O'REILLY®

Deep Learning avec Keras et TensorFlow

Mise en œuvre et cas concrets

3e édition

Aurélien Géron

Traduction de l'anglais par Hervé Soulard
Actualisation par Anne Bohy pour la 3e édition

DUNOD

Authorized French translation of material from the English edition of
Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3E
ISBN 9781098125974

Conception de la couverture : Karen Montgomery
Illustratrice : Kate Dullea

© Dunod, 2017, 2019, 2024
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-084769-3

Table des matières

Avant-propos

L'intelligence artificielle en pleine explosion

Auriez-vous cru, il y a seulement 10 ans, que vous pourriez aujourd'hui poser toutes sortes de questions à voix haute à votre téléphone, et qu'il réponde correctement ? Que des voitures autonomes sillonneraient déjà les rues (surtout américaines, pour l'instant) ? Qu'un logiciel, AlphaGo, parviendrait à vaincre Ke Jie, le champion du monde du jeu de go, alors que, jusqu'alors, aucune machine n'était jamais arrivée à la cheville d'un grand maître de ce jeu ? Que des intelligences artificielles (IA) génératives telles que DALL-E ou MidJourney pourraient produire toutes sortes d'images sur demande, allant même jusqu'à remporter des compétitions artistiques ? Ou encore qu'une IA conversationnelle telle que ChatGPT verrait le jour, capable de discuter de tout, de corriger ou traduire vos documents, d'inventer des histoires, de composer des poèmes, ou encore de programmer ?

Au rythme où vont les choses, on peut se demander ce qui sera possible dans 10 ans ! Les docteurs feront-ils quotidiennement appel à des IA pour les assister dans leurs diagnostics ? Les jeunes écouteront-ils des tubes personnalisés, composés spécialement pour eux par des machines analysant leurs habitudes, leurs goûts et leurs réactions ? Des robots pleins d'empathie tiendront-ils compagnie aux personnes âgées ? Quels sont vos pronostics ? Notez-les bien et rendez-vous dans 10 ans ! Une chose est sûre : le monde ressemble de plus en plus à un roman de science-fiction.

L'apprentissage automatique se démocratise

Au cœur de ces avancées extraordinaires se trouve le Machine Learning (ML, ou *apprentissage automatique*) : des systèmes informatiques capables d'apprendre à partir d'exemples. Bien que le ML existe depuis plus de 50 ans, il n'a véritablement pris son envol que depuis une douzaine d'années, d'abord dans les laboratoires de recherche, puis très vite chez les géants du web, notamment les GAFA (Google, Apple, Facebook et Amazon).

À présent, le Machine Learning envahit les entreprises de toutes tailles. Il les aide à analyser des volumes importants de données et à en extraire les informations les plus utiles (*data mining*). Il peut aussi détecter automatiquement les anomalies de

production, repérer les tentatives de fraude, segmenter une base de clients afin de mieux cibler les offres, prévoir les ventes (ou toute autre série temporelle), classer automatiquement les prospects à appeler en priorité, optimiser le nombre de conseillers de clientèle en fonction de la date, de l'heure et de mille autres paramètres, etc. La liste d'applications s'agrandit de jour en jour.

Cette diffusion rapide du Machine Learning est rendue possible en particulier par trois facteurs :

- Les entreprises sont pour la plupart passées au numérique depuis longtemps : elles ont ainsi des masses de données facilement disponibles, à la fois en interne et *via* Internet.
- La puissance de calcul considérable nécessaire pour l'apprentissage automatique est désormais à la portée de tous les budgets, en partie grâce à la loi de Moore[1], et en partie grâce à l'industrie du jeu vidéo : en effet, grâce à la production de masse de cartes graphiques puissantes, on peut aujourd'hui acheter pour un prix d'environ 1 000 € une carte graphique équipée d'un processeur GPU capable de réaliser des milliers de milliards de calculs par seconde[2]. En l'an 2000, le superordinateur ASCI White d'IBM avait déjà une puissance comparable… mais il avait coûté 110 millions de dollars ! Et bien sûr, si vous ne souhaitez pas investir dans du matériel, vous pouvez facilement louer des machines virtuelles dans le cloud.
- Enfin, grâce à l'ouverture grandissante de la communauté scientifique, toutes les découvertes sont disponibles quasi instantanément pour le monde entier, notamment sur *https://arxiv.org*. Dans combien d'autres domaines peut-on voir une idée scientifique publiée puis utilisée massivement en entreprise la même année ? À cela s'ajoute une ouverture comparable chez les GAFA : chacun s'efforce de devancer l'autre en matière de publication de logiciels libres, en partie pour soigner son image de marque, en partie pour que ses outils dominent et que ses solutions de cloud soient ainsi préférées, et, qui sait, peut-être aussi par altruisme (il n'est pas interdit de rêver). Il y a donc pléthore de logiciels libres d'excellente qualité pour le Machine Learning.

Dans ce livre, nous utiliserons TensorFlow, développé par Google et passé en open source fin 2015. Il s'agit d'un outil capable d'exécuter toutes sortes de calculs de façon distribuée, et particulièrement optimisé pour entraîner et exécuter des réseaux de neurones artificiels. Comme nous le verrons, TensorFlow contient notamment une excellente implémentation de l'API Keras, qui simplifie grandement la création et l'entraînement de réseaux de neurones artificiels.

L'avènement des réseaux de neurones

Le Machine Learning repose sur un grand nombre d'outils, provenant de plusieurs domaines de recherche : notamment la théorie de l'optimisation, les statistiques,

1. Une loi vérifiée empiriquement depuis 50 ans et qui affirme que la puissance de calcul des processeurs double environ tous les 18 mois.
2. Par exemple, 64 téraFLOPS pour la carte GeForce RTX 4080 de NVidia. Un téraFLOPS égale mille milliards de FLOPS. Un FLOPS est une opération à virgule flottante par seconde.

l'algèbre linéaire, la robotique, la génétique et bien sûr les neurosciences. Ces dernières ont inspiré les réseaux de neurones artificiels, des modèles simplifiés des réseaux de neurones biologiques qui composent votre cortex cérébral : c'était en 1943, il y a plus de 80 ans ! Après quelques années de tâtonnements, les chercheurs sont parvenus à leur faire apprendre diverses tâches, notamment de classification ou de régression (c'est-à-dire prévoir une valeur en fonction de plusieurs paramètres). Malheureusement, lorsqu'ils n'étaient composés que de quelques couches successives de neurones, ces réseaux ne semblaient capables d'apprendre que des tâches rudimentaires. Et lorsque l'on tentait de rajouter davantage de couches de neurones, on se heurtait à des problèmes en apparence insurmontables : d'une part, ces réseaux de neurones « profonds » exigeaient une puissance de calcul rédhibitoire pour l'époque, des quantités faramineuses de données, et surtout, ils s'arrêtaient obstinément d'apprendre après seulement quelques heures d'entraînement, sans que l'on sache pourquoi. Dépités, la plupart des chercheurs ont abandonné le *connexionnisme*, c'est-à-dire l'étude des réseaux de neurones, et se sont tournés vers d'autres techniques d'apprentissage automatique qui semblaient plus prometteuses, telles que les arbres de décision ou les machines à vecteurs de support (SVM).

Seuls quelques chercheurs particulièrement déterminés ont poursuivi leurs recherches : à la fin des années 1990, l'équipe de Yann Le Cun est parvenue à créer un réseau de neurones à convolution (CNN, ou ConvNet) capable d'apprendre à classer très efficacement des images de caractères manuscrits. Mais chat échaudé craint l'eau froide : il en fallait davantage pour que les réseaux de neurones ne reviennent en odeur de sainteté.

Enfin, une véritable révolution eut lieu en 2006 : Geoffrey Hinton et son équipe mirent au point une technique capable d'entraîner des réseaux de neurones profonds, et ils montrèrent que ceux-ci pouvaient apprendre à réaliser toutes sortes de tâches, bien au-delà de la classification d'images. L'apprentissage profond, ou *Deep Learning*, était né. Suite à cela, les progrès sont allés très vite, et, comme vous le verrez, la plupart des articles de recherche cités dans ce livre datent d'après 2010.

Objectif et approche

Pourquoi ce livre ? Quand je me suis mis au Machine Learning, j'ai trouvé plusieurs livres excellents, de même que des cours en ligne, des vidéos, des blogs, et bien d'autres ressources de grande qualité, mais j'ai été un peu frustré par le fait que le contenu était d'une part complètement éparpillé, et d'autre part généralement très théorique, et il était souvent très difficile de passer de la théorie à la pratique.

J'ai donc décidé d'écrire le livre *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (ou HOML), avec pour objectif de couvrir les principaux domaines du Machine Learning, des simples modèles linéaires aux SVM en passant par les arbres de décision et les forêts aléatoires, et bien sûr aussi le Deep Learning et même l'apprentissage par renforcement (Reinforcement Learning, ou RL). Je voulais que le livre soit utile à n'importe quelle personne ayant un minimum d'expérience de

programmation (si possible en Python[3]), en axant l'apprentissage sur la pratique, avec de nombreux exemples de code. Vous retrouverez ainsi tous les exemples de code de ce livre sur *https://github.com/ageron/handson-ml3*, sous la forme de notebooks Jupyter.

Notes sur l'édition française

La première moitié de HOML est une introduction au Machine Learning, reposant sur la bibliothèque Scikit-Learn[4]. La seconde moitié est une introduction au Deep Learning, reposant sur les bibliothèques Keras et TensorFlow. Dans l'édition française, ce livre a été scindé en deux :

- la première partie (chapitres 1 à 9) a été traduite dans le livre *Machine Learning avec Scikit-Learn*, aux éditions Dunod (3e édition, 2023) ;
- la seconde partie (chapitres 10 à 19) a été traduite dans le livre que vous tenez entre les mains, *Deep Learning avec Keras et TensorFlow*. Les chapitres ont été renumérotés de 2 à 11, et un nouveau chapitre 1 a été ajouté, reprenant les points essentiels de la première partie.

Exemples de code

Tous les exemples figurant dans ce livre sont en open source et disponibles sous *https://github.com/ageron/handson-ml3* en tant que notebooks Jupyter : il s'agit de documents interactifs comportant du texte, des images et des fragments de code exécutable (en Python dans notre cas). La façon la plus simple et la plus rapide de commencer est d'exécuter ces notebooks en utilisant Google Colab, un service gratuit vous permettant d'exécuter directement en ligne n'importe quel notebook Jupyter, sans avoir à installer quoi que ce soit sur votre machine. Vous aurez seulement besoin d'un navigateur internet et d'un compte Google.

Dans ce livre, je supposerai que vous utilisez Google Colab, mais j'ai aussi testé les notebooks sur d'autres plateformes en ligne comme Kaggle ou Binder, que vous pouvez donc utiliser si vous préférez. Sinon, vous pouvez aussi installer les bibliothèques et outils requis (ou l'image Docker de ce livre) et exécuter les notebooks directement sur votre propre ordinateur : reportez-vous aux instructions figurant sur la page *https://homl.info/install*.

Prérequis

Bien que ce livre ait été écrit plus particulièrement pour les ingénieurs en informatique, il peut aussi intéresser toute personne sachant programmer et ayant quelques bases mathématiques. Il ne requiert aucune connaissance préalable sur le Machine Learning mais il suppose les prérequis suivants :

3. J'ai choisi le langage Python d'une part parce que c'est mon langage de prédilection, mais aussi parce qu'il est simple et concis, ce qui permet de remplir le livre de nombreux exemples de code. En outre, il s'agit actuellement du langage le plus utilisé en Machine Learning.
4. Cette bibliothèque a été créée par David Cournapeau en 2007, et le projet est maintenant dirigé par une équipe de chercheurs à l'Institut national de recherche en informatique et en automatique (Inria).

- vous devez avoir un minimum d'expérience de programmation ;
- sans forcément être un expert, vous devez connaître le langage Python, et si possible également ses bibliothèques scientifiques, en particulier NumPy, pandas et Matplotlib ;
- enfin, si vous voulez comprendre comment les algorithmes fonctionnent (ce qui n'est pas forcément indispensable, mais est tout de même très recommandé), vous devez avoir certaines bases en mathématiques dans les domaines suivants :
  – l'algèbre linéaire, notamment comprendre les vecteurs et les matrices (par exemple comment multiplier deux matrices, transposer ou inverser une matrice),
  – le calcul différentiel, notamment comprendre la notion de dérivée, de dérivée partielle, et savoir comment calculer la dérivée d'une fonction.

Si vous ne connaissez pas encore Python, il existe de nombreux tutoriels sur Internet, que je vous encourage à suivre : ce langage est très simple et s'apprend vite. En ce qui concerne les bibliothèques scientifiques de Python et les bases mathématiques requises, le site *github.com/ageron/handson-ml3* propose quelques tutoriels (en anglais) sous la forme de notebooks Jupyter. De nombreux tutoriels en français sont disponibles sur Internet. Le site *fr.khanacademy.org* est particulièrement recommandé pour les mathématiques.

Plan du livre

- Le chapitre 1 reprend les éléments du livre *Machine Learning avec Scikit-Learn* qui sont indispensables pour comprendre le Deep Learning. Il présente d'abord Google Colab, qui est l'interface en ligne gratuite et recommandée pour exécuter facilement tous les exemples de code de ce livre sans avoir à installer quoi que ce soit sur votre ordinateur (des instructions d'installation sont disponibles sur *github.com/ageron/handson-ml3* si vous préférez exécuter le code sur votre machine). Puis il présente les bases du Machine Learning, comment entraîner divers modèles linéaires à l'aide de la descente de gradient, pour des tâches de régression et de classification, et il présente quelques techniques de régularisation.
- Le chapitre 2 introduit les réseaux de neurones artificiels et montre comment les mettre en œuvre avec Keras.
- Le chapitre 3 montre comment résoudre les difficultés particulières que l'on rencontre avec les réseaux de neurones profonds.
- Le chapitre 4 présente l'API de bas niveau de TensorFlow, utile lorsque l'on souhaite personnaliser les rouages internes des réseaux de neurones.
- Le chapitre 5 montre comment charger et transformer efficacement de gros volumes de données lors de l'entraînement d'un réseau de neurones artificiels.
- Le chapitre 6 présente les réseaux de neurones convolutifs et leur utilisation pour la vision par ordinateur.
- Le chapitre 7 montre comment analyser des séries temporelles à l'aide de réseaux de neurones récurrents, ou avec des réseaux de neurones convolutifs.

- Le chapitre 8 présente le traitement automatique du langage naturel à l'aide de réseaux de neurones récurrents, ou de réseaux de neurones dotés de mécanismes d'attention, notamment le fameux transformeur, au cœur de systèmes tels que ChatGPT.
- Le chapitre 9 traite de l'apprentissage automatique de représentations à l'aide d'autoencodeurs ou de réseaux antagonistes génératifs (GAN). L'objectif est de découvrir, avec ou sans supervision, des motifs dans les données. Ces architectures de réseaux de neurones artificiels sont également utiles pour générer de nouvelles données semblables à celles reçues en exemple (par exemple pour générer des images de visages). Ce chapitre présente également les modèles de diffusion, sur lesquels reposent de nombreux systèmes de génération d'images, tels que Dall-E 2 ou Stable Diffusion.
- Le chapitre 10 aborde l'apprentissage par renforcement, dans lequel un agent apprend par tâtonnements au sein d'un environnement dans lequel il peut recevoir des récompenses ou des punitions. Nous étudierons notamment la technique employée par DeepMind pour créer une IA capable d'apprendre toute seule à jouer à de nombreux jeux Atari, jusqu'à atteindre souvent un niveau surhumain.
- Le chapitre 11 présente comment entraîner et déployer à grande échelle les réseaux de neurones artificiels construits avec TensorFlow.

Les différents types de textes en exergue

Ce symbole indique une astuce ou une suggestion.

Ce symbole indique une précision ou une remarque générale.

Ce symbole indique une difficulté particulière ou un piège à éviter.

Remerciements

Jamais, dans mes rêves les plus fous, je n'aurais imaginé que la deuxième édition de ce livre rencontrerait un public aussi vaste. J'ai reçu de nombreux messages de lecteurs, avec beaucoup de questions, certains signalant gentiment des erreurs et la plupart m'envoyant des mots encourageants. Je suis extrêmement reconnaissant envers tous ces lecteurs pour leur formidable soutien. Merci beaucoup à vous tous ! N'hésitez pas à me contacter si vous voyez des erreurs dans les exemples de code ou simplement pour poser des questions (*https://homl.info/issues3*) ! Certains lecteurs ont également expliqué en quoi ce livre les avait aidés à obtenir leur premier emploi ou à résoudre un problème concret sur lequel ils travaillaient. Ces retours sont incroyablement motivants. Si vous trouvez ce livre utile, j'aimerais beaucoup que vous puissiez partager votre histoire avec moi, que ce soit en privé (par exemple, via *https://linkedin.*

com/in/aurelien-geron) ou en public (par exemple dans un tweet *via* @aureliengeron ou par le biais d'un commentaire Amazon).

Grand merci aussi à toutes les personnes merveilleuses qui ont offert de leur temps et leur expertise pour réviser cette troisième édition, en corrigeant des erreurs et en faisant d'innombrables suggestions. Cette édition est tellement meilleure grâce à cela : Olzhas Akpambetov, George Bonner, Francois Chollet, Siddha Ganju, Sam Goodman, Matt Harrison, Sasha Sobran, Lewis Tunstall, Leandro von Werra et mon cher frère Sylvain. Vous êtes tous incroyables !

Je suis également très reconnaissant envers toutes les personnes qui m'ont soutenu tout au long du chemin, en répondant à mes questions, en suggérant des améliorations et en contribuant au code sur GitHub : en particulier, Yannick Assogba, Ian Beauregard, Ulf Bissbort, Rick Chao, Peretz Cohen, Kyle Gallatin, Hannes Hapke, Victor Khaustov, Soonson Kwon, Éric Lebigot, Jason Mayes, Laurence Moroney, Sara Robinson, Joaquin Ruales et Yuefeng Zhou.

Ce livre n'existerait pas sans le personnel fantastique d'O'Reilly, en particulier Nicole Taché, qui m'a fait des commentaires perspicaces, toujours encourageants et utiles : je ne pouvais pas rêver d'un meilleur éditeur. Merci également à Michele Cronin, qui m'a encouragé et m'a permis d'arriver au bout. Merci à toute l'équipe de la production, en particulier Elizabeth Kelly et Kristen Brown. Merci également à Kim Cofer pour la révision minutieuse, et à Johnny O'Toole, qui a géré la relation avec Amazon et répondu à beaucoup de mes questions. Merci à Kate Dullea pour l'amélioration importante de mes illustrations. Merci à Marie Beaugureau, Ben Lorica, Mike Loukides et Laurel Ruma d'avoir cru en ce projet et de m'avoir aidé à le définir. Merci à Matt Hacker et à toute l'équipe d'Atlas pour avoir répondu à toutes mes questions techniques concernant AsciiDoc, MathML et LaTeX, ainsi qu'à Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis et tous les autres membres d'O'Reilly qui ont contribué à ce livre.

Je n'oublierai jamais les personnes formidables qui m'ont aidé sur les deux premières éditions du livre : amis, collègues, experts, dont de nombreux membres de l'équipe de TensorFlow. La liste est longue : Olzhas Akpambetov, Karmel Allison, Martin Andrews, David Andrzejewski, Paige Bailey, Lukas Biewald, Eugene Brevdo, William Chargin, François Chollet, Clément Courbet, Robert Crowe, Mark Daoust, Daniel « Wolff » Dobson, Julien Dubois, Mathias Kende, Daniel Kitachewsky, Nick Felt, Bruce Fontaine, Justin Francis, Goldie Gadde, Irene Giannoumis, Ingrid von Glehn, Vincent Guilbeau, Sandeep Gupta, Priya Gupta, Kevin Haas, Eddy Hung, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Jon Krohn, Allen Lavoie, Karim Matrah, Grégoire Mesnil, Clemens Mewald, Dan Moldovan, Dominic Monn, Sean Morgan, Tom O'Malley, James Pack, Alexander Pak, Haesun Park, Alexandre Passos, Ankur Patel, Josh Patterson, André Susano Pinto, Anthony Platanios, Anosh Raj, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Salim Sémaoune, Ryan Sepassi, Vitor Sessak, Jiri Simsa, Iain Smears, Xiaodan Song, Christina Sorokin, Michel Tessier, Wiktor Tomczak, Dustin Tran, Todd Wang, Pete Warden, Rich Washington, Martin Wicke, Edd Wilder-James, Sam Witteveen, Jason Zaman, Yuefeng Zhou, et mon frère Sylvain.

Je souhaite également remercier Jean-Luc Blanc, des éditions Dunod, pour avoir soutenu ce projet, et pour la gestion et les relectures attentives des deux premières éditions. Merci également à Matthieu Daniel, qui a pris la suite de Jean-Luc Blanc pour cette troisième édition. Je tiens aussi à remercier vivement Hervé Soulard pour la traduction des deux premières éditions et Anne Bohy pour la traduction de la troisième. Enfin, je remercie chaleureusement Brice Martin, des éditions Dunod, pour sa relecture extrêmement rigoureuse, ses excellentes suggestions et ses nombreuses corrections.

Pour finir, je suis infiniment reconnaissant à ma merveilleuse épouse, Emmanuelle, et à nos trois enfants, Alexandre, Rémi et Gabrielle, de m'avoir encouragé à travailler dur pour ce livre. Leur curiosité insatiable fut inestimable : expliquer certains des concepts les plus difficiles de ce livre à ma femme et à mes enfants m'a aidé à clarifier mes pensées et à améliorer directement de nombreuses parties de ce livre. J'ai même eu droit à des biscuits et du café, qui pourrait en demander davantage ?

1

Les fondamentaux du Machine Learning

Avant de partir à l'assaut du mont Blanc, il faut être entraîné et bien équipé. De même, avant d'attaquer le Deep Learning avec TensorFlow et Keras, il est indispensable de maîtriser les bases du Machine Learning. Si vous avez lu le livre *Machine Learning avec Scikit-Learn* (A. Géron, Dunod, 3e édition, 2023), vous êtes prêt(e) à passer directement au chapitre 2. Dans le cas contraire, ce chapitre vous donnera les bases indispensables pour la suite[5].

Nous commencerons par découvrir Google Colab, un service web gratuit qui est bien utile pour exécuter du code Python en ligne, sans avoir à installer quoi que ce soit sur votre machine.

Ensuite, nous étudierons la régression linéaire, l'une des techniques d'apprentissage automatique les plus simples qui soient. Cela nous permettra au passage de rappeler ce qu'est le Machine Learning, ainsi que le vocabulaire et les notations que nous emploierons tout au long de ce livre. Nous verrons deux façons très différentes d'entraîner un modèle de régression linéaire : premièrement, une méthode analytique qui trouve directement le modèle optimal (c'est-à-dire celui qui s'ajuste au mieux au jeu de données d'entraînement) ; deuxièmement, une méthode d'optimisation itérative appelée *descente de gradient* (en anglais, *gradient descent* ou GD), qui consiste à modifier graduellement les paramètres du modèle de façon à l'ajuster petit à petit au jeu de données d'entraînement.

Nous examinerons plusieurs variantes de cette méthode de descente de gradient que nous utiliserons à maintes reprises lorsque nous étudierons les réseaux de

5. Ce premier chapitre reprend en grande partie le chapitre 4 du livre *Machine Learning avec Scikit-Learn* (3e édition, 2023), ainsi que quelques éléments essentiels des chapitres 1 à 3 de ce dernier.

neurones artificiels : descente de gradient groupée (ou batch), descente de gradient par mini-lots (ou mini-batch) et descente de gradient stochastique.

Nous examinerons ensuite la régression polynomiale, un modèle plus complexe pouvant s'ajuster à des jeux de données non linéaires. Ce modèle ayant davantage de paramètres que la régression linéaire, il est plus enclin à surajuster (*overfit*, en anglais) le jeu d'entraînement. C'est pourquoi nous verrons comment détecter si c'est ou non le cas à l'aide de courbes d'apprentissage, puis nous examinerons plusieurs techniques de régularisation qui permettent de réduire le risque de surajustement du jeu d'entraînement.

Enfin, nous étudierons deux autres modèles qui sont couramment utilisés pour les tâches de classification : la régression logistique et la régression softmax.

Ces notions prises individuellement ne sont pas très compliquées, mais il y en a beaucoup à apprendre dans ce chapitre, et elles sont toutes indispensables pour la suite, alors accrochez-vous bien, c'est parti !

1.1 INTRODUCTION À GOOGLE COLAB

N'hésitez pas à prendre votre ordinateur et à exécuter pas à pas les exemples de code. Comme je l'ai indiqué dans la préface, tous les exemples figurant dans ce livre sont en open source et disponibles sous *https://github.com/ageron/handson-ml3* en tant que notebooks Jupyter : il s'agit de documents interactifs comportant du texte, des images et des fragments de code exécutable (en Python dans notre cas). Dans la suite de ce livre, je supposerai que vous utilisez ces notebooks dans Google Colab, un service gratuit vous permettant d'exécuter directement en ligne n'importe quel notebook Jupyter, sans avoir à installer quoi que ce soit sur votre machine. Si vous souhaitez utiliser une autre plateforme en ligne (comme par exemple Kaggle) ou si vous préférez tout installer localement sur votre propre ordinateur, reportez-vous aux instructions figurant sur la page GitHub de ce livre.

1.1.1 Exécuter les exemples de code en utilisant Google Colab

Ouvrez tout d'abord un navigateur web et tapez *https://homl.info/colab3* : ceci vous conduira sur Google Colab et affichera la liste des notebooks Jupyter pour ce livre (voir figure 1.1). Vous trouverez un notebook par chapitre, ainsi que quelques notebooks et aide-mémoire supplémentaires pour NumPy, Matplotlib, Pandas, l'algèbre linéaire et le calcul différentiel. Si vous cliquez par exemple sur `02_end_to_end_machine_learning_project.ipynb`, le notebook du chapitre 2 de la version originale s'ouvrira dans Google Colab (voir figure 1.2). Comme expliqué dans l'avant-propos, le chapitre que vous êtes en train de lire est un résumé de la première partie de la version originale, et il contient des exemples de code du chapitre 4 de la version originale. Les chapitres 2 à 11 de ce livre correspondent aux chapitres 10 à 19 de la version originale.

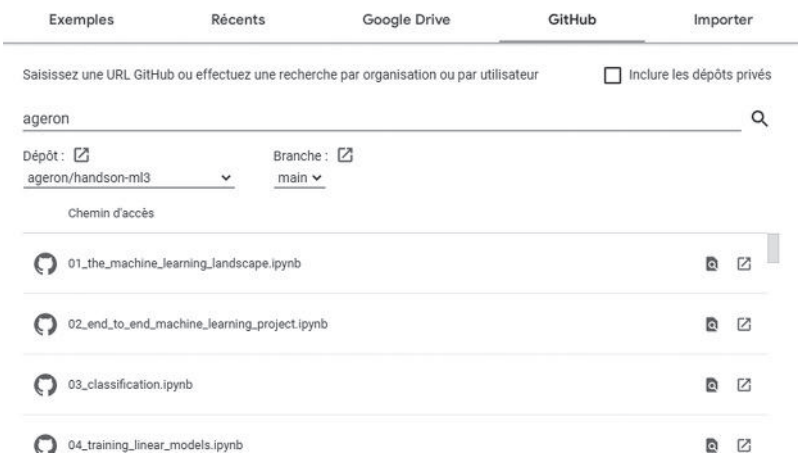

Figure 1.1 – Liste des notebooks dans Google Colab

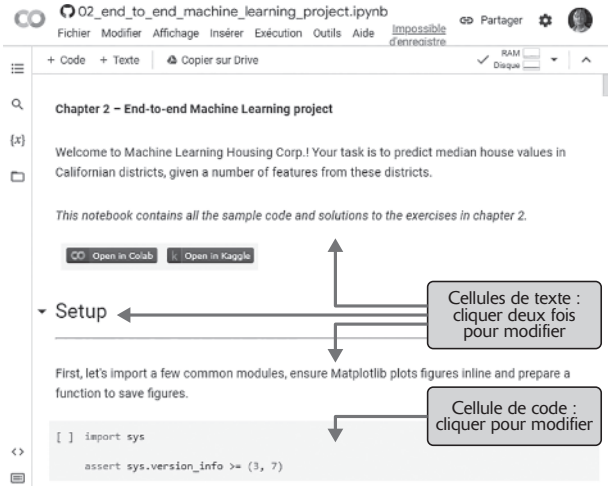

Figure 1.2 – Votre notebook dans Google Colab

Un notebook Jupyter est constitué d'une suite de cellules. Chaque cellule contient soit du code exécutable, soit du texte. Essayez de double-cliquer sur la première cellule de texte (qui contient la phrase « Welcome to Machine Learning Housing Corp.! »). Ceci ouvrira la cellule en mode Mise à jour. Notez que les notebooks Jupyter comportent des balises de formatage (p. ex. **gras**, *italiques*, # Titre, [texte du lien](URL), etc.). Essayez de modifier ce texte, puis appuyez sur Maj+Entrée pour voir le résultat.

Ensuite, créez une nouvelle cellule comportant du code en sélectionnant Insérer → Cellule de code dans le menu. Vous pouvez également utiliser le bouton + Code dans

la barre d'outils, ou placer le curseur de votre souris sur le bas de la cellule pour faire apparaître les options + Code et + Texte puis cliquer sur + Code. Saisissez du code Python dans la nouvelle cellule de code, comme par exemple `print("Bonjour !")`, puis appuyez sur Maj+Entrée pour l'exécuter (ou cliquez sur le bouton ▷ près du bord gauche de la cellule).

Si vous n'êtes pas encore connecté à votre compte Google, il vous sera demandé de le faire maintenant (si vous n'avez pas encore de compte Google, il vous faudra en créer un). Une fois connecté, vous verrez apparaître un avertissement de sécurité indiquant que ce notebook n'a pas été créé par Google. Une personne mal intentionnée pourrait créer un notebook tentant de vous leurrer pour vous faire saisir votre mot de passe Google lui permettant d'accéder à vos données personnelles, c'est pourquoi, avant d'exécuter un notebook, vous devez toujours vous assurer que vous pouvez faire confiance à son auteur (ou vérifier soigneusement ce que va faire chaque cellule de code avant de l'exécuter). Si vous me faites confiance (ou si vous comptez vérifier chaque cellule de code), vous pouvez cliquer maintenant sur « Exécuter malgré tout ».

Colab vous allouera un nouvel environnement d'exécution (ou runtime) : il s'agit d'une machine virtuelle gratuite, hébergée sur les serveurs de Google et incluant un grand nombre d'outils et de bibliothèques Python, en particulier tout ce qu'il vous faut pour la plupart des chapitres (dans quelques-uns d'entre eux, vous devrez exécuter une commande pour installer des bibliothèques supplémentaires). Cela prendra quelques secondes. Après quoi, Colab vous connectera automatiquement à ce runtime et l'utilisera pour exécuter le code de votre nouvelle cellule. Chose importante, le code s'exécute dans ce runtime, et non sur votre machine. La sortie associée à votre code s'affichera sous la cellule. Félicitations, vous venez d'exécuter du code Python dans Colab !

Pour insérer une nouvelle cellule de code, vous pouvez aussi taper Ctrl-M (ou Cmd-M sur macOS) suivi de A (comme « Above », pour insérer au-dessus de la cellule active) ou B (comme « Below », pour insérer en dessous). Vous disposez de nombreux autres raccourcis clavier : vous pouvez les visualiser et les modifier en tapant Ctrl-M (ou Cmd-M) puis H. Si vous choisissez d'exécuter les notebooks dans Kaggle ou sur votre propre machine en utilisant JupyterLab ou un environnement de développement tel que Visual Studio Code avec l'extension Jupyter, vous constaterez quelques différences mineures (les runtimes sont appelés *kernels*, l'interface utilisateur et les raccourcis clavier sont légèrement différents, etc.), mais il n'est pas trop difficile de passer d'un environnement Jupyter à un autre.

1.1.2 Sauvegarder vos modifications de code et vos données

Si vous apportez des modifications dans un notebook Colab, elles persisteront tant que l'onglet correspondant restera ouvert dans votre navigateur. Mais une fois que vous l'aurez fermé, les modifications seront perdues. Pour éviter cela, prenez soin d'enregistrer une copie de votre notebook dans votre drive Google en sélectionnant

Fichier → Enregistrer une copie dans Drive. Vous pouvez aussi télécharger le notebook sur votre ordinateur en sélectionnant Fichier → Télécharger → Télécharger le fichier .ipynb. Vous pourrez par la suite vous rendre sur *https://colab.research.google.com* et rouvrir le notebook (soit à partir de Google Drive, soit en le rechargeant à partir de votre ordinateur).

Google Colab n'est conçu que pour un usage interactif : vous pouvez vous amuser dans les notebooks et modifier le code à votre idée, mais vous ne pouvez pas demander aux notebooks de tourner pendant longtemps sans intervention de votre part, car dans un tel cas le runtime serait interrompu et toutes ses données seraient perdues.

Si le notebook génère des données importantes pour vous, veillez à télécharger ces données avant la fermeture de votre runtime. Pour cela, cliquez sur l'icône Fichiers (voir figure 1.3, étape 1), trouvez le fichier que vous voulez télécharger, cliquez sur la barre verticale pointillée à côté de celui-ci (étape 2), puis cliquez sur Télécharger (étape 3). Vous pouvez aussi monter votre drive Google sur le runtime, ce qui permet au notebook de lire et écrire des fichiers directement sur Google Drive comme s'il s'agissait d'un dossier local. Pour cela, cliquez sur l'icône Fichiers (étape 1), puis cliquez sur l'icône Google Drive (entourée d'un cercle sur la figure 1.3) et suivez les instructions à l'écran.

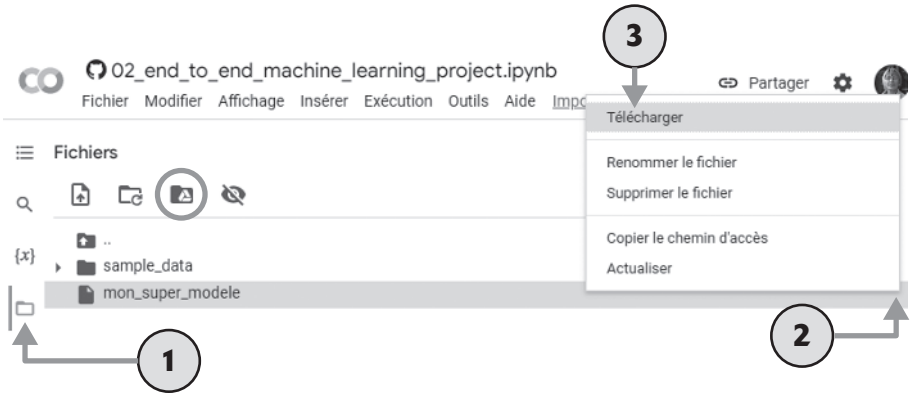

Figure 1.3 – Téléchargement d'un fichier à partir du runtime de Google Colab (étapes 1 à 3), ou montage de votre drive Google (icone encerclée)

Par défaut, votre drive Google sera monté sur */content/drive/MyDrive*. Si vous voulez sauvegarder un fichier de données, copiez-le simplement sous ce dossier en exécutant : `!cp /content/mon_super_modele /content/drive/MyDrive`. Toute commande débutant par un caractère « ! » (parfois appelé caractère bang) est interprétée comme une commande système et non comme une commande Python : dans cet exemple, `cp` est la commande Linux permettant de copier un fichier vers un autre emplacement. Remarquez que les runtimes de Colab s'exécutent sous Linux (et plus précisément, sa distribution Ubuntu).

1.1.3 Puissance et danger de l'interactivité

Les notebooks Jupyter sont interactifs, ce qui est très pratique : vous pouvez exécuter les cellules une par une, vous arrêter quand vous voulez, insérer une cellule, modifier le code, revenir en arrière et ré-exécuter la cellule, etc., et je vous encourage vivement à le faire. Si vous vous contentez d'exécuter les cellules l'une après l'autre sans jamais les modifier, vous n'apprendrez pas aussi vite. Cependant, cette souplesse a un prix : il est très facile d'exécuter les cellules dans le mauvais ordre ou d'oublier d'en exécuter une. Si cela se produit, il y a des chances que l'exécution des cellules suivantes échoue. En particulier, la première cellule de code de chaque notebook comporte des instructions d'initialisation (telles que des importations), par conséquent veillez à l'exécuter en premier, sinon rien ne fonctionnera.

Si vous obtenez une erreur que vous ne comprenez pas, essayez de redémarrer le runtime (en sélectionnant, dans le menu, Exécution → Redémarrer l'environnement d'exécution) puis ré-exécutez toutes les cellules depuis le début du notebook. Ceci résout souvent le problème. Sinon, il est probable que l'une de vos modifications a introduit une erreur majeure dans le notebook : revenez simplement au notebook d'origine et essayez à nouveau. En cas de nouvel échec, signalez le problème sur le projet GitHub.

1.1.4 Différences entre le code du livre et le code du notebook

Vous trouverez peut-être parfois quelques petites différences entre le code de ce livre et celui des notebooks. Il peut y avoir plusieurs raisons à cela :

- Une bibliothèque logicielle peut avoir évolué légèrement au moment où vous lirez ces lignes, ou peut-être qu'en dépit de ma vigilance j'ai fait une erreur dans le livre. Malheureusement, je n'ai pas de baguette magique pour corriger le code dans ce livre mais je peux corriger les notebooks. Par conséquent, si vous obtenez une erreur à l'exécution après avoir copié du code figurant dans ce livre, vérifiez si vous en trouvez une version modifiée dans les notebooks : je m'efforcerai de les conserver sans erreur et compatibles avec les versions les plus récentes des bibliothèques logicielles.
- Les notebooks comportent un peu de code additionnel pour améliorer les figures (pour ajouter des libellés, définir des tailles de police, etc.) et les enregistrer en haute résolution afin d'illustrer ce livre. Vous pouvez sans problème ignorer ce code supplémentaire si vous le souhaitez.

J'ai privilégié la lisibilité et la simplicité du code en le rendant aussi linéaire et plat que possible, en ne définissant que quelques fonctions et classes. Ceci, pour que le code que vous êtes en train d'exécuter se trouve sous vos yeux, sans avoir à le rechercher sous plusieurs couches d'abstraction, et aussi pour que vous puissiez le modifier aisément. Par souci de simplicité, la gestion des erreurs est limitée, et j'ai placé certains des imports utilisés le moins couramment à proximité de l'endroit où ils sont utilisés (au lieu de les placer en début de fichier, selon les recommandations de la PEP 8 Python). Ceci dit, votre code de production ne sera

pas très différent : juste un petit peu plus modulaire, avec davantage de tests et de gestion d'erreurs.

OK ! Maintenant que vous êtes familiarisé avec Colab et que vous pouvez exécuter du code Python, vous êtes prêt à apprendre les bases du Machine Learning.

1.2 QU'EST-CE QUE LE MACHINE LEARNING ?

Le *Machine Learning* (apprentissage automatique) est la science (et l'art) de programmer les ordinateurs de sorte qu'ils puissent apprendre à partir de données.

Voici une définition un peu plus générale :

> « [*L'apprentissage automatique est la*] *discipline donnant aux ordinateurs la capacité d'apprendre sans qu'ils soient explicitement programmés.* »
>
> Arthur Samuel, 1959

En voici une autre plus technique :

> « *Étant donné une tâche T et une mesure de performance P, on dit qu'un programme informatique apprend à partir d'une expérience E si les résultats obtenus sur T, mesurés par P, s'améliorent avec l'expérience E.* »
>
> Tom Mitchell, 1997

Votre filtre anti-spam, par exemple, est un programme d'apprentissage automatique qui peut apprendre à identifier les e-mails frauduleux à partir d'exemples de pourriels ou « *spam* » (par exemple, ceux signalés par les utilisateurs) et de messages normaux (parfois appelés « *ham* »). Les exemples utilisés par le système pour son apprentissage constituent le jeu d'entraînement (en anglais, *training set*). Chacun d'eux s'appelle une observation d'entraînement (on parle aussi d'échantillon ou d'instance). Dans le cas présent, la tâche T consiste à identifier parmi les nouveaux e-mails ceux qui sont frauduleux, l'expérience E est constituée par les données d'entraînement, et la mesure de performance P doit être définie. Vous pourrez prendre par exemple le pourcentage de courriels correctement classés. Cette mesure de performance particulière, appelée *exactitude* (en anglais, *accuracy*), est souvent utilisée dans les tâches de classification.

Pour cette tâche de classification, l'apprentissage requiert un jeu de données d'entraînement « étiqueté » (voir figure 1.4), c'est-à-dire pour lequel chaque observation est accompagnée de la réponse souhaitée, que l'on nomme étiquette ou cible (*label* ou *target* en anglais). On parle dans ce cas d'*apprentissage supervisé*.

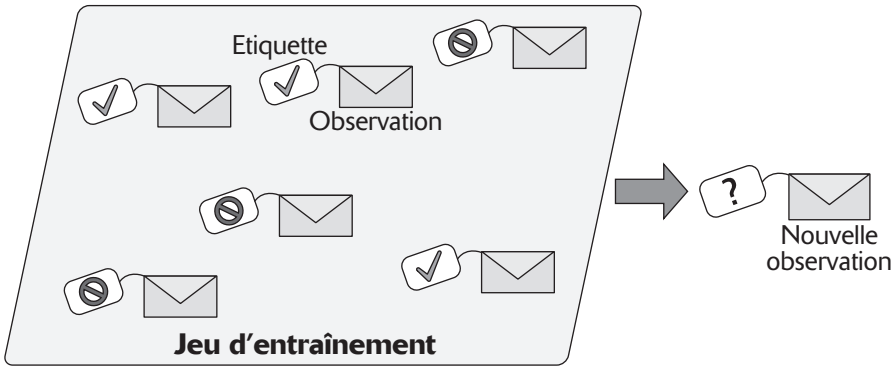

Figure 1.4 – Jeu d'entraînement étiqueté pour une tâche de classification (détection de spam)

Une autre tâche très commune pour un système d'auto-apprentissage est la tâche de « régression », c'est-à-dire la prédiction d'une valeur. Par exemple, on peut chercher à prédire le prix de vente d'une maison en fonction de divers paramètres (sa superficie, le revenu médian des habitants du quartier…). Tout comme la classification, il s'agit d'une tâche d'apprentissage supervisé : le jeu de données d'entraînement doit posséder, pour chaque observation, la valeur cible. Pour mesurer la performance du système, on peut par exemple calculer l'erreur moyenne commise par le système (ou, plus fréquemment, la racine carrée de l'erreur quadratique moyenne, comme nous le verrons dans un instant).

Il existe également des tâches de Machine Learning pour lesquelles le jeu d'entraînement n'est pas étiqueté. On parle alors d'apprentissage non supervisé. Par exemple, si l'on souhaite construire un système de détection d'anomalies (p. ex. pour détecter les produits défectueux dans une chaîne de production, ou pour détecter des tentatives de fraudes), on ne dispose généralement que de très peu d'exemples d'anomalies, donc il est difficile d'entraîner un système de classification supervisé. On peut toutefois entraîner un système performant en lui donnant des données non étiquetées (supposées en grande majorité normales), et ce système pourra ensuite détecter les nouvelles observations qui sortent de l'ordinaire. Un autre exemple d'apprentissage non supervisé est le partitionnement d'un jeu de données, par exemple pour segmenter les clients en groupes semblables, à des fins de marketing ciblé (voir figure 1.5). Enfin, la plupart des algorithmes de réduction de dimension, dont ceux dédiés à la visualisation des données, sont aussi des exemples d'algorithmes d'apprentissage non supervisé.

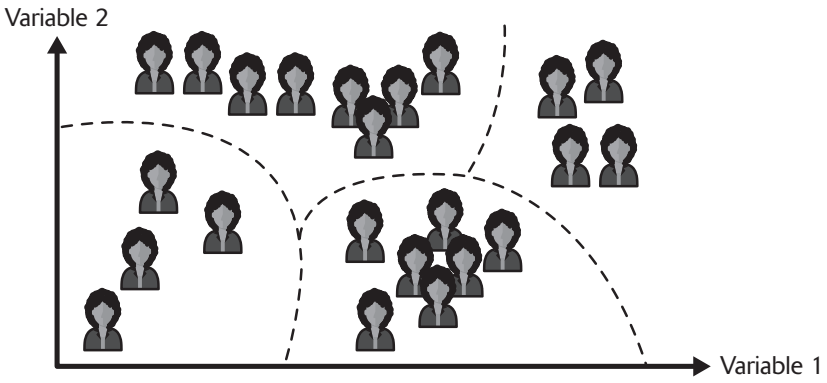

Figure 1.5 – Le partitionnement, un exemple d'apprentissage non supervisé

Résumons : on distingue les tâches d'apprentissage supervisé (classification, régression…), et les tâches d'apprentissage non supervisé (partitionnement, détection d'anomalie, réduction de dimension...). Un système de Machine Learning passe en général par deux phases : pendant la phase d'apprentissage il est entraîné sur un jeu de données d'entraînement, puis pendant la phase d'inférence il applique ce qu'il a appris sur de nouvelles données et effectue des prédictions. Il existe toutes sortes de variantes de ce schéma général, mais c'est le principe à garder à l'esprit.

1.3 COMMENT LE SYSTÈME APPREND-IL ?

L'approche la plus fréquente consiste à créer un modèle prédictif et d'en régler les paramètres afin qu'il fonctionne au mieux sur les données d'entraînement. Par exemple, pour prédire le prix d'une maison en fonction de sa superficie et du revenu médian des habitants du quartier, on pourrait choisir un modèle linéaire, c'est-à-dire dans lequel la valeur prédite est une somme pondérée des paramètres, plus un *terme constant* (en anglais, *intercept* ou *bias*). Cela donnerait l'équation suivante :

Équation 1.1 – Un modèle linéaire du prix des maisons

$$prix = \theta_0 + \theta_1 \times superficie + \theta_2 \times revenu\ médian$$

Dans cet exemple, le modèle a trois paramètres : θ_0, θ_1 et θ_2. Le premier est le terme constant, et les deux autres sont les *coefficients de pondération* (ou poids) des variables d'entrée. La phase d'entraînement de ce modèle consiste à trouver la valeur de ces paramètres qui minimise l'erreur du modèle sur le jeu de données d'entraînement.[6]

6. Le nom « terme constant » peut être un peu trompeur dans le contexte du Machine Learning car il s'agit bien de l'un des paramètres du modèle que l'on cherche à optimiser, et qui varie donc pendant l'apprentissage. Toutefois, dès que l'apprentissage est terminé, ce terme devient bel et bien constant. Le nom anglais *bias* porte lui aussi à confusion car il existe une autre notion de biais, sans aucun rapport, présentée plus loin dans ce chapitre.

Une fois les paramètres réglés, on peut utiliser le modèle pour faire des prédictions sur de nouvelles observations : c'est la phase d'inférence (ou de prédiction). L'espoir est que si le modèle fonctionne bien sur les données d'entraînement, il fonctionnera également bien sur de nouvelles observations (c'est-à-dire pour prédire le prix de nouvelles maisons). Si la performance est bien moindre, on dit que le modèle a « surajusté » le jeu de données d'entraînement. Cela arrive généralement quand le modèle possède trop de paramètres par rapport à la quantité de données d'entraînement disponibles et à la complexité de la tâche à réaliser. Une solution est de réentraîner le modèle sur un plus gros jeu de données d'entraînement, ou bien de choisir un modèle plus simple, ou encore de contraindre le modèle, ce qu'on appelle la *régularisation* (nous y reviendrons dans quelques paragraphes). À l'inverse, si le modèle est mauvais sur les données d'entraînement (et donc très probablement aussi sur les nouvelles données), on dit qu'il « sous-ajuste » les données d'entraînement. Il s'agit alors généralement d'utiliser un modèle plus puissant ou de diminuer le degré de régularisation.

Formalisons maintenant davantage le problème de la régression linéaire.

1.4 RÉGRESSION LINÉAIRE

Comme nous l'avons vu, un modèle linéaire effectue une prédiction en calculant simplement une somme pondérée des variables d'entrée, en y ajoutant un terme constant :

Équation 1.2 – Prédiction d'un modèle de régression linéaire

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Dans cette équation :

- $\hat{y}$ est la valeur prédite,
- n est le nombre de variables,
- x_i est la valeur de la $i^{ème}$ variable,
- θ_j est le $j^{ème}$ paramètre du modèle (terme constant θ_0 et coefficients de pondération des variables θ_1, θ_2, …, θ_n).

Ceci peut s'écrire de manière beaucoup plus concise sous forme vectorielle :

Équation 1.3 – Prédiction d'un modèle de régression linéaire (forme vectorielle)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

Dans cette équation :

- $\boldsymbol{\theta}$ est le *vecteur des paramètres* du modèle, il regroupe à la fois le terme constant θ_0 et les coefficients de pondération θ_1 à θ_n (ou poids) des variables. Notez que, dans le texte, les vecteurs sont représentés en minuscule et en gras (par exemple $\mathbf{x}$), les scalaires (les simples nombres) sont représentés en minuscule et en italique, par exemple n, et les matrices sont représentées en majuscule et en gras, par exemple $\mathbf{X}$.

- $\mathbf{x}$ est le *vecteur des valeurs* d'une observation, contenant les valeurs x_0 à x_n, où x_0 est toujours égal à 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ est le produit scalaire de $\boldsymbol{\theta}$ et de $\mathbf{x}$, qui est égal à $\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n$, et que l'on notera dans ce livre $\boldsymbol{\theta}^T \mathbf{x}$.
- $h_{\boldsymbol{\theta}}$ est la fonction hypothèse, utilisant les paramètres de modèle $\boldsymbol{\theta}$.

En Machine Learning, les vecteurs sont souvent représentés sous forme de vecteurs colonnes, qui sont des tableaux 2D avec une seule colonne. Si $\boldsymbol{\theta}$ et $\mathbf{x}$ sont des vecteurs colonnes, alors $\boldsymbol{\theta}^T$ est la transposée de $\boldsymbol{\theta}$ (c'est-à-dire une matrice à une seule ligne, ce qu'on appelle un *vecteur ligne*), et $\boldsymbol{\theta}^T \mathbf{x}$ représente le produit matriciel de $\boldsymbol{\theta}^T$ et de $\mathbf{x}$. C'est bien sûr la même prédiction, sauf qu'elle est maintenant représentée par une matrice à une seule cellule plutôt que par une valeur scalaire. Dans ce livre, j'utiliserai cette notation pour éviter d'alterner entre produit scalaire et produit matriciel.

Par souci d'efficacité, on réalise souvent plusieurs prédictions simultanément. Pour cela, on regroupe dans une même matrice $\mathbf{X}$ toutes les observations pour lesquelles on souhaite faire des prédictions (ou plus précisément tous leurs vecteurs de valeurs). Par exemple, si l'on souhaite faire une prédiction pour 3 observations dont les vecteurs de valeurs sont respectivement $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$ et $\mathbf{x}^{(3)}$, alors on les regroupe dans une matrice $\mathbf{X}$ dont la première ligne est la transposée de $\mathbf{x}^{(1)}$, la seconde ligne est la transposée de $\mathbf{x}^{(2)}$ et la troisième ligne est la transposée de $\mathbf{x}^{(3)}$:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ (\mathbf{x}^{(3)})^T \end{pmatrix}$$

Pour réaliser simultanément une prédiction pour toutes les observations, on peut alors simplement utiliser l'équation suivante :

Équation 1.4 – Prédictions multiples d'un modèle de régression linéaire

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$$

- $\hat{\mathbf{y}}$ est le vecteur des prédictions. Son $i^{\text{ème}}$ élément correspond à la prédiction du modèle pour la $i^{\text{ème}}$ observation.
- Plus haut, l'ordre n'importait pas car $\boldsymbol{\theta} \cdot \mathbf{x} = \mathbf{x} \cdot \boldsymbol{\theta}$, mais ici l'ordre est important. En effet, le produit matriciel n'est défini que quand le nombre de colonnes de la première matrice est égal au nombre de lignes de la seconde matrice. Ici, la matrice $\mathbf{X}$ possède 3 lignes (car il y a 3 observations) et n+1 colonnes (la première colonne est remplie de 1, et chaque autre colonne correspond à une variable d'entrée). Le vecteur colonne $\boldsymbol{\theta}$ possède n+1 lignes (une pour le terme constant, puis une pour chaque poids de variable d'entrée) et bien sûr une seule colonne. Le résultat $\hat{\mathbf{y}}$ est un vecteur colonne contenant 3 lignes (une par observation) et une colonne. De plus, si vous vous étonnez qu'il n'y ait plus de transposée dans cette équation, rappelez-vous que chaque ligne de $\mathbf{X}$ est déjà la transposée d'un vecteur de valeurs.

Voici donc ce qu'on appelle un *modèle de régression linéaire*. Voyons maintenant comment l'entraîner. Comme nous l'avons vu, entraîner un modèle consiste à définir ses paramètres de telle sorte que le modèle s'ajuste au mieux au jeu de données d'entraînement. Pour cela, nous avons tout d'abord besoin d'une mesure de performance qui nous indiquera si le modèle s'ajuste bien ou mal au jeu d'entraînement. Dans la pratique, on utilise généralement une mesure de l'erreur commise par le modèle sur le jeu d'entraînement, ce qu'on appelle une *fonction de coût*. La fonction de coût la plus courante pour un modèle de régression est la racine carrée de l'*erreur quadratique moyenne* (en anglais, *root mean square error* ou RMSE), définie dans l'équation 1.5 :

Équation 1.5 – Racine carrée de l'erreur quadratique moyenne (RMSE)

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left[h(\mathbf{x}^{(i)}) - y^{(i)}\right]^2}$$

- Notez que, pour alléger les notations, la fonction d'hypothèse est désormais notée h plutôt que $h_{\boldsymbol{\theta}}$, mais il ne faut pas oublier qu'elle est paramétrée par le vecteur $\boldsymbol{\theta}$. De même, nous écrirons simplement RMSE($\mathbf{X}$) par la suite, même s'il ne faut pas oublier que la RMSE dépend de l'hypothèse h.
- m est le nombre d'observations dans le jeu de données.

Pour entraîner un modèle de régression linéaire, il s'agit de trouver le vecteur $\boldsymbol{\theta}$ qui minimise la RMSE. En pratique, il est un peu plus simple et rapide de minimiser l'erreur quadratique moyenne (MSE, simplement le carré de la RMSE), et ceci conduit au même résultat, parce que la valeur qui minimise une fonction positive minimise aussi sa racine carrée.

1.4.1 Équation normale

Pour trouver la valeur de $\boldsymbol{\theta}$ qui minimise la fonction de coût, il s'avère qu'il existe une *solution analytique*, c'est-à-dire une formule mathématique nous fournissant directement le résultat. Celle-ci porte le nom d'*équation normale* (voir équation 1.6).

Équation 1.6 – Équation normale

$$\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

Dans cette équation :

- $\hat{\boldsymbol{\theta}}$ est la valeur de $\boldsymbol{\theta}$ qui minimise la fonction de coût,
- $\mathbf{y}$ est le vecteur des valeurs cibles $y^{(1)}$ à $y^{(m)}$.

Générons maintenant des données à l'allure linéaire sur lesquelles tester cette équation (figure 1.6) :

```
import numpy as np
np.random.seed(42) # pour rendre l'exemple reproductible
m = 100  # nombre d'observations
X = 2 * np.random.rand(m, 1)  # vecteur colonne
y = 4 + 3 * X + np.random.randn(m, 1)  # vecteur colonne
```

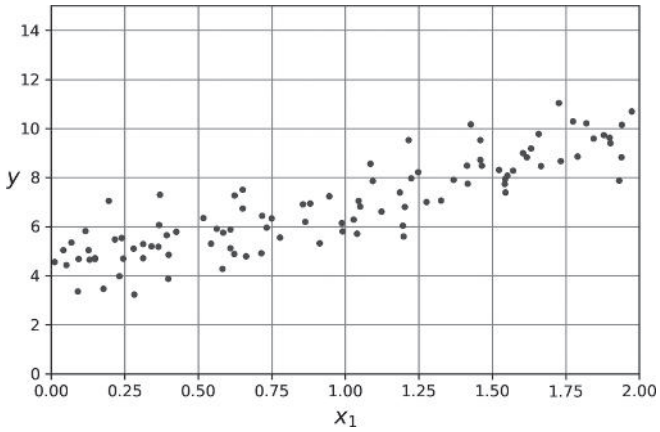

Figure 1.6 – Jeu de données généré aléatoirement

Calculons maintenant $\hat{\boldsymbol{\theta}}$ à l'aide de l'équation normale. Nous allons utiliser la fonction `inv()` du module d'algèbre linéaire `np.linalg` de NumPy pour l'inversion de matrice, et l'opérateur `@` pour les produits matriciels :

```
from sklearn.preprocessing import add_dummy_feature
X_b = add_dummy_feature(X)  # ajouter x0 = 1 à chaque observation
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

L'opérateur `@` effectue un produit matriciel. Si `A` et `B` sont des tableaux NumPy, alors `A @ B` est équivalent à `np.matmul(A,B)`. Beaucoup d'autres bibliothèques telles que TensorFlow, PyTorch et JAX acceptent aussi l'opérateur `@`. Cependant, vous ne pouvez pas utiliser `@` sur de purs tableaux Python (qui sont des listes de listes).

Nous avons utilisé la fonction $y = 4 + 3x_1 +$ bruit gaussien pour générer les données. Voyons ce que l'équation a trouvé :

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

Nous aurions aimé obtenir $\theta_0 = 4$ et $\theta_1 = 3$, au lieu de $\theta_0 = 4{,}215$ et $\theta_1 = 2{,}770$. C'est assez proche, mais le bruit n'a pas permis de retrouver les paramètres exacts de la fonction d'origine. Plus le jeu de données est petit et plus le bruit est important, plus c'est difficile.

Maintenant nous pouvons faire des prédictions à l'aide de $\hat{\boldsymbol{\theta}}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = add_dummy_feature(X_new)  # ajouter x0 = 1 à chaque observation
>>> y_predict = X_new_b(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

Représentons graphiquement les prédictions de ce modèle (figure 1.7) :

```
import matplotlib.pyplot as plt
plt.plot(X_new, y_predict, "r-",label="Prédictions")
plt.plot(X, y, "b.")
[...]  # finitions : libellés, axes, quadrillage et légende
plt.show()
```

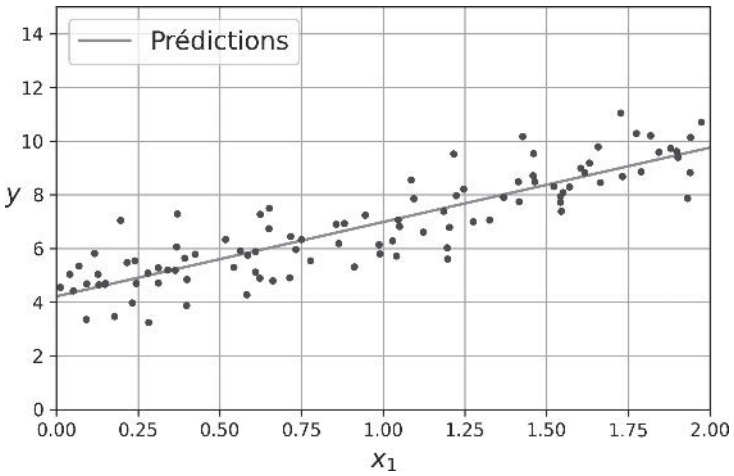

Figure 1.7 – Prédictions du modèle de régression linéaire

Effectuer une régression linéaire avec Scikit-Learn est relativement simple[7] :

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

Remarquez que Scikit-Learn sépare le terme constant (`intercept_`) des coefficients de pondération des variables (`coef_`). La classe `LinearRegression` repose sur la fonction `scipy.linalg.lstsq()` (le nom signifie *least squares*, c'est-à-dire « méthode des moindres carrés »). Vous pourriez l'appeler directement comme suit :

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

7. Notez que Scikit-Learn sépare le terme constant (`intercept_`) des coefficients de pondération des variables (`coef_`).

Cette fonction calcule $\hat{\theta} = \mathbf{X}^{+}\mathbf{y}$, où $\mathbf{X}^{+}$ est la pseudo-inverse de $\mathbf{X}$ (plus précisément la pseudo-inverse de Moore-Penrose). Vous pouvez utiliser `np.linalg.pinv()` pour calculer cette pseudo-inverse directement :

```
>>> np.linalg.pinv(X_b) @ y
array([[4.21509616],
       [2.77011339]])
```

Cette pseudo-inverse est elle-même calculée à l'aide d'une technique très classique de factorisation de matrice nommée *décomposition en valeurs singulières* (en anglais, *singular value decomposition* ou SVD). Cette technique parvient à décomposer le jeu d'entraînement $\mathbf{X}$ en produit de trois matrices $\mathbf{U}$, $\mathbf{\Sigma}$ et $\mathbf{V}^{T}$ (voir `numpy.linalg.svd()`). La pseudo-inverse se calcule ensuite ainsi : $\mathbf{X}^{+}=\mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{T}$. Pour calculer la matrice $\mathbf{\Sigma}^{+}$, l'algorithme prend $\mathbf{\Sigma}$ et met à zéro toute valeur plus petite qu'un seuil minuscule, puis il remplace les valeurs non nulles par leur inverse, et enfin il transpose la matrice. Cette approche est bien plus rapide que de calculer l'équation normale, et elle gère bien les cas limites : en effet, l'équation normale ne fonctionne pas lorsque la matrice $\mathbf{X}^{T}\mathbf{X}$ n'est pas inversible (notamment lorsque $m < n$ ou quand certains attributs sont redondants), alors que la pseudo-inverse est toujours définie.

1.4.2 Complexité algorithmique

L'équation normale calcule l'inverse de $\mathbf{X}^{T}\mathbf{X}$, qui est une matrice $(n+1) \times (n+1)$ (où n est le nombre de variables). La complexité algorithmique d'une inversion de matrice se situe entre $O(n^{2,4})$ et $O(n^{3})$, selon l'algorithme d'inversion utilisé. Autrement dit, si vous doublez le nombre de variables, le temps de calcul est multiplié par un facteur compris entre $2^{2,4} = 5,3$ et $2^{3} = 8$.

L'approche SVD utilisée par la classe `LinearRegression` de Scikit-Learn est d'ordre $O(n^{2})$. Si vous doublez le nombre de variables, vous multipliez approximativement le temps de calcul par 4.

L'équation normale et l'approche SVD deviennent toutes deux très lentes lorsque le nombre de variables devient grand (p. ex. 100 000). En revanche, les deux sont linéaires vis-à-vis du nombre d'observations dans le jeu d'entraînement (algorithmes en $O(m)$), donc elles peuvent bien gérer un gros volume de données, à condition qu'il tienne en mémoire.

Par ailleurs, une fois votre modèle de régression linéaire entraîné (en utilisant l'équation normale ou n'importe quel autre algorithme), obtenir une prédiction est extrêmement rapide : la complexité de l'algorithme est linéaire par rapport au nombre d'observations sur lesquelles vous voulez obtenir des prédictions et par rapport au nombre de variables. Autrement dit, si vous voulez obtenir des prédictions sur deux fois plus d'observations (ou avec deux fois plus de variables), le temps de calcul sera *grosso modo* multiplié par deux.

Nous allons maintenant étudier une méthode d'entraînement de modèle de régression linéaire très différente, mieux adaptée au cas où il y a beaucoup de variables ou trop d'observations pour tenir en mémoire.

1.5 DESCENTE DE GRADIENT

La *descente de gradient* est un algorithme d'optimisation très général, capable de trouver des solutions optimales à un grand nombre de problèmes. L'idée générale de la descente de gradient est de corriger petit à petit les paramètres dans le but de minimiser une fonction de coût.

Supposons que vous soyez perdu en montagne dans un épais brouillard et que vous puissiez uniquement sentir la pente du terrain sous vos pieds. Pour redescendre rapidement dans la vallée, une bonne stratégie consiste à avancer vers le bas dans la direction de plus grande pente. C'est exactement ce que fait la descente de gradient : elle calcule le gradient de la fonction de coût au point $\boldsymbol{\theta}$, puis progresse en direction du gradient descendant. Lorsque le gradient est nul, vous avez atteint un minimum !

En pratique, vous commencez par remplir $\boldsymbol{\theta}$ avec des valeurs aléatoires (c'est ce qu'on appelle l'*initialisation aléatoire*). Puis vous l'améliorez progressivement, pas à pas, en tentant à chaque étape de faire décroître la fonction de coût (ici la MSE), jusqu'à ce que celle-ci converge vers un minimum (voir figure 1.8).

Un élément important dans l'algorithme de descente de gradient est la dimension des pas, que l'on détermine par l'intermédiaire de l'hyperparamètre `learning_rate` (taux d'apprentissage).

Un hyperparamètre est un paramètre de l'algorithme d'apprentissage, et non un paramètre du modèle. Autrement dit, il ne fait pas partie des paramètres que l'on cherche à optimiser pendant l'apprentissage. Toutefois, on peut très bien lancer l'algorithme d'apprentissage plusieurs fois, en essayant à chaque fois une valeur différente pour chaque hyperparamètre, jusqu'à trouver une combinaison de valeurs qui permet à l'algorithme d'apprentissage de produire un modèle satisfaisant. Pour évaluer chaque modèle, on utilise alors un jeu de données distinct du jeu d'entraînement, appelé le *jeu de validation*. Ce réglage fin des hyperparamètres s'appelle *hyperparameter tuning* en anglais.

Si le taux d'apprentissage est trop petit, l'algorithme devra effectuer un grand nombre d'itérations pour converger et prendra beaucoup de temps (voir figure 1.9).