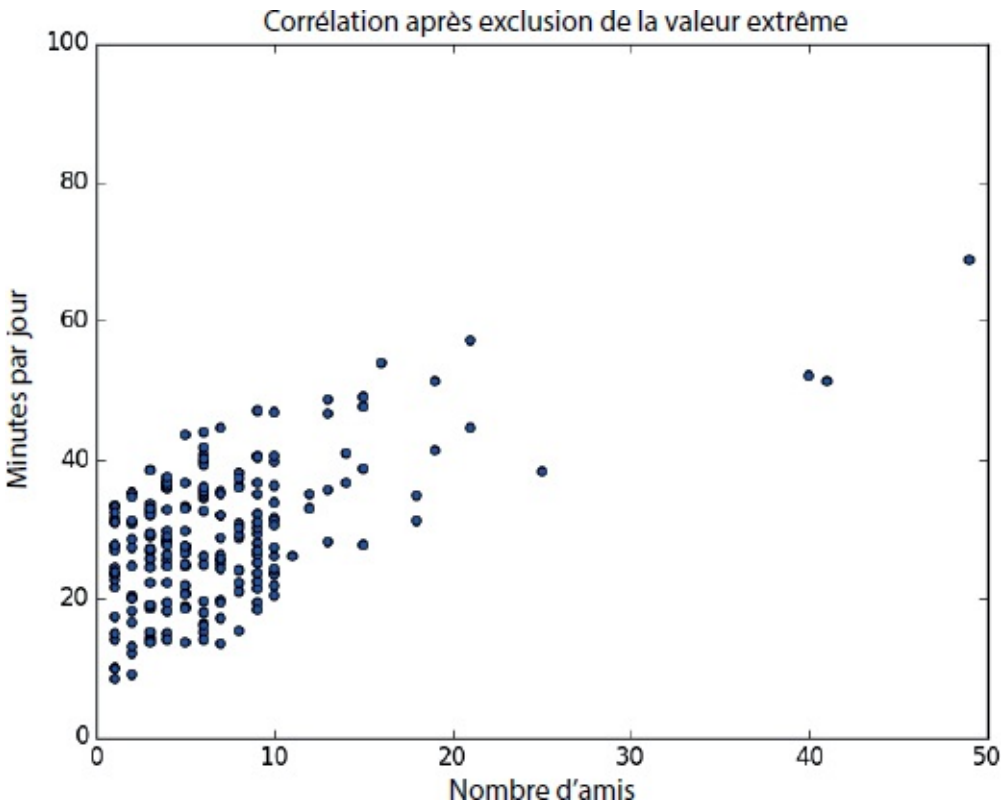
La personne aux 100 amis (qui passe seulement une minute par jour sur le site) est une valeur extrême. Or, les corrélations peuvent se montrer très sensibles aux valeurs extrêmes. Que se passe-t-il si nous l'ignorons ?

```
outlier = num_friends.index(100)  # index de l'extrême

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

correlation(num_friends_good, daily_minutes_good)  # 0.57
```

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

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.

Figure 5–3
Corrélation après exclusion de la valeur extrême



Le paradoxe de Simpson

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

Par exemple, imaginons que vous pouvez identifier tous les membres comme étant data scientists de la côte Est ou data scientists 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, ou du café, ou des produits bio, ou de l'ambiance cool et relax 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 *(PhD)*, 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, les data scientists de la côte Est ont aussi en moyenne plus d'amis !

Côte	Diplôme	Nombre de membres	Nombre moyen d'amis
Ouest	PhD	35	3,1
Est	PhD	70	3,2
Ouest	pas de PhD	66	10,9
Est	pas de PhD	33	13,9

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 est que la corrélation mesure la relation entre vos deux variables, tout le reste étant égal par ailleurs. Si vos classes de données sont assignées de manière aléatoire, comme elles devraient l'être dans une expérience bien conçue, « tout

le reste étant égal par ailleurs » n'est pas une hypothèse si mauvaise. Mais quand il y a une logique sous-jacente au choix des classes, « tout le reste étant égal par ailleurs » peut devenir une 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, ce n'est pas toujours possible. Si vous n'aviez pas connaissance du parcours universitaire de ces 200 data scientists, vous pourriez conclure simplement 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 exister quand même d'autres types de relations. Par exemple :

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

Dans ce cas, `x` et `y` ont zéro corrélation. Mais il y a certainement une relation, puisque chaque élément de `y` est égal à la valeur absolue de l'élément de `x` correspondant. Ce qu'ils n'ont pas, c'est une relation dans laquelle la comparaison de `x_i` et `mean(x)` nous donne des informations sur la comparaison de `y_i` et `mean(y)`. C'est le type de relation qui intéresse les corrélations.

De plus, la corrélation ne vous dit rien de l'importance de la relation. Les variables :

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

sont parfaitement corrélées, mais (suivant ce que vous mesurez) il est tout à fait possible que cette relation ne présente aucun intérêt.

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`. Il est possible qu'avoir plus d'amis sur le site incite les utilisateurs de DataSciencester à passer plus de temps sur le site. Ce sera le cas si chaque ami poste tous les jours un certain contenu, ce qui veut dire que plus vous avez d'amis, plus vous passez de temps sur le site pour rester au courant de leurs mises à jour.

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

Une troisième possibilité est que les utilisateurs les plus passionnés de data science passent plus de temps sur le site (car ils trouvent les débats plus intéressants) et ils y cherchent plus activement des amis de la data science (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 cobayes, vous pourriez choisir au hasard un sous-ensemble d'utilisateurs et leur montrer du 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, pandas et StatsModels 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 data scientist, ce serait une bonne idée de lire des ouvrages sur le sujet. Beaucoup sont disponibles en ligne (en anglais). J'apprécie particulièrement :
  - *OpenIntro Statistics* ;
  - *OpenStax 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 lorsque nous avons traité de la statistique au chapitre 5, 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, pensez à un lancer de dé. L'univers comprend tous les résultats possibles. Et chaque sous-ensemble de 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 disserter 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 face. Ces événements sont indépendants. Par contre, savoir que le premier lancer a donné face nous donne certainement une information concernant le fait que les deux lancers donnent pile. (Si le premier lancer produit face, alors certainement les deux lancers ne peuvent pas donner 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 est le produit de la probabilité de chacun :

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

Dans l'exemple ci-dessus, la probabilité de « premier lancer = face » est de 1/2 et la probabilité de « deux lancers = pile » est de 1/4, mais la probabilité de « premier lancer = face et les deux lancers = pile » est de zéro.

La 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 » si :

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

Il faut interpréter cela comme la probabilité que E se produise étant donné que nous savons que F s'est produit.

Nous écrirons souvent l'expression ci-dessus sous la forme :

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

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

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

ce qui est l'expression mathématique du fait que savoir que F s'est produit ne nous donne pas d'information supplémentaire pour savoir si E s'est produit.

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

1. chaque enfant a autant de chances d'être une fille que d'être un garçon ;
2. et 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 » à la probabilité 1/2 et l'événement « deux filles » a une probabilité de 1/4.

Nous pouvons nous demander quelle est la probabilité de l'événement (B) « les deux enfants sont des filles » 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 de l'événement « les

deux enfants sont des filles » conditionné à 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. Ce qui veut dire que nous avons :

$$P(B|L) = P(B,L)/P(L) = P(B)/P(L) = 1/3$$

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 :

```
def random_kid():
  return random.choice(["boy", "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 == "girl":
    older_girl += 1
  if older == "girl" and younger == "girl":
    both_girls += 1
  if older == "girl" or younger == "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
```

Le théorème de Bayes

Un des meilleurs amis de l'expert en data science est le théorème de Bayes qui offre une manière d'« inverser » les probabilités conditionnelles. Supposons que nous avons besoin de la probabilité d'un certain événement *E* conditionné par un autre événement *F*. Mais nous n'avons que des informations sur la probabilité de *F* conditionné par *E*. En utilisant deux fois la définition de la probabilité conditionnelle, nous déduisons que :

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

L'événement *F* peut être divisé en deux événements mutuellement exclusifs « *F* et *E* » et « *F* et pas *E* ». Si nous écrivons ¬*E* pour « pas *E* » (c'est-à-dire « *E* ne se produit pas »), alors :

$$P(F) = P(F,E) + P(F,\neg E)$$

de sorte que :

$$P(E|F) = P(F|E)\ P(E)/[P(F|E)P(E) + P(F|\neg E)P(\neg E)]$$

ce qui est la formulation la plus fréquente du théorème de Bayes.

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 qui donne le résultat correct (« infecté » si la personne est malade, « non infecté » si la personne n'est pas malade ») 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 :

$$P(D|T) = P(T|D)P(D)/P(T|D)P(D) + P(T|\neg D)P(\neg D)$$

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 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.

Note

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.

Alors que c'est un simple calcul pour un data scientist, la plupart des médecins vont deviner que $P(D \mid T)$ vaut environ 2.

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. Ce qui veut dire que vous attendez à ce que seulement 99 parmi (99 + 9999) personnes avec un test positif soit effectivement malades.

Les 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 vaut 1 si une pièce lancée indique face et 0 si la pièce indique pile. Une variable plus complexe, par exemple, mesure le nombre de faces observées quand on lance la pièce 10 fois ou correspond 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 d'une variable aléatoire qui est la moyenne de ses valeurs pondérée par leurs probabilités. La variable aléatoire associée à la pièce a une espérance de 1/2 (= 0 * 1/2 + 1 * 1/2) et la variable `range(10)` a une espérance de 4,5.

Les variables aléatoires peuvent être conditionnées par des événements comme n'importe quel événement. Si nous revenons à notre exemple des deux enfants (voir plus haut, « Probabilité conditionnelle »). Si *X* est la variable aléatoire représentant le nombre de filles, *X* est égal à 0 avec une probabilité de 1/4, égal à 1 avec une probabilité de 1/2 et égal 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 égal à 1 avec la probabilité 2/3 et égal à 2 avec la probabilité 1/3. Et une variable *Z* qui est le nombre de filles conditionné par le fait que l'aîné des enfants est une fille est égal à 1 avec la probabilité 1/2 et égal à 2 avec la probabilité 1/2.

La plupart du temps, nous utiliserons des variables aléatoires implicitement sans les mettre particulièrement en valeur. Mais vous les verrez si vous leur prêtez attention.

Les 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 être égal à zéro. Pour cette raison nous représenterons une distribution continue par une fonction de densité de probabilité (*probability density function* ou pdf) telle que la probabilité de voir une valeur dans un certain intervalle est égale à l'intégrale de la fonction sur cet intervalle.

> **Note**
>
> Si vos souvenirs de calcul intégral sont trop lointains, une manière simple de comprendre est de se dire que si une distribution a une densité de probabilité *f*, alors la probabilité de voir une valeur entre *x* et *x+h* est approximativement *h * f(x)* si *h* est petit.

La densité de la distribution uniforme est donc :

```
def uniform_pdf(x):
  return 1 if x >= 0 and x < 1 else 0
```

La probabilité qu'une variable aléatoire qui suit cette distribution soit entre 0,2 et 0,3 est de 1/10, comme vous pouviez le deviner.

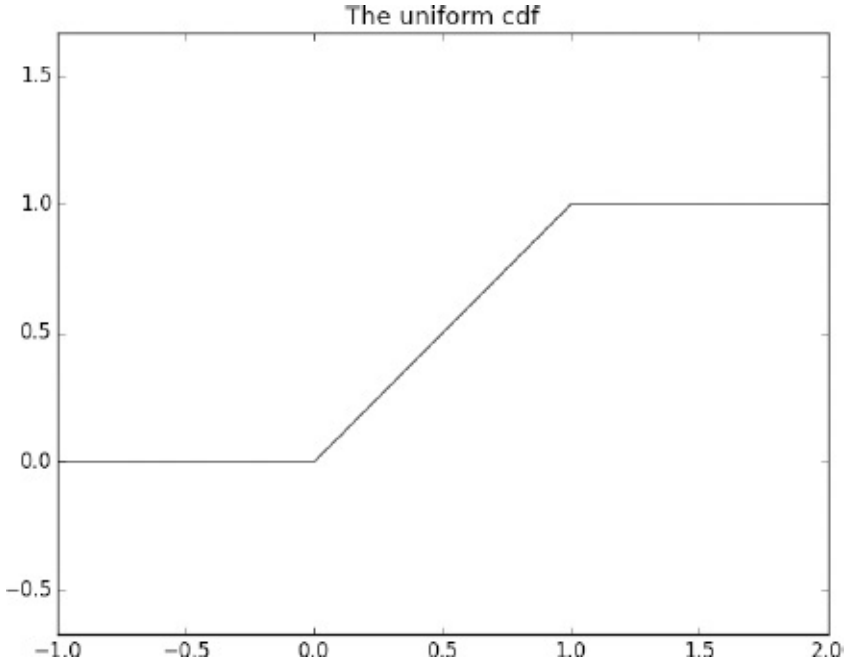
La variable Python `random.random()` est une variable (pseudo) aléatoire de densité uniforme.

Nous nous intéresserons davantage à la fonction de distribution cumulative (*cumulative distribution function* ou cdf) également appelée fonction de répartition, qui 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):
  """retourne la probabilité qu'une distribution aléatoire est <= x"""
  if x < 0:   return 0 # la distribution aléatoire uniforme n'est jamais inférieure à 0
  elif x < 1: return x # c'est-à-dire P(X <= 0.4) = 0.4
```

```
else:       return 1 # la distribution aléatoire uniforme est toujours inférieure à 1
```

Figure 6–1
La fonction de distribution cumulative de la distribution uniforme



La 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.

Elle a une fonction de densité :

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

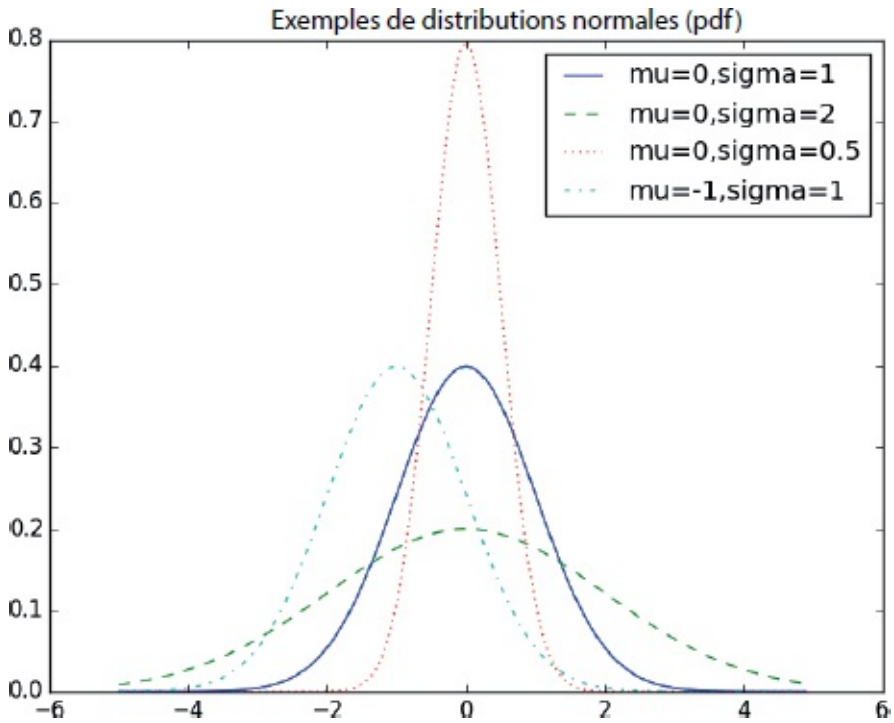
que nous pouvons implémenter de la manière suivante :

```
def normal_pdf(x, mu=0, sigma=1):
  sqrt_two_pi = math.sqrt(2 * math.pi)
  return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

Sur la figure 6-2, nous avons tracé quelques-unes de ces pdf pour voir à quoi elles ressemblent :

```
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.

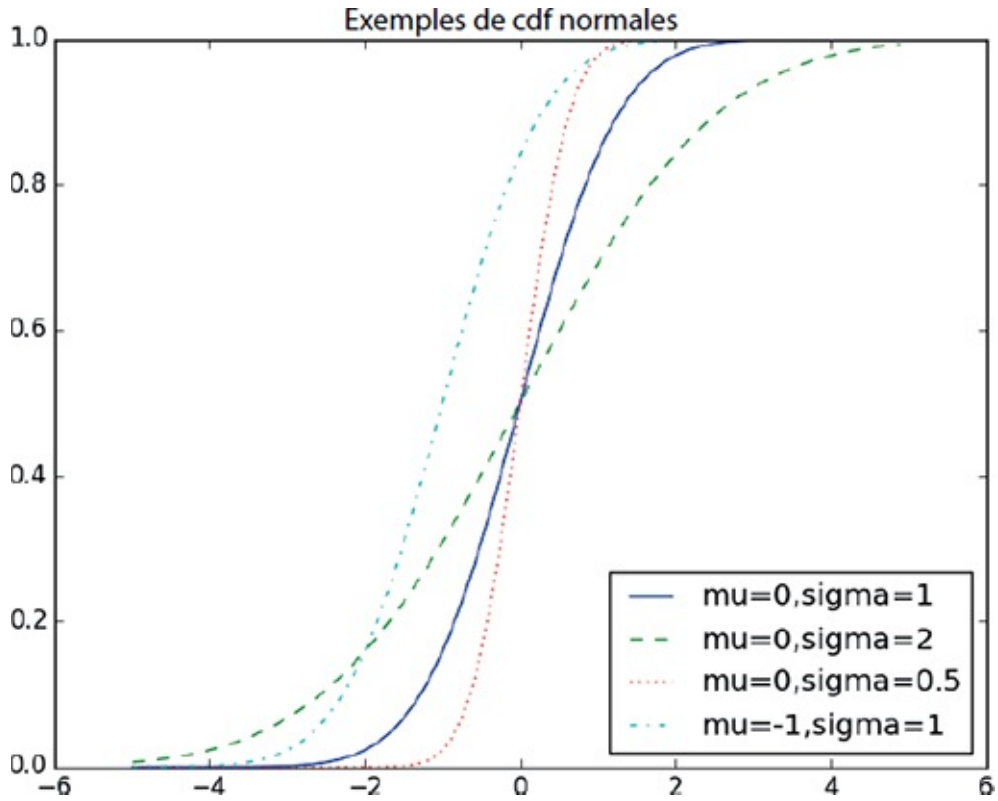
La fonction de répartition de la loi normale ne peut pas être écrite sous forme « élémentaire », mais nous pouvons l'écrire en Python à l'aide de `math.erf` :

```
def normal_cdf(x, mu=0,sigma=1):
  return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Dans la figure 6-3, nous avons représenté quelques exemples :

```
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 cdf normales")
plt.show()
```

Figure 6–3
Exemples de cdf normales



Parfois, nous aurons besoin d'inverser la fonction `normal_cdf` pour trouver la valeur correspondante 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 dichotomie :

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    """trouver l'inverse approximatif par 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 des cdf à cet endroit
        if mid_p < p:
            # le point median est encore trop bas, chercher au-dessus
            low_z = mid_z
        elif mid_p > p:
            # le point median est encore trop haut, chercher en dessous
            hi_z = mid_z
        else:
```

```
            break

    return mid_z
```

La fonction segmente l’intervalle en deux parties égales plusieurs fois jusqu’à le réduire à une valeur *Z* assez proche de la probabilité recherchée.

Le théorème de la limite centrale

Une des raisons de l'intérêt de la distribution normale est le 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 normale.

En particulier, si x_1, …, x_n sont des variables aléatoires d'espérance μ et d'écart-type σ, et si n est grand, alors :

$$\frac{1}{n}(x_1 + \cdots + x_n)$$

est approximativement de distribution normale avec l'espérance μ et l'écart-type $\sigma / \sqrt{n}$.

De même (et souvent plus intéressant) :

$$\frac{(x_1 + \cdots + x_n) - \mu n}{\sigma\sqrt{n}}$$

suit approximativement une loi normale, d'espérance 0 et d'écart-type 1.

Une manière simple d'illustrer ce résultat est d'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):
  return 1 if random.random() < p else 0


def binomial(n, 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 $\sigma = \sqrt{np(1-p)}$.

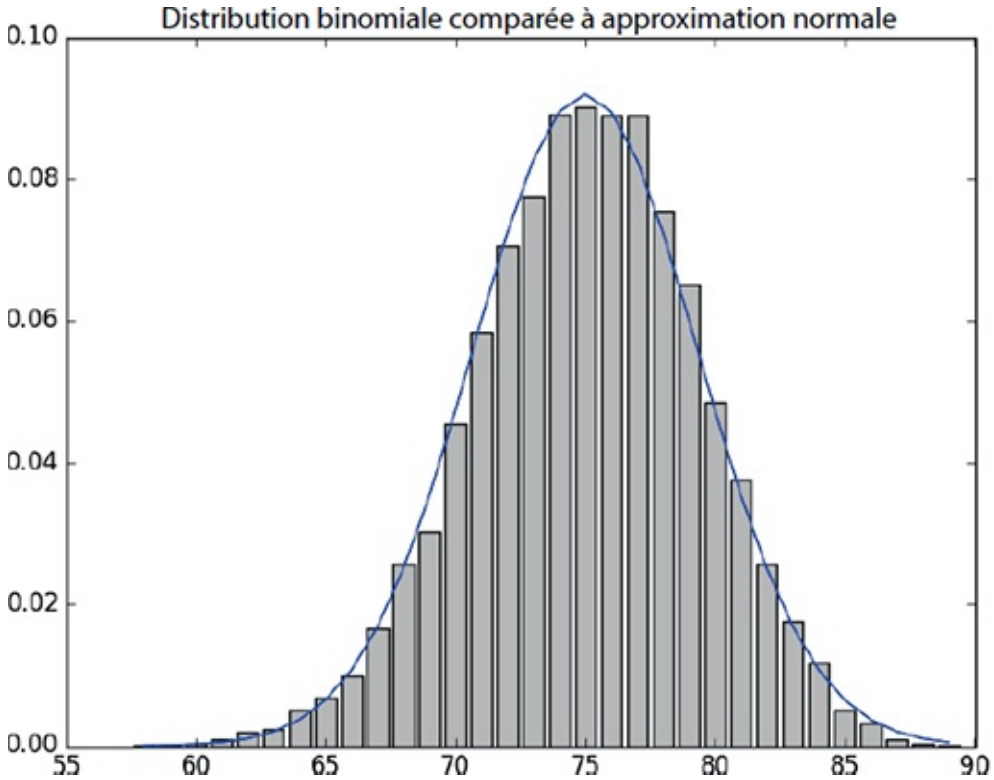
Si nous représentons graphiquement les deux, vous pourrez facilement voir la ressemblance :

```
def make_hist(p, n, num_points):

  data = [binomial(n, p) for _ in range(num_points)]

  # utiliser un diagramme en bâtons pour montrer 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))

  # utiliser une courbe pour montrer 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.

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 `Normal(50,5)` soit plus grand que 60, ce qui est plus facile que de calculer la cdf de `Binomial(100, 0,5)`. (En réalité, dans la plupart des applications vous disposerez probablement d'un logiciel statistique qui calculera à votre place toutes les probabilités que vous voudrez.)

Figure 6–4
Le résultat de make_hist


Distribution binomiale comparée à approximation normale
0.10
0.08
0.06
0.04
0.02
0.00
55
60
65
70
75
80
85
90

Pour aller plus loin

- scipy.stats contient des fonctions pdf et cdf pour la plupart des distributions de probabilité les plus courantes.
- Vous vous souvenez comment, à la fin du chapitre 5, je vous avais suggéré que ce serait une bonne idée de feuilleter un manuel de statistique ? Eh bien, ce serait une idée tout aussi bonne d'étudier un manuel de probabilités. Le meilleur que je connaisse est *Introduction to Probability*, disponible en ligne (en anglais).

7

Hypothèse et inférence

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 nos données et les processus qui les ont générées.

Le test statistique d’une d’hypothèse

Souvent les data scientists veulent tester si une hypothèse donnée a de fortes chances d’être vraie. Pour nous, les hypothèses sont des affirmations que nous pouvons traduire en termes de statistiques sur les données, comme « cette pièce n’est pas truquée » ou « les data scientists préfèrent Python à R » ou « 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 connues, ce qui nous permet de décider si 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 hypothèse alternative H_1 avec lesquelles nous voulons la comparer. Nous utilisons les statistiques pour décider si H_0 est fausse ou non. Un exemple sera certainement plus parlant.

Exemple : le lancer de pièce

Imaginons que vous ayez une pièce et que nous voulons tester si elle honnête (c'est-à-dire non truquée). Faisons l'hypothèse que la pièce a la probabilité *p* de donner face ; l'hypothèse nulle est que la pièce est non truquée, ce qui veut dire que $p = 0{,}5$. 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 de 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 assimiler approximativement par une distribution normale (chapitre 6) :

```
def normal_approximation_to_binomial(n, p):
  """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 soit à l'intérieur (ou à l'extérieur) d'un intervalle donné :

```
# le cdf normal est 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, mu=0, sigma=1):
  return 1 - normal_cdf(lo, mu, sigma)

# entre les deux si inférieure à hi, mais pas inférieure à lo
def normal_probability_between(lo, hi, mu=0, sigma=1):
  return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# à l'extérieur si elle n'est pas entre les deux
def normal_probability_outside(lo, hi, mu=0, sigma=1):
  return 1 - normal_probability_between(lo, hi, mu, sigma)
```

Nous pouvons aussi faire le raisonnement inverse : chercher la région excluant les queues 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 trouver les seuils à proximité desquels les extrémités inférieures et supérieures contiennent 20 % de la probabilité (ce qui laisse 60 %) :

```
def normal_upper_bound(probability, mu=0, sigma=1):
  """retourne la valeur de z pour laquelle P(Z <= z) = probabilité"""
  return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability, mu=0, sigma=1):
  """retourne la valeur z pour laquelle P(Z >= z) = probabilité"""
  return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability, mu=0, sigma=1):
  """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 d'honnêteté 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 :

```
normal_two_sided_bounds(0.95, mu_0, sigma_0)  # (469, 531)
```

En supposant que `p` est réellement égal 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 approximativement 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, 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 uniquement que p n'est pas égal à 0,5 ne vous renseigne pas vraiment sur la distribution de X.) En particulier, examinons ce qui arrive si p est réellement égal 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 quand X est encore dans notre intervalle d'origine (calculé sur
# la base de p = 0.5)
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 est que la pièce n'est pas biaisée en faveur de face, ou que $p \leq 0{,}5$. Dans ce cas, nous voulons un test unilatéral qui rejette l'hypothèse nulle quand X est beaucoup plus grand que 500, mais pas quand X est plus petit que 500. De fait, tester une valeur significative de 5 % veut 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_probabilité  # 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) et 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, calculons la probabilité (en supposant que H_0 est vraie) de voir une valeur au moins aussi extrême que celle que nous avons véritablement observée.

Pour notre test bilatéral destiné à savoir si la pièce est honnête, calculons :

```
def two_sided_p_value(x, mu=0, sigma=1):
  if x >= mu:
    # si x est plus grand que l'espérance, la queue de distribution est ce qui est
    # supérieur à x
    return 2 * normal_probability_above(x, mu, sigma)
  else:
    # si x est inférieur à l'espérance, la queue de distribution est ce qui est inférieur
    # à x
    return 2 * normal_probability_below(x, mu, sigma)
```

Si nous avions vu 530 fois face, nous ferions le calcul :

```
two_sided_p_value(529.5, mu_0, sigma_0)  # 0.062
```

Note

Pourquoi utiliser 529,5 au lieu de 530 ? C'est ce qui s'appelle une *correction de continuité*. 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 `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.

Pour vous convaincre que c'est une estimation raisonnable, faisons une simulation :

```
extreme_value_count = 0
for _ in range(100000):
  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:         # compte combien de fois
       extreme_value_count += 1                      # le # est 'extreme'

print extreme_value_count / 100000                   # 0.062
```

Comme la `p-value` est supérieure à notre valeur significative de 5 %, nous ne rejetons pas l'hypothèse nulle. 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 %, ce qui veut dire que nous rejetterions 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 avions vu sortir face 525 fois, nous ferions le calcul :

```
upper_p_value(524.5, mu_0, sigma_0)  # 0.061
```

ce qui veut dire ne pas rejeter l'hypothèse nulle. Si nous avions obtenu face 527 fois, le calcul serait alors :

```
upper_p_value(526.5, mu_0, sigma_0)  # 0.047
```

et nous aurions rejeté l'hypothèse nulle.

Attention

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 est : « si nous supposons que les données sont distribuées normalement, la chance … » 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. Commencer par une simple représentation graphique des données est toujours un bon début.

L’intervalle de confiance

Nous avons testé des hypothèses sur la probabilité `p` de voir sortir face, qui est 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 1000 lancers, nous estimerons que *p* est égal à 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 (vu plus haut) nous dit qu’en moyenne ces variables de Bernoulli devraient approximativement suivre une loi normale, d’espérance `p` et d’écart-type :

```
math.sqrt(p * (1 - p) / 1000)
```

Mais ici 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
```

Ce n’est pas totalement justifié, mais c’est couramment pratiqué. Avec l’approximation de la loi normale, nous pouvons conclure que nous sommes « confiant à 95 % » que l’intervalle suivant contient le vrai paramètre *p* :

```
normal_two_sided_bounds(0.95, mu, sigma)  # [0.4940, 0.5560]
```

Note

Cette affirmation concerne l’intervalle, mais pas *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 chaque fois) se situerait dans l’intervalle de confiance observé (qui peut être différent à chaque fois).

En particulier, 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 est honnête » ne se situe pas dans l'intervalle de confiance. (L'hypothèse « pièce honnête » ne réussit pas le test qu'elle devrait 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 :

```
def run_experiment():
  """lancer une pièce non truquée 1000 fois, Vrai = face, Faux = pile"""
  return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment):
  """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)])

print num_déjections  # 46
```

Ce qui 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. Enlevez les bons extrêmes et vous pourrez probablement faire passer votre valeur `p` en dessous de 0,05. (Nous avons fait quelque chose de vaguement ressemblant précédemment sous l'intitulé « Corrélation ». L'avez-vous remarqué ?)

Ce phénomène, parfois appelé *P-hacking*, est une conséquence de « l'inférence des `p-values` ». L'article intitulé « The Earth Is Round » (la terre est ronde) est une bonne critique de cette approche.

Si vous voulez adopter une approche scientifique sérieuse, vous devez formuler vos hypothèses avant de regarder vos données, vous devez nettoyer vos données sans penser à vos hypothèses et vous devez garder en tête que les `p-values` ne remplacent pas le bon sens. (L'inférence bayésienne est une autre approche abordée plus loin.)

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 (« quel régal ! ») et l'annonce B (« mon biaisé ! »).

En bon scientifique, vous décidez de faire 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 l'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 meilleur que B. Mais que conclure si les différences ne sont pas si marquées ? C'est là que nous allons utiliser 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. Alors, (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 $A=\sqrt{P_A(1-P_A)/N_A}$.

De même, n_B/N_B est approximativement une variable aléatoire normale d'espérance p_B et d'écart-type $B=\sqrt{P_B(1-P_B)/N_B}$:

```
def estimated_parameters(N, n):
  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 $\sqrt{\sigma_A^2 + \sigma_B^2}$.

Note

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 réellement une loi de Student. Mais pour un très grand nombre de données, 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 $pA - pB$ = zéro) en utilisant la statistique ci-dessous :

```
def a_b_test_statistic(N_A, n_A, N_B, n_B):
  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)
```

ce qui devrait correspondre approximativement à une distribution normale standard.

Par exemple, si le slogan A récolte 200 clics pour 1 000 lecteurs alors que le slogan B en récolte 180, alors la statistique vaut :

```
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
```

ce qui est suffisant pour vous empêcher de conclure qu'il y a une grande différence. D'un autre côté, si le slogan A reçoit seulement 150 clics, alors :

```
z = a_b_test_statistic(1000, 200, 1000, 150)  # -2.94
two_sided_p_value(z)                          # 0.003
```

ce qui signifie qu'il y a seulement une probabilité de 0,003 de voir une grande différence si les annonces avaient la même efficacité.

L’inférence bayésienne

La procédure que nous venons de voir nécessitait des décisions sur les probabilités dans nos tests : « Il n’y a que 3 % de chances d’observer des statistiques aussi extrêmes si notre hypothèse nulle était vraie. »

Une autre approche de l’inférence consiste à traiter les paramètres inconnus eux-mêmes comme des variables aléatoires. L’analyste (vous !) 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, beta):
  """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, alpha, beta):
  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 + bêta)
```

Plus alpha et bêta sont grands et plus la distribution est resserrée.

Par exemple, si alpha et bêta valent tous les deux 1, on retrouve une distribution uniforme (centrée sur 0,5 et très dispersée). Si alpha est beaucoup plus grand que bêta, l’essentiel du poids est proche de 1. Si alpha est beaucoup plus petit que bêta, l’essentiel du poids est proche de zéro. La figure 7-1 illustre différentes distributions bêta.

Soit une distribution a priori de *p*. Supposons que nous ne voulons pas préjuger de l’honnêteté de la pièce, et que nous décidons qu’alpha et bêta sont égaux à 1. Nous pourrions aussi être persuadés que la pièce fait sortir face 55 % du temps, et choisir `alpha` = 55 et `bêta` = 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 `bêta + t`.

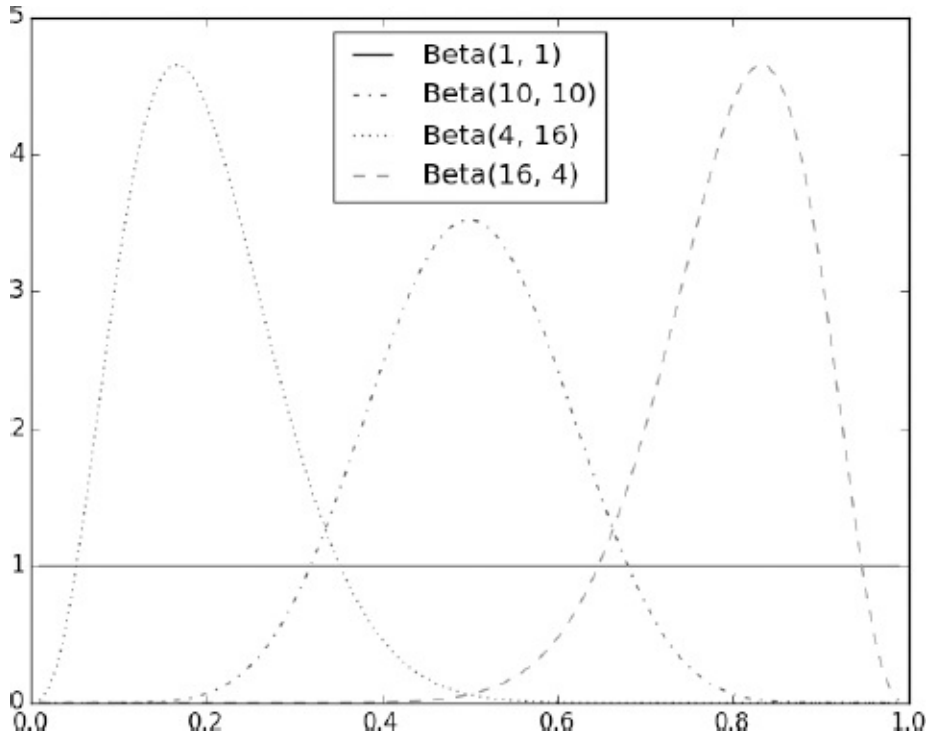
Note

Le fait que la distribution a posteriori suive la loi bêta n'a rien d'une coïncidence. Le nombre de retombées « face » est donné par une distribution binomiale et la loi bêta est le conjugué a priori de la distribution binomiale. En d'autres termes, quand vous mettez à jour une bêta a priori en utilisant des observations de la binomiale correspondante, vous retrouvez la bêta a posteriori.

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 est peut-être truquée) votre distribution a posteriori sera une bêta(4, 8) centrée sur 0,33.

Figure 7–1
Exemples de distributions bêta



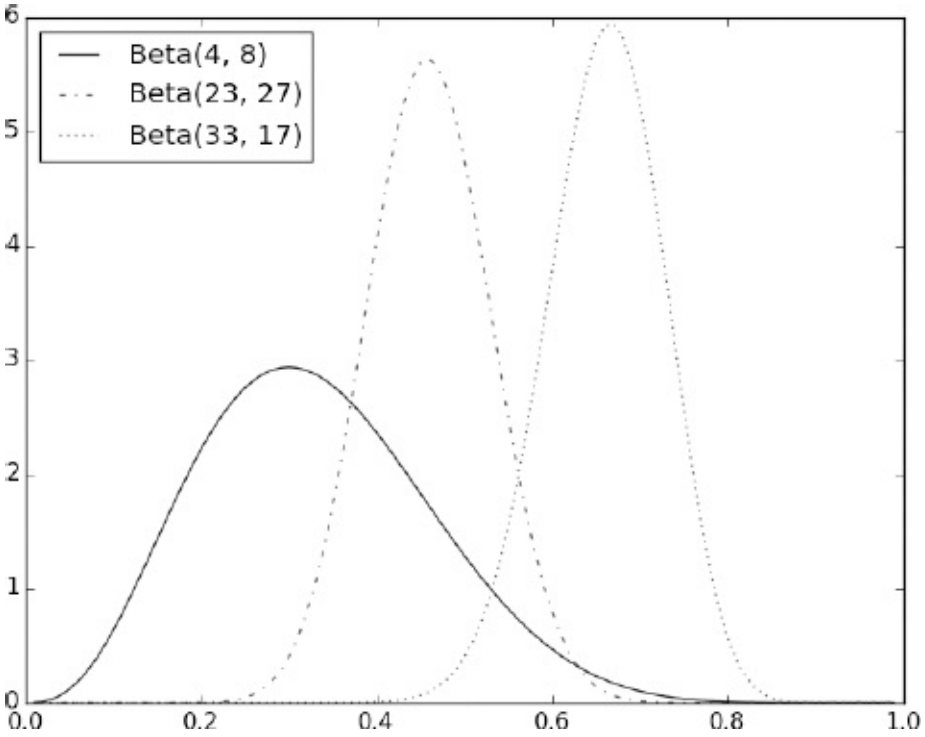
Comme vous avez considéré que toutes les probabilités ont la même vraisemblance, votre prévision est assez proche de la réalité observée.

Si vous avez commencé avec une bêta(20, 0) (exprimant par là que la pièce est globalement correcte), votre distribution a posteriori sera une bêta(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 bêta(30, 10) (exprimant votre conviction que la pièce est biaisée pour tomber sur face à 75 %) votre distribution a posteriori sera une bêta(33, 17) centrée sur 1,66. Dans ce cas, la pièce est biaisée en faveur de pile, mais pas autant que vous le pensiez initialement. Ces trois distributions a posteriori sont reprises sur la figure 7-2.

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 biais de la pièce, il sera difficile de vous en tenir à votre opinion initiale après 1 000 retombées « face » sur 2 000 lancers (à moins d'être un original qui a choisi pour distribution a priori quelque chose du style bêta(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 était non biaisée, 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* qui couvre plusieurs aspects de cette problématique.

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 essayerons 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 devrons donc nous attacher à résoudre un certain nombre de problèmes d'optimisation. Et en particulier, il faudra 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.

L’idée qui se cache 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. Voici un exemple simple :

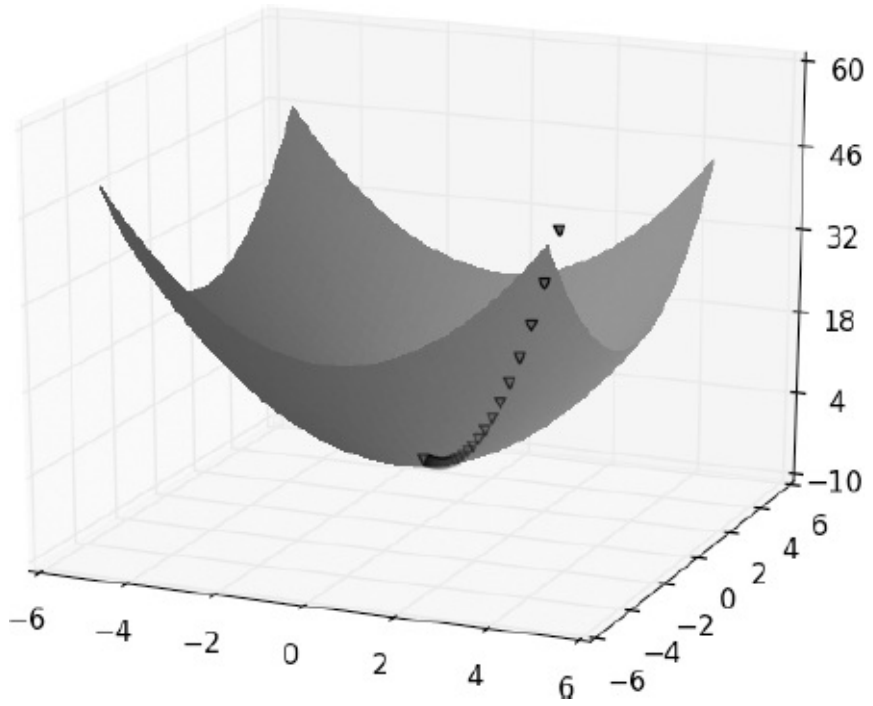
```
def sum_of_squares(v):
  """calcule la somme des carrés des éléments dans v"""
  return sum(v_i ** 2 for v_i in v)
```

Nous aurons souvent à maximiser (ou minimiser) ce genre de fonction. En d’autres termes, nous devrons trouver l’entrée `v` qui produit la plus grande (ou la plus petite) valeur possible.

Pour des fonctions comme les nôtres, le gradient (au cas où vous auriez oublié vos cours de calcul infinitésimal, c’est le vecteur des dérivées partielles) donne la direction vers laquelle la fonction augmente le plus rapidement. (Si vous avez tout oublié du calcul infinitésimal, croyez-moi sur parole ou vérifiez sur Internet.)

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 du gradient (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, comme le montre la figure 8-1.

Figure 8–1
Trouver un minimum à partir de la descente de gradient



Note

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 désigner celui qui n'est pas le minimum global ; dans ce cas, il faut 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

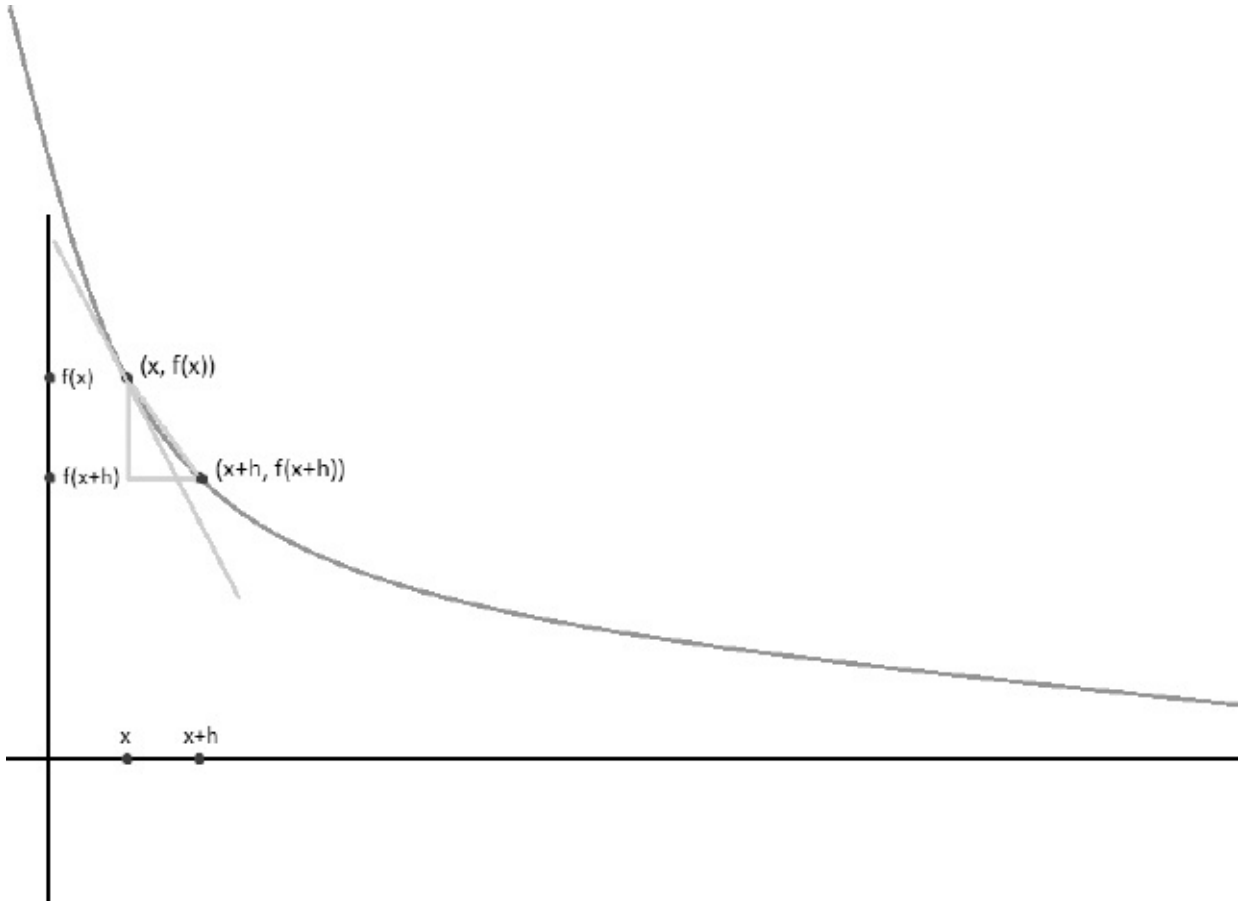
Si *f* est une fonction à une variable, sa dérivée au point *x* mesure comment `f(x)` évolue quand on effectue un petit changement sur *x*. Elle est définie comme la limite des quotients de différence :

```
def difference_quotient(f, x, h):
  return (f(x + h) - f(x)) / h
```

quand `h` tend vers zéro.

(De nombreux étudiants débutants en calcul infinitésimal se sont heurtés à la définition mathématique de la limite. Nous allons contourner cette difficulté en déclarant ici que la limite signifie ce que vous pensez qu'elle signifie.)

Figure 8–2
Approximation d'une dérivée avec un quotient de différence



La dérivée est la pente de la tangente à `(`*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).

Pour de nombreuses fonctions, il est facile de calculer les dérivées exactes. Par exemple, la fonction d'élévation au carré :

```
def square(x):
  return x * x
```

a pour dérivée :

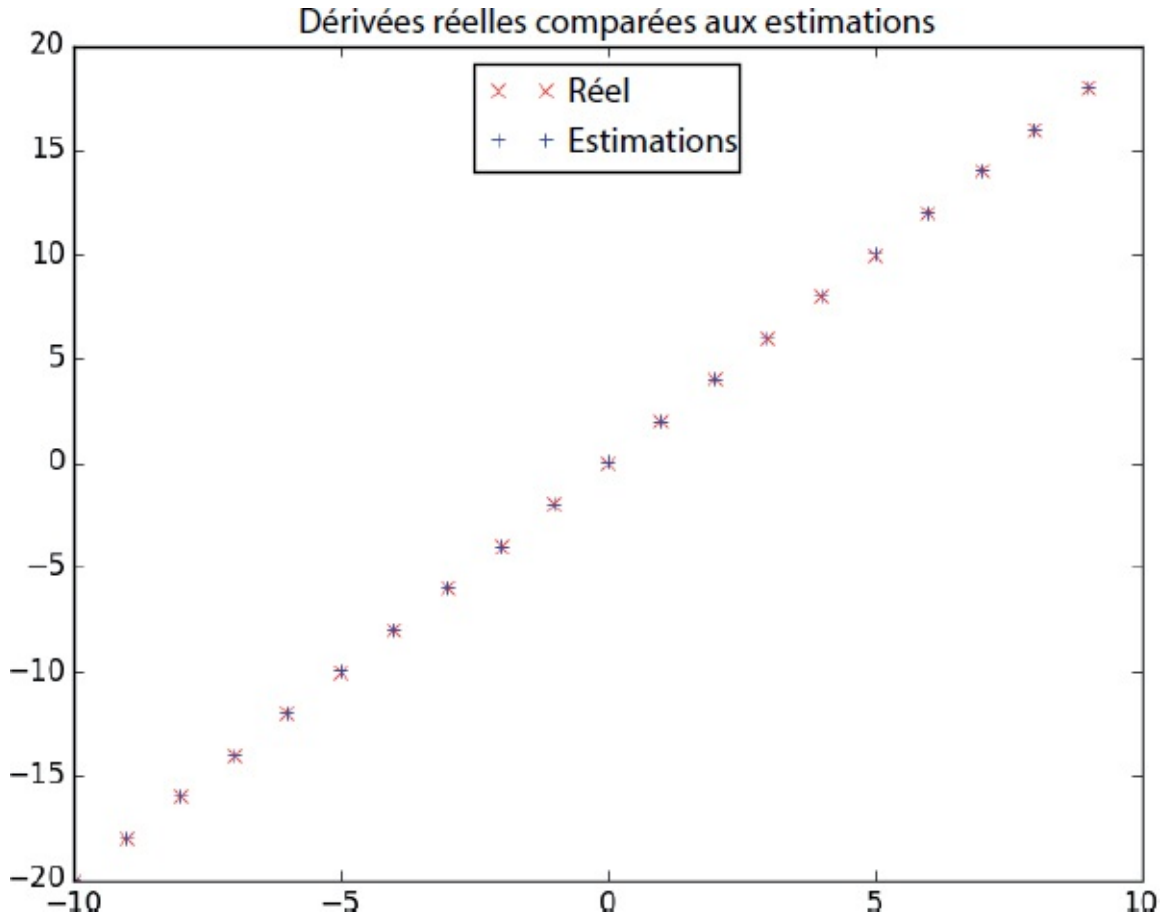
```
def derivative(x):
  return 2 * x
```

ce que vous pouvez vérifier (si vous en avez envie) en calculant véritablement le quotient des différences et en prenant la limite.

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 `e` très petite. La figure 8-3 montre le résultat d'une telle estimation :

```
derivative_estimate = partial(difference_quotient, square, h=0.00001)
# graphique pour prouver que ce sont fondamentalement les mêmes
import matplotlib.pyplot as plt
x = range(-10,10)
plt.title("Dérivées réelles comparées aux estimations")
plt.plot(x, map(derivative, x), 'rx', label='Actual')            # rouge x
plt.plot(x, map(derivative_estimate, x), '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* a plusieurs variables, elle a plusieurs dérivées partielles, chacune indiquant comment *f* change quand nous procédons à de petites modifications dans une seule des variables d'entrée.

Nous calculons sa $i^{ème}$ dérivée en la traitant comme une fonction à une seule variable *i*, tandis que les autres sont fixes :

```
def partial_difference_quotient(f, v, i, h):
  """calcule le i^ème quotient partiel des différences de f à 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
```

ce qui permet ensuite d'estimer le gradient de la même façon :

```
def estimate_gradient(f, v, h=0.00001):
  return [partial_difference_quotient(f, v, i, h)
          for i, _ in enumerate(v)]
```

Note

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 faites beaucoup de travail en trop.

L’utilisation du gradient

Il est facile de vérifier que la fonction `sum_of_squares` est la plus petite quand son entrée `v` est un vecteur de zéros. Mais imaginons que vous l’ignoriez. 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 atteignions un point où le gradient est très faible :

```
def step(v, direction, step_size):
  """se déplacer de step_size dans la direction v"""
  return [v_i + step_size * direction_i
          for v_i, direction_i in zip(v, direction)]

def sum_of_squares_gradient(v):
  return [2 * v_i for v_i in v]

# choisir un point de départ aléatoire
v = [random.randint(-10,10) for i in range(3)]

tolerance = 0.0000001

while True:
  gradient = sum_of_squares_gradient(v)  # calculer le gradient à v
  next_v = step(v, gradient, -0.01)      # effectuer un déplacement négatif
  if distance(next_v, v) < tolerance:    # s’arrêter en cas de convergence
    break
  v = next_v                             # continuer sinon
```

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 rapprochera 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 peut sembler parfaite, mais elle représente des calculs coûteux. Nous pouvons faire une approximation en essayant différentes tailles de déplacement et en choisissant celle qui donne pour résultat la plus petite valeur de la fonction objectif :

```
step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
```

Il est possible que certaines tailles de pas représentent des entrées invalides pour notre fonction. Nous devons donc créer une fonction « robuste » qui retourne l'infini (qui ne devrait jamais être la valeur minimum de quelque chose) pour les entrées invalides :

```
def safe(f):
  """retourne une nouvelle fonction qui est la même que f,
  à la différence près qu'elle renvoie l'infini quand f produit une erreur"""
  def safe_f(*args, **kwargs):
    try:
        return f(*args, **kwargs)
    except:
        return float('inf')  # signifie "infini" en Python
  return safe_f
```

Synthèse

Dans le cas général, nous avons une fonction cible `target_fn` que nous voulons minimiser et nous avons aussi son gradient `gradient_fn`. Par exemple, `target_fn` peut représenter les erreurs d'un modèle sous la forme d'une fonction de ses paramètres et nous cherchons alors les paramètres pour lesquels les erreurs sont aussi petites que possible.

De plus, supposons que nous avons choisi (d'une manière ou d'une autre) une valeur de départ pour les paramètres `theta_0`. Alors nous pouvons implémenter la descente de gradient :

```
def minimize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
  """utilise la descente du gradient pour trouver le theta qui minimise la fonction
  cible"""

step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

theta = theta_0               # initialise theta
target_fn = safe(target_fn)   # version sûre de target_fn
value = target_fn(theta)      # valeur que nous minimisons

while True:
  gradient = gradient_fn(theta)
  next_thetas = [step(theta, gradient, -step_size)
                 for step_size in step_sizes]

  # choisir celui qui minimise la fonction d'erreur
  next_theta = min(next_thetas, key=target_fn)
  next_value = target_fn(next_theta)

  # arrêter en cas de « convergence »
  if abs(value - next_value) < tolerance:
    return theta
  else:
    theta, value = next_theta, next_value
```

Nous avons appelé la fonction `minimize_batch` parce que pour chaque déplacement sur le gradient, elle examine le jeu de données complet (car `target_fn` retourne l'erreur sur l'ensemble du jeu de données). Dans la section suivante, nous ferons connaissance avec une autre méthode qui examine une seule donnée élémentaire à chaque fois.

Parfois, au contraire, nous voulons maximiser une fonction, ce qui peut se faire en minimisant sa négation (qui a un gradient correspondant négatif) :

```
def negate(f):
```

```
  """retourne une fonction qui renvoie -f(x) pour toute entrée x"""
  return lambda *args, **kwargs: -f(*args, **kwargs)

def negate_all(f):
  """la même chose quand f retourne une liste de nombres"""
  return lambda *args, **kwargs: [-y for y in f(*args, **kwargs)]

def maximize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
  return minimize_batch(negate(target_fn),
                        negate_all(gradient_fn),
                        theta_0,
                        tolerance)
```

La descente du gradient stochastique

Ainsi que nous l'avons déjà mentionné, nous utiliserons souvent la descente de gradient pour choisir les paramètres d'un modèle de manière à minimiser une forme d'erreur. Avec la précédente approche batch, chaque étape de gradient nous imposait de faire une prévision et de calculer le gradient pour l'ensemble des données, ce qui allonge chaque étape.

Or, en général, les fonctions d'erreurs sont additives, ce qui signifie que l'erreur d'ensemble prévue sur l'ensemble des données est la somme des erreurs prévues pour chaque donnée élémentaire.

Quand c'est le cas, nous pouvons appliquer à la place une technique dite « descente du gradient stochastique » qui revient à calculer le gradient (et à se déplacer d'un pas) pour un point à la fois. Le traitement boucle sur nos données jusqu'à ce qu'il atteigne un point d'arrêt.

Pendant chaque cycle, nous voulons que l'itération se fasse dans un ordre aléatoire :

```
def in_random_order(data):
  """générateur qui retourne les éléments de données dans un ordre aléatoire"""
  indexes = [i for i, _ in enumerate(data)]  # créer une liste d'index
  random.shuffle(indexes                     # les permuter
  for i in indexes:                          # retourne les données dans cet ordre
    yield data[i]
```

Et nous voulons le gradient pour chaque donnée élémentaire. Avec cette approche, il existe un risque de boucler sans fin à l'approche d'une valeur minimale. Donc, dès que le résultat cesse de s'améliorer, il faut diminuer la taille du pas et éventuellement arrêter :

```
def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):

  data = zip(x, y)
  theta = theta_0                            # prédiction initiale
  alpha = alpha_0                            # valeur de déplacement initiale
  min_theta, min_value = None, float("inf")  # le minimum jusqu'à présent
  iterations_with_no_improvement = 0

  # si on effectue 100 itérations sans constater d'amélioration, arrêter
  while iterations_with_no_improvement < 100:
    value = sum( target_fn(x_i, y_i, theta) for x_i, y_i in data )

    if value < min_value:
      # si on a trouvé un nouveau minimum, le conserver
      # et revenir à la valeur de déplacement d'origine
      min_theta, min_value = theta, value
```

```
      iterations_with_no_improvement = 0
      alpha = alpha_0
    else:
      # sinon il n'y a pas d'amélioration, il faut donc diminuer la valeur de déplacement
      iterations_with_no_improvement += 1
      alpha *= 0.9

      # et prendre un déplacement de la valeur du gradient pour chaque donnée isolée
      for x_i, y_i in in_random_order(data):
        gradient_i = gradient_fn(x_i, y_i, theta)
        theta = vector_subtract(theta, scalar_multiply(alpha, gradient_i))

  return min_theta
```

La version stochastique est généralement beaucoup plus rapide que l'approche batch. Évidemment, nous voulons aussi une version qui maximise :

```
def maximize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):
  return minimize_stochastic(negate(target_fn),
                             negate_all(gradient_fn),
                             x, y, theta_0, alpha_0)
```

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* semble plus agréable que les manuels que j'ai utilisés pour mon apprentissage.
- scikit-learn possède un module « Stochastic Gradient Descent » qui est moins généraliste que le nôtre par certains points et plus généraliste par d'autres. Cependant, il est vrai que dans la plupart des cas réels, vous utiliserez des bibliothèques qui prennent déjà en compte la gestion des problématiques d'optimisation et vous n'aurez pas de souci à vous faire (à part quand ça ne marchera pas correctement, ce qui se produira inévitablement un jour ou l'autre).

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 de la science des données a besoin de données. Il faut bien le reconnaître, un data scientist comme vous va passer l'essentiel de son temps à acquérir, nettoyer et transformer des données. En un mot, vous pouvez toujours taper les données vous-même (ou confier la frappe à vos assistants dévoués si vous en avez), mais en général ce n'est pas une bonne utilisation 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 textes 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 le regex spécifié pour la ligne de commande
regex = sys.argv[1]

# pour chaque ligne passée dans le script
for line in sys.stdin:
    # si elle correspond au 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 le nombre de lignes d'un fichier qui contiennent des nombres. Sous Windows, vous écrirez :

```
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
```

Le | est le caractère *pipe*, qui signifie « utiliser le résultat de la commande de gauche comme entrée de la commande de droite ».

Ce caractère permet de construire des pipelines de traitement de données assez élaborés.

Note

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 un peu plus de travail.

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             #
                  for word in line.strip().split() # séparer à partir des espaces
                  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 cela :

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193  the
51380  and
34753  of
13643  to
12799  that
12560  in
10263  he
9840   shall
8987   unto
8836   for
```

Note

Si vous êtes un programmeur Unix expérimenté, vous connaissez certainement un grand nombre d'outils de la ligne de commande (comme egrep) présents nativement dans votre système d'exploitation et qu'il convient d'utiliser, plutôt que de tout réinventer. Mais c'est une bonne chose de savoir ce qui est possible, au cas où.

La 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.

Les fondamentaux des fichiers texte

La première étape consiste à obtenir un objet permettant de traiter le fichier à l'aide de `open` :

```
# 'r' signifie lecture seule (read-only)
file_for_reading = open('reading_file.txt', 'r')

# 'w' signifie écrire (write) – détruit le fichier s'il existait déjà
file_for_writing = open('writing_file.txt', 'w')

# 'a' signifie ajouter à la fin du fichier existant (append)
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,'r') 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 à travers les lignes du fichier dans une boucle `for` :

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in f:                  # examine chaque ligne de fichier
        if re.match("^#",line):     # utilise un regex pour voir s'il commence par '#'
            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 avez un fichier d'adresses de messagerie,

une par ligne, et que vous voulez générer un histogramme des domaines. Pour extraire correctement les domaines, il existe des règles un peu subtiles (par exemple, la *Public Suffix List*), mais en première approximation vous pouvez prendre la partie de l'adresse qui se situe après le caractère `@` (ce qui donnera un mauvais résultat pour les adresses du type `joel@mail.datasciencester.com`).

```
def get_domain(email_address):
    """couper au niveau de '@' et retourner le dernier morceau"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Les fichiers à délimiteur

L'hypothétique fichier d'adresses que nous venons de traiter comportait une adresse par ligne. La plupart du temps, nos fichiers comportent de nombreuses données sur chaque ligne. Ces fichiers utilisent alors la virgule ou le caractère de tabulation comme séparateur. Chaque ligne comporte plusieurs champs, avec une virgule (ou un caractère de tabulation) pour indiquer la fin d'un champ et le début du suivant.

Les choses se compliquent quand certains champs contiennent eux-mêmes des virgules et des caractères de début de ligne (ce qui est inévitable). Pour cette raison, c'est presque toujours une mauvaise idée de vouloir analyser la syntaxe du contenu vous-même. À la place, utilisez plutôt le module csv de Python (ou la bibliothèque pandas). Pour des raisons techniques dont vous pouvez attribuer l'entière responsabilité à Microsoft si vous le voulez, vous devez toujours travailler en mode binaire avec les fichiers csv. Il suffit, pour cela, d'inclure un `b` après le `r` ou `w` (voir la communauté de programmeurs *Stack Overflow*).

Si votre fichier n'a pas d'en-tête (ce qui veut dire que vous voulez que chaque ligne soit une liste, ce qui vous impose de savoir ce que renferme chaque colonne), vous pouvez utiliser `csv.reader` pour parcourir les lignes, chacune d'entre elles devenant une liste éclatée.

Par exemple, si nous avons un fichier de cours d'actions délimité 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', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

Si votre fichier a 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
```

vous pouvez soit sauter la ligne d'en-tête (avec un appel initial de `reader.next()`), soit récupérer chaque ligne comme un dict (dont les en-têtes sont les clés) à l'aide de `csv.DictReader` :

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
  reader = csv.DictReader(f, delimiter=':')
  for row in reader:
    date = row["date"]
    symbol = row["symbol"]
    closing_price = float(row["closing_price"])
    process(date, symbol, closing_price)
```

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. Nous pouvons écrire de même des données délimitées à l'aide de `csv.writer` :

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        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 :

```
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 sans doute un fichier qui ressemble à cela :

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

et dont personne ne pourra rien tirer.

Le ratissage du Web

Une autre manière de récupérer des données consiste à les obtenir de pages web. Récupérer des pages web est assez facile ; en extraire des informations significatives et structurées est beaucoup plus difficile.

L'analyse syntaxique du HTML

Sur la Toile, les pages sont écrites en HTML. Le texte est normalement qualifié comme représentant des éléments ou leurs 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 idéal, 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 n'est pas très bien écrit en général, 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, qui construit un arbre à partir des différents éléments d'une page web et fournit une interface simple pour y accéder. Au moment où j'écris ce livre, la dernière version est Beautiful Soup 4.3.2 (`pip install beautifulsoup4`), c'est celle que nous allons utiliser. Nous utiliserons aussi la bibliothèque de récupération d'URL requests (à l'aide de `pip install requests`), qui offre des fonctionnalités plus conviviales que tous les outils natifs de Python.

Le parseur par défaut de Python n'est pas très permissif, ce qui 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 faut installer :

```
pip install html5lib
```

Pour utiliser Beautiful Soup, nous devons passer le code HTML à la fonction

`BeautifulSoup()`.

Dans nos exemples, ce sera le résultat d'un appel à `requests.get` :

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

Ensuite, il suffira d'utiliser quelques méthodes simples.

Nous ferons usage des objets Tags qui correspondent aux balises représentant la structure d'une page HTML.

Par exemple, pour trouver la première balise `<p>` (et son contenu), vous pouvez écrire :

```
first_paragraph = soup.find('p')  # ou simplement soup.p
```

Vous pouvez obtenir le texte associé à une balise à partir de la propriété du texte :

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

Et vous pouvez extraire les attributs d'une balise en la traitant comme un dict :

```
first_paragraph_id = soup.p['id']       # lance l'erreur KeyError si n'est pas 'id'
first_paragraph_id2 = soup.p.get('id')  # retourne None si n'est pas '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 avec 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 actions simples pour élaborer des stratégies plus complexes. Par exemple, si vous cherchez tous les éléments `<span>` qui contiennent un élément `<div>` :

```
# attention, retourne le même spam plusieurs fois de suite
# s'il est présent dans plusieurs div
```

```
# soyez plus malin dans ce cas
spans_inside_divs = [span
                     for div in soup('div')    # pour chaque <div> sur la page
                     for span in div('span')]  # trouver chaque <span> à l'intérieur
```

Ces quelques fonctionnalités nous permettront de faire beaucoup de choses. Si vous avez besoin de requêtes plus complexes, ou si vous êtes simplement curieux, consultez la documentation.

Il va de soi que les données importantes ne seront certainement pas labellisées 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.

Exemple : les ouvrages consacrés aux données

Un investisseur potentiel intéressé par DataSciencester pense que les données sont une simple mode éphémère. Pour lui prouver qu'il a tort, vous décidez de regarder combien l'éditeur O'Reilly (cela vaut également pour Eyrolles) a publié de livres consacrés aux données. Après avoir exploré son site web, vous constatez qu'il existe de nombreux livres consacrés aux données (ainsi que des vidéos), le tout accessible dans des pages web contenant 30 articles à la fois avec des URL de la forme : http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicationDate&page=1.

À moins de vous comporter comme un voyou (au risque de vous faire bannir pour vos méthodes musclées de récupération de données), avant d'extraire un site, vous devez d'abord vérifier s'il existe des règles d'accès.

Un coup d'œil à http://oreilly.com/terms/ vous apprend que rien ne s'oppose à votre projet.

En bons citoyens, nous vérifierons également le fichier `robots.txt`, qui dicte leur comportement aux robots explorateurs. Les lignes importantes du fichier `http://shop.oreilly.com/robots.txt` sont les suivantes :

```
Crawl-delay:  30
Request-rate: 1/30
```

La première ligne nous indique que nous devons attendre 30 secondes entre deux requêtes, la seconde que nous ne devons demander qu'une page toutes les 30 secondes. En fait, il s'agit de deux façons différentes de dire la même chose. (Il existe d'autres lignes précisant les répertoires à ne pas explorer, mais comme il ne s'agit pas de notre URL, pas de problème.)

Pour comprendre comment extraire les données, téléchargeons une de ces pages et envoyonsla à Beautiful Soup :

```
# il n'est pas nécessaire de couper l'URL ainsi sauf s'il faut qu'elle tienne
# dans un livre
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

Si vous regardez le code source de cette page (dans votre navigateur, faites un clic droit et choisissez « Voir le code source » ou « Voir la page source » ou toute autre formulation équivalente) vous verrez que chaque livre (ou vidéo) est contenu seul dans une cellule d'un tableau `<td>` dont la classe est `thumbtext`.

Voici une version abrégée du code HTML pour un livre :

```
<td class="thumbtext">
  <div class="thumbcontainer">
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummies</a>
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">December 2014</span>
    <div style="clear:both;">
      <div id="146 350">
        <span class="pricelabel">
                          Ebook :

                          <span class="price"> $29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

Pour commencer, cherchons toutes les balises élémentaires de `td thumbtext` :

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

Ensuite, nous aimerions éliminer les vidéos. (Le futur investisseur ne s'intéresse qu'aux livres.) Si nous inspectons le code HTML de plus près, nous voyons que chaque `td` contient un ou plusieurs éléments dont la classe est `pricelabel` et dont le texte ressemble à `Ebook:` ou `Video:` ou `Print:`.

Il apparaît que les vidéos contiennent un seul `pricelabel` dont le texte commence par `Video` (une fois enlevés les caractères espace du début). On pourrait appliquer le test suivant pour les vidéos :

```
def is_video(td):
    """c'est une vidéo si elle a exactement un label de prix et si
    le texte nettoyé à l'intérieur de ce label commence par 'Video'"""
    pricelabels = td('span', 'pricelabel')
    return (len(pricelabels) == 1 and
            pricelabels[0].text.strip().startswith("Video"))


print len([td for td in tds if not is_video(td)])
# 21 pour moi, sans doute différent pour vous
```

Nous sommes maintenant prêts à récupérer les données des éléments du tableau `td`.

Il semble que le titre du livre se trouve dans le texte entre les balises `<a>` à l'intérieur de la classe `<div class="thumbheader">` :

```
title = td.find("div", "thumbheader").a.text
```

Les auteurs figurent dans le texte `AuthorName <div>`. Ils sont précédés de `By` (dont nous allons nous débarrasser) et séparés par des virgules (que nous utiliserons comme séparateurs, puis nous devrons nous débarrasser des espaces) :

```
author_name = td.find('div', 'AuthorName').text
authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

L'ISBN semble faire partie du lien dans `thumbheader <div>` :

```
isbn_link = td.find("div", "thumbheader").a.get("href")


# re.match capture la partie du regex entre parenthèses
isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

Et la date est juste le contenu de la classe `<span class="directorydate">` :

```
date = td.find("span", "directorydate").text.strip()
```

Intégrons toutes ces actions dans une fonction :

```
def book_info(td):
    """soit une balise BeautifulSoup <td> représentant un livre,
    extraire les détails du livre et retourner un dict"""


    title = td.find("div", "thumbheader").a.text
    by_author = td.find('div', 'AuthorName').text
```

```
    authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
    isbn_link = td.find("div", "thumbheader").a.get("href")
    isbn = re.match("/product/(.*)\.do",    isbn_link).groups()[0]
    date = td.find("span", "directorydate").text.strip()

    return {
      "title" : title,
      "authors" : authors,
      "isbn" : isbn,
      "date" : date
    }
```

Nous sommes prêts à récupérer les données :

```
from bs4 import BeautifulSoup
import requests
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31    # quand j'ai écrit le livre, probablement davantage à ce jour

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

    # en bons citoyens, respectons les consignes de robots.txt!
    sleep(30)
```

Note

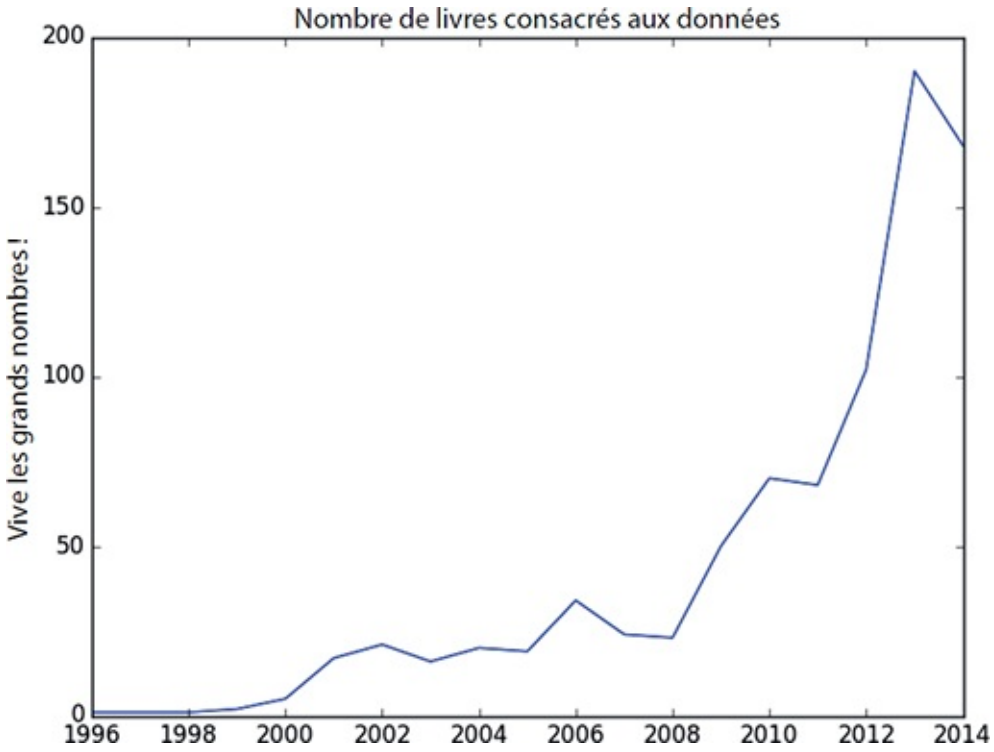
Ce genre de méthode d'extraction de données HTML s'apparente davantage à l'art qu'à la science. Il existe quantité d'autres démarches possibles pour trouver les livres et les titres qui sont certainement tout aussi efficaces.

Maintenant que nous avons les données, nous pouvons afficher le nombre de livres publiés chaque année (figure 9-1) :

```
def get_year(book):
   """book["date"] ressemble à 'November 2014'
   il faut couper au niveau de l'espace et prendre la deuxième partie"""
   return int(book["date"].split()[1])

# 2014 est la dernière année complète (quand j'ai exécuté ce code)
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
```

```
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
plt.ylabel("Nombre de livres consacrés aux données")
plt.title("Vive les grands nombres !")
plt.show()
```

Figure 9–1
Nombre de livres consacrés aux données par an



Malheureusement, notre investisseur a examiné le graphique et a conclu que 2013 marquait le pic de l’intérêt pour les données.

L’utilisation des API

De nombreux sites et services en ligne proposent des interfaces de programmation (API) qui vous permettent de passer des requêtes explicites dans un format structuré. Il est 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 JavaScript Object Notation (JSON).

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" : 2014,
   "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 désérialise une chaîne de caractères représentant un objet JSON en objet Python :

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2014,
                  "topics" : [ "data", "science", "data science"] }"""
# parse le JSON pour créer un dict Python
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
    print deserialized
```

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 comme nous l'avons fait pour les récupérer à partir du HTML. Pour en savoir plus, reportez-vous à la documentation.

L'utilisation d'une API non authentifiée

La plupart des 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 GitHub qui permet de faire des choses simples sans authentification.

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/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. (N'hésitez pas à substituer votre nom d'utilisateur et à prendre votre propre répertoire GitHub. Au fait, vous avez bien un compte GitHub, non ?)

Utilisons-le pour déterminer quels mois et quels jours de la semaine j'ai le plus de chances de créer un répertoire. Le seul problème, c'est que les dates dans la réponse sont des chaînes de caractères (Unicode) :

```
u'created_at': u' 2013-07-05T02:02:28Z'
```

Comme Python ne fournit pas par défaut d'analyseur de dates sérieux, il faut en installer un :

```
pip install python-dateutil
```

Seule la fonction `dateutil.parser.parse` vous sera sans doute réellement utile :

```
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 langues de mes cinq derniers répertoires :

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["created_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 niveau si bas qu'il impose de faire les requêtes et d'analyser les réponses nous-mêmes. Un des avantages de Python, c'est que pour la plupart des API qui vous intéressent quelqu'un a déjà développé une bibliothèque de modules d'accès. Quand elles sont bien écrites, ces bibliothèques peuvent vous épargner bien des errements au moment de mettre au point les détails d'accès. (Si elles n'ont pas été bien écrites, ou s'il s'avère qu'elles sont basées sur des versions obsolètes des API, 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 pour vous de connaître quelques détails.

À la recherche des API

Si vous avez besoin de données d'un site en particulier, recherchez une section réservée aux développeurs ou aux API sur le site, et faites une recherche plus générale sur le Web avec la requête « python API » pour trouver une bibliothèque.

Il existe une API Roten Tomatoes pour Python. (NDT : célèbre site américain de critiques de films qui utilise un système de notation.) Il existe de nombreuses bibliothèques d'encapsulation d'API en Python, par exemple pour Kloit, l'API Yelp, l'API IMDB, etc.

Si vous cherchez une liste des API qui ont des bibliothèques Python associées, les deux adresses sont Python API et Python for beginners.

Si vous cherchez un répertoire plus général d'API (qui ne contient pas nécessairement des bibliothèques Python), Programmable Web est une ressource recommandée qui comprend un répertoire d'API classées par catégories.

Et si vous n'avez toujours rien trouvé qui convient à vos besoins, il reste l'extraction de site web, l'ultime recours du data scientist.

Exemple : l'utilisation de l'API Twitter

Twitter est une formidable source de données. Vous pouvez l'utiliser pour obtenir des informations en temps réel. Vous pouvez l'utiliser pour mesurer des réactions à des événements récents en train de se produire. Vous pouvez l'utiliser pour 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.

Vous devrez installer la bibliothèque Twython pour accéder à l'API Twitter (`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). 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. (Même si elles ont déjà changé une fois depuis que j'ai écrit ce livre, alors bonne chance !)

1. Rendez-vous à l'adresse https://apps.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 *Créer nouvelle app*.
4. Donnez-lui un nom (comme `Data Science`) et une description, ainsi qu'une adresse d'URL en guise de site web (peu importe laquelle).
5. Acceptez les conditions générales du service et cliquez sur *Créer*.
6. Notez le couple clé/secret de l'utilisateur.
7. Cliquez sur *Créer mon jeton d'accès*.
8. Notez le jeton d'accès et le secret associé (vous devrez peut-être rafraîchir la page).

Le couple clé/secret de l'utilisateur informe Twitter sur l'application qui

accède à ses API, tandis que le jeton et son secret informent sur la personne qui est en train d'accéder à l'API. Si vous avez déjà utilisé votre compte Twitter pour vous connecter à un autre site, la page *Cliquer pour autoriser* a généré un jeton d'accès confié à ce site pour convaincre Twitter que c'est bien vous (ou du moins quelqu'un qui agit en votre nom). Comme nous n'avons pas besoin de cette fonctionnalité *Autoriser n'importe qui à se connecter*, nous pouvons utiliser le jeton d'accès à génération statique et le secret du jeton d'accès.

Attention

Les ensembles clé/secret du consommateur et clé/secret du jeton doivent être considérés comme des *mots de passe*. Vous ne devez pas les partager, les publier, ni les stocker dans votre répertoire 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`.

L'utilisation de Twython

Nous allons d'abord utiliser l'API Search, qui ne demande que le couple clé/secret consommateur, mais pas le jeton :

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# recherche les tweets contenant la phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

Note

Le code `.encode("utf-8")` est nécessaire pour prendre en compte le fait que les tweets contiennent souvent des caractères Unicode non gérés par `print`. (Si vous l'abandonnez, vous déclencherez sans doute une erreur `UnicodeEncodeError`.) Il est à peu près certain qu'à un moment ou un autre de votre carrière dans la data science, vous serez confronté à des problèmes Unicode sérieux, ce qui vous forcera à consulter la documentation Python ou à passer à Python 3, qui est bien plus performant avec le texte Unicode.

Si vous exécutez le code proposé, vous devriez récupérer des tweets tels que :

```
haithemnyc: Data scientists with the technical savvy &amp;amp; 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 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):
  """notre propre sous-classe de TwythonStreamer qui spécifie comment interagir avec le
stream"""

  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 aux tweets en anglais
    if data['lang'] == 'en':
       tweets.append(data)
       print "received tweet #", len(tweets)

    # arrêter quand il y assez de tweets
    if len(tweets) >= 1000:
      self.disconnect()

  def on_error(self, status_code, data):
    print status_code,
    data self.disconnect()
```

`MyStreamer` va se connecter au flux Twitter et attendre que Twitter lui fournisse des données.

Chaque fois qu'il reçoit des données (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 et déconnecte le streamer après avoir collecté 1 000 tweets. 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
# publics stream.statuses.sample()
```

L'exécution se poursuit jusqu'à collecter 1 000 tweets (ou jusqu'à rencontrer une erreur), et ensuite c'est l'arrêt qui vous permet d'analyser ces tweets.

Par exemple, vous pouvez trouver les mots dièses *(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 API Twitter.

Note

Dans le cadre d'un projet réel en général, 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 est la bibliothèque de référence de la data science pour travailler avec des données (et les importer).
- Scrapy est une bibliothèque très riche pour construire des scrapers du Web plus complexes qui sont capables, par exemple, de suivre des liens inconnus.

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. Jusque-là, nous avons surtout parlé de science, aussi dans ce chapitre nous allons parler de l'art.

L’exploration des données

Après avoir identifié les questions auxquelles vous cherchez des réponses et mis la main sur quelques données, la tentation est forte de plonger tout de suite dans le vif du sujet et de construire des modèles pour en tirer des réponses. Vous devez résister à la tentation, car votre première étape doit être l’exploration des données.

Explorer des données à une dimension

Le cas le plus simple correspond à un ensemble de données à une seule dimension, soit 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 le nombre de pages de chaque livre consacré à la data science dans votre bibliothèque.

La première étape consiste évidemment à réaliser quelques statistiques de synthèse. Vous voulez savoir combien vous avez de données élémentaires, quelle est la plus petite, la plus grande, l’espérance et l’écart-type.

Mais tout ceci ne vous aidera pas forcément à comprendre vos données. À l’étape suivante, vous devrez donc créer un histogramme ; vous devrez répartir les données en tranches distinctes et compter combien de données élémentaires se retrouvent dans chaque tranche :

```
def bucketize(point, bucket_size):
    """dirige le point vers le prochain plus petit multiple de bucket_size"""
    return bucket_size * math.floor(point / bucket_size)


def make_histogram(points, bucket_size):
    """sépare les points et compte le nombre de points dans chaque tranche"""
    return Counter(bucketize(point, bucket_size) for point in points)


def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
    plt.show()
```

Par exemple, soit les deux jeux de données suivants :

```
random.seed(0)


# uniforme entre -100 et 100
uniform = [200 * random.random() - 100 for _ in range(10 000)]
```

```
# distribution normale avec espérance 0 et écart-type 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(100 000):
```

Dans les deux cas, l'espérance est proche de 0 et l'écart-type de 58. Pourtant, les deux distributions sont très différentes. La figure 10-1 illustre la distribution uniforme :

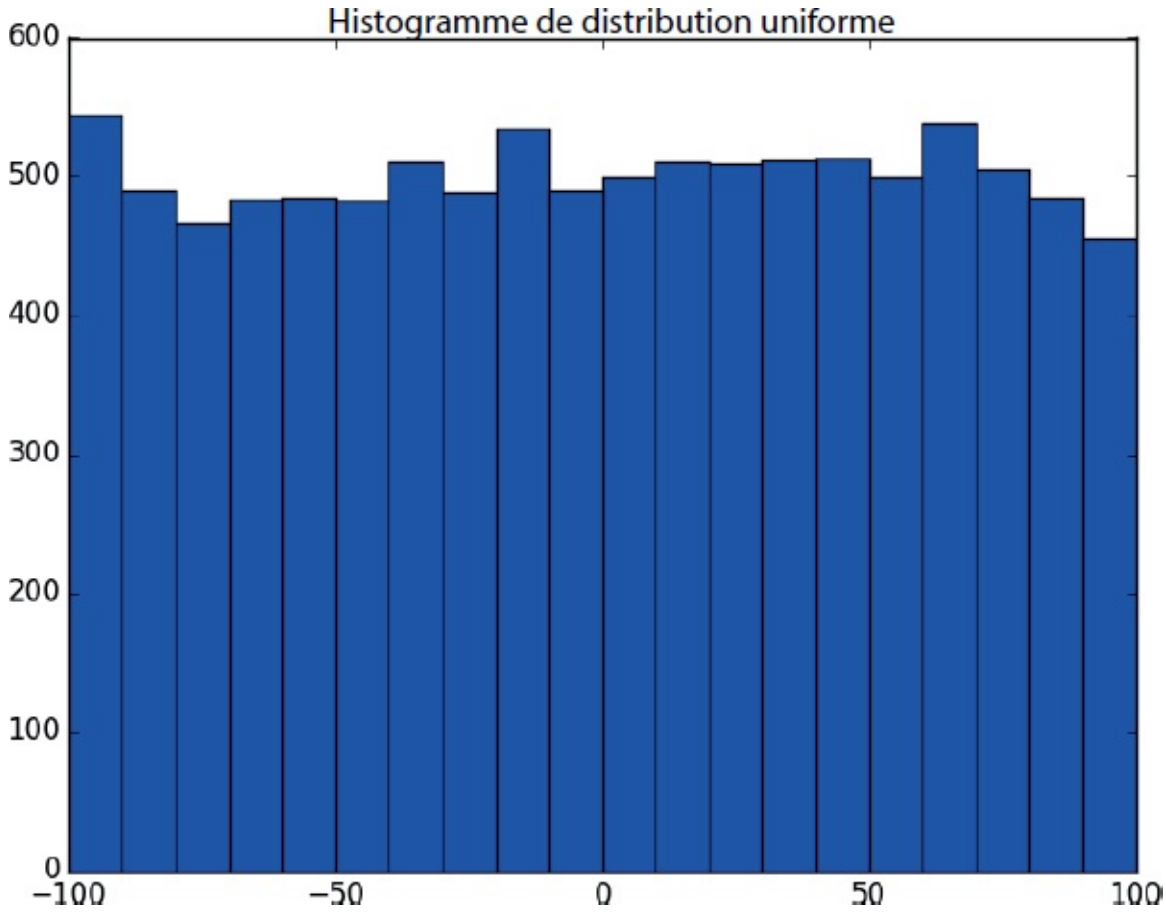
```
plot_histogram(uniform, 10, "Uniform Histogram")
```

alors que la figure 10-2 illustre la distribution normale :

```
plot_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 *comment* elles diffèrent.

Figure 10–1
Histogramme de distribution uniforme



Deux dimensions

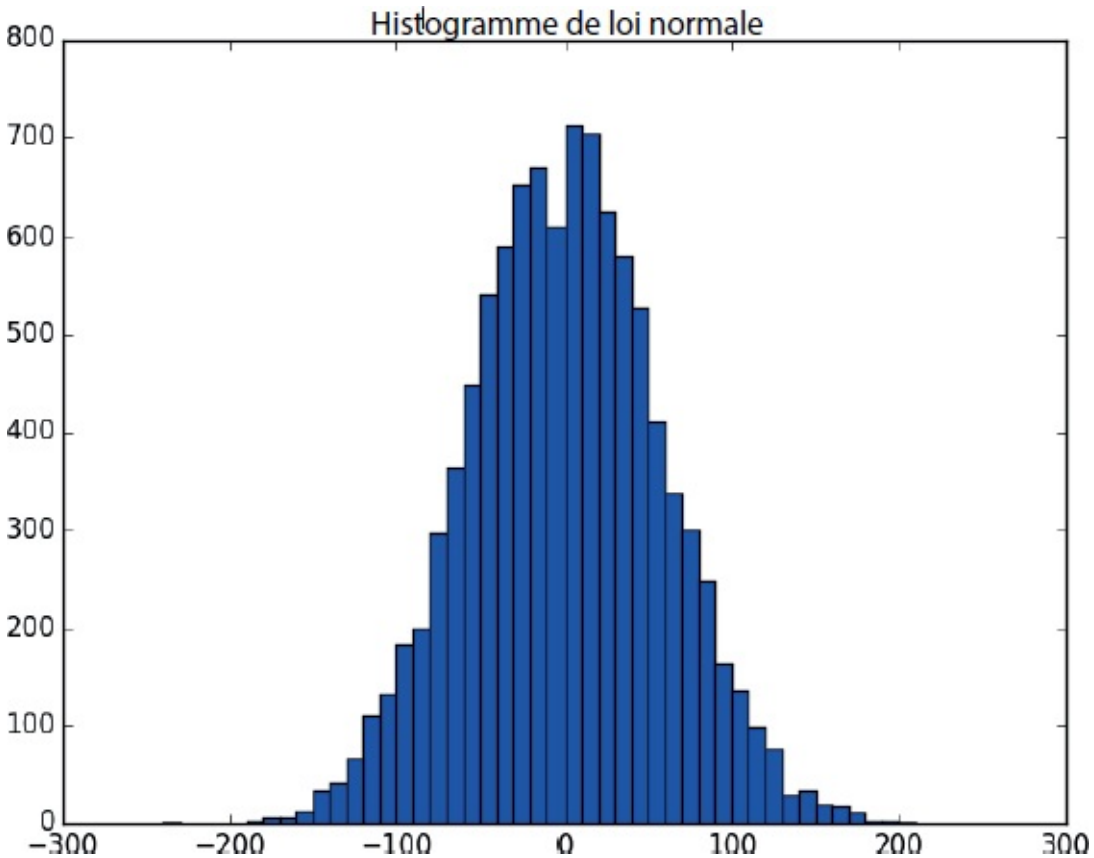
Imaginons maintenant que votre jeu de données a deux dimensions. Par exemple, en plus de la durée quotidienne en minutes de présence sur le site, vous avez les années d'expérience dans la data science par utilisateur. Bien sûr, vous voulez comprendre chaque dimension individuellement. Mais il est probable que vous vouliez surtout examiner la dispersion des données.

Par exemple, soit un autre jeu de données fictif :

```
def random_normal():
 """retourne un graphique aléatoire à partir d'une distribution normale"""
  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).

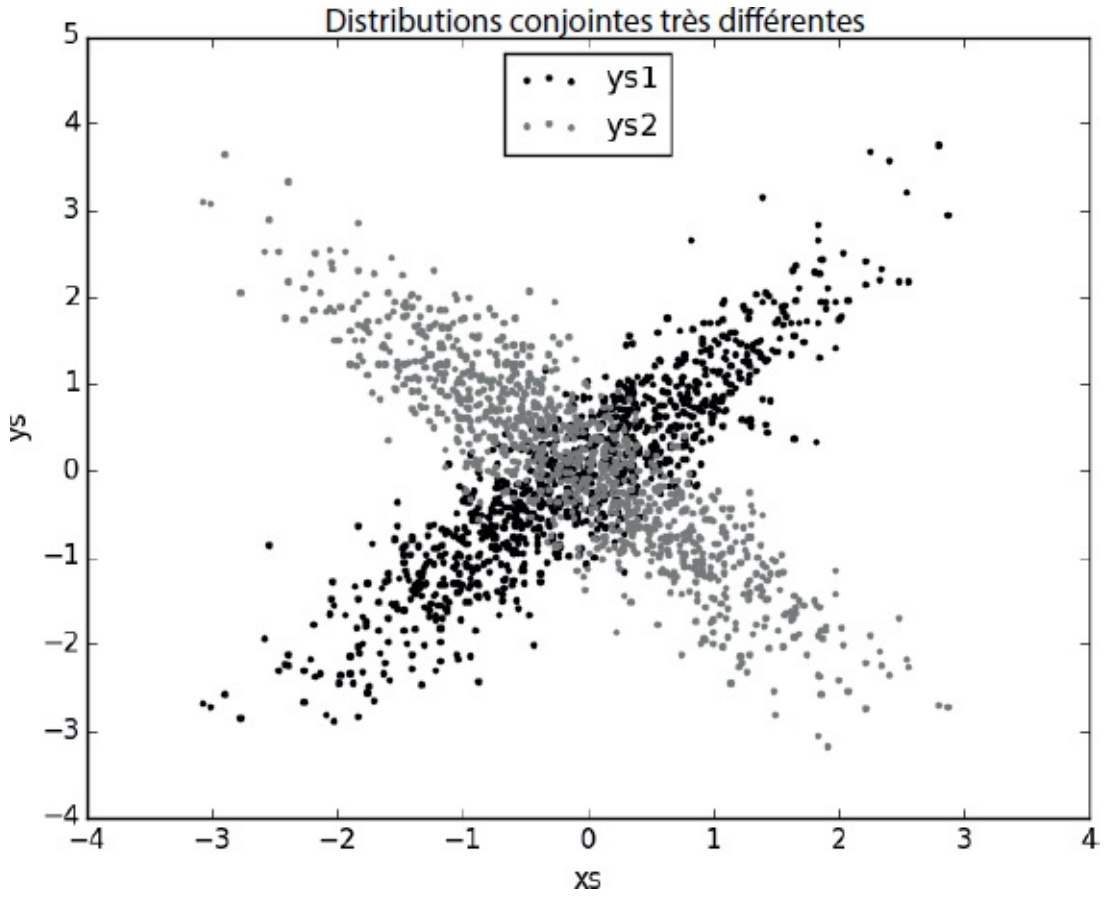
Figure 10–2
Histogramme de loi normale



Mais chacun d'entre eux a une distribution conjointe avec `xs` très différente, comme le montre la 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 :

```
print correlation(xs, ys1) # 0.9
print correlation(xs, ys2) # -0.9
```

Plusieurs dimensions

En présence de plusieurs dimensions, la question est de savoir comment ces dimensions 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 :

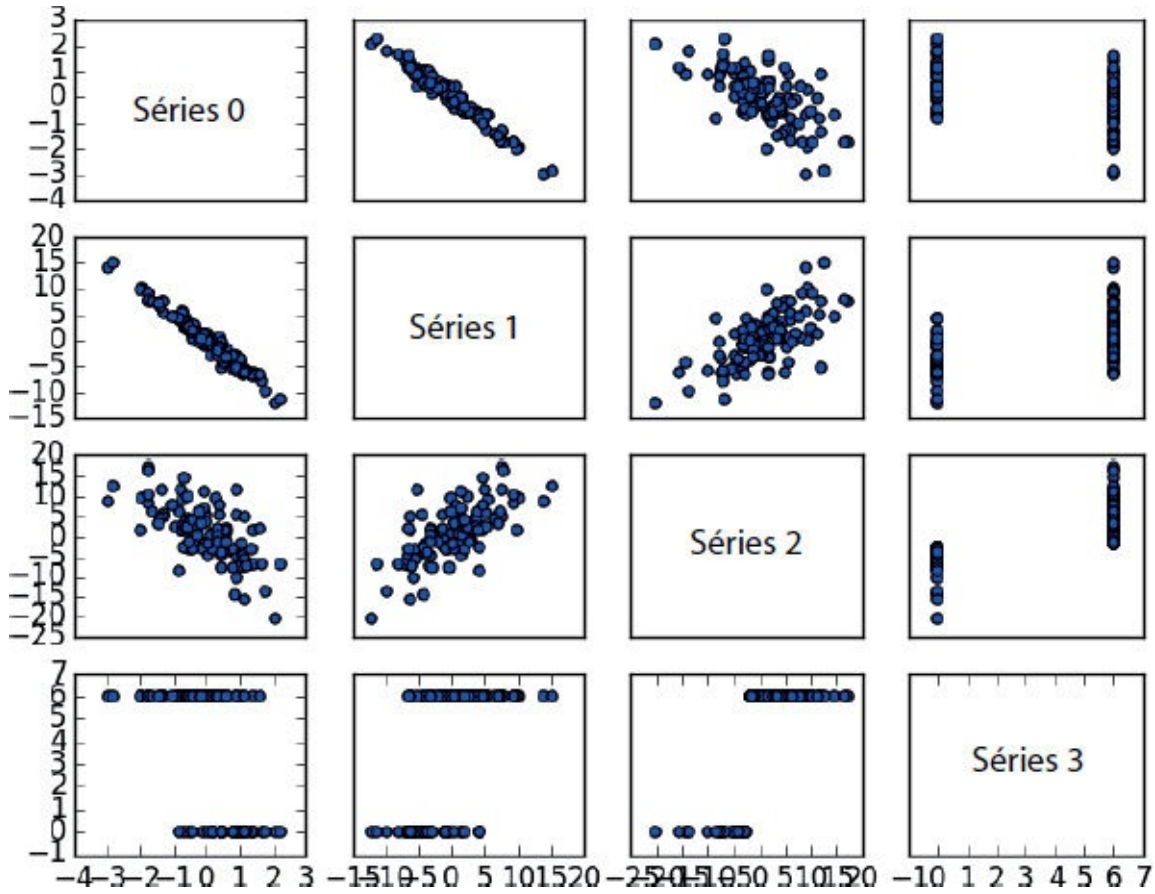
```
def correlation_matrix(data):
    """retourne la matrice num_columns x num_columns dont l'entrée de rang (i, j) est la
corrélation entre les colonnes i et j des données"""

    _, num_columns = shape(data)

    def matrix_entry(i, j):
        return correlation(get_column(data, i), get_column(data, j))
```

```
    return make_matrix(num_columns, num_columns, matrix_entry)
```

Pour une approche plus visuelle (si vous n'avez pas trop de dimensions), construisez une matrice de nuage de points (figure 10-4) pour illustrer tous les nuages correspondant à deux dimensions. Il faut utiliser `plt.subplots()` qui nous permet de créer des sous-fenêtres graphiques de 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) :

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

    # variation de column_j sur l'axe des x versus vs column_i sur l'axe des y
    if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

    # à moins que i == j, auquel cas montrer le nom de series
    else: ax[i][j].annotate("series " + 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_columns - 1: ax[i][j].xaxis.set_visible(False)
          if j > 0: ax[i][j].yaxis.set_visible(False)

# corrige les labels des axes en bas à droite et en haut à gauche, qui sont faux, car
# leurs 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 nuage de points



En examinant les nuages de points, vous pouvez voir que series 1 est très négativement corrélée avec series 0, series 2 est corrélée positivement avec series 1, et series 3 ne prend que les valeurs 0 et 6, 0 correspondant aux petites valeurs de series 2 et 6 correspondant aux 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).

Nettoyage et transformation

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. Il convient de convertir les chaînes de caractères en numériques à virgule flottante ou en entiers, avant de pouvoir les utiliser. C'est ce que nous avons fait jusqu'ici avant toute utilisation des données :

```
closing_price = float(row[2])
```

Mais il y a probablement moins de risques d'erreurs à effectuer l'analyse à la lecture, en créant pour cela une fonction encapsulant `csv.reader`. Nous lui apporterons une liste de parseurs, chacun spécifiant comment analyser les colonnes. Et nous utiliserons `None` pour spécifier « ne pas toucher à cette colonne » :

```
def parse_row(input_row, parsers):
    """pour une liste donnée de parseurs (certains d'entre eux éventuellement à None)
    appliquer le bon parseur à chaque élément colonne de la ligne en entrée"""

    return [parser(value) if parser is not None else value
        for value, parser in zip(input_row, parsers)]

def parse_rows_with(reader, parsers):
    """encapsule un lecteur pour appliquer les parseurs à chacune de ses lignes"""
    for row in reader:
      yield parse_row(row, parsers)
```

Que se passe-t-il en cas de données non conformes, par exemple une valeur en virgule flottante qui ne représenterait pas un nombre ? Nous préférons une réponse `None` plutôt qu'un plantage du programme. Pour cela, il faut l'aide d'une fonction support :

```
def try_or_none(f):
    """enveloppe f pour retourner None si f déclenche une exception
    suppose que f prend seulement une entrée"""
    def f_or_none(x):
        try: return f(x)
        except: return None
    return f_or_none
```

ce qui nous permet de réécrire `parse_row` pour l'utiliser :

```
def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
             for value, parser in zip(input_row, parsers)]
```

Par exemple, si nous avons des cours séparés par des virgules et contenant quelques données non conformes :

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

Nous sommes désormais en mesure de lire et analyser en une seule étape :

```
import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)
```

Il ne nous reste plus qu'à vérifier les lignes à la valeur `None` :

```
for row in data:
    if any(x is None for x in row):
        print row
```

et à décider ce qu'il convient de leur appliquer. (En général, les trois options sont : s'en débarrasser, revenir à la source et essayer de corriger les entrées non conformes ou absentes, ou alors ne rien faire et croiser les doigts.)

On peut créer le même genre de fonction support pour `csv.DictReader`. Dans ce cas, on ajoutera sans doute un dict de parseurs par nom de champ. Par exemple :

```
def try_parse_field(field_name, value, parser_dict):
    """tente d'analyser la valeur en utilisant la fonction adaptée tirée de
parser_dict"""
    parser = parser_dict.get(field_name)  # None en l'absence d'une telle entrée
    if parser is not None:
        return try_or_none(parser)(value)
    else:
        return value

def parse_dict(input_dict, parser_dict):
    return { field_name : try_parse_field(field_name, value, parser_dict)
             for field_name, value in input_dict.iteritems() }
```

Au cours de l'étape suivante, il est conseillé de contrôler les valeurs extrêmes à l'aide des techniques « L'exploration des données » de la page 125 ou par des recherches ad hoc. Par exemple, avez-vous remarqué qu'une des dates du

fichier stock 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 absentes, 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 ?)

La manipulation des 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 qui ressemblent à cela :

```
data = [
  {'closing_price': 102.06,
  'date': datetime.datetime(2014, 8, 29, 0, 0),
  'symbol':'AAPL'},
  # ...
]
```

Conceptuellement, nous considérerons qu'il s'agit de lignes (comme dans une feuille de calcul).

Posons-nous des questions 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 cherchons le cours de clôture le plus élevé pour AAPL. Nous décomposons la recherche en étapes simples.

1. Se limiter aux lignes AAPL.
2. Rendre le cours de clôture contenu dans chaque ligne.
3. Trouver le maximum de ces cours.

Nous pouvons réaliser ces trois étapes d'un seul coup à l'aide d'une list comprehension :

```
max_aapl_price = max(row["closing_price"]
                     for row in data
                     if row["symbol"] == "AAPL")
```

Plus généralement, nous pouvons chercher le cours de clôture le plus élevé pour chaque action de notre jeu de données. Une solution :

1. regrouper toutes les lignes avec le même symbole ;
2. à l'intérieur de chaque groupe, faire comme auparavant :

```
# regrouper les lignes par symbole
```

```
by_symbol = defaultdict(list)
for row in data :
    by_symbol[row["symbol"]].append(row)

# utiliser un dict comprehension pour trouver le maximum de chaque symbole
max_price_by_symbol = { symbol : max(row["closing_price"]
                                     for row in grouped_rows)
                         for symbol, grouped_rows in by_symbol.iteritems() }
```

Il existe des motifs répétitifs. Dans les deux exemples, nous avons besoin d'extraire le cours de clôture de chaque dict. Créons donc une fonction pour extraire un champ d'un dict et une autre fonction pour faire la même chose à partir d'une collection de dict :

```
def picker(field_name):
    """retourne une fonction qui sélectionne un champ à partir d'un dict"""
    return lambda row: row[field_name]

def pluck(field_name, rows):
    """transforme une liste de dict en liste de valeurs field_name"""
    return map(picker(field_name), rows)
```

Nous pouvons aussi créer une fonction pour grouper les lignes par résultats d'une fonction de regroupement et pour appliquer éventuellement une transformation `value_transform` à chaque groupe :

```
def group_by(grouper, rows, value_transform=None):
  # la clé est la sortie de la fonction de groupement, la valeur est une liste de lignes
  grouped = defaultdict(list)
  for row in rows:
    grouped[grouper(row)].append(row)

  if value_transform is None:
    return grouped
  else:
    return { key : value_transform(rows)
             for key, rows in grouped.iteritems() }
```

Nous pouvons ainsi réécrire très simplement notre exemple précédent. Par exemple :

```
max_price_by_symbol = group_by(picker("symbol"),
                               data,
                               lambda rows: max(pluck("closing_price", rows)))
```

Nous sommes à présent prêts à passer à des questions plus complexes, telles qu'identifier les variations de cours les plus fortes et les plus petites chaque 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. 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 ».

Pour commencer, écrivons une fonction dédiée au travail au sein d'un groupe donné :

```
def percent_price_change(yesterday, today):
    return today["closing_price"] / yesterday["closing_price"] – 1

def day_over_day_changes(grouped_rows):
    # trier les lignes par date
    ordered = sorted(grouped_rows, key=picker("date"))

    # zipper avec un offset pour obtenir des paires de jours consécutifs
    return [{ "symbol" : today["symbol"],
              "date" : today["date"],
              "change" : percent_price_change(yesterday, today) }
            for yesterday, today in zip(ordered, ordered[1:])]
```

Utilisons ensuite le résultat comme `value_transform` dans un `group_by` :

```
# la clé est le symbole, la valeur est une liste de dict "change"
changes_by_symbol = group_by(picker("symbol"), data, day_over_day_changes)

# rassembler tous les dict "change" dans une longue liste
all_changes = [change
               for changes in changes_by_symbol.values()
               for change in changes]
```

À ce stade, il est facile de trouver le plus grand et le plus petit :

```
max(all_changes, key=picker("change"))
# {'change':0.3283582089552237,
# 'date': datetime.datetime(1997, 8, 6, 0, 0),
# 'symbol':'AAPL'}
# consulter par exemple http://news.cnet.com/2100-1001-202143.html

min(all_changes, key=picker("change"))
# {'change': -0.5193370165745856,
# 'date': datetime.datetime(2000, 9, 29, 0, 0),
# 'symbol':'AAPL'}
# consulter par exemple http://money.cnn.com/2000/09/29/markets/techwrap/
```

Ce nouveau jeu de données `all_changes` peut nous servir à trouver quel est le meilleur mois pour investir dans les actions technologiques. Groupons d'abord les variations par mois, puis calculons la variation globale pour chaque groupe.

Encore une fois, nous écrivons une `value_transform` appropriée et nous utilisons `group_by` :

```
# pour combiner les variations en pourcentage nous ajoutons 1 à chacune,
# nous les multiplions, et nous soustrayons 1
# par exemple, si nous combinons +10 % et -20 %, la variation globale est
# (1 + 10 %) * (1 - 20 %) - 1 = 1.1 * .8 - 1 = -12 %
def combine_pct_changes(pct_change1, pct_change2):
    return (1 + pct_change1) * (1 + pct_change2) - 1

def overall_change(changes):
    return reduce(combine_pct_changes, pluck("change", changes))

overall_change_by_month = group_by(lambda row: row['date'].month,
                                   all_changes,
                                   overall_change)
```

Nous nous livrerons à ce genre de manipulations tout au long du livre sans donner trop d’explications sur le sujet.

Le changement d'échelle

De nombreuses techniques sont sensibles à l'échelle utilisée pour vos données. Par exemple, étant donnés la taille et le poids de centaines de data scientists, supposez que vous essayez de constituer des groupes (ou *clusters*) selon les 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 méthode naturelle de faire serait de considérer les paires (taille, poids) comme des points dans un espace à deux dimensions. Prenons les personnes du tableau 10-1.

Tableau 10–1 Taille et poids

Personne	Taille (pouces)	Taille (centimètres)	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 est le voisin plus proche de A :

```
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 le plus proche de B est C :

```
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 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 de chaque

colonne :

```
def scale(data_matrix):
    """retourne l'espérance et l'écart-type de chaque colonne"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
             for j in range(num_cols)]
    return means, stdevs
```

Ensuite, nous allons les utiliser pour créer une nouvelle matrice de données :

```
def rescale(data_matrix):
  """Change l'échelle des entrées de sorte que chaque colonne a pour espérance 0
  et pour écart-type 1. Laisse les colonnes sans écart-type """
  means, stdevs = scale(data_matrix)

  def rescaled(i, j):
    if stdevs[j] > 0:
      return (data_matrix[i][j] - means[j]) / stdevs[j]
    else:
      return data_matrix[i][j]

  num_rows, num_cols = shape(data_matrix)
  return make_matrix(num_rows, num_cols, rescaled)
```

Comme toujours, vous devez faire preuve de discernement. Si vous avez un énorme jeu de données avec les tailles et les poids et que 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 de fond et que vous ne vouliez pas traiter son écart-type sur un pied d'égalité avec ceux des autres dimensions.

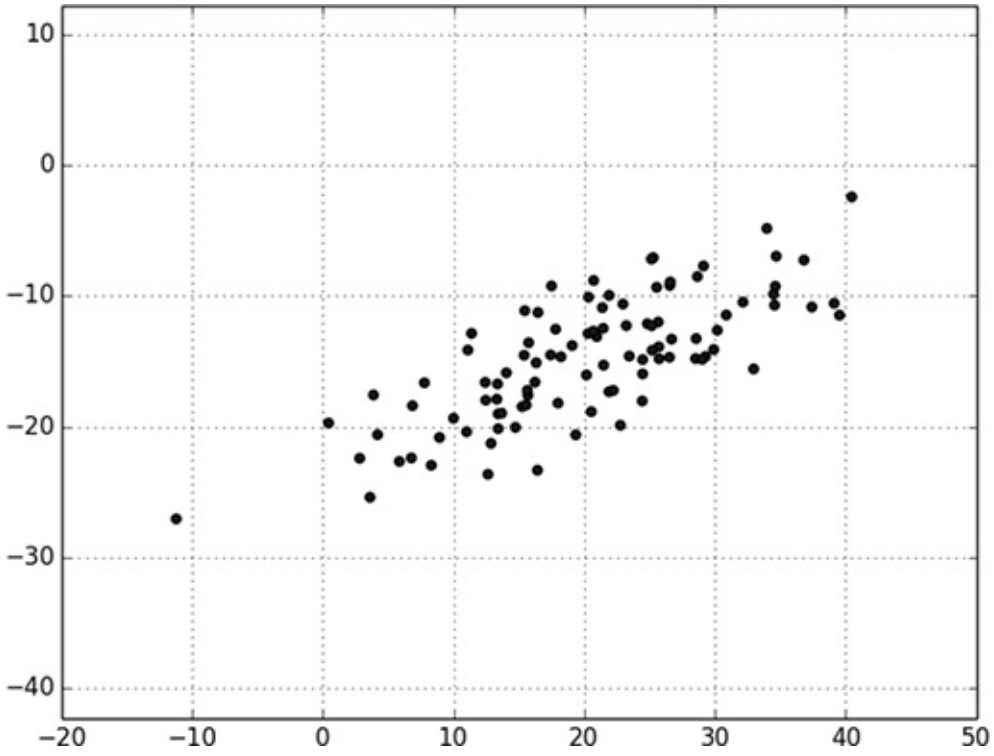
La réduction de dimensionnalité

Parfois, les dimensions « effectives » (ou utiles) des données ne correspondent pas aux dimensions que nous avons. Prenons l'exemple de la figure 10-5.

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 » pour extraire une ou plusieurs dimensions afin de capturer la plus grande partie de la variation des données.

Figure 10–5
Données avec des axes « non conformes »



Note

En pratique, on n'utilise pas cette technique sur un jeu de données comportant aussi peu de dimensions. La réduction de dimensionnalité présente plus d'utilité quand votre jeu de données possède un grand nombre de 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 à deux dimensions.

La première étape consiste à transformer les données de manière à ce que chaque dimension ait pour espérance zéro :

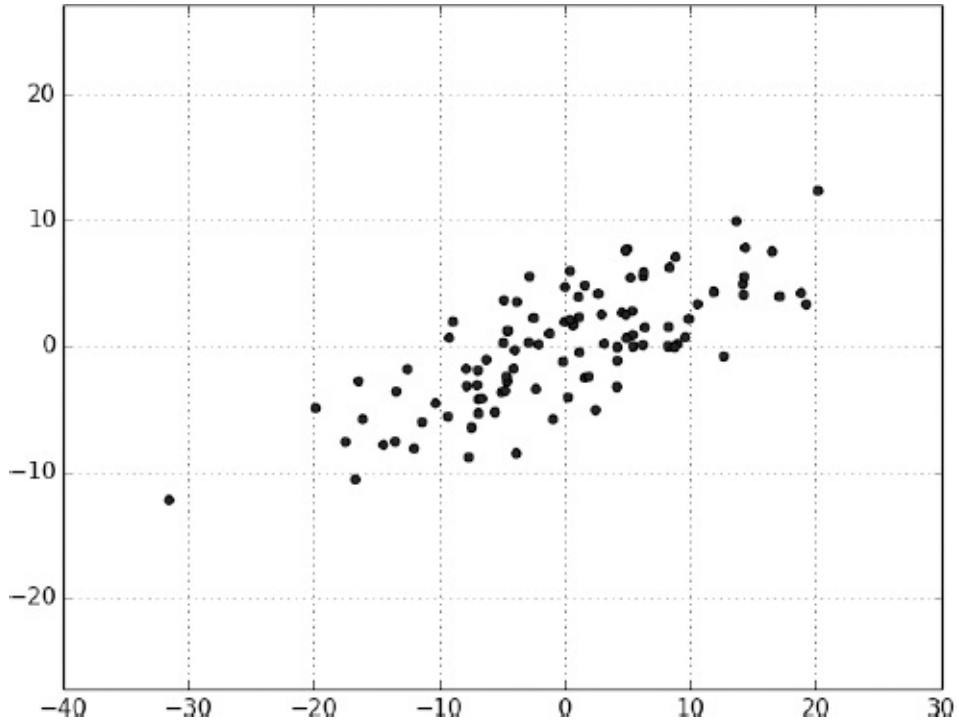
```
def de_mean_matrix(A):
    """retourne le résultat de la soustraction de l'espérance de sa colonne de  chaque
valeur de A . la matrice résultante a l'espérance 0 dans chaque colonne"""
    nr, nc = shape(A)
    column_means, _ = scale(A)
    return make_matrix(nr, nc, lambda i, j:A[i][j] - column_means[j])
```

(Si nous ne le faisions pas, nos méthodes risqueraient d'identifier l'espérance elle-même, plutôt que la variation de données.)

La figure 10-6 montre un exemple de données ainsi traitées.

Maintenant, étant donnée une matrice `X`, nous pouvons demander quelle est la direction qui capte la plus grande variance de données.

Figure 10–6
Les données après retrait de l'espérance



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 `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 :

```
def direction(w):
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

De fait, pour un vecteur non nul, nous pouvons calculer la variance de nos

données dans la direction déterminée par `w` :

```
def directional_variance_i(x_i, w):
    """la variance de la ligne x_i dans la direction déterminée par w"""
    return dot(x_i, direction(w)) ** 2

def directional_variance(X, w):
    """la variance des données dans la direction déterminée par w"""
    return sum(directional_variance_i(x_i, w)
               for x_i in X)
```

Pour trouver la direction qui maximise la variance, nous pouvons utiliser la descente de gradient dès lors que nous avons une fonction gradient :

```
def directional_variance_gradient_i(x_i, w):
    """la contribution de la ligne x_i au gradient de la variance direction-w """
    projection_length = dot(x_i, direction(w))
    return [2 * projection_length * x_ij for x_ij in x_i]

def directional_variance_gradient(X, w):
    return  vector_sum(directional_variance_gradient_i(x_i,w)
                       for x_i in X)
```

La première composante principale est tout simplement la direction qui maximise la fonction

`directional_variance` :

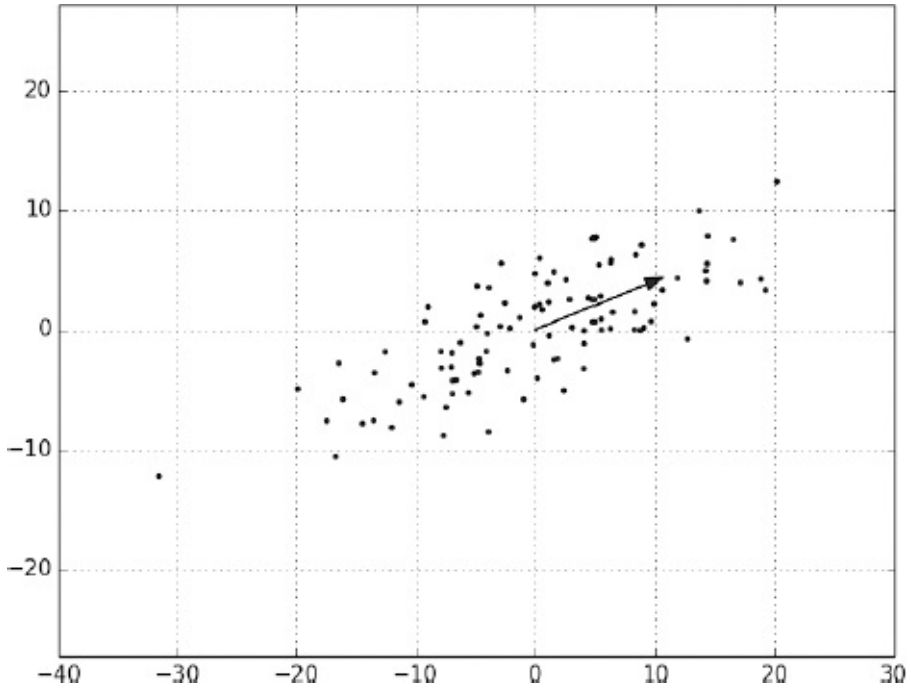
```
def first_principal_component(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_batch(
        partial(directional_variance, X),          # est maintenant une fonction de w
        partial(directional_variance_gradient, X), # est maintenant une fonction de w
        guess)
    return direction(unscaled_maximizer)
```

Ou, si vous préférez la descente du gradient stochastique :

```
# ici pas de "y", nous avons juste passé un vecteur de valeurs None
# et des fonctions qui ignorent cette entrée :
def first_principal_component_sgd(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_stochastic(
       lambda x, _, w: directional_variance_i(x, w),
       lambda x, _, w: directional_variance_gradient_i(x, w),
       X,
       [None for _ in X],  # le faux "y"
       guess)
    return direction(unscaled_maximizer)
```

Pour le jeu de données réduit, ce programme renvoie la direction `[0.924, 0.383]`, qui semble capter l'axe primaire quand nos données varient (figure 10-7).

Figure 10–7
Premier composant principal



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 :

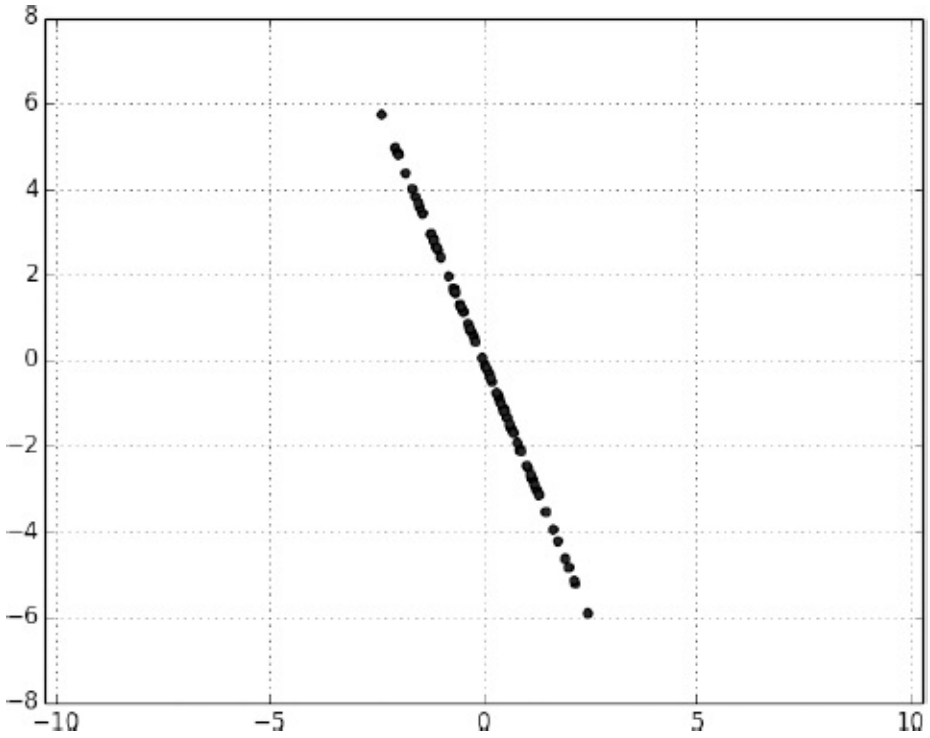
```
def project(v, w):
    """retourne la projection de v vers la direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

Si nous voulons trouver d'autres composantes, nous devons d'abord retirer les projections de ces données :

```
def remove_projection_from_vector(v, w):
    """projette v vers w et retranche le résultat de v"""
    return vector_subtract(v, project(v, w))

def remove_projection(X, w):
    """pour chaque ligne de X
    projette la ligne vers w, et retranche le résultat de la ligne"""
    return [remove_projection_from_vector(x_i, w) for x_i in X]
```

Comme cet exemple est seulement à deux dimensions, une fois le premier composant enlevé, il ne reste plus qu'une seule dimension (figure 10-8).

Figure 10–8
Les données après retrait du premier composant principal



À ce stade, nous trouverons le prochain composant principal en répétant le processus sur le résultat de `remove_projection` (figure 10-9).

Avec un jeu de données à grand nombre de dimensions, nous pouvons trouver autant de composants que nous le souhaitons :

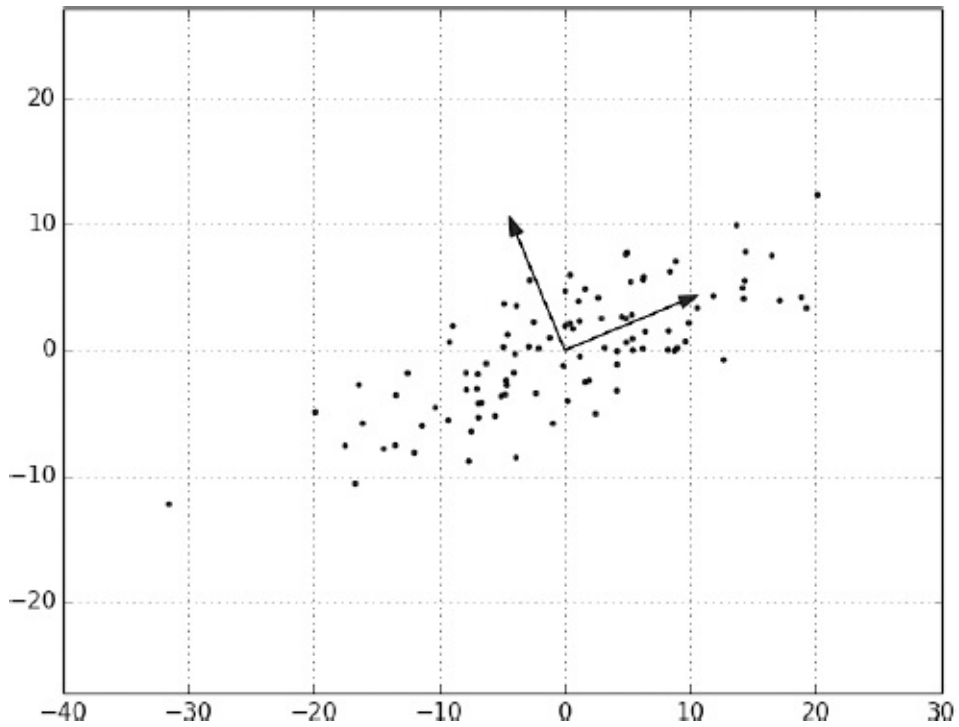
```
def principal_component_analysis(X, num_components) :
    components = []
    for _ in range(num_components):
        component = first_principal_component(X)
        components.append(component)
        X = remove_projection(X, component)

    return components
```

Nous pouvons transformer les données en les transportant dans un espace qui comporte une dimension de moins grâce aux éléments :

```
def transform_vector(v, components):
    return [dot(v, w) for w in components]

def transform(X, components):
    return [transform_vector(x_i, components) for x_i in X]
```

Cette technique est utile pour plusieurs raisons. Premièrement, elle peut nous aider à nettoyer les données en éliminant les dimensions parasites et en consolidant les dimensions qui sont hautement corrélées.

Figure 10–9
Les deux principaux composants



Deuxièmement, après avoir extrait une représentation à petit nombre de dimensions, nous pouvons utiliser une grande variété de techniques qui ne marchent pas aussi bien sur un espace à grand nombre de dimensions. Nous verrons des exemples de ces techniques à travers le livre.

En même temps, si cette méthode aide à construire de meilleurs modèles, elle peut aussi rendre ces modèles plus difficiles à interpréter. Il est facile de comprendre 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 le troisième composant principal ajoute 10 000 $ de salaire en moyenne ».

Pour aller plus loin

- Comme indiqué à la fin du chapitre 9, pandas est probablement l'outil Python de base pour nettoyer, transformer, manipuler les données et travailler avec. Tous les exemples que nous avons exécutés à la main dans ce chapitre peuvent être simplifiés en faisant appel à pandas. *L'analyse de données en Python*, 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 matrices, dont PCA (*Principal Component Analysis* ou analyse en composantes Principales, vue dans ce chapitre).

11

Apprentissage automatique

Je suis toujours prêt à apprendre, mais je n'aime pas toujours qu'on m'apprenne.
- Winston Churchill

Beaucoup s'imaginent que la data science est surtout de l'apprentissage automatique et que les experts passent leur temps à construire, entraîner et ajuster des modèles d'apprentissage automatique toute la journée. (N'oublions pas que la plupart des 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 les données, comprendre les données, nettoyer les données, formater les données ; après quoi l'apprentissage automatique est une préoccupation secondaire. Même ainsi, c'est une préoccupation secondaire intéressante et essentielle que vous ne pouvez pas ignorer si vous voulez vous consacrer à la data science.

La modélisation

Avant de parler de l'apprentissage automatique (ou *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 comme entrées « nombre d'utilisateurs » et « revenu des annonces par utilisateur » et « nombre d'employés » et en sortie votre bénéfice annuel pour les prochaines années. Un livre de recettes suit un modèle qui relie des entrées « nombre de convives » et leur état plus ou moins affamé avec les 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 des cartes dans le sabot.

Le modèle économique est probablement fondé sur des relations mathématiques simples : le bénéfice est égal aux recettes moins les dépenses, le revenu est égal au nombre d'articles vendus multiplié par le 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 poker et quelques hypothèses raisonnables sur la manipulation aléatoire des cartes.

Qu’est-ce que l’apprentissage automatique ?

Chacun possède sa définition personnelle, mais nous parlerons 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 », mais nous nous en tiendrons ici à l’expression « apprentissage automatique ». Notre objectif est d’utiliser les données existantes pour développer des modèles que nous pourrons utiliser pour prédire divers résultats à partir de données nouvelles. Par exemple :

- prédire si un courriel est du spam ou non ;
- prédire si une transaction par carte bancaire est frauduleuse ;
- prédire sur quelle annonce un acheteur a le plus de chance de cliquer ;
- prédire quelle équipe de football 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 labellisées avec les bonnes réponses) et aux modèles non supervisés (pour lesquels il n’y a pas de données labellisées). Il existe d’autres types de modèles semi-supervisés (pour lesquels seules quelques-unes des données sont labellisées) et en ligne (pour lesquels le modèle a besoin en permanence de s’ajuster aux données nouvellement arrivées), mais nous ne les traiterons pas dans ce livre.

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 pourrons 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 nous pouvons décider qu’un arbre de décision est une bonne méthode pour diagnostiquer les maladies de nos patients et utiliser les données pour apprendre l’arbre « optimal ». Dans la suite de ce livre, nous allons étudier différentes familles de modèles à apprendre.

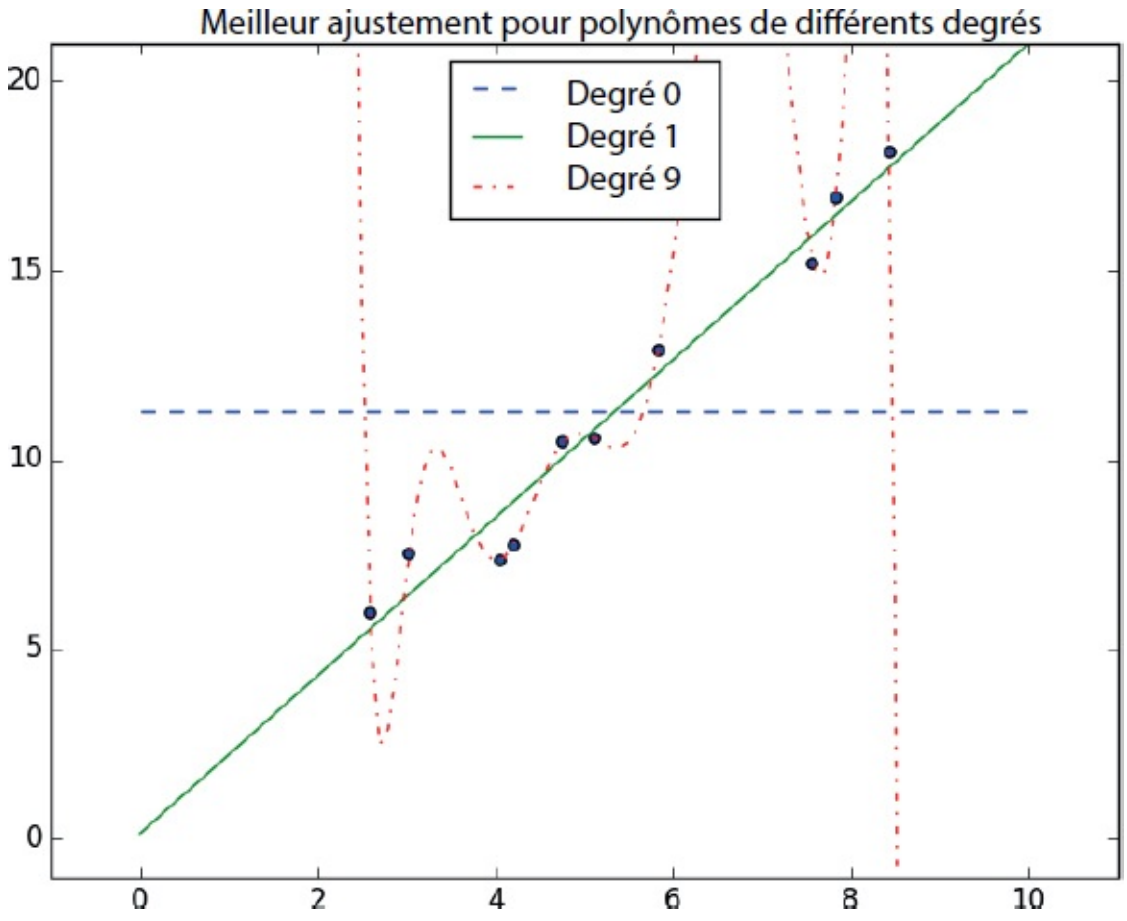
Mais auparavant nous devons mieux comprendre les principes de l’apprentissage automatique. Dans la suite de ce chapitre, nous introduirons quelques concepts de base avant de nous intéresser aux modèles eux-mêmes.

Surajustement et sous-ajustement

Un risque classique en apprentissage automatique est d'être trop ajusté : produire un modèle excellent avec vos données d'apprentissage, mais qui se généralise mal aux données nouvelles. Ce qui revient à apprendre du « bruit » au niveau du modèle. Ou autrement dit, cela revient à apprendre à identifier des entrées spécifiques et non représentatives d'une règle générale plutôt que les facteurs réellement prédictifs pour la sortie attendue.

Le risque opposé est le sous-ajustement : produire un modèle aux performances médiocres sur les données d'apprentissage, même si normalement face à un tel comportement il est naturel de décider que votre modèle n'est pas assez bon et d'en chercher un meilleur.

Figure 11–1
Surajustement *(overfitting)* et sous-ajustement *(underfitting)*



Sur 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.)

La ligne horizontale montre le polynôme de degré 0 (c'est-à-dire une valeur constante) le mieux adapté. Il est très sous-adapté 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 bien 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 faire l'apprentissage du modèle et pour le tester.

La manière la plus simple consiste à diviser votre jeu de données, en utilisant par exemple les deux tiers pour l'apprentissage et le tiers restant pour des mesures de performance :

```
def split_data(data, prob):
    """sépare les données en fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

Souvent, nous avons une matrice x de variables en entrée et un vecteur y de 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 de contrôle :

```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                    # fait correspondre les valeurs d'apprentissage
    train, test = split_data(data, 1 - test_pct)  # sépare le jeu de données
    x_train, y_train = zip(*train)                # astuce magique unzip
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

de façon à produire quelque chose comme ceci :

```
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 était trop ajusté aux données d'apprentissage, on peut s'attendre alors 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, il y a des cas où cela est faux.

Le premier cas se produit s'il y a des motifs 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, la plupart des utilisateurs vont apparaître dans les données d'apprentissage et les données de test et certains modèles peuvent apprendre à identifier ces utilisateurs plutôt qu'à découvrir des relations entre les attributs. Ce n'est pas un gros problème, toutefois 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 trop ajusté, la décision « choisir un modèle qui est le plus performant sur le jeu de données de test » est un méta-apprentissage qui considère le jeu de test comme un deuxième jeu d'apprentissage. (Évidemment le modèle le plus performant en test sera également performant sur le jeu de test.)

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 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 un nouveau-né pour prédire, avec une précision supérieure à 99 %, s'il développera 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 (ça ressemble un peu à « leucémie »).

Comme nous le verrons bientôt, le test est fiable à 98 %. Il est aussi particulièrement stupide, et c'est une bonne illustration de la raison pour laquelle on n'utilise pas l'exactitude (*accuracy* en anglais) pour mesurer la qualité 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 d'avion est-il un terroriste caché ?

Soit un jeu de données labellisées et un modèle prédictif de ce type, chaque donnée élémentaire tombe dans une des quatre catégories suivantes.

- Vrai positif : « Ce message est un spam et nous l'avons reconnu correctement comme spam. »
- Faux positif (erreur de type 1) : « Ce message n'est pas un spam, mais nous l'avons reconnu à tort comme spam. »
- Faux négatif (erreur de type 2) : « Ce message est un spam, mais nous avons prédit que ce n'est pas un spam. »
- Vrai négatif : « Ce message n'est pas un spam, et nous avons bien reconnu que ce n'est pas un spam. »

Souvent nous représentons ces résultats comme des compteurs dans une matrice de confusion (ou tableau de contingence) :

	Spam	**Non-spam**
Prédit (reconnaît) spam	Vrai positif	Faux positif
Prédit (reconnaît) 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 pendant la vie est de 1,4 % environ, soit 14 personnes pour 1 000.

Si nous pensons que ces deux facteurs sont indépendants et que nous

appliquons mon test « Lucie veut dire leucémie » à 1 million de personnes, nous attendons une matrice de confusion de la forme.

	Leucémie	Non leucémie	Total
Lucie	70	4 930	5 000
Non Lucie	3 930	981 070	995 000
Total	14 000	986 000	1 000 000

Nous pouvons utiliser ces résultats pour calculer différentes statistiques sur la performance du modèle. Par exemple, la précision est définie comme la fraction de prédictions correctes :

```
def accuracy(vp, fp, fn, vn):
    correct = vp + vn
    total = vp + fp + fn + vn
    return correct / total

print accuracy(70, 4930, 13930, 981070) # 0.98114
```

Voilà un nombre impressionnant. 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 de regarder 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(vp, fp, fn, vn):
    return vp / (vp + fp)

print precision(70, 4930, 13930, 981070) # 0.014
```

Le rappel (`recall`) mesure quelle fraction de vrais positifs notre modèle a identifiée :

```
def recall(vp, fp, fn, vn):
  return vp / (vp + fn)

print recall(70, 4930, 13930, 981070) # 0.005
```

Ces deux nombres sont mauvais, ce qui reflète la piètre qualité de notre modèle.

Parfois, la précision et le rappel se combinent dans le score `f1`, défini comme suit :

```
def f1_score(vp, fp, fn, vp):
    p = precision(vp, fp, fn, vn)
    r = recall(vp, fp, fn, vn)

    return 2 * p * r / (p + r)
```

C’est la moyenne harmonique de la précision et du rappel, et elle est nécessairement située entre les deux. En général, le choix d’un modèle suppose d’arbitrer entre la précision et le rappel. 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.

Vous pouvez aussi considérer la situation comme un arbitrage 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 avez de 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. Quand 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 voir 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 à 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 avoir les mêmes valeurs moyennes). Nous dirons que la variance est faible. Un biais élevé et une variance faible sont caractéristiques d'un sous-ajustement.

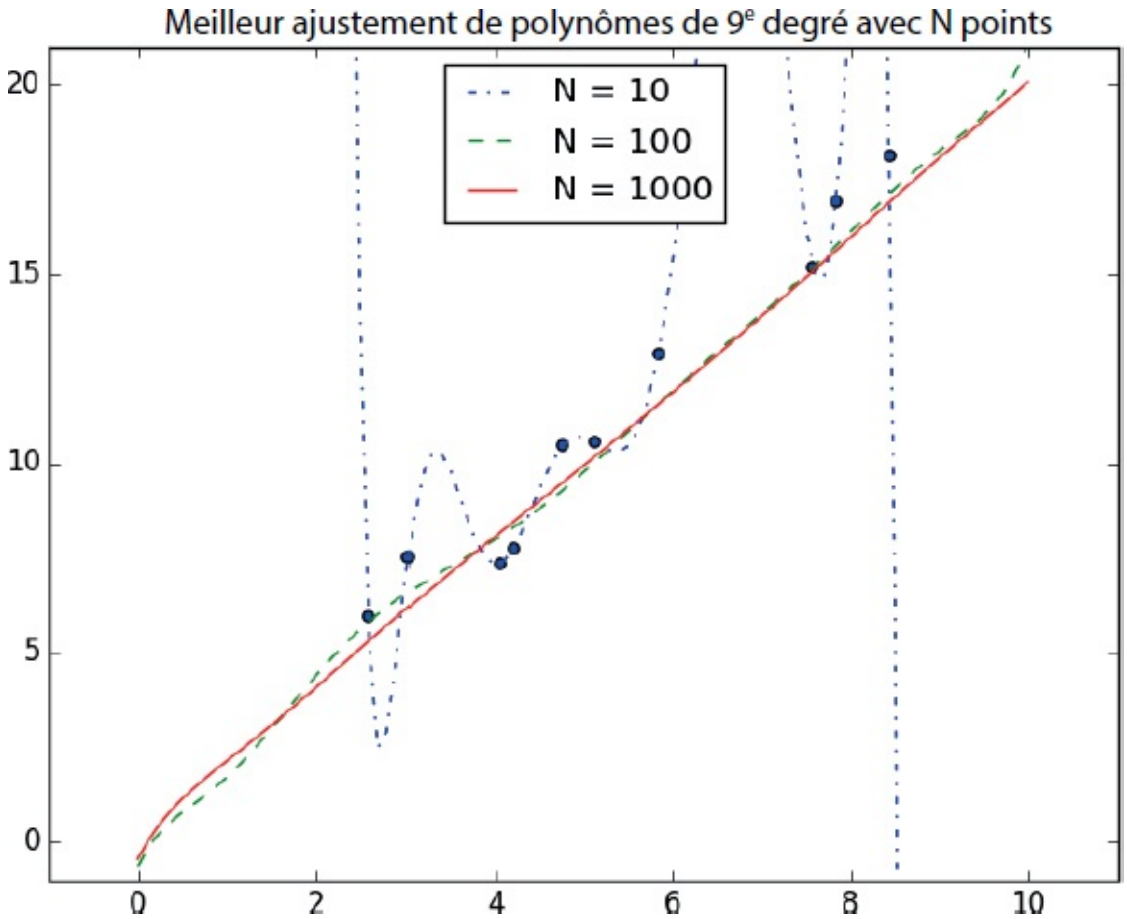
À l'opposé, le modèle de degré 9 correspond très exactement au jeu de données d'apprentissage. Le biais est très faible, mais la variance est forte (il est probable que deux jeux de données d'apprentissage donneront des modèles très différents). Cela est typique d'une situation de surajustement.

Aborder les questions de modèle sous cet angle peut vous aider à comprendre pourquoi votre modèle ne fonctionne pas très bien.

Si votre modèle est très biaisé (ce qui veut dire qu'il n'est pas performant avec vos données d'apprentissage), essayez d'ajouter davantage de variables (ou *features*). Passer du modèle de degré 0 présenté plus haut, « Surajustement et sous-ajustement », au modèle de degré 1 était une amélioration importante.

Si votre modèle a une variance élevée, alors au contraire, *enlevez* des variables. Une autre solution est d'obtenir davantage de données (si cela est possible).

Figure 11–2
Réduction de la variance avec davantage de données



Sur la figure 11-2, nous avons ajusté un polynôme de degré 9 à différentes tailles d'échantillons. Le modèle basé sur 10 données élémentaires est étalé partout comme nous l'avons vu précédemment. Si nous prenons 100 données élémentaires d'apprentissage, alors au contraire il est 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.

Extraction et sélection des variables

Comme nous l'avons déjà dit, quand vos données ne présentent pas assez de variables, votre modèle sera vraisemblablement sous-ajusté. Et quand vos données ont trop de variable, il est facile de trop ajuster. Mais que sont ces variables et d'où viennent-elles ?

Les variables, ce sont les entrées que nous fournissons à notre modèle.

Dans le cas le plus simple, les variables vous sont fournies, c'est tout. Si vous voulez prédire le salaire d'une personne à partir de ses années d'expérience, alors les années d'expérience sont la seule variable dont vous avez besoin.

(Bien que, comme nous l'avons vu précédemment au paragraphe sur l'ajustement et le sous-ajustement, 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 antispam destiné à prédire si un message reçu est un pourriel ou non. La plupart des modèles ne savent pas traiter un e-mail brut, qui n'est qu'une collection de textes. Il faut donc extraire des variables. Par exemple :

- Est-ce que le message contient le mot « viagra » ?
- Combien de fois la lettre « d » apparaît-elle ?
- Quel est le domaine de l'expéditeur ?

La première est simplement un oui ou un non, que nous encoderons 1 ou 0. La deuxième est un nombre. La troisième est un choix parmi un ensemble d'options indépendantes.

La plupart du temps, voire toujours, nous devrons extraire de nos données des variables correspondant à une de ces trois catégories. De plus, le type des variables utilisées va contraindre le type de modèles utilisables.

Le classifieur naïf de Bayes *(Naive Bayes)*, que nous développerons au chapitre 13, est adapté aux fonctions oui-ou-non telles que la première de la liste précédente.

Les modèles de régression, que nous étudierons aux chapitres 14 et 16, demandent des fonctions numériques (qui peuvent comporter des variables indicatrices prenant uniquement les valeurs 0 ou 1).

Et les arbres de décision, que nous verrons au chapitre 17, peuvent traiter des données numériques ou des catégories.

Nous avons cherché comment créer des variables dans le cas du filtre antispam, 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, ce peut être une bonne idée de les réduire à une poignée de dimensions importantes (comme dans la réduction dimensionnelle vue plus haut) et d'utiliser seulement ce petit nombre de variables. Ou il peut être préférable d'utiliser une technique (comme la « régularisation », que nous aborderons plus loin) qui pénalise les modèles qui ont trop de variables.

Comment choisir les variables ? 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 est un bon indicateur pour repérer les spams. Et vous avez sans doute aussi l'intuition que le nombre de lettres « d » est vraisemblablement un indicateur pas très intéressant pour détecter des spams. Mais en général vous devrez vous livrer à différents essais, ce qui fait toute la saveur de l'exercice.

Pour aller plus loin

- Poursuivez vos lectures ! Les chapitres suivants sont consacrés à différentes familles de modèles d'apprentissage automatique.
- Le cours Coursera *Machine Learning* est le MOOC d'origine sur le sujet, c'est le bon endroit pour comprendre en profondeur les fondements de l'apprentissage automatique. Le MOOC Caltech *Machine Learning* est également un bon choix.
- *Elements of Statistical Learning* est un ouvrage de référence qui peut être téléchargé gratuitement. Mais prenez garde, il est réservé aux forts en maths !

12

k plus proches voisins

Si vous voulez embarrasser vos voisins, racontez la vérité sur eux.
– 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 que vous avez les données qui vont bien), il existe une méthode qui consiste à poser la même question à mes voisins. Habitant comme moi au 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 pour moi.

Imaginons maintenant que vous en savez plus sur moi que ma simple position 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, interroger mes voisins les plus proches pour toutes ces dimensions donnera probablement une meilleure prédiction qu'en s'adressant à l'ensemble de mes voisins. Telle est l'idée qui sous-tend la classification par « plus proches voisins ».

Le modèle

Le modèle des plus proches voisins est un des modèles prédictifs les plus simples. Il ne fait aucune hypothèse mathématique et ne demande pas non plus 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.

La plupart des techniques examinées dans ce livre envisagent le jeu de données dans son ensemble pour y découvrir des motifs répétés. Au contraire, la méthode des plus proches voisins ignore volontairement de nombreuses informations, car la prédiction pour chaque nouveau point 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, alors que 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 de labels correspondants. Les labels peuvent être `True` ou `False` (respectivement vrai ou faux) ; dans ce cas, ils indiquent si chaque entrée vérifie une condition comme : « Est-ce du spam ? », « Est-ce toxique ? » ou « Est-ce que ce serait agréable à regarder ? » Elles peuvent aussi être des catégories, comme des notations de films (G, PG, PG-13, R, NC-17) ou les noms des candidats à l'élection présidentielle, ou des langages de programmation préférés.

Dans notre cas, les données élémentaires seront des vecteurs, ce qui veut dire que nous pouvons 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 labels des `k` données élémentaires 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 est :

```
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

Mais 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):
    """supposons que les labels sont triés du plus proche au plus lointain"""
    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:
        return winner                     # un seul gagnant bien défini, on le retourne
    else:
        return majority_vote(labels[:-1]) # recommencer en éliminant le plus éloigné
```

Cette méthode est certaine d'aboutir, car dans le pire des cas nous irons jusqu'à ne garder qu'un seul label qui sera alors déclaré vainqueur.

Avec cette fonction, la création du classificateur est aisée :

```
def knn_classify(k, labeled_points, new_point):
    """chaque point labellisé doit être une paire (point, label)"""

    # trier les données labellisées de la plus proche à la plus éloignée
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # chercher les labels pour le k le plus proche
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # et les faire voter
    return majority_vote(k_nearest_labels)
```

Examinons son fonctionnement de plus près.

Exemple : langages préférés

Les résultats de la première enquête de DataSciencester sont connus et nous avons cherché les langages de programmation préférés de nos utilisateurs dans quelques grandes villes :

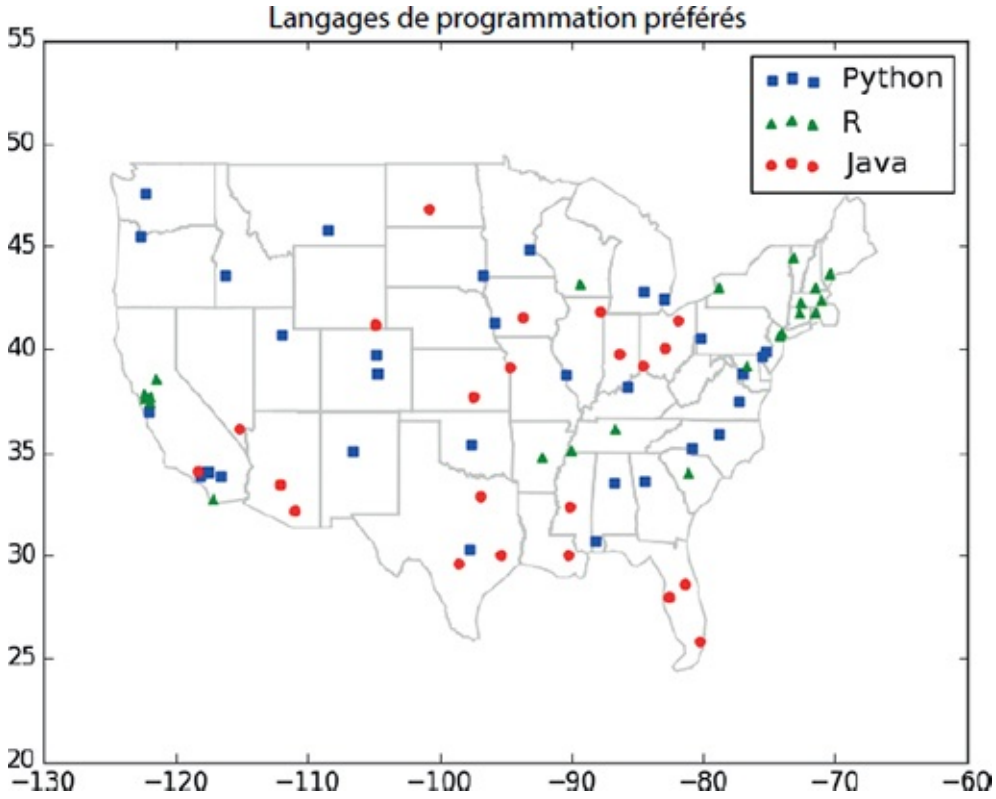
```
# chaque entrée est sous la forme ([longitude, latitude], favorite_language)

cities = [([-122.3 , 47.53], "Python"), # Seattle
         ([-96.85, 32.85], "Java"),     # Austin
         ([-89.33, 43.13], "R"),        # Madison
         # ... et ainsi de suite
]
```

Le responsable Engagement communautaire veut savoir si nous pouvons utiliser ces résultats pour prédire le langage préféré de nos utilisateurs dans des villes qui ne faisaient pas partie de notre enquête.

Comme toujours, la première étape consistera à représenter les données sur un graphique (figure 12-1) :

```
# clé = langage, valeur = paire (longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# nous voulons un marqueur et une couleur différents par langage
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# création d'une série de nuages de points pour chaque langage
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
                label=language, zorder=10)

plot_state_borders(plt)      # supposons que nous disposons d'une fonction pour
                             # afficher le contour des états

plt.legend(loc=0)            # laisser matplotlib choisir l'emplacement de la légende
plt.axis([-130,-60,20,55])   # définir les axes

plt.title("Langages de programmation préférés")
plt.show()
```

Figure 12–1
Langages de programmation préférés



Comme il semble que dans les lieux voisins on préfère le même langage, la méthode des k plus proches voisins apparaît comme un modèle prédictif raisonnable.

Note

Vous avez sans doute remarqué l'appel à `plot_state_borders()`, une fonction que nous n'avons pas encore définie. Il en existe une version sur la page GitHub, mais c'est un bon exercice que de la créer vous-même.

1. Chercher sur le Web la longitude et la latitude des limites des États.
2. Convertir les données récupérées en liste de segments `[(long1, lat1), (long2, lat2)]`.
3. Utiliser `plt.plot()` pour représenter les segments.

Pour commencer, voyons ce qui se passe si nous effectuons une prédiction pour chaque ville à partir de ses voisines :

```
# essayer différentes valeurs de k
for k in [1, 3, 5, 7]:
    num_correct = 0

    for city in cities:
        location, actual_language = city
        other_cities = [other_city
                        for other_city in cities
                        if other_city != city]

        predicted_language = knn_classify(k, other_cities, location)
```

```
        if predicted_language == actual_language:
            num_correct += 1

    print k, "neighbor[s]:", num_correct, "correct out of", len(cities)
```

Il semble que le meilleur résultat soit obtenu avec les 3 plus proches voisins, avec un résultat correct dans plus de 59 % des cas :

```
1 voisin[s]: 40 résultats corrects sur 75
3 voisin[s]: 44 résultats corrects sur 75
5 voisin[s]: 41 résultats corrects sur 75
7 voisin[s]: 35 résultats corrects sur 75
```

Maintenant, voyons quelles régions peuvent être classées avec quel langage à partir du modèle des plus proches voisins. Pour cela, nous allons classer toute une grille de points et ensuite les représenter graphiquement, comme nous l'avons fait pour les villes :

```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # ou 3, ou 5, ou ...

for longitude in range(-130, -60):
    for latitude in range(20, 55):
        predicted_language = knn_classify(k, cities, [longitude, latitude])
        plots[predicted_language][0].append(longitude)
        plots[predicted_language][1].append(latitude)
```

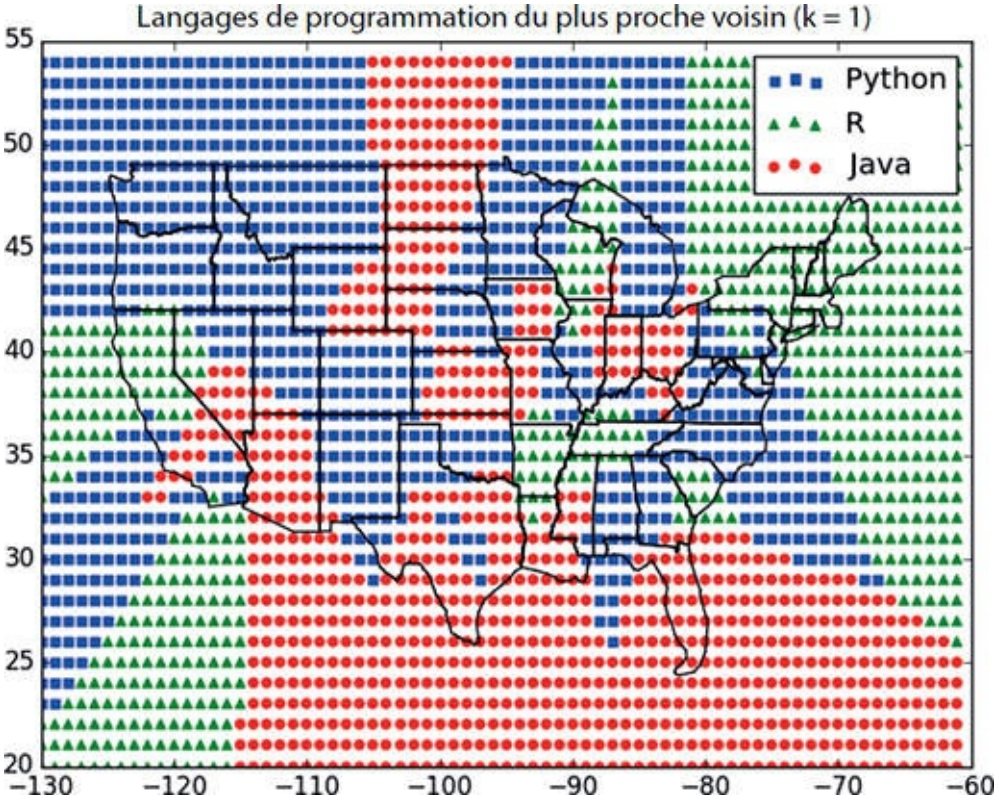
Par exemple, la figure 12-2 montre ce qui se passe si nous ne considérons que le plus proche voisin, soit `k = 1`.

Nous observons beaucoup de changements brutaux d'un langage à l'autre, avec des limites bien nettes. Si nous passons à 3 voisins, nous découvrons des régions plus lisses pour chaque langage (figure 12-3).

Et si nous augmentons le nombre de voisins à 5, les limites sont encore plus lisses (figure 12-4).

En gros, nos dimensions sont comparables. Dans le cas contraire, il aurait fallu changer d'échelle comme nous l'avons déjà fait (voir plus haut).

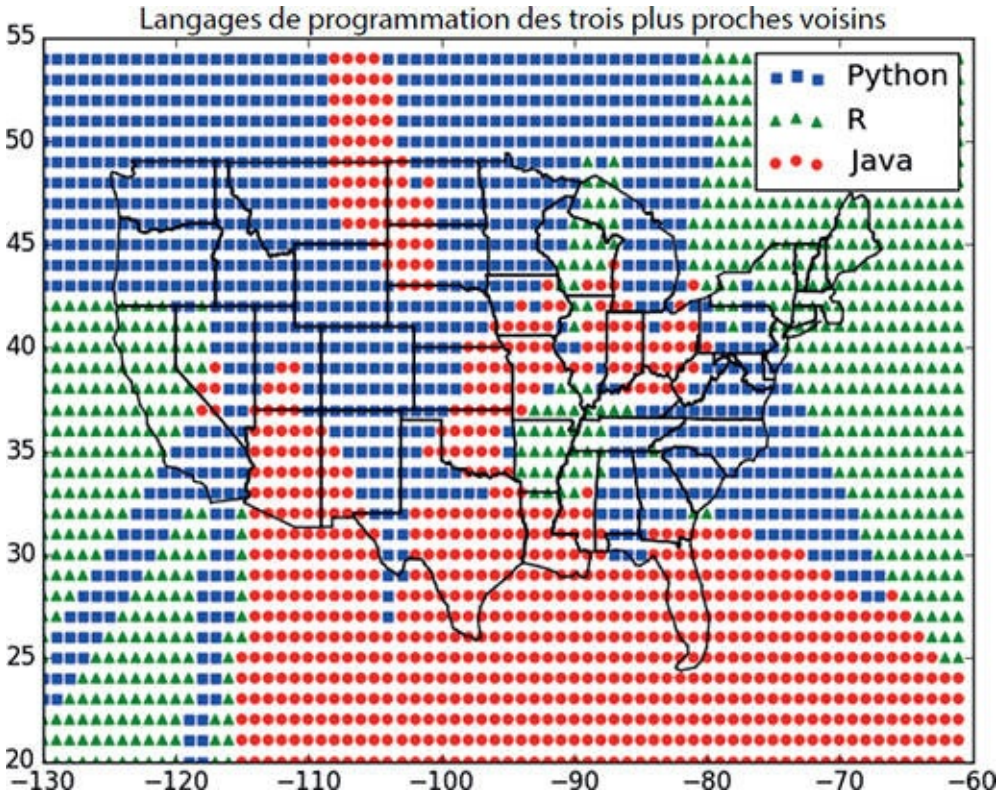
Figure 12–2
Langages de programmation du plus proche voisin (k = 1)


Langages de programmation du plus proche voisin (k = 1)
Python
R
Java
55
50
45
40
35
30
25
20
−130
−120
−110
−100
−90
−80
−70
−60

La 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 anglais), car les espaces de grande dimension sont immenses. Dans les espaces avec 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 eux.

Figure 12–3
Langages de programmation des 3 plus proches voisins



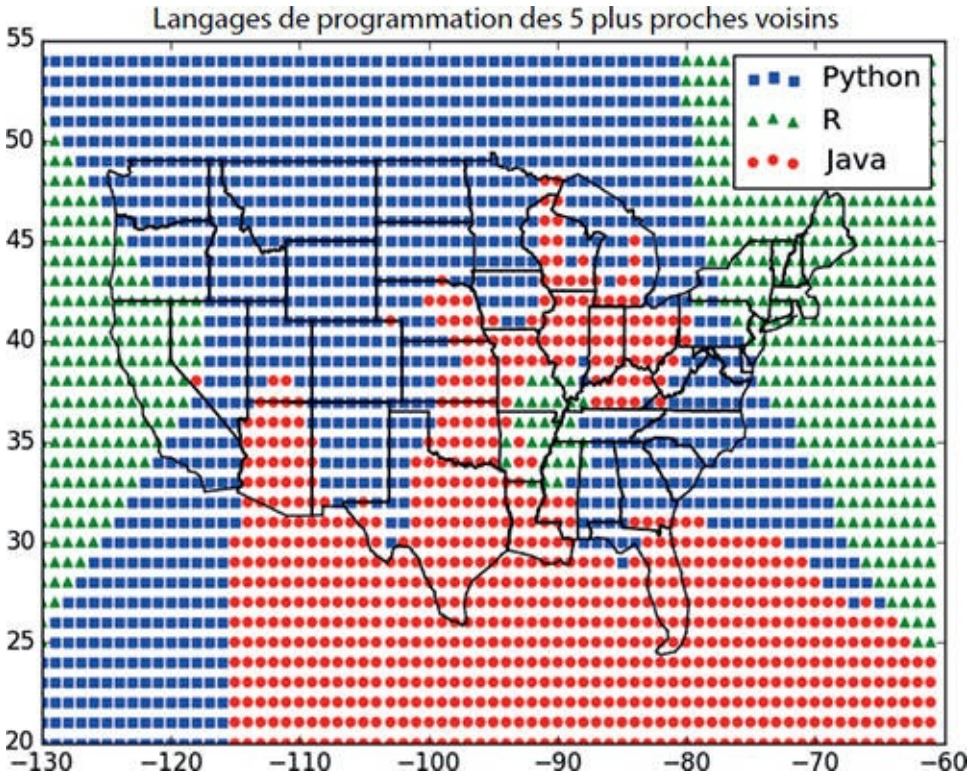
Désormais, la génération aléatoire de points devrait être une seconde nature :

```
def random_point(dim):
    return [random.random() for _ in range(dim)]
```

de même que l'écriture de fonctions pour générer les distances :

```
def random_distances(dim, num_pairs):
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

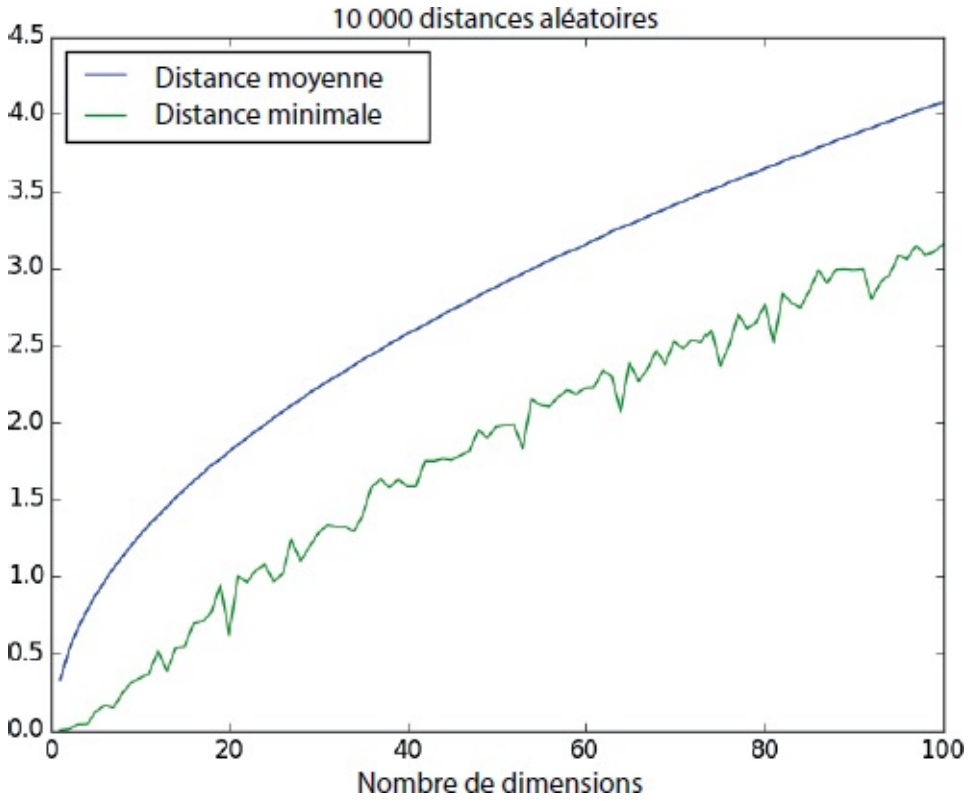
Figure 12–4
Langages de programmation des 5 plus proches voisins



Pour chaque dimension de 1 à 100, nous allons calculer 10 000 distances et les utiliser pour déterminer la distance moyenne entre les points et la distance minimale entre les points de chaque dimension (figure 12-5) :

```
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in dimensions:
    distances = random_distances(dim, 10000)   # 10 000 paires aléatoires
    avg_distances.append(mean(distances))      # noter la moyenne
    min_distances.append(min(distances))       # noter le minimum
```

Figure 12–5
La malédiction de la dimension



La distance moyenne entre les points augmente avec le nombre de dimensions. Mais c'est le ratio entre la distance la plus proche et la distance moyenne qui pose le plus gros problème (figure 12-6) :

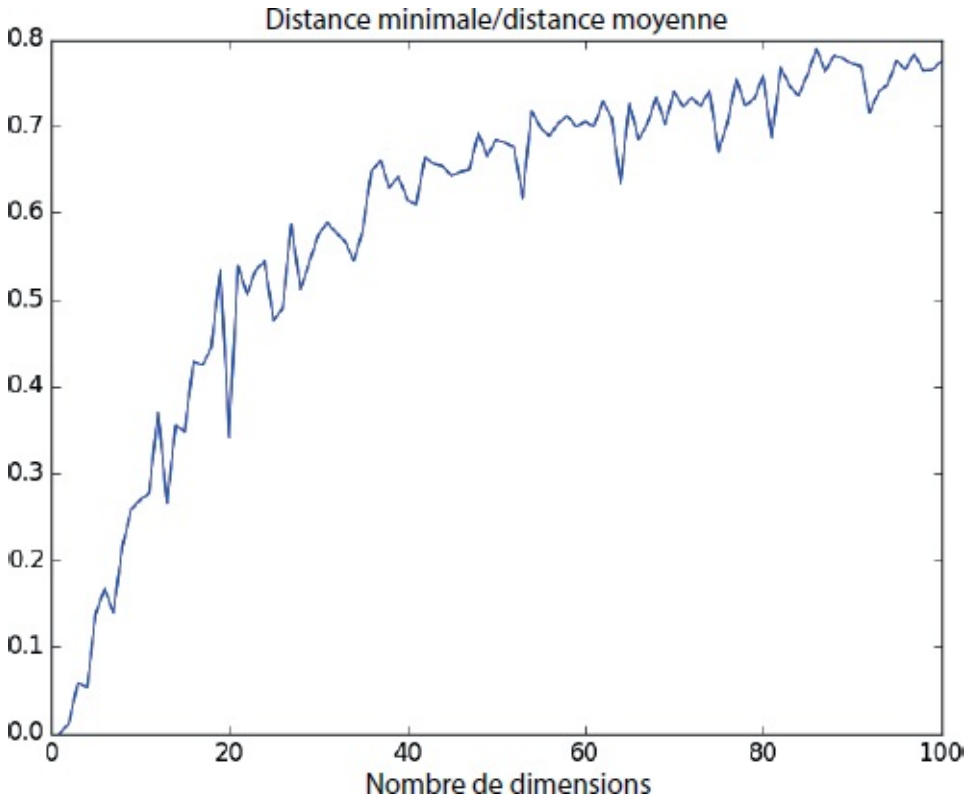
```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

Avec les jeux de données de faible dimension, les points les plus proches sont souvent beaucoup plus proches que la moyenne. Mais deux points ne sont proches que s'ils sont proches dans chaque dimension. Et chaque dimension supplémentaire, même si elle ne contient que du bruit, représente pour chaque point une nouvelle occasion de s'éloigner de chacun des autres points. Quand vous avez beaucoup de dimensions, il est probable que les points les plus proches ne soient pas beaucoup plus proches que la 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 de grande dimension.

Figure 12–6

Revoici la malédiction de la dimension

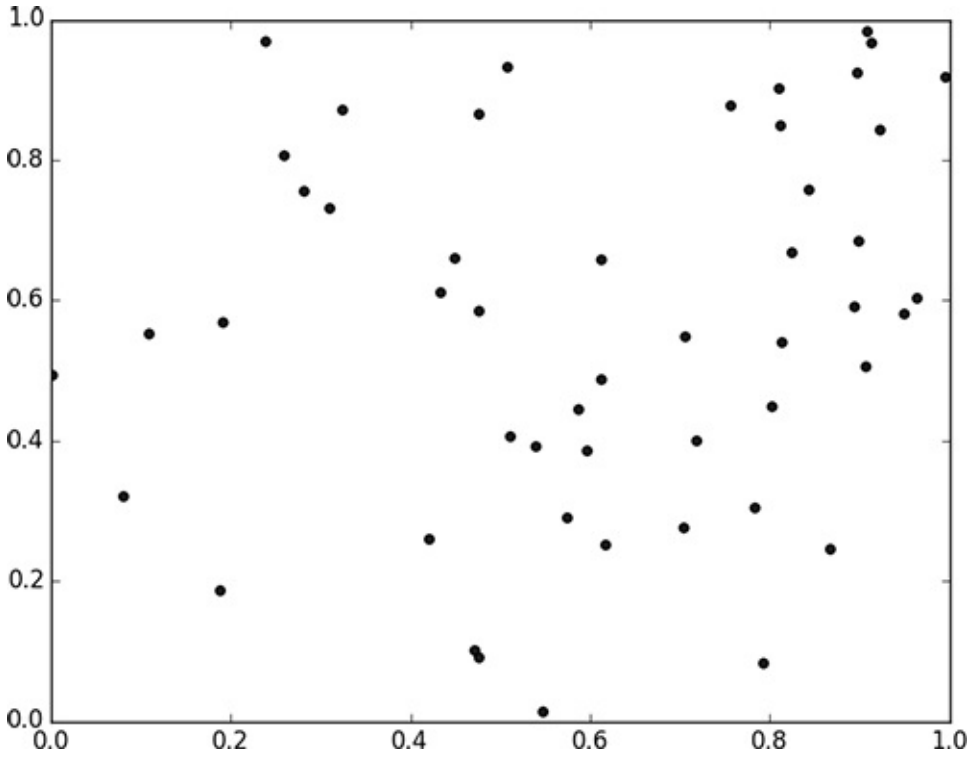


Si vous effectuez un tirage aléatoire de 50 nombres entre 0 et 1, vous obtiendrez sans doute un assez bon échantillon de l'intervalle d'une unité (figure 12-7).

Figure 12–7
Cinquante points aléatoires dans une dimension

Si vous tirez 50 points dans le carré unitaire, vous obtiendrez une couverture moins dense (figure 12-8).

Figure 12–8
Cinquante points aléatoires dans deux dimensions

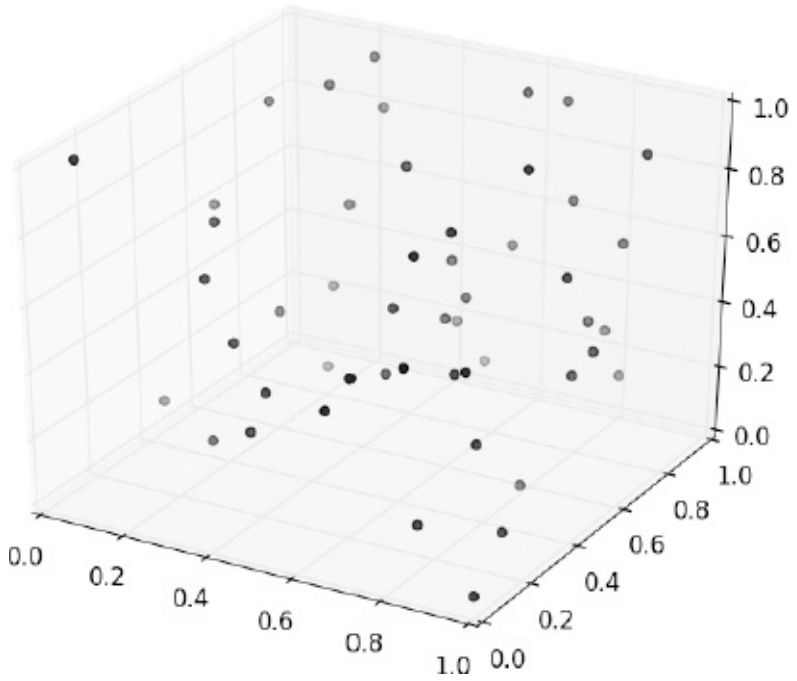


Et encore moins avec trois dimensions (figure 12-9).

matplotlib n'a pas la capacité de représenter quatre dimensions, aussi nous nous arrêterons là. Mais 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.

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.

Figure 12–9
Cinquante points aléatoires dans trois dimensions


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

Pour aller plus loin

scikit-learn propose de nombreux modèles des plus proches voisins.

13

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. C'est pourquoi DataScienceter possède une fonctionnalité populaire qui permet à ses membres de s'envoyer des messages les uns aux autres. La plupart de ses 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 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 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 *V* l'événement « ce message contient le terme Viagra ». Le théorème de Bayes nous apprend que la probabilité que le message soit un spam sachant qu'il contient le mot « Viagra » est :

$$P(S|V) = [P(V|S)P(S)]/[P(V|S)P(S) + P(V|\neg S)P(\neg S)]$$

Le numérateur représente la probabilité qu'un message soit du spam et qu'il contienne « Viagra ». Le dénominateur représente seulement la probabilité qu'un message contienne « Viagra ». Vous pouvez envisager le résultat de ce calcul comme la proportion de messages qui sont du spam parmi tous ceux qui contiennent « Viagra ».

Si nous avons une grande quantité de messages connus comme des spams, et une grande quantité de messages connus comme n'étant pas des spams, il est facile d'estimer $P(V|S)$ et $P(V|\neg S)$. Si, en plus, nous supposons que tout message a la même probabilité d'être un spam ou un non-spam (de sorte que $P(S) = P(\neg S) = 0{,}5$, alors :

$$P(S|V) = P(V|S)/[P(V|S) + P(V|\neg S)]$$

Par exemple, si le mot « Viagra » se retrouve dans 50 % des spams mais dans seulement 1 % des non-spams, alors la probabilité qu'un message donné contenant « Viagra » soit du spam est de :

$$0{,}5/(0{,}5 + 0{,}01) = 98\ \%$$

Un filtre antispam plus élaboré

Imaginons que vous avez un vocabulaire de plusieurs mots w_1, …, w_n. Pour respecter les conventions de la théorie des probabilités, nous écrirons X_i pour désigner l'événement « un message contient le mot w_i ». Imaginons également que, grâce à un processus non détaillé à ce stade, vous avez pu déterminer une estimation $P(X_i|S)$ de la probabilité qu'un message de spam contienne le mot de rang i, et une estimation similaire $P(Xi|\neg S)$ pour la probabilité 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 la présence ou l'absence de ces mots sont indépendantes les unes des autres selon que le message est un spam ou pas. Intuitivement, cette hypothèse signifie que savoir si un certain message spam contient le mot « Viagra » ne vous apporte aucune information sur le fait que ce même message contient le mot « Rolex ». Exprimé en termes mathématiques :

$$P(X_1 = x_1, \ldots, X_n = x_n|S) = P(X_1 = x_1|S) \times \ldots \times P(X_n = x_n|S)$$

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 aux mots « Viagra » et « Rolex » et que la moitié des messages de spam contiennent *cheap Viagra* (Viagra pas cher) et l'autre moitié contiennent *authentic Rolex* (Rolex authentique), alors la probabilité de Bayes naïf qu'un message de spam contienne les deux mots « Viagra » et « Rolex » devient :

$$P(X_1 = 1, X_2 = 1|S) = P(X_1 = 1|S)P(X_2 = 1|S) = 0.5 \text{ x } 0.5 = 0.25$$

car nous avons ignoré volontairement l'information selon laquelle la présence des mots « Viagra » et « Rolex » pouvait être corrélée. Malgré le caractère peu réaliste de cette hypothèse, ce modèle est souvent assez efficace. C'est pour cela qu'il est appliqué aux filtres antispam réels.

Le raisonnement du théorème de Bayes auquel nous avons fait appel pour notre filtre antispam « Viagra seulement » nous dit aussi que nous pouvons calculer la probabilité qu'un message soit du spam à l'aide de l'équation :

$$P(S|X = x) = P(X = x|S)/[P(X = x)|S) + P(X = x|\neg S)]$$

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 « soupassement » (ou dépassement de capacité inférieure, ou *underflow* en anglais) : les ordinateurs gèrent mal les nombres à virgule flottante trop proches de zéro. Mais 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 $p_1 * \ldots * p_n$ sous la forme équivalente (cette formule est mieux adaptée à la virgule flottante) :

$$\exp(\log(p_1) + \ldots + \log(p_n))$$

À présent, il ne nous reste plus qu'à calculer les estimations de $P(X_i|S)$ et $P(X_i|\neg S)$, c'est-àdire les probabilités qu'un message de spam (ou un message non spam) contienne le mot w_i. Si nous disposons d'une quantité raisonnable de messages « d'apprentissage » labellisés comme spam et 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.

Mais cela cause un gros problème. Imaginons que dans votre jeu d'apprentissage, le vocable « data » (données) apparaît seulement dans les messages non-spam. Nous estimerons alors $P(\text{'data'}|S) = 0$. Le résultat est que notre classificateur Bayes naïf assignera toujours au spam la probabilité 0 pour n'importe quel message qui contient le mot « data », même si c'est un message comme « data on cheap viagra and authentic rolex watches » (données sur du Viagra pas cher 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 $i^{\text{ème}}$ mot dans un spam comme :

$P(Xi|S) = (k + \text{nombre de spams contenant } w_i)/(2k + \text{nombre de spam total})$

Il en va de même pour $P(Xi|\neg S)$. En fait, quand nous calculons les probabilités de spam pour le $i^{\text{ème}}$ 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 = 0,01, ce qui permet à notre classificateur d'assigner aux messages qui contiennent le mot « data » quelques probabilités de spam qui seront 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 allons extraire les « mots » composés de lettres, nombres et apostrophes. Enfin, nous utiliserons `set()` pour récupérer les mots distincts :

```
def tokenize(message):
    message = message.lower()                       # convertit en minuscules
    all_words = re.findall("[a-z0-9']+", message)  # extrait les mots
    return set(all_words)                           # supprime les doublons
```

Notre seconde fonction consiste à compter les mots dans un jeu de données d'apprentissage labellisé. Elle retourne un dictionnaire dont les clés sont les mots et dont les valeurs sont des listes à deux éléments `[spam_count, non_spam_count]`, qui correspondent au nombre de fois où nous avons vu le mot en question dans les spams et dans les non-spams, respectivement :

```
def count_words(training_set):
    """jeu d'apprentissage composé de paires (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

L'étape suivante consiste à transformer ces deux nombres en probabilités qui seront estimées à l'aide du lissage décrit précédemment. Notre fonction retourne une liste de triplets constitués d'un mot, de la probabilité de le trouver dans un message de spam et de la probabilité de le trouver dans un message non spam :

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
     """transforme le compteur word_counts en liste de triplets w, p(w | spam)
        et p(w | ~spam)"""
    return [(w,
        (spam + k) / (total_spams + 2 * k),
        (non_spam + k) / (total_non_spams + 2 * k))
        for w, (spam, non_spam) in counts.iteritems()]
```

Enfin, nous utiliserons les probabilités de ce mot (et nos hypothèses de classification naïve bayésienne) pour assigner des probabilités aux messages :

```
def spam_probability(word_probs, message):
    message_words = okenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # boucler sur tous les mots de notre vocabulaire
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # si un mot *word* apparaît dans le message,
        # ajouter log (probabilité de le voir)
        if word in message_words:
         log_prob_if_spam += math.log(prob_if_spam)
         log_prob_if_not_spam += math.log(prob_if_not_spam)

    # si un mot *word* n'apparaît pas dans le message
    # ajouter log (probabilité de ne pas le voir)
    # qui vaut log (1 – probabilité de le voir)
    else:
         log_prob_if_spam += math.log(1.0 - prob_if_spam)
         log_prob_if_not_spam += math.log(1.0 - prob_if_not_spam)

    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_not_spam = math.exp(log_prob_if_not_spam)
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)
```

Nous pouvons réunir le tout dans notre classificateur naïf bayésien :

```
class NaiveBayesClassifier:

    def init (self, k=0.5):
        self.k = k
        self.word_probs = []

    def train(self, training_set):

       # compter les messages spam et non spam
       num_spams = len([is_spam
                        for message, is_spam in training_set
                        if is_spam])
    num_non_spams = len(training_set) - num_spams

    # faire passer les données d'apprentissage dans notre « pipeline »
    word_counts = count_words(training_set)
    self.word_probs = word_probabilities(word_counts,
                                         num_spams,
                                         num_non_spams,
                                         self.k)

    def classify(self, message):
        return spam_probability(self.word_probs, message)
```

Tester le modèle

Le corpus public de SpamAssassin est un bon jeu de données (bien qu'un peu ancien). Observons les fichiers préfixés par 20022010. (Sous Windows, il vous faudra un programme comme 7-Zip pour les décompresser et les extraire.)

Après avoir extrait les données (par exemple sur `C:\spam`), vous devriez avoir trois répertoires : `spam`, `easy_ham` et `hard_ham` (*ham* est le contraire de *spam*). Chaque répertoire contient de nombreux fichiers, chacun correspondant à un message unique. Pour se limiter à des choses simples, contentons-nous de regarder la ligne contenant l'objet de chaque message.

Comment identifier la ligne « objet » (*subject line* en anglais) ? En parcourant les fichiers, on voit qu'elles commencent toutes par `Subject:`. C'est ce que nous allons rechercher :

```
import glob, re

# modifier le chemin d'accès avec la configuration de vos répertoires
path = r"C:\spam\*\*"

data = []

# glob.glob retourne chaque nom de fichier qui correspond au masque de chemin fourni en
entrée
for fn in glob.glob(path):
    is_spam = "ham" not in fn

    with open(fn,'r') as file:
       for line in file:
            if line.startswith("Subject:"):
            # supprimer le "Subject: " en tête de ligne et conserver ce qui reste
            subject = re.sub(r"^Subject: ", "", line).strip()
            data.append((subject, is_spam))
```

Nous pouvons désormais séparer les données en ensemble d'apprentissage et ensemble de test. Nous serons alors prêts à construire un classificateur :

```
random.seed(0)        # juste pour avoir les mêmes réponses que moi
train_data, test_data = split_data(data, 0.75)

classifier = NaiveBayesClassifier()
classifier.train(train_data)
```

Nous pouvons maintenant vérifier ce que fait notre modèle :

```
# triplets (subject, actual is_spam, predicted spam probability)
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]


# supposer que spam_probability > 0.5 correspond à la prédiction de spam
# et compter les combinaisons de (actual is_spam, predicted is_spam)
counts = Counter((is_spam, spam_probability > 0.5)
                 for _, is_spam, spam_probability in classified)
```

Le résultat est de 101 vrais positifs (spam classé comme spam), 33 faux positifs (ham classé comme spam), 704 vrais négatifs (ham classé comme ham) et 38 faux négatifs (spam classé comme ham). Notre précision est donc de 101/(101 + 33) = 75 %, et notre rappel est de 101/(101 + 38) = 73 %. Ce n'est pas mal pour un modèle aussi simple.

Il est intéressant d'observer ce qui a été le plus mal classé :

```
# trier par spam_probability de la plus petite à la plus grande
classified.sort(key=lambda row: row[2])


# la probabilité de spams la plus élevée parmi les non spams
spammiest_hams = filter(lambda row: not row[1], classified)[-5:]


# la probabilité de spams la plus faible parmi les vrais spams
hammiest_spams = filter(lambda row: row[1], classified)[:5]
```

Les deux hams les plus indésirables (autrement dit les plus proches d'un spam) contiennent les mots *needed* (demandés, dont la présence est 77 fois plus probable dans un spam), *insurance* (assurance, 30 fois plus probable dans un spam) et *important* (10 fois plus probable dans un spam).

Le spam le moins indésirable est trop court *(Re:girls)* pour être significatif. Et le suivant est une sollicitation pour une carte bancaire dont la plupart des mots ne figuraient pas dans le jeu d'apprentissage.

Nous pouvons également examiner les mots les plus indésirables :

```
def p_spam_given_word(word_prob):
    """fait appel au théorème de Bayes pour calculer p(spam | message contains word)"""


    # word_prob est un des triplets produit par word_probabilities
    word, prob_if_spam, prob_if_not_spam = word_prob
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)


words = sorted(classifier.word_probs, key=p_spam_given_word)


spammiest_words = words[-5:]
hammiest_words = words[:5]
```

Les mots les plus indésirables sont *money* (argent), *systemworks* (suite

logicielle), *rate* (tarif), *sale* (vente) et *year* (année). Tous semblent en relation avec le fait de pousser le destinataire du message à acheter quelque chose. Et les mots les moins indésirables sont *spambayes*, *users* (utilisateurs), *razor*, *zzzzteana* et *sadev*, des termes dont la plupart semblent liés à la prévention du spam, ce qui est assez étrange.

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.

- On pourrait 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. On pourrait le modifier afin de filtrer les mots selon un seuil minimal optionnel `min_count`. Ainsi, on pourrait ignorer les mots dont le nombre d'occurrences est inférieur à ce minimum.
- Le système d'extraction des données ignore la similitude entre mots, par exemple *cheap* et *cheapest* (« pas cher » et « moins cher »). On pourrait modifier le classificateur en y intégrant une fonction facultative de *racinisation* (ou désuffixation, *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 (ou *stemmer*) de qualité. Le plus souvent, on fait appel à celui de Martin Porter pour la langue anglaise.
- 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'analyseur afin qu'il les émette quand cela est nécessaire.

Pour aller plus loin

- Vous pouvez consulter les articles de Paul Graham « A plan for Spam » (Un plan pour le spam) et « Better Bayesian Filtering » (De meilleurs filtres de Bayes) pour approfondir vos connaissances sur les principes de construction des filtres à spam.
- sickit-learn contient un modèle appelé `BernoulliNB`, qui implémente l'algorithme naïf de Bayes que nous avons mis en œuvre ici, ainsi que différentes variations de ce modèle.

14

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 en plus comprendre la nature de cette relation. C'est là qu'intervient la régression linéaire simple.

Le modèle

Comme vous vous en souvenez, nous recherchions la relation entre le nombre d'amis d'un utilisateur DataSciencester et le temps qu'il passe sur le site chaque jour. Supposons que vous êtes convaincu qu'avoir plus d'amis conduit à passer plus de temps sur le site, plutôt qu'une autre des explications dont nous avions discuté.

Le 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 alpha et bêta telles que :

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

Dans cette expression, y_i est le nombre de minutes qu'un utilisateur i passe quotidiennement sur le site, x_i est le nombre d'amis de l'utilisateur i et ε_i 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 alpha et un bêta donnés, on peut alors faire des prédictions de manière simple avec la fonction :

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha
```

Mais comment choisir alpha et bêta ? Chaque choix d'alpha et bêta 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 error(alpha, beta, x_i, y_i):
    """l'erreur de la prédiction beta * x_i + alpha
    quand la valeur réelle est y_i"""
    return y_i - predict(alpha, beta, x_i)
```

Nous aimerions connaître l'erreur totale sur l'ensemble du jeu de données. Mais nous ne voulons pas juste ajouter 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 :

```
def sum_of_squared_errors(alpha, beta, x, y):
    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 `alpha` et `beta` pour rendre `sum_of_squared_errors` le plus petit possible.

En faisant appel au calcul infinitésimal (ou à un calcul algébrique fastidieux), on obtient les valeurs alpha et bêta qui minimisent l'erreur :

```
def least_squares_fit(x, y):
    """étant données des valeurs d'apprentissage pour x et y,
    trouver les valeurs moindres carré de alpha et beta"""
    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 que c'est une solution raisonnable ? Le choix d'alpha dit simplement que quand nous connaissons la valeur moyenne de la variable indépendante `x`, nous pouvons prédire la valeur moyenne de la variable dépendante `y`.

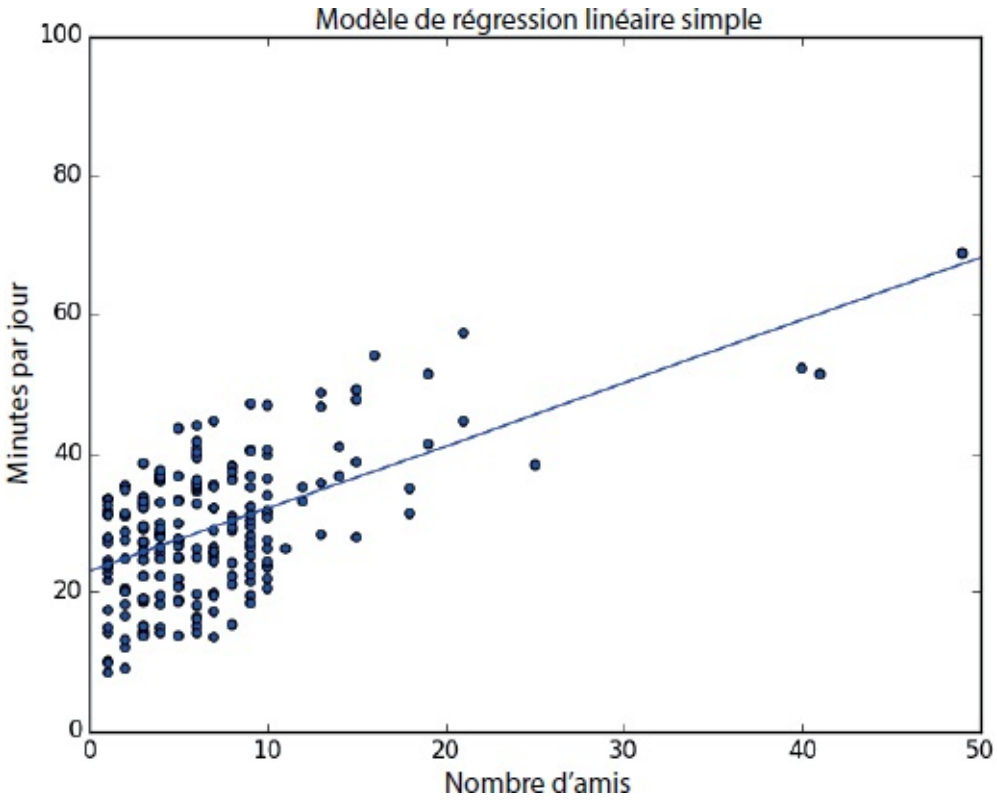
Le choix de `beta` signifie que quand la variable d'entrée augmente de la valeur de l'écart-type `standard_deviation(x)`, la prédiction augmente de `correlation(x,y) * standard_deviation(y).` Dans le cas où `x` et `y` sont parfaitement corrélés, une augmentation d'un écart-type de `x` entraîne une augmentation d'un écart-type de `y` dans la prédiction. S'ils sont parfaitement anticorrélés, l'augmentation de `x` entraîne une diminution de la prédiction. Et quand la corrélation est égale à zéro, bêta vaut zéro, ce qui veut dire que les modifications de `x` n'ont aucun impact sur la prédiction. On peut facilement appliquer ces principes aux données du chapitre 5 (en excluant les valeurs extrêmes) :

```
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
```

Ceci nous donne les valeurs alpha = 22,95 et bêta = 0,903. Notre modèle nous dit que nous nous attendons à ce qu'un utilisateur avec `n` amis passe 22,95 + `n`*0,903 minutes sur le site chaque jour. Ce qui veut dire aussi qu'un utilisateur qui n'a aucun ami chez DataSciencester passera 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 la 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 du graphique à l'œil nu. Une mesure classique est donnée par le coefficient de détermination (ou R^2), qui mesure la fraction de la variation totale de la variable dépendante (`y` ici) capturée par le modèle :

```
def total_sum_of_squares(y):
    """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, beta, x, y):
    """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_squared_errors(alpha, beta, x, y) / total_sum_of_squares(y))

r_squared(alpha, beta, num_friends_good, daily_minutes_good) # 0.329
```

Maintenant, nous choisissons les alpha et bêta qui minimisent la somme des erreurs des prédictions au carré. Un autre modèle linéaire possible aurait pu être *always predict mean(y)* (toujours prédire l'espérance de *y*) correspondant

à `alpha = mean(y)` et `beta = 0`). Dans ce modèle, la somme des erreurs au carré est exactement égale à la somme totale des carrés. Cela signifie que 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 doit être au moins égal à zéro. Et, comme la somme des erreurs au carré doit être au moins égale à 0, cela veut dire que, 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 à considérer.

L’utilisation de la descente de gradient

Si nous posons `theta = [alpha, beta]`, nous pouvons résoudre notre problème par la descente de gradient :

```
def squared_error(x_i, y_i, theta):
    alpha, beta = theta
    return error(alpha, beta, x_i, y_i) ** 2


def squared_error_gradient(x_i, y_i, theta):
    alpha, beta = theta
     return [-2 * error(alpha, beta, x_i, y_i),        # dérivée partielle par rapport à
alpha
              -2 * error(alpha, beta, x_i, y_i) * x_i] # dérivée partielle par rapport à
beta
# choisir une valeur aléatoire pour commencer
random.seed(0)
theta = [random.random(), random.random()]
alpha, beta = minimize_stochastic(squared_error,
                                  squared_error_gradient,
                                  num_friends_good,
                                  daily_minutes_good,
                                  theta,
                                  0.0001)
print alpha, beta
```

À partir des mêmes données, nous obtenons `alpha = 22.93` et `beta = 0.905` ce qui est très proche de la réponse exacte de la méthode précédente.

L’estimation du maximum de vraisemblance

Pourquoi choisir les moindres carrés ? Une des justifications réside dans l’estimation du maximum de vraisemblance.

Imaginons un échantillon de données $v_{1,} \ldots v_{n}$ qui provient d’une distribution dépendante d’un paramètre inconnu θ :

$$P(v_1, \ldots, v_n \mid \theta)$$

Si nous ignorons thêta, nous pouvons l’approcher et le concevoir comme la vraisemblance de θ selon l’échantillon :

$$L(\theta \mid v_1, \ldots, v_n)$$

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 probabilité 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 :

$$L(\alpha, \beta | xi, yi, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-(yi - \alpha - \beta xi)^2 / 2\sigma^2\right.$$

La vraisemblance basée sur le jeu de données complet est le produit des vraisemblances élémentaires, qui se trouve être la plus grande précisément quand `alpha` et `beta` 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 au chapitre 15 !

15

Régression linéaire multiple

Quand je considère un problème, je n'insère pas de variables qui n'ont 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 :

$$\text{minutes} = \alpha + \beta_1 \text{ amis} + \beta_2 \text{ heures travaillées} + \beta_3 \text{ doctorat} + \varepsilon$$

Il est évident que le fait de posséder ou pas un doctorat n'est pas une valeur numérique. Mais nous pouvons introduire (comme cela a été mentionné au chapitre 11) une variable indicatrice qui vaut 1 pour les utilisateurs ayant un doctorat et 0 pour les autres.

Le modèle

Au chapitre 14, nous avions un modèle de la forme :

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

Imaginons que chaque entrée x_i n'est pas un nombre unique, mais un vecteur de k nombres x_{i1}, …, x_{ik}. Le modèle de régression linéaire multiple fait l'hypothèse que :

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i$$

En régression linéaire multiple, le vecteur de paramètres (incluant alpha) est en général désigné par β. Nous voulons aussi introduire une constante, ce que nous ferons en ajoutant une colonne de 1 à nos données :

```
beta = [alpha, beta_1, ..., beta_k]
```

et :

```
x_i = [1, x_i1, ..., x_ik]
```

Notre modèle devient alors :

```
def predict(x_i, beta):
    """suppose que le premier élément de chaque x_i est 1"""
    return dot(x_i, beta)
```

Dans ce cas particulier, notre variable indépendante `x` sera une liste de vecteurs, chacun de la forme :

```
[1,            # constante
 49,           # nombre d'amis
 4,            # heures de travail par jour
 0]            # n'a pas de doctorat
```

Les autres hypothèses du modèle des moindres carrés

Deux hypothèses complémentaires sont nécessaires pour que ce modèle (et notre solution) ait un sens.

La première est de 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 `beta`. Pour visualiser un cas extrême, supposons que nous avons un champ supplémentaire `num_acquaintances` dans nos données qui est exactement égal à `num_friends` pour chaque utilisateur.

Alors, quel que soit le `beta` choisi, si nous ajoutons une quantité quelconque au coefficient `num_friends` et que nous retranchons la même quantité au coefficient `num_acquaintances`, la prédiction du modèle reste inchangée. Ce qui 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 `beta` 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 plus sur le site chaque jour.

Imaginons qu'il est également vrai que :

- les personnes qui travaillent le plus passent aussi moins de temps sur le site ;
- et les personnes avec peu d'amis ont tendance à travailler plus longtemps.

Imaginons maintenant que le modèle « réel » est le suivant :

$$\text{minutes} = \alpha + \beta_1 \text{ amis} + \beta_2 \text{ heures travaillées} + \varepsilon$$

et qu'il existe une corrélation positive entre le temps de travail et les amis. Dans ce cas, quand nous minimisons les erreurs du modèle à une variable :

$$\text{minutes} = \alpha + \beta_1 \text{ amis} + \varepsilon$$

nous sous-estimons β_1.

Pensez à ce qui arriverait si nous faisions des prédictions à partir du modèle à une variable avec la valeur « réelle » de β_1 (c'est-à-dire la valeur qui est trouvée en minimisant les erreurs de ce que nous appelons le modèle « réel »). Les prédictions tendraient à être trop petites pour les utilisateurs qui travaillent beaucoup et trop grandes 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 trop petites pour les utilisateurs qui ont beaucoup d'amis et trop grandes pour ceux qui ont peu d'amis.

Le résultat, c'est que 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 avec une variable 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.

Ajuster le modèle

Comme nous l'avons fait précédemment avec le modèle linéaire simple, nous allons choisir `beta` pour minimiser la somme des carrés des erreurs. Il n'est pas facile de trouver une solution à la main, aussi allons-nous faire appel à la descente de gradient. Commençons par créer une fonction erreur à minimiser. Pour la descente du gradient stochastique, nous voulons seulement l'erreur au carré qui correspond à la prédiction d'une valeur unique :

```
def error(x_i, y_i, beta):
    return y_i - predict(x_i, beta)

def squared_error(x_i, y_i, beta):
    return error(x_i, y_i, beta) ** 2
```

Si vous maîtrisez le calcul infinitésimal, vous pouvez calculer :

```
def squared_error_gradient(x_i, y_i, beta):
      """le gradient (par rapport à beta) correspondant au ième terme du carré de
l'erreur"""
    return [-2 * x_ij * error(x_i, y_i, beta)
            for x_ij in x_i]
```

Sinon, il va falloir me faire confiance.

À ce stade, nous sommes prêts à chercher le bêta optimal avec la descente du gradient stochastique :

```
def estimate_beta(x, y):
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(squared_error,
                               squared_error_gradient,
                               x, y,
                               beta_initial,
                               0.001)

random.seed(0)
beta = estimate_beta(x, daily_minutes_good) # [30.63, 0.972, -1.868, 0.911]
```

Ce qui veut dire que notre modèle ressemble à :

minutes = 30,63 + 0,972 amis – 1,868 heures travaillées + 0,911 doctorat

L’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 avoir 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 est égale au produit de *amis* et *heures travaillées*. Ainsi, le coefficient *heures travaillées* peut augmenter (ou diminuer) quand 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 les amis en plus vous font passer moins de temps sur le site. (Peut-être qu’avec trop d’amis, l’expérience est simplement trop bouleversante ?) Nous pourrions essayer de capturer cette réalité dans notre modèle en ajoutant une nouvelle variable qui est le carré du nombre d’amis.

Chaque fois que nous ajoutons une variable, nous devons nous poser la question : 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.

Le bon ajustement du modèle

Une fois de plus, nous allons examiner le R^2 qui a désormais atteint 0,68 :

```
def multiple_r_squared(x, y, beta):
    sum_of_squared_errors = sum(error(x_i, y_i, beta) ** 2
                                for x_i, y_i in zip(x, y))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(y)
```

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 fixé 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 soit assez bien ajustée à nos données, mais si certaines des variables indépendantes sont corrélées (ou non pertinentes) leurs coefficients ne voudront peut-être pas dire 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) peut utiliser l'algèbre linéaire pour déterminer l'erreur standard de chaque coefficient. Plus elle est grande, moins notre modèle est fiable pour ce coefficient. Malheureusement, nous ne sommes pas équipés pour effectuer ce calcul d'algèbre linéaire à partir de rien.

Digression : l'amorce *(Bootstrap)*

Soit un échantillon de `n` données élémentaires générées suivant une distribution quelconque (inconnue de nous) :

```
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 toutes les données de l'échantillon sont très proches de 100, il est vraisemblable que la médiane sera elle aussi proche de 100. Si environ la moitié des données est proche de 0 et l'autre proche de 200, alors nous ne pouvons pas 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 et examiner la distribution de ces médianes. Mais en général, c'est impossible. À la place, nous pouvons « amorcer » de nouveaux jeux de données en choisissant *n* données élémentaires avec remplacement à partir de nos données et calculer les médianes de ces jeux de données de synthèse :

```
def bootstrap_sample(data):
    """échantillon aléatoire de len(data) éléments avec remplacement"""
    return [random.choice(data) for _ in data]


def bootstrap_statistic(data, stats_fn, num_samples):
       """évalue stats_fn sur les num_samples 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)])
```

Si vous calculez chaque médiane, les deux sont très proches de 100. Cependant,

si vous regardez ceci :

```
bootstrap_statistic(close_to_100, median, 100)
```

vous verrez seulement des nombres réellement proches de 100. Alors que si vous regardez cela :

```
bootstrap_statistic(far_from_100, median, 100)
```

vous verrez beaucoup de nombres proches de 0 et beaucoup d'autres proches de 200.

L'écart-type du premier jeu de médianes est proche de 0, alors que l'écart-type du second jeu de médianes est proche de 100. (Ce cas extrême serait assez facile à comprendre en examinant les données une à une, mais en général ce n'est pas aussi évident.)

Les erreurs standards des coefficients de régression

Nous pouvons adopter la même démarche pour estimer les erreurs standards de nos coefficients de régression. Nous prenons à plusieurs reprises un échantillon `bootstrap_sample` à partir duquel nous estimons `beta`. 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 ne pouvons pas avoir confiance dans nos estimations.

La seule subtilité est 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 prises dans le même échantillon. Donc `bootstrap_sample` retourne une liste de paires `(x_i, y_i)` que nous devrons rassembler dans `x_sample` et `y_sample` :

```
def estimate_sample_beta(sample):
    """l'échantillon est une liste de paires (x_i, y_i)"""
    x_sample, y_sample = zip(*sample) # astuce magique pour dézipper
    return estimate_beta(x_sample, y_sample)

random.seed(0) # pour obtenir les mêmes résultats que moi

bootstrap_betas = bootstrap_statistic(zip(x, daily_minutes_good),
                                      estimate_sample_beta,
                                      100)
```

Après cela, nous pourrons estimer l'écart-type de chaque coefficient :

```
bootstrap_standard_errors = [
   standard_deviation([beta[i] for beta in bootstrap_betas])
   for i in range(4)]

# [1.174,   # constante, erreur mesurée = 1,19
# 0.079,    # nombre d'amis,erreur mesurée = 0,080
# 0.131,    # heures travaillées,erreur mesurée = 0,127
# 0.990]    # doctorat,erreur mesurée = 0,998
```

Nous pouvons nous en servir pour tester des hypothèses telles que « est-ce que β_i est égal à zéro ? » Avec l'hypothèse nulle $\beta_i = 0$ (et avec nos autres hypothèses sur la distribution de ε_i), le résultat statistique est :

$$t_j = \hat{\beta}_j / \hat{\sigma}_j$$

Ce résultat, qui représente notre estimation de β_j divisée par notre estimation de

son erreur standard, suit une loi *t* de Student avec « *n* – *k* degrés de liberté ».

Si nous avions une fonction `students_t_cdf`, nous pourrions calculer les `p-value` pour chaque coefficient des moindres carrés. Cela nous permettrait d'indiquer avec quelle vraisemblance on pourrait observer une telle valeur si le coefficient réel valait zéro. Malheureusement, nous ne disposons pas d'une telle fonction (ce qui serait différent si nous ne partions pas de rien).

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 :

```
def p_value(beta_hat_j, sigma_hat_j):
    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 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # sinon, deux fois la probabilité de voir une valeur « plus petite »
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

p_value(30.63, 1.174)   # ~0 (constante)
p_value(0.972, 0.079)   # ~0 (nombre d'amis)
p_value(-1.868, 0.131)  # ~0 (temps de travail)
p_value(0.911, 0.990)   # 0.36 (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 la plupart des coefficients ont de toutes petites `p-value` (ce qui suggère qu'ils sont en fait non nuls), le coefficient de « 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 « β_1 égale β_2 et β_3 égale β_4 ». Cela serait possible avec un test F, mais celui-ci dépasse malheureusement le périmètre de ce livre.

La régularisation

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 aurez de coefficients non nuls, plus il sera difficile de leur donner un sens. Si l'objectif est d'expliquer un phénomène, un modèle limité à trois facteurs est parfois plus utile qu'un modèle légèrement meilleur à centaines de facteurs.

La régularisation est une méthode qui consiste à ajouter à l'erreur une pénalité qui augmente en même temps que `beta`. 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* en anglais), nous ajoutons une pénalité proportionnelle à la somme des carrés de `beta_i` – à l'exception de `beta_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, alpha):
    return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x_i, y_i, beta, alpha):
    """estime l'erreur plus la pénalité d'arête sur beta"""
    return error(x_i, y_i, beta) ** 2 + ridge_penalty(beta, alpha)
```

Nous pouvons intégrer cette pénalité à la descente de gradient de la manière habituelle :

```
def ridge_penalty_gradient(beta, alpha):
    """gradient de la pénalité d'arête seulement"""
    return [0] + [2 * alpha * beta_j for beta_j in beta[1:]]

def squared_error_ridge_gradient(x_i, y_i, beta, alpha):
    """le gradient correspondant à la i^ème erreur au carré incluant la pénalité d'arête"""
    return vector_add(squared_error_gradient(x_i, y_i, beta),
                      ridge_penalty_gradient(beta, alpha))

def estimate_beta_ridge(x, y, alpha):
    """utilise la descente de gradient pour ajuster une régression d'arête avec la
pénalité alpha"""
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(partial(squared_error_ridge, alpha=alpha),
                               partial(squared_error_ridge_gradient, alpha=alpha),
                               x, y,
```

```
                                    beta_initial, 0.001)
```

Quand alpha est mis à zéro, il n'y plus aucune pénalité et nous obtenons le même résultat qu'auparavant :

```
random.seed(0)
beta_0 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.0)
# [30.6, 0.97, -1.87, 0.91]
dot(beta_0[1:], beta_0[1:]) # 5.26
multiple_r_squared(x, daily_minutes_good, beta_0) # 0.680
```

Quand nous augmentons `alpha`, l'ajustement du modèle se dégrade, mais `beta` diminue :

```
beta_0_01 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.01)
# [30.6, 0.97, -1.86, 0.89]
dot(beta_0_01[1:], beta_0_01[1:]) # 5.19
multiple_r_squared(x, daily_minutes_good, beta_0_01) # 0.680

beta_0_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.1)
# [30.8, 0.95, -1.84, 0.54]
dot(beta_0_1[1:], beta_0_1[1:]) # 4.60
multiple_r_squared(x, daily_minutes_good, beta_0_1) # 0.680

beta_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=1)
# [30.7, 0.90, -1.69, 0.085]
dot(beta_1[1:], beta_1[1:]) # 3.69
multiple_r_squared(x, daily_minutes_good, beta_1) # 0.676

beta_10 = estimate_beta_ridge(x, daily_minutes_good, alpha=10)
# [28.3, 0.72, -0.91, -0.017]
dot(beta_10[1:], beta_10[1:]) # 1.36
multiple_r_squared(x, daily_minutes_good, beta_10) # 0.573
```

En particulier, le coefficient de « doctorat » disparaît lorsque nous augmentons la pénalité, ce qui est en accord avec notre résultat précédent selon lequel il n'est pas significativement différent de zéro.

Note

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 de 100 et se trouve brusquement pénalisé davantage, bien que le modèle reste le même.

Une autre méthode est la régression Lasso, qui utilise la pénalité :

```
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, la pénalité 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 utilisable dans le cadre d'une descente du gradient, ce qui veut dire que nous ne pourrons 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 dans Wikipédia.
- sickit-learn dispose d'un module `linear_model` qui fournit un modèle simple `LinearRegression` similaire au nôtre, ainsi que des régressions d'arête (`Ridge`), de `Lasso` et d'autres types de régularisation.
- Statsmodels est un autre module Python qui contient (entre autres trésors) des modèles de régression linéaire.

16

Régression logistique

Beaucoup pensent que la ligne est ténue entre le génie et la folie. Je ne pense pas que c'est une ligne ténue, je pense que c'est un fossé 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 revoir cette question dans ce chapitre.

Le problème

Nous disposons d'un jeu de données anonymisées concernant 200 utilisateurs avec pour chacun le salaire, le nombre d'années d'expérience comme *data scientist* et l'information sur le choix d'un compte premium (figure 16-1). Comme pour toutes les variables de type catégorie, nous représenterons la variable de dépendance sous la forme 0 (pas de compte premium) ou 1 (compte premium).

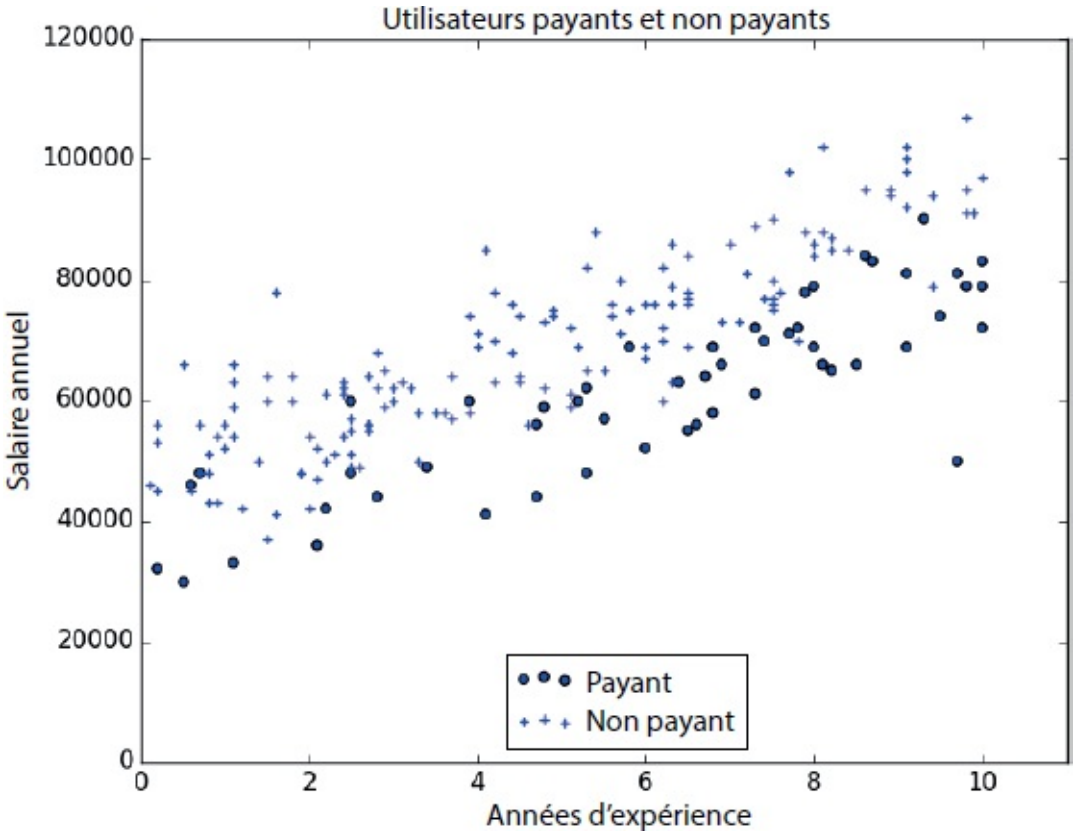
Comme d'habitude, nos données sont dans une matrice dont chaque ligne est une liste `[experience, salary, paid_account]`. Convertissons-les au format qui nous intéresse :

```
x = [[1] + row[:2] for row in data]  # chaque élément est [1, experience, salary]
y = [row[2] for row in data]         # chaque élément est paid_account
```

Une première tentative évidente serait d'utiliser une régression linéaire afin de trouver le meilleur modèle :

$$\text{paid account} = \beta_0 + \beta_1 \text{ experience} + \beta_2 \text{ salary} + \varepsilon$$

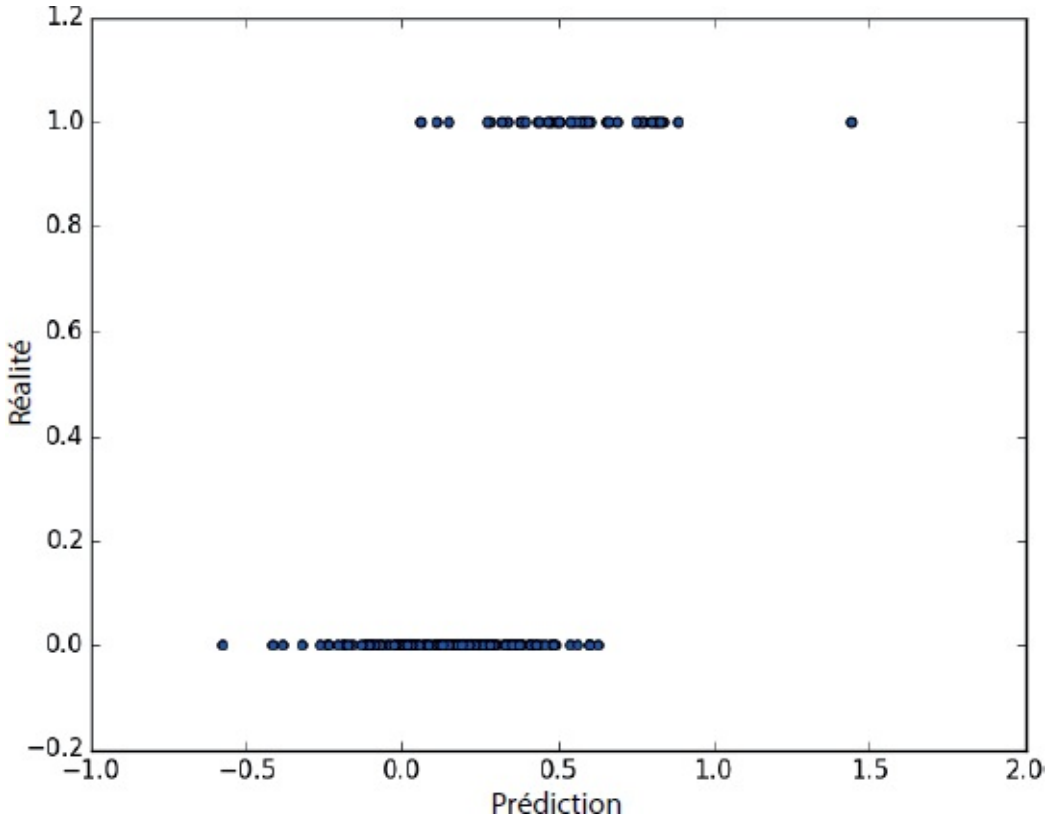
Figure 16–1
Utilisateurs payants et non payants



Rien ne nous empêche de choisir cette modélisation. Les résultats sont présentés en figure 16-2.

```
rescaled_x = rescale(x)
beta = estimate_beta(rescaled_x, y) # [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_x]


plt.scatter(predictions, y)
plt.xlabel("Prédiction")
plt.ylabel("Réalité")
plt.show()
```

Figure 16–2
Utilisation de la régression linéaire pour prédire les comptes premium



Mais cette méthode apporte son lot de problèmes immédiats.

- Nous aimerions que nos prédictions soient 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 : un résultat de 0,25 signifie 25 % de chance d'être membre payant. Mais les résultats du modèle linéaire peuvent être de grands nombres positifs ou négatifs qui ne sont pas évidents à interpréter. En fait, 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*. Mais ici, le coefficient pour la variable *experience* est de 0,43, indiquant que davantage d'expérience conduit à une plus grande probabilité de souscription au 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 `beta` est biaisée.

Ce que nous aimerions, à la place, c'est que les grandes valeurs positives de

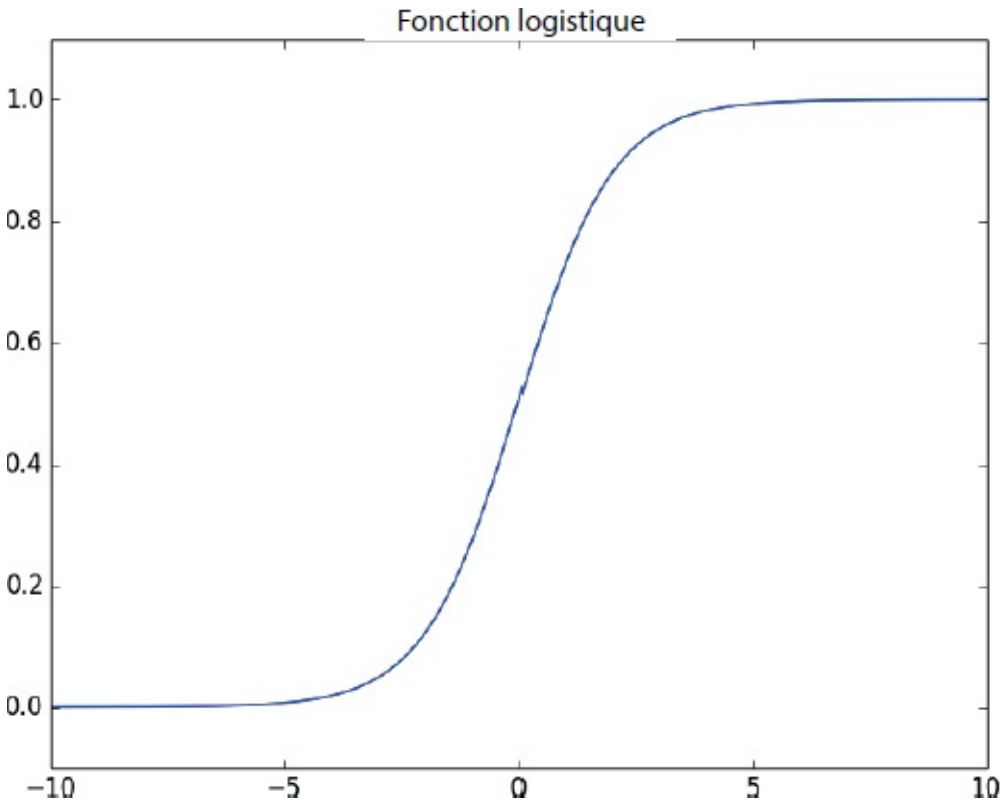
`dot(x_i, beta)` correspondent à des probabilités proches de 1 et que les grandes valeurs négatives correspondent à des probabilités proches de 0. Cela est possible à condition d'appliquer une autre fonction au résultat.

La fonction logistique

Dans le cas de la régression logistique, nous faisons appel à la fonction logistique, représentée en figure 16-3 :

```
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

Figure 16–3
La fonction logistique



Quand ses entrées sont grandes et positives, elle tend vers 1. Et quand elles sont grandes et négatives, elle tend vers 0. De plus, sa dérivée possède la propriété intéressante de pouvoir s'écrire sous cette forme :

```
def logistic_prime(x):
     return logistic(x) * (1 - logistic(x))
```

Nous utiliserons bientôt cette propriété pour ajuster un modèle :

$$y_i = f(x_i\beta) + \varepsilon_i$$

dans lequel *f* est la fonction logistique.

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 certain β, 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 y_i peut s'écrire :

$$p(y_i|x_i,\beta) = f(x_i\beta)^{y_i}(1 - f(x_i\beta))^{1-y_i}$$

puisque si y_i est égal à 0, elle vaut :

$$1 - f(x_i\beta)$$

et si y_i est égal à 1 :

$$f(x_i\beta)$$

Il s'avère qu'il est plus facile de maximiser le logarithme de la fonction de vraisemblance :

$$\log L(\beta \mid x_i,y_i) = y_i \log f(x_i\beta) + (1 - y_i) \log(1 - f(x_i\beta))$$

Comme log est une fonction strictement croissante, tout `beta` qui maximise le log de la vraisemblance maximise aussi la fonction de vraisemblance, et vice versa.

```
def logistic_log_likelihood_i(x_i, y_i, beta):
    if y_i == 1:
        return math.log(logistic(dot(x_i, beta)))
    else:
        return math.log(1 - logistic(dot(x_i, beta)))
```

Si nous supposons que des données élémentaires différentes sont indépendantes les unes des autres, la fonction de vraisemblance globale est simplement le produit des fonctions de vraisemblance individuelles. Ce qui signifie que le log de la vraisemblance globale est la somme des logs des

fonctions de vraisemblance individuelles :

```
def logistic_log_likelihood(x, y, beta):
    return sum(logistic_log_likelihood_i(x_i, y_i, beta)
               for x_i, y_i in zip(x, y))
```

Et maintenant, un peu de calcul infinitésimal pour obtenir le gradient :

```
def logistic_log_partial_ij(x_i, y_i, beta, j):
    """ici i est l'index de la donnée élémentaire,
    j l'index de la dérivée"""

    return (y_i - logistic(dot(x_i, beta))) * x_i[j]

def logistic_log_gradient_i(x_i, y_i, beta):
    """le gradient du log de probabilité correspondant à la i^ème donnée élémentaire"""

    return [logistic_log_partial_ij(x_i, y_i, beta, j)
            for j, _ in enumerate(beta)]

def logistic_log_gradient(x, y, beta):
    return reduce(vector_add,
                  [logistic_log_gradient_i(x_i, y_i, beta)
                   for x_i, y_i in zip(x,y)])
```

Nous avons désormais toutes les pièces nécessaires.

L’application du modèle

Séparons nos données en jeu d’apprentissage et jeu de test :

```
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_x, y, 0.33)

# pour maximiser le log de la vraisemblance sur les données d’apprentissage
fn = partial(logistic_log_likelihood, x_train, y_train)
gradient_fn = partial(logistic_log_gradient, x_train, y_train)

# choisir un point de départ aléatoire
beta_0 = [random.random() for _ in range(3)]

# et maximiser avec la descente du gradient
beta_hat = maximize_batch(fn, gradient_fn, beta_0)
```

Une autre approche consiste à utiliser la descente du gradient stochastique :

```
beta_hat = maximize_stochastic(logistic_log_likelihood_i,
                               logistic_log_gradient_i,
                               x_train, y_train, beta_0)
```

Dans les deux cas, nous obtenons approximativement :

```
beta_hat = [-1.90, 4.05, -3.87]
```

Ce sont là les coefficients pour les données après changement d’échelle, mais nous pouvons revenir aux données d’origine :

```
beta_hat_unscaled = [7.61, 1.42, -0.000249]
```

Malheureusement, ces coefficients ne sont pas aussi aisés à interpréter que ceux de la régression linéaire classique. Toutes choses égales par ailleurs, une année d’expérience de plus ajoute 1,42 en entrée de la fonction logistique et 10 000 $ de salaire de plus ajoute 2,49 en entrée de la fonction logistique.

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 beaucoup augmenter la probabilité.

Tout ce que nous pouvons affirmer, toutes choses 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. (Ceci était déjà plus ou moins apparent sur le graphique des données.)

Le bon ajustement du modèle

Nous n'avons pas encore utilisé les données de test que nous avions mises de côté. Que se passe-t-il si nous faisons une prédiction de compte payant dès que la probabilité dépasse 0,5 :

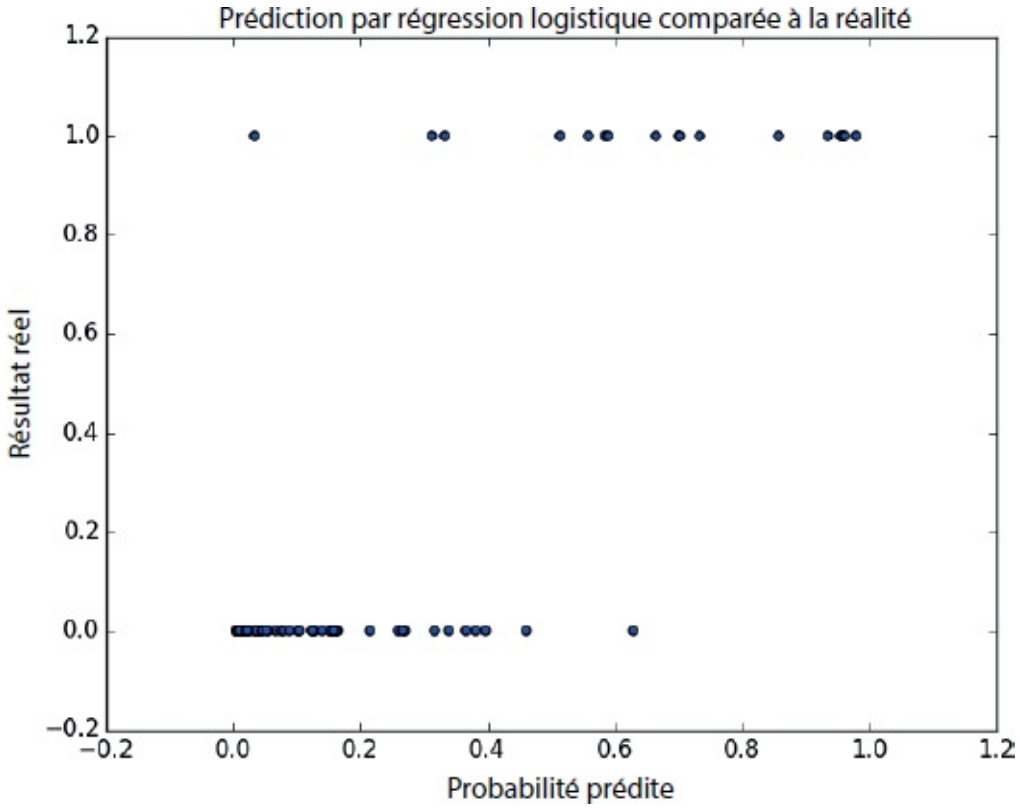
```
true_positives = false_positives = true_negatives = false_negatives = 0

for x_i, y_i in zip(x_test, y_test):
    predict = logistic(dot(beta_hat, x_i))

    if y_i == 1 and predict >= 0.5: # TP: payant et nous avions prédit payant
        true_positives += 1
    elif y_i == 1:                  # FN: payant et nous avions prédit non payant
        false_negatives += 1
    elif predict >= 0.5:         # FP: non payant et nous avions prédit payant
        false_positives += 1
    else:                        # TN: non payant et nous avions prédit non payant
        true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

Nous obtenons une précision de 93 % (« quand nous prédisons un compte payant, nous avons raison dans 93 % des cas ») et un rappel de 82 % (« quand un utilisateur a un compte payant, nous prédisons un compte payant dans 82 % des cas »). Ce sont là des résultats plus qu'honorables.

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_hat, x_i)) for x_i in x_test] plt.scatter(predictions,
y_test)
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


Prédiction par régression logistique comparée à la réalité
Résultat réel
Probabilité prédite
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

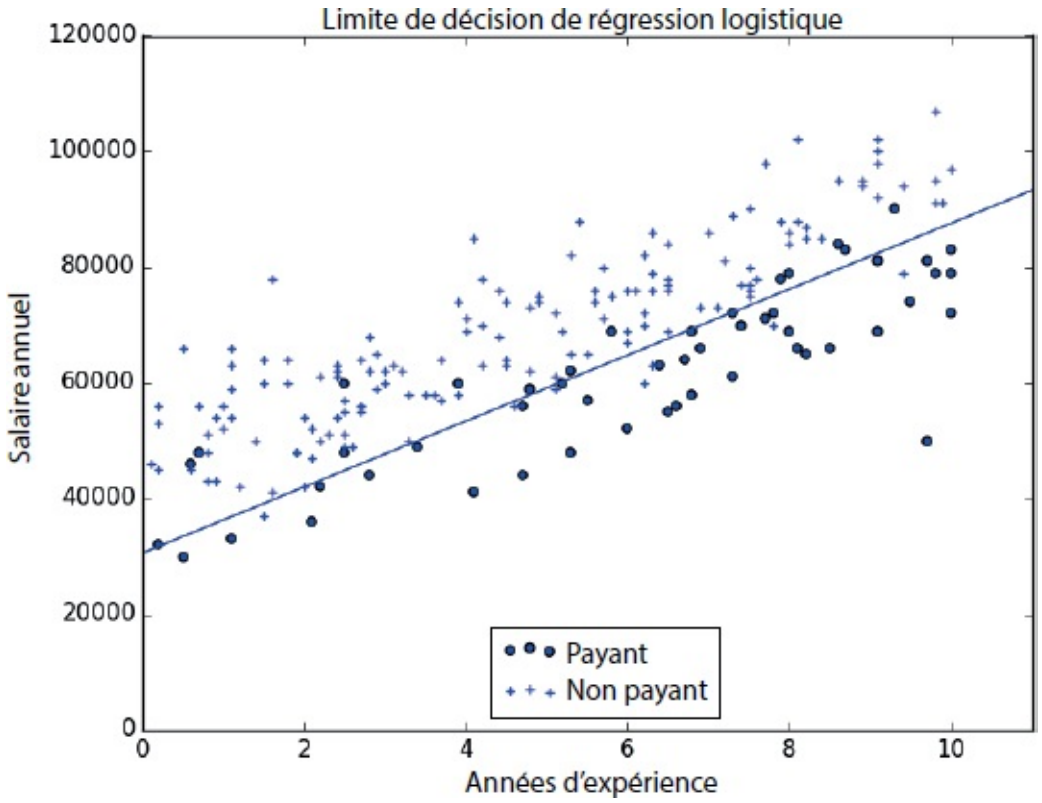
Les machines à vecteurs support

Le jeu de points pour lequel `dot(beta_hat, 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).

Cette limite est un hyperplan qui sépare l'espace des paramètres en deux demi-espaces. Chaque demi-espace correspond aux prédictions compte payant et compte non payant. Nous l'avons trouvé par un effet collatéral du modèle logistique le plus probable.

Il existe une autre approche de la classification qui consiste à chercher l'hyperplan qui sépare « le mieux » les classes dans les données d'apprentissage. C'est l'idée qui sous-tend les machines à vecteurs support : on recherche l'hyperplan qui maximise la distance entre les points les plus proches dans chaque classe (figure 16-6).

Figure 16–5
Utilisateurs payants et non payants avec limite de décision



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, ce qui est encore un autre problème. Dans notre jeu de données « qui paye », il n'existe aucune ligne séparant parfaitement les utilisateurs payants des utilisateurs non payants.

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.

Il est évident qu'il n'existe aucun hyperplan qui sépare les exemples positifs des exemples négatifs. Cependant, voyez ce qu'il se passe lorsqu'on projette ce jeu de données dans deux dimensions en faisant correspondre à `x` le vecteur (`x, x**2`). Tout d'un coup, il devient possible de trouver un hyperplan qui sépare les données (figure 16-8). 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 que la correspondance est difficile), nous préférons utiliser une fonction « noyau » pour calculer les produits scalaires dans l'espace de grande dimension et ensuite les utiliser pour trouver un hyperplan.

Figure 16–6
Un hyperplan de séparation

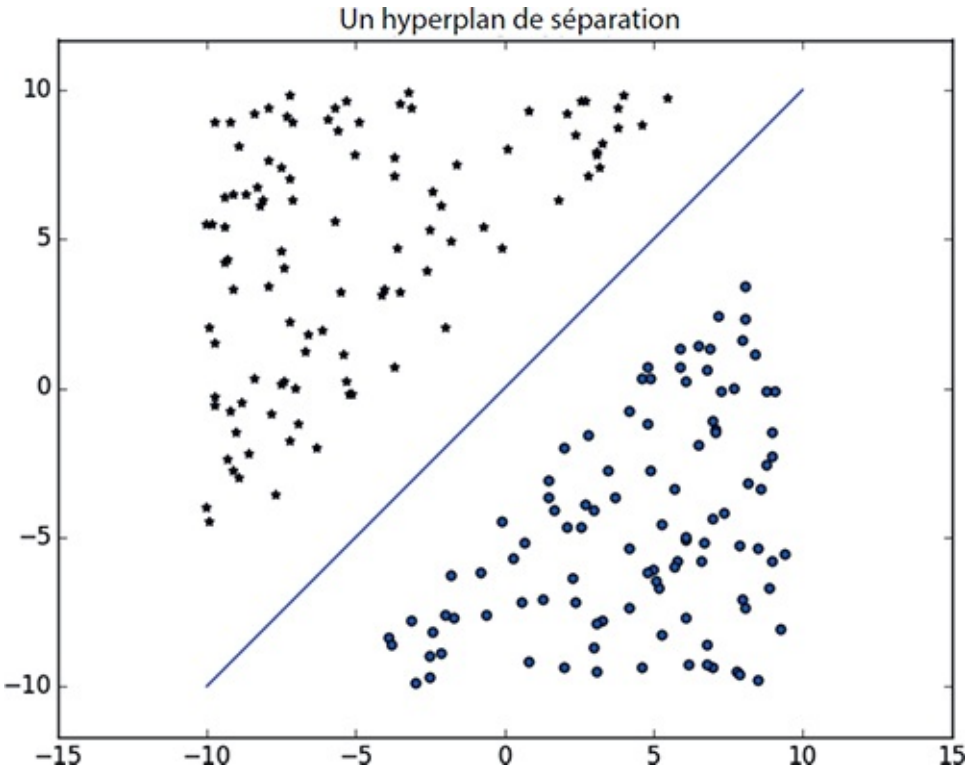
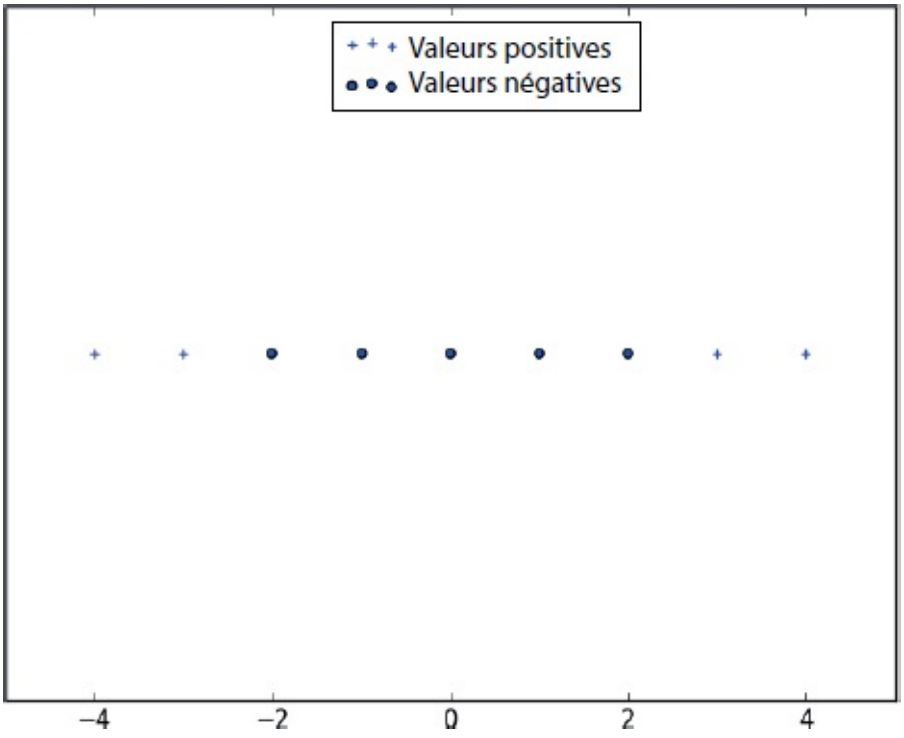
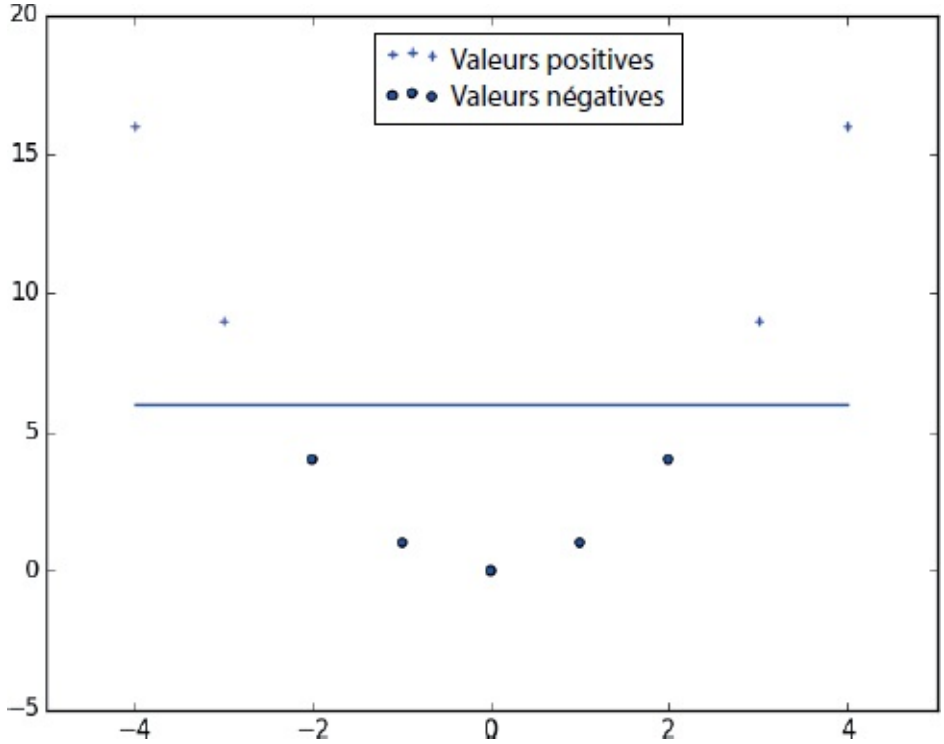


Figure 16–7
Un jeu de données à une dimension non séparable



Il est difficile (et sans doute pas très futé) de faire appel aux machines à vecteurs de support sans s'appuyer sur un logiciel d'optimisation spécialisé écrit par des personnes suffisamment expertes, aussi nous arrêterons notre traitement à ce stade.

Figure 16–8
Le jeu de données devient séparable dans un espace de plus grandes dimensions.


Valeurs positives
Valeurs négatives
20
15
10
5
0
−5
−4
−2
0
2
4

Pour aller plus loin

- scikit-learn propose des modules pour la régression logistique et les machines à vecteurs support (SVM en anglais).
- libsvm est l'implémentation de machine à vecteurs support qui est utilisée par scikit-learn. Son site web propose une documentation utile et variée sur cette famille d'algorithmes.

17

Arbres de décision

Un arbre est un mystère incompréhensible.
– Jim Woodring

Le responsable Talents de DataSciencester a interrogé de nombreux candidats à l'embauche avec des résultats mitigés. Il a collecté des jeux de données reprenant plusieurs attributs des candidats ainsi que l'opinion sur la prestation du candidat en entretien. Sa demande est la suivante : « Pourriez-vous construire un modèle qui me permettra d'identifier les candidats qui réaliseront un bon entretien de sorte que je ne perde pas mon temps lors des entretiens ? »

C'est là un joli cas d'école pour utiliser un arbre de décision, un autre de ces outils de modélisation prédictive présents dans la trousse à outils du data scientist.

Qu’est-ce qu’un arbre de décision ?

Un arbre de décision est une structure arborescente qui représente différents choix possibles et un résultat pour chaque cheminement.

Si vous déjà joué au jeu des questions, alors vous connaissez déjà les arbres de décision. Voici un exemple.

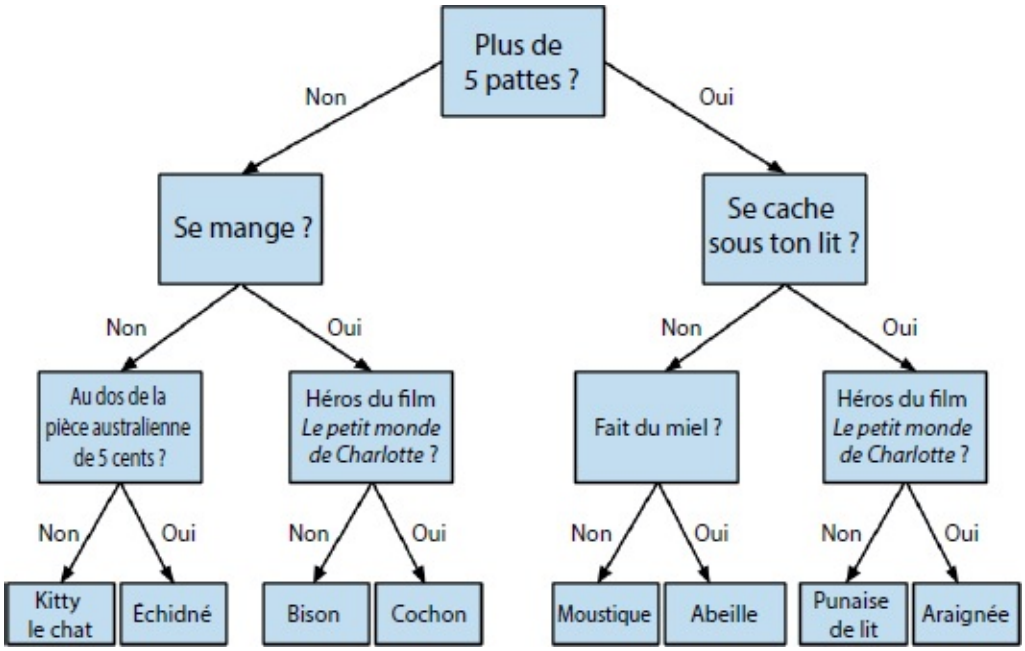
- « Je pense à un animal.
- Est-ce qu’il a plus de cinq pattes ?
- Non
- Est-ce qu’il se mange ?
- Non
- Est-ce qu’il apparaît sur le dos de la pièce australienne de 5 cents ?
- Oui
- Est-ce un échidné ?
- Oui, c’est ça ! »

Le cheminement suivi correspond à :

pas plus de 5 pattes -> ne se mange pas -> sur la pièce de 5 cents -> échidné !

dans un arbre de décision idiosyncratique (et pas vraiment exhaustif) du jeu « devinez cet animal » (figure 17-1).

Figure 17–1
Un arbre de décision « devinez cet animal »



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 classifier les données pour lesquelles il manque un attribut. En outre, trouver un arbre de décision « optimal » pour un jeu de données d'apprentissage est 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 que ce soit malgré tout 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 aux données d'apprentissage et qui ne se généralise pas bien aux données nouvelles. Voyons comment procéder pour éviter cela.

La plupart des experts divisent les arbres de décision en arbres de classification (avec comme résultats finaux des catégories) et en arbres de régression (avec 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 labellisées, ce qui devrait nous aider à en comprendre le mécanisme. Et pour faciliter les choses encore plus, nous nous limiterons à des problèmes à réponses binaires, tels que « dois-je embaucher ce candidat ? », « dois-je

afficher la publicité A ou B à ce visiteur de mon site web ? » ou « la nourriture que j'ai trouvée dans le réfrigérateur au bureau va-t-elle me rendre malade ? ».

L’entropie

Pour construire un arbre de décision, nous devons sélectionner les questions à poser et décider de l’ordre dans lequel nous les présenterons. À chaque étape, il y a 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. Nous n’avons pas éliminé la possibilité 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 à réponse oui/non pour laquelle « oui » correspond toujours à des résultats vrais et « non » correspond toujours à des résultats faux (ou vice versa), alors cette question est à retenir. Au contraire, 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 » est mesurée 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 de données S dans lequel chaque membre est labellisé comme appartenant à une des classes de la liste finie C_1, …, 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.

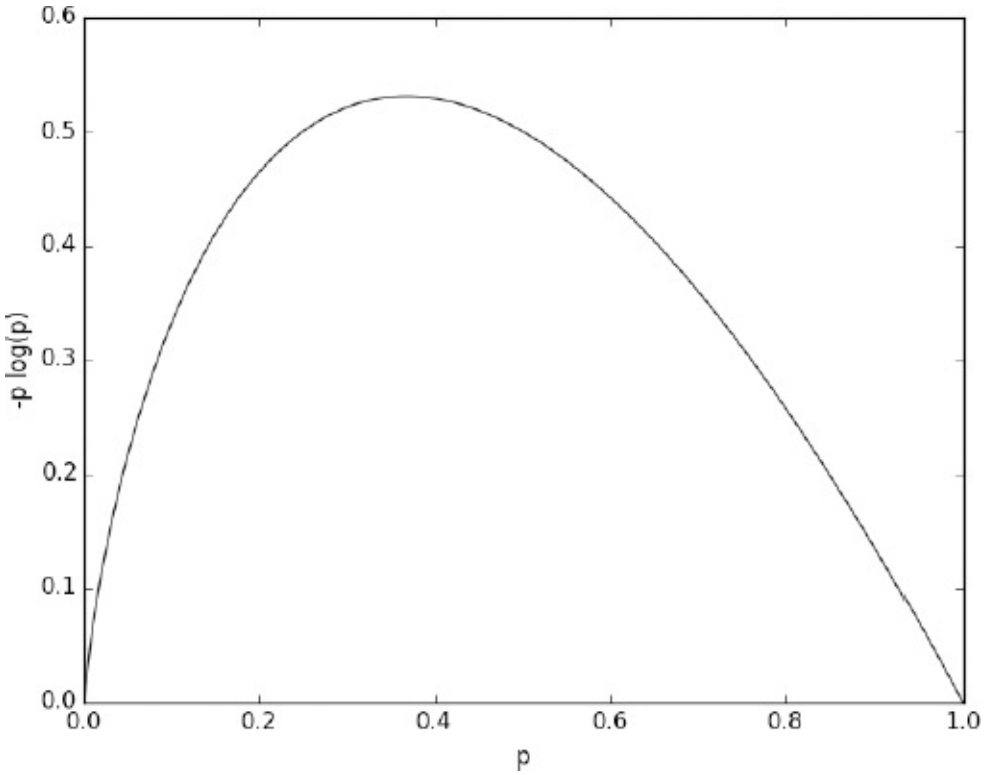
En termes mathématiques, si p_i est la proportion de données labellisées comme appartenant à la classe c_i, nous définirons l’entropie comme :

$$H(S) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n$$

avec pour convention standard que 0 log 0 = 0.

Sans trop entrer dans les détails, il suffit de se garder en tête que chaque terme $-p_i \log_2 p_i$ est non négatif et proche de zéro précisément quand p_i est proche soit de zéro, soit de un (figure 17-2).

Figure 17–2
Graphique de -p log p



Cela veut dire que l'entropie est faible quand chaque p_i est proche de 0 ou 1 (autrement dit, quand la plupart des données sont 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 :

```
def entropy(class_probabilities):
    """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)                  # ignore les probabilités à zéro
```

Nos données sont des paires (entrée, label), 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 quel label est associé à quelle probabilité, mais que nous nous intéressons uniquement aux valeurs des probabilités :

```
def class_probabilities(labels):
    total_count = len(labels)
```

```
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)
```

L’entropie d’une partition

Jusqu’à présent, nous avons déterminé l’entropie (pensez : « incertitude ») d’un seul jeu de données labellisées. Maintenant, chaque étape de l’arbre décisionnel nécessite de poser une question dont la réponse sépare les données dans un ou (espérons-le) plusieurs sous-ensembles. Par exemple, notre question « a-t-il plus de cinq pattes » sépare les animaux en deux groupes : ceux qui ont plus de cinq pattes (comme les araignées) et ceux qui n’ont pas plus de cinq pattes (comme les échidnés).

De même, nous aimerions connaître l’entropie qui résulte de la partition d’un jeu de données d’une certaine façon. Nous voulons une partition à faible entropie si celle-ci sépare les données en sous-ensembles qui ont 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 de cinq cents » était assez stupide (mais tout de même plutôt réussie !) car elle a séparé les animaux restants à ce stade en S_1 = {échidné} et S_2 = {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, …, S_m contenant des proportions q_1, …, q_m des données, et si nous calculons l’entropie de la partition sous la forme d’une somme pondérée, alors :

$$H = q_1 H(S_1) + \ldots + q_m H(S_m)$$

que nous pouvons implémenter de la manière suivante :

```
def partition_entropy(subsets):
    """chercher l’entropie de cette partition des données en sous-ensembles,
    les sous-ensembles sont une liste de listes de données labellisées"""

    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )
```

Note

Un des problèmes de cette méthode vient du fait que si on partitionne les données selon un attribut comportant beaucoup de valeurs différentes, cela nous conduira à une entropie très basse à cause du

surajustement. Par exemple, imaginons que vous travaillez dans une banque et que 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 de chaque membre (SSN). Si vous partitionnez vos données selon le SSN, vous obtiendrez des sous-ensembles d'une seule personne, dont chacun aura nécessairement une entropie de 0. Un modèle de type SSN 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 regrouper si c'est pertinent) les attributs qui ont un grand nombre de valeurs possibles.

La construction d’un arbre de décision

Le responsable vous fournit les données recueillies en entretien. Ces données sont composées de paires (entrée, label) conformes à vos spécifications. Chaque entrée est un dict représentant les attributs du candidat et chaque label est soit `True` (le candidat a été bon en entretien) soit `False` (le candidat a été assez mauvais). En particulier, vous connaissez le niveau de chaque candidat, son langage préféré, son degré d’activité sur Twitter et s’il a un doctorat :

```
inputs = [
   ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'no'},     False),
   ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'yes'},    False),
   ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'no'},       True),
   ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'no'},    True),
   ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'no'},        True),
   ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'yes'},      False),
   ({'level':'Mid', 'lang':'R', 'tweets':'yes', 'phd':'yes'},          True),
   ({'level':'Senior', 'lang':'Python', 'tweets':'no', 'phd':'no'},   False),
   ({'level':'Senior', 'lang':'R', 'tweets':'yes', 'phd':'no'},        True),
   ({'level':'Junior', 'lang':'Python', 'tweets':'yes', 'phd':'no'},   True),
   ({'level':'Senior', 'lang':'Python', 'tweets':'yes', 'phd':'yes'},  True),
   ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'yes'},      True),
   ({'level':'Mid', 'lang':'Java', 'tweets':'yes', 'phd':'no'},        True),
   ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'yes'},   False)
]
```

Notre arbre se compose de nœuds décisionnels (qui posent une question et nous orientent différemment selon la réponse) et de nœuds terminaux, ou feuilles (qui nous fournissent une prédiction). L’algorithme ID3 va nous permettre de le construire facilement. Cet algorithme opère de la manière suivante. Soit des données avec label et une liste d’attributs à leur affecter.

- Si les données ont toutes le même label, créez un nœud terminal pour prédire ce label et arrêtez.
- Si la liste d’attributs est vide (il n’y a plus de questions à poser), créez un nœud terminal pour prédire le label le plus courant et arrêtez.
- Sinon, essayez de séparer les données en fonction de chacun des attributs.
- Choisissez la partition à plus faible entropie.
- Ajoutez un nœud décisionnel en vous appuyant sur l’attribut choisi.
- Répétez 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 dans le contexte considéré. Pour un jeu de données particulier, il existe

peut-être un arbre meilleur 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.

Livrons-nous à cette exploration à la main sur les données issues des entretiens. Le jeu de données contient des labels `False` et `True` et nous avons quatre attributs pour effectuer 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 :

```
def partition_by(inputs, attribute):
   """chaque input est une paire (attribute_dict, label).
      retourne un dict : attribute_value -> inputs"""
      groups = defaultdict(list)
      for input in inputs:
          key = input[0][attribute]   # récupère les valeurs de l'attribut spécifié
          groups[key].append(input)   # puis ajoute cette entrée à la liste correcte
      return groups
```

et une autre fonction qui l'utilisera pour calculer l'entropie :

```
def partition_entropy_by(inputs, attribute):
    """calcule l'entropie correspondant à la partition donnée"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

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)

# level 0,693536138896
# lang 0,860131712855
# tweets 0,788450457308
# phd 0,892158928262
```

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 possible de niveau. Tout candidat de niveau intermédiaire (`Mid`) est labellisé `True`, ce qui veut dire que le sous-arbre `Mid` est un nœud terminal qui prédit `True`. Et pour les candidats `Senior`, nous avons à la fois des valeurs `True` et `False`, donc il faut encore séparer les données :

```
senior_inputs = [(input, label)
                 for input, label in inputs if input["level"] == "Senior"]
```

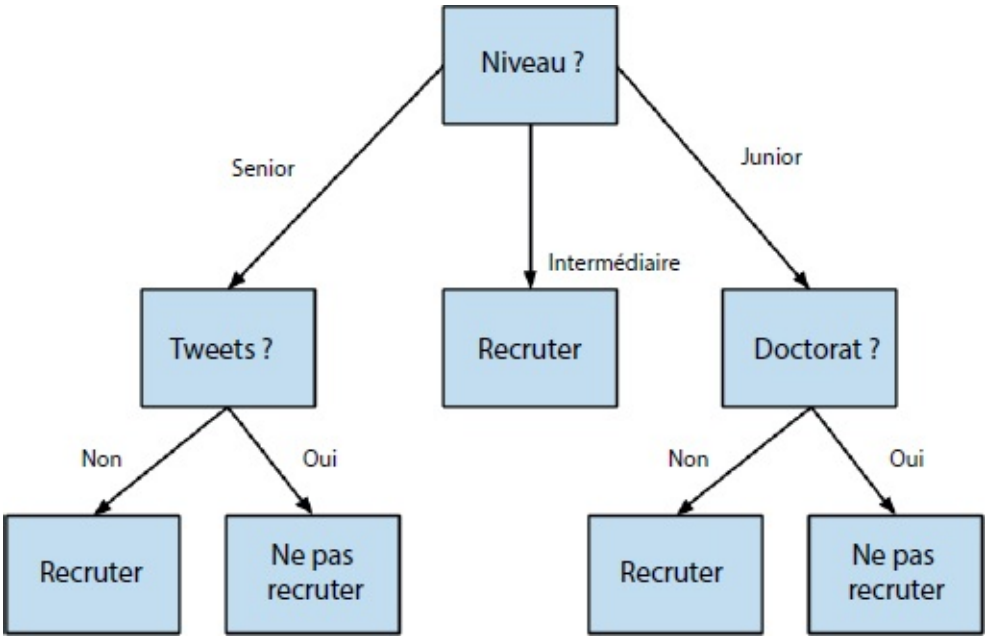
```
for key in ['lang', 'tweets', 'phd']:
    print key, partition_entropy_by(senior_inputs, key)

# lang 0,4
# tweets 0,0
# phd 0,950977500433
```

Cela nous montre que notre prochaine séparation se fera sur les tweets avec pour conséquence une partition à entropie 0. Pour les candidats de niveau `Senior`, l'attribut tweets à « Yes » donne toujours pour résultat `True` et tweets à « No » conduit toujours à `False`.

Pour finir, si nous appliquons la même méthode aux candidats `Junior`, nous devons faire la partition sur l'attribut `phd`, après quoi nous trouvons que pas de doctorat résulte toujours en `True` et un doctorat résulte toujours en `False`.

La figure 17-3 représente l'arbre de décision complet.

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. Nous définissons un arbre comme étant un des éléments suivants :

- `True`
- `False`
- et un tuple `(attribute, subtree_dict)`

Ici, `True` représente un nœud terminal (ou feuille) qui renvoie `True` pour n'importe quelle entrée, `False` représente un nœud terminal qui renvoie `False` pour n'importe quelle entrée, et un tuple représente un nœud décisionnel qui, pour chaque entrée donnée, trouve la valeur de l'attribut et classifie l'entrée selon le résultat de la décision du sous-arbre correspondant.

Avec cette représentation, notre modèle d'arbre de recrutement ressemble à ceci :

```
('level',
 {'Junior': ('phd', {'no': True, 'yes': False}),
  'Mid': True,
  'Senior': ('tweets', {'no': False, 'yes': 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 » ? Nous allons ajouter une clé `None` qui prédit le label le plus commun. (Ce serait bien sûr une mauvaise idée si `None` était une valeur qui apparaît réellement dans les données).

Avec une telle représentation, nous pouvons classer les entrées de la manière suivante :

```
def classify(tree, input):
    """catégoriser l'entrée à l'aide de l'arbre de décision"""

    # si c'est un nœud terminal, retourner la valeur
    if tree in [True, False]:
        return tree

    # sinon cet arbre se compose d'un attribut à considérer pour aiguiller la décision
    # et d'un dictionnaire dont les clés sont des valeurs de cet attribut
    # et dont les valeurs sont les sous-arbres à envisager ensuite
```

```
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)  # None si l'entrée est un attribut manquant

    if subtree_key not in subtree_dict: # si pas de sous-arbre correspondant à la clé,
        subtree_key = None              # utilisons le sous-arbre None

    subtree = subtree_dict[subtree_key] # choisir le sous-arbre adapté
    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, split_candidates=None):

    # si c'est le premier passage,
    # toutes les clés du premier passage sont candidates pour la séparation

    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # compter les True et False dans les entrées
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues

    if num_trues == 0: return False  # pas de True ? retourner un nœud terminal "False"
    if num_falses == 0: return True  # pas de False ? retourner un nœud terminal "True"

    if not split_candidates:           # s'il ne reste pas de candidat pour la séparation
        return num_trues >= num_falses # retourner le nœud terminal majoritaire

    # sinon séparer sur le meilleur attribut
    best_attribute = min(split_candidates,
                   key=partial(partition_entropy_by, inputs))
    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # construire les sous-arbres de manière récursive
     subtrees = { attribute_value : build_tree_id3(subset, new_candidates)
                  for attribute_value, subset in partitions.iteritems() }

    subtrees[None] = num_trues > num_falses    # cas par défaut

    return (best_attribute, subtrees)
```

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)

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "no"} )            # True

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "yes"} )           # False
```

Et aussi aux données avec des valeurs absentes ou inattendues :

```
classify(tree, { "level" : "Intern" } ) # True
classify(tree, { "level" : "Senior" } ) # False
```

Note

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 bon modèle pour un cas réel, nous aurions collecté davantage de données en les séparant en sous-ensembles d’apprentissage, de validation et de test.

Les forêts aléatoires *(Random Forests)*

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 d'arbres décisionnels. Cette technique consiste à construire plusieurs arbres décisionnels et à les laisser voter sur la manière de classer les entrées :

```
def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]
```

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*) (rappelez-vous la section « Digression : l'amorce », plus haut dans ce livre). Plutôt que d'entraîner chaque arbre sur toutes les entrées de l'ensemble d'apprentissage, on entraîne chaque arbre sur les résultats de `bootstrap_sample(inputs)`. Comme chaque arbre est construit à partir de données différentes, chacun sera différent des autres. (Un avantage collatéral est qu'il est possible sans tomber dans le surajustement d'utiliser les données non échantillonnées pour tester chaque arbre, ce qui signifie que vous pouvez vous permettre d'utiliser toutes vos données comme jeu d'apprentissage si vous mesurez la performance astucieusement.) Cette technique est connue sous le nom d'ensachage (*bagging* ou *bootstrap aggregating*).

Une autre source de hasard provient du changement de l'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 celui qui est le meilleur :

```
# s'il existe déjà trop peu de candidats pour le partitionnement, 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=partial(partition_entropy_by, inputs))
```

```
partitions = partition_by(inputs, best_attribute)
```

Ceci est un exemple d’une technique plus générale appelée apprentissage ensembliste *(ensemble learning)* et suivant laquelle nous combinons plusieurs modèles d’apprentissage faibles (modèles à biais élevé et variance faible) afin de produire un modèle général puissant.

Les forêts d’arbres de décision sont un des modèles les plus populaires et les plus polyvalents disponibles.

Pour aller plus loin

- scikit-learn propose plusieurs modèles d'arbres de décision. Il propose aussi un module `ensemble` qui inclut le modèle `RandomForestClassifier` ainsi que d'autres méthodes ensemblistes.
- Dans ce chapitre, nous n'avons fait que survoler les arbres de décision et leurs algorithmes. Wikipédia est un bon point de départ pour explorer plus largement ce sujet.

18

Réseaux neuronaux

J'aime le non-sens, ça réveille les neurones.
– Docteur 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 une collection 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 pas (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 peuvent résoudre de nombreux problèmes tels que la reconnaissance de l'écriture manuscrite d'une personne ou la reconnaissance faciale. Ils sont largement utilisés en apprentissage approfondi (*deep learning* en anglais), un des champs d'études les plus prometteurs de la data science. Cependant, la plupart des réseaux neuronaux sont 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 petit data scientist, ils ne seront donc probablement pas le bon choix. Un jour, quand vous vous attèlerez à la construction d'une intelligence artificielle qui mettra en avant la Singularité (permettra de faire éclore l'Humanité 2.0 de Ray Kurzweil), ce sera peut-être différent.

Le 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 :

```
def step_function(x):
    return 1 if x >= 0 else 0


def perceptron_output(weights, bias, x):
    """retourne 1 si le perceptron « se déclenche », 0 sinon"""
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

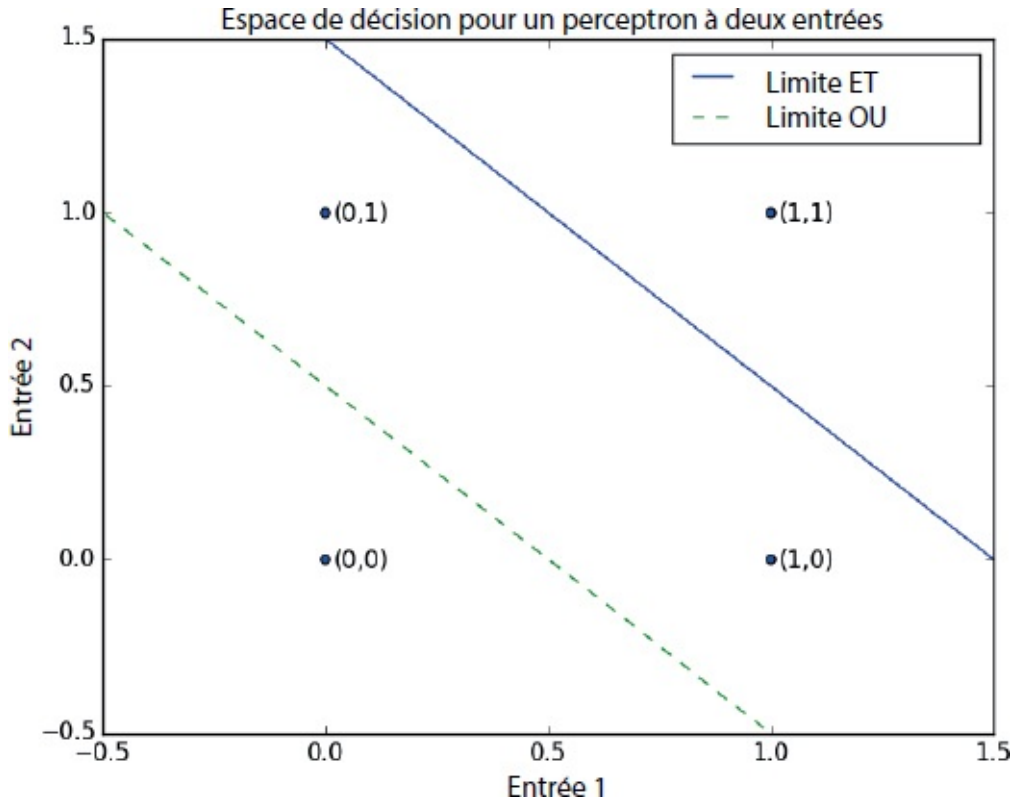
Le perceptron fait simplement la distinction entre les demi-espaces séparés par l'hyperplan des points `x` pour lesquels :

```
dot(weights,x) + bias == 0
```

Lorsqu'on choisit correctement les pondérations, les perceptrons peuvent résoudre un grand nombre de problèmes simples (figure 18-1). Par exemple, nous pouvons créer une porte logique *ET* (qui renvoie 1 si les deux entrées sont 1, ou 0 si une des entrées est 0) avec :

```
weights = [2, 2]
bias = -3
```

Figure 18–1
Espace de décision pour un perceptron à deux entrées



Si les deux entrées sont 1, le calcul revient à 2 + 2 – 3 = 1 et la sortie est 1. Si une des deux entrées seulement est 1, le calcul revient à 2 + 0 – 3 = -1 et la sortie est 0. Si les deux entrées sont 0, le calcul revient à -3 et la sortie est 0. De même, nous pouvons créer une porte logique *OU* avec :

```
weights = [2, 2]
bias = -1
```

Et nous pouvons créer une porte *NON* (à une entrée, qui convertit 0 en 1 et 1 en 0) de la sorte :

```
weights = [-2]
bias = 1
```

Cependant, il existe des problèmes qui ne peuvent tout simplement pas être résolus par un perceptron unique.

Par exemple, malgré tous vos efforts, vous ne pourrez pas utiliser un perceptron pour construire une porte logique *XOR* (*OU exclusif*) qui envoie 1 si exactement une des entrées est 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.

Les réseaux neuronaux à propagation vers l'avant *(Feed-Forward)*

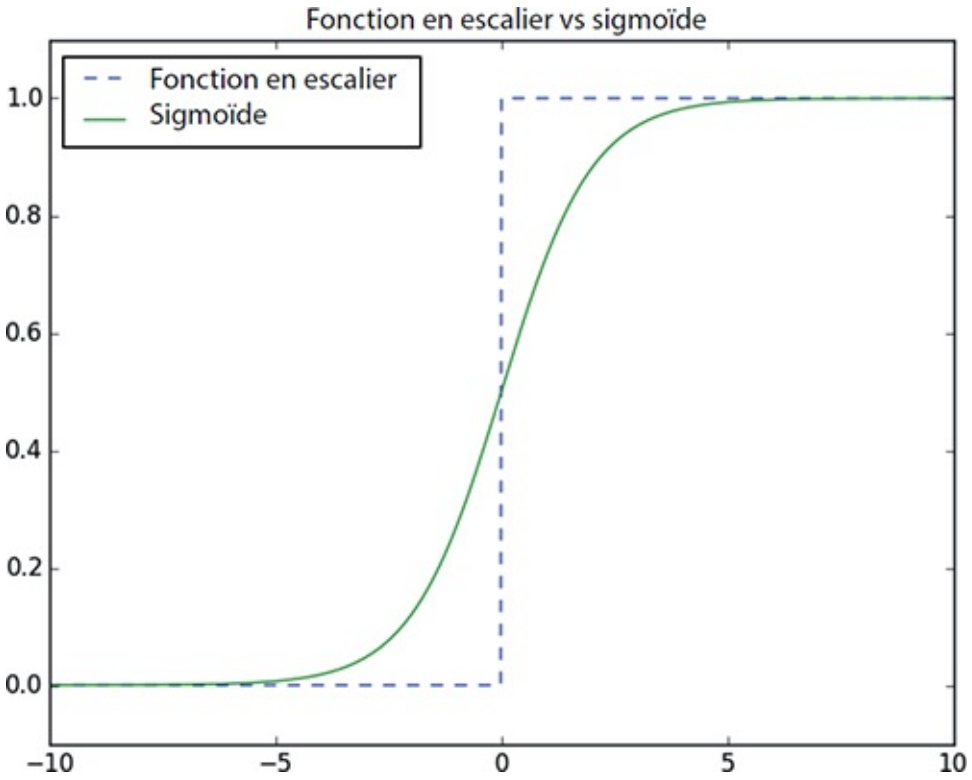
La topologie du cerveau est extrêmement compliquée. Il est courant de la simuler par un réseau neuronal idéal à propagation vers l'avant consistant en couches discrètes de neurones, chacune connectée à la suivante. On parle alors de couche d'entrée (qui reçoit les entrées et les transmet vers l'avant sans les changer), d'une ou plusieurs couches « cachées » (dont chacune est composée de neurones qui reçoivent les sorties de la couche précédente, effectuent des calculs et font suivre 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) a 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. Mais plutôt que d'appliquer la fonction `step_function` au produit pour générer la sortie, nous retournerons une approximation lissée de la fonction en escalier. En particulier, nous utiliserons la fonction sigmoïde (figure 18-2) :

```
def sigmoid(t):
    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 nouveau réseau neuronal, nous devons faire appel au calcul infinitésimal et pour cela il nous faut des fonctions lissées. La fonction en escalier n'est pas uniformément continue et la sigmoïde en est une bonne approximation.

Note

Vous vous rappelez peut-être 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 elle-même. Mais en réalité, les deux termes sont interchangeables.

Nous pouvons alors calculer la sortie de cette manière :

```
def neuron_output(weights, inputs):
    return sigmoid(dot(weights, inputs))
```

Étant donnée cette fonction, nous pouvons représenter un neurone simplement comme une liste de poids dont la longueur est supérieure de un au nombre d'entrées (à cause du biais de pondération). Ensuite, nous pouvons représenter un réseau neuronal comme une liste de couches (autres que couche d'entrée) dont chacune est une liste des neurones qu'elle contient.

C'est-à-dire que nous représenterons un réseau neuronal comme une liste

(couches) de listes (neurones) de listes (poids).

Avec une telle représentation, l'utilisation du réseau neuronal devient plutôt simple :

```
def feed_forward(neural_network, input_vector):
    """accepte un réseau neuronal en entrée
    (représenté comme une liste de listes de listes de poids)
    et retourne le résultat par propagation de l'entrée vers
    les couches supérieures"""

    outputs = []

    # traiter une couche à la fois
    for layer in neural_network:
       input_with_bias = input_vector + [1]             # ajoute un biais d'entrée
       output = [neuron_output(neuron, input_with_bias) # calcule le résultat
                 for neuron in layer]                   # pour chaque neurone
       outputs.append(output)                           # et s'en souvient

      # ensuite l'entrée de la couche suivante est la sortie de cette couche-ci
      input_vector = output

   return outputs
```

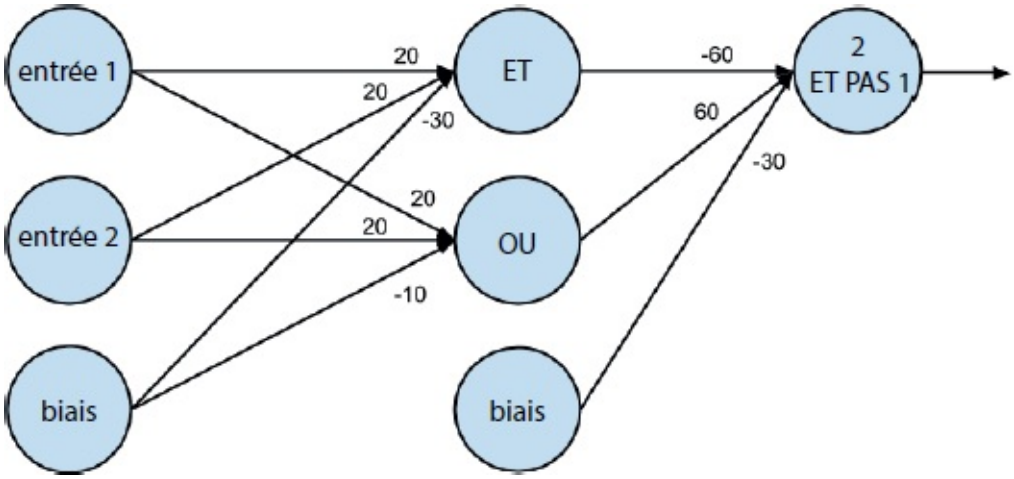
Il devient alors facile de construire la porte *XOR*, qui était impossible à créer avec un simple perceptron. Il suffit pour cela de grossir l'échelle des poids de telle sorte que les résultats (`neuron_outputs`) soient très proches de 0 ou de 1 :

```
xor_network = [# couche cachée
               [[20, 20, -30],   # neurone ET
                [20, 20, -10]],  # neurone OU
               # couche sortie
               [[-60, 60, -30]]] # neurone « 2e entrée mais pas 1re entrée »
for x in [0, 1]:
    for y in [0, 1]:
        # la propagation vers l'avant produit les sorties de chaque neurone
        # feed_forward[-1] retourne les résultats des neurones de la couche sortie
        print x, y, feed_forward(xor_network,[x, y])[-1]

# 0 0 [9.38314668300676e-14]
# 0 1 [0.9999999999999059]
# 1 0 [0.9999999999999059]
# 1 1 [9.383146683006828e-14]
```

En utilisant une couche cachée, nous pouvons construire les sorties d'un neurone *ET* et d'un neurone *OU* et le fournir en entrée d'un neurone « deuxième entrée mais pas première entrée ». Le résultat est un réseau qui réalise des « ou, mais pas et » ce qui correspond précisément au *XOR* (*OU exclusif* de la figure 18-3).

Figure 18–3

Un réseau neuronal pour XOR



La rétropropagation

Les réseaux neuronaux ne sont généralement pas créés à la main. D'une part parce que 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, parce que le plus souvent nous ne pouvons 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 populaire, pour cette tâche, est l'algorithme de rétropropagation, qui présente des similarités avec la descente de gradient rencontrée plus haut.

Imaginons un jeu de données d'apprentissage composé de vecteurs d'entrée et des vecteurs de sortie correspondants. Par exemple, dans `xor_network` précédent, le vecteur d'entrée `[1, 0]` correspondait à la sortie cible `[1]`. Et imaginons que votre réseau comporte des ensembles distincts 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 pour tous les neurones du réseau.
2. Il en résulte une erreur pour chaque neurone de sortie qui est égale à la différence entre sa sortie et sa cible.
3. Calculer le gradient de cette erreur sous la forme d'une fonction des poids du neurone et ajuster ses poids dans la direction qui réduit le plus cette erreur.
4. « Propager » ces erreurs de sorties vers l'arrière pour déduire les erreurs de la couche cachée.
5. Calculer les gradients de ces erreurs et ajuster les poids de la couche cachée de la même manière.

D'habitude, on exécute cet algorithme plusieurs fois sur notre jeu d'apprentissage complet jusqu'à ce que le réseau converge :

```
def backpropagate(network, input_vector, targets):

    hidden_outputs, outputs = feed_forward(network, input_vector)

    # le terme en sortie output * (1 - output) vient de la dérivée de sigmoïde
    output_deltas = [output * (1 - output) * (output - target)
                     for output, target in zip(outputs, targets)]
```

```
    # ajuster les poids de la couche de sortie, un neurone à la fois
    for i, output_neuron in enumerate(network[-1]):
        # focus sur le i^ème neurone de couche de sortie
        for j, hidden_output in enumerate(hidden_outputs + [1]):
            # ajuster le j^ème poids basé à la fois
            # sur le delta de ce neurone et sa j^ème entrée
            output_neuron[j] -= output_deltas[i] * hidden_output

    # rétropropagation des erreurs vers la couche cachée
    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)]

    # ajuster les poids pour la couche cachée, un neurone à la fois
    for i, hidden_neuron in enumerate(network[0]):
        for j, input in enumerate(input_vector + [1]):
            hidden_neuron[j] -= hidden_deltas[i] * input
```

Cela revient à peu près à écrire explicitement le carré de l'erreur comme une fonction des poids et à utiliser la fonction `minimize_stochastic` que nous avons créée au chapitre 8.

Dans ce cas, le développement explicite de la fonction de gradient s'avère particulièrement pénible. Si vous connaissez le calcul infinitésimal et la règle de chaînage, les détails mathématiques vous seront relativement faciles à comprendre. Mais réussir à garder de manière rigoureuse la notation (« la dérivée partielle de la fonction d'erreur par rapport au poids du neurone que j'assigne à l'entrée venant du neurone j ») n'est pas très sexy.

Exemple : déjouer un CAPTCHA

Pour être certain que les personnes qui s'inscrivent sur votre site sont bien des humains, le responsable Gestion de produits veut introduire un CAPTCHA dans la procédure d'inscription. En particulier, il aimerait montrer aux utilisateurs une image d'un chiffre et leur demander de le saisir afin de prouver qu'ils sont humains.

Comme il ne vous croit pas quand vous lui expliquez que les ordinateurs sont capables de résoudre ce problème facilement, vous décidez de le convaincre en créant un tel programme.

Nous représenterons chaque chiffre comme une image 5 × 5 :

```
@@@@@ ..@.. @@@@@ @@@@@ @...@ @@@@@ @@@@@ @@@@@ @@@@@ @@@@@
@...@ ..@.. ....@ ....@ @...@ @.... @.... ....@ @...@ @...@
@...@ ..@.. @@@@@ @@@@@ @@@@@ @@@@@ @@@@@ ....@ @@@@@ @@@@@
@...@ ..@.. @.... ....@ ....@ ....@ @...@ ....@ @...@ ....@
@@@@@ ..@.. @@@@@ @@@@@ ....@ @@@@@ @@@@@ ....@ @@@@@ @@@@@
```

Notre réseau neuronal réclame en entrée un vecteur de nombres. Nous allons donc transformer chaque image en un vecteur de longueur 25, dont les éléments sont soit 1 (« ce pixel est dans l'image ») ou 0 (ce pixel n'est pas dans l'image »).

Par exemple, le chiffre zéro peut être représenté comme :

```
zero_digit = [1,1,1,1,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,1,1,1,1]
```

Nous voulons que notre sortie indique à quel chiffre le réseau neuronal croit avoir affaire, il nous faut donc 10 sorties. Par exemple, la sortie correcte pour le chiffre 4 sera :

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Si nous supposons que nos entrées sont correctement triées de 0 à 9, nos objectifs seront :

```
targets = [[1 if i == j else 0 for i in range(10)]
           for j in range(10)]
```

de sorte que, par exemple, `targets[4]` est la sortie correcte pour le chiffre 4.

Nous sommes maintenant prêts à construire notre réseau neuronal :

```
random.seed(0)   # pour obtenir des résultats reproductibles
input_size = 25  # chaque entrée est un vecteur de longueur 25
num_hidden = 5   # nous aurons 5 neurones dans la couche cachée
output_size = 10 # il nous faut 10 sorties pour chaque entrée

# chaque neurone caché a un poids par entrée plus un poids de biais
hidden_layer = [[random.random() for __ in range(input_size + 1)]
                for __ in range(num_hidden)]

# chaque neurone de sortie a un poids par neurone caché plus un poids de biais
output_layer = [[random.random() for __ in range(num_hidden + 1)]
                for __ in range(output_size)]

# le réseau commence avec des poids aléatoires
network = [hidden_layer, output_layer]
```

Et nous pouvons l'entraîner avec l'algorithme de rétropropagation :

```
# 10 000 itérations semblent suffisantes pour converger
for __ in range(10000):
    for input_vector, target_vector in zip(inputs, targets):
        backpropagate(network, input_vector, target_vector)
```

Bien évidemment, l'algorithme fonctionne bien sur le jeu de données d'apprentissage :

```
def predict(input):
    return feed_forward(network, input)[-1]

predict(inputs[7])
# [0.026, 0.0, 0.0, 0.018, 0.001, 0.0, 0.0, 0.967, 0.0, 0.0]
```

Le vecteur obtenu signifie que le neurone de sortie du chiffre 7 produit 0,97 alors que les autres neurones de sortie produisent des nombres très faibles.

Mais nous pouvons aussi l'appliquer à des chiffres dessinés différemment, comme mon 3 stylisé :

```
predict([0,1,1,1,0,  # .@@@.
         0,0,0,1,1,  # ...@@
         0,0,1,1,0,  # ..@@.
         0,0,0,1,1,  # ...@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.92, 0.0, 0.0, 0.0, 0.01, 0.0, 0.12]
```

Le réseau pense encore qu'il s'agit d'un 3, tandis que mon 8 stylisé reçoit des

votes qui le désignent comme un 5, un 8 et un 9 :

```
predict([0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.0, 0.0, 0.55, 0.0, 0.0, 0.93, 1.0]
```

Disposer d'un jeu d'apprentissage plus grand nous aiderait certainement.

Bien que le fonctionnement du réseau ne soit pas exactement transparent, nous pouvons examiner les poids de la couche cachée pour tenter de comprendre ce qui est reconnu. En particulier, nous pouvons représenter les poids de chaque neurone sur une grille 5 × 5 qui correspond aux 5 × 5 entrées. Dans la vie réelle, vous choisirez sans doute de représenter les poids zéro en blanc, les poids de plus en plus positifs en vert de plus en plus foncé et les poids de plus en plus négatifs en rouge de plus en plus foncé (par exemple). Malheureusement, cela est difficile à réaliser dans un livre en noir et blanc.

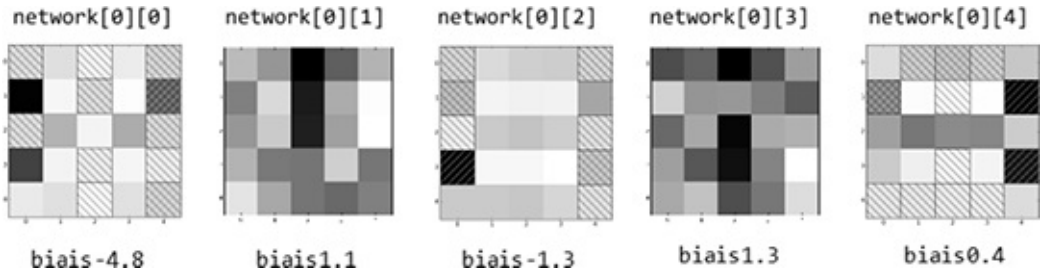
Nous allons donc représenter les poids zéro en blanc, avec des points de plus en plus foncés quand on s'éloigne de zéro. Et nous utiliserons des hachures pour les poids à valeur négative.

Pour créer cette représentation, nous utiliserons `pyplot.imshow`, que nous rencontrons pour la première fois. Cet outil permet de représenter des images pixel par pixel. Normalement, cela ne sert pas à grand-chose en data science, mais dans notre cas, c'est un bon choix :

```
import matplotlib
weights = network[0][0]             # premier neurone de couche cachée
abs_weights = map(abs, weights)     # l'intensité dépend seulement de la valeur absolue

grid = [abs_weights[row:(row+5)]    # transforme les poids en grille 5 x 5
        for row in range(0,25,5)]   # [abs_weights[0:5], ..., abs_weights[20:25]]

ax = plt.gca()                      # pour hachurer il nous faut les axes

ax.imshow(grid,                     # ici la même chose que plt.imshow
          cmap=matplotlib.cm.binary, # utilise une échelle de couleurs noir & blanc
          interpolation='none')     # représente les blocs comme des blocs

def patch(x, y, hatch, color):
    """retourne un objet matplotlib 'patch' avec l'emplacement spécifié,
    le motif de hachage et la couleur"""
    return matplotlib.patches.Rectangle((x - 0.5, y - 0.5), 1, 1,
                                        hatch=hatch, fill=False, color=color)

# hachurer les poids négatifs
```

```
for i in range(5):                # ligne
    for j in range(5):            # colonne
        if weights[5*i + j] < 0:  # ligne i, colonne j = weights[5*i + j]
            # ajoute des hachures noires et blanches, visibles avec peu
            # ou beaucoup de luminosité
            ax.add_patch(patch(j, i, '/', "white"))
            ax.add_patch(patch(j, i, '\\', "black"))

plt.show()
```

Figure 18–4
Poids de la couche cachée



Sur la figure 18-4, nous voyons que le premier neurone caché a des poids élevés positifs dans la colonne de gauche et au centre de la ligne du milieu, et des poids élevés négatifs dans la colonne de droite. (Vous constatez aussi qu'il a un biais négatif important, ce qui veut dire qu'il ne se déclenchera pas fortement à moins de recevoir précisément les entrées positives « attendues ».)

En fait, avec ces entrées, il se comporte comme vous l'espériez :

```
left_column_only = [1, 0, 0, 0, 0] * 5
print feed_forward(network, left_column_only)[0][0]   # 1.0

center_middle_row = [0, 0, 0, 0, 0] * 2 + [0, 1, 1, 1, 0] + [0, 0, 0, 0, 0] * 2
print feed_forward(network, center_middle_row)[0][0]  # 0.95
right_column_only = [0, 0, 0, 0, 1] * 5
print feed_forward(network, right_column_only)[0][0]  # 0.0
```

De même, le neurone caché du centre semble « aimer » les lignes horizontales mais pas les lignes verticales latérales. Et le dernier neurone caché semble « aimer » la ligne du centre mais pas la colonne de droite. (Les deux autres neurones sont difficiles à interpréter.)

Que se passera-t-il si nous faisons lire mon 3 stylisé au réseau ?

```
my_three = [0,1,1,1,0, # .@@@.
            0,0,0,1,1, # ...@@
            0,0,1,1,0, # ..@@.
            0,0,0,1,1, # ...@@
            0,1,1,1,0] # .@@@.
```

```
hidden, output = feed_forward(network, my_three)
```

Les sorties cachées sont :

```
0.121080 # de network[0][0], probablement pénalisé par (1, 4)
0.999979 # de network[0][1], grosses contributions de (0, 2) et (2, 2)
0.999999 # de network[0][2], toujours positif sauf (3, 4)
0.999992 # de network[0][3], encore de grosses contributions de (0, 2) et (2, 2)
0.000000 # de network[0][4], toujours négatif ou nul sauf la ligne centrale
```

ce qui donne au neurone de sortie « three » (trois) les poids de `network[-1][3]` :

```
-11,61 # poids pour hidden[0]
 -2,17 # poids pour hidden[1]
  9,31 # poids pour hidden[2]
 -1,38 # poids pour hidden[3]
-11,47 # poids pour hidden[4]
 -1,92 # poids pour l'entrée biais
```

Le neurone calcule donc :

```
sigmoid(.121 * -11.61 + 1 * -2.17 + 1 * 9.31 - 1.38 * 1 - 0 * 11.47 - 1.92)
```

ce qui donne 0,92, comme nous l'avons déjà vu. En substance, la couche cachée calcule cinq partitions différentes de l'espace à 25 dimensions et associe chaque entrée de ces 25 dimensions à cinq nombres. Ensuite, chaque neurone de sortie utilise seulement le résultat de ces cinq partitions.

Comme nous l'avons vu, mon nombre `my_three` est plutôt du côté « faible » de la partition 0 (c'est-à-dire qu'il active faiblement le neurone 0), loin sur le côté « élevé » des partitions 1, 2 et 3 (c'est-à-dire qu'il active fortement ces neurones cachés) et loin sur le côté faible de la partition 4 (c'est à-dire qu'il n'active pas du tout ce neurone).

Et chacun des 10 neurones de sortie utilise uniquement ces cinq activations pour décider si `my_three` est leur chiffre ou pas.

Pour aller plus loin

- Coursera propose un cours gratuit *Neural Networks for Machine Learning* (Réseaux neuronaux pour l'apprentissage automatique). À l'heure où j'écris ce livre, la dernière session remonte à 2012, mais les éléments du cours sont encore disponibles.
- Michael Nielsen travaille à la rédaction d'un livre en ligne gratuit intitulé *Neural Networks and Deep Learning* (Réseaux neuronaux et apprentissage en profondeur). Lorsque vous lirez ceci, il sera sans doute terminé.
- PyBrain est une bibliothèque très simple de réseaux neuronaux en Python.
- Pylearn2 est une bibliothèque de réseaux neuronaux beaucoup plus avancée (et donc d'utilisation plus difficile).

19

Clustering

Quand de tels groupes nous avions
Qui furieux nous rendaient, mais pas fous.
– Robert Herrick

La plupart des algorithmes de ce livre font de l'apprentissage supervisé : ils commencent avec un ensemble de données labellisées qu'ils utilisent comme base pour effectuer des prédictions sur des données nouvelles non labellisées. Au contraire, le clustering, ou partitionnement de données, est un exemple d'apprentissage non supervisé : les données à étudier sont totalement dépourvues de label (ou ont des labels que nous ignorons).

L’idée

Lorsque vous examinez des données, il est probable que vous y voyez des groupes d’une manière ou d’une autre. Un ensemble de données indiquant où vivent les millionnaires présenterait probablement des groupes comme Beverly Hills ou Manhattan. Un ensemble de données sur le temps de travail hebdomadaire présenterait probablement un groupe autour de 40 (et 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 montrerait 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 leurs analyses par les statisticiens et les analystes politiques.

À la différence des cas que nous avons examinés jusqu’ici, il n’existe en général pas de partitionnement « correct ». Une autre étude pourrait tout aussi bien regrouper une partie de « génération du millénaire au chômage » en « étudiants diplômés » et l’autre en « ceux qui squattent chez leurs parents ». Aucune solution n’est la meilleure. 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 se labelliser tout seuls. Nous devons les qualifier en examinant les données sous-jacentes.

Le 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, auquel cas le but serait de trouver des groupes d'articles similaires, peut-être pour comprendre les sujets traités par nos utilisateurs sur leur blog. Ou alors, imaginons que vous disposez d'une image avec des milliers de couleurs (rouge, vert, bleu) et que vous avez besoin d'une impression écran limitée à 10 couleurs. Le partitionnement peut nous aider à choisir les 10 couleurs qui vont minimiser « l'erreur globale sur la couleur ».

Une des méthodes de partitionnement la plus simple est celle des *k*-moyennes *(k-means)* : le nombre de groupes désiré en sortie *k* est choisi d'avance et on partitionne les entrées en ensembles S_1, …, S_k de telle sorte qu'on minimise la somme totale des carrés des distances de chaque point à 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 de trouver un bon partitionnement.

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 partir l'étape 2.

En utilisant la fonction `vector_mean` vue au chapitre 4, il est assez facile de créer une classe qui réalise cela :

```
class KMeans:
    """réalise un partitionnement par les k-moyennes"""
```

```
def __init__(self, k):
    self.k = k          # nombre de groupes (clusters)
    self.means = None   # moyennes des clusters

def classify(self, input):
     """retourner l'index du cluster le plus proche de la valeur d'entrée"""
     return min(range(self.k),
                key=lambda i: squared_distance(input, self.means[i]))

def train(self, inputs):
    # choisir k points aléatoires comme moyennes initiales
    self.means = random.sample(inputs, self.k)
    assignments = None
    while True:
        # Trouver les nouvelles affectations
        new_assignments = map(self.classify, inputs)

        # si aucune affectation n'a changé, c'est fini
        if assignments == new_assignments:
             return

        # Sinon conserver les nouvelles affectations
        assignments = new_assignments

        # Et calculer de nouvelles moyennes sur la base des nouvelles affectations
        for i in range(self.k):
            # chercher tous les points affectés au cluster i
            i_points = [p for p, a in zip(inputs, assignments) if a == i]

            # s'assurer que i_points n'est pas vide pour ne pas diviser par 0
            if i_points:
                self.means[i] = vector_mean(i_points)
```

Regardons de plus près comment ça marche.

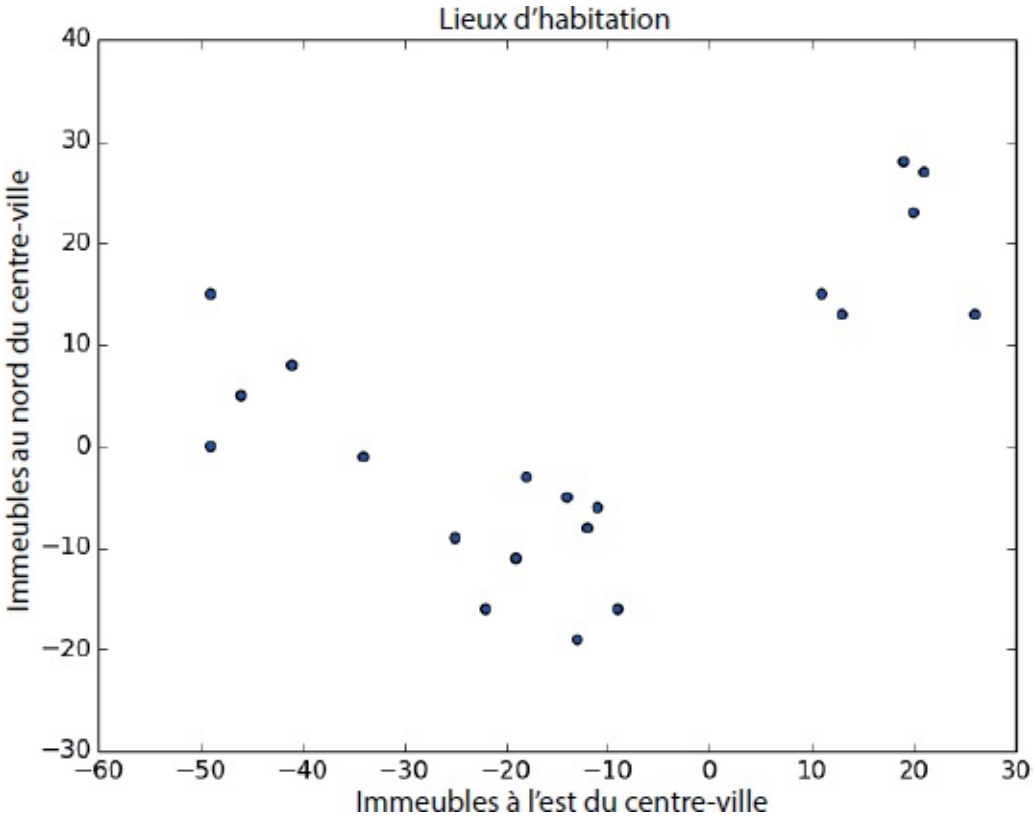
Exemple : rencontres

Pour célébrer la croissance de DataSciencester, la responsable Prix et récompenses veut organiser plusieurs rencontres « en vrai » pour les utilisateurs de votre ville, avec bière, pizza et T-shirts DataSciencester. Vous savez où habitent vos utilisateurs (figure 19-1), elle aimerait donc choisir des lieux de rencontre qui soient pratiques pour tout le monde.

Selon la manière dont vous regardez les emplacements, vous pouvez probablement identifier deux ou trois groupes. (Et ça marche bien parce qu'il n'y ait que deux dimensions. Avec davantage de dimensions, ce serait nettement plus difficile à l'œil nu.) Imaginons qu'elle dispose d'un budget pour organiser trois rencontres. Vous démarrez votre ordinateur et vous essayez ceci :

```
random.seed(0)              # pour avoir le même résultat que moi
clusterer = KMeans(3)
clusterer.train(inputs)
print clusterer.means
```

Figure 19–1
Où habitent les utilisateurs de votre ville ?



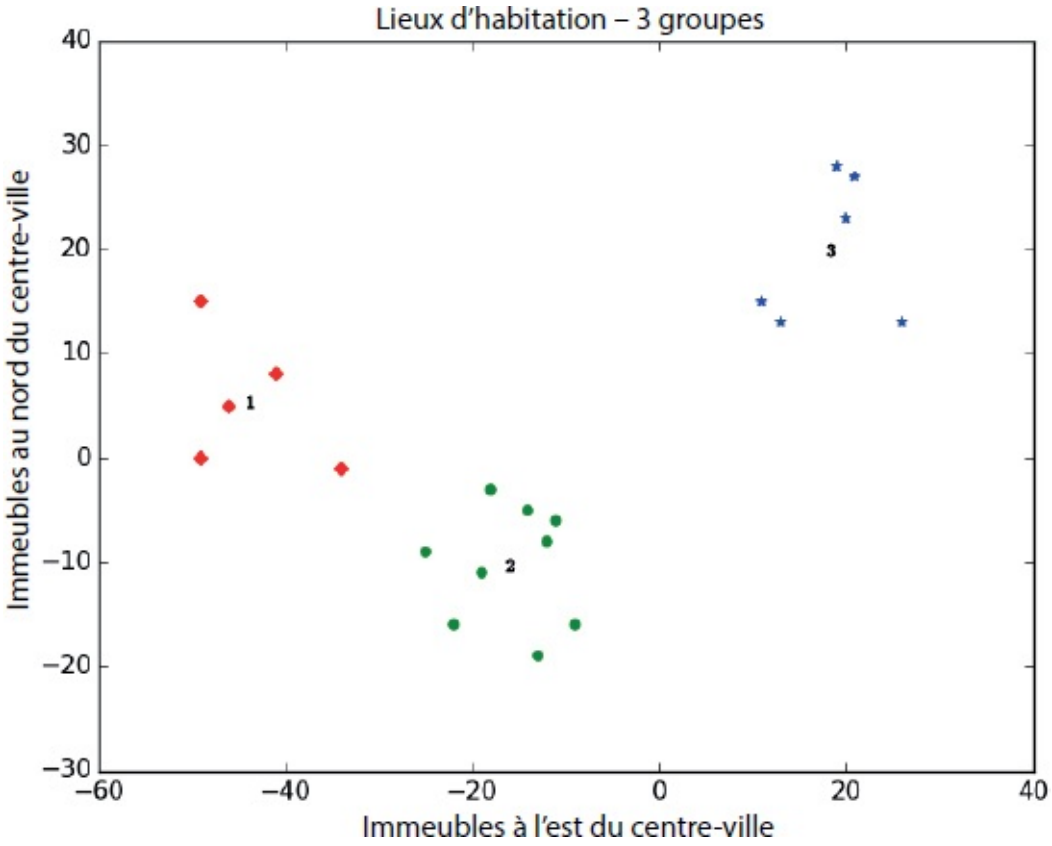
Vous trouvez trois groupes centrés sur [-44, 5], [-16, -10] et [18, 20], et vous cherchez des lieux de rencontre à proximité (figure 19-2).

Vous faites part de votre proposition à la responsable, qui vous informe que son budget ne permet plus désormais que deux rencontres.

« Aucun problème ! », dites-vous :

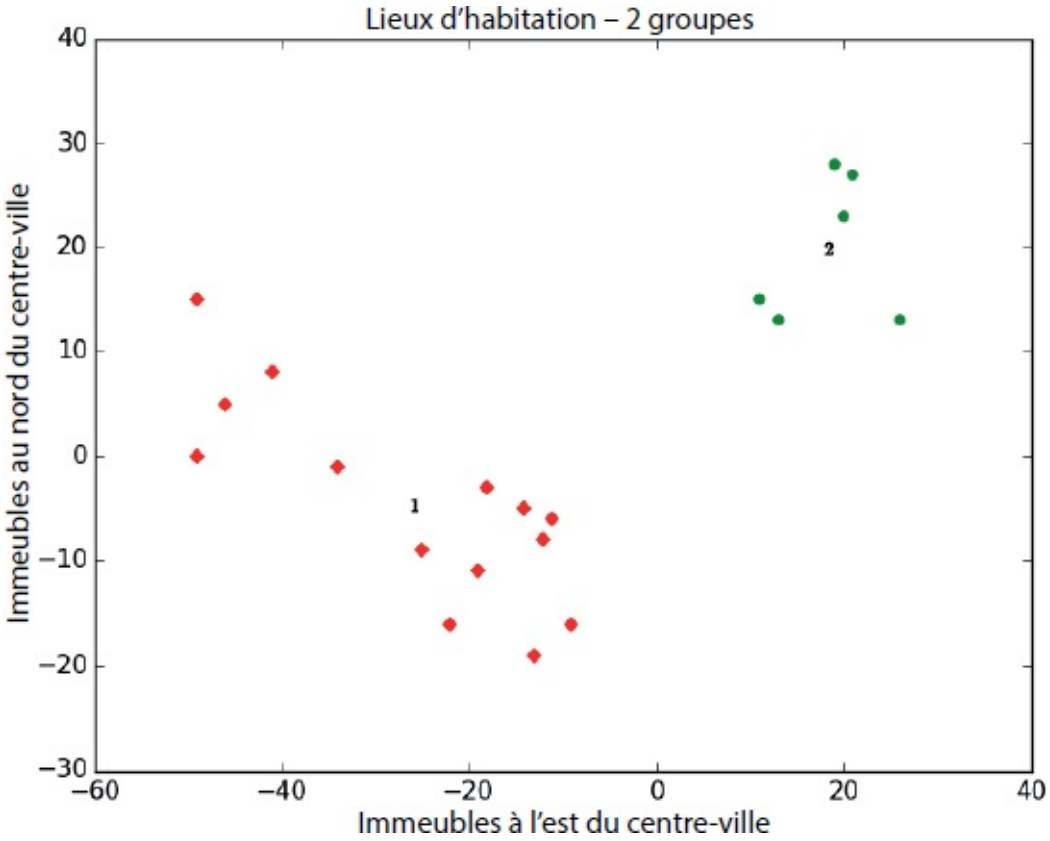
```
random.seed(0)
clusterer = KMeans(2)
clusterer.train(inputs)
print clusterer.means
```

Figure 19–2
Où habitent les utilisateurs : trois groupes



Comme le montre la figure 19-3, une rencontre peut se tenir près de [18, 20] et l'autre devra avoir lieu près de [-26, -5].

Figure 19–3
Où habitent les utilisateurs : deux groupes

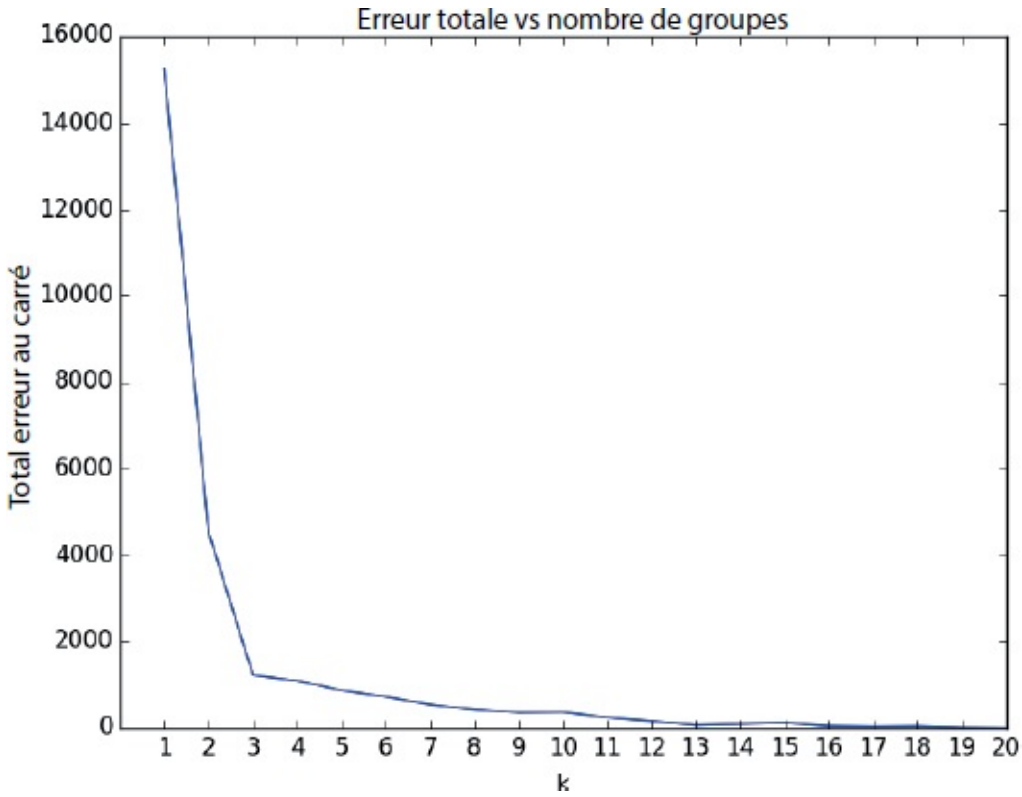

Lieux d'habitation – 2 groupes
Immeubles au nord du centre-ville
Immeubles à l'est du centre-ville
40
30
20
10
0
−10
−20
−30
−60
−40
−20
0
20
40
1
2

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 passe par la représentation graphique de la somme des carrés des erreurs (entre chaque point et la moyenne de son groupe) comme une fonction de *k* et la recherche du point où le graphique « s'infléchit » :

```
def squared_clustering_errors(inputs, k):
    """cherche la somme totale des carrés des erreurs des k-moyennes partitionnant
    les entrées"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = map(clusterer.classify, inputs)

    return sum(squared_distance(input, means[cluster])
        for input, cluster in zip(inputs, assignments))

# tracer la courbe de 1 à len(inputs) partitionnements

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 vs nombre de groupes")
plt.show()
```

Figure 19–4
Choisir une valeur de k



D’après la figure 19-4, cette méthode correspond à notre première impression visuelle : 3 est le « bon » nombre de groupes.

Exemple : partitionnement de couleurs

Le responsable Décorations 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 des Beaux-arts est en congé sabbatique, le responsable Décorations vous demande s'il est possible de partir de son modèle et de le modifier pour ne garder que cinq couleurs.

Les images informatiques 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 qui peut 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. Cela est possible avec matplotlib :

```
path_to_png_file = r"C:\images\image.png" # chemin d'accès à votre image
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)
```

Derrière `img` se cache un tableau NumPy. Mais 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` ; et chaque pixel est une liste `[red, green, blue]` de nombres entre `0` et `1` indiquant sa couleur.

```
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 ainsi :

```
pixels = [pixel for row in img for pixel in row]
```

et nous pouvons l'utiliser pour alimenter notre partitionneur :

```
clusterer = KMeans(5)
clusterer.train(pixels) # peut prendre un certain temps
```

Une fois fini, il ne nous reste plus qu'à construire une nouvelle image de même format :

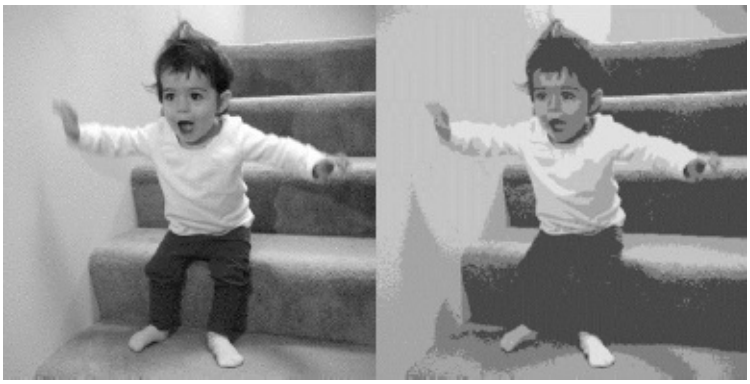
```
def recolor(pixel):
    cluster = clusterer.classify(pixel)       # index du cluster le plus proche
    return clusterer.means[cluster]           # moyenne du cluster le plus proche

new_img = [[recolor(pixel) for pixel in row]  # recolorer cette ligne de pixels
           for row in img]                    # pour chaque ligne de l'image
```

et à l'afficher à l'aide de `plt.imshow()` :

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

Difficile de montrer un résultat en couleurs dans un livre en noir et blanc, mais la figure 19-5 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 19–5
L'image originale et sa version recolorée en cinq couleurs



Le partitionnement hiérarchique de bas en haut

Une autre méthode de partitionnement consiste à « construire » des groupes de bas en haut (du plus détaillé au plus général). Pour cela, il faut suivre ce processus.

1. Faire de chaque entrée son propre groupe de 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 les fusions. Par exemple, si nous voulons trois groupes, il nous suffit de défaire les deux dernières fusions.

Nous utiliserons une représentation très simple des groupes. Nos valeurs seront dans des clusters feuilles que nous représenterons comme des tuples à 1 élément :

```
leaf1 = ([10, 20],) # pour faire un tuple de dimension 1 il faut la virgule de continuité
leaf2 = ([30, -15],) # sinon Python traite les parenthèses comme des parenthèses
```

Nous les utiliserons pour construire les groupes fusionnés, que nous représenterons comme des tuples à 2 éléments (ordre de tri, enfant) :

```
merged = (1, [leaf1, leaf2])
```

Nous reparlerons de l'ordre de fusion par la suite. Créons d'abord quelques fonctions pratiques :

```
def is_leaf(cluster):
    """un cluster est une feuille s'il a comme longueur 1"""
    return len(cluster) == 1


def get_children(cluster):
    """retourne les deux enfants de ce cluster si c'est un cluster issu de fusion ;
    déclenche une exception si c'est un cluster feuille"""
    if is_leaf(cluster):
        raise TypeError("a leaf cluster has no children")
    else:
        return cluster[1]


def get_values(cluster):
    """retourne la valeur dans ce cluster (si c'est un cluster feuille)
ou toutes les valeurs dans le cluster feuille de rang inférieur (s'il n'en est pas un)"""
    if is_leaf(cluster):
        return cluster # est déjà un 1-tuple contenant la valeur
```

```
    else:
        return [value
                for child in get_children(cluster)
                for value in get_values(child)]
```

Pour fusionner les clusters les plus proches, nous devons préciser la notion de distance entre clusters. Nous utiliserons la distance minimale entre les éléments de ces deux clusters, ce qui aura pour effet de fusionner les deux clusters les plus proches au point de se toucher (mais cela produit parfois de grands clusters en forme de chaînes qui ne sont pas très proches les uns des autres). Si nous souhaitons obtenir des clusters de type sphérique plutôt proches, nous utiliserons plutôt la distance maximale, car elle fusionne les deux clusters qui tiennent dans le cercle le plus petit. Les deux choix sont courants, comme l'est aussi le choix de la distance moyenne :

```
def cluster_distance(cluster1, cluster2, distance_agg=min):
    """calculer toutes les paires de distances entre cluster1 and cluster2
    et  appliquer apply _distance_agg_ à la liste résultante"""
    return distance_agg([distance(input1, input2)
                         for input1 in get_values(cluster1)
                         for input2 in get_values(cluster2)])
```

Nous utiliserons la partie de tuple consacré à l'ordre de tri pour suivre l'ordre suivant lequel a été réalisée la fusion. Les nombres les plus petits représentent les fusions tardives. Cela signifie que pour inverser la fusion des groupes, on part de l'ordre de fusion le plus bas jusqu'au plus élevé. Comme les clusters feuilles ne proviennent pas d'une fusion (et que donc nous n'allons pas chercher à les séparer), nous leur affecterons l'infini :

```
def get_merge_order(cluster):
    if is_leaf(cluster):
        return float('inf')
    else:
        return cluster[0] # merge_order est le premier élément du tuple à 2 dimensions
```

Nous sommes maintenant prêts à créer notre algorithme de partitionnement :

```
def bottom_up_cluster(inputs, distance_agg=min):
    # commencer avec toutes les entrées des clusters feuille/1-tuple
    clusters = [(input,) for input in inputs]


    # tant qu'il y a plus d'un cluster...
    while len(clusters) > 1:
        # chercher les deux clusters les plus proches
        c1, c2 = min([(cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]],
                      key=lambda (x, y): cluster_distance(x, y, distance_agg))
```

```
        # les retirer de la liste des clusters
        clusters = [c for c in clusters if c != c1 and c != c2]

        # les fusionner en utilisant merge_order = nbre de clusters restants
        merged_cluster = (len(clusters), [c1, c2])

        # et ajouter la fusion
        clusters.append(merged_cluster)

    # quand il ne reste plus qu'un cluster, le retourner
    return clusters[0]
```

Son utilisation est très simple :

```
base_cluster = bottom_up_cluster(inputs)
```

Il en résulte un cluster dont la représentation, pas très jolie, est :

```
(0, [(1, [(3, [(14, [(18, [([19, 28],),
                           ([21, 27],)]),
                       ([20, 23],)]),
                 ([26, 13],)]),
           (16, [([11, 15],),
                 ([13, 13],)])]),
    (2, [(4, [(5, [(9, [(11, [([-49, 0],),
                              ([-46, 5],)]),
                         ([-41, 8],)]),
                    ([-49, 15],)]),
               ([-34, -1],)]),
          (6, [(7, [(8, [(10, [([-22, -16],),
                               ([-19, -11],)]),
                          ([-25, -9],)]),
                    (13, [(15, [(17, [([-11, -6],),
                                      ([-12, -8],)]),
                                ([-14, -5],)]),
                          ([-18, -3],)])]),
               (12, [([-13, -19],),
                     ([-9, -16],)])])])])
```

Pour chaque cluster fusionné, j'ai aligné les enfants verticalement. Si nous appelons « cluster 0 » le cluster fusionné avec l'ordre de fusion 0, vous pouvez considérer que :

- le cluster 0 est la fusion des clusters 1 et 2 ;
- le cluster 1 est la fusion des clusters 3 et 16 ;
- le cluster 16 est la fusion des feuilles `[11, 15]` et `[13, 13]` ;
- et ainsi de suite.

Comme nous avons 20 entrées, 19 mouvements de fusion sont nécessaires pour arriver à ce cluster unique. La première fusion a créé le cluster 18 par combinaison des feuilles `[19, 28]` et `[21, 27]`. Et la dernière fusion a créé le

cluster 0.

Cependant, généralement nous ne voulons pas nous abîmer les yeux sur des représentations graphiques de ce genre. (Même si ce serait un exercice intéressant de créer un graphique plus convivial de la hiérarchie de partitionnement.) 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, num_clusters):
    # commencer avec une liste contenant seulement le cluster de base
    clusters = [base_cluster]

    # tant que nous n'avons pas encore assez de clusters
    while len(clusters) < num_clusters:
        # choisir le dernier cluster fusionné
        next_cluster = min(clusters, key=get_merge_order)
        # le retirer de la liste
        clusters = [c for c in clusters if c != next_cluster]
        # et 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 clusters...
    return clusters
```

Par exemple, si nous voulons générer trois groupes :

```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

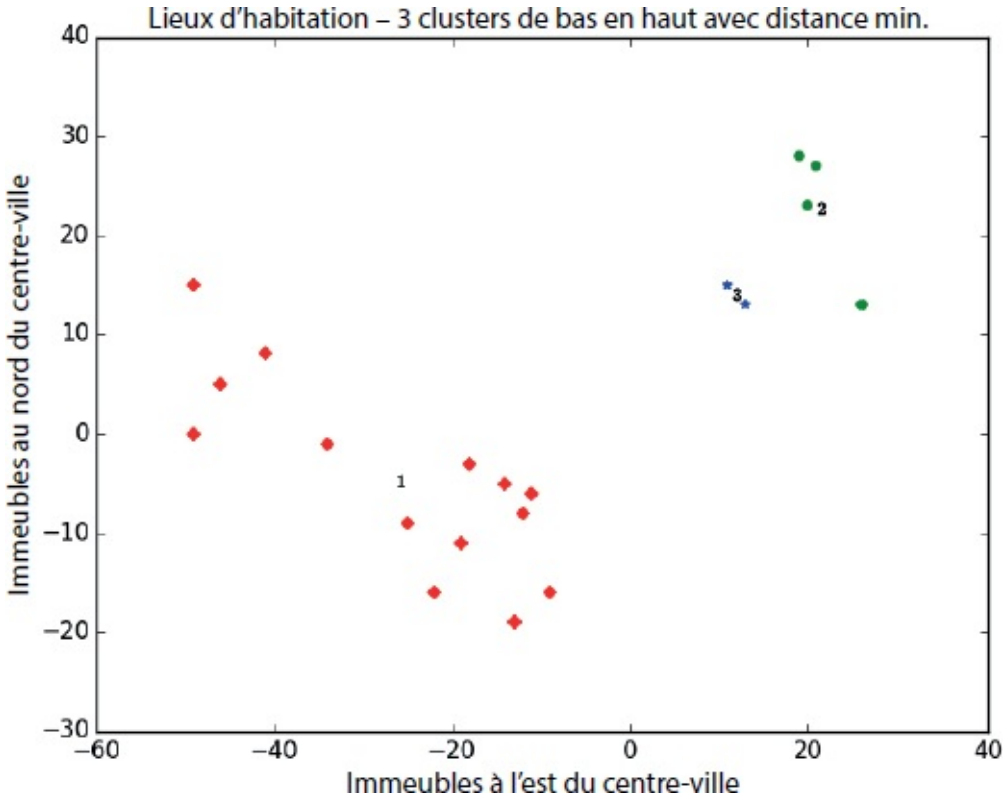
ce qui est facile à représenter graphiquement :

```
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 cluster
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')

plt.title("Lieux d'habitation – 3 clusters de bas en haut avec distance min.")
plt.xlabel("Immeubles à l'est du centre-ville")
plt.ylabel("Immeubles au nord du centre-ville")
plt.show()
```

Les résultats sont très différents de ceux obtenus avec les k-moyennes, comme le montre la figure 19-6.

Figure 19–6

Trois clusters construits de bas en haut avec la méthode de la distance minimale

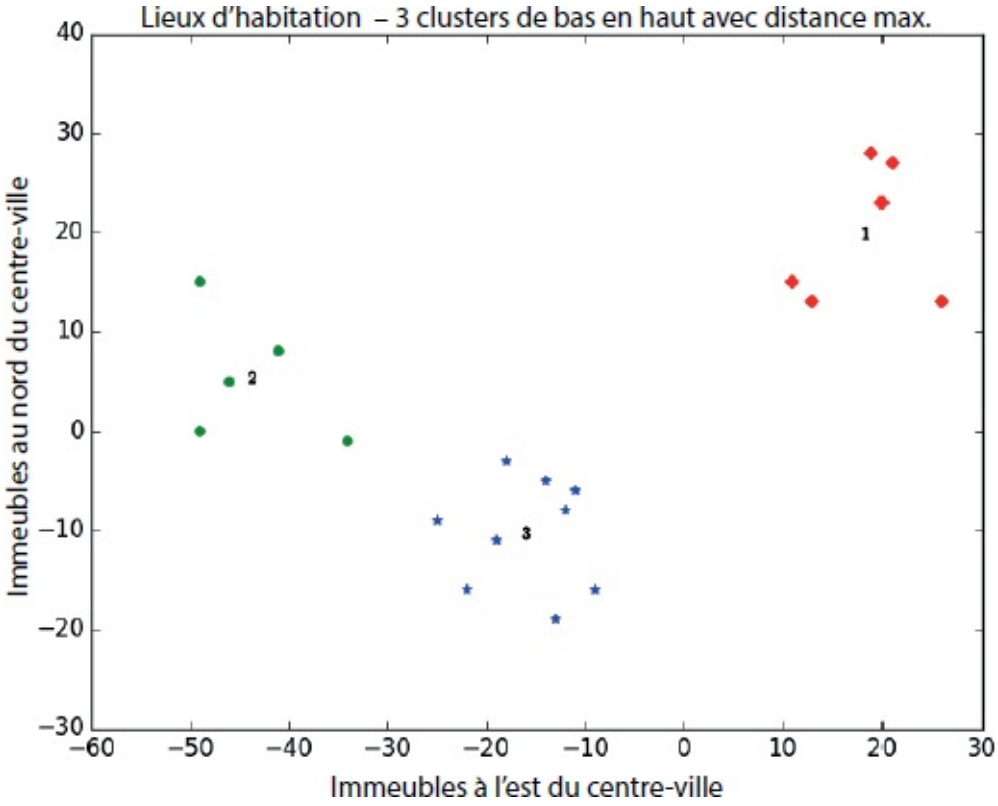


En effet, comme nous l'avons déjà expliqué, l'utilisation du minimum de `cluster_distance` tend à donner des clusters en forme de chaînes. Si nous utilisons le maximum (qui donne des clusters serrés), le résultat est le même que celui des k-moyennes avec $k = 3$ (figure 19-7).

Note

La mise en œuvre du partitionnement de bas en haut est relativement simple, mais elle est aussi remarquablement inefficace. En particulier, elle recalcule la distance entre chaque paire d'entrées à chaque étape. Il serait plus efficace de précalculer les 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 aussi mémoriser les valeurs `cluster_distance` de l'étape précédente.

Figure 19–7
Trois clusters construits de bas en haut avec la méthode de la distance maximale


Lieux d'habitation – 3 clusters de bas en haut avec distance max.
Immeubles au nord du centre-ville
Immeubles à l'est du centre-ville
40
30
20
10
0
−10
−20
−30
−60
−50
−40
−30
−20
−10
0
10
20
30
1
2
3

Pour aller plus loin

- scikit-learn propose un module entier `sklearn.cluster` qui contient plusieurs algorithmes de partitionnement, y compris `KMeans` et l'algorithme hiérarchique de `Ward` (qui utilise un critère différent du nôtre pour fusionner les clusters).
- SciPy contient deux modèles de partitionnement `scipy.cluster.vq` (méthode des kmoyennes) et `scipy.cluster.hierarchy` (qui propose divers algorithmes de partitionnement hiérarchique).

20

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) – *Natural Language Processing* ou NLP – est un ensemble de techniques informatiques appliquées au langage. Il s'agit d'un champ d'étude très vaste, mais ici, nous allons nous intéresser seulement à quelques techniques plus ou moins simples.

Les nuages de mots

Au chapitre 1, nous avons calculé des nombre 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 nuages de mots, en grande partie parce que la position des mots elle-même 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)]
```

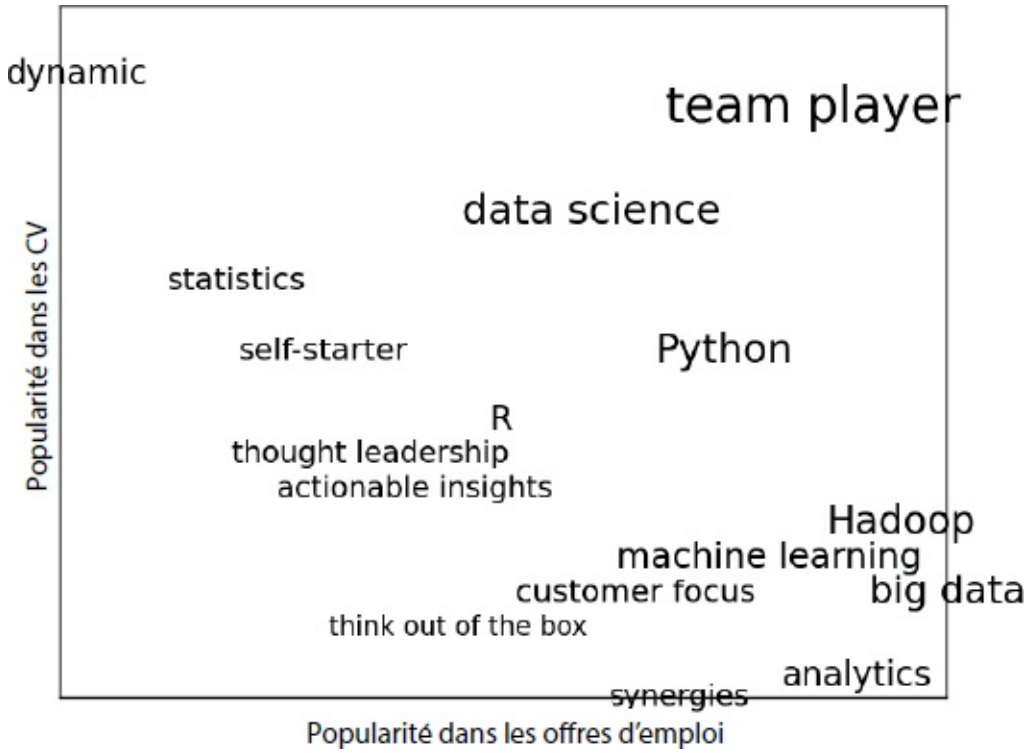
La méthode du nuage de mots consiste à arranger les mots sur une page pour donner une image artistique (figure 20-1).

Figure 20–1
Un nuage de mots à la mode

Tout cela est bien joli, mais ça 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 la popularité dans les CV, ce qui donne une représentation ayant un peu plus de sens (figure 20-2) :

```
def text_size(total):
    """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 d'emploi")
plt.ylabel("Popularité dans les CV")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

Figure 20–2
Un nuage de mots plus significatif (mais moins artistique)


dynamic
team player
data science
statistics
self-starter
Python
R
thought leadership
actionable insights
Hadoop
machine learning
customer focus
big data
think out of the box
analytics
synergies
Popularité dans les CV
Popularité dans les offres d'emploi

Les 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 ait 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?* de Mike Loukides (O'Reilly, non traduit).

Comme au chapitre 9, nous utiliserons les modules `requests` et `BeautifulSoup` 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):
  return text.replace(u"\u2019", "'")
```

Le second problème est qu'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 dire où se terminent les phrases). Nous pouvons utiliser `re.findall()`, pour cela :

```
from bs4 import BeautifulSoup
import requests
url = "http://radar.oreilly.com/2010/06/what-is-data-science.html"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "entry-content") # trouver des div de classe entry-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 nous devrions) nettoyer encore un peu les données. Le document contient encore des parts de texte superflu (par exemple, le premier mot est « Section »). De plus, nous avons parfois découpé les phrases sur la base de points qui se trouvent en milieu de phrase (par exemple « Web 2.0 »). Enfin, il reste une poignée de légendes et de listes un peu partout. 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 (ici : « isn't », « a », « shows », demonstrates » et « teaches »). 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, ce qui veut dire que c'est la fin de la phrase. Nous appelons cela un modèle bigramme, car il est complètement déterminé par les fréquences des bigrammes (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 lorsqu'une de ses entrées est terminée, de sorte que `zip(document, document[1:])` apporte précisément les paires d'éléments consécutifs du document :

```
bigrams = zip(document, document[1:])
transitions = defaultdict(list)
for prev, current in bigrams:
    transitions[prev].append(current)
```

Nous sommes prêts à générer des phrases :

```
def generate_using_bigrams():
    current = "." # ceci veut dire 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 au hasard
        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 « data scientifique ». Par exemple (en anglais) :

« 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. »

Ce qui correspond en français à quelque chose du style :

« 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 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 est déjà suffisant pour nous.) Les transitions vont dépendre des deux mots précédents :

```
trigrams = zip(document, document[1:], document[2:])
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in trigrams:
    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)
```

Notez que maintenant il faut suivre les mots de départ séparément. Nous pouvons générer des phrases à peu près de la même manière :

```
def generate_using_trigrams():
    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. »

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 quelles 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 thème de la data science.

Les 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 langage et comment les combiner. Par exemple, si votre instituteur était vraiment très mauvais, vous pourriez dire 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 :

```
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, `"_S"` est la règle « sentence (phrase) » qui produit une règle `"_NP"` (*noun phrase* ou syntagme nominal) suivie par la règle `"_VP"` (*verb phrase* ou syntagme verbal).

La règle syntagme verbal peut produire soit la règle `"_V"` (verbe), soit la règle verbe suivie de la règle syntagme nominal.

Notez que la règle `"_NP"` se contient elle-même dans un de ses produits. Les grammaires peuvent être récursives, ce qui 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):
     return token[0] != "_"
```

Ensuite, nous devons écrire une fonction pour transformer une liste d'unités lexicales en une phrase. Cherchons la première unité non terminale. Si nous n'en trouvons aucune, c'est que nous avons 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. Si on fait la synthèse, on obtient :

```
def expand(grammar, tokens):
    for i, token in enumerate(tokens):

        # ignorer les terminaisons
        if is_terminal(token): continue

        # si nous sommes ici, nous avons trouvé une unité non terminale
        # donc nous devons choisir son remplaçant au hasard
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

        # maintenant appelons expand sur la nouvelle liste d'éléments lexicaux
        return expand(grammar, tokens)

    # si nous sommes ici nous n'avons plus que des terminaisons et nous avons terminé
    return tokens
```

Et nous pouvons commencer à générer des phrases :

```
def generate_sentence(grammar):
    return expand(grammar, ["_S"])
```

Essayez de changer la grammaire : ajoutez davantage de mots et de règles et insérez vos propres composantes jusqu'à ce que vous soyez prêt à générer autant de pages web que votre société en a besoin.

En réalité, les grammaires sont plus intéressantes lorsqu'on les utilise dans l'autre sens. On utilise plus souvent une grammaire pour *analyser* une phrase, ce qui nous permet d'identifier ses sujets et verbes et donc de 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 », en fin de chapitre, une liste de bibliothèques utiles.)

Aparté : l'échantillonnage de Gibbs

On peut facilement 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())
```

Mais certaines distributions sont plus difficiles à échantillonner. L'échantillonnage de Gibbs est une technique de génération d'échantillons à partir de distributions multidimensionnelles qui est 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 dé et `y` la somme des dés. Imaginons que vous voulez générer de nombreuses paires `(x,y)`. Dans ce cas, il est facile de générer les échantillons directement :

```
def roll_a_die():
    return random.choice([1,2,3,4,5,6])

def direct_sample():
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

Mais imaginons que vous ne connaissez 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 à `x + 1`, `x + 2`, `x + 3`, `x + 4`, `x + 5` ou `x + 6` :

```
def random_y_given_x(x):
    """probabilité égale de donner x + 1, x + 2, ..., x + 6"""
    return x + roll_a_die()
```

Mais cela est 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 ou à 2. De même, si `y` est égal à 11, alors `x` doit être égal à 5 ou 6.

```
def random_x_given_y(y):
    if y <= 7:
```

```
        # si le total est 7 ou < 7, le premier dé a la même probabilité de valoir
        # 1, 2, ..., (total - 1)
        return random.randrange(1, y)
    else:
        # si le total est 7 ou > 7, le premier dé a la même probabilité de valoir
        # (total - 6), (total - 5), ..., 6
        return random.randrange(y - 6, 7)
```

Comment marche 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 sa valeur aléatoire sachant `y`, et le remplacement de `y` sa 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=100):
    x, y = 1, 2 # sans importance
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

Nous pouvons vérifier que le résultat est similaire à celui de l'échantillonnage direct :

```
def compare_distributions(num_samples=1000):
    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.

La 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 » (LDA ou *Latent Dirichlet Analysis*) 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 possède 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 fixe de thèmes, K.
- Il existe une variable aléatoire qui affecte à chaque thème à une distribution probabiliste selon les mots. Vous pouvez considérer cette distribution comme la probabilité de rencontrer un mot w 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 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 où 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 les 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. Tout ce que nous avons à faire, c'est 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 basé sur un jeu de poids arbitraires :

```
def sample_from(weights):
    """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`.

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"]
]
```

Et 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.

Combien de fois chaque thème est-il affecté à chaque document :

```
# une liste de compteurs (Counter), un pour chaque document
document_topic_counts = [Counter() for _ in documents]
```

Combien de fois chaque mot est-il affecté à chaque thème :

```
# une liste de compteurs (Counter), un pour chaque thème
topic_word_counts = [Counter() for _ in range(K)]
```

Le nombre total de mots affectés à chaque thème est :

```
# une liste de nombres, un pour chaque thème
topic_counts = [0 for _ in range(K)]
```

Le nombre total de mots contenus dans chaque document est :

```
# une liste de nombres, un pour chaque document
document_lengths = map(len, documents)
```

Le nombre de mots distincts est :

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

Et le nombre de documents est :

```
D = len(documents)
```

Une fois que nous les avons remplis, nous pouvons par exemple chercher le nombre de mots dans `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 pour 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, d, alpha=0.1):
    """la proportion de mots dans le document _d_
    qui sont affectés au thème _topic_ (plus lissage)"""

    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    """la proportion de mots affectés à  _topic_
    qui valent _word_ (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, word, k):
    """étant donné un document et un mot de ce document,
    retourner les 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, word):
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

Il existe de solides raisons mathématiques qui permettent d'expliquer pourquoi `topic_weight` 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 tout ce dont nous avons besoin. Nous commencerons 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 représentatif de la distribution jointe de la distribution thèmes-mots et de la distribution des thèmes des documents. Pour cela, nous utiliserons une forme d'échantillonnage de Gibbs qui fait appel aux probabilités conditionnelles définies précédemment :

```
for iter in range(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 20-1) :

```
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 20–1 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	machine learning
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 document
    for topic, count in topic_counts.most_common():
        if count > 0:
            print topic_names[topic], count,
    print
```

ce qui 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 `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.

Pour aller plus loin

- *Natural Language Toolkit* (NLTK) est un kit de traitement du langage naturel populaire qui contient une bibliothèque assez complète d'outils de TLN pour Python. Elle est décrite dans un livre dédié que vous pouvez lire en ligne.
- gensim est une bibliothèque Python pour la modélisation thématique qui offre une meilleure solution que notre modèle construit à partir de rien.

21

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 comme des 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 Facebook, alors vous êtes forcément ami avec moi. On dit que les arêtes ne sont pas orientées. Les hyperliens, eux, le sont : mon site contient un hyperlien vers la Maison-Blanche, whitehouse.gov, 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.

La centralité internœud

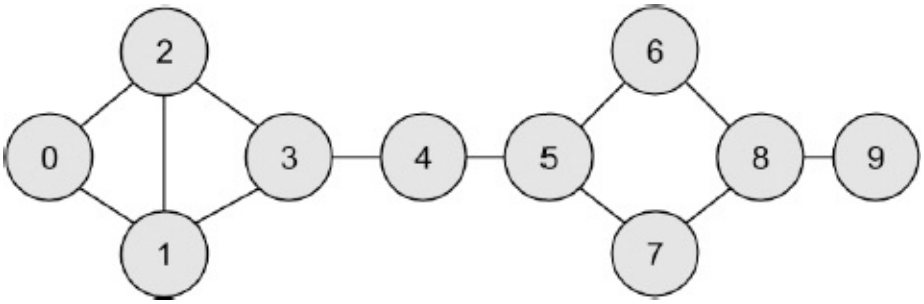
Au chapitre 1, nous avons identifié les connecteurs clés du réseau DataSciencester en comptant le nombre d'amis de chaque utilisateur. Désormais, nous sommes suffisamment bien équipés pour étudier d'autres approches. Souvenez-vous que le réseau (figure 21-1) comprend des utilisateurs :

```
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" }
]
```

et des liens d'amitié :

```
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)]
```

Figure 21–1
Le réseau DataSciencester



Nous avons aussi ajouté des listes d'amis pour chaque dict utilisateur :

```
for user in users:
    user["friends"] = []

for i, j in friendships:
    # cela fonctionne car users[i] est l'utilisateur dont l'id est i
    users[i]["friends"].append(users[j]) # ajouter i comme ami de j
    users[j]["friends"].append(users[i]) # ajouter j comme ami de i
```

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. En particulier, la centralité internœud d'un nœud `i` se calcule en ajoutant pour chaque paire de nœuds `j` et `k` la proportion de plus courts chemins entre le nœud `j` et le nœud `k` qui passe 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 qu'aucun des chemins les plus courts entre Hero (id 0) et Chi (id 3) ne passe par Thor. Pour la première étape, 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) 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 comme une liste d'identifiants d'utilisateurs. Comme chaque chemin part de `from_user`, nous n'inclurons pas son identifiant dans la liste. Cela signifie que la longueur de la liste représentant le chemin est la longueur du chemin lui-même.
3. Nous maintiendrons un dictionnaire `shortest_paths_to` dont les clés sont les identifiants (`id`) des utilisateurs et les valeurs sont des listes de chemins qui se terminent avec l'utilisateur de l'`id` spécifié. 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 que nous souhaitons explorer dans l'ordre dans lequel nous souhaitons les explorer. Nous les stockerons comme des paires (`prev_user, user`) pour savoir comment nous sommes parvenus à chacun d'entre eux. Nous initialiserons la file d'attente avec tous les voisins de `from_user`. (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 implémentées dans `collections.deque` qui est en fait une file d'attente à double extrémité.)

5. Tandis que nous explorons le graphe, chaque fois que nous trouvons de nouveaux voisins pour lesquels nous n'avons pas encore de chemins les plus courts, nous les ajoutons à la fin de la file d'attente pour les explorer plus tard avec l'utilisateur actuel tenant lieu de `prev_user`.
6. Si nous retirons de la file d'attente un utilisateur que nous n'avions pas encore rencontré, cela signifie que nous avons trouvé un ou plusieurs chemins les plus courts menant à lui : lesquels sont 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 avions déjà rencontré auparavant, soit nous avons trouvé un autre chemin le plus court (et dans ce cas nous devrions 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

def shortest_paths_from(from_user):

     # un dictionnaire de "user_id" vers *tous* les chemins les plus courts vers cet
utilisateur
    shortest_paths_to = { from_user["id"] : [[]] }

    # une file de (previous user, next user) à vérifier.
    # commence avec toutes les paires (from_user, friend_of_from_user)

    frontier = deque((from_user, friend)
            for friend in from_user["friends"])

    # continuer jusqu'à vider la file d'attente
    while frontier:

        prev_user, user = frontier.popleft()  # retire l'utilisateur qui
        user_id = user["id"]                  # est le premier de la file d'attente

        # Du fait de 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, [])

        # quel est le 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 inconnus à la file d'attente
        frontier.extend((user, friend)
                        for friend in user["friends"]
                        if friend["id"] not in shortest_paths_to)

    return shortest_paths_to
```

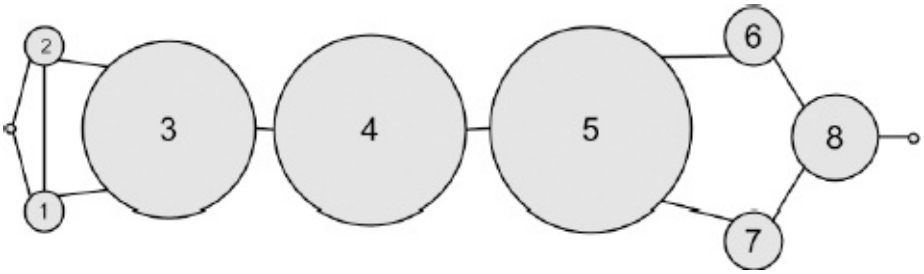
Maintenant, nous pouvons alimenter des dict avec les plus courts chemins de chaque nœud :

```
for user in users:
    user["shortest_paths"] = shortest_paths_from(user)
```

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 :

```
for user in users:
    user["betweenness_centrality"] = 0.0

for source in users:
    source_id = source["id"]
    for target_id, paths in source["shortest_paths"].iteritems():
        if source_id < target_id:   # ne pas compter en double
            num_paths = len(paths)  # combien y a-t-il de chemins les plus courts ?
            contrib = 1 / num_paths # contribution à la centralité
            for path in paths:
                for id in path:
                    if id not in [source_id, target_id]:
                        users[id]["betweenness_centrality"] += contrib
```

Figure 21–2
Le réseau DataSciencester dimensionné par centralité internœud



Comme le montre la figure 21-2, 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 ont tous une centralité élevée (car tous les trois se trouvent sur plusieurs plus courts chemins).

Note

En général, les valeurs de centralité n'ont pas de sens en elles-mêmes. Ce qui nous intéresse, c'est de comparer les valeurs de chaque nœud aux valeurs des autres nœuds.

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.

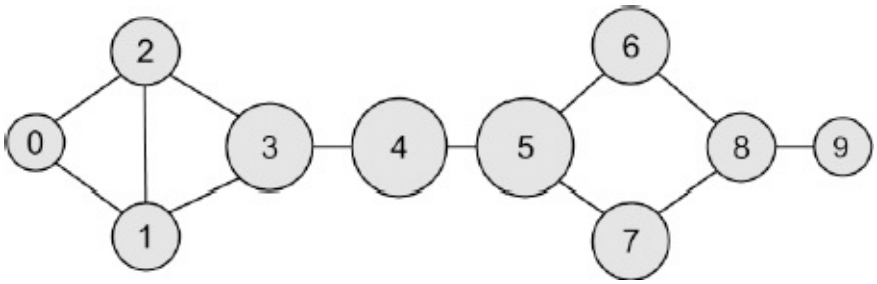
Comme nous avons déjà calculé les plus courts chemins, on peut facilement ajouter leurs longueurs. (En cas de chemins multiples, ils ont tous la même longueur, il suffit alors de prendre le premier.)

```
def farness(user):
    """la somme des longueurs des plus courts chemins vers chaque autre utilisateur"""
    return sum(len(paths[0])
               for paths in user["shortest_paths"].values())
```

après quoi, il ne manque plus grand-chose pour calculer la centralité de proximité (figure 21-3) :

```
for user in users:
    user["closeness_centrality"] = 1 / farness(user)
```

Figure 21–3
Le réseau DataSciencester dimensionné par centralité de proximité



Ici, les variations sont beaucoup plus réduites, car même les nœuds les plus centraux sont 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.

La centralité de vecteur propre

Avant d'examiner la centralité de vecteur propre, nous devons d'abord voir les vecteurs propres *(eigenvectors)*, et avant cela, nous devons parler de multiplication matricielle.

La multiplication matricielle

Si `A` est une matrice $n_1 \times k_1$ et `B` est une matrice $n_2 \times k_2$ et si $k_1 = n_2$, alors leur produit `AB` est la matrice $n_1 \times k_2$ dont l'entrée (`i,j`) est :

$$A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{ik}B_{kj}$$

Ce qui est justement le produit scalaire (`dot`) de la ligne *i* de *A* (vue comme un vecteur) avec la colonne *j* de *B* (également vue comme un vecteur) :

```
def matrix_product_entry(A, B, i, j):
   return dot(get_row(A, i), get_column(B, j))
```

qui nous donne :

```
def matrix_multiply(A, B):
    n1, k1 = shape(A)
    n2, k2 = shape(B)
    if k1 != n2:
        raise ArithmeticError("incompatible shapes!")

    return make_matrix(n1, k2, partial(matrix_product_entry, A, B))
```

Notez que si *A* est une matrice $n \times k$ et *B* une matrice $k \times 1$, alors *AB* est une matrice $n \times 1$. Si nous assimilons un vecteur à une matrice à une colonne, nous pouvons envisager *A* comme une fonction qui fait correspondre des vecteurs de *k* dimensions à des vecteurs de *n* dimensions, cette fonction étant une multiplication de matrices.

Auparavant, nous avions représenté les vecteurs comme des listes, ce qui n'est pas tout à fait la même chose :

```
v = [1, 2, 3]
v_as_matrix = [[1],
               [2],
               [3]]
```

Nous allons avoir besoin de fonctions support pour effectuer des conversions

dans les deux sens entre les deux représentations :

```
def vector_as_matrix(v):
    """retourner le vecteur v (représenté comme une liste) comme une matrice n x 1"""
    return [[v_i] for v_i in v]


def vector_from_matrix(v_as_matrix):
    """retourne les matrices n x 1 comme liste de valeurs"""
    return [row[0] for row in v_as_matrix]
```

après quoi nous serons en mesure de définir l'opération matricielle utilisant `matrix_multiply` :

```
def matrix_operate(A, v):
    v_as_matrix = vector_as_matrix(v)
    product = matrix_multiply(A, v_as_matrix)
    return vector_from_matrix(product)
```

Quand *A* est une matrice carrée, cette opération fait correspondre des vecteurs de *n* dimensions à d'autres vecteurs de vecteurs de `n` dimensions. Il est possible que, pour certaines matrices *A* et certains vecteurs *v*, quand *A* est multiplié *v*, nous obtenions un scalaire multiple de *v*. C'est-àdire que le résultat peut être un vecteur qui pointe dans la même direction que *v*. Dans un tel cas (et quand plus *v* n'est pas un vecteur de zéros), nous appelons *v* un vecteur propre de *A (eigenvector)*. Et nous appelons le scalaire multiple de *v* une valeur propre *(eigenvalue)*.

Une méthode possible pour trouver un vecteur propre de *A* consiste à récupérer un vecteur initial *v*, à lui appliquer `matrix_operate`, puis changer l'échelle du résultat pour qu'il soit de longueur 1 et à répéter jusqu'à la convergence du processus :

```
def find_eigenvector(A, tolerance=0.00001):
    guess = [random.random() for __ in A]

    while True:
        result = matrix_operate(A, guess)
        length = magnitude(result)
        next_guess = scalar_multiply(1/length, result)
        if distance(guess, next_guess) < tolerance:
            return next_guess, length # eigenvector, eigenvalue
        guess = next_guess
```

Par construction, la valeur `guess` retournée est un vecteur tel que, si vous lui appliquez `matrix_operate` et que vous changez l'échelle pour obtenir la longueur 1, alors vous récupérez en retour ce vecteur lui-même (ou 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. Par exemple, la matrice :

```
rotate = [[ 0, 1],
          [-1, 0]]
```

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 ayant un vecteur propre peuvent parfois être coincées indéfiniment. Prenons la matrice :

```
flip = [[0, 1],
        [1, 0]]
```

Elle fait correspondre à tout vecteur `[x,y]` le vecteur `[y,x]`. 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.

La 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 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, j):
    return 1 if (i, j) in friendships or (j, i) in friendships else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

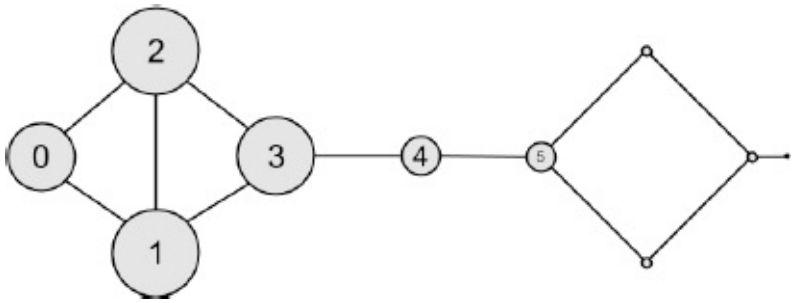
La centralité de vecteur propre pour chaque utilisateur est donc l'entrée correspondante à cet utilisateur dans le vecteur propre retourné par `find_eigenvector` (figure 21-4) :

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

Note

Pour des raisons techniques qui sortent du cadre de ce livre, toute matrice adjacente non nulle a nécessairement un vecteur propre dont toutes les valeurs sont non négatives. Heureusement pour nous, pour cette matrice `adjacency_matrix`, notre fonction `find_eignevector` peut trouver le vecteur propre.

Figure 21–4
Le réseau DataSciencester dimensionné par centralité de vecteur propre



Les utilisateurs avec une centralité de vecteur propre élevée devraient être ceux qui ont 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 ont elles-mêmes 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 peuvent affecter 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_operate(adjacency_matrix, eigenvector_centralities)
```

le résultat est un scalaire multiple de `eigenvector_centralities`.

Si vous observez le fonctionnement de la multiplication de matrices, vous verrez que `matrix_operate` produit un vecteur dont l'élément de rang *i* est :

```
dot(get_row(adjacency_matrix, i), eigenvector_centralities)
```

ce qui 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 connectés directement avec des personnes centrales, plus vous être central. C'est évidemment une définition circulaire : les vecteurs propres sont le moyen de casser cette circularité.

Une autre interprétation revient à réfléchir à la manière dont `find_eigenvector fonctionne`. Pour commencer, on commence par affecter une centralité aléatoire à chaque nœud. Ensuite on 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 réaliser même sur des graphes de très grande taille.

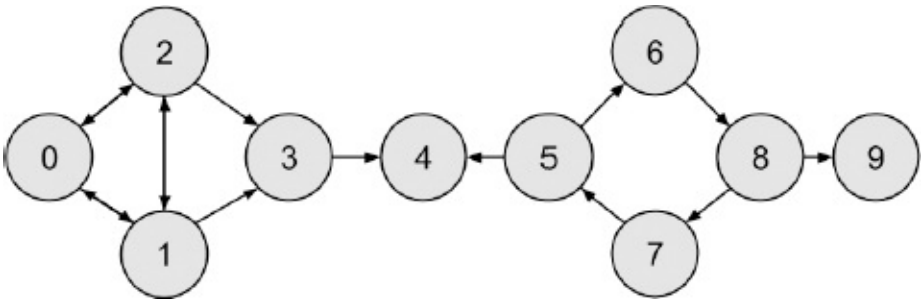
Graphes orientés et PageRank

DataSciencester n'attire pas beaucoup, de sorte que le 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 aimeraient beaucoup savoir quels data scientists sont respectés par leurs pairs.

Dans ce nouveau modèle, nous suivrons les recommandations (`source`, `cible`) qui ne représentent plus une relation de réciprocité, mais plutôt le fait que la `source` recommande la `cible` comme un scientifique de valeur (figure 21-5). 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)]

for user in users:
    user["endorses"] = []    # ajouter une liste pour suivre qui recommande qui
    user["endorsed_by"] = [] # et une autre pour qui est recommandé par qui

for source_id, target_id in endorsements:
    users[source_id]["endorses"].append(users[target_id])
    users[target_id]["endorsed_by"].append(users[source_id])
```

Figure 21–5
Le réseau DataSciencester de recommandations



Ensuite, nous pourrons facilement trouver les data scientists les plus recommandés (`most_endorsed`) et vendre cette information aux chasseurs de têtes :

```
endorsements_by_id = [(user["id"], len(user["endorsed_by"]))
                      for user in users]

sorted(endorsements_by_id,
       key=lambda (user_id, num_endorsements): num_endorsements,
       reverse=True)
```

Cependant, le « nombre de recommandations » est un indicateur assez facile à truquer. Il suffit 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 qui vous a recommandé. Les recommandations de personnes qui ont elles-mêmes beaucoup de recommandations devraient compter davantage, d'une manière ou d'une autre, que les recommandations de personnes peu recommandées. C'est le cœur de l'algorithme de PageRank utilisé par Google pour classer les sites web à partir des autres sites qui pointent vers eux, des sites qui pointent vers ces derniers, et ainsi de suite.

(Si tout ceci vous fait penser à la centralité de vecteur propre, alors vous êtes sur la bonne voie.)

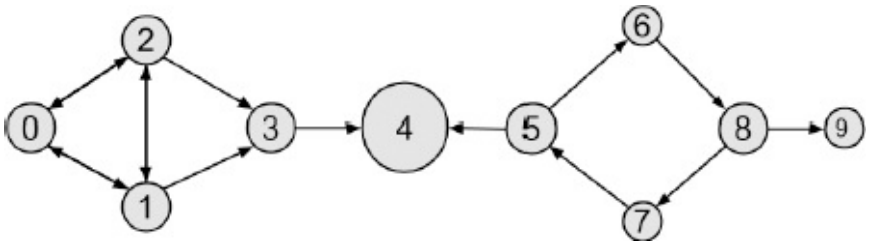
Une version simplifiée ressemble à ceci.

1. Il y a un total de 1,0 (ou 100 %) de PageRank dans le réseau.
2. Initialement, ce PageRank est distribué équitablement entre tous les nœuds.
3. À chaque étape, une large fraction de PageRank de chaque nœud est distribuée uniformément entre les liens sortants.
4. À chaque étape, le PageRank restant de chaque nœud est distribuée uniformément entre tous les nœuds.

```
def page_rank(users, damping = 0.85, num_iters = 100):

    # initialement, distribuer le PageRank uniformément
    num_users = len(users)
    pr = { user["id"] : 1 / num_users for user in users }

    # ceci est la petite fraction de PageRank
    # que reçoit chaque nœud à chaque itération
    base_pr = (1 - damping) / num_users

    for __ in range(num_iters):
        next_pr = { user["id"] : base_pr for user in users }
        for user in users:

            # distribuer le PageRank aux liens sortants
            links_pr = pr[user["id"]] * damping
            for endorsee in user["endorses"]:
                next_pr[endorsee["id"]] += links_pr / len(user["endorses"])

        pr = next_pr

    return pr
```

Le PageRank (figure 21-6) identifie l'utilisateur 4 (Thor) comme l'expert de rang le plus élevé.

Figure 21–6
Le réseau DataSciencester dimensionné par PageRank



Bien qu'il ait moins de recommandations (2) que les utilisateurs 0, 1 et 2, ses recommandations apportent avec elles le rang de leurs recommandations. 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 centralité en plus de celles que nous avons utilisées (bien que nous ayons abordé ici les plus populaires d'entre elles).

- NetworkX est une bibliothèque Python pour l'analyse de réseau. Elle contient des fonctions de calcul de centralité et de représentation de graphes.
- Gephi est un outil de représentation de réseaux basé sur une interface graphique utilisateur. On l'adore ou on le déteste.

22

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 (écrivain anglais du XVIII^e siècle)

Produire des recommandations est un autre problème classique qu'on peut rencontrer 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 :

```
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"]
]
```

Et nous réfléchirons à la possibilité de recommander de nouveaux centres d'intérêt à un utilisateur à partir de ses centres d'intérêt actuels.

La méthode manuelle

Avant Internet, pour chercher des recommandations de lecture, vous alliez à la bibliothèque, où le bibliothécaire 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 être facile pour vous de passer une après-midi à fournir des recommandations personnalisées à chacun d'entre eux. Mais cette méthode ne devient plus gérable quand le nombre d'utilisateurs augmente et elle est limitée par vos connaissances personnelles et votre imagination. (Ne me faites pas dire que vous avez des connaissances et une imagination limitées.) Réfléchissons plutôt au moyen d'exploiter des données.

Recommander ce qui est populaire

Une méthode simple consiste à recommander ce qui a du succès :

```
popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests).most_common()
```

ce qui ressemble à ceci :

```
[('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 :

```
def most_popular_new_interests(user_interests, max_results=5):
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests
                   if interest not in user_interests]
    return suggestions[:max_results]
```

Ainsi, si vous êtes l'utilisateur 1, dont les centres d'intérêt sont :

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

nous vous recommanderons :

```
most_popular_new_interests(users_interests[1], 5)


# [('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

Si vous êtes l'utilisateur 5, qui a déjà choisi la plupart de ces sujets, nous vous recommanderons :

```
[('Java', 3),
 ('HBase', 3),
 ('Big Data', 3),
 ('neural networks', 2),
 ('Hadoop', 2)]
```

Bien sûr, annoncer « beaucoup de personnes s'intéressent à Python, peut-être

que vous devriez essayer vous aussi » n’est sans doute pas le plus percutant des argumentaires de vente. Mais si quelqu’un est nouveau sur notre site et que nous ignorons tout de lui, c’est sans doute ce qu’on peut faire de mieux. Voyons maintenant comment améliorer nos recommandations en nous appuyant sur les centres d’intérêt connus de chaque utilisateur.

Le filtrage collaboratif sur la base des utilisateurs

Pour tenir compte des centres d'intérêt d'un utilisateur, une méthode consiste à rechercher d'autres utilisateurs qui présentent des similitudes avec lui et à lui suggérer les sujets qui intéressent ces utilisateurs.

Il nous faut donc un moyen de mesurer le degré de similitude entre deux utilisateurs. Nous allons utiliser un indicateur appelé « similarité cosinus ». Étant donnés deux vecteurs `v` et `w`, l'indicateur se définit comme ceci :

```
def cosine_similarity(v, w):
    return dot(v, w) / math.sqrt(dot(v, v) * dot(w, w))
```

L'indicateur mesure « l'angle » entre `v` et `w`. Si `v` et `w` pointent vers la même direction, le numérateur et le dénominateur sont égaux et la similarité cosinus est égale à `1`. Si `v` et `w` pointent dans des directions opposées, alors la similarité cosinus est égale à -1. Si `v` vaut 0, mais pas `w` (ou l'inverse), alors `dot(v,w)` vaut 0 et la similarité cosinus est égale à 0.

Nous appliquerons cette méthode à des vecteurs constitués de `0` et de `1`, chaque vecteur `v` représentant les centres d'intérêt d'un utilisateur. `v[i]` est égal à `1` si l'utilisateur spécifie l'intérêt de rang `i`, et à `0` sinon. De même, « des utilisateurs similaires » sont des utilisateurs dont les vecteurs d'intérêt pointent presque dans la même direction. Les utilisateurs ayant les mêmes centres d'intérêt ont la similarité `1` ; les utilisateurs sans aucun centre d'intérêt commun ont la similarité `0`. Dans les autres cas, la similarité se situera entre les valeurs proches de `1` signifiant « très similaire » et les valeurs proches de `0` signifiant « pas très similaire ».

Rassembler les centres d'intérêt déclarés et leur affecter (implicitement) des indices constitue un bon point de départ. Pour cela, nous pouvons utiliser un *set comprehension* pour trouver les centres d'intérêt uniques, les rassembler dans une liste et les trier. Le premier centre d'intérêt de la liste sera le centre d'intérêt 0, et ainsi de suite :

```
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

Nous obtenons une liste qui commence ainsi :

```
['Big Data',
 'C++',
```

```
 'Cassandra',
 'HBase',
 'Hadoop',
 'Haskell',
 #...
]
```

Puis, nous produirons un « vecteur intérêt » composé de `0` et de `1` pour chaque utilisateur. Il suffira alors de boucler sur la liste `unique_interests` en mettant `1` si l'utilisateur a déclaré ce centre d'intérêt, et sinon `0` :

```
def make_user_interest_vector(user_interests):
    """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 matrice des centres d'intérêt simplement en appliquant cette fonction à la liste des listes de centres d'intérêt :

```
user_interest_matrix = map(make_user_interest_vector, users_interests)
```

Maintenant, `user_interest_matrix[i][j]` vaut `1` si l'utilisateur `i` a déclaré le sujet `j`, `0` sinon.

Comme notre jeu de données est petit, ce n'est pas un problème de calculer les similarités par paires d'utilisateurs entre tous les utilisateurs :

```
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_matrix]
                     for interest_vector_i in user_interest_matrix]
```

Après ce calcul, `user_similarities[i][j]` nous donne la similarité entre les utilisateurs `i` et `j`.

Par exemple, `user_similarities[0][9]` est à 0,57, car ces deux utilisateurs partagent un intérêt pour `Hadoop`, `Java` et `Big Data`. D'un autre côté, `user_similarities[0][8]` est seulement à 0,19, car les utilisateurs 0 et 8 ont un seul centre d'intérêt commun : `Big Data`.

En particulier, `user_similarities[i]` est le vecteur des similarités de l'utilisateur `i` avec chaque autre utilisateur. Nous pouvons utiliser cela pour écrire une fonction de recherche des utilisateurs qui ressemblent le plus à un utilisateur donné. Prenez garde, à cette étape, de ne pas inclure l'utilisateur lui-même ni aucun utilisateur ayant une similarité nulle. Nous trions à présent les résultats du plus similaire au moins similaire :

```
def most_similar_users_to(user_id):
    pairs = [(other_user_id, similarity)                # Trouve d'autres
             for other_user_id, similarity in           # utilisateurs avec
                 enumerate(user_similarities[user_id])  # une similarité non nulle
             if user_id != other_user_id and similarity > 0]

    return sorted(pairs,                                # Trie ceux qui
                  key=lambda (_, similarity): similarity, # sont les plus similaires
                  reverse=True)                         # en premier
```

Par exemple, si nous appelons `most_similar_users_to(0)`, nous obtenons :

```
[(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 centre d'intérêt, nous pouvons ajouter les similarités des autres utilisateurs intéressés par ce centre d'intérêt :

```
def user_based_suggestions(user_id, include_current_interests=False):
    # faire la somme des similarités
    suggestions = 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 (_, weight): weight,
                         reverse=True)

    # et exclure éventuellement les centres d'intérêt actuels de l'utilisateur considéré
    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)`, le premier des centres d'intérêt suggérés est :

```
[('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 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 marche pas très bien si le nombre de centres d'intérêt possibles est très grand. Souvenez-vous de la malédiction de la dimension évoquée au chapitre 12 : dans les espaces vectoriels de grandes dimensions, la plupart des vecteurs sont très éloignés (et pointent tous 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 chez qui 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 avec un historique d'achats même vaguement ressemblant 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.

Le 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-intérêt pour que les lignes soient les centres d'intérêt et les colonnes les utilisateurs :

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_matrix]
                        for j, _ in enumerate(unique_interests)]
```

À quoi ressemble le résultat ? La ligne `j` de la matrice `interest_user_matrix` est la colonne `j` de `user_interest_matrix`. C'est-à-dire qu'elle contient `1` pour chaque utilisateur déclarant ce centre d'intérêt, et `0` pour chaque utilisateur qui l'ignore complètement.

Par exemple, `unique_interests[0]` est `Big Data`, donc `interest_user_matrix[0]` est :

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

parce que les utilisateurs `0`, `8` et `9` ont déclaré `Big Data` comme un de leurs centres d'intérêt.

Nous pouvons utiliser encore une fois la similarité de cosinus. Si précisément les mêmes utilisateurs s'intéressent à deux sujets, alors leur similarité sera à `1`. Mais si aucun des utilisateurs ne s'intéresse aux deux sujets, alors leur similarité sera nulle :

```
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 (interest 0)` avec :

```
def most_similar_interests_to(interest_id):
    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 (_, similarity): similarity,
                  reverse=True)
```

ce qui 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, include_current_interests=False):
    # faire la somme des intérêts similaires
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_matrix[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

    # trier par poids
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, similarity): similarity,
                         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`, on obtient ces recommandations, qui paraissent assez 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)]
```

Pour aller plus loin

- Crab est un framework permettant de construire des systèmes de recommandation avec Python.
- Graphlab propose également une boîte à outils pour générer des recommandations.
- Et le Netflix Prize était un concours assez célèbre destiné à construire un meilleur système de recommandations de films aux abonnés de Netflix.

23

Base de données et SQL

La mémoire est le meilleur ami de l'homme et son pire ennemi.
– Gilbert Parker (romancier et politicien canadien du début du XX^e s)

Les données dont vous avez besoin sont souvent présentes dans des bases de données. Ces dernières sont des systèmes conçus pour stocker et rechercher les données efficacement. La majorité d'entre elles sont de type relationnel comme Oracle, MySQL et SQL Server, qui stockent les données dans des tables et qu'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 implémentation Python de quelque chose qui n'est pas tout à fait une base de données. Nous parlerons aussi des fondamentaux de SQL en voyant comment ils marchent sur notre « presque base de données », ce qui pourra vous aider à comprendre comment ça marche en supposant qu'on ne parte de rien. J'espère sincèrement que la résolution des problèmes dans notre presque base de données NotQuiteABase vous montrera comment appréhender les mêmes problèmes avec SQL.

CREATE TABLE et INSERT

Une base de données relationnelle est une collection de tables (et de relations entre elles). 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 son nombre d'amis (soit `user_id, name` et `num_friends`) :

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

En SQL, nous allons créer cette table ainsi :

```
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 être `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. NotQuiteABase ne prend pas en compte les types, mais nous allons faire comme si c'était le cas.

Note

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 de nommage.

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 pouvez créer une table simplement en spécifiant les noms de ses colonnes. Pour insérer une ligne dans la table, vous utiliserez ensuite la méthode `insert()`. Cette instruction prend en entrée une liste de valeurs brutes 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 de colonnes. Une vraie base de données n'utiliserait jamais une représentation qui gaspille autant de place, mais ce choix rend NotQuiteABase beaucoup plus facile à utiliser :

```
class Table:
    def __init__(self, columns):
        self.columns = columns
        self.rows = []

    def __repr__(self):
        """Représentation lisible de la table : colonnes puis lignes"""
        return str(self.columns) + "\n" + "\n".join(map(str, self.rows))

    def insert(self, row_values):
        if len(row_values) != len(self.columns):
            raise TypeError("wrong number of elements")
        row_dict = dict(zip(self.columns, row_values))
        self.rows.append(row_dict)
```

Par exemple, nous pouvons définir :

```
users = Table(["user_id", "name", "num_friends"])
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])
```

Et si vous utilisez l'instruction `print users`, vous aurez :

```
['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}
...
```

UPDATE

Il peut vous arriver d'avoir besoin de mettre à jour les données qui sont déjà dans une table. Par exemple, si Dunn a un nouvel ami, vous serez tenté de faire ceci :

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

Les points importants à prendre en compte pour cette opération sont :

- quelle est la table à mettre à jour ?
- quelles sont les lignes à mettre à jour ?
- quels sont les champs à mettre à jour ?
- quelles sont les nouvelles valeurs ?

Nous allons ajouter une méthode `update` similaire de mise à jour 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 sera un prédicat qui retourne `True` (vrai) pour les lignes à mettre à jour, `False` (faux) sinon :

```
def update(self, updates, predicate):
    for row in self.rows:
        if predicate(row):
           for column, new_value in updates.iteritems():
                row[column] = new_value
```

Ensuite, nous pourrons nous contenter d'écrire ceci :

```
    users.update({'num_friends' : 3},      # maj  num_friends = 3
          lambda row: row['user_id'] == 1) # pour les lignes où user_id == 1
```

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` et supprime 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=lambda row: True):
    """supprimer les lignes qui vérifient le prédicat
    ou toutes les lignes si aucun prédicat n'est fourni"""
    self.rows = [row for row in self.rows if not(predicate(row))]
```

Si vous fournissez une fonction prédicat (par exemple, une clause `WHERE`), seules les lignes qui vérifient la condition seront supprimées. Mais si vous ne fournissez pas de prédicat, le prédicat par défaut retournera toujours `True` et toutes les lignes seront supprimées.

Voici un exemple :

```
users.delete(lambda row: row["user_id"] == 1) # supprime les lignes pour lesquelles
                                              # user_id == 1
users.delete()                                # supprime toutes les lignes
```

SELECT

Il est très rare qu'on examine directement une table SQL. À la place, on les interroge en utilisant l'instruction `SELECT` :

```
SELECT * FROM users;                          -- tout le contenu
SELECT * FROM users LIMIT 2;                  -- seulement les deux premières lignes
SELECT user_id FROM users;                    -- seulement les colonnes spécifiées
SELECT user_id FROM users WHERE name = 'Dunn'; -- seulement les lignes spécifiées
```

Les instructions `SELECT` permettent aussi de déduire de nouveaux champs à partir de ceux existants :

```
SELECT LENGTH(name) AS name_length FROM users;
```

Nous allons donner 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` sera un dictionnaire dont les clés sont les noms des nouvelles colonnes et dont les valeurs sont des fonctions spécifiant comment calculer les valeurs des nouvelles colonnes.

Si vous ne fournissez aucun des deux, vous récupérerez une simple copie de la table.

```
def select(self, keep_columns=None, additional_columns=None):

    if keep_columns is None:        # si aucune colonne spécifiée,
        keep_columns = self.columns # retourne toutes les colonnes

    if additional_columns is None:
        additional_columns = {}

    # nouvelle Table pour les résultats
    result_table = Table(keep_columns + additional_columns.keys())

    for row in self.rows:
        new_row = [row[column] for column in keep_columns]
        for column_name, calculation in additional_columns.iteritems():
            new_row.append(calculation(row))
        result_table.insert(new_row)

    return result_table
```

Notre `select()` retourne une nouvelle `Table`, tandis que le `SELECT` SQL classique

produit un résultat transitoire (à moins d'insérer explicitement les résultats dans une table).

Il nous faut aussi des méthodes `where()` et `limit()`. Les deux sont très simples :

```
def where(self, predicate=lambda row: True):
    """retourne seulement les lignes qui vérifient le prédicat fourni"""
    where_table = Table(self.columns)
    where_table.rows = filter(predicate, self.rows)
    return where_table


def limit(self, num_rows):
    """retourne seulement les num_rows premières lignes"""
    limit_table = Table(self.columns)
    limit_table.rows = self.rows[:num_rows]
    return limit_table
```

Ainsi, nous pouvons facilement construire des équivalents NotQuiteABase aux instructions SQL déjà vues :

```
# SELECT * FROM users;
users.select()

# SELECT * FROM users LIMIT 2;
users.limit(2)

# SELECT user_id FROM users;
users.select(keep_columns=["user_id"])

# SELECT user_id FROM users WHERE name = 'Dunn';
users.where(lambda row: row["name"] == "Dunn") \
     .select(keep_columns=["user_id"])

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row): return len(row["name"])

users.select(keep_columns=[],
             additional_columns = { "name_length" : name_length })
```

Notez que, à la différence du reste de ce livre, je fais usage du caractère `\` pour poursuivre des instructions sur plus d'une ligne. Je trouve que cela facilite la lecture des requêtes chaînées de NotQuiteABase mieux que toute autre convention d'écriture.

GROUP BY

Une autre opération souvent utilisée en SQL est `GROUP BY`, qui permet de regrouper des lignes de valeurs identiques dans des colonnes spécifiées et de produire des agrégats sur d'autres colonnes avec des fonctions comme `MIN`, `MAX`, `COUNT` (compteur) et `SUM` (somme). (Ceci devrait vous rappeler la fonction `group_by` de la section « La manipulation des données » en page 133.)

Par exemple, si vous cherchez le nombre d'utilisateurs et le plus petit identifiant `user_id` pour chaque longueur de nom possible, voici comment faire :

```
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 être soit dans la clause `GROUP BY` (comme `name_length`) soit agrégé d'une manière ou d'une autre (comme le sont `min_user_id` et `num_users`).

SQL accepte également une clause `HAVING` qui se comporte comme une clause `WHERE` sauf 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;
```

(Les fonctions de traitement des chaînes de caractères sont différentes selon les implémentations de SQL ; certaines bases de données utilisent parfois `SUBSTRING` ou autre chose.)

Vous pouvez aussi calculer des agrégats globaux. Dans ce cas, vous pourrez laisser tomber le `GROUP BY` :

```
SELECT SUM(user_id) as user_id_sum
FROM users
WHERE user_id > 1;
```

Pour ajouter cette fonctionnalité à nos tables NotQuiteABase, nous allons ajouter une méthode `group_by()`.

Cette méthode prendra 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. Crée un `defaultdict` qui fait correspondre les tuples (des valeurs de `group-by`) aux lignes (correspondant aux valeurs de `group-by`). N'oubliez pas que vous ne pouvez pas utiliser des listes comme clés : vous devez utiliser des tuples.
2. Boucle sur les lignes de la table en remplissant le `defaultdict`.
3. Crée une nouvelle table avec les colonnes correctes en sortie.
4. Boucle sur le defaultdict et remplit la table de sortie en appliquant le filtre `having` le cas échéant.

(Une vraie base de données aurait certainement recours à une méthode plus efficace.)

```
def group_by(self, group_by_columns, aggregates, having=None):

    grouped_rows = defaultdict(list)

    # remplit les groupes
    for row in self.rows:
        key = tuple(row[column] for column in group_by_columns)
        grouped_rows[key].append(row)

    # la table résultante consiste en colonnes de group_by et agrégats
    result_table = Table(group_by_columns + aggregates.keys())

    for key, rows in grouped_rows.iteritems():
        if having is None or having(rows):
           new_row = list(key)
           for aggregate_name, aggregate_fn in aggregates.iteritems():
               new_row.append(aggregate_fn(rows))
           result_table.insert(new_row)

    return result_table
```

Une fois encore, voyons comment nous pourrions faire la même chose que les instructions SQL précédentes. Les indicateurs de longueur de nom sont :

```
def min_user_id(rows): return min(row["user_id"] for row in 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" : len })
```

Les indicateurs de première lettre sont :

```
def first_letter_of_name(row):
   return row["name"][0] if row["name"] else ""

def average_num_friends(rows):
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows):
    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)
```

Et la somme des `user_id` est :

```
def sum_user_ids(rows): 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 souhaiterez trier les résultats de vos calculs. Par exemple, on veut connaître les deux premiers noms de nos 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):
    new_table = self.select() # fait 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` de SQL vous permet de spécifier un ordre `ASC` (croissant) ou `DESC` (décroissant) pour chaque champ. Ici, nous devons intégrer cet aspect dans notre fonction `order` (en paramètre de `order_by`).

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 sujets à chaque utilisateur.

Les tables SQL, elles, ne peuvent pas contenir de listes. La solution classique consiste à créer une deuxième table `user_interests` décrivant les relations `1-n` entre les `user_id` et les centres d'intérêt. En SQL, on peut écrire ceci :

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

alors qu'en NotQuiteABase on créerait une table ainsi :

```
user_interests = Table(["user_id", "interest"])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

Note

Cette méthode de stockage contient encore de nombreuses redondances, comme le centre d'intérêt « SQL » stocké à deux endroits différents. Dans une vraie base de données, on stockerait `user_id` et `interest_id` dans la table `user_interests` et on créerait une troisième table `interests` pour faire correspondre `interest_id` à `interest` de façon à ne stocker les noms des sujets qu'une seule fois. Ici, cela ne ferait que rendre nos exemples plus compliqués que nécessaire.

Comment peut-on analyser nos données lorsqu'elles sont disséminées dans différentes tables ? En créant des jointures sur les tables. `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 est spécifiée dans la clause de jointure.

Par exemple, pour trouver les utilisateurs intéressés par le SQL, votre requête sera :

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

Dans cet exemple, `JOIN` dit que, pour chaque ligne de la table users, nous devons regarder le `user_id` et associer cette ligne 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. Il s'agit d'une instruction `INNER JOIN`, qui 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, en plus des combinaisons de lignes correspondantes, retourne une ligne pour chaque ligne de la table de gauche sans correspondance des lignes (dans ce cas, les champs auraient dû venir de la table de droite sont mis à `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
```

Le `LEFT JOIN` garantit que tous les utilisateurs sans centre d'intérêt auront encore des lignes dans l'ensemble de données jointes (avec des valeurs `NULL` pour les champs provenant de `user_interests`). Et `COUNT` ne compte que les valeurs non nulles.

La méthode NotQuiteABase `join()` sera plus restrictive : elle réalisera la jointure de deux tables sur l'ensemble des colonnes qu'elles ont en commun. Mais, même ainsi, la fonction n'est pas facile à écrire :

```
def join(self, other_table, left_join=False):

    join_on_columns = [c for c in self.columns           # colonnes dans
                       if c in other_table.columns]      # les deux tables

    additional_columns = [c for c in other_table.columns # colonnes seulement
                          if c not in join_on_columns]   # dans table de droite

    # toutes les colonnes de la table de gauche + additional_columns de la table de droite
    join_table = Table(self.columns + additional_columns)

    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

        # chaque ligne qui correspond à celle-ci produit une ligne résultante
```

```
        for other_row in other_rows:
            join_table.insert([row[c] for c in self.columns] +
                              [other_row[c] for c in additional_columns])

        # si aucune correspondance de ligne et LEFT JOIN, résultat avec Nones
        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
```

Donc, nous pourrions trouver les utilisateurs intéressés par le SQL avec :

```
sql_users = users \
    .join(user_interests) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=["name"])
```

Nous pourrions obtenir le nombre de centre d'intérêt par utilisateur ainsi :

```
def count_interests(rows):
    """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 un `RIGHT JOIN`, qui conserve les lignes de la table de droite sans correspondance, et un `FULL OUTER JOIN`, qui conserve les lignes de deux tables sans correspondance.

Nous ne les implémenterons pas.

Les 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 faire une sous-requête. (On peut faire le même calcul avec une instruction `JOIN`, mais cela n'illustrerait pas bien 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, nous pouvons déjà le faire (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 })
```

Les index

Pour chercher des lignes qui contiennent une valeur particulière (par exemple, dont le nom est « Hero »), NotQuiteABase doit inspecter chaque ligne de la table. Si cette dernière contient un grand nombre de lignes, cette recherche peut prendre assez longtemps.

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.

De plus, vous avez 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, ce qui vous permet de faire des recherches rapides de lignes par les colonnes clés, de joindre des tables avec efficacité et de mettre en place 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 nature de la base de données), mais si vous comptez utiliser intensivement vos bases de données, l'effort de départ portera ses fruits.

L’optimisation des requêtes

Souvenez-vous de la requête qui vous permettait de trouver tous les utilisateurs qui s’intéressent 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 vous pouvez 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. Mais filtrer avant est très certainement plus efficace, car 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 est de se tourner vers des bases de données non relationnelles « NoSQL » qui ne représentent pas les données dans des tables. Par exemple, MongoDB, une base de données très utilisée, est sans schéma et ses éléments sont des documents JSON plus ou moins complexes plutôt que des lignes.

Il existe également des bases de données colonnes, qui stockent les données dans des colonnes plutôt que dans des lignes (parfaitement adaptées quand les données contiennent un grand nombre de colonnes, mais que les requêtes n'en utilisent que quelques-unes), des stockages optimisés par clé-valeur pour récupérer des valeurs uniques (complexes) par leurs clés, des bases de données pour stocker et parcourir des graphes, des bases de données optimisées pour s'exécuter sur plusieurs datacenters, des bases de données conçues pour s'exécuter en mémoire, des bases de données pour stocker des séries temporelles, et des centaines d'autres.

Ce qui sera à la mode 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. Donc, maintenant, vous savez : il existe.

Pour aller plus loin

- Si vous souhaitez télécharger une base de données relationnelle pour vous exercer, SQLite est rapide et peu encombrante, alors que MySQL et PostgreSQL sont plus volumineuses et offrent de nombreuses possibilités. Toutes sont gratuites et très bien documentées.
- Si vous voulez explorer NoSQL, MongoDB 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 contient certainement des liens vers des bases de données qui n'existaient pas encore au moment de l'écriture de ce livre.

24

MapReduce

Le futur est déjà arrivé, mais il n'a pas encore été distribué partout.
– William Gibson (auteur de science-fiction canadien du XX^{e} s)

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. Ces derniers peuvent par exemple être des journaux de sites web, les textes de différents livres, des fichiers images, ou toute autre chose.

Une version de base de l'algorithme MapReduce suivra les étapes ci-dessous.

1. Utiliser une fonction de correspondance (`mapper`) pour transformer chaque élément en zéro ou plus de paires clé-valeur. (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.

Tout ceci est toutefois un peu abstrait. Voyons maintenant un exemple. Il existe peu de règles absolues en data science ; toutefois, une d'elles dit que votre premier exemple en MapReduce doit compter des mots.

Exemple : un 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 personnes dans leurs mises à jour de statut. 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, tout était facile :

```
def word_count_old(documents):
    """comptage des mots sans utiliser MapReduce"""
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

Mais aujourd'hui, avec des millions d'utilisateurs, le contenu de la variable `documents` (mises à jour des statuts) est devenu trop gros pour votre ordinateur. Mais si vous arrivez à le gérer dans un modèle MapReduce, vous pourrez faire appel à une infrastructure Big Data implémentée 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 mot, ce qui veut dire que 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 :

```
def wc_mapper(document):
    """pour chaque mot du document, émettre (mot,1)"""
    for word in tokenize(document):
        yield (word, 1)
```

Sans nous attarder sur la plomberie de l'étape 2 pour le moment, imaginons que vous avez collecté, pour un mot donné, une liste des compteurs correspondants émis. Pour produire le comptage global pour ce mot, vous utiliserez cette fonction :

```
def wc_reducer(word, counts):
    """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 ordinateur seulement :

```
def word_count(documents):
    """compter les mots dans les documents en entrée à l'aide de MapReduce"""

    # lieu de stockage des valeurs groupées
    collector = defaultdict(list)

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.iteritems()
            for output in wc_reducer(word, counts)]
```

Imaginons maintenant que nous avons trois documents `["data science", "Big Data", "science fiction »]`.

`wc_mapper` appliqué au premier document retourne les deux paires `("data", 1)` et `("science", 1)`. Et 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 ?

Comme je l'ai déjà mentionné, le premier avantage de MapReduce est qu'il nous permet de distribuer les calculs en déplaçant le traitement vers les données. Imaginons que vous voulez 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 ou 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.

> **Note**
> En réalité, elle peut en traiter plusieurs à la fois si elle est multiprocesseur et si le code est réécrit pour bénéficier de cet avantage. Mais, même ainsi, tous les documents doivent transiter par 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 suivre cette démarche :

- 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 machine « de réduction », grouper les paires de clés produites en exécutant le `reducer` sur chaque ensemble de valeurs ;
- et enfin, 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 associés au 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 a 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 qu'au prix de quelques modifications, nous disposons d'un cadre beaucoup plus général (qui s'exécute toujours sur une seule machine) :

```
def map_reduce(inputs, mapper, reducer):
"""exécute MapReduce sur les entrées avec mapper et 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.iteritems()
           for output in reducer(key,values)]
```

Nous pouvons alors compter les mots simplement :

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

Cette flexibilité nous permet de faire face à une grande variété de situations.

Avant de continuer, notez que `wc_reducer` ne fait que sommer les valeurs correspondant à chaque clé. Ce genre d'agrégation est assez courant et il est intéressant de le faire apparaître :

```
def reduce_values_using(aggregation_fn, key, values):
    """réduit une paire de clé-valeur en appliquant aggregation_fn à ces valeurs"""
    yield (key, aggregation_fn(values))

def values_reducer(aggregation_fn):
    """transforme une fonction (valeurs -> sortie) en fonction de réduction qui fait
    correspondre à (clé-valeur) -> (clé-sortie)"""
    return partial(reduce_values_using, aggregation_fn)
```

après quoi nous pouvons créer facilement :

```
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)))
```

et ainsi de suite.

Exemple : analyser les mises à jour de statuts

Le responsable Contenu est impressionné par le comptage des mots. Il veut savoir ce que vous pouvez apprendre d'autre des mises à jour de statuts des utilisateurs. Vous décidez d'extraire un jeu de données de mises à jour de statuts qui ressemble à ceci :

```
{"id": 1,
 "username" : "joelgrus",
 "text" : "Is anyone interested in a data science book?",
 "created_at" : datetime.datetime(2013, 12, 21, 11, 47, 0),
 "liked_by" : ["data_guy", "data_gal", "mike"] }
```

Par exemple, cherchons quel jour de la semaine voit le plus de conversations autour de la data science. Pour cela, nous allons nous contenter de compter le nombre de mises à jour contenant les mots data science par jour de la semaine. Cela signifie qu'il faut faire un groupement sur le jour de la semaine, qui sera donc notre clé. Et si nous émettons une valeur de `1` pour chaque mise à jour qui contient `data science`, nous pouvons calculer le total très facilement avec `sum` :

```
def data_science_day_mapper(status_update):
    """renvoie (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 avez besoin pour chaque utilisateur de trouver le mot le plus usité dans ses mises à jour de statut. Trois méthodes permettent de faire cela avec la fonction `mapper` :

- 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 ;
- ou 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 celui qui est le plus

populaire. La première option est donc la bonne :

```
def words_per_user_mapper(status_update):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
        yield (user, (word, 1))

def most_popular_word_reducer(user, words_and_counts):
    """pour une séquence donnée de paires (mot-compteur)
    retourner le mot avec le comptage global 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 « likers » distincts pour chaque utilisateur :

```
def liker_mapper(status_update):
    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 : la multiplication matricielle

Vous vous souvenez sûrement de la « multiplication matricielle » de la page 258 affirmant qu'étant donné une matrice A $m \times n$ et une matrice B $n \times k$, le résultat de leur multiplication sera une matrice C $m \times k$. L'élément de C à la ligne i et de la colonne j étant donné par :

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{in}B_{nj}$$

Comme nous l'avons vu, il est « naturel » de représenter une matrice $m \times n$ comme une liste de listes, A_{ij} étant le $j^{ème}$ élément de la $i^{ème}$ liste.

Les très grandes matrices sont souvent peu denses, c'est-à-dire que la plupart des éléments sont nuls. Avec de larges matrices peu denses, une représentation sous forme de liste de listes peut donc représenter une grande perte de place.

Dans ce cas, une liste de tuples (`name, i, j, value`) est plus compacte. `name` étant le nom de la matrice, et `i, j, value` indiquent un emplacement non nul.

Par exemple, une matrice 1 milliard × 1 milliard possède un trillion d'entrées, ce qui ne va pas être facile à stocker sur un ordinateur. Mais 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 effectuer des produits matriciels de manière distribuée.

Pour comprendre notre algorithme, notez que chaque élément A_{ij} est seulement utilisé pour calculer les éléments de C dans la ligne i, et chaque élément B_{ij} 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 émette des clés identifiant une entrée unique de C.

Cela nous amène à suggérer les fonctions suivantes :

```
def matrix_multiply_mapper(A_rows, B_cols, element):
    """element est un tuple (matrix_name, i, j, value)"""
    matrix, i, j, value = element


    if matrix == "A":
        for column in range(B_cols):
            # A_ij est la j-ième entrée de la somme de chaque C_i_column
            yield((i, column), (j, value))
    else:
        for row in range(A_rows):
            # B_ij est la i-ième entrée de la somme de chaque C_row_j
            yield((row, j), (i, value))
```

```
def matrix_multiply_reducer(key, indexed_values):
    results_by_index = defaultdict(list)
    for index, value in indexed_values:
        results_by_index[index].append(value)

      # fait la somme de tous les produits des positions pour lesquelles on a deux résultats
    sum_product = sum(results[0] * results[1]
                      for results in results_by_index.values()
                      if len(results) == 2)

    if sum_product != 0.0:
        yield (key, sum_product)
```

Par exemple, si vous avez deux matrices :

```
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 = [("A", 0, 0, 3), ("A", 0, 1, 2),
           ("B", 0, 0, 4), ("B", 0, 1, -1), ("B", 1, 0, 10)]
mapper = partial(matrix_multiply_mapper, 2, 3)
reducer = matrix_multiply_reducer
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 combiners

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.

Un des raisons qui nous en a empêchés est que, dans un environnement distribué, nous voulons parfois utiliser des fonctions `combiner` pour réduire la quantité de données à transférer de machine en machine. Si une de nos machines `mapper` voit le mot « data » 500 fois, 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 réductrice. Ainsi, on a beaucoup moins de données en mouvement, ce qui peut accélérer substantiellement notre algorithme.

Notre implémentation 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

- Le système MapReduce le plus largement utilisé est Hadoop, qui mériterait plusieurs livres à lui tout seul. Il en existe des distributions commerciales et non commerciales, accompagnées d'un énorme ecosystème d'outils.

  Pour l'utiliser, vous devez mettre en place votre propre cluster (ou trouver quelqu'un qui vous laisse utiliser le sien), une tâche qui n'est pas à la portée du premier venu. Les mappers et les reducers Hadoop sont souvent écrits en Java, toutefois il existe un module « Hadoop streaming » qui vous permet de les écrire dans un autre langage (dont Python).
- Amazon.com propose un service Elastic MapReduce qui peut créer et détruire des clusters de manière automatisée, en ne vous faisant payer que le temps passé à les utiliser.
- mrjob est un paquetage Python qui permet de faire l'interface avec Hadoop (ou Elastic MapReduce).
- Les jobs Hadoop ont en général un long temps de latence, ce qui les disqualifie pour les analyses « en temps réel », mais il existe différents outils « temps réel » qui s'ajoutent à Hadoop, ainsi que de nombreux frameworks alternatifs de plus en plus appréciés. Les deux plus populaires sont Spark et Storm.
- Ceci étant dit, il est très probable que désormais la tendance soit à un tout nouveau framework distribué qui n'existait pas quand j'ai écrit ce livre. Vous devrez le trouver par vous-même.

25

En avant pour la data science

Et maintenant, encore une fois, je priai ma hideuse créature d'aller et de prospérer.
– Mary Shelley (Frankenstein)

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.

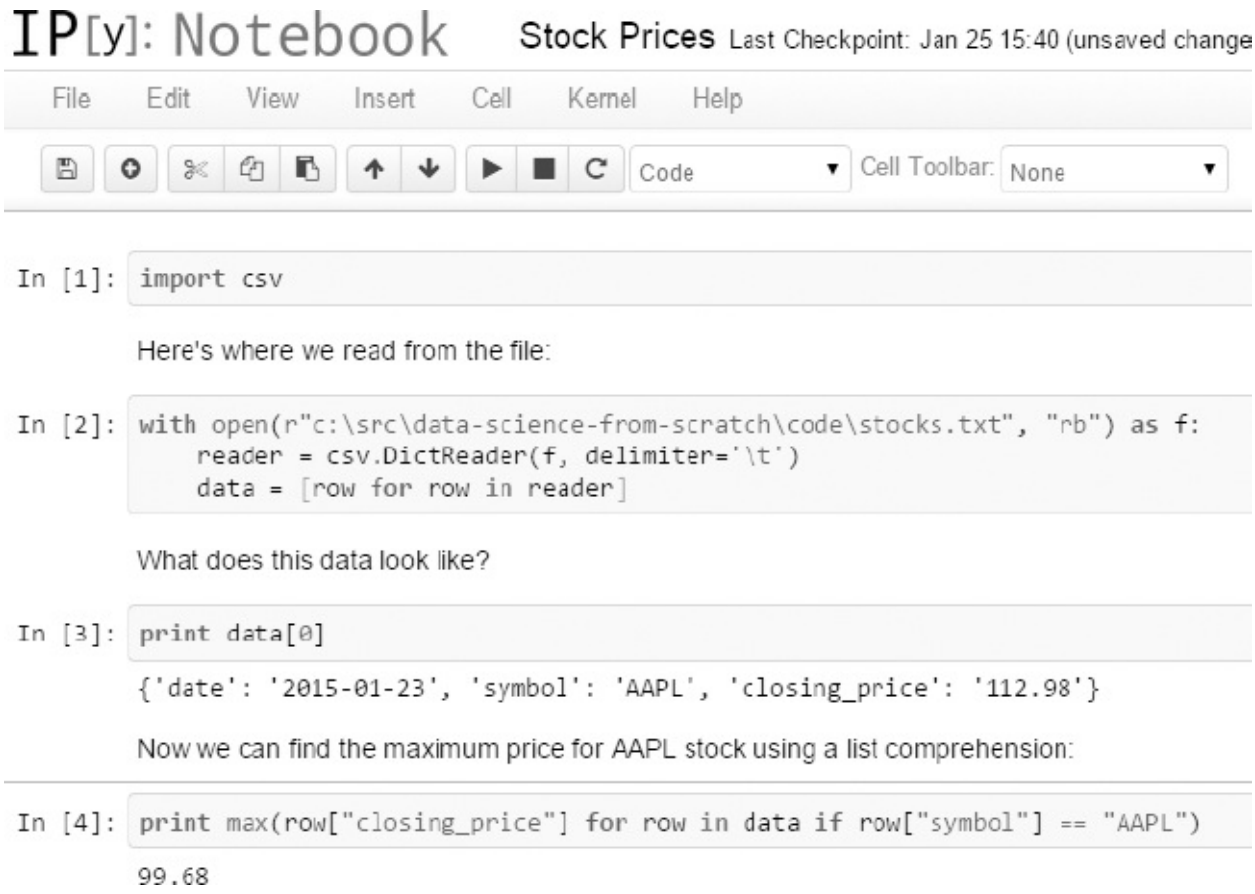
IPython

Nous avons mentionné IPython plus haut dans ce livre. C'est un shell beaucoup plus riche en fonctionnalités que le shell standard de Python : il ajoute des « fonctions magiques » qui vous permettent (entre autres choses) de copier-coller du code facilement (ce qui est normalement assez compliqué en présence de lignes blanches et de mise en page avec des espaces) et d'exécuter des scripts depuis le shell.

La maîtrise d'IPython vous facilitera la vie considérablement (même si vous n'en apprenez qu'un tout petit bout).

De plus, il permet de combiner dans les « notebooks » du texte, du code Python interactif et des représentations graphiques que vous pouvez partager avec d'autres personnes ou simplement conserver dans un journal de vos actions (figure 25-1).

Figure 25–1
Le bloc-notes IPython



Les mathématiques

Tout au long de ce livre, nous avons abordé l'algèbre linéaire (chapitre 4), les statistiques (chapitre 5), les probabilités (chapitre 6) et différents 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

Implémenter des fonctions « à partir de rien » est une excellente technique pour comprendre leur fonctionnement. Mais en général, ce n'est pas très bon pour les performances (sauf si les performances font partie de vos préoccupations premières), et pas très facile, ni rapide à prototyper, ni commode pour gérer les erreurs. En pratique, vous utiliserez le plus souvent des bibliothèques bien conçues qui implémentent les fondamentaux.

NumPy

NumPy (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 matrices de listes de listes et offre beaucoup de fonctions numériques prêtes à l'emploi.

NumPy est un bloc de base qui sert de socle à de nombreuses bibliothèques, ce qui le rend particulièrement intéressant.

pandas

pandas propose des structures de données supplémentaires pour travailler avec des jeux de données en Python. Son concept de base est le `DataFrame`, similaire à une classe `Table` de NotQuiteABase telle que nous l'avons construite au chapitre 23, 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 est certainement la bibliothèque la plus populaire pour l'apprentissage automatique en Python. Elle contient tous les modèles que nous avons implémentés et beaucoup 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.

La documentation de cette bibliothèque contient beaucoup d’exemples de ce qu’il peut faire (et plus généralement de ce que l’apprentissage automatique peut faire).

Les 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, vous avez plusieurs options.

La première est d’explorer matplotlib de manière plus approfondie, car nous n’avons vu que quelques-unes de ses possibilités. Son site web contient de nombreux exemples et une galerie regroupant ceux qui sont les plus intéressants. Si vous voulez créer des représentations statiques (pour un livre), c’est sans doute l’étape suivante pour vous.

Vous devriez aussi explorer seaborn, 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, une bibliothèque JavaScript pour créer des « Data Driven Documents » (les 3 D, des Documents Dirigés par les Données). Si vous ne connaissez pas très bien JavaScript, vous pouvez choisir des exemples de la galerie D3 et les adapter à vos traitements de données. (Les bons data scientists copient depuis la galerie D3 ; les excellents data scientists pillent la galerie D3.) Même si la D3 ne vous intéresse pas, faire défiler sa galerie donne des idées incroyables en matière de représentation des données.

Enfin, Bokeh est un projet qui offre des fonctionnalités de style D3 à Python.

R

Même s’il est possible d’ignorer totalement R, 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 au débat éternel et passionné qui oppose les partisans de R et de Python.

Les tutoriels, cours et livres concernant R sont d'une abondance incroyable.

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). Mais que faire si vous vous intéressez à la data science à titre personnel ? Les données sont partout ; voici quelques points de départ.

- data.gov est le portail ouvert du gouvernement américain. Si vous vous intéressez à ce que fait le gouvernement, c'est votre point de départ.
- redit propose quelques forums, r/datasets et r/data où vous pourrez découvrir et demander des données.
- Amazon.com offre une collection de jeux de données publics pour vous encourager à analyser leurs produits (mais vous êtes libre de les analyser avec ce que vous voulez).
- Rob Seaton a une liste assez originale de données en curation sur son blog.
- Enfin, Kaggle est un site qui héberge des compétitions en data science. Je n'ai jamais réussi à y pénétrer (je n'ai pas l'esprit de compétition quand on parle de data science)… à vous de voir.

Faites de la data science

Explorer des catalogues de données est parfait, mais les meilleurs projets sont ceux qui vous passionnent. En voici quelques-uns que j'ai personnellement réalisés.

Hacker News

Hacker News est un agrégateur d'informations et un site de discussion autour des informations technologiques. Il regroupe de nombreux articles dont la plupart ne m'intéressent pas. Il y a quelques années, j'avais décidé de mettre en place un classificateur d'articles d'Hacker News pour prédire si un article donné pourrait m'intéresser ou pas. 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 du site.

Pour mener à bien mon projet, j'ai labellisé 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 Bayes naïf sur le modèle de notre filtre antispam.

Pour des raisons que l'histoire ne retiendra pas, j'avais programmé en Ruby pour ce projet… Que mes erreurs vous servent de leçon.

Camions de pompiers

J'habite sur une grande artère au centre de Seattle, à mi-chemin entre une station 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 Realtime 911 qui recense toutes les alertes incendie et les véhicules qui y ont été envoyés.

Pour me faire plaisir, j'ai donc récupéré des années de données sur les alertes incendie et je les ai utilisées pour réaliser une analyse de type réseau social sur les camions de pompiers. 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 petite fille et c'est un regret permanent pour moi depuis sa naissance de voir combien les T-shirts « pour filles » sont tristes, comparés aux T-shirts 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 petits garçons. Un jour, je me suis demandé si je pouvais entraîner un modèle à reconnaître les différences.

Fin du suspens : oui.

Pour cela j'ai téléchargé des centaines d'images de T-shirts, je les ai compressées pour 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 un T-shirt ; une autre a cherché les 10 composants principaux des images vectorielles et a classé chaque T-shirt à partir de ses projections dans l'espace à 10 dimensions engendré par les *eigenshirts* (figure 25-2).

Figure 25–2
Eigenshirts correspondants au premier composant principal.

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 faites de la data science.

Partager vos trouvailles ! Envoyez-moi un message à joelgrus@gmail.com ou rejoignez-moi sur Twitter @joelgrus.

Index

A

algèbre linéaire 51
algorithme ID3 204
all 29
allocation de Dirichlet latente 246
Anaconda 15
analyse
  des réseaux 253
  syntaxique du HTML 111
API non authentifiée 119
API Roten Tomatoes 120
apprentissage automatique 7, 145
arbre de décision 199
args 37
argument 19
arithmétique 18
assignations multiples 23
assigner 19

B

barre oblique inverse 17
base de données relationnelle 275
Beautiful Soup 111
BernoulliNB 172
Bokeh 50
booléens 28
Bootstrap 183
boucle tant que 27
break 27

C

CAPTCHA 217
caractère

pipe 106
Unicode 240
caractères spéciaux 20
causalité 69
centralité 260
de vecteur propre 258
internœud 253
chaînes de caractères 20
changement d'échelle 136
classification naïve bayésienne 165
clés de dictionnaires 24
clustering 223
Combiners 296
compteur 26
concaténer des listes 22
connecteurs clés 3
continue 27
corrélation 64, 69
courbes 47
Crab 273
CREATE TABLE 275
csv.writer 110
curryfication 34

D

D3.js 50
data mining 146
data science 2
DataSciencester 3
deep learning 211
defaultdict 24
DELETE 278
descente de gradient 95, 176
stochastique 101, 194
diagramme en bâtons 43
dict 4
dictionnaire 23
dispersion 63
distribution

continue 75
normale 77

E

échantillonnage de Gibbs 244
ensemble (set) 26
entropie 201
d'une partition 203
enumerate 36
except 20
exceptions 20
exploration des données 125
expressions rationnelles 33
extraction de variables 152

F

False (faux) 28
Feed-Forward 213
fichiers à délimiteur 109
filtrage collaboratif 267
fonction 19
logistique 191
forêts aléatoires 209
forme (shape) 56

G

générateurs 30
gensim 252
Gephi 264
gestionnaire de paquetage Python 16
ggplot 50
ggplot2 50
GitHub 119
grammaire 242
graphes orientés 262
Graphlab 273
GROUP BY 280

H

Hadoop 241

hyperplan 196
hypothèse 83

I

if 27
if-then-else 27
in 26
indentation 16
inférence bayésienne 91
INSERT 275
installer Python 15
intervalle de confiance 88
IPython 16, 299
itérateurs 30
itération 8

J

jeu de données 59
JOIN 283
JSON 118

K

k plus proches voisins 155
KeyError 23
keys 23
kwargs 37

L

langage de programmation statistique R VI
lecture de fichiers 108
libsvm 198
list comprehension 30
listes 21

M

machine learning 145
malédiction de la dimension 160
manipulation des données 133
MapReduce 289
matplotlib 41, 164, 301

matrice 55
maximum de vraisemblance 177
médiane 62
méthode
  d'extraction 13
  de tri (sort) 29
  get 24
  items() 32
  iteritems() 32
modélisation 145
  thématique 246
module
  csv 109
  linear_model 188
  Python 17
MongoDB 288
moyenne 62
mrjob 297
multiplication matricielle 258, 295

N

Natural Language Toolkit (NLTK) 252
Netflix Prize 273
nettoyage des données 130
NetworkX 264
n-grammes 239
nœud terminal (ou feuille) 206
nœuds décisionnels 204
nombres aléatoires 32
None 28
NoSQL 288
NotQuiteABase 275
nuage
  de mots 237
  de points 48
NumPy VI, 58, 300

O

open 108

ORDER BY 283

P

PageRank 262
paires clé-valeur 24
pandas VI, 69, 124, 301
paradoxe de Simpson 67
partial 35
partitionnement
  de couleurs 228
  hiérarchique 230
perceptron 212
P-hacking 90
pip 16
plt.axis 45
plt.bar 44
plt.plot 47
porte logique 212
probabilité conditionnelle 72
probabilités 71
produire des recommandations 265
produit scalaire 54, 65
programmation orientée objet 33
p-values 86
PyBrain 222
Pylearn2 222
pyplot.imshow 220
Python VI

Q

quantile 62

R

random 32
Random Forests 209
random.random() 76
range 63
re.findall() 167
réduction de dimensionnalité 137

régression
  linéaire multiple 179
  linéaire simple 173
  logistique 189
réseau de neurones 211
rétropropagation 216

S

scikit-learn VI, 103, 143, 164, 301
SciPy 69, 235
scipy.stats 81
Scrapy 124
Seaborn 50
SELECT 279
sélection des variables 152
shell Python 17
sigmoïde 214
similarité cosinus 267
sous-ajustement 147
sous-requête 286
SpamAssassin 169
SQL 275
SQLite 288
StatsModels 69, 188
stdin 105
stdout 105
strip() 108
structures de contrôle 27
surajustement 147

T

théorème
  de Bayes 74
  de la limite centrale 79
traitement du langage naturel (TLN) 237
transformation des données 130
True (vrai) 28
try 20
tuples 22

Twython 120

U
UPDATE 277
utilisation des API 118

V
ValueError 22
values 23
variable aléatoire 75
vecteurs 51
visualisation des données 41

W
while 27

X
XML 118

Y
yield 31

Z
Zen 16
zip 37

This book was posted by AlenMiler on AvaxHome!
https://avxhm.se/blogs/AlenMiler

Pour suivre toutes les nouveautés numériques du Groupe Eyrolles, retrouvez-nous sur Twitter et Facebook

@ebookEyrolles

EbooksEyrolles

Et retrouvez toutes les nouveautés papier sur

@Eyrolles

Eyrolles