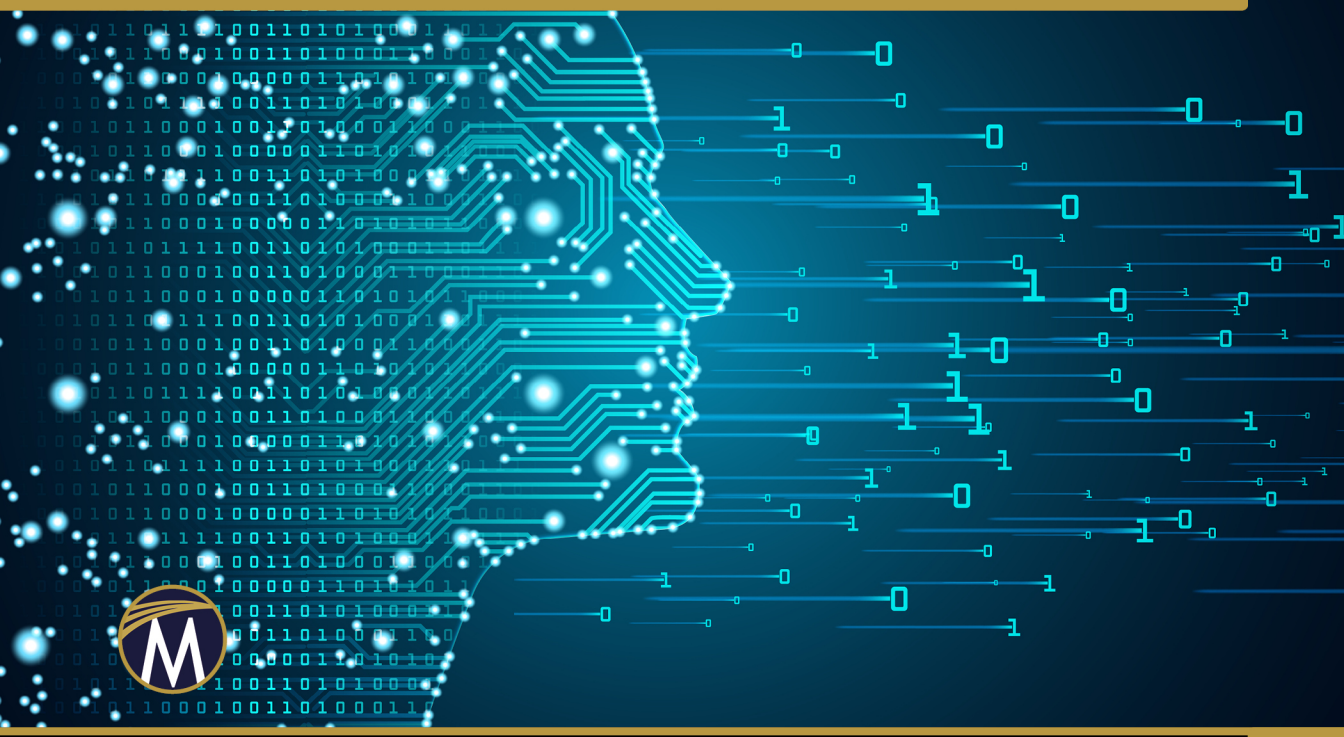
Artificial Intelligence Machine Learning and Deep Learning



O. Campesato

Artificial Intelligence Machine Learning and Deep Learning

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and its companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information, files, or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, production, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Companion files also available for downloading from the publisher by writing to info@merclearning.com.

Artificial Intelligence Machine Learning and Deep Learning

Oswald Campesato

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
1-800-232-0223

O. Campesato. *Artificial Intelligence, Machine Learning and Deep Learning.*
ISBN: 978-1-68392-467-8

Library of Congress Control Number: 2019957226

202122321 Printed on acid-free paper in the United States of America.

I'd like to dedicate this book to my parents –
may this bring joy and happiness into their lives.

CONTENTS

PREFACE: THE ML AND DL LANDSCAPE

What Is the Goal?

The goal of this book is to introduce advanced beginners to basic machine learning and deep learning concepts and algorithms. It is intended to be a fast-paced introduction to various "core" features of machine learning and deep learning, with code samples that are included in a university course. The material in the chapters illustrates how to solve some tasks using Keras, after which you can do further reading to deepen your knowledge.

This book will also save you the time required to search for code samples, which is a potentially time-consuming process. In any case, if you're not sure whether or not you can absorb the material presented here, then glance through the code samples to get a feel for the level of complexity.

At the risk of stating the obvious, please keep in mind the following point: *you will not become an expert in machine learning or deep learning by reading this book.*

What Will I Learn from This Book?

The first chapter contains a very short introduction to AI, followed by a chapter devoted to Pandas for managing the contents of datasets. The third chapter introduces you to machine learning concepts (supervised and unsupervised learning), types of tasks (regression, classification, and clustering),

and linear regression (the second half of the chapter). The fourth chapter is devoted to classification algorithms, such as kNN, Naïve Bayes, decision trees, random forests, and SVM (Support Vector Machines).

The fifth chapter introduces deep learning and delves into CNNs (Convolutional Neural Networks). The sixth chapter covers deep learning architectures such as RNNs (recurrent neural networks) and LSTMs (Long Short Term Memory).

The sixth chapter introduces you to aspects of NLP (Natural Language Processing, with some basic concepts and algorithms, followed by RL (Reinforcement Learning) and the Bellman equation. The first appendix covers `Keras`, whereas the second appendix covers TensorFlow 2.0.

Another point: although Jupyter is popular, all the code samples in this book are Python scripts. However, you can quickly learn the useful features of Jupyter through various online tutorials. In addition, it's worth looking at Google *Colaboratory* that is entirely online and is based on Jupyter notebooks, along with free GPU usage.

How Much `Keras` Knowledge Is Needed for this Book?

Some exposure to `Keras` is helpful, and you can read the appendix if `Keras` is new to you. If you also want to learn about `Keras` and logistic regression, there is an example in Chapter 3. This example requires some theoretical knowledge involving activation functions, optimizers, and cost functions, all of which are discussed in Chapter 4.

Please keep in mind that `Keras` is well-integrated into TensorFlow 2 (in the `tf.keras` namespace), and it provides a layer of abstraction over "pure" TensorFlow that will enable you to develop prototypes more quickly.

Do I Need to Learn the Theory Portions of this Book?

Once again, the answer depends on the extent to which you plan to become involved in machine learning. In addition to creating a model, you will use various algorithms to see which ones provide the level of accuracy (or some other metric) that you need for your project. If you fall short, the theoretical aspects of machine learning can help you perform a "forensic" analysis of your model and your data, and ideally assist in determining how to improve your model.

How Were the Code Samples Created?

The code samples in this book were created and tested using Python 3 and Keras that's built into TensorFlow 2 on a MacBook Pro with OS X 10.12.6 (MacOS Sierra). Regarding their content: the code samples are derived primarily from the author for his *Deep Learning and Keras* graduate course. In some cases there are code samples that incorporate short sections of code from discussions in online forums. The key point to remember is that the code samples follow the "Four Cs": they must be Clear, Concise, Complete, and Correct to the extent that it's possible to do so, given the size of this book.

What Are the Technical Prerequisites for This Book?

You need some familiarity with Python, and also know how to launch Python code from the command line (in a Unix-like environment for Mac users). In addition, a mixture of basic linear algebra (vectors and matrices), probability/statistics, (mean, median, standard deviation) and basic concepts in calculus (such as derivatives) will help you master the material. Some knowledge of `NumPy` and `Matplotlib` is also helpful, and the assumption is that you are familiar with basic functionality (such as `NumPy` arrays).

One other prerequisite is important for understanding the code samples in the second half of this book: some familiarity with neural networks, which includes the concept of hidden layers and activation functions (even if you don't fully understand them). Knowledge of cross entropy is also helpful for some of the code samples.

What Are the Non-technical Prerequisites for This Book?

Although the answer to this question is more difficult to quantify, it's very important to have a strong desire to learn about machine learning, along with the motivation and discipline to read and understand the code samples.

Even simple machine language APIs can be a challenge to understand them at first encounter, so be prepared to read the code samples several times.

How Do I Set up a Command Shell?

If you are a Mac user, there are three ways to do so. The first method is to use `Finder` to navigate to `Applications > Utilities` and then double click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open/Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a MacBook from a command shell that is already visible simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install Cygwin (open source *https://cygwin.com/*) that simulates bash commands, or use another toolkit such as MKS (a commercial product). Please read the online documentation that describes the download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

Companion Files

All of the code samples and figures in this book may be obtained for download by writing to the publisher at info@merclearning.com.

What Are the "Next Steps" after Finishing this Book?

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. The best answer is to try a new tool or technique from the book out on a problem or task you care about, professionally or personally. Precisely what that might be depends on who you are, as the needs of a data scientist, manager, student or developer are all different. In addition, keep what you learned in mind as you tackle new challenges.

O. Campesato
San Francisco, CA

CHAPTER 1

INTRODUCTION TO AI

This chapter provides a gentle introduction to AI, primarily as a broad overview of this diverse topic. Unlike the other chapters in this book, this introductory chapter is chapter is "light" in terms of technical content. However, it's easy to read and also worth skimming through its contents. Machine learning and deep learning are briefly introduced toward the end of this chapter, both of which are discussed in more detail in subsequent chapters.

Keep in mind that many AI-focused books tend to discuss AI from the perspective of computer science and a discussion of traditional algorithms and data structures. By contrast, this book treats AI as an "umbrella" for machine learning and deep learning, and therefore it's discussed in a cursory manner as a precursor to the other chapters.

The first part of this chapter starts with a discussion regarding the term *artificial intelligence*, various potential ways to determine the presence of intelligence, as well as the difference between Strong AI and Weak AI. You will also learn about the Turing Test, which is a well-known test for intelligence.

The second part of this chapter discusses some AI uses-cases and the early approaches to neural computing, evolutionary computation, NLP, and bioinformatics.

The third part of this chapter introduces you to major subfields of AI, which include natural language processing (with NLU and NLG), machine learning, deep learning, reinforcement learning, and deep reinforcement learning.

Although code-specific samples are not discussed in this chapter, the companion files for this chapter do contain a Java-based code sample for solving the Red Donkey problem, and also a Python-based code sample (that requires Python 2.x) for solving Rubik's Cube.

What Is Artificial Intelligence?

The literal meaning of the word *artificial* is synthetic, which often has a negative connotation of being an inferior substitute. However, artificial objects (e.g., flowers) can closely approximate their counterparts, and sometimes they can be advantageous when they do not have any maintenance requirements (sunshine, water, and so forth).

By contrast, a definition for *intelligence* is more elusive than a definition of the word *artificial*. R. Sternberg, in a text on human consciousness, provides the following useful definition: "Intelligence is the cognitive ability of an individual to learn from experience, to reason well, to remember important information, and to cope with the demands of daily living."

You probably remember standardized tests with questions that ask for the next number in a given sequence, such as 1, 3, 6, 10, 15, 21. The first thing to observe is that the gap between successive numbers increases by one: from 1 to 3, the increase is two, whereas from 3 to 6, it is three, and so on. Based on this pattern, the plausible response is 28. Such questions are designed to measure our proficiency at identifying salient features in patterns.

Incidentally, there can be multiple answers to a "next-in-sequence" numeric problem. For example, the sequence 2, 4, 8 might suggest 16 as the next number in this sequence, which is correct if the generating formula is `2^n`. However, if the generating formula is `2^n + (n-1)*(n-2)*(n-3)`, then the next number in the sequence is 22 (not 16). There are many formulas that can match 2, 4, and 8 as the initial sequence of numbers, and yet the next number can be different from 16 or 22.

Let's return to R. Sternberg's definition for intelligence, and consider the following questions:

- How do you decide if someone (something?) is intelligent?
- Are animals intelligent?
- If animals are intelligent, how do you measure their intelligence?

We tend to assess people's intelligence through interaction with them: we ask questions and observe their answers. Although this method is indirect, we often rely on this method to gauge other people's intelligence.

In the case of animal intelligence, we also observe their behavior to make an assessment. Clever Hans was a famous horse that lived in Berlin, Germany, circa 1900, and allegedly had a proficiency in arithmetic, such as adding numbers and calculating square roots.

In reality, Hans was able to identify human emotions and, in conjunction with his astute hearing, he could sense the reaction of audience members as Hans came closer to a correct answer. Interestingly, Hans performed poorly without the presence of an audience. You might be reluctant to attribute Clever Hans's actions to intelligence; however, review Sternberg's definition before reaching a conclusion.

As another example, some creatures exhibit intelligence only in groups. Although ants are simple insects, and their isolated behavior would hardly warrant inclusion in a text on AI, ant colonies exhibit extraordinary solutions to complex problems. In fact, ants can figure out the optimal route from a nest to a food source, how to carry heavy objects, and how to form bridges. Thus, a *collective* intelligence arises from effective communication among individual insects.

The ratios of brain mass and brain-to-body mass are indicators of intelligence, and dolphins compare favorably with humans in both metrics. Breathing in dolphins is under voluntary control, which could account for excess brain mass, as well as the fact that alternate halves of a dolphin's brain take turns sleeping. Dolphins score well on animal self-awareness tests such as the mirror test, in which they recognize that the image in the mirror is actually their own image. They can also perform complex tricks, as visitors to Sea World can testify. This illustrates the ability of dolphins to remember and perform complex sequences of physical motions.

The use of tools is another litmus test for intelligence and is often used to separate *homo erectus* from earlier ancestors of human beings. Dolphins also share this trait with humans: dolphins use deep-sea sponges to protect their spouts while foraging for food. Thus, intelligence is not an attribute possessed by humans alone. Many living forms possess some degree of intelligence.

Now consider the following question: can inanimate objects, such as computers, possess intelligence? The declared goal of artificial Intelligence is to create computer software and/or hardware systems that exhibit thinking comparable to that of humans, in other words, to display characteristics usually associated with human intelligence.

What about the capacity to think, and can machines think? Keep in mind the distinction between thinking and intelligence. Thinking is the facility to reason, analyze, evaluate, and formulate ideas and concepts. Therefore, not every being capable of thinking is intelligent. Intelligence is perhaps akin to efficient and effective thinking.

Many people approach this issue with biases, saying that computers are made of silicon and power supplies and therefore are not capable of thinking. At the other extreme, computers perform much faster than humans and therefore must be more intelligent than humans. The truth is most likely somewhere between these two extremes. As we have discussed, different animal species possess intelligence to varying degrees. However, we are more interested in a test to ascertain the existence of machine intelligence than in developing standardized IQ tests for animals. Perhaps Raphael put it best: artificial intelligence is the science of making machines do things that would require intelligence if done by man.

Strong AI versus Weak AI

Currently there are two main camp regarding AI. The *weak AI* approach is associated with the Massachusetts Institute of Technology, and it views any system that exhibits intelligent behavior as an example of AI. This camp focuses on whether a program performs correctly, regardless of whether the artifact performs its task in the same way humans do. The results of AI projects in electrical engineering, robotics, and related fields are primarily concerned with satisfactory performance.

The other approach to AI is called *biological plausibility*, and it's associated with Carnegie-Mellon University. According to this approach, when an artifact exhibits intelligent behavior, its performance should be based upon the same methodologies used by humans. For instance, consider a system capable of hearing: proponents of strong AI might aim to achieve success by simulating the human hearing system, whereas weak AI proponents would be concerned merely with the system's performance. This simulation would include the equivalents to cochlea, hearing canal, eardrum, and other parts of the ear, each performing its required tasks in the system.

Hence, proponents of weak AI measure the success of the systems that they build based on their performance alone. They maintain that the *raison d'etre* of AI research is to solve difficult problems regardless of how they are actually solved.

On the other hand, proponents of strong AI are concerned with the structure of the systems they build. They maintain that by sheer dint of possessing heuristics, algorithms, and knowledge of AI programs, computers can possess a sense of consciousness and intelligence. As you know, Hollywood has produced various movies (e.g., *I, Robot* and *Blade Runner*) that belong to the strong AI camp.

The Turing Test

The previous section posed three questions, and the first two questions have already been addressed: how do you determine intelligence, and are animals intelligent? The answer to the second question is not necessarily yes or no. Some people are smarter than others and some animals are smarter than others. The question of machine intelligence is equally problematic.

Alan Turing sought to answer the question of intelligence in operational terms. He wanted to separate functionality (what something does) from implementation (how something is built). He devised something that's called the *Turing Test*, which is discussed in the next section.

Definition of the Turing Test

Alan Turing proposed two imitation games, in which one person or entity behaves as if he were another. In the first game, a person (called an interrogator) is in a room with a curtain that runs across the center of the room. On the other side of the curtain is a person, and the interrogator must determine whether it is a man or a woman. The interrogator (whose gender is irrelevant) accomplishes this task by asking a series of questions.

This game assumes that the man will perhaps lie in his responses, but the woman is always truthful. In order that the interrogator cannot determine gender from voice, communication is via computer rather than through spoken words. If it is a man on the other side of the curtain, and he is successful in deceiving the interrogator, then he wins the imitation game.

In Turing's original format for this test, both a man and a woman were seated behind a curtain and the interrogator had to identify both correctly.

Turing might have based this test on a game that was popular during this period, which may even have been the impetus behind his machine intelligence test.

Additional interesting updates regarding the Turing test are discussed in these two links:

https://futurism.com/the-byte/scientists-invented-new-turing-test

https://theconversation.com/our-turing-test-for-androids-will-judge-how-lifelike-humanoid-robots-can-be-120696

In case you didn't already know, Erich Fromm was a well-known sociologist and psychoanalyst in the twentieth century who believed that men and women are equal but not necessarily the same. For instance, the genders might differ in their knowledge of colors, flowers, or the amount of time spent shopping. What does distinguishing a man from a woman have to do with the question of intelligence? Turing understood that there might be different types of thinking, and it is important to both understand these differences and to be tolerant of them.

An Interrogator Test

This second game is more appropriate to the study of AI. Once again, an interrogator is in a room with a curtain. This time, a computer or a person is behind the curtain, and the machine plays the role of the male and could also find it convenient on occasion to lie.

The person, on the other hand, is consistently truthful. The interrogator asks questions and then evaluates the responses to determine whether she is communicating with a person or a machine. If the computer is successful in deceiving the interrogator, it passes the Turing Test and is thereby considered intelligent.

Heuristics

Heuristics can be very useful, and AI applications often rely on the application of heuristics. A *heuristic* is essentially a "rule of thumb" for solving a problem. In other words, a heuristic is a set of guidelines that often works to solve a problem. Contrast a heuristic with an algorithm, which is a prescribed set of rules to solve a problem and whose output is entirely predictable.

A heuristic is a technique for finding an approximate solution that can be used when other methods are too time-consuming or too complex (or both). With a heuristic, a favorable outcome is likely but not guaranteed, and heuristic methods were especially popular in the early days of AI.

Various heuristics appear in daily life. For example, many people prefer using heuristics instead of asking for driving directions. For instance, when exiting a highway at night, sometimes it's difficult to find the route back to the main thoroughfare. One heuristic that could prove helpful is to proceed in the direction with more streetlights whenever they come to a fork in the road. You might have a favorite ploy for recovering a dropped contact lens or for finding a parking space in a crowded shopping mall. Both are examples of heuristics.

AI problems tend to be large and computationally complex, and frequently they cannot be solved via straightforward algorithms. AI problems and their domains tend to embody a large amount of human expertise, especially if tackled by strong AI methods. Some types of problems are better solved using AI, whereas others are more suitable for traditional computer science approaches involving simple decision-making or exact computations to produce solutions. Let us consider a few examples:

- Medical diagnosis
- Shopping using a cash register with barcode scanning
- ATMs
- Two-person games such as chess and checkers

Medical diagnosis is a field of science that has benefited for many years from AI-based contributions, particularly through the development of expert systems. Expert systems are typically built in domains where there is considerable human expertise and where there exist many rules that are often of the form: if-condition-then-action. As a trivial example: if you have a headache, then take two aspirins and call me in the morning.

In particular, expert systems became very popular (and very useful) because they can store far more rules than humans can hold in their head. Expert systems are among the most successful AI techniques for producing results that are comprehensive and effective. In fact, expert systems can help humans make more accurate decisions (and even "challenge" incorrect choices).

Genetic Algorithms

One promising paradigm is Darwin's theory of evolution, which involves natural selection that occurs in nature at a rate of thousands or millions of years. By contrast, evolution inside a computer proceeds much faster than natural selection.

A genetic algorithm is a heuristic that "mimics" the process of natural selection, which involves selecting the fittest individuals for reproduction to sire the offspring of the subsequent generation.

Let's compare and contrast the use of AI with the process of evolution in the plant and animal world, in which species adapt to their environments through the genetic operators of natural selection, reproduction, mutation, and recombination.

Genetic algorithms (GA) are a specific methodology from the general field known as evolutionary computation, which is that branch of AI wherein proposed solutions to a problem adapt much as animal creatures adapt to their environments in the real world.

In case you're interested, the following link contains some interesting details regarding genetic algorithms:

https://en.wikipedia.org/wiki/Genetic_algorithm

Knowledge Representation

The issue of representation becomes important when we consider AI-related problems. AI systems that acquire and store knowledge in order to process it and produce intelligent results also need the ability to identify and represent that knowledge. The choice of a representation is intrinsic to the nature of problem solving and understanding.

As George Polya (a famous mathematician) remarked, a good representation choice is almost as important as the algorithm or solution plan devised for a particular problem. Good and natural representations facilitate fast and comprehensible solutions.

As an example of a representation choice, consider the well-known Missionaries and Cannibals Problem, where the goal is to transfer three missionaries and three cannibals from the west bank to the east bank of a river with a boat. At any point during the transitions from west to east, you

can see the solution path by selecting an appropriate representation. There are two constraints in this problem: the boat can hold no more than two people at any time and the cannibals on any bank can never outnumber the number of missionaries.

A solution for this problem (as well as the related "jealous husbands" problem) is here:

https://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem#targetText=The%20missionaries%20and%20cannibals%20problem,an%20example%20of%20problem%20representation

Logic-based Solutions

AI researchers have used a logic-based approach for knowledge representation and problem-solving technique. A seminal example of using logic for this purpose is Terry Winograd's Blocks World (1972), in which a robot arm interacts with blocks on a tabletop. This program encompassed issues of language understanding and scene analysis as well as other aspects of AI.

In addition, production rules and production systems are used to construct many successful expert systems. The appeal of production rules and expert systems is based on the feasibility of representing heuristics clearly and concisely. Thousands of expert systems have been built incorporating this methodology.

Semantic Networks

Semantic networks are another graphical, though complex, representation of knowledge. Semantic networks precede object-oriented languages, which use inheritance (wherein an object from a particular class inherits many of the properties of a superclass).

Much of the work employing semantic networks has focused on representing the knowledge and structure of language. Examples include Stuart Shapiro SNePS (Semantic Net Processing System) and the work of Roger Schank in natural language processing.

Additional alternatives exist for knowledge representation: graphical approaches offer greater appeal to the senses, such as vision, space, and motion. Possibly the earliest graphical approaches were state-space representations, which display all the possible states of a system.

AI and Games

Since the middle of the twentieth century and the advent of computers, significant progress in computer science and proficiency in programming techniques was acquired through the challenges of training computers to play and master complex board games. Some examples of games whose play by computer have benefitted from the application of AI insights and methodologies have included chess, checkers, Go, and Othello.

Games have spurred the development and interest in AI. Early efforts were highlighted by the efforts of Arthur Samuel in 1959 on the game of checkers. His program was based on tables of fifty heuristics and was used to play against different versions of itself. The losing program in a series of matches would adopt the heuristics of the winning program. It played strong checkers, but never mastered the game.

People have been trying to train machines to play strong chess for several centuries. The infatuation with chess machines probably stems from the generally accepted view that it requires intelligence to play chess well.

In 1959, Newell, Simon, and Shaw developed the first real chess program, which followed the Shannon-Turing Paradigm. Richard Greenblatt's program was the first to play club-level chess. Computer chess programs improved steadily in the 1970s until, by the end of that decade, they reached the Expert level (equivalent to the top 1% of chess tournament players).

In 1983, Ken Thompson's Belle was the first program to officially achieve the Master level. This was followed by the success of Hitech, from Carnegie-Mellon University, which successfully accomplished a major milestone as the first Senior Master (over 2400-rated) program. Shortly thereafter the program Deep Thought (also from Carnegie-Mellon) was developed and became the first program capable of beating Grandmasters on a regular basis.

Deep Thought evolved into Deep Blue when IBM took over the project in the 1990s, and Deep Blue played a six-game match with World Champion Garry Kasparov, who saved mankind by winning a match in Philadelphia in 1996. In 1997, however, against Deeper Blue, the successor of Deep Blue, Kasparov lost, and the chess world was shaken.

In subsequent six-game matches against Kasparov, Kramnik, and other World Championship-level players, programs have fared well, but these were not World Championship Matches. Although it is generally agreed that these programs might still be slightly inferior to the best human players, most would be willing to concede that top programs play chess indistinguishably from the most accomplished humans (if one is thinking of the Turing Test).

In 1989, Jonathan Schaeffer, at the University of Alberta in Edmonton, began his long-term goal of conquering the game of checkers with his program Chinook. In a forty-game match in 1992 against longtime Checkers World Champion Marion Tinsley, Chinook lost four, with thirty-four draws. In 1994 their match was tied after six games, when Tinsley had to forfeit because of health reasons. Since that time, Schaeffer and his team have been working to solve checkers from both the end of the game (all eight-pieces and fewer endings) as well as from the beginning.

Other games that use AI techniques include backgammon, poker, bridge, Othello, and Go (often called the new drosophila).

The Success of AlphaZero

Google created AlphaZero, which is an AI-based software program that used self-play to learn how to play games. AlphaZero is the successor to Alpha Go that defeated the world's best human Go player in 2016. AlphaZero easily defeated Alpha Go in the game of Go.

Moreover, after learning the rules of chess, AlphaZero trained itself (again using self-play) and within a single day became the top chess player in the world. AlphaZero can defeat any human chess player as well as any chess-playing computer program.

The really interesting point is that AlphaZero developed its own strategy for playing chess, which not only differs from humans, but also involves chess moves that are considered counterintuitive.

Unfortunately, AlphaZero is unable to tell us how it developed a strategy that is superior to any previously developed approach for playing chess. Since AlphaZero is 100% self-taught and is the top-ranked chess player in the world, does AlphaZero qualify as intelligent?

Expert Systems

Expert systems are one of the areas that have been investigated for almost as long as AI itself has existed. It is one discipline that AI can claim as a great success. Expert systems have many characteristics that make them desirable for AI research and development. These include separation of the knowledge base from the inference engine, being more than the sum of any or all of their experts, relationship of knowledge to search techniques, reasoning, and uncertainty.

One of the earliest and most often referenced systems was heuristic DENDRAL. Its purpose was to identify unknown chemical compounds on the basis of their mass spectrographs. DENDRAL was developed at Stanford University with the goal of performing a chemical analysis of the Martian soil. It was one of the first systems to illustrate the feasibility of encoding domain-expert knowledge in a particular discipline.

Perhaps the most famous expert system is MYCIN, also from Stanford University (1984). Mycin was developed to facilitate the investigation of infectious blood diseases. Even more important than its domain, however, was the example that Mycin established for the design of all subsequent knowledge-based systems. It had over 400 rules, which were eventually used to provide a training dialogue for residents at the Stanford hospital.

In the 1970s, PROSPECTOR (also at Stanford University) was developed for mineral exploration. PROSPECTOR was also an early and valuable example of the use of inference networks.

Other famous and successful systems that followed in the 1970s were XCON (with some 10,000 rules), which was developed to help configure electrical circuit boards on VAX computers; GUIDON, a tutoring system that was an offshoot of Mycin; TEIRESIAS, a knowledge acquisition tool for Mycin; and HEARSAY I and II, the premier examples of speech understanding using the Blackboard Architecture.

The AM (Artificial Mathematician) system of Doug Lenat was another important result of research and development efforts in the 1970s, as well as the Dempster-Schafer Theory for reasoning under uncertainty, together with Zadeh's work in fuzzy logic.

Since the 1980s, thousands of expert systems have been developed in such areas as configuration, diagnosis, instruction, monitoring, planning,

prognosis, remedy, and control. Today, in addition to stand-alone expert systems, many expert systems have been embedded into other software systems for control purposes, including those in medical equipment and automobiles (for example, when should traction control engage in an automobile?).

In addition, many expert systems shells, such as Emycin, OPS, EXSYS, and CLIPS, have become industry standards. Many knowledge representation languages have also been developed. Today, numerous expert systems work behind the scenes to enhance day-to-day experiences, such as the online shopping cart.

Neural Computing

McCulloch and Pitts conducted early research in neural computing because they were trying to understand the behavior of animal nervous systems. Their model of artificial neural networks (ANN) had one serious drawback: it did not include a mechanism for learning.

Frank Rosenblatt developed an iterative algorithm known as the Perceptron Learning Rule for finding the appropriate weights in a single-layered network (a network in which all neurons are directly connected to inputs). Research in this burgeoning discipline might have been severely hindered by the pronouncement by Minsky and Papert that certain problems could not be solved by single-layer perceptrons, such as the exclusive OR (XOR) function. Federal funding for neural network research was severely curtailed immediately after this proclamation.

The field witnessed a second flurry of activity in the early 1980s with the work of Hopfield. His asynchronous network model (Hopfield networks) used an energy function to approximate solutions to NP-complete problems.

The mid-1980s also witnessed the discovery of back propagation (usually called *backprop*), a learning algorithm appropriate for multilayered networks. Back propagation-based networks are routinely employed to predict Dow Jones averages and to read printed material in optical character recognition systems.

Neural networks are also used in control systems. ALVINN was a project at Carnegie Mellon University in which a back propagation network senses the highway and assists in the steering of a Navlab vehicle.

One immediate application of this work was to warn a driver impaired by lack of sleep, excess of alcohol, or other conditions whenever the vehicle strayed from its highway lane. Looking toward the future, it is hoped that, someday, similar systems will drive vehicles so that we are free to read newspapers and talk on our cell phones to take advantage of the extra free time.

Evolutionary Computation

Genetic algorithms are more generally classified as evolutionary computation. Genetic algorithms use probability and parallelism to solve combinatorial problems (also called optimization problems), which is an approach developed by John Holland.

However, evolutionary computation is not solely concerned with optimization problems. Rodney Brooks was formerly the director of the MIT Computer Science and AI Laboratory. His approach to the successful creation of a human-level Artificial Intelligence, which he aptly cites as the holy grail of AI research, renounces reliance on the symbol-based approach. This latter approach relies upon the use of heuristics and representational paradigms.

In his view, intelligent systems can be designed in multiple layers in which higher leveled layers rely upon those layers beneath them. For example, if you wanted to build a robot capable of avoiding obstacles, the obstacle avoidance routine would be built upon a lower layer, which would merely be responsible for robotic locomotion.

Brooks maintains that intelligence emerges through the interaction of an agent with its environment. He is perhaps most well known for the insectlike robots built in his lab that embody this philosophy of intelligence, wherein a community of autonomous robots interact with their environment and with each other.

Natural Language Processing

If we wish to build intelligent systems, it seems natural to ask that our systems possess a language-understanding facility. This is an axiom that was well understood by many early practitioners. Eliza is one well-known early application program, which was developed by Joseph Weizenbaum, an MIT

computer scientist who worked with Kenneth Colby (a Stanford University psychiatrist).

Eliza was intended to imitate the role played by a psychiatrist of the Carl Rogers School. For instance, if the user typed in "I feel tired," Eliza was a back propagation application that learned the correct pronunciation for English text. It was claimed to pronounce English sounds with 95% accuracy. Obviously, problems arose because of inconsistencies inherent in the pronunciation of English words, such as *rough* and *through*, and the pronunciation of words derived from other languages, such as *pizza* and *fizzy*.

Terry Winograd wrote another well-known program that was named after the second set of these letters of the pair ETAOIN SHRDLU, which are the most frequently used letters in the English language on linotype machines. Winograd's program might respond with, "You say you feel tired. Tell me more." The "conversation" would continue in this manner, with the machine contributing little or nothing in terms of originality to the dialogue. A live psychoanalyst might behave in this fashion in the hope that the patient would discover their true (perhaps hidden) feelings and frustrations. Meanwhile, Eliza is merely using pattern matching to feign humanlike interaction.

Curiously, Weizenbaum was disturbed by the avid interest that his students (and the public in general) took in interacting with Eliza, even though they were fully aware that Eliza was only a program. Meanwhile, Colby remained dedicated to the project and went on to author a successful program called DOCTOR.

Although Eliza has contributed little to natural language processing (NLP), it is software that pretends to possess what is perhaps our last vestige of specialness, our ability to feel emotions. What will happen when the line between a human and machine (example: android) becomes less clear, perhaps in some fifty years, and these androids will be less mortal and more like immortals?

More recently, several MIT robots, including Cog, Kismet, and Paro, have been developed with the uncanny ability to feign human emotions and evoke emotional responses from those with whom they interact. Turkle has studied the relationships that children and older persons in nursing homes have formed with these robots; relationships that involve genuine emotion and caring. Turkle speaks of the need to

perhaps redefine the word *relationship* to include the encounters that people have with these so-called relational artifacts. She remains confident, however, that such relationships will never replace the bonds that can only occur between human beings who must confront their mortality on a daily basis.

Winograd's Blocks World involved a robot arm that was able to achieve various goals. For example, if SHRDLU was asked to lift a red block upon which there was a small green block, it knew that it must remove the green block before it could lift the red one. Unlike Eliza, SHRDLU was able to understand English commands and respond to them appropriately.

HEARSAY, an ambitious program in speech recognition, employed a blackboard architecture wherein independent knowledge sources (agents) for various components of language, such as phonetics and phrases, could freely communicate. Both syntax and semantics were used to prune improbable word combinations.

The HWIM (pronounced "whim" and short for Hear What I Mean) Project used augmented transition networks to understand spoken language. It had a vocabulary of 1,000 words dealing with travel budget management. Perhaps this project was too ambitious in scope because it did not perform as well as HEARSAY II.

Parsing played an integral part in the success of these natural language programs. SHRDLU employed a context-free grammar to help parse English commands. Context-free grammars provide a syntactic structure for dealing with strings of symbols. However, to effectively process natural languages, semantics must be considered as well.

A *parse tree* provides the relationship between the words that compose a sentence. For example, many sentences can be broken down into both a subject and a predicate. Subjects can be broken down perhaps into a noun phrase followed by a prepositional phrase and so on. Essentially, a parse tree gives the semantics that is the meaning of the sentence.

Each of the these early language processing systems employed world knowledge to some extent. However, in the late 1980s the greatest stumbling block for progress in NLP was the problem of common sense knowledge. For example, although many successful programs were built in particular areas of NLP and AI, these were often criticized as microworlds, meaning that the programs did not have general, real-world knowledge or common

sense. For example, a program might know a lot about a particular scenario, such as ordering food in a restaurant, but it would have no knowledge of whether the waiter or waitress was alive or whether they would ordinarily be wearing any clothing. During the past twenty-five years, Douglas Lenat of MCC in Austin, Texas, has been building the largest repository of common-sense knowledge to address this issue.

NLP has undergone some interesting developments. After its initial stage (as described earlier in this section), NLP relied on statistics to govern the parse trees for sentences. Charniak describes how context-free grammars (CFGs) can be augmented such that each rule has an associated probability. These associated probabilities could be taken from the Penn Treebank, which contains more than one million words of English text that have been parsed manually, mostly from the *Wall Street Journal*. Charniak demonstrated how this statistical approach successfully obtained a parse for a sentence from the front page of the *New York Times* (no trivial feat, even for most humans).

The next step in the evolution of NLP involves deep-learning architectures called RNNs, LSTMs, and bidirectional LSTMs, which are discussed in Chapter 5. The most recent architecture is called a *transformer*, which was developed by Google in 2017. BERT is based on transformers (as well as "attention") and is one of the most powerful open-source systems currently available for solving NLP tasks. Yet another approach for NLP involves Deep Reinforcement Learning (discussed briefly in Chapter 6).

Bioinformatics

Bioinformatics is the nascent discipline that concerns the application of the algorithms and techniques of computer science to molecular biology. It is mainly concerned with the management and analysis of biological data. In structural genomics, one attempts to specify a structure for each observed protein. Automated discovery and data mining could help in this pursuit.

Juristica and Glasgow demonstrate how case-based reasoning could assist in the discovery of the representative structure for each protein. In their 2004 survey article in the AAAI special issue on *AI and Bioinformatics*, Glasgow, Jurisica, and Rost note: "Possibly the most rapidly growing area of recent activity in bioinformatics is the analysis of microarray data."

Microbiologists are overwhelmed with both the variety and quantity of data available to them. They are being asked to comprehend molecular sequence, structure, and data based solely on huge databases. Many researchers believe that AI techniques from knowledge representation and machine learning will prove beneficial as well.

The next portion of this chapter provides a quick introduction to the major parts of AI, which include machine learning and deep learning.

Major Parts of AI

The subsequent chapters in this book delve into various important parts of AI, which include:

- ML (Machine Learning)
- DL (Deep Learning)
- NLP (Natural Language Processing)
- RL (Reinforcement Learning)
- DRL (Deep Reinforcement Learning)

Traditional AI (twentieth century) is based on collections of rules, which led to expert systems in the 1980s. Traditional AI also involved LISP, which was created by John McCarthy (one of the members of the first official AI meeting in 1956).

Traditional AI is primarily a set of rules in conjunction with conditional logic, which is also true for the powerful expert systems that were developed in the 1980s. However, a rule-based system for making decisions can involve thousands of rules. Even simple objects require many rules: try to come up with a set of rules that define a chair, a table, or even just an apple. Traditional AI has some significant limitations, mainly because of the number of rules that are required.

Machine Learning

Around the middle of the twentieth century machine learning (a subset of AI) relied primarily on data to optimize and "learn" how to perform tasks, often accompanied by new or improved algorithms, such as linear regression, k-NN, decision trees, random forests, and SVMs; with the exception of linear regression, all the other algorithms are classifiers.

As you will see, machine learning is a diverse and vibrant field that includes other subfields.

Since data (instead of rules) is so important in machine learning, it's typically one of the following types:

- Supervised learning (lots of labeled data)
- Semi-supervised learning (lots of partially labeled data)
- Unsupervised learning: lots of data, clustering
- Reinforcement learning: trial, feedback, and improvement

According to Andrew Ng (the cofounder of Coursera), "99% of all machine learning is supervised."

In addition to categorizing data, machine learning algorithms can be categorized into the following major types:

- Classifiers (for images, spam, fraud, etc.)
- Regression (stock price, housing price, etc.)
- Clustering (unsupervised classifiers)

Deep Learning

One important subfield of machine learning is deep learning, which also has its roots in the middle of the twentieth century. Deep-learning architectures rely on the *perceptron* as the basis of neural networks, often involving large or massive datasets. Such architectures also involve heuristics and empirical results. Nowadays deep learning can surpass humans for some image classification.

While machine learning involves MLPs (multilayer perceptrons), deep learning introduces deep neural networks, with new algorithms and new architectures (e.g., convolutional neural networks, RNNs, and LSTMs).

Reinforcement Learning

Reinforcement learning (also a subset of machine learning) involves trial-and-error in order to maximize a reward for a so-called agent. Deep reinforcement learning combines the strengths of deep learning with reinforcement learning. In particular, the agent in reinforcement learning is replaced with a neural network.

Deep reinforcement learning has applications in many diverse fields, and three of the most popular are:

- Games (Go, Chess, etc.)
- Robotics
- NLP

Some well-known and successful examples of the use of reinforcement learning in games include:

- Alpha Go (hybrid RL)
- Alpha Zero (complete RL)
- Often involve Greedy algorithms
- Deep RL: Combines Deep Learning and RL

Robotics

Robots have entered our personal and professional lives in myriad ways, including:

- Surgery (assisting surgeons)
- Radiology (detecting cancer)
- Drug mismanagement
- Comparative theories of religion
- Law/real estate/military/science
- Comedy (including stand-up)
- Music (conducting orchestras)
- Restaurants (gourmet meals)
- Coordinated dancing teams
- Many other fields

Robot truck drivers are displacing jobs, but they also have advantages: their only cost is the upkeep of the machinery. In addition, robots aren't distracted the way that humans are, they don't engage in activities that contribute to accidents, and they don't require salaries or any sort of time off. Yet despite the surprising achievements of robots, *Star Trek*'s character Data is still just a dream.

NLP is an area of computer science and AI that involves interaction between computers and human languages. In the early days, NLP involved rule-based techniques or statistical techniques. NLP and machine learning can process/analyze volumes of natural language data, where computer programs perform that processing.

There are many NLP tasks that are solved with machine learning techniques. Some areas of interest that involve NLP include:

- Translating between languages
- Finding meaningful information from text
- Summarizing documents
- Detecting hate speech

Despite all the advances and advantages of machine learning, et al., there are issues that need to be resolved. One issue is occupational bias: an AI system inferred that white males were doctors and white females were housewives. Another issue involves detecting gender bias. For example, in Wikipedia (circa 2018) 18% of its biographies are of women, while 84% to 90% of Wikipedia editors are male.

Yet another issue, analyzed in the following article, involves data bias versus algorithmic bias:

https://www.forbes.com/sites/charlestowersclark/2018/09/19/can-we-make-artificial-intelligence-accountable

Finally, there is the question of the interaction of AI and ethics, which includes some thought-provoking questions (such as unemployment and robot rights). The following article contains an extensive list of ethical questions:

https://www.weforum.org/agenda/2016/10/top-10-ethical-issues-in-artificial-intelligence/

Code Samples

The companion disc contains the following files:

- `RubiksCube.py`
- `Board.java`
- `Search.java`

The Python file is a solution for Rubik's Cube, and the two Java files are for the solution to the Red Donkey problem.

In order to run a Java program, download the Java Runtime Environment (JRE) here:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

In order to compile and run a Java program, download the Java SDK here:

https://www.java.com/en/

If you do not have Python installed, the Python-related download is here:

http://www.python.org/getit/

If you do not have Java installed, you can find online for instructions doing so, as well as instructions for compiling and launching Java code.

Summary

In this chapter, you learned about AI, strong versus weak AI, and the Turing Test for intelligence. Then you learned about heuristics and their usefulness in algorithms, followed by genetic algorithms, and knowledge representation. Next you saw how AI was initially applied to diverse areas such as games and expert systems.

You also learned about the early approaches to neural computing, evolutionary computation, NLP, and bioinformatics. In addition, you got an introduction to the major subfields of AI, which include natural language processing, machine learning, deep learning, reinforcement learning, and deep reinforcement learning.

CHAPTER 2

INTRODUCTION TO MACHINE LEARNING

This chapter introduces numerous concepts in machine learning, such as feature selection, feature engineering, data cleaning, training sets, and test sets.

The first part of this chapter briefly discusses machine learning and the sequence of steps that are typically required in order to prepare a dataset. These steps include "feature selection" or "feature extraction" that can be performed using various algorithms.

The second section describes the types of data that you can encounter, issues that can arise with the data in datasets, and how to rectify them. You will also learn about the difference between "hold out" and "k-fold" when you perform the training step.

The third part of this chapter briefly discusses the basic concepts involved in linear regression. Although linear regression was developed more than 200 years ago, this technique is still one of the "core" techniques for solving (albeit simple) problems in statistics and machine learning. In fact, the technique known as "Mean Squared Error" (MSE) for finding a best-fitting line for data points in a 2D plane (or a hyperplane for higher dimensions) is implemented in Python and TensorFlow in order to minimize so-called "cost" functions that are discussed later.

The fourth section in this chapter contains additional code samples involving linear regression tasks using standard techniques in `NumPy`. Hence, if you are comfortable with this topic, you can probably skim quickly

through the first two sections of this chapter. The third section shows you how to solve linear regression using `Keras`.

One point to keep in mind is that some algorithms are mentioned without delving into details about them. For instance, the section pertaining to supervised learning contains a list of algorithms that appear later in the chapter in the section that pertains to classification algorithms. The algorithms that are displayed in bold in a list are the algorithms that are of greater interest for this book. In some cases the algorithms are discussed in greater detail in the next chapter; otherwise, you can perform an online search for additional information about the algorithms that are not discussed in detail in this book.

What is Machine Learning?

In high level terms, machine learning is a subset of AI that can solve tasks that are infeasible or too cumbersome with "traditional" programming languages. A spam filter for email is an early example of machine learning. Machine learning generally supersedes the accuracy of older algorithms.

Despite the variety of machine learning algorithms, the data is arguably more important than the selected algorithm. Many issues can arise with data, such as insufficient data, poor quality of data, incorrect data, missing data, irrelevant data, duplicate data values, and so forth. Later in this chapter you will see techniques that address many of these data-related issues.

If you are unfamiliar with machine learning terminology, a dataset is a collection of data values, which can be in the form of a CSV file or a spreadsheet. Each column is called a feature, and each row is a datapoint that contains a set of specific values for each feature. If a dataset contains information about customers, then each row pertains to a specific customer.

Types of Machine Learning

There are three main types of machine learning (combinations of these are also possible) that you will encounter:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning

Supervised learning means that the datapoints in a dataset have a label that identifies its contents. For example, the `MNIST` dataset contains 28x28 PNG files, each of which contains a single hand-drawn digit (i.e. 0 through 9 inclusive). Every image with the digit 0 has the label 0; every image with the digit 1 has the label 1; all other images are labeled according to the digit that is displayed in those images.

As another example, the columns in the Titanic dataset are features about passengers, such as their gender, the cabin class, the price of their ticket, whether or not the passenger survived, and so forth. Each row contains information about a single passenger, including the value 1 if the passenger survived. *The MNIST dataset and the Titanic dataset involve classification tasks: the goal is to train a model based on a training dataset and then predict the class of each row in a test dataset.*

In general, the datasets for classification tasks have a small number of possible values: one of nine digits in the range of 0 through 9, one of four animals (dog, cat, horse, giraffe), one of two values (survived versus perished, purchased versus not purchased). As a rule of thumb, if the number of outcomes can be displayed "reasonably well" in a drop-down list, then it's probably a classification task.

In the case of a dataset that contains real estate data, each row contains information about a specific house, such as the number of bedrooms, the square feet of the house, the number of bathrooms, the price of the house, and so forth. In this dataset the price of the house is the label for each row. Notice that the range of possible prices is too large to fit "reasonably well" in a drop-down list. *A real estate dataset involves a regression task: the goal is to train a model based on a training dataset and then predict the price of each house in a test dataset.*

Unsupervised learning involves unlabeled data, which is typically the case for clustering algorithms (discussed later). Some important unsupervised learning algorithms that involve *clustering* are listed below:

- k-Means
- Hierarchical Cluster Analysis (HCA)
- Expectation Maximization

Some important unsupervised learning algorithms that involve *dimensionality reduction* (discussed in more detail later) are listed below:

- PCA (Principal Component Analysis)
- Kernel PCA
- LLE (Locally Linear Embedding)
- t-SNE (t-distributed Stochastic Neighbor Embedding)

There is one more very important unsupervised task called anomaly detection. This task is relevant for fraud detection and detecting outliers (discussed later in more detail).

Semi-supervised learning is a combination of supervised and unsupervised learning: some datapoints are labeled and some are unlabeled. One technique involves using the labeled data in order to classify (i.e., label) the unlabeled data, after which you can apply a classification algorithm.

Types of Machine Learning Algorithms

There are three main types of machine learning algorithms:

- Regression (ex: linear regression)
- Classification (ex: k-Nearest-Neighbor)
- Clustering (ex: kMeans)

Regression is a supervised learning technique to predict numerical quantities. An example of a regression task is predicting the value of a particular stock. Note that this task is different from predicting whether the value of a particular stock will increase or decrease tomorrow (or some other future time period). Another example of a regression task involves predicting the cost of a house in a real estate dataset. Both of these tasks are examples of a regression task.

Regression algorithms in machine learning include linear regression and generalized linear regression (also called multivariate analysis in traditional statistics).

Classification is also a supervised learning technique, but it's for predicting categorical quantities. An example of a classification task is detecting

the occurrence of spam, fraud, or determining the digit in a PNG file (such as the `MNIST` dataset). In this case, the data is already labeled, so you can compare the prediction with the label that was assigned to the given PNG.

Classification algorithms in machine learning include the following list of algorithms (they are discussed in greater detail in the next chapter):

- Decision Trees (a single tree)
- Random Forests (multiple trees)
- kNN (k Nearest Neighbor)
- Logistic regression (despite its name)
- Naïve Bayes
- SVM (Support Vector Machines)

Some machine learning algorithms (such as SVMs, random forests, and kNN) support regression as well as classification. In the case of SVMs, the scikit-learn implementation of this algorithm provides two APIs: SVC for classification and SVR for regression.

Each of the preceding algorithms involves a model that is trained on a dataset, after which the model is used to make a prediction. By contrast, a random forest consists of *multiple* independent trees (the number is specified by you), and each tree makes a prediction regarding the value of a feature. If the feature is numeric, take the mean or the mode (or perform some other calculation) in order to determine the "final" prediction. If the feature is categorical, use the mode (i.e., the most frequently occurring class) as the result; in the case of a tie you can select one of them in a random fashion.

Incidentally, the following link contains more information regarding the kNN algorithm for classification as well as regression:

http://saedsayad.com/k_nearest_neighbors_reg.htm

Clustering is an unsupervised learning technique for grouping similar data together. Clustering algorithms put data points in different clusters without knowing the nature of the data points. After the data has been separated into different clusters, you can use the SVM (Support Vector Machine) algorithm to perform classification.

Clustering algorithms in machine learning include the following (some of which are variations of each other):

- k-Means
- Meanshift
- Hierarchical Cluster Analysis (HCA)
- Expectation Maximization

Keep in mind the following points. First, the value of `k` in `k-Means` is a hyper parameter, and it's usually an odd number to avoid ties between two classes. Next, the `meanshift` algorithm is a variation of the `k-Means` algorithm that does *not* require you to specify a value for k. In fact, the `meanshift` algorithm determines the optimal number of clusters. However, this algorithm does not scale well for large datasets.

Machine Learning Tasks

Unless you have a dataset that has already been sanitized, you need to examine the data in a dataset to make sure that it's in a suitable condition. The data preparation phase involves 1) examining the rows ("data cleaning") to ensure that they contain valid data (which might require domain-specific knowledge), and 2) examining the columns (feature selection or feature extraction) to determine if you can retain only the most important columns.

A high-level list of the sequence of machine learning tasks (some of which might not be required) is shown below:

- Obtain a dataset
- Data cleaning
- Feature selection
- Dimensionality reduction
- Algorithm selection
- Train-versus-test data
- Training a model
- Testing a model

- Fine-tuning a model
- Obtain metrics for the model

First, you obviously need to obtain a dataset for your task. In the ideal scenario, this dataset already exists; otherwise, you need to cull the data from one or more data sources (e.g., a CSV file, a relational database, a no-SQL database, a Web service, and so forth).

Second, you need to perform *data cleaning*, which you can do via the following techniques:

- Missing Value Ratio
- Low Variance Filter
- High Correlation Filter

In general, data cleaning involves checking the data values in a dataset in order to resolve one or more of the following:

- Fix incorrect values
- Resolve duplicate values
- Resolve missing values
- Decide what to do with outliers

Use the Missing Value Ratio technique if the dataset has too many missing values. In extreme cases, you might be able to drop features with a large number of missing values. Use the Low Variance filter technique to identify and drop features with constant values from the dataset. Use the High Correlation filter technique to find highly correlated features, which increase multicollinearity in the dataset: such features can be removed from a dataset (but check with your domain expert before doing so).

Depending on your background and the nature of the dataset, you might need to work with a domain expert, which is a person who has a deep understanding of the contents of the dataset.

For example, you can use a statistical value (mean, mode, and so forth) to replace incorrect values with suitable values. Duplicate values can be handled in a similar fashion. You can replace missing numeric values with zero, the minimum, the mean, the mode, or the maximum value in a numeric column. You can replace missing categorical values with the mode of the categorical column.

If a row in a dataset contains a value that is an outlier, you have three choices:

- Delete the row
- Keep the row
- Replace the outlier with some other value (mean?)

When a dataset contains an outlier, you need to make a decision based on domain knowledge that is specific to the given dataset.

Suppose that a dataset contains stock-related information. As you know, there was a stock market crash in 1929, which you can view as an outlier. Such an occurrence is rare, but it can contain meaningful information. Incidentally, the source of wealth for some families in the 20th century was based on buying massive amounts of stock are very low prices during the Great Depression.

Feature Engineering, Selection, and Extraction

In addition to creating a dataset and "cleaning" its values, you also need to examine the features in that dataset to determine whether or not you can reduce the dimensionality (i.e., the number of columns) of the dataset. The process for doing so involves three main techniques:

- Feature engineering
- Feature selection
- Feature extraction (aka feature projection)

Feature engineering is the process of determining a new set of features that are based on a combination of existing features in order to create a meaningful dataset for a given task. Domain expertise is often required for this process, even in cases of relatively simple datasets. Feature engineering can be tedious and expensive, and in some cases you might consider using automated feature learning. After you have created a dataset, it's a good idea to perform feature selection or feature extraction (or both) to ensure that you have a high quality dataset.

Feature selection is also called variable selection, attribute selection or variable subset selection. Feature selection involves selecting the subset of relevant features in a dataset. In essence, feature selection involves

selecting the "most important" features in a dataset, which provides these advantages:

- Reduced training time
- Simpler models are easier to interpret
- Avoidance of the curse of dimensionality
- Better generalization due to a reduction in overfitting ("reduction of variance")

Feature selection techniques are often used in domains where there are many features and comparatively few samples (or data points). Keep in mind that a low-value feature can be redundant or irrelevant, which are two different concepts. For instance, a relevant feature might be redundant when it's combined with another strongly correlated feature.

Feature selection can involve three strategies: the filter strategy (e.g. information gain), the wrapper strategy (e.g. search guided by accuracy), and the embedded strategy (prediction errors are used to determine whether features are included or excluded while developing a model). One other interesting point is that feature selection can also be useful for regression as well as classification tasks.

Feature extraction creates new features from functions that produce combinations of the original features. By contrast, feature selection involves determining a subset of the existing features.

Feature selection and feature extraction both result in *dimensionality reduction* for a given dataset, which is the topic of the next section.

Dimensionality Reduction

Dimensionality Reduction refers to algorithms that reduce the number of features in a dataset: hence the term "dimensionality reduction." As you will see, there are many techniques available, and they involve either feature selection or feature extraction.

Algorithms that use feature selection to perform dimensionality reduction are listed here:

- Backward Feature Elimination
- Forward Feature Selection

- Factor Analysis
- Independent Component Analysis

Algorithms that use feature extraction to perform dimensionality reduction are listed here:

- Principal component analysis (PCA)
- Non-negative matrix factorization (NMF)
- Kernel PCA
- Graph-based kernel PCA
- Linear discriminant analysis (LDA)
- Generalized discriminant analysis (GDA)
- Autoencoder

The following algorithms combine feature extraction and dimensionality reduction:

- Principal component analysis (PCA)
- Linear discriminant analysis (LDA)
- Canonical correlation analysis (CCA)
- Non-negative matrix factorization (NMF)

These algorithms can be used during a pre-processing step before using clustering or some other algorithm (such as kNN) on a dataset.

One other group of algorithms involves methods based on projections, which includes t-Distributed Stochastic Neighbor Embedding (t-SNE) as well as UMAP.

This chapter discusses PCA, and you can perform an online search to find more information about the other algorithms.

PCA

Principal Components are new components that are linear combinations of the initial variables in a dataset. In addition, these components are uncorrelated and the most meaningful or important information is contained in these new components.

There are two advantages to PCA: 1) reduced computation time due to far fewer features and 2) the ability to graph the components when there are at most three components. If you have four or five components, you won't be able to display them visually, but you could select subsets of three components for visualization, and perhaps gain some additional insight into the dataset.

PCA uses the variance as a measure of information: the higher the variance, the more important the component. In fact, just to jump ahead slightly: PCA determines the eigenvalues and eigenvectors of a covariance matrix (discussed later), and constructs a new matrix whose columns are eigenvectors, ordered from left-to-right based on the maximum eigenvalue in the left-most column, decreasing until the right-most eigenvector also has the smallest eigenvalue.

Covariance Matrix

As a reminder, the statistical quantity called the variance of a random variable X is defined as follows:

```
variance(x) = [SUM (x – xbar)*(x-xbar)]/n
```

A covariance matrix `C` is an nxn matrix whose values on the main diagonal are the variance of the variables `X1, X2, . . ., Xn`. The other values of `C` are the covariance values of each pair of variables `Xi` and `Xj`.

The formula for the covariance of the variables `X` and `Y` is a generalization of the variance of a variable, and the formula is shown here:

```
covariance(X, Y) = [SUM (x – xbar)*(y-ybar)]/n
```

Notice that you can reverse the order of the product of terms (multiplication is commutative), and therefore the covariance matrix `C` is a symmetric matrix:

```
covariance(X, Y) = covariance(Y,X)
```

PCA calculates the eigenvalues and the eigenvectors of the covariance matrix A.

Working with Datasets

In addition to data cleaning, there are several other steps that you need to perform, such as selecting training data versus test data, and deciding whether to use "hold out" or cross-validation during the training process. More details are provided in the subsequent sections.

Training Data Versus Test Data

After you have performed the tasks described earlier in this chapter (i.e., data cleaning and perhaps dimensionality reduction), you are ready to split the dataset into two parts. The first part is the *training set*, which is used to train a model, and the second part is the *test set*, which is used for "inferencing" (another term for making predictions). Make sure that you conform to the following guidelines for your test sets:

- The set is large enough to yield statistically meaningful results
- It's representative of the data set as a whole
- Never train on test data
- Never test on training data

What Is Cross-validation?

The purpose of cross-validation is to test a model with non- overlapping test sets, which is performed in the following manner:

- Step 1) Split the data into k subsets of equal size
- Step 2) Select one subset for testing and the others for training
- Step 3) Repeat step 2 for the other k-1 subsets

This process is called *k-fold cross-validation*, and the overall error estimate is the average of the error estimates. A standard method for evaluation involves ten-fold cross-validation. Extensive experiments have shown that 10 subsets is the best choice to obtain an accurate estimate. In fact, you can repeat ten-fold cross-validation ten times and compute the average of the results, which helps to reduce the variance.

The next section discusses regularization, which is an important yet optional topic if you are primarily interested in TF 2 code. If you plan to become proficient in machine learning, you will need to learn about regularization.

What Is Regularization?

Regularization helps to solve overfitting problem, which occurs when a model performs well on training data but poorly on validation or test data.

Regularization solves this problem by adding a penalty term to the cost function, thereby controlling the model complexity with this penalty term.

Regularization is generally useful for:

- Large number of variables
- Low ratio of (# observations)/(# of variables)
- High multi-collinearity

There are two main types of regularization: L1 Regularization (which is related to MAE, or the absolute value of differences) and L2 Regularization (which is related to MSE, or the square of differences). In general, L2 performs better than L1, and it's efficient in terms of computation.

ML and Feature Scaling

Feature Scaling standardizes the range of features of data. This step is performed during the data preprocessing step, in part because gradient descent benefits from feature scaling.

The assumption is that the data conforms to a standard normal distribution, and standardization involves subtracting the mean and divide by the standard deviation for every data point, which results in a N(0,1) normal distribution.

Data Normalization vs Standardization

Data normalization is a linear scaling technique. Let's assume that a dataset has the values `{X1, X2, . . . , Xn}` along with the following terms:

```
Minx = minimum of Xi values

Maxx = maximum of Xi values
```

Now calculate a set of new Xi values as follows:

```
Xi = (Xi – Minx)/[Maxx – Minx]
```

The new `Xi` values are now scaled so that they are between 0 and 1.

The Bias-Variance Tradeoff

Bias in machine learning can be due to an error from wrong assumptions in a learning algorithm. High bias might cause an algorithm to miss relevant relations between features and target outputs (underfitting). Prediction bias can occur because of "noisy" data, an incomplete feature set, or a biased training sample.

Error due to bias is the difference between the expected (or average) prediction of your model and the correct value that you want to predict. Repeat the model building process multiple times, and gather new data each time, and also perform an analysis to produce a new model. The resulting models have a range of predictions because the underlying data sets have a degree of randomness. Bias measures the extent to the predictions for these models are from the correct value.

Variance in machine learning is the expected value of the squared deviation from the mean. High variance can/might cause an algorithm to model the random noise in the training data, rather than the intended outputs (aka overfitting).

Adding parameters to a model increases its complexity, increases the variance, and decreases the bias. Dealing with bias and variance is dealing with underfitting and overfitting.

Error due to variance is the variability of a model prediction for a given data point. As before, repeat the entire model building process, and the variance is the extent to which predictions for a given point vary among different "instances" of the model.

Metrics for Measuring Models

One of the most frequently used metrics is R-squared, which measures how close the data is to the fitted regression line (regression coefficient). The R-squared value is always a percentage between 0 and 100%. The value 0% indicates that the model explains none of the variability of the response data around its mean. The value 100% indicates that the model explains all the variability of the response data around its mean. In general, a higher R-squared value indicates a better model.

Limitations of R-Squared

Although high R-squared values are preferred, they are not necessarily always good values. Similarly, low R-squared values are not always bad. For example, an R-squared value for predicting human behavior is often less than 50%. Moreover, R-squared cannot determine whether the coefficient estimates and predictions are biased. In addition, an R-squared value does not indicate whether a regression model is adequate. Thus, it's possible to have a low R-squared value for a good model, or a high R-squared value

for a poorly fitting model. Evaluate R-squared values in conjunction with residual plots, other model statistics, and subject area knowledge.

Confusion Matrix

In its simplest form, a confusion matrix (also called an error matrix) is a type of contingency table with two rows and two columns that contains the # of false positives, false negatives, true positives, and true negatives. The four entries in a 2x2 confusion matrix can be labeled as follows:

```
TP: True Positive
FP: False Positive
TN: True Negative
FN: False Negative
```

The diagonal values of the confusion matrix are correct, whereas the off-diagonal values are incorrect predictions. In general a lower FP value is better than a FN value. For example, an FP indicates that a healthy person was incorrectly diagnosed with a disease, whereas an FN indicates that an unhealthy person was incorrectly diagnosed as healthy.

Accuracy vs Precision vs Recall

A 2x2 confusion matrix has four entries that that represent the various combinations of correct and incorrect classifications. Given the definitions in the preceding section, the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
recall    = TP/[TP + FN]
```

Accuracy can be an unreliable metric because it yields misleading results in unbalanced data sets. When the number of observations in different classes are significantly different, it gives equal importance to both false positive and false negative classifications. For example, declaring cancer as benign is worse than incorrectly informing patients that they are suffering from cancer. Unfortunately, accuracy won't differentiate between these two cases.

Keep in mind that the confusion matrix can be an nxn matrix and not just a 2x2 matrix. For example, if a class has 5 possible values, then the confusion matrix is a 5x5 matrix, and the numbers on the main diagonal are the "true positive" results.

The ROC Curve

The ROC (receiver operating characteristic) curve is a curve that plots the TPR, which is the true positive rate (i.e., the recall) against the FPR, which is the false positive rate). Note that the TNR (the true negative rate) is also called the specificity.

The following link contains a Python code sample using SKLearn and the Iris dataset, and also code for plotting the ROC:

https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

The following link contains an assortment of Python code samples for plotting the ROC:

https://stackoverflow.com/questions/25009284/how-to-plot-roc-curve-in-python

Other Useful Statistical Terms

Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- `RSS`
- `TSS`
- `R^2`
- `F1 score`
- `p-value`

The definitions of `RSS`, `TSS`, and `R^2` are shown below, where `y^` is the y-coordinate of a point on a best-fitting line and `y_` is the mean of the y-values of the points in the dataset:

```
RSS = sum of squares of residuals (y - y^)**2
TSS = toal sum of squares         (y - y_)**2
R^2 = 1 - RSS/TSS
```

What Is an F1 Score?

The F1 score is a measure of the accuracy of a test, and it's defined as the harmonic mean of precision and recall. Here are the relevant formulas, where `p` is the precision and `r` is the recall:

```
p = (# of correct positive results)/(# of all
       positive results)
```

```
r = (# of correct positive results)/(# of all
       relevant samples)
F1-score  = 1/[((1/r) + (1/p))/2]
          = 2*[p*r]/[p+r]
```

The best value of an F1 score is 0 and the worse value is 0. Keep in mind that an F1 score tends to be used for categorical classification problems, whereas the R^2 value is typically for regression tasks (such as linear regression).

What Is a p-value?

The p-value is used to reject the null hypothesis if the p-value is small enough (< 0.005) which indicates a higher significance. Recall that the null hypothesis states that there is no correlation between a dependent variable (such as y) and an independent variable (such as x). The threshold value for p is typically 1% or 5%.

There is no straightforward formula for calculating p-values, which are values that are always between 0 and 1. In fact, p-values are statistical quantities to evaluate the so-called "null hypothesis," and they are calculated by means of p-value tables or via spreadsheet/statistical software.

What Is Linear Regression?

The goal of linear regression is to find the best fitting line that "represents" a dataset. Keep in mind two key points. First, the best fitting line does not necessarily pass through all (or even most of) the points in the dataset. The purpose of a best fitting line is to minimize the vertical distance of that line from the points in dataset. Second, linear regression does not determine the best-fitting polynomial: the latter involves finding a higher-degree polynomial that passes through many of the points in a dataset.

Moreover, a dataset in the plane can contain two or more points that lie on the same *vertical* line, which is to say that those points have the same x value. However, a function *cannot* pass through such a pair of points: if two points `(x1,y1)` and `(x2,y2)` have the same x value then they must have the same y value (i.e., `y1=y2`). On the other hand, a function can have two or more points that lie on the same *horizontal* line.

Now consider a scatter plot with many points in the plane that are sort of "clustered" in an elongated cloud-like shape: a best-fitting line will probably intersect only limited number of points (in fact, a best-fitting line might not intersect *any* of the points).

One other scenario to keep in mind: suppose a dataset contains a set of points that lie on the same line. For instance, let's say the x values are in the set `{1,2,3,...,10}` and the y values are in the set `{2,4,6,...,20}`. Then the equation of the best-fitting line is `y=2*x+0`. In this scenario, all the points are *collinear*, which is to say that they lie on the same line.

Linear Regression vs Curve-Fitting

Suppose a dataset consists of n data points of the form (x, y), and no two of those data points have the same `x` value. Then according to a well-known result in mathematics, there is a polynomial of degree less than or equal to n-1 that passes through those n points (if you are really interested, you can find a mathematical proof of this statement in online articles). For example, a line is a polynomial of degree one and it can intersect any pair of non-vertical points in the plane. For any triple of points (that are not all on the same line) in the plane, there is a quadratic equation that passes through those points.

In addition, sometimes a lower degree polynomial is available. For instance, consider the set of 100 points in which the `x` value equals the `y` value: in this case, the line `y = x` (which is a polynomial of degree one) passes through all 100 points.

However, keep in mind that the extent to which a line "represents" a set of points in the plane depends on how closely those points can be approximated by a line, which is measured by the *variance* of the points (the variance is a statistical quantity). The more collinear the points, the smaller the variance; conversely, the more "spread out" the points are, the larger the variance.

When Are Solutions Exact Values?

Although statistics-based solutions provide closed-form solutions for linear regression, neural networks provide *approximate* solutions. This is due to the fact that machine learning algorithms for linear regression involve a sequence of approximations that "converges" to optimal values, which

means that machine learning algorithms produce estimates of the exact values. For example, the slope `m` and y-intercept `b` of a best-fitting line for a set of points a 2D plane have a closed-form solution in statistics, but they can only be approximated via machine learning algorithms (exceptions do exist, but they are rare situations).

Keep in mind that even though a closed-form solution for "traditional" linear regression provides an exact value for both `m` and `b`, sometimes you can only use an approximation of the exact value. For instance, suppose that the slope m of a best-fitting line equals the square root of 3 and the y-intercept b is the square root of 2. If you plan to use these values in source code, you can only work with an approximation of these two numbers. In the same scenario, a Neural Network computes approximations for `m` and `b`, regardless of whether or not the exact values for `m` and `b` are irrational, rational, or integer values. However, machine learning algorithms are better suited for complex, non-linear, multi-dimensional datasets, which is beyond the capacity of linear regression.

As a simple example, suppose that the closed form solution for a linear regression problem produces integer or rational values for both `m` and `b`. Specifically, let's suppose that a closed form solution yields the values 2.0 and 1.0 for the slope and y-intercept, respectively, of a best-fitting line. The equation of the line looks like this:

```
y = 2.0 * x + 1.0
```

However, the corresponding solution from training a neural network might produce the values 2.0001 and 0.9997 for the slope m and the y-intercept `b`, respectively, as the values of `m` and `b` for a best-fitting line. Always keep this point in mind, especially when you are training a Neural Network.

What Is Multivariate Analysis?

Multivariate analysis generalizes the equation of a line in the Euclidean plane to higher dimensions, and it's called a *hyper plane* instead of a line. The generalized equation has the following form:

```
y = w1*x1 + w2*x2 + . . . + wn*xn + b
```

In the case of 2D linear regression, you only need to find the value of the slope (`m`) and the y-intercept `(b)`, whereas in multivariate analysis you

need to find the values for `w1, w2, . . ., wn`. Note that multivariate analysis is a term from statistics, and in machine learning it's often referred to as "generalized linear regression."

Keep in mind that most of the code samples in this book that pertain to linear regression involve 2D points in the Euclidean plane.

Other Types of Regression

Linear regression finds the best fitting line that "represents" a dataset, but what happens if a line in the plane is not a good fit for the dataset? This is a relevant question when you work with datasets.

Some alternatives to linear regression include quadratic equations, cubic equations, or higher-degree polynomials. However, these alternatives involve trade-offs, as we'll discuss later.

Another possibility is a sort of hybrid approach that involves piece-wise linear functions, which comprises a set of line segments. If contiguous line segments are connected then it's a piece-wise linear continuous function; otherwise it's a piece-wise linear discontinuous function.

Thus, given a set of points in the plane, regression involves addressing the following questions:

- What type of curve fits the data well? How do we know?
- Does another type of curve fit the data better?
- What does "best fit" mean?

One way to check if a line fits the data involves a visual check, but this approach does not work for data points that are higher than two dimensions. Moreover, this is a subjective decision, and some sample datasets are displayed later in this chapter. By visual inspection of a dataset, you might decide that a quadratic or cubic (or even higher degree) polynomial has the potential of being a better fit for the data. However, visual inspection is probably limited to points in a 2D plane or in three dimensions.

Let's defer the non-linear scenario and let's make the assumption that a line would be a good fit for the data. There is a well-known technique for finding the "best fitting" line for such a dataset that involves minimizing the Mean Squared Error (MSE) that we'll discuss later in this chapter.

The next section provides a quick review of linear equations in the plane, along with some images that illustrate examples of linear equations.

Working with Lines in the Plane (optional)

This section contains a short review of lines in the Euclidean plane, so you can skip this section if you are comfortable with this topic. A minor point that's often overlooked is that lines in the Euclidean plane have infinite length. If you select two distinct points of a line, then all the points between those two selected points is a *line segment*. A *ray* is a "half infinite" line: when you select one point as an endpoint, then all the points on one side of the line constitutes a ray.

For example, the points in the plane whose y-coordinate is 0 is a line and also the x-axis, whereas the points between (0,0) and (1,0) on the x-axis form a line segment. In addition, the points on the x-axis that are to the right of (0,0) form a ray, and the points on the x-axis that are to the left of (0,0) also form a ray.

For simplicity and convenience, in this book we'll use the terms "line" and "line segment" interchangeably, and now let's delve into the details of lines in the Euclidean plane. Just in case you're a bit fuzzy on the details, here is the equation of a (non-vertical) line in the Euclidean plane:

```
y = m*x + b
```

The value of `m` is the slope of the line and the value of `b` is the y-intercept (i.e., the place where the line intersects the y-axis).

If need be, you can use a more general equation that can also represent vertical lines, as shown here:

```
a*x + b*y + c = 0
```

However, we won't be working with vertical lines, so we'll stick with the first formula.

Figure 2.1 displays three horizontal lines whose equations (from top to bottom) are `y = 3`, `y = 0`, and `y = -3`, respectively.

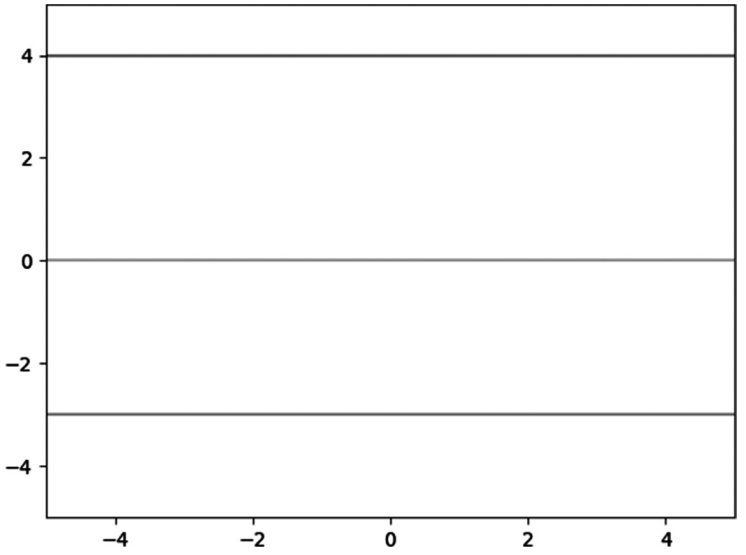


FIGURE 2.1 A Graph of Three Horizontal Line Segments.

Figure 2.2 displays two slanted lines whose equations are `y = x` and `y = -x`, respectively.

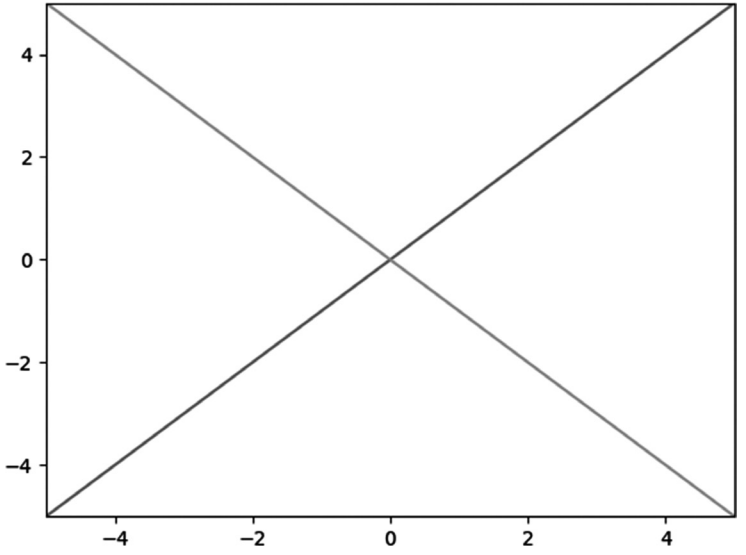


FIGURE 2.2 A Graph of Two Diagonal Line Segments.

Figure 2.3 displays two slanted parallel lines whose equations are `y = 2*x` and `y = 2*x + 3`, respectively.

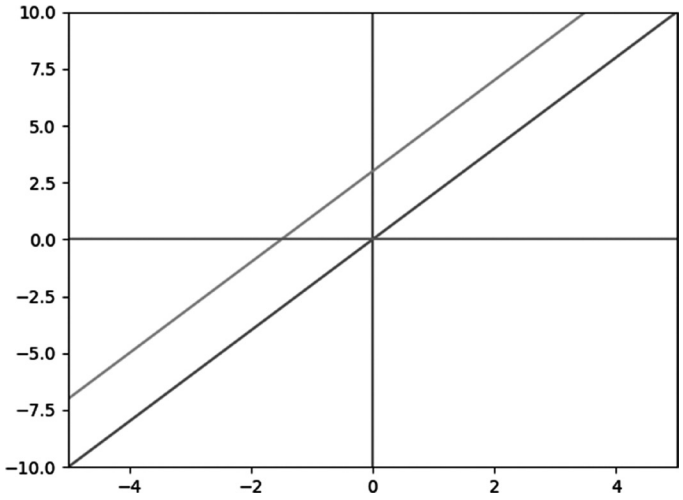


FIGURE 2.3 A Graph of Two Slanted Parallel Line Segments.

Figure 2.4 displays a piece-wise linear graph consisting of connected line segments.

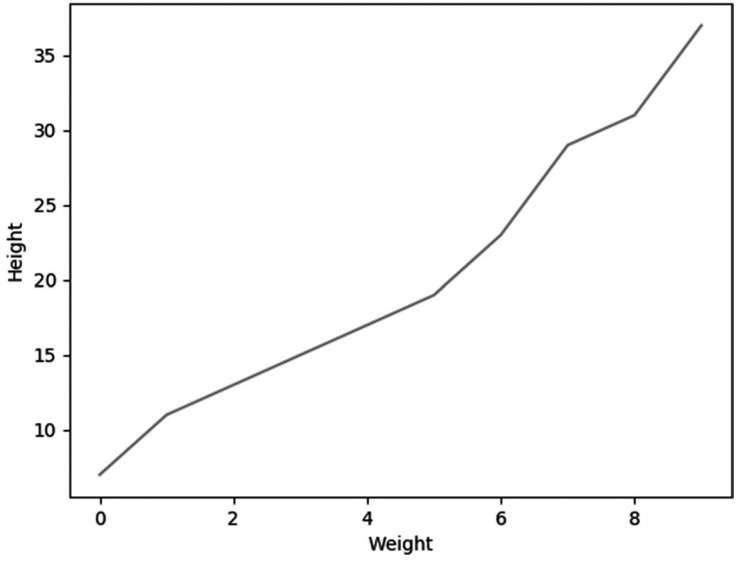


FIGURE 2.4 A Piece-wise Linear Graph of Line Segments.

Now let's turn our attention to generating quasi-random data using a NumPy API, and then we'll plot the data using Matplotlib.

Scatter Plots with NumPy and Matplotlib (1)

Listing 2.1 displays the contents of np_plot1.py that illustrates how to use the Numpy randn() API to generate a dataset and then the scatter() API in Matplotlib to plot the points in the dataset.

One detail to note is that all the adjacent horizontal values are equally spaced, whereas the vertical values are based on a linear equation plus a "perturbation" value. This "perturbation technique" (which is not a standard term) is used in other code samples in this chapter in order to add a slightly randomized effect when the points are plotted. The advantage of this technique is that the best-fitting values for m and b are known in advance, and therefore we do not need to guess their values.

Listing 2.1: np_plot1.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.randn(15,1)
y = 2.5*x + 5 + 0.2*np.random.randn(15,1)

print("x:",x)
print("y:",y)

plt.scatter(x,y)
plt.show()
```

Listing 2.1 contains two import statements and then initializes the array variable x with 15 random numbers between 0 and 1.

Next, the array variable y is defined in two parts: the first part is a linear equation 2.5*x + 5 and the second part is a "perturbation" value that is based on a random number. Thus, the array variable y simulates a set of values that closely approximate a line segment.

This technique is used in code samples that simulate a line segment, and then the training portion approximates the values of m and b for the

best-fitting line. Obviously we already *know* the equation of the best fitting-line: the purpose of this technique is to compare the trained values for the slope `m` and y-intercept `b` with the known values (which in this case are 2.5 and 5).

A partial output from Listing 2.1 is here:

```
x: [[-1.42736308]
 [ 0.09482338]
 [-0.45071331]
 [ 0.19536304]
 [-0.22295205]
 // values omitted for brevity
y: [[1.12530514]
 [5.05168677]
 [3.93320782]
 [5.49760999]
 [4.46994978]
 // values omitted for brevity
```

Figure 2.5 displays a scatter plot of points based on the values of `x` and `y`.

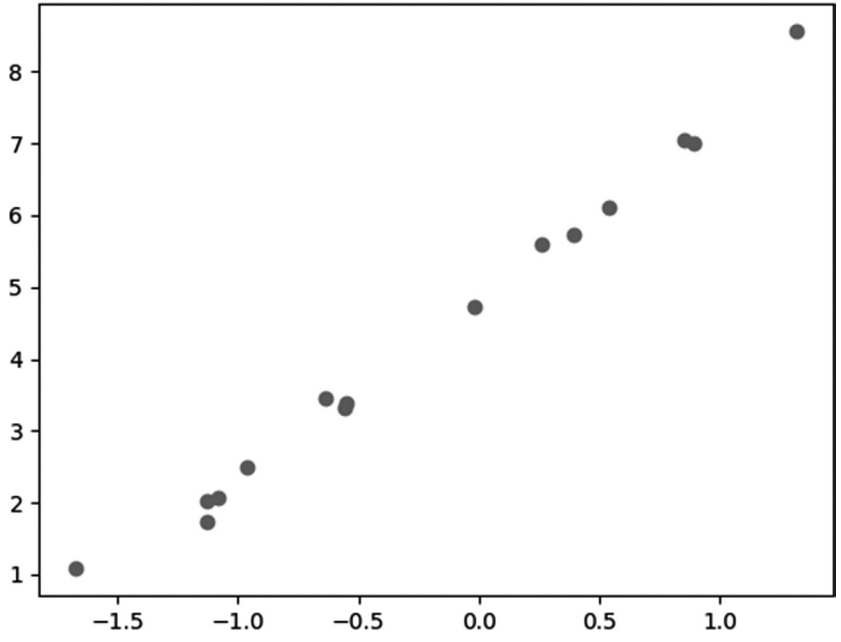


FIGURE 2.5 A Scatter Plot of Points for a Line Segment.

Why the "Perturbation Technique" Is Useful

You already saw how to use the "perturbation technique" and by way of comparison, consider a dataset with the following points that are defined in the Python array variables X and Y:

```
X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51, 0.34,0.1,0.19,0.53,1.0,0.58]
```

If you need to find the best fitting line for the preceding dataset, how would you guess the values for the slope m and the y-intercept b? In most cases, you probably cannot guess their values. On the other hand, the "perturbation technique" enables you to "jiggle" the points on a line whose value for the slope m (and optionally the value for the y-intercept b) is specified in advance.

Keep in mind that the "perturbation technique" only works when you introduce small random values that do not result in different values for m and b.

Scatter Plots with NumPy and Matplotlib (2)

The code in Listing 2.1 assigned random values to the variable x, whereas a hard-coded value is assigned to the slope m. The y values are a hard-coded multiple of the x values, plus a random value that is calculated via the "perturbation technique". Hence we do not know the value of the y-intercept b.

In this section the values for trainX are based on the np.linspace() API, and the values for trainY involve the "perturbation technique" that is described in the previous section.

The code in this example simply prints the values for trainX and trainY, which correspond to data points in the Euclidean plane. Listing 2.2 displays the contents of np_plot2.py that illustrates how to simulate a linear dataset in NumPy.

Listing 2.2: np_plot2.py

```
import numpy as np

trainX = np.linspace(-1, 1, 11)
trainY = 4*trainX + np.random.randn(*trainX.shape)*0.5

print("trainX: ",trainX)
print("trainY: ",trainY)
```

Listing 2.6 initializes the NumPy array variable trainX via the NumPy linspace() API, followed by the array variable trainY that is defined in two parts. The first part is the linear term 4*trainX and the second part involves the "perturbation technique" that is a randomly generated number. The output from Listing 2.6 is here:

```
trainX:  [-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4
          0.6  0.8  1. ]
trainY:  [-3.60147459 -2.66593108 -2.26491189
          -1.65121314 -0.56454605  0.22746004
          0.86830728  1.60673482  2.51151543
          3.59573877  3.05506056]
```

The next section contains an example that is similar to Listing 2.2, using the same "perturbation technique" to generate a set of points that approximate a quadratic equation instead of a line segment.

A Quadratic Scatterplot with NumPy and Matplotlib

Listing 2.3 displays the contents of np_plot_quadratic.py that illustrates how to plot a quadratic function in the plane.

Listing 2.3: np_plot_quadratic.py

```
import numpy as np
import matplotlib.pyplot as plt

#see what happens with this set of values:
#x = np.linspace(-5,5,num=100)

x = np.linspace(-5,5,num=100)[:,None]
y = -0.5 + 2.2*x +0.3*x**2 + 2*np.random.
    randn(100,1)
print("x:",x)

plt.plot(x,y)
plt.show()
```

Listing 2.3 initializes the array variable x with the values that are generated via the np.linspace() API, which in this case is a set of 100 equally spaced decimal numbers between -5 and 5. Notice the snippet [:,None] in

the initialization of x, which results in an array of elements, each of which is an array consisting of a single number.

The array variable y is defined in two parts: the first part is a quadratic equation `-0.5 + 2.2*x +0.3*x**2` and the second part is a "perturbation" value that is based on a random number (similar to the code in Listing 2.1). Thus, the array variable y simulates a set of values that approximates a quadratic equation. The output from Listing 2.3 is here:

```
x:
[[-5.        ]
 [-4.8989899 ]
 [-4.7979798 ]
 [-4.6969697 ]
 [-4.5959596 ]
 [-4.49494949]
 // values omitted for brevity
 [ 4.8989899 ]
 [ 5.        ]]
```

Figure 2.6 displays a scatter plot of points based on the values of x and y, which have an approximate shape of a quadratic equation.

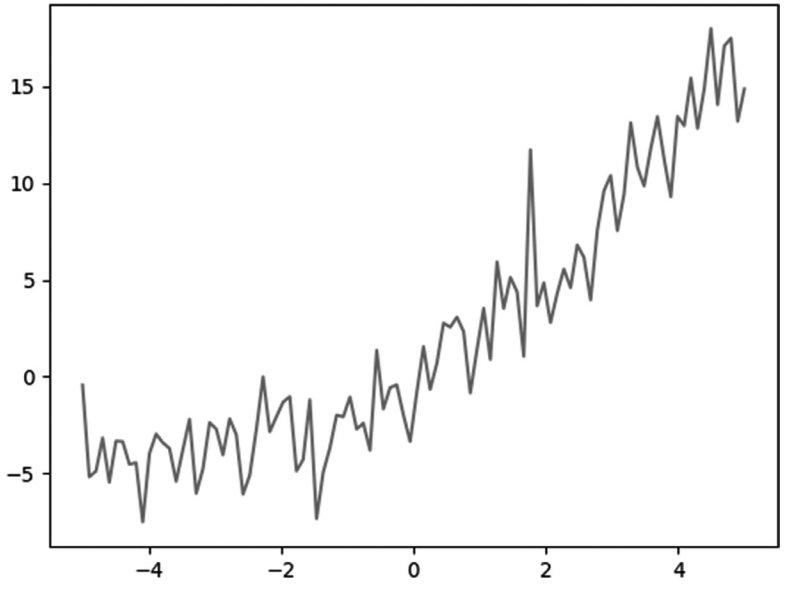


FIGURE 2.6 A Scatter Plot of Points for a Quadratic Equation.

The Mean Squared Error (MSE) Formula

In plain English, the MSE is the sum of the squares of the difference between an actual y value and the predicted y value, divided by the number of points. Notice that the predicted y value is the y value that each point would have if that point were actually on the best-fitting line.

Although the MSE is popular for linear regression, there are other error types available, some of which are discussed briefly in the next section.

A List of Error Types

Although we will only discuss MSE for linear regression in this book, there are other types of formulas that you can use for linear regression, some of which are listed here:

- MSE
- RMSE
- RMSPROP
- MAE

The MSE is the basis for the preceding error types. For example, RMSE is "Root Mean Squared Error," which is the square root of MSE.

On the other hand, MAE is "Mean Absolute Error," which is the sum of *the absolute value of the differences of the y terms* (*not* the square of the differences of the y terms), which is then divided by the number of terms.

The RMSProp optimizer utilizes the magnitude of recent gradients to normalize the gradients. Specifically, RMSProp maintain a moving average over the RMS (root mean squared) gradients, and then divides that term by the current gradient.

Although it's easier to compute the derivative of MSE, it's also true that MSE is more susceptible to outliers, whereas MAE is less susceptible to outliers. The reason is simple: a squared term can be significantly larger than the absolute value of a term. For example, if a difference term is 10, then a squared term of 100 is added to MSE, whereas only 10 is added to MAE. Similarly, if a difference term is -20, then a squared term 400 is added to MSE, whereas only 20 (which is the absolute value of -20) is added to MAE.

Non-linear Least Squares

When predicting housing prices, where the dataset contains a wide range of values, techniques such as linear regression or random forests can cause the model to overfit the samples with the highest values in order to reduce quantities such as mean absolute error.

In this scenario, you probably want an error metric, such as relative error, that reduces the importance of fitting the samples with the largest values. This technique is called *non-linear least squares*, which may use a log-based transformation of labels and predicted values.

The next section contains several code samples, the first of which involves calculating the MSE manually, followed by an example that uses `NumPy` formulas to perform the calculations. Finally, we'll look at a TensorFlow example for calculating the MSE.

Calculating the MSE Manually

This section contains two line graphs, both of which contain a line that approximates a set of points in a scatter plot.

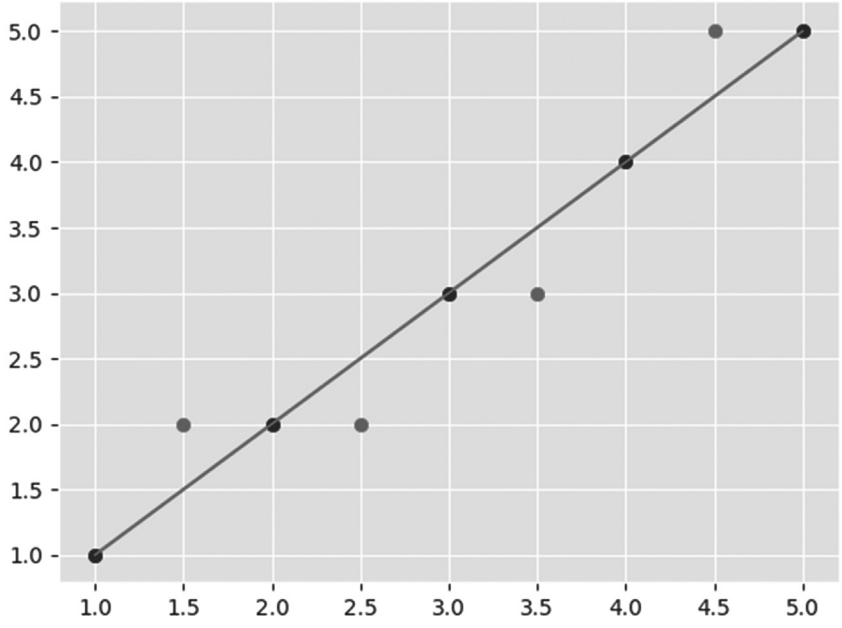


FIGURE 2.7 A Line Graph that Approximates Points of a Scatter Plot.

Figure 2.7 displays a line segment that approximates a scatter plot of points (some of which intersect the line segment). The MSE for the line in Figure 2.7 is computed as follows:

```
MSE = (1*1 + (-1)*(-1) + (-1)*(-1) + 1*1)/7 = 4/7
```

Figure 2.8 displays a set of points and a line that is a potential candidate for best-fitting line for the data. The MSE for the line in Figure 2.8 is computed as follows:

```
MSE = ((-2)*(-2) + 2*2)/7 = 8/7
```

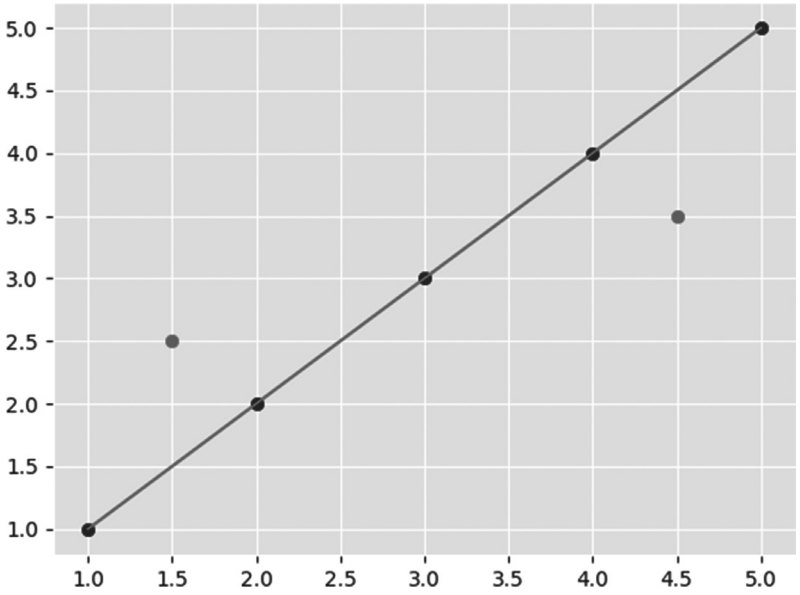


FIGURE 2.8 A Line Graph that Approximates Points of a Scatter Plot.

Thus, the line in Figure 2.7 has a smaller MSE than the line in Figure 2.8, which might have surprised you (or did you guess correctly?)

In these two figures we calculated the MSE easily and quickly, but in general it's significantly more difficult. For instance, if we plot 10 points in the Euclidean plane that do not closely fit a line, with individual terms that involve non-integer values, we would probably need a calculator.

A better solution involves `NumPy` functions, such as the `np.linspace()` API, as discussed in the next section.

Approximating Linear Data with `np.linspace()`

Listing 2.4 displays the contents of `np_linspace1.py` that illustrates how to generate some data with the `np.linspace()` API in conjunction with the "perturbation technique."

Listing 2.4: np_linspace1.py

```
import numpy as np

trainX = np.linspace(-1, 1, 6)
trainY = 3*trainX+ np.random.randn(*trainX.
         shape)*0.5

print("trainX: ", trainX)
print("trainY: ", trainY)
```

The purpose of this code sample is merely to generate and display a set of randomly generated numbers. Later in this chapter we will use this code as a starting point for an actual linear regression task.

Listing 2.4 starts with the definition of the array variable `trainX` that is initialized via the `np.linspace()` API. Next, the array variable `trainY` is defined via the "perturbation technique" that you have seen in previous code samples. The output from Listing 2.4 is here:

```
trainX:  [-1.  -0.6 -0.2  0.2  0.6  1. ]
trainY:  [-2.9008553  -2.26684745 -0.59516253
          0.66452207  1.82669051  2.30549295]
trainX:  [-1.  -0.6 -0.2  0.2  0.6  1. ]
trainY:  [-2.9008553  -2.26684745 -0.59516253
          0.66452207  1.82669051  2.30549295]
```

Now that we know how to generate (`x,y`) values for a linear equation, let's learn how to calculate the MSE, which is discussed in the next section.

The next example generates a set of data values using the `np.linspace()` method and the `np.random.randn()` method in order to introduces some randomness in the data points.

Calculating MSE with `np.linspace()` API

The code sample in this section differs from many of the earlier code samples in this chapter: it uses a hard-coded array of values for `X` and also for `Y` instead of the "perturbation" technique. Hence, you will *not* know the correct value for the slope and y-intercept (and you probably will not be able to guess their correct values). Listing 2.5 displays the contents of `plain_linreg1.py` that illustrates how to compute the MSE with simulated data.

Listing 2.5: plain_linreg1.py

```
import numpy as np
import matplotlib.pyplot as plt

X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51,
   0.34,0.1,0.19,0.53,1.0,0.58]

costs = []
#Step 1: Parameter initialization
W = 0.45
b = 0.75

for i in range(1, 100):
  #Step 2: Calculate Cost
  Y_pred = np.multiply(W, X) + b
  Loss_error = 0.5 * (Y_pred - Y)**2
  cost = np.sum(Loss_error)/10

  #Step 3: Calculate dW and db
  db = np.sum((Y_pred - Y))
  dw = np.dot((Y_pred - Y), X)
  costs.append(cost)

  #Step 4: Update parameters:
  W = W - 0.01*dw
  b = b - 0.01*db

  if i%10 == 0:
    print("Cost at", i,"iteration = ", cost)
```

(Continued)

```
#Step 5: Repeat via a for loop with 1000 iterations

#Plot cost versus # of iterations
print("W = ", W,"& b = ",  b)
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per tens)')
plt.show()
```

Listing 2.5 initializes the array variables `X` and `Y` with hard-coded values, and then initializes the scalar variables `W` and `b`. The next portion of Listing 2.5 contains a `for` loop that iterates 100 times. After each iteration of the loop, the variables `Y_pred, Loss_error`, and `cost` are calculated. Next, the values for `dw` and `db` are calculated, based on the sum of the terms in the array `Y_pred-Y`, and the inner product of `Y_pred-y` and `X`, respectively.

Notice how `W` and b are updated: their values are decremented by the term `0.01*dw` and `0.01*db`, respectively. This calculation ought to look somewhat familiar: the code is programmatically calculating an approximate value of the gradient for `W` and `b`, both of which are multiplied by the learning rate (the hard-coded value 0.01), and the resulting term is decremented from the current values of `W` and `b` in order to produce a new approximation for `W` and `b`. Although this technique is very simple, it does calculate reasonable values for `W` and `b`.

The final block of code in Listing 2.5 displays the intermediate approximations for `W` and `b`, along with a plot of the cost (vertical axis) versus the number of iterations (horizontal axis). The output from Listing 2.5 is here:

```
Cost at 10 iteration =  0.04114630674619492
Cost at 20 iteration =  0.026706242729839392
Cost at 30 iteration =  0.024738889446900423
Cost at 40 iteration =  0.023850565034634254
Cost at 50 iteration =  0.0231499048706651
Cost at 60 iteration =  0.02255361434242207
Cost at 70 iteration =  0.0220425055291673
Cost at 80 iteration =  0.021604128492245713
Cost at 90 iteration =  0.021228111750568435
W =  0.47256473531193927 & b =  0.19578262688662174
```

Figure 2.9 displays a scatter plot of points generated by the code in Listing 2.5.

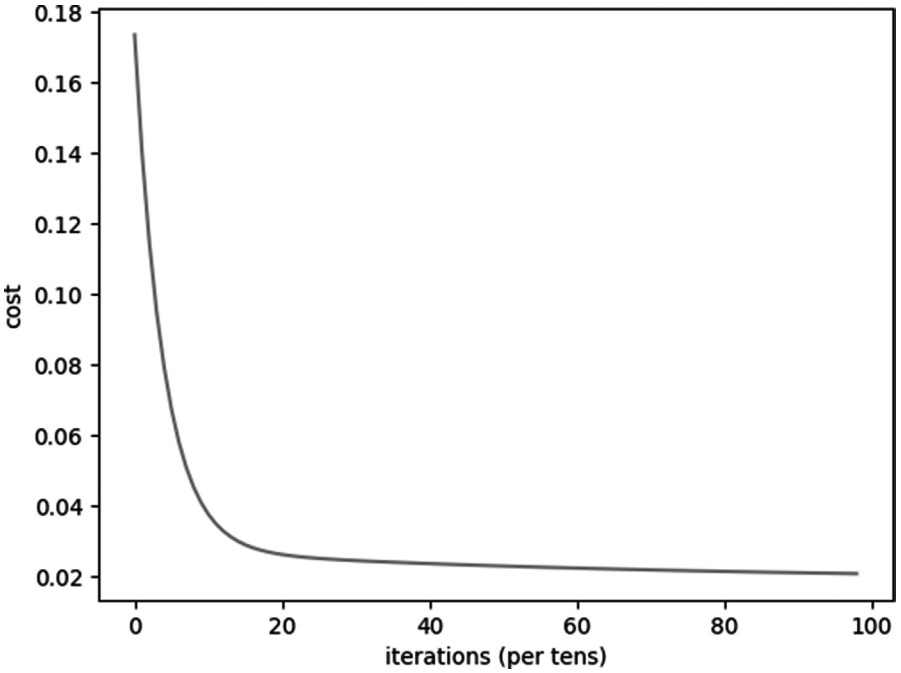


FIGURE 2.9 MSE Values With Linear Regression.

The code sample `plain-linreg2.py` is similar to the code in Listing 2.5: the difference is that instead of a single loop with 100 iterations, there is an outer loop that execute 100 times, and during each iteration of the outer loop, the inner loop also execute 100 times.

Linear Regression with `Keras`

The code sample in this section contains primarily `Keras` code in order to perform linear regression. If you have read the previous examples in this chapter, this section will be easier for you to understand because the steps for linear regression are the same.

Listing 2.6 displays the contents of `keras_linear_regression.py` that illustrates how to perform linear regression in `Keras`.

Listing 2.6: keras_linear_regression.py

```
##################################################
   ##############
#Keep in mind the following important points:
#1) Always standardize both input features and
    target variable:
#doing so only on input feature produces incorrect
   predictions
#2) Data might not be normally distributed: check
    the data and
#based on the distribution apply StandardScaler,
   MinMaxScaler,
#Normalizer or RobustScaler
##################################################
   ##############

import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_
   split

df = pd.read_csv('housing.csv')
X  = df.iloc[:,0:13]
y  = df.iloc[:,13].values

mmsc = MinMaxScaler()
X  = mmsc.fit_transform(X)
y  = y.reshape(-1,1)
y  = mmsc.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_
   split(X, y, test_size=0.3)
```

```
# this Python method creates a Keras model
def build_keras_model():
  model = tf.keras.models.Sequential()
  model.add(tf.keras.layers.Dense(units=13,
    input_dim=13))
  model.add(tf.keras.layers.Dense(units=1))

  model.compile(optimizer='adam',loss='mean_
    squared_error',metrics=['mae','accuracy'])
  return model

batch_size=32
epochs = 40

# specify the Python method 'build_keras_model'
   to create a Keras model
# using the implementation of the scikit-learn
   regressor API for Keras
model = tf.keras.wrappers.scikit_learn.
        KerasRegressor(build_fn=build_
        keras_model, batch_size=batch_
        size,epochs=epochs)

# train ('fit') the model and then make
   predictions:
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
#print("y_test:",y_test)
#print("y_pred:",y_pred)

# scatter plot of test values-vs-predictions
fig, ax = plt.subplots()
ax.scatter(y_test, y_pred)
ax.plot([y_test.min(), y_test.max()], [y_test.
   min(), y_test.max()], 'r*--')
ax.set_xlabel('Calculated')
ax.set_ylabel('Predictions')
plt.show()
```

Listing 2.6 starts with multiple `import` statements and then initializes the dataframe `df` with the contents of the CSV file `housing.csv` (a portion of which is shown in Listing 2.7). Notice that the training set `X` is initialized with the contents of the first 13 columns of the dataset `housing.csv`, and the variable y contains the rightmost column of the dataset housing.csv.

The next section in Listing 2.6 uses the `MinMaxScaler` class to calculate the mean and standard deviation, and then invokes the `fit_transform()` method in order to update the `X` values and the `y` values so that they have a mean of 0 and a standard deviation of 1.

Next, the `build_keras_mode()` Python method creates a `Keras`-based model with two dense layers. Notice that the input layer has size 13, which is the number of columns in the dataframe `X`. The next code snippet compiles the model with an `adam` optimizer, the MSE loss function, and also specifies the MAE and accuracy for the metrics. The compiled model is then returned to the caller.

The next portion of Listing 2.6 initializes the `batch_size` variable to 32 and the epochs variable to 40, and specifies them in the code snippet that creates the model, as shown here:

```
model =
tf.keras.wrappers.scikit_learn.
   KerasRegressor(build_fn=build_keras_model,
   batch_size=batch_size,epochs=epochs)
```

The short comment block that appears in Listing 2.6 explains the purpose of the preceding code snippet, which constructs our `Keras` model.

The next portion of Listing 2.6 invokes the `fit()` method to train the model and then invokes the `predict()` method on the `X_test` data to calculate a set of predictions and initialize the variable `y_pred` with those predictions.

The final portion of Listing 2.6 displays a scatter plot in which the horizontal axis is the values in `y_test` (the actual values from the CSV file housing.csv) and the vertical axis is the set of predicted values.

Figure 2.5 displays a scatter plot of points based on the test values and the predictions for those test values.

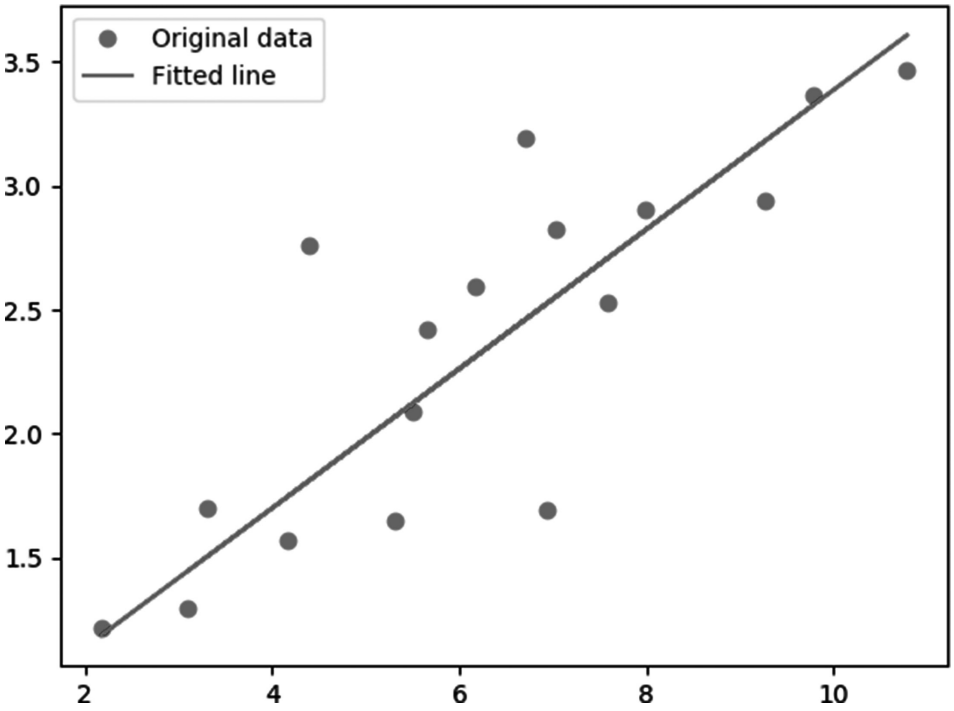


FIGURE 2.10 A Scatter Plot and a Best-Fitting Line.

Listing 2.7 displays the first four rows of the CSV file `housing.csv` that is used in the Python code in Listing 2.6.

Listing 2.7: housing.csv

```
0.00632,18,2.31,0,0.538,6.575,65.2,4.09,1,296,15.3,
   396.9,4.98,24
0.02731,0,7.07,0,0.469,6.421,78.9,4.9671,2,242,17
   .8,396.9,9.14,21.6
0.02729,0,7.07,0,0.469,7.185,61.1,4.9671,2,242,17
   .8,392.83,4.03,34.7
0.03237,0,2.18,0,0.458,6.998,45.8,6.0622,3,222,18
   .7,394.63,2.94,33.4
```

Summary

This chapter introduced you to machine learning and concepts such as feature selection, feature engineering, data cleaning, training sets, and test sets. Next you learned about supervised, unsupervised, and semi-supervised learning. Then you learned regression tasks, classification tasks, and clustering, as well as the steps that are typically required in order to prepare a dataset. These steps include "feature selection" or "feature extraction" that can be performed using various algorithms. Then you learned about issue that can arise with the data in datasets, and how to rectify them.

In addition, you also learned about linear regression, along with a brief description of how to calculate a best-fitting line for a dataset of values in the Euclidean plane. You saw how to perform linear regression using `NumPy` in order to initialize arrays with data values, along with a "perturbation" technique that introduces some randomness for the y values. This technique is useful because you will know the correct values for the slope and y-intercept of the best-fitting line, which you can then compare with the trained values.

You then learned how to perform linear regression in code samples that involve `Keras`. In addition, you saw how to use Matplotlib in order to display line graphs for best-fitting lines and graphs that display the cost versus the number of iterations during the training-related code blocks.

CHAPTER 3

CLASSIFIERS IN MACHINE LEARNING

This chapter presents numerous classification algorithms in machine learning. This includes algorithms such as the kNN (k Nearest Neighbor) algorithm, logistic regression (despite its name it *is* a classifier), decision trees, random forests, SVMs, and Bayesian classifiers. The emphasis on algorithms is intended to introduce you to machine learning, which includes a tree-based code sample that relies on `scikit-learn`. The latter portion of this chapter contains `Keras`-based code samples for standard datasets.

Due to space constraints, this chapter does not cover other well-known algorithms, such as Linear Discriminant Analysis and the kMeans algorithm (which is for unsupervised learning and clustering). However, there are many online tutorials available that discuss these and other algorithms in machine learning.

With these points in mind, the first section of this chapter briefly discusses the classifiers that are mentioned in the introductory paragraph. The second section of this chapter provides an overview of activation functions, which will be very useful if you decide to learn about deep neural networks. In this section you will learn how and why they are used in neural networks. This section also contains a list of the TensorFlow APIs for activation functions, followed by a description of some of their merits.

The third section introduces logistic regression, which relies on the sigmoid function, which is also used in RNNs (Recurrent Neural

Networks) and LSTMs (Long Short Term Memory). The fourth part of this chapter contains a code sample involving Logistic Regression and the `MNIST` dataset.

In order to give you some context, classifiers are one of three major types of algorithms: regression algorithms (such as linear regression in Chapter 4), classification algorithms (discussed in this chapter), and clustering algorithms (such as kMeans, which is not discussed in this book).

Another point: the section pertaining to activation functions does involve a basic understanding of hidden layers in a neural network. Depending on your comfort level, you might benefit from reading some preparatory material before diving into this section (there are many articles available online).

What Is Classification?

Given a dataset that contains observations whose class membership is known, classification is the task of determining the class to which a new datapoint belongs. *Classes* refer to categories and are also called *targets* or *labels*. For example, spam detection in email service providers involves binary classification (only two classes). The `MNIST` dataset contains a set of images, where each image is a single digit, which means there are ten labels. Some applications in classification include credit approval, medical diagnosis, and target marketing.

What Are Classifiers?

In the previous chapter you learned that linear regression uses supervised learning in conjunction with numeric data: the goal is to train a model that can make numeric predictions (e.g., the price of stock tomorrow, the temperature of a system, its barometric pressure, and so forth). By contrast, classifiers use supervised learning in conjunction with various classes of data: the goal is to train a model that can make categorical predictions.

For instance, suppose that each row in a dataset is a specific wine, and each column pertains to a specific wine feature (tannin, acidity, and so forth). Suppose further that there are five classes of wine in the dataset: for simplicity, let's label them A, B, C, D, and E. Given a new data point, which is to say a new row of data, a classifier for this dataset attempts to determine the label for this wine.

Some of the classifiers in this chapter can perform categorical classification and make numeric predictions (i.e., they can be used for regression as well as classification).

Common Classifiers

Some of the most popular classifiers for machine learning are listed here (in no particular order):

- Linear classifiers
- kNN
- Logistic regression
- Decision trees
- Random forests
- SVMs
- Bayesian classifiers
- CNNs (deep learning)

Keep in mind that different classifiers have different advantages and disadvantages, which often involve a trade-off between complexity and accuracy, similar to algorithms in fields that are outside of AI.

In the case of deep learning, CNNs (Convolutional Neural Networks) perform image classification, which makes them classifiers (they can also be used for audio and text processing).

The next sections provide a brief description of these ML classifiers.

Binary vs MultiClass Classification

Binary classifiers work with datasets that have two classes, whereas multiclass classifiers (sometimes called multinomial classifiers) distinguish more than two classes. Random forest classifiers and naïve Bayes classifiers support multiple classes, whereas SVMs and linear classifiers can only be used as binary classifiers (but multi-class extensions for SVM exist).

In addition, there are techniques for multiclass classification that are based on binary classifiers: One-versus-All (OvA) and One-versus-One (OvO).

The OvA technique (also called one-versus-the-rest) involves multiple binary classifiers that are equal to the number of classes. For example, if a dataset has five classes, then OvA uses five binary classifiers, each of which detects one of the five classes. In order to classify a datapoint in this dataset, select the binary classifier that has output the highest score.

The OvO technique also involves multiple binary classifiers, but in this case a binary classifier is used to train on a pair of classes. For instance, if the classes are A, B, C, D, and E, then ten binary classifiers are required: one for A and B, one for A and C, one for A and D, and so forth, until the last binary classifier for D and E.

In general, if there are `n` classes, then `n*(n-1)/2` binary classifiers are required. Although the OvO technique requires considerably more binary classifiers (e.g., 190 are required for 20 classes) than the OvA technique (e.g., a mere 20 binary classifiers for 20 classes), the OvO technique has the advantage that each binary classifier is only trained on the portion of the dataset that pertains to its two chosen classes.

MultiLabel Classification

Multilabel classification involves assigning multiple labels to an instance from a dataset. Hence, multilabel classification generalizes multiclass classification (discussed in the previous section), where the latter involves assigning a single label to an instance belonging to a dataset that has multiple classes. An article involving multilabel classification that contains `Keras`-based code is here:

https://medium.com/@vijayabhaskar96/multi-label-image-classification-tutorial-with-keras-imagedatagenerator-cd541f8eaf24

You can also perform an online search for articles that involve SKLearn or PyTorch for multilabel classification tasks.

What Are Linear Classifiers?

A linear classifier separates a dataset into two classes. A linear classifier is a line for 2D points, a plane for 3D points, and a hyper plane (a generalization of a plane) for higher dimensional points.

Linear classifiers are often the fastest classifiers, so they are often used when the speed of classification is of high importance. Linear classifiers usually work well when the input vectors are sparse (i.e., mostly zero values) or when the number of dimensions is large.

What Is kNN?

The kNN (k Nearest Neighbor) algorithm is a classification algorithm. In brief, data points that are near each other are classified as belonging to the same class. When a new point is introduced, it's added to the class of the majority of its nearest neighbor. For example, suppose that k equals 3, and a new data point is introduced. Look at the class of its three nearest neighbors: let's say they are A, A, and B. Then by majority vote, the new data point is labeled as a data point of class A.

The kNN algorithm is essentially a heuristic and not a technique with complex mathematical underpinnings, and yet it's still an effective and useful algorithm.

Try the kNN algorithm if you want to use a simple algorithm or when you believe that the nature of your dataset is highly unstructured. The kNN algorithm can produce highly nonlinear decisions despite being very simple. You can use kNN in search applications where you are searching for similar items.

Measure similarity by creating a vector representation of the items, and then compare the vectors using an appropriate distance metric (such as Euclidean distance).

Some concrete examples of kNN search include searching for semantically similar documents.

How to Handle a Tie in kNN

An odd value for k is less likely to result in a tie vote, but it's not impossible. For example, suppose that k equals 7, and when a new data point is introduced, its seven nearest neighbors belong to the set {A,B,A,B,A,B,C}. As you can see, there is no majority vote, because there are 3 points in class A, 3 points in class B, and 1 point in class C.

There are several techniques for handling a tie in kNN:

- Assign higher weights to closer points
- Increase the value of k until a winner is determined
- Decrease the value of k until a winner is determined
- Randomly select one class

If you reduce k until it equals 1, it's still possible to have a tie vote: there might be two points that are equally distant from the new point, so you need a mechanism for deciding which of those two points to select as the 1-neighbor.

If there is a tie between classes A and B, then randomly select either class A or class B. Another variant is to keep track of the tie votes, and alternate round-robin style to ensure a more even distribution.

What Are Decision Trees?

Decision trees are another type of classification algorithm that involves a treelike structure. In a generic tree, the placement of a data point is determined by simple conditional logic. As a simple illustration, suppose that a dataset contains a set of numbers that represent ages of people, and let's also suppose that the first number is 50. This number is chosen as the root of the tree, and all numbers that are smaller than 50 are added on the left branch of the tree, whereas all numbers that are greater than 50 are added on the right branch of the tree.

For example, suppose we have the sequence of numbers is {50, 25, 70, 40}. Then we can construct a tree as follows: 50 is the root node; 25 is the left child of 50; 70 is the right child of 50; and 40 is the right child of 20. Each additional numeric value that we add to this dataset is processed to determine which direction to proceed (left or right) at each node in the tree.

Listing 3.1 displays the contents of `sklearn_tree2.py` that defines a set of 2D points in the Euclidean plane, along with their labels, and then predicts the label (i.e., the class) of several other 2D points in the Euclidean plane.

Listing 3.1: sklearn_tree2.py

```
from sklearn import tree

# X = pairs of 2D points and Y = the class of each
   point
X = [[0, 0], [1, 1], [2,2]]
Y = [0, 1, 1]

tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, Y)

#predict the class of samples:
print("predict class of [-1., -1.]:")
print(tree_clf.predict([[-1., -1.]]))

print("predict class of [2., 2.]:")
print(tree_clf.predict([[2., 2.]]))

# the percentage of training samples of the same
   class
# in a leaf note equals the probability of each
   class
print("probability of each class in [2.,2.]:")
print(tree_clf.predict_proba([[2., 2.]]))
```

Listing 3.1 imports the tree class from `sklearn` and then initializes the arrays `X` and `y` with data values. Next, the variable `tree_clf` is initialized as an instance of the `DecisionTreeClassifier` class, after which it is trained by invoking the `fit()` method with the values of `X` and `y`.

Now launch the code in Listing 3.3 and you will see the following output:

```
predict class of [-1., -1.]:
[0]
predict class of [2., 2.]:
[1]
probability of each class in [2.,2.]:
[[0. 1.]]
```

As you can see, the points [-1,-1] and [2,2] are correctly labeled with the values 0 and 1, respectively, which is probably what you expected.

Listing 3.2 displays the contents of `sklearn_tree3.py` that extends the code in Listing 3.1 by adding a third label, and also by predicting the label of three points instead of two points in the Euclidean plane (the modifications are shown in bold).

Listing 3.2: sklearn_tree3.py

```
from sklearn import tree

# X = pairs of 2D points and Y = the class of each
   point
X = [[0, 0], [1, 1], [2,2]]
Y = [0, 1, 2]

tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, Y)

#predict the class of samples:
print("predict class of [-1., -1.]:")
print(tree_clf.predict([[-1., -1.]]))

print("predict class of [0.8, 0.8]:")
print(tree_clf.predict([[0.8, 0.8]]))

print("predict class of [2., 2.]:")
print(tree_clf.predict([[2., 2.]]))

# the percentage of training samples of the same
   class
# in a leaf note equals the probability of each
   class
print("probability of each class in [2.,2.]:")
print(tree_clf.predict_proba([[2., 2.]]))
```

Now launch the code in Listing 3.2 and you will see the following output:

```
predict class of [-1., -1.]:
[0]
predict class of [0.8, 0.8]:
[1]
predict class of [2., 2.]:
[2]
probability of each class in [2.,2.]:
[[0. 0. 1.]]
```

As you can see, the points [-1,-1], [0.8, 0.8], and [2,2] are correctly labeled with the values 0, 1, and 2, respectively, which again is probably what you expected.

Listing 3.3 displays a portion of the dataset `partial_wine.csv`, which contains two features and a label column (there are three classes). The total row count for this dataset is 178.

Listing 3.3: partial_wine.csv

```
Alcohol, Malic acid, class
14.23,1.71,1
13.2,1.78,1
13.16,2.36,1
14.37,1.95,1
13.24,2.59,1
14.2,1.76,1
```

Listing 3.4 displays contents of `tree_classifier.py` that uses a decision tree in order to train a model on the dataset `partial_wine.csv`.

Listing 3.4: tree_classifier.py

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('partial_wine.csv')
```

(Continued)

```
X = dataset.iloc[:, [0, 1]].values
y = dataset.iloc[:, 2].values

# split the dataset into a training set and a test set
from sklearn.model_selection import train_test_
   split
X_train, X_test, y_train, y_test = train_test_
   split(X, y, test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# ====> INSERT YOUR CLASSIFIER CODE HERE <====
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy
   ',random_state=0)
classifier.fit(X_train, y_train)
# ====> INSERT YOUR CLASSIFIER CODE HERE <====

# predict the test set results
y_pred = classifier.predict(X_test)

# generate the confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)
```

Listing 3.4 contains some `import` statements and then populates the `Pandas` DataFrame `dataset` with the contents of the CSV file `partial_wine.csv`. Next, the variable `X` is initialized with the first two columns (and all the rows) of dataset, and the variable `y` is initialized with the third column (and all the rows) of dataset.

Next, the variables `X_train, X_test, y_train, y_test` are populated with data from `X` and `y` using a 75/25 split proportion. Notice that the variable `sc` (which is an instance of the `StandardScalar` class) performs a scaling operation on the variables `X_train` and `X_test`.

The code block shown in bold in Listing 3.4 is where we create an instance of the `DecisionTreeClassifier` class and then train the instance with the data in the variables `X_train` and `X_test`.

The next portion of Listing 3.4 populates the variable `y_pred` with a set of predictions that are generated from the data in the `X_test` variable. The last portion of Listing 3.4 creates a confusion matrix based on the data in `y_test` and the predicted data in `y_pred`.

Remember that all the diagonal elements of a confusion matrix are correct predictions (such as true positive and true negative); all the other cells contain a numeric value that specifies the number of predictions that are incorrect (such as false positive and false negative).

Now launch the code in Listing 3.4 and you will see the following output for the confusion matrix in which there are thirty-six correct predictions and nine incorrect predictions (with an accuracy of 80%):

```
confusion matrix:
[[13  1  2]
 [ 0 17  4]
 [ 1  1  6]]
from sklearn.metrics import confusion_matrix
```

There is a total of forty-five entries in the preceding 3x3 matrix, and the diagonal entries are correctly identified labels. Hence the accuracy is 36/45 = 0.80.

What Are Random Forests?

Random Forests are a generalization of decision trees: this classification algorithm involves multiple trees (the number is specified by you). If the data involves making a numeric prediction, the average of the predictions of the trees is computed. If the data involves a categorical prediction, the mode of the predictions of the trees is determined.

By way of analogy, random forests operate in a manner similar to financial portfolio diversification: the goal is to balance the losses with higher gains. Random forests use a majority vote to make predictions, which operates under the assumption that selecting the majority vote is more likely to be correct (and more often) than any individual prediction from a single tree.

You can easily modify the code in Listing 3.4 to use a random forest by replacing the two lines shown in bold with the following code:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10,
   criterion='entropy', random_state = 0)
```

Make this code change, launch the code, and examine the confusion matrix to compare its accuracy with the accuracy of the decision tree in Listing 3.4.

What Are SVMs?

Support Vector Machines involve a supervised ML algorithm and can be used for classification or regression problems. SVM can work with nonlinearly separable data as well as linearly separable data. SVM uses a technique called the *kernel trick* to transform data and then finds an optimal boundary the transform involves higher dimensionality. This technique results in a separation of the transformed data, after which it's possible to find a hyperplane that separates the data into two classes.

SVMs are more common in classification tasks than regression tasks. Some use cases for SVMs include:

- Text classification tasks: category assignment
- Detecting spam/sentiment analysis
- Image recognition: aspect-based recognition, color-based classification
- Handwritten digit recognition (postal automation)

Tradeoffs of SVMs

Although SVMs are extremely powerful, there are tradeoffs involved. Some of the advantages of SVMs are:

- has high accuracy
- works well on smaller cleaner datasets

- can be more efficient because it uses a subset of training points
- can be an alternative to CNNs in cases of limited datasets
- captures more complex relationships between data points

Despite the power of SVMS, there are some disadvantages of SVMs:

- not suited to larger datasets: training time can be high
- less effective on noisier datasets with overlapping classes

SVMs involve more parameters than decision trees and random forests

Suggestion: modify Listing 3.4 to use an SVM by replacing the two lines shown in bold with the following two lines shown in bold:

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear',
            random_state = 0)
```

You now have an SVM-based model, simply by making the previous code update! Make the code change, then launch the code and examine the confusion matrix in order to compare its accuracy with the accuracy of the decision tree model and the random forest model earlier in this chapter.

What Is Bayesian Inference?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes' theorem to update the probability for a hypothesis as more information becomes available. Bayesian inference is often called *Bayesian probability*, and it's important in dynamic analysis of sequential data.

Bayes Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

```
P(A) = probability of being in set A
P(B) = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)
```

Then the following formulas are also true:

```
P(A|B) = P(Both)/P(B) (#1)
P(B|A) = P(Both)/P(A) (#2)
```

Multiply the preceding pair of equations by the term that appears in the denominator and we get these equations:

```
P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)
```

Now set the left-side of equations #3 and #4 equal to each another and that gives us this equation:

```
P(B)*P(A|B) = P(A)*P(B|A) (#5)
```

Divide both sides of #5 by P(B) and we get this well-known equation:

```
P(A|B) = P(A)*P(A|B)/P(B) (#6)
```

Some Bayesian Terminology,

In the previous section, we derived the following relationship:

```
P(h|d) = (P(d|h) * P(h)) / P(d)
```

There is a name for each of the four terms in the preceding equation:

First, the *posterior probability* is `P(h|d)`, which is the probability of hypothesis `h` given the data `d`.

Second, `P(d|h)` is the probability of data `d` given that the hypothesis `h` was true.

Third, the *prior probability* of `h` is `P(h)`, which is the probability of hypothesis `h` being true (regardless of the data).

Finally, `P(d)` is the probability of the data (regardless of the hypothesis)

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(d) and P(d|h).

What Is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

```
MAP(h) = max(P(h|d))
```

or:

```
MAP(h) = max((P(d|h) * P(h)) / P(d))
```

or:

```
MAP(h) = max(P(d|h) * P(h))
```

Why Use Bayes' Theorem?

Bayes Theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes rule.

What Is a Bayesian Classifier?

A Naive Bayes Classifier is a probabilistic classifier inspired by the Bayes theorem. An NB classifier assumes the attributes are conditionally independent and it works well even when assumption is not true. This assumption greatly reduces computational cost, and it's a simple algorithm to implement that only requires linear time. Moreover, a NB classifier easily scalable to larger datasets and good results are obtained in most cases. Other advantages of a NB classifier include that it:

- can be used for Binary & Multiclass classification
- provides different types of NB algorithms
- is good choice for Text Classification problems
- is a popular choice for spam email classification
- can be easily trained on small datasets

As you can probably surmise, NB classifiers do have some disadvantages, such as:

- All features are assumed unrelated
- It cannot learn relationships between features
- It can suffer from the "zero probability problem"

The "zero probability problem" refers to the case when the conditional probability is zero for an attribute, it fails to give a valid prediction. However, can be fixed explicitly using a Laplacian estimator.

Types of Naïve Bayes Classifiers

There are three major types of NB classifiers:

- Gaussian Naive Bayes
- MultinomialNB Naive Bayes
- Bernoulli Naive Bayes

Details of these classifiers are beyond the scope of this chapter, but you can perform an online search for more information.

Training Classifiers

Some common techniques for training classifiers are:

- Holdout method
- k-fold cross-validation

The *holdout method* is the most common method, which starts by dividing the dataset into two partitions called *train* and *test* (80% and 20%, respectively). The train set is used for training the model, and the test data tests its predictive power.

The *k-fold cross-validation* technique is used to verify that the model is not over-fitted. The dataset is randomly partitioned into k mutually exclusive subsets, where each partition has equal size. One partition is for testing and the other partitions are for training. Iterate throughout the whole of the k folds.

Evaluating Classifiers

Whenever you select a classifier for a dataset, it's obviously important to evaluate the accuracy of that classifier. Some common techniques for evaluating classifiers are:

- Precision and Recall
- ROC curve (Receiver Operating Characteristics)

Precision and recall are discussed in Chapter 2 and reproduced here for your convenience. Let's define the following variables:

```
TP = the number of true positive results
FP = the number of false positive results
TN = the number of true negative results
FN = the number of false negative results
```

Then the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
recall    = TP/[TP + FN]
```

The *ROC curve (Receiver Operating Characteristics)* is used for visual comparison of classification models that shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve is a measure of the accuracy of the model. When a model is closer to the diagonal, it is less accurate, and the model with perfect accuracy will have an area of 1.0.

The ROC curve plots True Positive Rate versus False Positive Rate. Another type of curve is the PR curve that plots Precision versus Recall. When dealing with highly skewed datasets (strong class imbalance), Precision-Recall (PR) curves give better results.

Later in this chapter you will see many of the `Keras`-based classes (located in the `tf.keras.metrics` namespace) that correspond to common statistical terms, which includes some of the terms in this section.

This concludes the portion of the chapter pertaining to statistical terms and techniques for measuring the validity of a dataset. Now let's look at activation functions in machine learning.

What Are Activation Functions?

A one-sentence description: an activation function is (usually) a nonlinear function that introduces nonlinearity into a neural network, thereby preventing a "consolidation" of the hidden layers in neural network. Specifically, suppose that every pair of adjacent layers in a neural network involves just a matrix transformation and no activation function. *Such a network is a linear system, which means that its layers can be consolidated into a much smaller system.*

First, the weights of the edges that connect the input layer with the first hidden layer can be represented by a matrix: let's call it `W1`. Next, the weights of the edges that connect the first hidden layer with the second hidden layer can also be represented by a matrix: let's call it `W2`. Repeat this process until we reach the edges that connect the final hidden layer with the output layer: let's call this matrix `Wk`. Since we do not have an activation function, we can simply multiply the matrices `W1`, `W2`, …, `Wk` together and produce one matrix: let's call it `W`. We have now replaced the original neural network with an equivalent neural network that contains one input layer, a single matrix of weights `W`, and an output layer. In other words, we no longer have our original multi-layered neural network!

Fortunately, we can prevent the previous scenario from happening when we specify an activation function between every pair of adjacent layers. In other words, *an activation function at each layer prevents this "matrix consolidation."* Hence, we can maintain all the intermediate hidden layers during the process of training the neural network.

For simplicity, let's assume that we have the same activation function between every pair of adjacent layers (we'll remove this assumption shortly). The process for using an activation function in a neural network is a *two step*, described as follows:

- Step 1. Start with an input vector `x1` of numbers
- Step 2. Multiply `x1` by the matrix of weights W1 that represents the edges that connect the input layer with the first hidden layer: the result is a new vector `x2`
- Step 3. Apply the activation function to each element of `x2` to create another vector `x3`

Now repeat steps 2 and 3, except that we use the starting vector x3 and the weights matrix `W2` for the edges that connect the first hidden layer with the second hidden layer (or just the output layer if there is only one hidden layer).

After completing the preceding process, we have preserved the neural network, which means that it can be trained on a dataset. One other thing: instead of using the same activation function at each step, you can replace each activation function by a different activation function (the choice is yours).

Why Do We Need Activation Functions?

The previous section outlines the process for transforming an input vector from the input layer and then through the hidden layers until it reaches the output layer. The purpose of activation functions in neural networks is vitally important, so it's worth repeating here: activation functions "maintain" the structure of neural networks and prevent them from being reduced to an input layer and an output layer. In other words, if we specify a nonlinear activation function between every pair of consecutive layers, then the neural network cannot be replaced with a neural network that contains fewer layers unless you explicitly remove them.

Without a nonlinear activation function, we simply multiply a weight matrix for a given pair of consecutive layers with the output vector that is produced from the previous pair of consecutive layers. We repeat this simple multiplication until we reach the output layer of the neural network. After reaching the output layer, we have effectively replaced multiple matrices with a single matrix that "connects" the input layer with the output layer.

How Do Activation Functions Work?

If this is the first time you have encountered the concept of an activation function, it's probably confusing, so here's an analogy that might be helpful. Suppose you're driving your car late at night and there's nobody else on the highway. You can drive at a constant speed for as long as there are no obstacles (stop signs, traffic lights, and so forth). On the other hand, suppose you drive into the parking lot of a large grocery store. When you approach a speed bump you must slow down, cross the speed bump, and increase speed again, and repeat this process for every speed bump.