Deep Learning avec TensorFlow

MISE EN ŒUVRE ET CAS CONCRETS

Aurélien Géron

Traduit de l'anglais par Hervé Soulard

DUNOD

Authorized French translation of material from the English edition of
Hands-On Machine Learning with Scikit-Learn and TensorFlow,
ISBN 9781491962299

Conception de la couverture : Randy Comer
Illustratrice : Rebecca Demarest

75993 - (I) - OSB 90° - PCA - EMR
Imprimerie Chirat - 42540 Saint-Just-la-Pendue
Dépôt légal : novembre 2017

Imprimé en France

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-075993-4

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 chaque jour vous utiliseriez des dizaines d'applications intelligentes, des outils de recherche à la traduction automatique en passant par les systèmes de recommandations ?

Au rythme où vont les choses, on peut se demander ce qui sera possible dans 10 ans ! Les docteurs feront-ils appel à des intelligences artificielles (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 dizaine 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 dorer 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 rendu 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.

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, l'algèbre linéaire, la robotique, la génétique et bien sûr les neurosciences. Ces dernières ont inspiré les réseaux de neurones artificiels (RNA), des modèles simplifiés des réseaux de neurones biologiques qui composent votre cortex cérébral : c'était en 1943, il y a plus de 70 ans ! Après quelques années de tâtonnements, les chercheurs sont parvenus à leur faire apprendre diverses tâches, notamment de classification ou

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, 11,8 téraFLOPS pour la carte GeForce GTX 1080 Ti de NVidia. Un téraFLOPS égale mille milliards de FLOPS. Un FLOPS est une opération à virgule flottante par seconde.

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, les RNA 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 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]), et axer l'apprentissage autour de 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-ml*, sous la forme de notebooks Jupyter.

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 (avec le langage R).

Notes sur l'édition française

La première moitié de HOML est une introduction au Machine Learning, reposant sur la librairie Scikit-Learn. La seconde moitié est une introduction au Deep Learning, reposant sur la librairie TensorFlow. Dans l'édition française, ce livre a été scindé en deux :

- la première partie (chapitres 1 à 8) a été traduite dans le livre *Machine Learning avec Scikit-Learn*, aux éditions Dunod ;
- la seconde partie (chapitres 9 à 16) a été traduite dans le livre que vous tenez entre les mains, *Deep Learning avec TensorFlow*. Les chapitres ont été renumérotés de 2 à 9, et un nouveau chapitre 1 a été ajouté, reprenant les points essentiels de la première partie.

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 :

- 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 librairies 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 librairies scientifiques de Python et les bases mathématiques requises, le site *github.com/ageron/handson-ml* 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 montre d'abord comment installer TensorFlow et le projet contenant les exemples de code du livre (ainsi que les librairies dont il dépend), 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 la librairie TensorFlow.

- Le chapitre 3 présente les réseaux de neurones et comment les mettre en œuvre avec TensorFlow.
- Le chapitre 4 montre comment résoudre les difficultés particulières que l'on rencontre avec les réseaux de neurones profonds.
- Le chapitre 5 explique comment distribuer des réseaux de neurones sur plusieurs processeurs, éventuellement sur plusieurs serveurs.
- Le chapitre 6 présente les réseaux de neurones à convolution, excellents pour analyser des images.
- Le chapitre 7 présente les réseaux de neurones récurrents, capables de traiter des séquences, telles que des séries temporelles ou les mots d'une phrase.
- Le chapitre 8 traite des autoencodeurs : ces réseaux de neurones sont capables d'apprendre, sans supervision, à détecter des motifs dans les données. Ils sont également très utiles pour générer de nouvelles données semblables à celles reçues en exemple (par exemple pour générer des images de visages).
- Le chapitre 9 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 construirons en particulier un agent capable d'apprendre à jouer tout seul à Ms. Pac-Man.

Conventions

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique un nouveau terme, une URL, une adresse email ou un nom de fichier.

`Largeur fixe`

Utilisé pour les exemples de code, ainsi qu'au sein du texte pour faire référence aux éléments d'un programme, tels que des instructions, des mots clés, des noms de variables, de fonctions, de base de données, de types de données ou encore de variables d'environnement.

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

Je tiens tout d'abord à remercier mes collègues de Google, en particulier l'équipe de classification de vidéos YouTube, pour m'avoir tant appris sur l'apprentissage automatique. Je n'aurais jamais pu lancer ce projet sans eux. Un merci tout particulier à mes gourous ML personnels : Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington et tout le monde à YouTube Paris.

Je suis extrêmement reconnaissant à toutes les personnes qui ont pris le temps d'examiner mon livre avec une grande attention. Merci à Pete Warden, membre de l'équipe TensorFlow, qui a répondu à toutes mes questions au sujet de TensorFlow, a relu les chapitres sur le Deep Learning et m'a apporté de nombreux éclairages. N'hésitez pas à aller voir son blog (*petewarden.com*), il en vaut la peine. Un grand merci aussi à Lukas Biewald, qui a relu les chapitres sur le Deep Learning en détail : aucun paragraphe n'a échappé à son regard rigoureux, il a testé tout le code (et permis de résoudre quelques erreurs), il a fait de nombreuses excellentes suggestions, et son enthousiasme a été particulièrement contagieux. Vous devriez également consulter son blog (*lukasbiewald.com*) et ses adorables petits robots (*goo.gl/Eu5u28*). Merci également à Justin Francis, qui a relu les chapitres sur le Deep Learning avec grande attention, découvrant quelques erreurs ici et là et apportant de nombreuses idées, en particulier dans le chapitre sur l'apprentissage par renforcement. N'hésitez pas à découvrir ses articles sur TensorFlow (*goo.gl/28ve8z*) ! Merci à Grégoire Mesnil, qui a relu les chapitres sur le Deep Learning et apporté de nombreuses suggestions pratiques sur la meilleure façon d'entraîner un réseau de neurones.

Concernant la première partie du livre original, un immense merci à David Andrzejewski, qui a relu et a fait des retours très utiles, en identifiant des sections à clarifier et en suggérant comment les améliorer. N'hésitez pas à visiter son site web (*david-andrzejewski.com*). Merci aussi à Eddy Hung, Salim Sémaoune, Karim Matrah, Ingrid von Glehn, Iain Smears et Vincent Guilbeau pour leur lecture attentive et leurs nombreuses suggestions.

Et je souhaite également remercier mon beau-père, Michel Tessier, ancien professeur de mathématiques et maintenant grand traducteur d'Anton Tchekhov, pour m'avoir aidé à éliminer certaines erreurs mathématiques et à clarifier les notations dans ce livre et dans les notebooks Jupyter.

Et bien sûr un immense merci à mon cher frère Sylvain, qui a passé en revue chaque chapitre, a testé toutes les lignes de code, a fourni des commentaires sur pratiquement toutes les sections et m'a encouragé de la première ligne à la dernière. *Love you, bro!*

Un grand merci au personnel fantastique d'O'Reilly, en particulier à Nicole Tache, qui m'a donné des commentaires perspicaces, toujours encourageants et utiles. Merci aussi 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 le formatage asciidoc et LaTeX. Merci également à Rachel Monaghan, Nick Adams et à toute l'équipe de production pour leur relecture finale et leurs centaines de corrections.

Je souhaite également remercier Jean-Luc Blanc, des éditions Dunod, pour avoir soutenu et géré ce projet, et pour ses relectures attentives. Je tiens aussi à remercier vivement Hervé Soulard pour sa traduction. Enfin, je remercie chaleureusement Brice Martin, des éditions Dunod, pour sa relecture extrêmement rigoureuse, ses excellentes suggestions et ses très nombreuses corrections. Je lui dois même un double remerciement car il a également relu et corrigé avec le même souci du détail le livre *Machine Learning avec Scikit-Learn*, dans lequel j'ai malheureusement oublié de le remercier.

Pour finir, je suis infiniment reconnaissant à ma merveilleuse épouse, Emmanuelle, et à nos trois enfants, Alexandre, Rémi et Gabrielle, pour leur patience et leur soutien sans faille pendant l'écriture de ce livre. J'ai même eu droit à des biscuits et du café, comment rêver mieux ?

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, il est indispensable de maîtriser les bases du Machine Learning. Si vous avez lu le livre *Machine Learning avec Scikit-Learn* (A. Géron, Dunod, 2017), vous êtes prêt(e) à passer directement au chapitre suivant. Dans le cas contraire, ce chapitre vous donnera les bases indispensables pour la suite[4].

Nous commencerons par installer TensorFlow et les autres librairies Python dont vous aurez besoin pour exécuter les nombreux exemples de code. Dans ce premier chapitre nous utiliserons uniquement NumPy, Matplotlib et Scikit-Learn : nous attaquerons TensorFlow à partir du prochain chapitre.

Ensuite, nous étudierons la régression linéaire, l'une des techniques d'apprentissage automatique les plus simples qui soient. Cela nous permettra au passage de rappeller 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

4. Ce premier chapitre reprend en grande partie le chapitre 4 du livre *Machine Learning avec Scikit-Learn*, ainsi que quelques éléments essentiels des chapitres 1 à 3.

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 INSTALLER TENSORFLOW

Commençons par vérifier que vous disposez bien de Python 3. Pour cela, ouvrez un terminal et tapez la commande suivante (ne saisissez que ce qui se trouve après le `$`) :

```
$ python3 -V
Python 3.5.3
```

La version de Python doit s'afficher (dans cet exemple, c'est la version 3.5.3). Une version supérieure ou égale à 3.5 est recommandée, mais pas obligatoire. Vous pouvez éventuellement utiliser plutôt Python 2.7 (dans ce cas, utilisez la commande `python` plutôt que `python3`). Toutefois, Python 2 arrive en fin de vie et il est recommandé de passer à Python 3 dès que possible. Si nécessaire, installez donc la version de Python que vous préférez. Par la suite, nous supposerons que vous utilisez Python 3 (note : si vous utilisez une distribution spéciale de Python telle qu'Anaconda, vous devrez adapter les instructions suivantes à votre environnement).

Pour installer TensorFlow, nous utiliserons ici le logiciel pip : il s'agit du gestionnaire de librairies inclus avec Python. Commencez par vous assurer que vous possédez une version récente de pip :

```
$ pip3 install --user --upgrade pip
```

Remplacez la commande `pip3` par `pip` si vous utilisez Python 2. L'option `--user` indique à pip d'installer la mise à jour uniquement pour l'utilisateur actuel, ce qui a l'avantage de ne requérir aucun droit particulier. Si vous préférez une installation système (donc accessible à l'ensemble des utilisateurs de votre machine), vous devez enlever cette option, mais cela exigera alors sans doute les droits administrateur : sous macOS ou Linux, il vous faudra remplacer `pip3` par `sudo pip3`, et sous Windows il vous faudra ouvrir le terminal en mode administrateur. Cette remarque s'applique aussi aux commandes ci-dessous où l'option `--user` apparaît.

Si vous souhaitez installer TensorFlow dans un environnement Python bien isolé du système et des autres projets (ce qui permet à chacun de vos projets de disposer de

sa propre liste de librairies, potentiellement avec des versions de librairies distinctes), il vous suffit d'installer la librairie `virtualenv` et de l'utiliser pour créer un nouvel environnement Python, reposant sur la version de Python de votre choix :

```
$ pip3 install --user --upgrade virtualenv
$ mkdir $HOME/ml
$ cd $HOME/ml
$ virtualenv -p `which python3` env
```

Nous utilisons dans ce livre le répertoire de travail `$HOME/ml` mais vous pouvez le remplacer par tout autre répertoire de votre choix.

Désormais, lorsque vous souhaiterez utiliser cet environnement, il vous suffira d'ouvrir un terminal et de taper les commandes suivantes :

```
$ cd $HOME/ml
$ source env/bin/activate
```

Voilà ! Que vous ayez choisi de créer un environnement Python isolé ou non, vous êtes maintenant prêt(e) à installer TensorFlow. Pour cela, il suffit de lancer la commande suivante (attention : l'option `--user` doit être omise si vous utilisez un environnement isolé, et aucun droit d'accès particulier ne sera alors nécessaire) :

```
$ pip3 install --user --upgrade tensorflow
```

Si vous disposez d'un processeur GPU compatible avec TensorFlow, vous pouvez remplacer `tensorflow` par `tensorflow-gpu` (voir le chapitre 5 pour plus de détails).

Maintenant, vérifions que TensorFlow est bien installé :

```
$ python3 -c 'import tensorflow as tf; print(tf.__version__)'
1.3.0
```

Si la version de TensorFlow s'affiche (par exemple 1.3.0), alors bravo ! TensorFlow est correctement installé.

1.2 INSTALLER LE PROJET HANDSON-ML

Tous les exemples de code de ce livre, ainsi que du livre *Machine Learning avec Scikit-Learn*, sont disponibles sur GitHub dans le projet open source suivant :
https://github.com/ageron/handson-ml

Ce projet contient un notebook (bloc-note) Jupyter pour chaque chapitre. Jupyter est une interface web permettant d'exécuter du code Python (ou autres langages) de façon graphique et interactive. Les notebooks 1 à 8 correspondent aux chapitres 1 à 8 du livre *Machine Learning avec Scikit-Learn*, tandis que les notebooks 9 à 16 correspondent aux chapitres 2 à 9 du livre que vous tenez entre les mains (en effet, ces deux livres n'en font qu'un dans la version originale). Bien que ce soit facultatif, vous êtes fortement encouragé(e) à installer ce projet, à jouer avec le code, à faire les exercises pratiques et à regarder de près les corrections qui figurent à la fin de

chaque notebook : le Machine Learning et le Deep Learning s'apprennent surtout par la pratique !

Pour installer ce projet, il vous faudra le logiciel git. S'il n'est pas déjà présent sur votre système, suivez les instructions d'installation disponibles sur *https://git-scm.com/*. Ensuite, clonez le projet sur votre machine, à l'aide des commandes suivantes :

```
$ cd $HOME/ml
$ git clone https://github.com/ageron/handson-ml.git
```

Si vous avez créé un environnement isolé, vous devez maintenant l'activer (*cf.* plus haut). Puis, installez les librairies requises par ce projet. Cela inclut bien sûr Jupyter, ainsi que les librairies scientifiques les plus populaires en Python[5] (NumPy, SciPy, Pandas, Matplotlib), Scikit-Learn (qui met en œuvre de nombreux algorithmes de Machine Learning), et quelques autres.

```
$ cd handson-ml
$ pip3 install --user --upgrade -r requirements.txt
```

Comme expliqué plus haut, vous devez omettre l'option `--user` si vous utilisez un environnement isolé ou si vous souhaitez installer les librairies pour l'ensemble du système (pas juste pour l'utilisateur actuel). Vous pouvez enfin démarrer Jupyter :

```
$ jupyter notebook
```

Cette commande démarre le serveur web de Jupyter, et ouvre votre navigateur Internet sur l'adresse *http://localhost:8888/tree*. Vous devez y voir le contenu du répertoire courant. Il ne vous reste plus qu'à cliquer sur le notebook `04_training_linear_models.ipynb` : il s'agit du notebook correspondant au chapitre actuel (pour les chapitres suivants, rajoutez 7 au numéro du chapitre).

Si vous n'avez jamais utilisé Jupyter, le principe est simple : chaque notebook est constitué d'une liste de cellules. Chacune peut contenir du texte formaté ou du code (Python, dans notre cas). Lorsqu'on exécute une cellule de code, le résultat s'affiche sous la cellule. Cliquez sur le menu Help > User Interface Tour pour un tour rapide de l'interface. Pour vous entraîner, insérez quelques cellules de code au début du notebook, et exécutez quelques commandes Python, telles que `print("Hello world!")` (voir la figure 1.1). Vous pouvez renommer le notebook en cliquant sur son nom (voir la flèche 1 sur la figure). Cliquez dans une cellule de code et saisissez le code à exécuter (flèche 2), puis exécutez le code de la cellule en tapant Shift-Entrée ou en cliquant sur le bouton d'exécution (flèche 3). Lorsque vous cliquez à l'intérieur d'une cellule, vous passez en mode édition (la cellule est alors encadrée en vert). Lorsque vous tapez la touche Echap (Esc) ou que vous cliquez juste à gauche de la cellule, vous passez en mode commande (la cellule est alors encadrée en bleu). Lorsque vous êtes en mode commande, tapez la touche H pour afficher les nombreux raccourcis clavier disponibles.

5. Pour des tutoriels sur NumPy, Pandas et Matplotlib, voir les notebooks correspondants dans le projet.

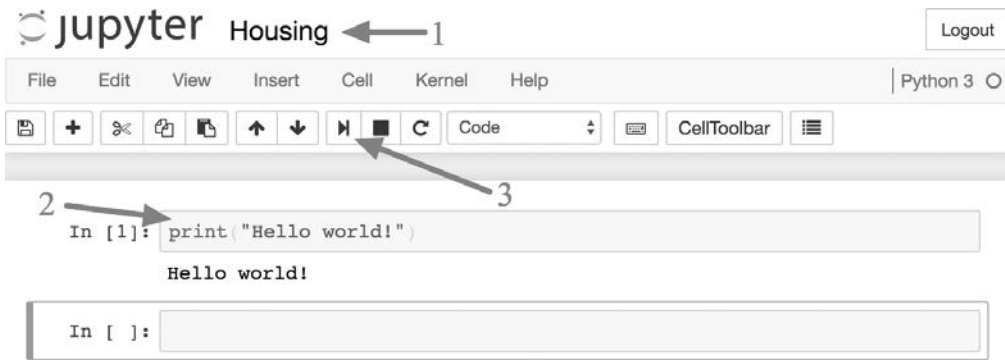

Figure 1.1 – Afficher « Hello world! » dans un notebook Jupyter

Vous pouvez exécuter toutes les cellules du notebook en cliquant sur le menu Cell > Run All.

Vous remarquerez que tous les notebooks du projet commencent par une cellule qui contient (entre autres) le code suivant :

```
from __future__ import division, print_function, unicode_literals
```

Cette ligne permet aux exemples de code de fonctionner indifféremment sous Python 2 ou Python 3.

Par ailleurs, certains commentaires dans les exemples de code contiennent des caractères spéciaux (p. ex. des accents). Cela ne devrait pas poser de problèmes dans Jupyter, car il utilise par défaut l'encodage UTF-8, qui supporte ces caractères, mais si vous recopiez ces exemples de code dans des modules Python 2, vous aurez une erreur car Python 2 utilise par défaut l'encodage ASCII, qui ne supporte que les caractères anglo-saxons. Vous devrez soit retirer ces caractères spéciaux, soit rajouter la ligne suivante au tout début du fichier pour forcer Python 2 à utiliser l'encodage UTF-8 :

```
# -*- coding: utf-8 -*-
```

Parfait ! Vous pouvez désormais exécuter tous les exemples de code de ce livre. Attaquons maintenant les bases du Machine Learning.

1.3 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 la figure 1.2), 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.

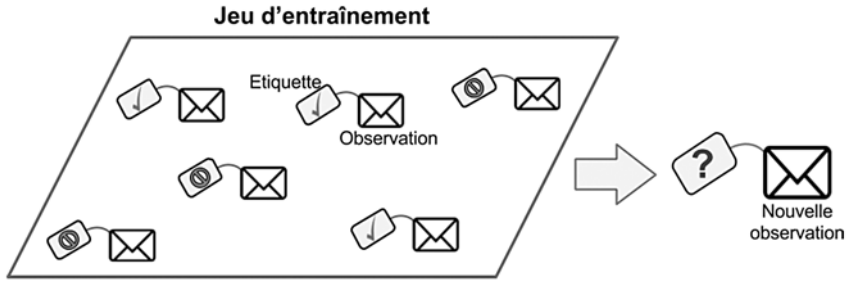

Figure 1.2 – Jeu d'entraînement étiqueté pour une tâche de classification (détection de spam)

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 la figure 1.3). Enfin, la plupart des algorithmes de réduction de la dimensionalité, dont ceux dédiés à la visualisation des données, sont aussi des exemples d'algorithmes d'apprentissage non supervisé.

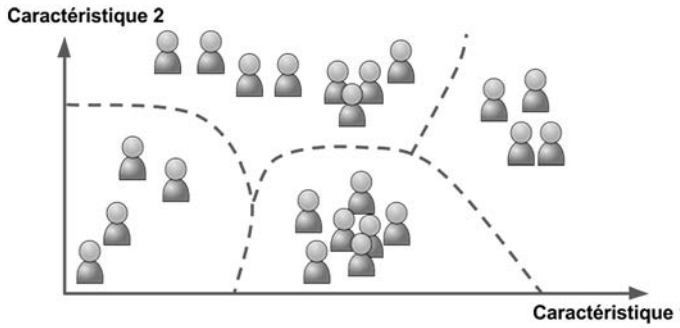

Figure 1.3 – Le partitionnement, un exemple d'apprentissage non supervisé

Résumons : on distingue les tâches d'apprentissage supervisé (classification, régression…), et les tâches d'apprentissage non supervisé (partitionnement, détection d'anomalie, réduction de dimensionalité...). 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. Il existe toutes sortes de variantes de ce schéma général, mais c'est le principe à garder à l'esprit.

1.4 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 minimisent l'erreur du modèle sur le jeu de données d'entraînement.[6]

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 test). 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.

1.5 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*

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

Cette équation introduit des notations que nous emploierons tout au long de ce livre :

- $\hat{y}$ est la valeur prédite ($\hat{y}$ se prononce généralement « y-chapeau »),
- n est le nombre de variables,
- x_i est la valeur de la $i^{ième}$ variable,

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.

- θ_j est le $j^{\text{ième}}$ 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)***

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

- $\boldsymbol{\theta}$ est le *vecteur des paramètres* du modèle, il regroupe à la fois le terme constant θ_0 et les poids θ_1 à θ_n des variables. Notez que, dans le texte, les vecteurs sont représentés en minuscule et en gras, 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 $\mathbf{X}$.
- $\boldsymbol{\theta}^T$ est la transposée de $\boldsymbol{\theta}$. En Machine Learning, comme on manipule beaucoup de matrices, on représente souvent les vecteurs sous la forme de matrices avec une seule colonne. On parle de *vecteur colonne*. Autrement dit, on les représente sous la forme de tableaux à 2 dimensions (p. ex. `[[1],[2],[3]]`) plutôt qu'une seule (p. ex. `[1,2,3]`). La transposée (notée T en exposant) intervertit les deux dimensions : ainsi, un vecteur colonne devient un vecteur ligne (p. ex. `[[1,2,3]]`).
- $\mathbf{x}$ est le *vecteur des valeurs* d'une observation, contenant les valeurs x_0 à x_n, où x_0 est toujours égal à 1.
- $\boldsymbol{\theta}^T \cdot \mathbf{x}$ est le produit matriciel de $\boldsymbol{\theta}^T$ et de $\mathbf{x}$ (ce qui correspondrait au produit scalaire $\langle \boldsymbol{\theta} | \mathbf{x} \rangle$ si $\boldsymbol{\theta}$ et $\mathbf{x}$ étaient des vecteurs simples, et non des vecteurs colonnes). Ce produit est égal à $\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n$. Notez qu'un produit matriciel se note en principe sans le point, par exemple $\boldsymbol{\theta}^T \mathbf{x}$, mais nous préférons employer celui-ci pour être plus explicite.
- $h_{\boldsymbol{\theta}}$ est la fonction hypothèse, utilisant les paramètres de modèle $\boldsymbol{\theta}$ et les variables d'entrée $\mathbf{x}$ pour réaliser une prédiction. Dans le cas de la régression linéaire, il s'agit du simple produit $\boldsymbol{\theta}^T \cdot \mathbf{x}$.

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***

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

- $\hat{\mathbf{y}}$ est le vecteur des prédictions. Son $i^{\text{ième}}$ élément correspond à la prédiction du modèle pour la $i^{\text{ième}}$ observation.
- Plus haut, l'ordre n'importait pas car $\boldsymbol{\theta}^T \cdot \mathbf{x} = \mathbf{x}^T \cdot \boldsymbol{\theta}$, 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)*

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

- 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.

1.5.1 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*[7].

7. Dans le cadre de ce livre, nous ne démontrerons pas que cette fonction renvoie la valeur de $\boldsymbol{\theta}$ qui minimise la fonction de coût.

Équation 1.6 – *Équation normale*

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

- $\hat{\theta}$ est la valeur de θ qui minimise la fonction de coût.
- L'exposant – 1 indique que l'on calcule l'inverse de la matrice $X^T \cdot X$. En théorie, cet inverse n'est pas forcément défini, mais dans la pratique il l'est quasi toujours pourvu qu'il y ait bien davantage d'observations que de variables.
- y est le vecteur des valeurs cibles $y^{(1)}$ à $y^{(m)}$ (une par observation).

Générons maintenant des données à l'allure linéaire sur lesquelles tester cette équation. Nous utiliserons pour cela la librairie NumPy :

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

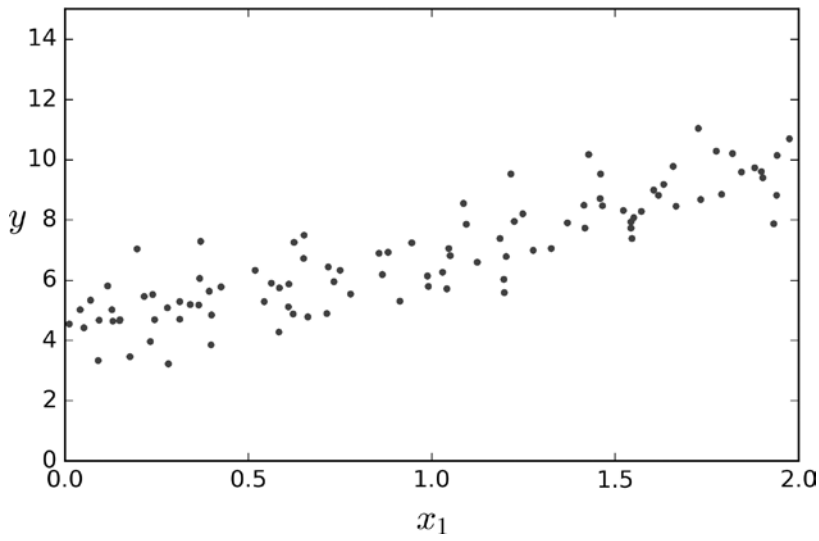

Figure 1.4 – Jeu de données généré aléatoirement

Calculons maintenant $\hat{\theta}$ 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 la méthode `dot()` pour les produits matriciels[8] :

```
X_b = np.c_[np.ones((100, 1)), X] # ajouter x0 = 1 à chaque obs.
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

8. Un opérateur `@` représentant la multiplication matricielle a été introduit à partir de Python 3.5. Il est supporté par NumPy à partir de la version 1.10. Cela permet donc d'écrire `A @ B` plutôt que `A.dot(B)` si `A` et `B` sont des matrices NumPy, ce qui peut rendre le code beaucoup plus lisible.

Nous avons utilisé la fonction $y = 4 + 3x_1 + \textit{bruit gaussien}$ 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.

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

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # ajouter x0 = 1
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Représentons graphiquement les prédictions de ce modèle. Pour cela, nous commençons par la commande suivante, spécifique à Jupyter, qui lui indique que nous souhaitons qu'il affiche les graphiques directement dans le notebook :

```
%matplotlib inline
```

Ensuite, nous importons le module matplotlib et nous affichons les données d'entraînement et le modèle linéaire que nous avons entraîné :

```
import matplotlib
import matplotlib.pyplot as plt
plt.plot(X, y, "b.")
plt.plot(X_new, y_predict, "r-")
plt.axis([0, 2, 0, 15])
plt.show()
```

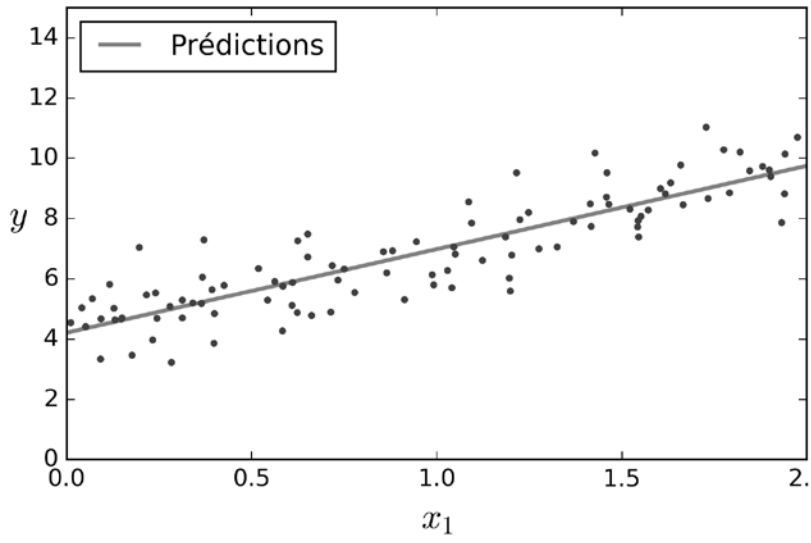

Figure 1.5 – Prédictions du modèle de régression linéaire

Voici le code équivalent lorsqu'on utilise Scikit-Learn[9] :

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression() # création du modèle linéaire
>>> lin_reg.fit(X, y) # entraînement du modèle
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new) # effectuer des prédictions
array([[ 4.21509616],
       [ 9.75532293]])
```

1.5.2 Complexité algorithmique

L'équation normale calcule l'inverse de $\mathbf{X}^T \cdot \mathbf{X}$, qui est une matrice $(n + 1) \times (n + 1)$ (où n est le nombre de variables). La complexité algorithmique d'une inversion de matrice de taille $n \times n$ 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 *grosso modo* multiplié par un facteur compris entre $2^{2,4} = 5,3$ et $2^3 = 8$.

La résolution de l'équation normale prend beaucoup de temps lorsque le nombre de variables devient grand (p. ex. 100 000).

L'aspect positif, c'est que cette équation est linéaire par rapport au nombre d'observations m du jeu d'entraînement (algorithme en $O(m)$), ce qui lui permet de traiter efficacement des jeux de données de grande taille, à condition que ceux-ci puissent tenir 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 à peu près multiplié par deux.

Nous allons maintenant étudier des méthodes d'entraînement de modèle de régression linéaire très différentes, mieux adaptées au cas où il y a beaucoup de variables ou trop d'observations pour tenir en mémoire.

1.6 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 essentielle de la descente de gradient est de corriger petit à petit les paramètres dans le but de minimiser une fonction de coût.

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

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 **θ**, puis progresse en direction du gradient descendant. Lorsque le gradient est nul, vous avez atteint un minimum !

En pratique, vous commencez par remplir **θ** 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 l'algorithme converge vers un minimum (voir la figure 1.6).

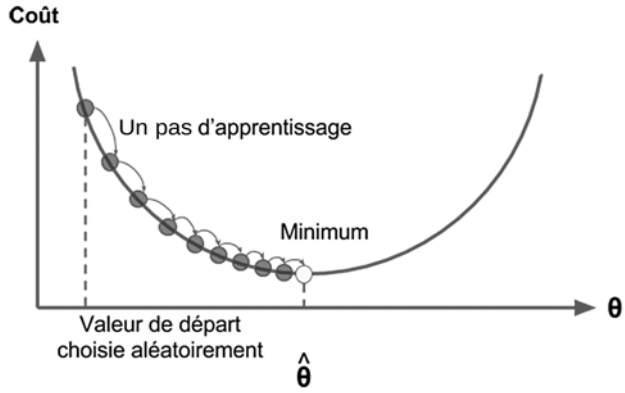

Figure 1.6 – Descente de gradient

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 le *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 la figure 1.7).

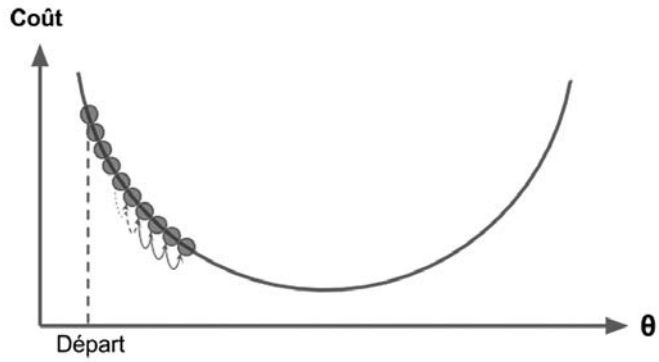

Figure 1.7 – Taux d'apprentissage trop petit

Inversement, si le taux d'apprentissage est trop élevé, vous risquez de dépasser le point le plus bas et de vous retrouver de l'autre côté, peut-être même plus haut qu'avant. Ceci pourrait faire diverger l'algorithme, avec des valeurs de plus en plus grandes, ce qui empêcherait de trouver une bonne solution (voir la figure 1.8).

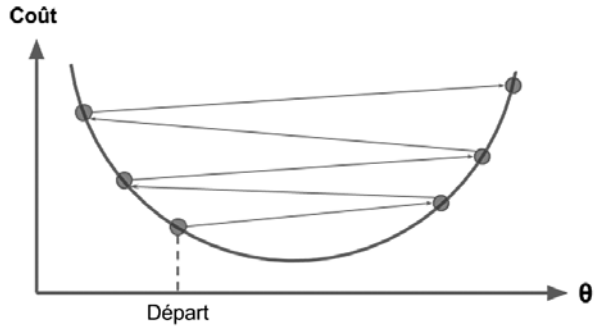

Figure 1.8 – Taux d'apprentissage trop élevé

Enfin, toutes les fonctions de coût n'ont pas la forme d'une jolie cuvette régulière. Il peut y avoir des trous, des crêtes, des plateaux et toutes sortes de terrains irréguliers, ce qui complique la convergence vers le minimum. La figure 1.9 illustre les deux principaux pièges de la descente de gradient : si l'initialisation aléatoire démarre l'algorithme sur la gauche, alors l'algorithme convergera vers un *minimum local*, qui n'est pas le *minimum global*. Si l'initialisation commence sur la droite, alors il lui faudra très longtemps pour traverser le plateau ; si l'algorithme est arrêté prématurément, vous n'atteindrez jamais le minimum global.

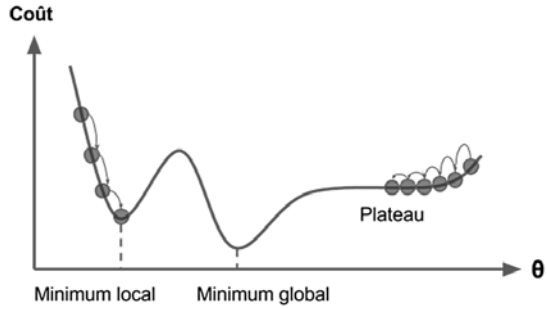

Figure 1.9 – Pièges de la descente de gradient

Heureusement, la fonction de coût MSE appliquée au modèle de régression linéaire est par chance une *fonction convexe*, ce qui signifie que si vous prenez deux points quelconques de la courbe, le segment de droite les joignant ne coupe jamais la courbe. Cela implique qu'il n'y a pas de minima locaux, mais juste un minimum global. C'est aussi une fonction continue dont la pente ne varie jamais abruptement[10]. Ces deux faits ont une conséquence importante : il est garanti que la descente de gradient s'approchera aussi près que l'on veut du minimum global (si vous attendez suffisamment longtemps et si le taux d'apprentissage n'est pas trop élevé).

En fait, la fonction de coût a la forme d'un bol, mais il peut s'agir d'un bol déformé si les variables ont des échelles très différentes. La figure 1.10 présente deux descentes de gradient, l'une (à gauche) sur un jeu d'entraînement où les variables 1 et 2 ont la même échelle, l'autre (à droite) sur un jeu d'entraînement où la variable 1 a des valeurs beaucoup plus petites que la variable 2.[11]

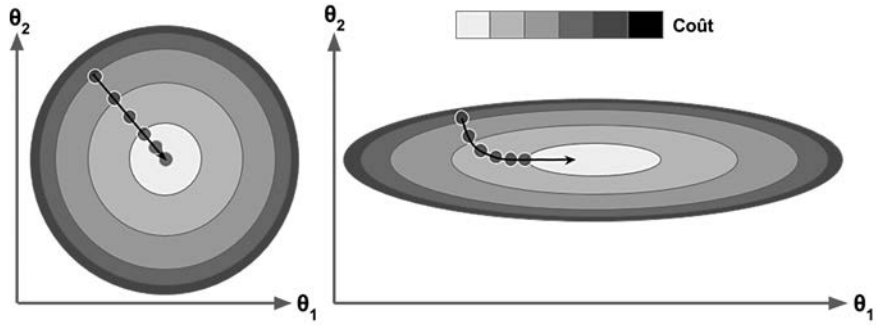

Figure 1.10 – Descente de gradient avec et sans normalisation des variables

10. Techniquement parlant, sa dérivée est *lipschitzsienne*, et par conséquent uniformément continue.
11. La variable 1 étant plus petite, il faut une variation plus importante de θ_1 pour affecter la fonction de coût, c'est pourquoi la cuvette s'allonge le long de l'axe θ_1.