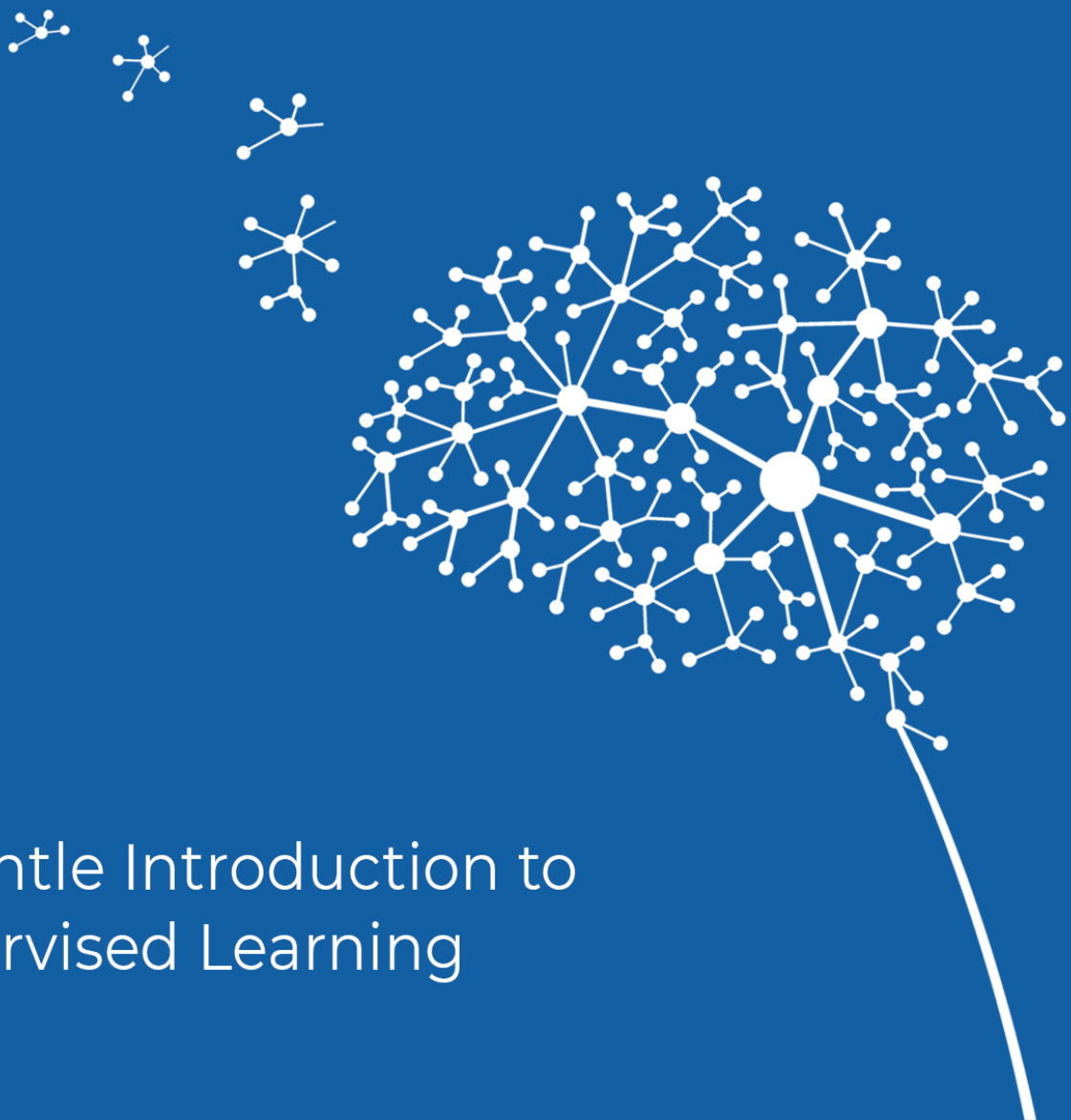
ANDREW WOLF

MACHINE LEARNING SIMPLIFIED



A Gentle Introduction to Supervised Learning

MACHINE LEARNING SIMPLIFIED: A GENTLE INTRODUCTION TO SUPERVISED LEARNING
ANDREW WOLF

THEMLSBOOK.COM
GITHUB.COM/5X12/THEMLSBOOK

LICENSE
1.0.0
First release, January 2022

Contents

PREFACE

It could be said that machine learning is my life. I am a machine learning engineer by day and an enthusiastic STEM tutor by night. I am consistently inspired by this infinitely exciting field and it has become one of my greatest passions. My interest in the machine learning dates back to 2012 when I came across an article describing a machine learning experiment conducted by the Google Brain team. The team, led by Andrew Ng and Jeff Dean, created a neural network that learned to recognize cats by watching images taken from frames of YouTube videos. I began to consider the possibilities and I was hooked.

Why I Wrote This Book

I, for one, eagerly look forward to a future in which ML will blossom and reveal its full potential. However, in my conversations with friends and colleagues outside the ML field, I've observed that they are often perplexed by the seeming complexity of it. Many of them are intrigued by the field and want to learn more, but find a dearth of clear, reliable resources on the internet. Sources are either rife with academic trilogies filled with theorems designed for experienced researchers and professionals (I couldn't even get through half of one) or are sprinkled with fishy fairy tales about artificial intelligence, data-science magic, and jobs of the future.

This book is dedicated to them — and thousands more - who want to truly understand the methods and use cases of ML both from conceptual and mathematical points of view, but who may not have the luxury of time, which is required to comb through thousands of hours of technical literature, full of intimidating formulas and academic jargon.

What This Book Is About

My goal for this book is to help make machine learning available to as many people as possible whether technical or not. It is easily accessible for a non-technical reader, but also contains way enough mathematical detail to serve as an introduction to machine learning for a technical reader. Nevertheless, some prior knowledge of mathematics, statistics and the Python programming language is recommended to get the most out of this book.

I've done my best to make this book both comprehensive and fun to read — mind you, that's no easy feat! I've worked to combine mathematical rigor with simple, intuitive explanations based on examples from our everyday lives. For example, deciding what to do over the weekend, or guessing a friend's favorite color based on something like their height and weight (I am only half-kidding

here). You will find answers to these questions and many more as you read this book.

How to Use This Book

This book is divided into two parts. Part I discusses the fundamentals of (supervised) machine learning, and Part II discusses more advanced machine learning algorithms. I divided the book in this way for a very important reason. One mistake many students make is to jump right into the algorithms (often after hearing one of their names, like Support Vector Machines) without a proper foundation. In doing so, they often fail to understand, or misunderstand, the algorithms. Some of these students get frustrated and quit after this experience. In writing this book, I assumed the chapters would be read sequentially. The book has a specific story line and most explanations appear in the text only once to avoid redundancy.

I have also supplemented this book with a GitHub repository that contains python implementations of concepts explained in the book. For more information, scan the QR code located in the 'Try It Now' box at the end of each chapter, or just go directly to *github.com/5x12/themlsbook*.

Final Words

Hopefully this book persuades you that machine learning is not the intimidating technology that it initially appears to be. Whatever your background and aspirations, you will find this book a useful introduction to this fascinating field.

Should you have any questions or suggestions, feel free to reach out to me at *contact@themlsbook.com*. I appreciate your feedback, and I hope that it will make the future editions of this book even more valuable.

Good luck in your machine learning journey,

Your author

Part I

FUNDAMENTALS OF SUPERVISED LEARNING

1. Introduction

Aims and Objectives

The aim of this chapter is to explain:

1. Machine Learning and its subfields: Supervised Learning, Unsupervised Learning, and Deep Learning.
2. Differences between Machine Learning, Data Science and MLOps.
3. Different types of AI and how they compose the high-level picture.

Learning Outcomes

By the end of this chapter, you will be able to understand:

- What kind of problems supervised (and unsupervised) ML can be used to solve.
- The difference between supervised ML and many other fields and terms, including deep learning, unsupervised learning, data science, MLOps, and artificial intelligence.

In this book you will learn about *supervised machine learning*, or supervised ML. Supervised ML is one of the most powerful and exciting fields in science right now and has many practical business applications. Broadly speaking, the goal of supervised ML is to make *predictions* about unknown quantities given known quantities, such as predicting a house's sale price based on its location and square footage, or predicting a fruit category given the fruit's width and height. Supervised ML does this by learning from, or discovering patterns in, past data, such as past house sale prices.

This chapter has two goals. The first is to give you a taste for the type of problems supervised ML can be used to solve. The second is to help you form a high-level mental map of the ML World. It will help you understand the difference between supervised ML and many other fields and terms that are sometimes (incorrectly!) used synonymously with it or are (correctly) used in similar or subtly different contexts. Some of these terms include: deep learning, unsupervised learning, data science, ML operations (or MLOps), and artificial intelligence. At the end of this chapter you should understand what supervised ML deals with and what these related fields deal with. This will help focus your attention on the right things as we explore supervised ML over the course of the book.

1.1 Machine Learning

The basic idea of machine learning, or ML, is to *learn* to do a certain task from data. Despite many popular sci-fi depictions, the learning process is not something magical or mysterious – at a high-level it can be understood as appropriately recognizing and extracting *patterns* from the data. There are two types of machine learning: supervised ML, which is the subject of this book and which we introduce in the next section, and unsupervised ML, which we will not discuss in this book but will give a high-level overview of it in the succeeding section.[1]

1.1.1 Supervised Learning

Many real-world problems have the following structure: given an object with a set of *known*, or observed, measurements, *predict* the value of an *unknown*, or target variable[2]. Simply put, supervised machine learning models try to predict either a **categorical** target variable (e.g., predicting a *fruit category* to be an apple, an orange, or a lemon, based on its weight and height), or a **numerical** target variable (e.g., predicting the *price* of an apartment, based on its area and location). Hence, there are two types of problems in the supervised machine learning domain: classification problems and regression problems.

In a **classification** problem we try to predict an unknown category, called a *class label* based on a set of known, or observed, variables. There is an endless number of classification problems: as long as a target variable is categorical, a set of classes could be anything. For instance, any yes-no question is a classification problem (see examples #1 - #4 below). These are solved with *binary classifiers*[3], since the target variable has only two possible outcomes. Classification problems with more than two possible outcomes (see example #5) are solved with *multiclass classifiers*. Both are discussed in this book. Some examples of classification problems include:

1. Is a new email spam or not? We use the text of that email to detect trigger words to decide if an email should be sent to the spam folder.
2. Will a user buy a product or not? We use information about the user, such as his or her age, gender, location, previous items bought, and other measurements to make this prediction.
3. Is tomorrow going to be rainy or not? We want to use measurements like humidity, wind speed, and cloudiness to make this prediction.
4. Is a credit card transaction legitimate or fraudulent? We want to use information about what items a person bought in the past, how much he usually spends, and other factors to determine this.
5. Is an unknown fruit an apple, a mandarin, or a lemon? We could use information about the fruit's properties, such as width and height.

In a **regression** problem we try to predict an unknown *number* based on a set of known, or observed, variables. Some example regression problems include:

1. What would be the *unknown* price of a house given a set of observed measurements about it such as its size, number of rooms, proximity to the ocean, crime rate in the neighbourhood and other measurements?

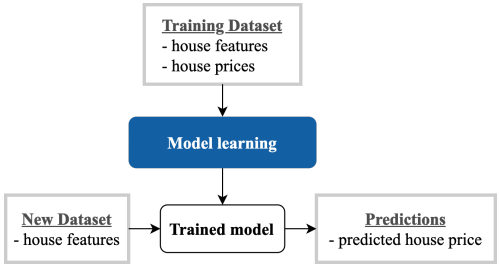
[1] For the sake of simplicity, hybrids such as semi-supervised and more complex methods like reinforcement learning were left out.

[2] The target variable *(in statistics: a dependent variable)* is the variable whose value is to be modeled and predicted by other variables.

[3] A classifier is another word for a classification model

	features			label / target variable
Criminal rate	Ocean proximity	Floor location	Floor size	**House price**
0.1	0.4 km	2	30 m^2	$310,000
0.6	2.2 km	4	46 m^2	$260,000
0.2	1.3 km	12	60 m^2	$800,000
0.7	0.1 km	2	45 m^2	$310,000

(a) Training dataset



(b) Supervised Learning Algorithm

Figure 1.1: A high-level overview of supervised ML for housing price predictions example. (a) Each row in the table on the left represents measurements of a single house. The first four columns are observed *features*, such as crime rate and ocean proximity, while the last column is the target variable, the house price. (b) The ML model *learns* to predict house prices from the training dataset which contains both the features and the price of houses (it does this by finding patterns in the data, e.g., that high crime rate lowers price). The trained model is then used to predict the *unknown price* of a *new* house given its features (crime rate, ocean proximity, etc.), as shown in the block diagram on the right.

2. How many customers will come to our restaurant next Saturday? This can be useful for making sure we have the right amount of food and workers on staff. We want to make this prediction based on factors like the number of guests that came last weekend, the number of guests each day this week, the expected weather this weekend, and if there is a holiday or special event.

Both classification and regression models help us solve numerous business problems in everyday life. However, how can we make a model that automatically makes the desired predictions (as accurately as possible)? At first it may seem that there are some rules that we can encode into the model - for example that houses by the ocean have higher value. But usually we will not know any simple rules in advance nor know exactly how the rules interact with each other, either reinforcing one another or cancelling each other out – exactly *how much* does proximity to the ocean increase the house's value? If the house is in a dangerous neighborhood does that decrease its value? What if it is in a dangerous neighborhood but also close to the ocean? As we can start to see, the number of interactions among variables grows very fast. In real problems where are thousands, millions, or even more variables, we can't possibly understand how they all interact.

The goal of **supervised machine learning**, or supervised ML, is to automatically *learn* the correct rules and how they interact with each other. How is this possible? It does so by learning from the provided *dataset*: In supervised ML we assume that we are given a dataset (e.g., a table in Figure 1.1a) consisting of *both* the measurements of an object (e.g., house features like floor location, floor size, criminal rate, etc.) *and* the answer to the desired label (house price). (This final piece of information, the answer, is precisely what is meant by "supervised" learning.) Supervised ML learns the relationships between different measurements and the label. After the model is learned (or trained), we can then use it to *predict* the label of a new (unseen) data point from its known measurements (but totally *unknown* label). Coming back to the housing example, the model would first learn from the dataset how different measurements affect the house price. Then, it will be able to predict the unknown price of a new house based entirely on its measurements (shown in Figure

1.1b). This prediction may be very accurate or very inaccurate. The accuracy depends on many factors, such as the difficulty of the problem, quality of the dataset, and quality of the ML algorithm. We will discuss these factors in depth throughout this book.

Definition 1.1.1 — Supervised Machine Learning Algorithm. A supervised learning algorithm is an algorithm that learns rules and how they interact with each other from the labelled dataset in order to perform regression or classification tasks.

Supervised ML may still seem a bit abstract to you, but that's okay. In the next chapter we will dive into a more detailed introduction of supervised ML: we will precisely define terms used by supervised ML, show you how to represent the learning problem as a math problem, and illustrate these concepts on a simple dataset and prediction task. The rest of this book will expand upon this basic framework to explore many different supervised ML algorithms.

Deep Learning

Deep learning is a term that you have probably heard of before. Deep learning models are very complex supervised ML models[4] that perform very complicated tasks in areas where more advanced or faster analysis is required and traditional ML fails. They have produced impressive results on a number of problems recently, such as image recognition tasks. We will not discuss deep learning in this book, due to its complexity. But, at a high level, a deep neural network (what is usually meant by the term 'deep learning') are so powerful because it is related to **neural network algorithms** - specific algorithms designed to model and imitate how the human brain thinks and senses information. After mastering the basics of supervised learning algorithms presented in this book, you will have a good foundation for your study of deep neural networks later. I recommend *deeplearning.ai* as a good source for beginners on deep learning.

It should be noted, before we continue, that although deep learning models are very complex and have produced great results on some problems, they are not universally better than other ML methods, despite how they are sometimes portrayed. Deep learning models often require huge amounts of training data, huge computational cost, a lot of tinkering, and have several other downsides that make them inappropriate for many problems. In many cases, much simpler and more traditional ML models, such as the ones you will learn about in this book, actually perform better.

1.1.2 Unsupervised Learning

This book will focus on supervised machine learning. But there is another important subfield of machine learning, known as **unsupervised learning**, that you should be aware of. In supervised ML, the human provides the target labels (such as the house price in Table 1.1a) which the machine tries to learn to imitate. The process of generating and supplying the machine with these labels is said to *supervise* the learning process. *Unsupervised learning* is a subfield of ML that solves a complementary set of problems (to supervised ML) that do not require human input labels. Unlike supervised ML that learns rules from data to predict the unknown value of an object, the goal of unsupervised ML is to learn and generate distinctive *groups* or *clusters* of data points in the dataset.

Market segmentation analysis is a practical example of where unsupervised learning methods are useful. Consider, for example, the management team of a large shopping mall that would like to understand the types of people who are visiting their mall. They know that there are a few different

[4] You should also be aware that there are deep learning methods for unsupervised learning problems too.

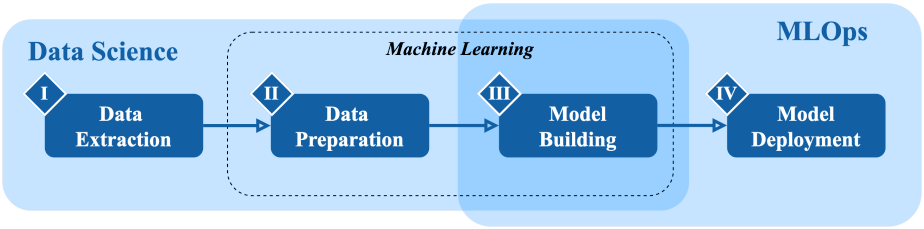


Figure 1.2: Typical pipeline for a supervised machine learning problem. Steps I and II are responsible for extracting and preparing the data, Step III is responsible for building the machine learning model (most of the rest of this book will detail the steps in this process), and Step IV is responsible for deploying the model. In a small project, one person can probably perform all tasks. However, in a large project, Steps I and II may be performed by a data science specialist, Steps III may be performed by either a data scientist or machine learning operations, or MLOps, engineer, and Step IV may be performed by an MLOps engineer.

market segments, and they are considering designing and positioning the shopping mall services to target a few profitable market segments or to differentiate their services (e.g., invitations to events or discounts) across market segments. This scenario uses unsupervised ML algorithms to cluster a dataset of surveys that describes attitudes of people about shopping in the mall.[1-1] In this example, supervised learning would be useless since it requires a target variable that we want to predict. Because we are trying to understand what people want instead of predicting their actions, we use unsupervised learning to categorize their opinions.

After completing this book on supervised ML, you are encouraged to learn about unsupervised ML. As you will find in your studies, many of the key concepts, ideas, and building blocks of algorithms that we will learn about in this book in the supervised setting have similar analogues in the unsupervised setting. In other words, mastering supervised ML will make it much easier for you to master unsupervised ML as well. One last note: I've left some more information on unsupervised learning in Appendix A - feel free to read it through.

1.2 Machine Learning Pipeline

As we saw in the last section, the goal of supervised ML is to learn how to make a prediction. This is usually treated as a "clean" problem, as if your data and everything is set up in an "ideal" way. However, the real world is never so ideal. To perform ML in the real world, we often require a few sequential stages, forming a pipeline, as depicted in Figure 1.2. To understand this pipeline and the real-world issues that arise, consider the following example.

Imagine an online store "E Corp" that sells hardware for computers. E Corp wants to understand their customers' shopping habits to figure out which customer is likely to buy which product. They then want to build a system that predicts how best to market to each user. They have hired you to build such a system. How should you proceed? You brainstorm the following plan, depicted in the block-diagram in Figure 1.2:

I. First, you have to obtain or **extract** data that will be used for algorithm training. The initial step is to collect, historical data of E Corp on their past customers over a certain period of time.

II. Second, you have to **prepare** the dataset for your ML model. One of the preparation procedures can be to check the quality of collected data. In most real-world problems, the data collected is extremely "dirty". For example, the data might contain values that are out of range (e.g., salary data listing negative income) or have impossible data combinations (e.g., medical data listing a male as pregnant). Sometimes this is a result of human error in entering data. Training the model with data that hasn't been carefully screened for these or many other problems will produce an inaccurate and misleading model. Another preparation procedure might be to use statistical and ML techniques to artificially increase the size of the dataset so that the model would have more data to learn from. (We will cover all preparation procedures in Chapter 7.)

III. Next, you **build** your ML model - you feed the prepared dataset into your ML algorithm to train it. At the end of training, you have a model that can predict the probability of a specific person purchasing an E Corp product based on their individual parameters. (Don't worry if you don't fully understand what the term "training" means just yet – this will be a huge subject of the rest of the book!)

IV. Finally, you **deploy** your trained and (presumably) working model on a client's computer or cloud computing service as an easy-to-use application. The application may be a nice user interface that interacts with the ML model.

If E-Corp is a small company with a small dataset, you could perform all of these steps yourself. However, for much bigger ML problems you will probably need to separate these steps into parts which can be performed by different people with different expertise. Usually, a *data scientist* will be responsible for steps I and II, and an *ML operations engineer*, or MLOps engineer, will be responsible for step IV. Step III is an overlap, and can be performed by both a Data scientist and an MLOps engineer. (In my personal experience quite often it is the former one.) In the next two sections we discuss data science and MLOps in more detail. Our goal is to give you an overview of these fields, and make sure you understand how they differ. Many times all of these terms are used synonymously, even in job descriptions, and this tends to confuse students of machine learning.

1.2.1 Data Science

Traditionally, the data science domain is about obtaining and *analyzing* data from the *past* to extract insightful information. The analysis typically includes visualizing and interpreting data, and computing simple statistics (such as the median sales value in each month) on the data. The data analysts then sews these pieces together to tell a story about the data. The story can, for example, give business insights into its customers, such as which day of the week the most customers, on average, came to the store last year.

Nowadays, however, data science isn't concerned solely with interpreting past data. Data scientists now use machine learning to *prepare data* (step II of Figure 1.2) and *build algorithms* (step III) based on that data to *predict* what will happen in the *future*. (For example, *predict* the number of customers that will come to the store *next* Saturday.) Data preparation and model building are tightened together. As we will discuss in more detail later in the book, we cannot just start building an ML algorithm blindly from a new data set. We need to first clean the data and understand it to intelligently construct ML algorithms, to understand when and why they are failing, or to generate insightful features which are often critical to making ML algorithms work. Usually, data scientists are dealing with the following questions:

- Where will the data come from? How should we extract it in the most efficient way?

- How can we ensure data quality? What do we need to perform to ensure that we have a clean and ready-to-use dataset?
- Are different datasets or features scaled differently? Do we need to normalize them?
- How could we generate more insightful features from the data we have?
- What are the top three features that influence the target variable?
- How should we visualize the data? How could we tell the story to our client in the best way?
- What is the purpose of a model? What business problem can it solve?
- How do we plan to evaluate a trained model? What would be an acceptable model accuracy?
- Based on the data we have and the business problem we deal with, what algorithm should be selected? What cost function should be used for that algorithm?

1.2.2 ML Operations

The main goal of ML Operations, or MLOps, is getting models out of the lab and into production (step IV of Figure 1.2). Although MLOps engineers also build models, their main focus lies in the integration and deployment of those models on a client's IT infrastructure. An MLOps engineer typically deals with the following questions:

- How to move from Jupyter notebook (in Python programming environment) to production, i.e., to scale the prediction system to millions of users? Do we deploy on cloud (e.g., AWS, GCP, or Azure), prem, or hybrid infrastructure?
- What front-end, back-end, or data services should the product integrate with? How can we integrate our model with a web-based interface?
- What are the data drift and vicious feedback loops?[5] Will data in the real world behave just like data we used in the lab to build the ML model? How can we detect when the new data is behaving in an 'unexpected' way? How frequently should the data be refreshed?
- How to automate model retraining? What tools should we use for continuous integration (CI) or continuous deployment (CD)?
- How to version the code, model, data, and experiments?
- How do we monitor the application and infrastructure health? What unit tests do we have to perform before running the model or application? Do we have logging and other means of incidents investigation?
- How do we serve security and privacy requirements? Are there any security controls to take into account?

1.3 Artificial Intelligence

Supervised ML, the subject of this book, can be seen as a subfield of artificial intelligence, or AI. The goal of this section is to provide a high-level overview of what AI is so that you understand where supervised ML (and other aforementioned terms such as data science, machine learning or deep learning) fits in and how other AI problems are different. Broadly speaking, the goal of artificial intelligence, or AI, is to build computer systems that can perform tasks that typically require human intelligence. Listed below are some problems which AI has been used to solve:

1. One of the most basic examples is a smart elevators in a skyscraper building. The fact that they can learn to optimize the transportation of people to different floors based on the press of

[5] In practice, the data will often "drift"; for example, the patterns we learned on in the past 10 years may not hold next year.

the up and down buttons shows that it is an example of AI, even if it is an incredibly basic example.

2. The spam detector in your email needs to intelligently decide if a new email is spam email or real email. It makes this decision using information about the message, such as its sender, the presence of certain words or phrases, e.g., a message with the words 'subscribe' or 'buy' may be more likely to be spam. (Remember that this is an example of supervised ML that we saw earlier in this chapter.)
3. A much more complex intelligent system is a self-driving car. This requires many difficult AI tasks, such as detecting red lights, identifying pedestrians, and deciding how to interact with other cars and predict what other cars will do.

All these AI machines perform their tasks roughly based on a set of simple "if-then" rules:

- an elevator B will get you to the top floor if it is closer to you than elevators A and C.
- a spam detector sends an email to the spam folder if that email contains certain words such as "sale", "buy," or "subscribe".
- a self-driving car stops at an intersection if the stoplight is red.

However, these machines are different in complexity. So, what defines their complexity?

1.3.1 Information Processing

What makes a smart elevator a less complex form of AI than a self-driving car? The simple answer is the level of information that the machine processes to make a decision. You can actually organize the complexity of the three aforementioned examples of AI: A spam detector is more complex than a smart elevator, and a self-driving car is more complicated than a spam detector.

- A smart elevator processes *almost no information* to understand it is the closest one to you and needs to go your way - it takes into account (calculates) just two static variables - the direction and the distance between itself and a passenger.
- An e-mail spam detector processes *some information* every time a new e-mail is received – specifically, it parses/analyses the text of each email received – to detect trigger words to decide if an email should be sent to the spam folder.
- A self-driving car processes *a lot of information* every second to understand the direction and speed it needs to go, and if it needs to stop for a light, pedestrian, or other obstacle.

The level of information processing required for machine decision-making defines the complexity of AI. With more information processing comes more patterns that the machine needs to extract and understand. AI is categorized based on how complex it is. These categories are outlined in the next section.

1.3.2 Types of AI

Each AI task requires its own set of techniques to solve. Some researchers have categorized AI into four categories, roughly based on complexity:

Basic AI: Basic AI is the simplest form of AI. It does not form memories of past experience to influence present or future decisions. In other words, it does not *learn*, it only *reacts* to currently existing conditions. Basic AI often utilizes a predefined set of simple rules encoded by a human. An example of Basic AI is the smart elevator discussed above.

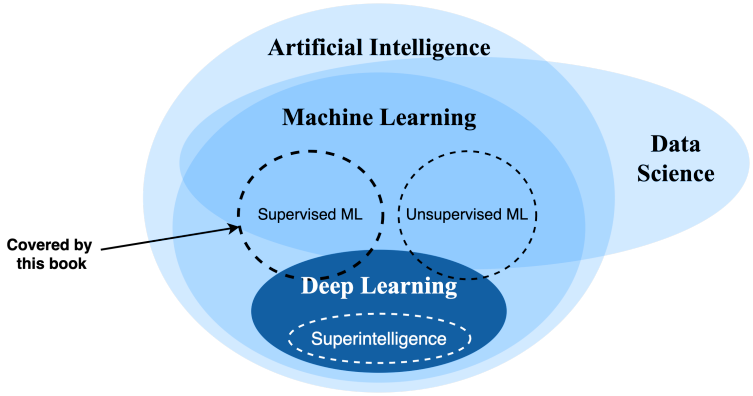


Figure 1.3: A "mind-map" diagram illustrating the relationship between different areas of artificial intelligence.

Limited AI: Limited AI systems require a substantial amount of information, or data, to make decisions. The advantage is they can make decisions without being explicitly programmed to do so. The email spam detector discussed above is an example of limited AI – the spam detector learns rules for predicting if an email is spam based on a dataset of emails that are known to be spam or not spam. (Usually a human being must provide these labels that the machine learns from.) Machine Learning is the main and most well-known example of Limited AI, and the email spam detector is an example of supervised ML, which we discuss in the next section.

Advanced AI: Advanced AI systems will possess the intelligence of human beings. They will be able to, for example, drive your car, recognize your face, have conversations with other human beings, understand their emotions, and perform any task that an intelligent human could do. These systems are often described as a computational "brain".

Superintelligence: Superintelligent AI systems will possess intelligence that far surpasses the abilities of any human being.

So far we haven't explained just how AI systems perform intelligent tasks. You may be wondering: Are these machines "actually" intelligent? As it turns out, this is a very deep philosophical question and one that we will not go into here. But we will note that the term "artificial intelligence" as it is used in society often gives an unrealistic, sci-fi, depiction of AI systems. However, many AI systems actually come down to solving carefully crafted math problems – and that's a large part of their beauty. You will see this process – the reduction of a problem requiring intelligence to a math problem – in the context of supervised ML throughout this book.

Supervised ML can actually be viewed as a subfield of AI. Figure 1.3 shows a nested set of circles representing the subfields of AI and how supervised ML fits into it.

1.4 Overview of this Book

This book thoroughly covers data preparation and model building blocks - steps II and III of Figure 1.2. The book is also divided into two parts. In Part I you will learn about the fundamentals of

machine learning, and in Part II you will learn the details about more complex machine learning algorithms.

I chose to split this book into these two parts for a very important reason. Many times people hear the names of certain powerful machine learning algorithms, like support vector machines, neural networks, or boosted decision trees, and become entranced by them. They try to jump into learning complex algorithms too quickly. As you can imagine, it is very difficult for them to learn complex models without learning simpler ones first. But also, as you are probably not aware of yet, most ML models encounter common problems and difficulties (such as overfitting) which you will learn about in the next chapter. Therefore, while it may be tempting to delve into specific algorithms right away, you should take time to get the high-level picture of the ML world as well as all the details of the fundamental principles it was built on (e.g. how an algorithm actually learns from data from a mathematical point of view). It's kind of like working on a puzzle with the picture on the cover; sure, you can finish the puzzle without the image, but it will take a lot longer, and you could make a lot of mistakes. Once you master the fundamentals of ML you will have a clear idea of these common problems and you will be able to understand how to avoid common pitfalls, regardless of which learning algorithm you use for a specific problem.

The next chapter gives an overview of the fundamentals of ML in the context of an example. At a high level, we will see a detailed sequence of steps that represents the model building stage (Step III) of the ML pipeline shown in Figure 1.2. After giving you a high level overview of the steps in ML model building, that chapter will also provide more details about how the rest of the book is laid out, chapter-by-chapter.

Key concepts

- Artificial Intelligence
- Supervised Learning, Unsupervised Learning, Deep Learning
- MLOps, Data Science

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Understand what supervised ML deals with and how it fits into the wider world of AI.
- Form a typical pipeline for a supervised machine learning problem.

2. Overview of Supervised Learning

Aims and Objectives

The aim of this chapter is to explain:

1. Supervised machine learning algorithm;
2. Decision boundaries, Loss function;
3. Train and Test sets;
4. Overfitting and underfitting;

Learning Outcomes

By the end of this chapter, you will be able to understand:

- How a typical supervised learning algorithm works;
- The concepts of overfitting and underfitting;
- Detailed pipeline for a full ML system;

In this chapter we will present an overview of *supervised machine learning* (ML). Supervised ML systems are fairly complicated but, for the most part, they all share the same basic building blocks. We begin this chapter by presenting an overview of supervised ML in the context of a simple example out of which these building blocks will emerge. We will then move from a specific to a general form, clearly articulating the building blocks. This general structure will serve as a roadmap for the rest of the book which will delve into the mathematical details and the various trade-offs and choices associated with each building block.

2.1 ML Pipeline: Example

In this section we give an overview of the supervised ML pipeline in the context of a simple example with a synthetic dataset shown in Table 2.1. The data table represents the height and width measurements of twenty pieces of fruit, recorded in the first two columns, along with the type of fruit, recorded in the last column. In the language of ML, the height and width are called the *input variables* or *features*, and the fruit type is called the *target variable* or *class label*. Eventually, our

goal will be to build an ML model that predicts a target variable – the type of a new fruit – based only on its features – its height and width measurements. But first, we present some mathematical notation and explore this dataset further.

2.1.1 Problem Representation

Mathematically, a data point with p features is represented by a vector $x = (x^{(1)}, \ldots x^{(p)})$ where the superscript $x^{(j)}$ represents the j^{th} feature. Usually, we have a large number $n > 1$ of data points. We distinguish the i^{th} data point with a subscript $x_i = (x_i^{(1)}, \ldots, x_i^{(p)})$, where $x_i^{(j)}$ is the j^{th} input feature of the i^{th} data point. In our example, there are $n = 20$ data points (pieces of fruit), and each data point has $p = 2$ features (width and height measurements). The i^{th} row of the table represents the width and height features of the i^{th} fruit. For example, the third data point, $i = 3$, has $x_3^{(1)} = 10.48$ as its first feature (height), $x_3^{(2)} = 7.32$ as its second feature (width), as can be seen from the third row of the table. The vector of all features for this datapoint is written as $x_3 = (x_3^{(1)}, x_3^{(2)}) = (10.48, 7.32)$.

We represent the output variable (also called a target variable) by a scalar y. For classification problems y takes one of a discrete set of values. For our fruit classification problem, the target variable takes a value representing the fruit type, that is, $y \in \{\texttt{Apple}, \texttt{Mandarin}, \texttt{Lemon}\}$. The output variable of the i^{th} datapoint is represented using the subscript y_i. For example, for $i = 3$, $y_i = \texttt{Lemon}$, as can be seen from the third row of the table.

Data Visualization

Since it is difficult for us humans to make sense of a raw data table, it is often useful to *visualize* the data. For a classification problem with two input features the data is often visualized with a *scatter plot*. A scatter plot for our example dataset is shown in Figure 2.1. In the scatter plot, the horizontal axis corresponds to the first feature and the vertical axis corresponds to the second feature. The i^{th} data point corresponds to a dot with coordinates equal to its feature values, $(x_i^{(1)}, x_i^{(2)})$, and colored according to its class label, y_i, where the colors associated with each class are designated by the legend in the top left corner of the figure. In our fruit classification example, the horizontal axis represents the width feature, the vertical axis represents the height feature, each fruit is a dot whose coordinates are its width and height, and the dot is colored red if the fruit is an apple, green if it is a mandarin, and blue if it is a lemon. For example, the third fruit, $i = 3$, corresponds to a point with coordinates $(10.48, 7.32)$ and colored blue (class `Lemon`) in the figure.

2.1.2 Learning a Prediction Function

How can we use this dataset to help predict the type of a new (unseen) piece of fruit? If I didn't tell you anything about the fruit, you would pretty much just be left making a wild guess. However, if I tell you that the fruit is 4 cm tall and 5 cm wide, just by eyeballing Figure 2.1 you might guess that the fruit is a mandarin, since many mandarins cluster around that point. The goal of supervised ML is to learn how to make predictions in a systematic manner.

Formally, supervised ML seeks to learn a *prediction function* f that maps each joint setting of the input variables features to a value of the target variable, $X \to Y$. In our fruit classification example, we must learn a function f that maps any combination of height and width of a fruit to a (predicted) fruit type. The specific choice of prediction function f is a fundamental problem in supervised ML – many of the specific algorithms you will learn about in this book, such as linear models, K-Nearest Neighbor models, Bayesian models, maximum margin models, and many others, correspond to

height (cm)	width (cm)	**Fruit Type**
3.91	5.76	mandarin
7.09	7.69	apple
10.48	7.32	lemon
9.21	7.20	lemon
7.95	5.90	lemon
7.62	7.51	apple
7.95	5.32	mandarin
4.69	6.19	mandarin
7.50	5.99	lemon
7.11	7.02	apple
4.15	5.60	mandarin
7.29	8.38	apple
8.49	6.52	lemon
7.44	7.89	apple
7.86	7.60	apple
3.93	6.12	apple
4.40	5.90	mandarin
5.5	4.5	lemon
8.10	6.15	lemon
8.69	5.82	lemon

Table 2.1: An example dataset with input measurements of twenty pieces of fruit (first two columns) along with the fruit's type (last column).

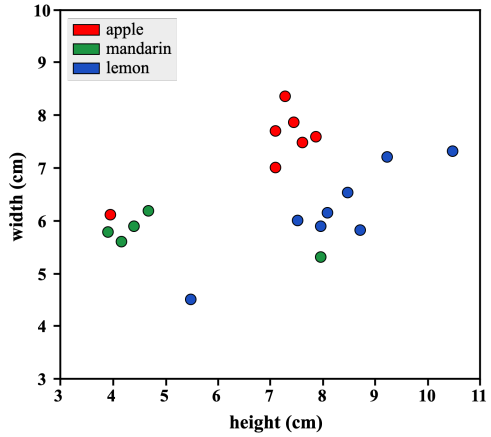


Figure 2.1: Scatter plot visualization of our example fruit dataset. The axes represent the two measured properties (height on the horizontal $x^{(1)}$-axis and width on the vertical $x^{(2)}$-axis). Each dot represents a fruit whose coordinates correspond to the two measurements and whose color corresponds to the fruit's type (color code specified by the legend at the top left of the figure).

choosing a different type of prediction function f. In this section we focus on the K-Nearest neighbor (KNN) classifier due to its simplicity.

KNN classifies a new point x by first finding the k points in the training set that are nearest to the point x. It then predicts that the (unknown) label of point x is equal to the most popular class among these nearest neighbors (remember that we know the class label of the neighbors since they are in our training set). k is a parameter that we have freedom to choose. If we choose $k = 1$, our predictor is especially simple: we find the closest point in the training set and predict that the new data point has the same class label as that training point.

How would the 1-Nearest neighbor classifier predict the type of a new fruit that is 4 cm tall and 5 cm wide? First, we find the piece of fruit in our labeled training dataset whose height and width are closest to the coordinates $(4cm,\ 6cm)$, and then guess that the unknown fruit is the same type as this nearest neighbor. From our dataset in Table 2.1, the nearest fruit to the $(4cm, 5cm)$ measurement is in the row $j = 1$ with width and height of $(3.91,\ 5.76)$, and has type `mandarin`. (You can check that this is the nearest neighbor by brute force: measure the distance to each data point and then find the smallest distance.) Thus, we would guess that the unknown fruit is a `mandarin`.

Visualizing Decision Regions

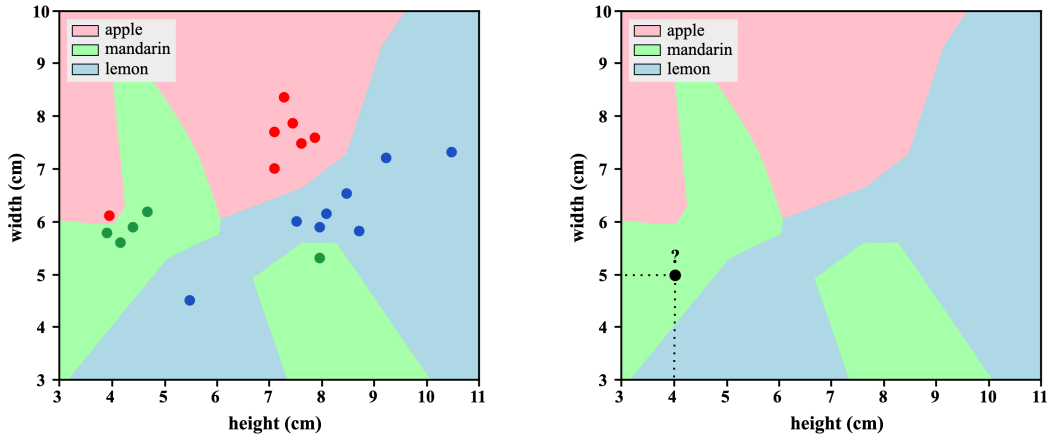
It is often difficult for us humans to understand what predictions the prediction function f will make from the feature values. To understand the predictions better, it is often useful to make a visualization. For many large-scale ML problems, with thousands or millions of features, visualization is a very difficult task. However, when the input features have only two dimensions, we can simply show how each point in the plane would be classified by f. This visualization is shown in Figure 2.2a for our fruit classification example. Each location in the plane, representing a height and width measurement, is given a color equal to the color of its nearest data point in our training set (dot in figure). Each region of the same color is called a *decision region* for each class (e.g., the green is the decision region for `mandarin`), and a curve that makes up the boundary between decision regions is called a *decision boundary* (e.g., the boundary between green and red).

With this visualization in hand, we can easily see what fruit type our learned classifier would predict given height and width measurements. For example, how would we classify our unlabeled test fruit that is 4cm tall and 5cm wide? We look at the decision regions and see that this point falls in the green regime, corresponding to the mandarin fruit type, and we predict `mandarin`. This is shown in Figure 2.2b.

Is the prediction correct? There is no way to know for certain without knowing the true type of this fruit. Based on what we saw in the training set with this clearly defined boundary, it seems likely that this new fruit follows the same pattern, but there is no guarantee. It is quite possible that the fruit's real type is "lemon" – albeit a lemon of a very different size than any lemon that we saw in the training data. A crucial part of any ML project is evaluating the quality of your classifier. We address this in the next section.

2.1.3 How Good is our Prediction Function?

Now, let's try to evaluate this classifier. We will start by defining a quantitative (numeric) metric that tells how good a classifier is on a dataset. We will then evaluate our classifier on the data we trained on, then on new, unseen data. We will discover a key problem, known as overfitting, which means



(a) Learned decision regions (and training data)

(b) Classify new point

Figure 2.2: Visualization of learned decision regions for a 1-Nearest neighbor classifier. Figure (a) shows the training data points along with the learned decision regions (colored areas). Figure (b) shows how this classifier predicts the (unknown) type of a new fruit with height and width measurements $(5cm, 4cm)$. This point lands in the green region so the classifier predicts `mandarin`.

that the model performs exceptionally well on training data, and thus it's an unrealistic estimate of real-world error.

Loss Function

First, we need to summarize how well our predictions match the true label with a single number. Let's compute the *misclassification rate* by counting the total number of misclassifications and dividing by the total number of data points:

$$L(f, X, Y) = \frac{1}{n} \cdot \sum_{i=1}^{n} [f(x_i) \neq y_i] \tag{2.1}$$

where f is the prediction function, $X = \{x_i : i = 1 \ldots n\}$ is a dataset of features, $Y = \{y_i : i = 1 \ldots n\}$ are the class labels. The square brackets $[\cdot]$ represent a value that is 1 if the argument inside is true and 0 if it is false. Here, its value is 1 if the prediction $f(x_i)$ does not equal the known fruit type y_i, i.e., the prediction is incorrect. The notation "L" is meant to signify that the misclassification rate is one type of *loss function*. I won't discuss loss functions in detail here, but will just tell you that they play an important role in ML as you will see later in this book.

Evaluate on Train Set

Now that we have a way to measure the performance of our learned prediction function f, let's see how good we do. What would happen if we predicted the type of each fruit in our training set from its height and width measurements using prediction function f (ignoring the known fruit type for now)? To make the prediction with f we can plot the points on the graph and look at the decision regions. This plot is shown in Figure 2.2a. What do we see? Each point is the same color (its true label) as the decision region it falls in (its predicted label). This means that the classifier makes every single prediction correctly! In other words, $f(x_i) = y_i$ and $[f(x_i) \neq y_i] = 0$ for each $i = 1, \ldots, n$, and, hence, our misclassification rate is zero $L(f, X, Y) = 0$ (since the definition in Eq. 2.1 adds n zeros

$height(cm)$	$width(cm)$	**Fruit Type**
4	6.5	mandarin
4.47	7.13	mandarin
6.49	7	apple
7.51	5.01	lemon
8.34	4.23	lemon

(a) Test Set

$height(cm)$	$width(cm)$	**Fruit Type**
4	6.5	?
4.47	7.13	?
6.49	7	?
7.51	5.01	?
8.34	4.23	?

(b) Withhold Test Label

Table 2.2: Table (a) shows the test set consisting of measurements and known fruit label. Table (b) illustrates a data set where we withhold the fruit type (from our learned model).

Actual	**Predicted**	**Error?**
mandarin	apple	1
mandarin	mandarin	0
apple	apple	0
lemon	mandarin	1
lemon	mandarin	1

Table 2.3: Table showing fruit's actual type y_i (first column), its predicted type $f(x_i)$ (second column), and its misclassification indicator $[y_i \neq f(x_i)]$ (third column).

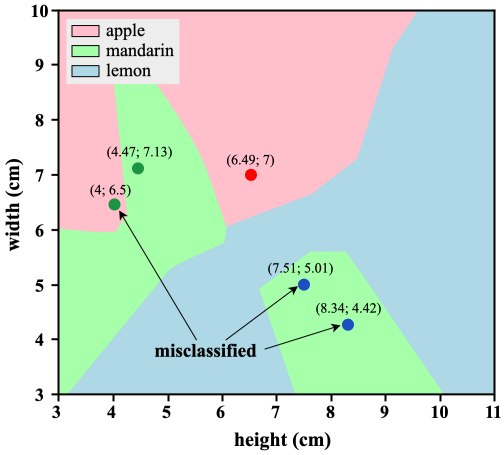


Figure 2.3: Predictions on the test set of the model f learned with $k = 1$ shown as decision regions in Figure 2.2b. Each test point is colored with its true type; its predicted type is the colored region it falls in.

together).

Evaluate on Test Set

You feel very confident in your fruit detector: you think you can perfectly predict a fruit's type based only on its measurements. Now suppose I told you that I have a new batch of fruit, with the width and height measurements of each fruit as well as the known fruit type (shown in Table 2.2a). But, I am going to withhold the fruit type from you and give you only the width and height measurements (shown in Table 2.2b). Next, I want you to use your classifier to predict the fruit's type. Let's see what happens. You take each data point in the table and compute the classification $f(x_i)$ – this is shown visually in Figure 2.3 by identifying the colored decision region each point falls in. You record these predictions in the first column of Table 2.3. Now, it is time for me to grade you. Using the true values of the fruit (which I know from Table 2.2a), I mark your answers as right or wrong and record it in the third column of Table 2.3. What is the result? Some of your answers are wrong, such as the first fruit, $x_1 = (4, 6.5)$, which is really a `Mandarin`, but your classifier predicted it was an `Apple` (since the point lands in the red region). All in all, you made five predictions, two were

correct, and three were incorrect. Your misclassification rate on the the new batch of fruit, called the *test set*, was $L(f, X_{test}, Y_{test}) = \frac{3}{5}$.

Overfitting

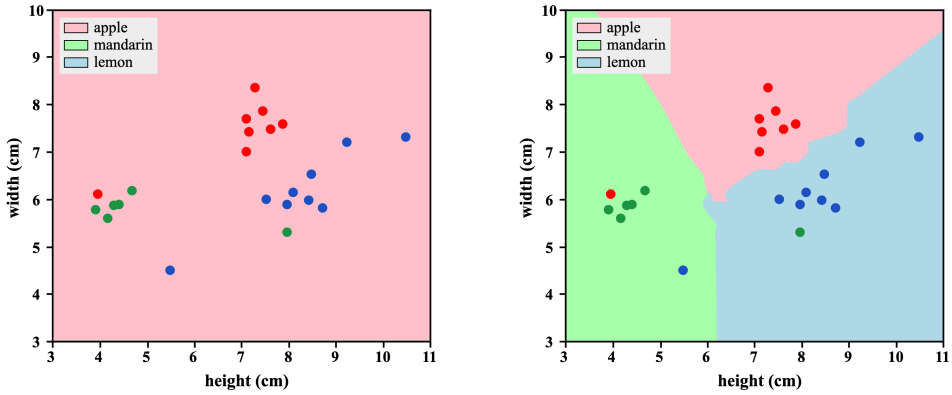
What happened? Although you got a very low error on the *training set* (0% wrong) you got a higher error on the *test set* (60% wrong). This problem, where the error is low on the training set, but much higher on an unseen test set, is a central problem in ML known as *overfitting*. Essentially, what happened is your classifier (nearly) memorized the training data. One potential indication of this is a rough and jagged decision boundary, as is the case in Figure 2.2a, which is strongly influenced by every single data point.

Overfitting is not just a problem in ML, but is a problem us humans suffer from while learning as well. Consider the following example which is probably similar to an experience you've had before. Two students, Ben and Grace, have a difficult multiple-choice math exam tomorrow. To prepare his students for the exam, the professor gave each of them a past exam and told them that tomorrow's exam would be similar. Ben spent most of his time memorizing the correct answers to the practice exam, while Grace spent her time working through the problems to understand *how* to calculate the correct answers. Before they went to bed that night, both Ben and Grace took the practice exam. Ben got a 98% (or 2% error) while Grace only got a 83% (or 17% error). Ben was confident that he would ace the exam tomorrow and that his friend Grace would not do so well. But what happened the next day when the professor gave them a new 100-question exam? Ben got incredibly lucky because 20 of the questions were nearly identical to the ones on the practice exam. He gave the answer based on his memory and got them all right. But the other 80 questions did not resemble the practice questions. On these questions, he was left pretty much making a wild guess and only got 20 of them correct. All-in-all his score was 40% (or 60% error), much worse than his score on the practice exam. Since Grace, on the other hand, learned the *concepts* from the practice exams, she was able to adapt them to the problems on the new exam and scored 80% (or 20% error), almost the same as she scored on the practice exam the night before. Grace learned better than Ben and, despite doing worse than him on the practice exam, did much better than him on the new exam. Ben's learning was *overfit* to the practice exam, just as our learned fruit classifier was *overfit* to the training data.

2.1.4 Controlling Model Complexity

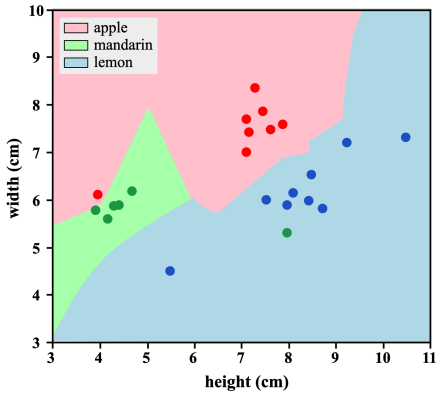
Now we have seen why learning a model that does well on the training set is not such a great idea - since it might overfit and perform poorly on test data. On the other hand, we avoid overfitting by learning a trivially simple model, like predicting the same fruit type, `Apple`, for all new fruits. Such a model would get roughly the same score for training and test (hence not overfit), but its performance would be terrible (since we are not using height and width information) – such a model is said to *underfit* the data.

A fundamental problem in ML is to learn a model that neither overfits nor underfits the data. To see how we do this for the KNN classifier, let's see what happens when we vary the parameter k, the number of nearest neighbors used to classify a point. Figure 2.4 illustrates the decision regions for different values of k. In the top left subfigure, Figure 2.4a, $k = n = 20$ leads to a severely underfit model – each classification uses all of the $n = 20$ datapoints in the training set and hence predicts the same value, `Apple`, for each point. As we decrease the value of k in the remaining subfigures, the model gives more attention to each datapoint, and hence we get a more complex classifier (yielding

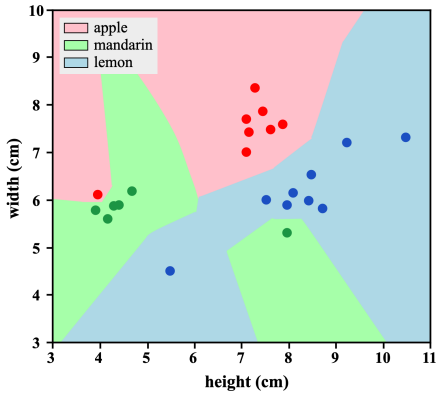


(a) $k = n = 23$ (severely under-fit)

(b) $k = 5$ (optimal fit)



(c) $k = 2$ (mildly overfit)



(d) $k = 1$ (severely over-fit)

Figure 2.4: Decision regions for decreasing values of k (the number of neighbors in the KNN classifier). In figure (a) we underfit the data, learning a function that predicts `Apple` regardless of height and width; figure (b) obtains a more precise fit; figure (c) an even more precise fit (obtaining minimal test set error - see Figure 2.5); figure (d) is severely overfit.

rougher decision boundaries). When k is decreased all the way until $k = 1$ in the bottom right subfigure, Figure 2.4d, we learn the most complex model (classifier pays attention to every single data point).

The goal in Machine Learning is to find a *generic* model – the model that optimally balances the overfitting and underfitting of data. Model complexity is defined with the hyperparameters of the model itself. Each model has its own set of hyperparameters. The complexity of a KNN classifier is defined by the *hyper-parameter* k. What is the right value of k to produce a good the model?

Figure 2.5 plots the train and test error for each value of k ranging from $k = n$ on the left down to $k = 1$ on the right. On the left side of the plot (near $k = n$) we underfit the data: the train and test error are very similar (almost touching) but they are both very high. On the other hand, on the right side of the plot (near $k = 1$) we overfit the data: the training error is very low (0%, as we calculated), but the test error is much higher (60%). The model with $k = 5$ achieves the optimal test error, and hence the optimal trade off between overfitting and underfitting.

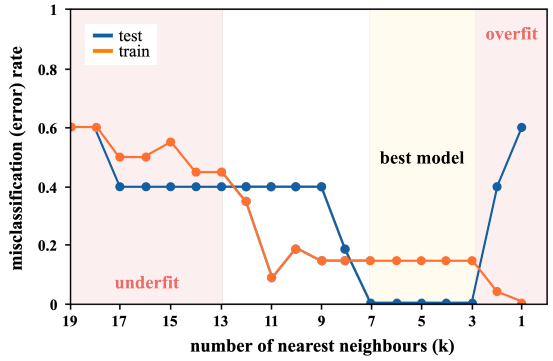


Figure 2.5: Misclassification error rate for training and test sets as we vary k, the number of neighbors in our KNN model. Models on the left side of the plot (near $k = n = 20$) use too large a neighborhood and produce an underfit model. Models on the right side of the plot (near $k = 1$) use too small a neighborhood and produce an overfit model. Models between $k = 3$ and $k = 7$ perform optimally on the test set – in other words, they optimally balance overfitting and underfitting.

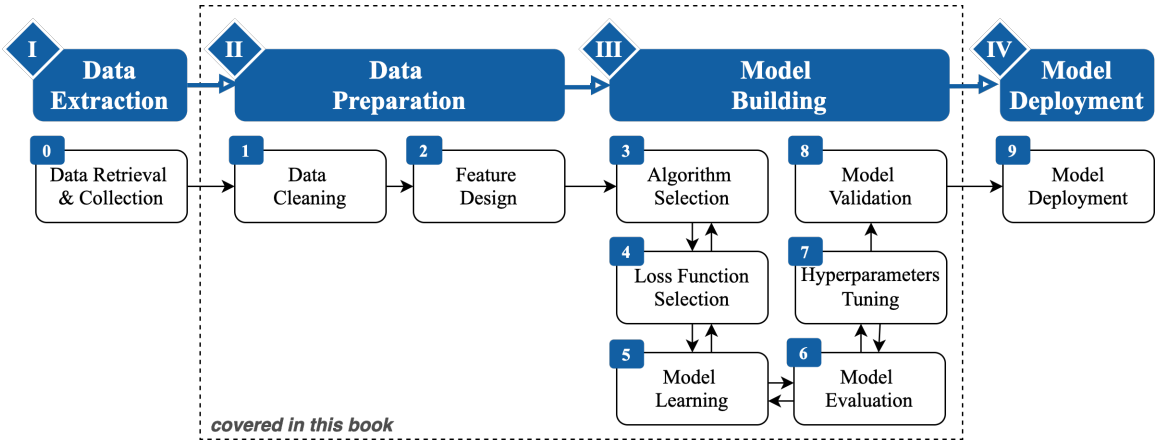


Figure 2.6: Detailed pipeline for a full ML system. The blue boxes represent the high-level ML pipeline we saw in Figure 1.2 of Chapter 1. The white boxes represent more detailed steps (underneath each high-level step) of the pipeline which we discuss in this chapter. This diagram can serve as your "mind map" which guides you through the rest of this book in which we will present details for (most of) the white boxes.

The plot above is typical of most ML algorithms: low complexity models (left part of the figure) underfit the data; as model complexity increases (moving right on the figure), the training error decreases, but the test error begins to decrease, hits a minimum error, then begins to increase as model complexity is further increased, signifying overfitting. (Note that this explanation holds approximately, as there are several 'bumps' in the figure.)

2.2 ML Pipeline: General Form

In the last section we saw how to perform the basic steps of ML in the context of a simple example. These steps are actually part of a more general pipeline that can be used to solve most ML problems.

We actually saw a very high-level view of this pipeline in Figure 1.2 of Chapter 1. We are now ready to see a more detailed form of the pipeline, shown in Figure 2.6. The pipeline consists of four main stages – Data Extraction, Data Preparation, Model Building, and Model Deployment – shown in the blue boxes at the top. Each main stage consists of several intermediate stages which are shown in the white boxes below the main stage's box. Of these high level stages, the *Model Building* stage contains most of the complex and difficult ML components. The majority of this book will focus on this stage, hence it is highlighted in the figure. The following sections give a brief overview of each stage and also tells you where in the book that stage will be discussed in detail. Thus, this section will also serve as a roadmap for your study of the rest of the book.

2.2.1 Data Extraction

Before discussing the steps of the ML pipeline, we briefly discuss how we obtain data and labels for training. Labels are usually obtained in one of two ways: either by past observations (such as what price a house actually sold for) or by subjective evaluations of a human being (such as determining if an email is spam or not). It is important to understand that the supervised ML algorithm is not independently smart (as some sci-fi depictions of AI suggest) but instead is only learning to mimic the labels you gave it. If a human provides bad subjective labels, such as labeling non-spam emails as spam - either accidentally or maliciously - the supervised ML algorithm doesn't know the human's labels were wrong. The supervised ML will try to learn to find patterns that predict the "incorrect" labeling that you gave it. For example, in our fruit classification problem, if you provided bad labels by, say, labeling half of the mandarins as lemons (either by accident or maliciously), even the best ML algorithms would learn an incorrect way to predict a fruit's type.

Although the rest of the ML pipeline is clearly important, and is the part that you will spend nearly all of your time learning to master, we cannot underestimate the importance of high quality data. In fact, in recent years, many ML systems obtained huge increases in performance not by creating better ML algorithms, but by a dedicated effort to collect and provide high quality labels for large datasets.

2.2.2 Data Preparation

The core of the ML pipeline, the Model Building stage, assumes the data is in an idealized form. Essentially, this means that our dataset is a clean table with one row for each datapoint, as in our fruit dataset in Table 2.1. In real life problems it is usually necessary to preprocess the dataset to get it into this format before any ML algorithms can be run. In this section, we give a brief overview of data preparation which is necessary in all real life ML problems.

Data Cleaning

Many ML algorithms cannot be applied directly to 'raw' datasets that we obtain in practice. Practical datasets often have missing values, improperly scaled measurements, erroneous or outlier data points, or non-numeric structured data (like strings) that cannot be directly fed into the ML algorithm. The goal of data cleaning is to pre-process the raw data into a structured numeric format (basically, a data table like we have in our example in Table 2.1) that the ML algorithm can act upon.

Other techniques similar to data cleaning (which get the data into an acceptable format to perform ML), are: to convert numerical values into categories (called feature binning); to convert from categorical values into numerical (called feature encoding); to scale feature measurements to a

similar range. We discuss data cleaning, feature encoding, and related techniques and give more detailed examples of why and when we need them in Chapter 7.

Feature Design

Data cleaning is necessary to get the data into an acceptable format for ML. The goal of feature design techniques is to improve the performance of the ML model by combining raw features into new features or removing irrelevant features. The two main feature design paradigms are:

Feature Transformation: Feature transformation is the process of transforming the human-readable data into a machine-interpretable data. For instance, many algorithm cannot treat categorical values, such as "yes" and "no", and we have to transform those into numerical form, such as 0 and 1, respectively.

Feature Engineering: Feature engineering is the process of combining (raw) features into new features that do a better job of capturing the problem structure. For example, in our fruit classification problem, it may be useful to build a model not just on the height and width of a fruit, but also on the ratio of height to width of the fruit. This ratio feature is a simple example of feature engineering. Feature engineering is usually more of an art than science, where finding good features requires (perhaps a lot of) trial and error and can require consulting an expert with domain knowledge specific to your problem (for example, consulting a doctor if you are building a classification model for a medical problem).

Feature Selection: Feature selection is another idea whose goal is to increase the quality of the final ML model. Instead of creating new features, the goal of feature selection is to identify and remove features that are useless to the prediction problem (or that have low predictive power). Removing useless features ensures the model does not learn to make predictions based on this erroneous information. This can significantly improve our performance on unseen data (test set).

Feature transformation, feature engineering and feature selection are crucial steps that can make or break an ML project. We will discuss them in Chapters 6 and 7.

2.2.3 Model Building

Once we have pre-processed the data into an acceptable format, we then build an ML model. This is where most of the "real ML" takes place, and will be the subject of most of the book. We have seen most of these stages in our fruit classification example problem. This section presents the general form of each building block.

Algorithm Selection

The next step is to select the *form* of prediction function $\hat{f}$. The form of $\hat{f}$ should be chosen such that it accurately captures the "true" function f. Of course, the true f is unknown (that is why we are trying to learn $\hat{f}$ from data!). The vast array of ML algorithms that have been developed correspond to choosing a different form of the function $\hat{f}$. We discuss most of these algorithms in Part II of this book. These algorithms include:

- Linear and Polynomial models
- Logit models
- Bayesian models
- Maximum margin models
- Tree-based models

- Ensemble models

Many of these algorithms can be applied to both classification and regression problems (with slight changes). As we learn about these algorithms in the second part of Machine Learning Simplified book, it will be helpful for you to keep in mind that all of these algorithms simply amount to changing this one part of the ML pipeline (while keeping the rest of it fixed).

Loss Function Selection

After selecting a specific algorithm, we need to decide on its loss function: the method which the algorithm would use to *learn from the data*. (Yes, there are different ways of learning for an algorithm, and we will thoroughly discuss this topic later in this book.) For example, a linear regression typically uses the famous least squares error loss function, which we present in Chapter 3. However, there are other loss functions for linear regression (such as the least absolute deviation) that we will talk about in the same chapter. Finally, the selection of the learning algorithm and the selection of the loss function are, of course, coupled: for example, you cannot use a software package that implements a linear regression algorithm with a misclassification loss function.

Model Learning

Once we have selected the learning algorithm and its loss function, we need to train the model. The beauty of machine learning is that the actual learning is not something mystical or conceptually difficult – it is simply a mathematical optimization problem. In Chapter 3 I will clearly show how maths work behind the scene during model learning, and what it really is from the mathematical point of view. But at a high level, learning amounts to finding the set of parameters that minimizes the loss function on the training data.

Model Evaluation

The next key component of the ML algorithm is assessing how well the trained model will perform on unseen test data. As we saw on our fruit classification example above, we cannot simply evaluate the error on the training set (otherwise we could simply memorize the answers and get a perfect score). In an ideal world, we would be supplied with a set of test data, separate from the training data, on which we could evaluate the model. However, in practice, we are often presented with one large dataset, and then it's our job to split it into training and test datasets. How exactly we split the dataset into two sets is a separate topic that deserves attention. In Chapter 5.2 we will discuss more advanced methods, like cross validation, which creates many different splits of the data and evaluates the model's performance on each different split.

Hyper-parameter Tuning

As we saw in our fruit classification example, it is often very easy to build an ML model that performs perfectly (or very well) on training data, but very poorly on unseen test data. We encounter the same difficulty in almost all ML problems, and a central challenge is to train a model that balances the competing problems of *overfitting* and *underfitting*. This balance comes with the right values of hyperparameters.

How can we select the best hyperparameters? Usually there is no way to know which hyper-parameters will work best beforehand. Often many different combinations of hyperparameters are tried and the ones that produces the best results is chosen. For instance, in fruit example with a KNN model, we tried many number of neighbors of the model's hyperparameter k. For a large number of neighbors, the model tend to underfit, while for a small number of neighbors it tend to overfit. This

process is known as *hyperparameter tuning*, and will be discussed in the second part of the Machine Learning Simplified book. Keep in mind that each algorithm has its own set of hyperparameters.

Model Validation

After all of the preceding Data Preparation and Model Building steps, we must validate that our model performs as we expect it will. The problem is that, usually, you will do a lot of experimentation and tinkering with parts of the pipeline – with different feature sets, different learning algorithms, different hyper-parameters, and many other choices. The pitfall is that this meta-level tuning is susceptible to a type of over-fitting itself. In order to validate the quality of the model, we need to test it on a separate portion of data (usually set aside at the beginning of the project) to ensure it is performing as expected. If it is not, we must go back to our pipeline and diagnose why it is not performing well. We will talk about model validation together with the hyperparameter tuning chapter in the second part of the book.

Iterative Development of ML pipeline

The basic ML pipeline consists of running the above steps sequentially, one after the other. However, in practice, the steps are also, often iterated to find the best model. For example, it is often not possible to determine which ML algorithm (e.g., decision tree or neural network) will work best for a particular problem or a particular data set. It is common to run the entire pipeline with one algorithm, then go back (perhaps several weeks later) and try a different algorithm. Similarly, it is common to experiment with many different feature sets, hyper-parameters, and sometimes loss functions to find the best combination for your problem. In other words, when the model is *fully tuned*, the steps in the ML pipeline are run *sequentially* to make a prediction, but the process of *developing and tuning* the model is an *iterative* process. For this reason, I drew arrows going in each direction between components in the ML Pipeline in Figure 2.6.

2.2.4 Model Deployment

Model deployment refers to integrating a machine learning model into an existing production environment or IT landscape. Typically, you will develop your model locally using, for example, a Python Jupyter notebook. In the model deployment stage, you take your code out of the notebook and into production, where "production" can be cloud, local server, or anything else. Frankly, deploying a model is a completely different world, and that is where MLOps engineers come to help. They can use virtual machines and container orchestration tools like Docker and Kubernetes to successfully deploy and scale ML apps. Model deployment is not covered in this book. Since it is a purely applied skill, the best way to master it is with hands-on programming.

Try It Now in Python

Scan the following QR code to get a git repository that provides a step-by-step guide to coding the following topics in Python:

1. **K-Nearest Neighbours**: building models with different k values.
   : `knn.ipynb`

Key concepts

- Supervised ML Classifier
- Overfitting and underfitting
- Supervised ML Pipeline

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Explain how a simple supervised ML classifier performs classification tasks.
- Explain how the level of model complexity affects overfitting and underfitting.
- Explain the basic building blocks of the ML pipeline.

3. Model Learning

Aims and Objectives

The aim of this chapter is to explain:

1. Linear Models
2. Goodness-of-Fit and Cost Functions
3. Gradient Descent Algorithm

Learning Outcomes

By the end of this chapter, you will be able to understand:

- How to quantitatively evaluate any model.
- The inner workings of the gradient descent algorithm.

An overview of the (supervised) ML pipeline was presented in the last chapter. The next few chapters describe the parts of the ML pipeline in detail. We begin in this chapter by describing the part of the ML pipeline that intrigues most people: how a supervised ML algorithm learns from the data (Step 5 from Figure 2.6). First, we will see that the learning problem is in fact nothing more than a mathematical optimization problem. After converting the learning problem to an optimization problem, we will see how the actual learning can be performed by solving the optimization with a simple algorithm known as the *gradient descent algorithm*. I know this conversion takes away a bit of the mystery and excitement surrounding machine learning, but this is where tech really excels: the boring stuff turns what seems like science-fiction into a reality.

3.1 Linear Regression

To see how a ML model is learned, we will begin with showing how to learn a linear regression model on a single variable. If you are already familiar with linear regression, you will probably get through this section pretty easily, though it could still be a great refresher for you. On the other hand, if you feel that you could benefit from more in-depth explanations of the concepts discussed below, I suggest you consult Coursera's Inferential Statistics by the University of Amsterdam.

Amsterdam Apartments		
n.	Area (m^2)	Price (€10,000)
A	30	31
B	46	30
C	60	80
D	65	49
E	77	70
F	95	118

Table 3.1: Hypothetical dataset of apartment prices in Amsterdam.

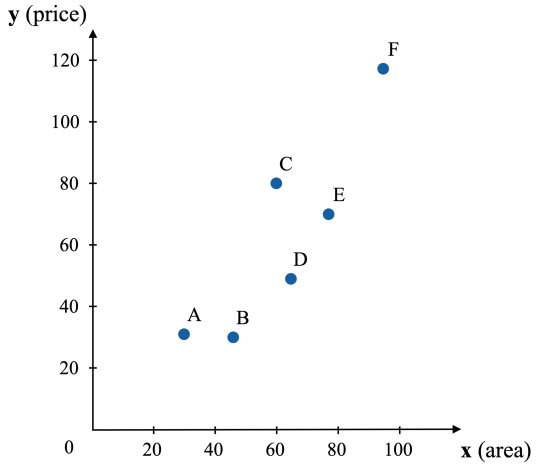


Figure 3.1: A Scatter Plot of the Amsterdam housing prices dataset.

3.1.1 Linear Models

Table 3.1 contains a hypothetical dataset of six apartments located in the center of Amsterdam along with their prices (in €10,000) and floor areas (in square meters). Figure 3.1 shows a plot of these data points where the x-axis is the floor area of an apartment and the y-axis is its corresponding price. Our goal is to predict the price of an apartment given its floor area.

One way to model the relationship between the target variable y (price) and the response variable x (area) is as a *linear relationship*. This linear relationship can be expressed mathematically as:

$$\hat{f}(x) = a \cdot x + b \tag{3.1}$$

where $\hat{f}$ is a *function* representing the prediction, x is the response variable (m^2), and a and b are the *parameters* of $\hat{f}$, where a is the coefficient (slope), and b is the point of intercept. The "hat" over the f indicates that the function is estimated from data. (The parameters a and b are also learned from data, but we do not put a "hat" over them to avoid cluttering notation.)

Finding the line that best fits the data is known as a *linear regression* and is one of the most popular tools in statistics, econometrics, and many other fields. For our housing dataset, the line of best fit, shown in Figure 3.2, is $\hat{f}(x) = 1.3 \cdot x - 18$, which has slope coefficient $a = 1.3$ and point of intercept (on the y-axis) $b = -18$ (we'll discuss how to calculate the line of best fit a little later). This formula allows us to predict the price of an Amsterdam apartment by substituting its floor area for the variable x. For instance, let's say you want to predict the price of an apartment with a floor area of 70 square meters. Simply substitute 70 for x:

$$\begin{aligned}\hat{f}(x) &= 1.3 \cdot x - 18 \\ &= 1.3 \cdot 70 - 18 \\ &= 73\end{aligned}$$

and we see that its predicted price is €730,000. We illustrate this prediction graphically in Figure 3.2.

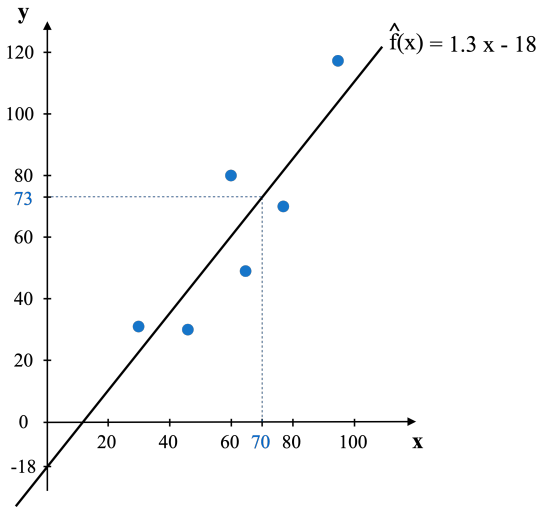


Figure 3.2: Linear regression fit to the Amsterdam housing prices dataset. The learned linear model can be used to predict the price of a new house given its area in square meters. Shown in blue is the prediction that an apartment with floor area of 70 (in m^2) has a price of 73 (in €10,000).

Now we know how to make a prediction. But let's take a step back. How can we build an *algorithm* – a specific sequence of steps – to determine the best parameters for the function $\hat{f}(x)$? For this example, how can we discover that $a = 1.3$ and $b = -18$ is the best out of all possible choices of parameters a and b (without being told, obviously)? Conceptually, this requires two parts:

- First, need a way to numerically measure a *goodness-of-fit*, or how well a model fits the data, for one particular setting of parameters. For example, figure 3.3 plots regression models for different settings of the parameters a and b. How do we know which one is better than the others? The label under each subfigure lists a goodness-of-fit metric known as the *SSR* score, which will be presented in the next subsection.
- After we know how to measure how well some specific settings of parameters a and b is, we need to figure out how to search over all possible values of a and b to find the best one. A naive approach would be to try all possibilities but this would take forever. In Section 3.1.3 we discuss a much better algorithm known as gradient descent.

3.1.2 Goodness-of-Fit

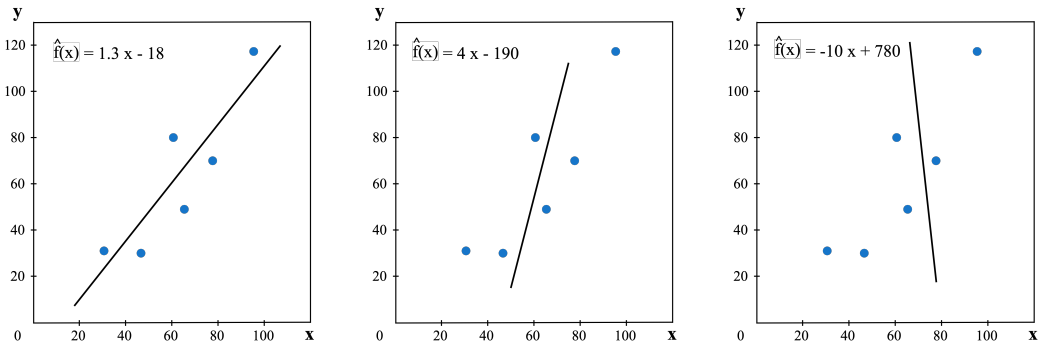
To evaluate the goodness of fit we need to first learn about *residuals*.

Definition 3.1.1: Residual

A **residual** is the difference between the observed value and the value that the model predicts for that observation. The residual is defined as:

$$r_i = y_i - \hat{f}(x_i)$$

where y_i is the actual target value of the i^{th} data point, $\hat{f}(x_i)$ is a predicted value for data point x_i, and r_i is the i^{th} residual.



(a) $a = 1.3$ and $b = -18$; $SSR = 1,248.15$.

(b) $a = 4$ and $b = -190$; $SSR = 20,326$.

(c) $a = 10$ and $b = 780$; $SSR = 388,806$.

Figure 3.3: Linear models with different settings of parameters a and b plotted along with the Amsterdam housing prices dataset. Each label contains the model's *SSR* which measure its goodness-of-fit (lower is better).

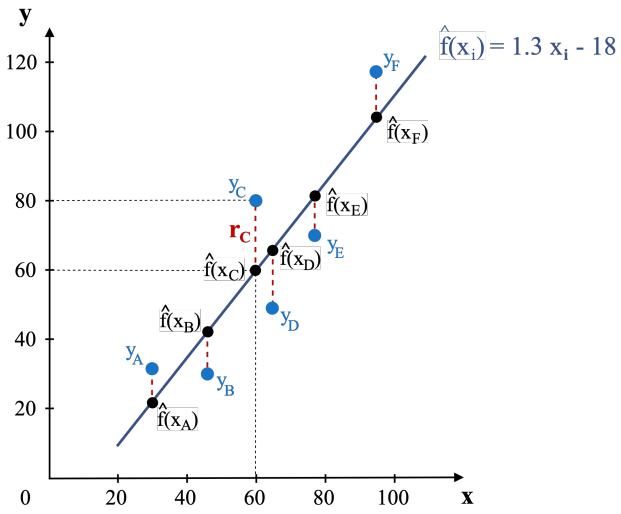


Figure 3.4: Residuals associated with the linear model in Figure 3.3a are depicted with red dashed lines. To keep the figure uncluttered, only the residual r_C for point C is explicitly labeled.

To get a better understanding of residuals, let's illustrate them graphically on our example. Figure 3.4 uses blue data points (●) to represent observed prices of apartments taken from Table 3.1 (denoted as y_i). The figure uses black data points (●) to represent prices of apartments of the same corresponding squared area, but is predicted by our function $\hat{f}$.

In Figure 3.4, data point C shows that the actual price y_C of a $60m^2$ apartment in the market is €800,000. However, the model's predicted price $\hat{f}(x_C)$ for an apartment with the same area is €600,000, because $\hat{f}(x_C) = 1.3 \cdot 60 - 18 = 60$ (in €10,000). The difference of the actual to the

predicted price, is known as the *residual*. For this data point the residual is

$$\begin{aligned} r_C &= y_C - \hat{f}(x_C) \\ &= 80 - (1.3 \cdot x_C - 18) \\ &= 80 - (1.3 \cdot 60 - 18) \\ &= 20, \end{aligned}$$

or €200,000, since the units are €10,000. The residuals for the remaining data points A, B, D, E, and F in our example are

$$\begin{aligned} r_A &= y_A - \hat{f}(x_A) = 31 - (1.3 \cdot 30 - 18) = 10 \\ r_B &= y_B - \hat{f}(x_B) = 30 - (1.3 \cdot 46 - 18) = -11.8 \\ r_D &= y_D - \hat{f}(x_D) = 49 - (1.3 \cdot 65 - 18) = -17.5 \\ r_E &= y_E - \hat{f}(x_E) = 70 - (1.3 \cdot 77 - 18) = -12.1 \\ r_F &= y_F - \hat{f}(x_F) = 118 - (1.3 \cdot 95 - 18) = 12.5. \end{aligned}$$

These residuals are represented with vertical red dashed lines in Figure 3.4. Now we know how to compute the residual of each data point, but how do we calculate the error of the model on the entire dataset? The most popular method is to compute the squared error or *sum of squared residuals* defined as follows.

Definition 3.1.2: Sum of Squared Residuals (SSR)

The *sum of squared residuals* (SSR) is the sum of the squared differences between the observed value and the value that the model predicts for that observation. For a prediction function $\hat{f}$, we calculate the SSR as

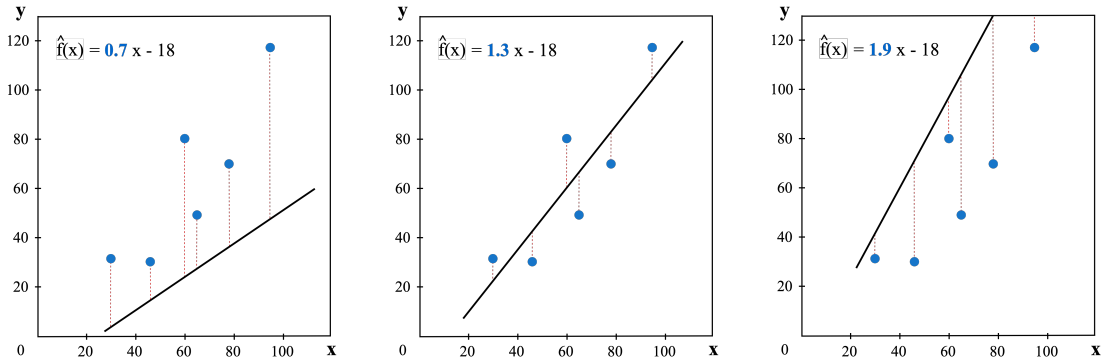
$$SSR(\hat{f}) = \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2 = \sum_{i=1}^{n} r_i^2 \tag{3.2}$$

$SSR(\hat{f})$ measures how well the specific model $\hat{f}$ fits the data: the lower the SSR, the better the fit. SSR is the most widely used loss function, but others are also possible as discussed in Appendix **??**.

In our running example, we compute the sum of the squared residuals as:

$$\begin{aligned} SSR(\hat{f}) &= r_A^2 + r_B^2 + r_C^2 + r_D^2 + r_E^2 + r_F^2 \\ &= (10)^2 + (-11.8)^2 + (20)^2 + (-17.5)^2 + (-12.1)^2 + (12.5)^2 \\ &= 100 + 139.24 + 400 + 306.25 + 146.41 + 156.25 \\ &= 1248.15. \end{aligned}$$

The SSR measures how well the specific model $\hat{f}(x) = 1.3x - 18$ fits the data – the lower the SSR, the better the fit. For instance, Figure 3.3 shows SSR values for the model we just calculated in Figure 3.3a, and two other values shown in Figure 3.3b and Figure 3.3c. Based on these values, we know that the model in Figure 3.3b is better than model in Figure 3.3c, but worse than the model in Figure 3.3a.



(a) $a = 0.7$; $SSR = 10,624.95$. (b) $a = 1.3$; $SSR = 1,248.15$. (c) $a = 1.9$; $SSR = 10,443.75$.

Figure 3.5: Regression models with a different parameter a along with sum of squared residual (SSR) error listed in label.

Parameterized SSR

For our learning (optimization) procedure in the next section, we will be interested in the SSR at many potential settings of parameters, not just at one specific choice. In other words, we want to write the SSR as a function of the parameters in our model. Let's start with an example. For simplicity, let's pretend that the parameter $b = -18$ is known. This leaves us with $\hat{f}(x) = a \cdot x - 18$, a function of a single parameter a. The SSR as a function of parameter of a is

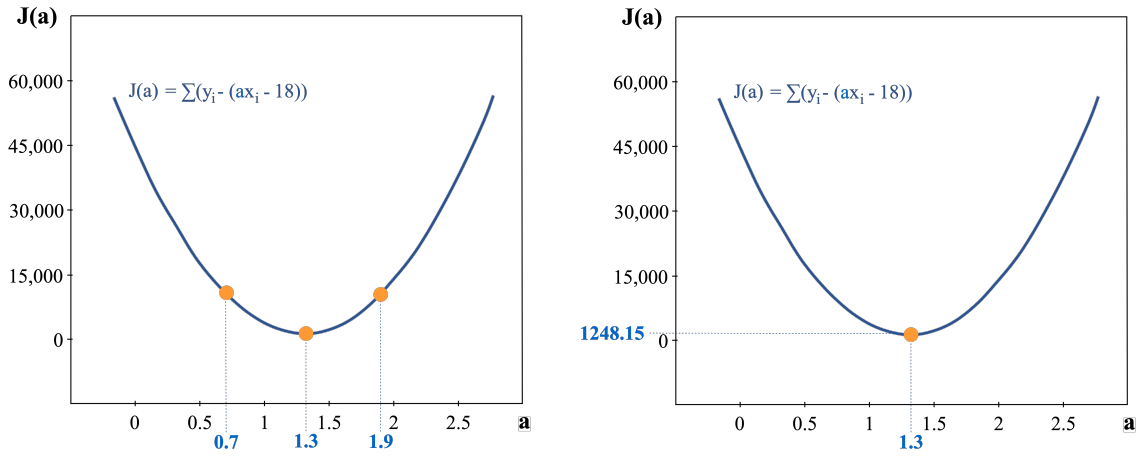
$$\begin{aligned} SSR(a) &= \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2 \\ &= \sum_{i=1}^{n} (y_i - (a \cdot x_i - 18))^2 \end{aligned}$$

Let's evaluate the SSR for some values of the parameter a. When $a = 0.7$, we have

$$\begin{aligned} SSR(a) =& \sum_{i=1}^{n} (y_i - (0.7 \cdot x_i - 18))^2 \\ =& (31 - (0.7 \cdot 30 - 18))^2 + (30 - (0.7 \cdot 46 - 18))^2 + (80 - (0.7 \cdot 60 - 18))^2 + \\ & (49 - (0.7 \cdot 65 - 18))^2 + (70 - (0.7 \cdot 77 - 18))^2 + (118 - (0.7 \cdot 95 - 18))^2 \\ =& 10624.95 \end{aligned}$$

where we substituted the values of x_i and y_i from Table 3.1. Following the same procedure, we find out that when $a = 1.3$, we get $SSR(a) = 1248.15$, and when $a = 1.9$, we get $SSR(a) = 10443.75$. Figure 4.2 shows a plot for the SSR evaluated at each value of a.

Now, let's plot the calculated SSR values over a changing parameter a on the graph, where the x-axis is an a value and the y-axis is the value of the SSR, as shown in Figure 3.6a. In the language of mathematical optimization which we use in the next section, we represent the errors with a *cost function* denoted as J. In other words, $J(a) = SSR(a)$ and we will be using only the notation J from now on.



(a) Cost function plotted against parameter a with a few values highlighted.

(b) Minimum point of cost function marked.

Figure 3.6: The parameterized *SSR*, the SSR as a function of parameter a, for our example dataset and linear model.

3.1.3 Gradient Descent Algorithm

Now that we understand how we can quantitatively evaluate the model, we want to identify the best model, which is the model that produces the lowest value of the cost function. Mathematically, we write this as:

$$\begin{aligned} a^* &= \arg\min_a J(a) \\ &= \arg\min_a \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2 \end{aligned} \tag{3.3}$$

where the value of a that gives the minimum cost is denoted with a "star" superscript as a^*. This values is also known as a *minimum point* of the cost function $J(a)$. In our example with the value of b fixed at $b = -18$, the optimal value of a is $a^* = 1.3$ (which we stated earlier).

But how do we find the optimal value? A slow and painful way would be to "plug and chug" numerous values of a until we find one that gives the lowest cost function $J(a)$. But that's going to take an infuriating, mind-numbing amount of time.[1] There are many algorithms in mathematical optimization that have been developed to find the minimum faster (and there is still much research and many improvements made in this area).

In this section, we will learn how to perform the minimization using the *gradient descent algorithm*. At a high level, the gradient descent algorithm is fairly straightforward. It starts with a random value, or a good guess, of the parameter a. It then finds the direction in which the function decreases fastest and takes a "step" in that direction.[2] It repeats this process of finding the direction of steepest

[1] This statement should be taken qualitatively – technically, for a continuous value of a (with arbitrarily many decimal places) we would not find the exact minimum this way.

[2] Remember from your calculus class that the direction of steepest descent of a function is calculated as the *gradient* of the function. This is where you should stop and refresh your knowledge about the basics of derivatives and gradients if you are uncertain how to use them; from this point, our discussion will assume you understand the basics.

descent and taking a step in that direction until it converges to the minimum value of the function. The details of the algorithm are given in Algorithm **??**. We now give a detailed walk-through of the algorithm and explanation of the math.

Definition 3.1.3: Gradient Descent (with a Single Parameter)

1. Choose a fixed learning rate l.
2. Initialize parameter a to an arbitrary value.
3. While (termination condition not met)
   3.a. Compute $\frac{\partial J}{\partial a}$, the derivative of the cost function J at value a.
   3.b. Take a *step* in the gradient direction, scaled by learning rate l:

$$a_i = a_{i-1} - l \cdot \frac{\partial J}{\partial a}$$

Gradient Derivation

We first compute the gradient of the cost function $J(a) = \sum_{i=1}^{n} \Big(y_i - (ax_i - 18)\Big)^2$ algebraically as

$$\begin{aligned}
\frac{\partial J}{\partial a} &= \frac{\partial}{\partial a} \sum_{i=1}^{n} \Big(y_i - (ax_i - 18)\Big)^2 \\
&= \sum_{i=1}^{n} \frac{\partial}{\partial a} \Big(y_i - (ax_i - 18)\Big)^2 \\
&= \sum_{i=1}^{n} -2 \cdot x_i \cdot \Big(y_i - (ax_i - 18)\Big)
\end{aligned}$$

where the first equality follows since the gradient of a sum of terms is equal to the sum of the gradients of each individual term (here, each individual data point). We now illustrate a step-by-step derivation of the gradient of the cost function at the current value of parameter a. First, we expand the cost function for the data points in our example:

$$\begin{aligned}
J(a) = &\Big(31 - (a \cdot 30 - 18)\Big)^2 + \Big(30 - (a \cdot 46 - 18)\Big)^2 \\
&+ \Big(80 - (a \cdot 60 - 18)\Big)^2 + \Big(49 - (a \cdot 65 - 18)\Big)^2 \\
&+ \Big(70 - (a \cdot 77 - 18)\Big)^2 + \Big(118 - (a \cdot 95 - 18)\Big)^2
\end{aligned}$$

We then take the derivative of this function with respect to parameter a yielding

$$\begin{aligned}
\frac{\partial J}{\partial a} = &\frac{\partial}{\partial a}\Big(31 - (a \cdot 30 - 18)\Big)^2 + \frac{\partial}{\partial a}\Big(30 - (a \cdot 46 - 18)\Big)^2 \\
&+ \frac{\partial}{\partial a}\Big(80 - (a \cdot 60 - 18)\Big)^2 + \frac{\partial}{\partial a}\Big(49 - (a \cdot 65 - 18)\Big)^2 \\
&+ \frac{\partial}{\partial a}\Big(70 - (a \cdot 77 - 18)\Big)^2 + \frac{\partial}{\partial a}\Big(118 - (a \cdot 95 - 18)\Big)^2
\end{aligned}$$

where we used a basic property of derivatives – that the derivative of a sum of terms is equal to the sum of derivatives of each term. To calculate the derivative in each term, we apply the chain rule

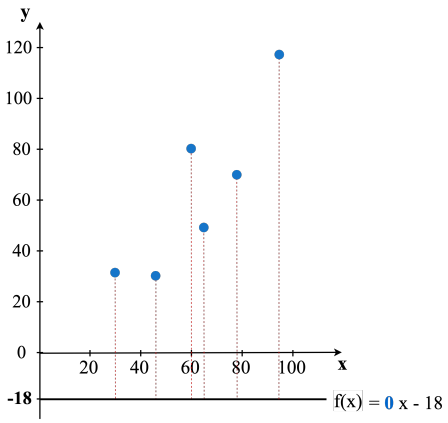


Figure 3.7: Residuals and fit at the initial parameter value $a = 0$.

(another basic property of derivatives) yielding:

$$\begin{aligned}\frac{\partial J}{\partial a} =& (-2)\Big(31-(a\cdot 30-18)\Big)+(-2)\Big(30-(a\cdot 46-18)\Big)\\ &+(-2)\Big(80-(a\cdot 60-18)\Big)+(-2)\Big(49-(a\cdot 65-18)\Big)\\ &+(-2)\Big(70-(a\cdot 77-18)\Big)+(-2)\Big(118-(a\cdot 95-18)\Big)\end{aligned}$$

Initialization and First Iteration

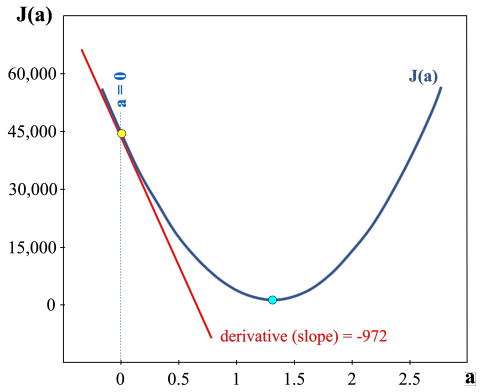
Now that we know how to compute the derivative, let's see how the gradient descent algorithm works on our example. The algorithm initializes the parameter $a = 0$, corresponding to an initial model function $\hat{f}(x) = 0 \cdot x - 18$. This function and its corresponding residuals are plotted in Figure 3.7. The gradient of the cost function at the current point $a = 0$ is

$$\begin{aligned}\frac{\partial J}{\partial a} =& (-2)(31-(\mathbf{0}\cdot 30-18))+(-2)(30-(\mathbf{0}\cdot 46-18))\\ &+(-2)(80-(\mathbf{0}\cdot 60-18))+(-2)(49-(\mathbf{0}\cdot 65-18))\\ &+(-2)(70-(\mathbf{0}\cdot 77-18))+(-2)(118-(\mathbf{0}\cdot 95-18))\\ =& -972.\end{aligned} \tag{3.4}$$

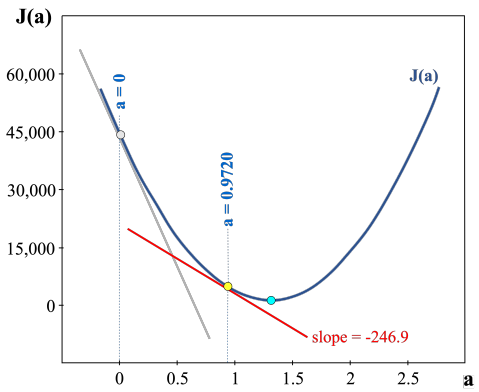
In other words, when $a = 0$, the slope of the curve is equal to -972, as illustrated in Figure 3.8a. We then use this gradient to update the parameter a:

$$a_{new} = a - \underbrace{\frac{\partial J}{\partial a}\cdot l}_{\text{Step Size}} \tag{3.5}$$

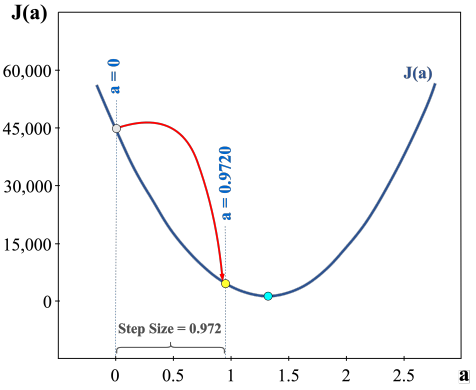
where the *step size* is obtained by multiplying the derivative with a predefined *learning rate* l. The learning rate controls the size of the step we take and will be discussed more later, but for now just keep in mind that the learning rate is usually set to a small value, such as $l = 0.001$. Using the value of the derivative calculated in Equation 3.4 along with a learning rate of $l = 0.001$, the updated parameter value is:



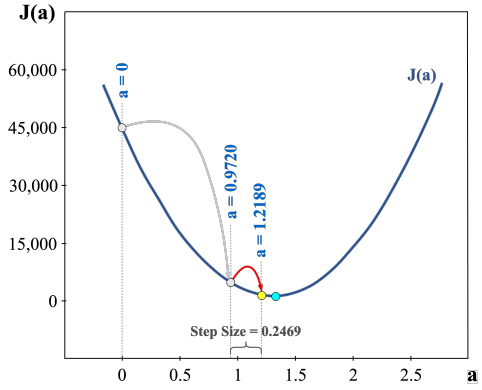
(a) Derivative at the first iteration



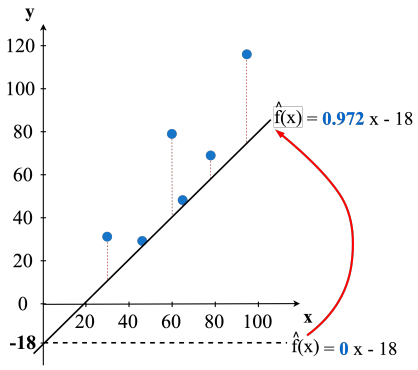
(d) Derivative at second iteration



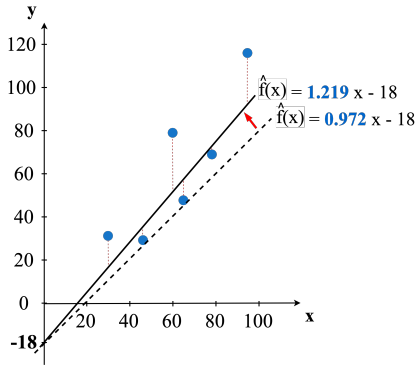
(b) Step at first iteration



(e) Step at second iteration



(c) Model after first iteration



(f) Model after second iteration

Figure 3.8: Illustrations of first two iterations of gradient descent. Plots for the first iteration are in the left column, and plots for the second iteration are in the right column. The figures in column illustrate the operations at each iteration: first computing the gradient, second taking a gradient step to update parameters a, and third displaying the fit and calculating the SSR with the new value of parameter a.

$$
\begin{aligned}
a_{new} &= 0 - \underbrace{(-972) \cdot 0.001}_{\text{Step Size}} \\
&= 0 - \underbrace{(-0.972)}_{\text{Step Size}} \\
&= 0.972
\end{aligned}
$$

Figure 3.8b shows that with the new value for a, we move much closer to the optimal value. We can also see in Figure 3.8c how much the residuals shrink when $a = 0.972$, compared to the previous function with $a = 0$.

Second Iteration

At the second iteration, we take another step toward the optimal value using the same routine. We substitute the current value of a into the cost function gradient and use the result to take a step to update a via Eq. 3.5. To take this step, we go back to the derivative and plug in the new value $a = 0.972$.

$$\begin{aligned}\frac{\partial J}{\partial a} &= (-2)(31-(\mathbf{0.972}\cdot 30-18))+(-2)(30-(\mathbf{0.972}\cdot 46-18))\\ &\quad+(-2)(80-(\mathbf{0.972}\cdot 60-18))+(-2)(49-(\mathbf{0.972}\cdot 65-18))\\ &\quad+(-2)(70-(\mathbf{0.972}\cdot 77-18))+(-2)(118-(\mathbf{0.972}\cdot 95-18))\\ &= -246.9\end{aligned} \tag{3.6}$$

And this tells us that when $a = 0.972$, the slope of the curve = -246.9 as shown in Figure 3.8d. Let's update a by plugging in the current a minus the step size, which is the gradient multiplied by the fixed learning rate $l = 0.001$:

$$\begin{aligned}a_{new} &= 0.972-\underbrace{(-246.9)\cdot 0.001}_{\text{Step Size}}\\ &= 0.972-\underbrace{(-0.2469)}_{\text{Step Size}}\\ &= 1.219\end{aligned} \tag{3.7}$$

We've completed another step towards obtaining a minimum optimal value, shown in Figure 3.8e. We can also compare the residuals when $a = 1.219$ to when $a = 0.972$, shown in Figure 3.8f. We see that the cost function $J(a)$ is getting smaller. The next step is to calculate the derivative of the loss function at the point $a = 1.219$:

$$\begin{aligned}\frac{\partial J}{\partial a} &= (-2)(31-(\mathbf{1.219}\cdot 30-18))+(-2)(30-(\mathbf{1.219}\cdot 46-18))\\ &\quad+(-2)(80-(\mathbf{1.219}\cdot 60-18))+(-2)(49-(\mathbf{1.219}\cdot 65-18))\\ &\quad+(-2)(70-(\mathbf{1.219}\cdot 77-18))+(-2)(118-(\mathbf{1.219}\cdot 95-18))\\ &= -62.71\end{aligned} \tag{3.8}$$

This tells us that when $a = 1.219$, the slope of the curve is equal to -62.71. It's time to calculate the new value for a.

$$\begin{aligned}a_{new} &= 1.219-\underbrace{(-62.71)\cdot 0.001}_{\text{Step Size}}\\ &= 1.219-\underbrace{(-0.0627)}_{\text{Step Size}}\\ &= 1.282\end{aligned} \tag{3.9}$$

Iterations of Gradient Descent			
Iter	$\frac{\partial J}{\partial a}$	*Step Size*	a
1	-972.0000	-0.9720	0.9720
2	-246.8880	-0.2469	1.2189
3	-62.7096	-0.0627	1.2816
4	-15.9282	-0.0159	1.2975
5	-4.0458	-0.0040	1.3016
6	-1.0276	-0.0010	1.3026
7	-0.2610	-0.0003	1.3029
8	-0.0663	-0.0000	1.3029

Table 3.2: Iterations of gradient descent on our example. Each row of the table represents an iteration with model parameter a and the gradient of the loss function J at the value a.

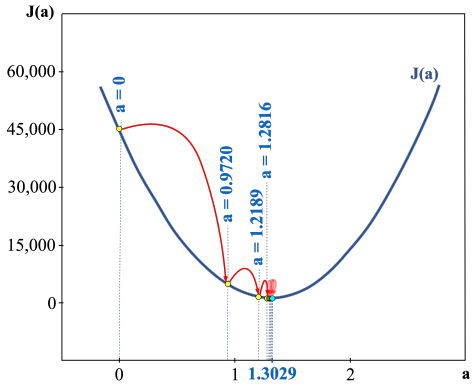


Figure 3.9: Red arrows indicate change in parameter value a at each iteration of gradient descent (starting at $a = 0$)

Additional Iterations and Termination

Our initial estimate for the parameter a was $a = 0$ and after three iterations of the gradient descent algorithm our parameter estimate is $a = 1.282$. Further iterations of gradient descent can be performed by following the procedure and calculations above – I am leaving these calculations for you to perform to test your understanding of the gradient descent algorithm. You can check your calculations with the results in Table 3.2 which shows the quantities computed over eight iterations of the gradient descent algorithm. Figure 3.9 plots the cost function graph $J(a)$ along with a series of red arrows showing how the parameter a changes at each iteration. Note that the closer we get to the optimal value for a, the smaller the step size gets. Furthermore, at the minimum value, the derivative is exactly equal to zero, and hence, the step size is exactly equal to zero, and the gradient descent algorithm terminates.

In practice, the gradient descent algorithm will never obtain the exact optimum and an exactly zero gradient. Instead, the gradient descent algorithm is usually terminated when the magnitude of the gradient is below some small user-defined threshold. For example, if if we choose to terminate when the absolute value of a step size is less than 0.001, then iteration 6 will be the last iteration (since $|-0.0003| = 0.0003 < 0.001$) with a value of $a = 1.3026$. Another possible termination condition is to pre-specify a number of gradient descent iterations. For example, only allow 10 total iterations.

Different Initialization

If you remember, we started by initiating our gradient descent with $a = 0$. The truth is, no matter what initialization we use, we will still (eventually) find the optimum value. Let's see what happens if instead of initializing $a = 0$, we initialize $a = 2.25$ (assume we use the same learning rate of $l = 0.001$). Once again I leave you to groan as you start the actual math work, and just show my end-result calculations in Table 3.3. By looking at Figure 3.10, you can see that even when we start with a different a value, we still find the minimum point of the cost function.

Gradient Descent Iterations			
Iter	$\frac{\partial J}{\partial a}$	*Step Size*	a
1	706.5000	0.7065	2.2500
2	179.4510	0.1795	1.5435
3	45.5806	0.0456	1.3640
4	11.5775	0.0116	1.3185
5	2.9407	0.0029	1.3069
6	0.7469	0.0007	1.3040
7	0.1897	0.0002	1.3032
8	0.0482	0.0000	1.3030

Table 3.3: Each row contains the quantities for an iteration of the gradient descent algorithm on our example problem. The first column is the gradient, the second column is the step size $l \cdot \frac{\partial J}{\partial a}$, and the third column is the updated parameter value.

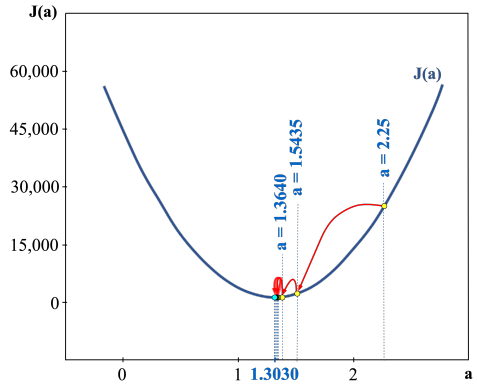


Figure 3.10: All steps of gradient descent when the initial parameter value is $a = 2.25$

The Learning Rate

It's important to choose an appropriate learning rate (denoted as l in Eq. 3.5) in the gradient descent algorithm: a learning rate that is excessively small or excessively large will cause the algorithm to converge slowly or fail to converge at all. If the learning rate is too small, we will take unnecessarily small steps causing us to take an excessively long time to reach the optimum value. On the other hand, if the learning rate is too large, we will take large steps that jump back and forth over the optimal value many times, unnecessarily; this will be very slow and may cause us to not reach the optimum at all. An illustration of the problem of choosing a learning rate that is too small or too large is shown on our running example in Figure 3.11.

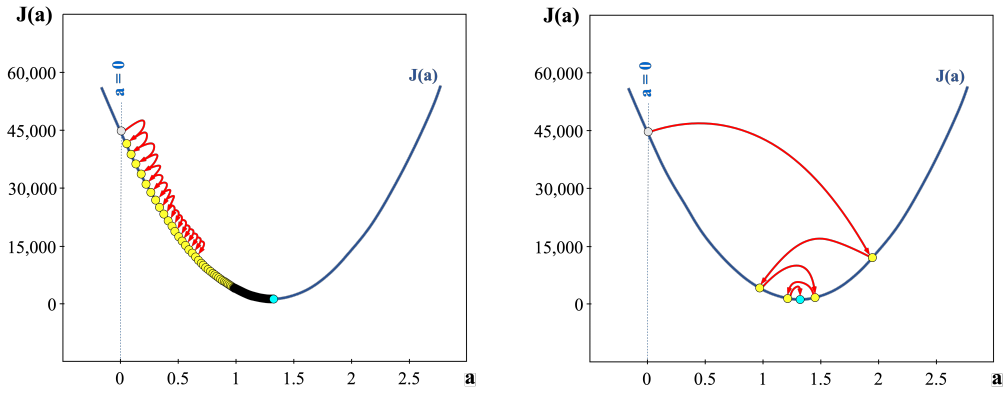
Stochastic Gradient Descent

The gradient descent algorithm we just learnt is called *batch gradient descent*, since computation of the gradient requires touching each data point, or the entire *batch*. Our example dataset contains only six data points, so the cost of touching each data point is not very large. However, real life data sets often contain millions or billions of data points, making the cost of touching each data point very large. The *Stochastic gradient descent (SGD)* algorithm utilizes a low-cost approximation to the gradient, in fact, an approximation that is correct *on average*, that touches only a *single* data point, rather than the entire data set. That is, the SGD update is:

$$\theta := \theta - \nabla_\theta J(\theta;\, x_i;\, y_i) \cdot l \tag{3.10}$$

at a data point index i, where i is chosen at random at each iteration of the SGD algorithm. Since the data points are chosen at random, each data point will eventually participate in the learning process, albeit not at every iteration.

Although SGD is much faster than the *full-batch* gradient descent, the gradient at a single point might be an inaccurate representation of the full data-set, causing us, in fact, to step in a bad direction. In practice, a good trade-off between SGD and *full-batch* gradient descent that maintains both speed



(a) l =?? (b) l =??

Figure 3.11: Progression of the gradient descent algorithm with excessively small or large learning rate plotted against the cost function. (a) With small learning rate, l =??, the algorithm takes many steps to slowly reach the optimum. (b) With large learning rate, l =??, the algorithm jumps over the optimum many times before reaching it.

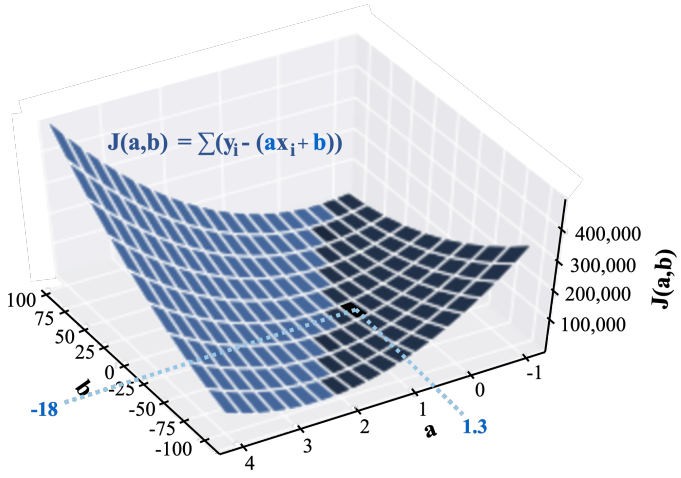


Figure 3.12: Cost function with two unknown weights. Gradient descent finds the minimum of the bowl.

and accuracy is to compute the gradient on a *mini-batch* consisting of $k > 1$ samples, where k is a hyper-parameter, typically between $k = 10$ and $k = 1000$. The mini-batch update using k data points is:

$$\theta := \theta - \nabla_\theta J(\theta;\, x_{i:i+k-1};\, y_{i:i+k-1}) \cdot l \tag{3.11}$$

where $x_{i:i+k-1}$ are the k consecutive data points starting at a random index i.

3.1.4 Gradient Descent with More Parameters

Now that we have seen how the gradient descent algorithm works on a simple model with one parameter. But real problems have many more parameters that we need to estimate. Can gradient

descent be applied to these problems? The answer is yes. It is a very simple modification which we explain in this section. We also present the results using vector algebra which you will see often in ML books.

Let's start by looking at how gradient descent estimates not just one, but two parameters. In our housing example we assumed that we knew the true value of the intercept parameter b, so the cost function $J(a)$ had a single parameter a. When both a and b are unknown, the cost function $J(a,b)$ has two parameters, a and b, to estimate. Gradient descent uses exactly the same logic to estimate the parameters. First we initialize a and b at any value to calculate the derivative $J(a,b)$, but this time with respect to *each* parameter a and b where the other parameter is held fixed.

First, what is the gradient with respect to the parameter a when b is held fixed? This is exactly what we saw in the last section:

$$\frac{\partial J}{\partial a} = \sum_{i=1}^{n} 2 \cdot x_i \cdot (y_i - (ax_i - b))$$

Second, what is the gradient with respect to the parameter b when a is held fixed?

$$\frac{\partial J}{\partial b} = \sum_{i=1}^{n} -2 \cdot (y_i - (ax_i - b))$$

But what happens if we need to find both a and b? If the parameter b is also unknown, the cost function $J(a,b)$ is a function of two parameters and can be visualized with a 3-D graph. Figure 3.12 shows the cost function for different values for the coefficient a and the intercept b.

In higher dimensional cases, the cost-function cannot be visualized easily. However, the gradient descent algorithm can be applied in just the same way.

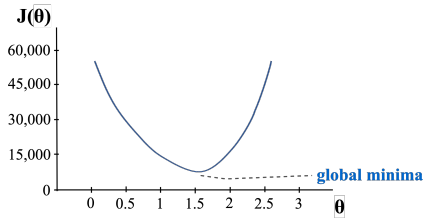
3.2 Gradient Descent in Other ML Models

In the last section we saw how to apply the gradient descent algorithm to learn a linear model that minimizes the SSR cost function. But the gradient descent algorithm is not specific to this particular model or cost function. Different ML models, such as neural networks, produce different optimization problems, but the gradient descent algorithm can still be applied to it: that is, find the direction of steepest descent and take a step in that direction. However, in more complex (and real-life) models, gradient descent may not find the global minimum of the function due to a *more complex cost functions*.

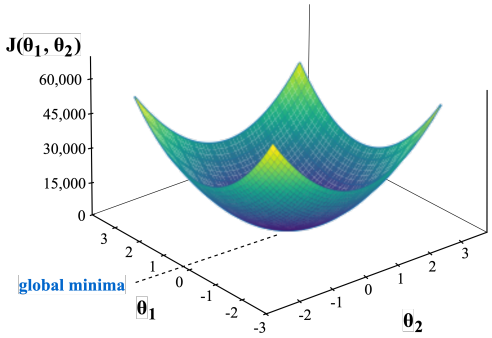
In particular, cost-functions that are not well-behaved may either cause the gradient descent to either (i) stuck in local minima, or (ii) overshoot global minima. Additionally, cost functions can be (iii) non-differentiable, which make it difficult to apply gradient descent. We discuss each of these problems in this section. However, before we do that, I'd like to take a moment and explain two different types of cost functions - convex and non-convex cost functions. This will significantly help us in understanding aforementioned problems that the gradient descent tackles.

Convex and Non-Convex Cost Functions

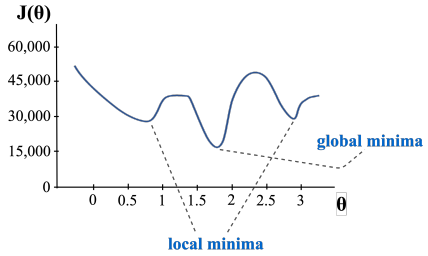
The SSR loss function we have used so far is known as a *convex* function while more complex ML models may have *non-convex* cost-functions. When plotted, a convex cost-function is oriented *upward*, making it look like half a circle or an oval, while a non-convex cost-function has at least one interior angle that is greater than 180°. The shape of convex cost functions has pointy ends and



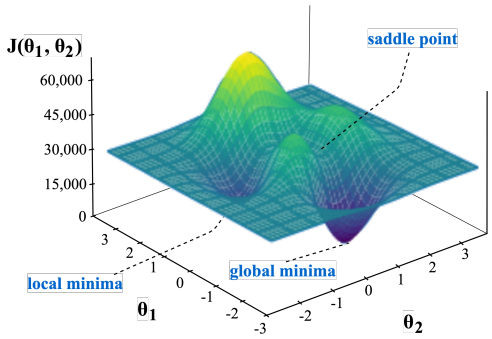
(a) Convex, one parameter.



(b) Convex, two parameters.



(c) Non-convex, one parameter; note that there is no saddle point in function of 1 parameter (saddle point occurs when gradient is zero, but point is minimum in one direction and maximum in another).



(d) Non-convex, two parameters

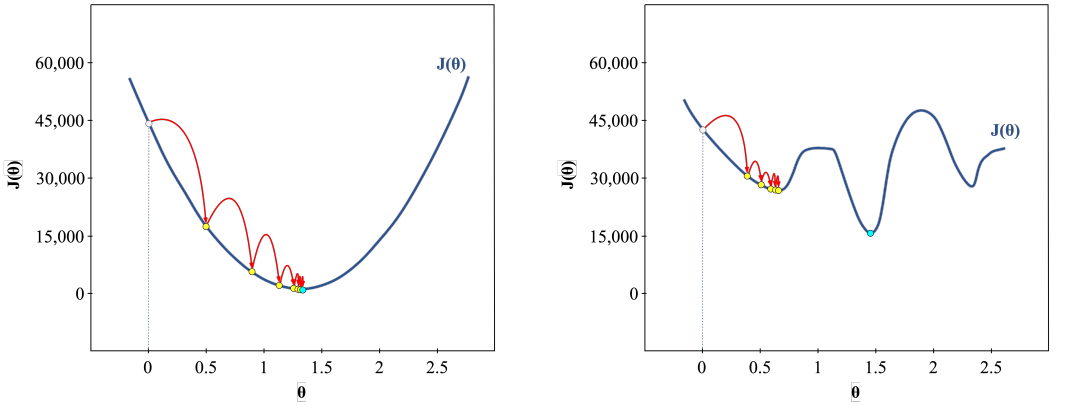
Figure 3.13: Example comparing convex cost functions (top row) and non-convex cost functions (bottom row). The subfigures on the left show a cost-function with one parameter (hence a 2-D plot) while the subfigures on the right show a cost-function with two parameters (hence a 3-D plot).

a thick middle – called a global minimum. A comparison of convex and non-convex cost-functions is shown in Figure 3.13. This figure shows example cost functions with a single parameter (giving rise to 2-D plots), as well as cost functions with two parameters (giving rise to 3-D plots).

For non-convex cost-functions with two parameters, there are three types of critical points that have gradient zero, and are thus relevant to the optimization:

- **Saddle points** are the plateau-like regions.
- **Local minima** is the smallest values of the function within a specific range.
- **Global minimum** is the smallest value of the function on the entire domain.

A non-convex cost functions can have multiple local minima, that are, the smallest points within a specific range of the cost function. It can also have multiple global minimums with equal value, although it is rarely occurred. The objective of the gradient descent is to find an optimal value, which is, *any global minimum point*.



(a) Gradient Descent on a Convex Cost Function.

(b) Gradient Descent on a Non-convex Cost Function.

Figure 3.14: Comparison of the gradient descent algorithm applied to a convex cost function (left) and a non-convex cost function (right). For a convex cost function we find the global optimum, while for a non-convex cost function we can get stuck in a local optimum.

3.2.1 Getting Stuck in a Local Minimum

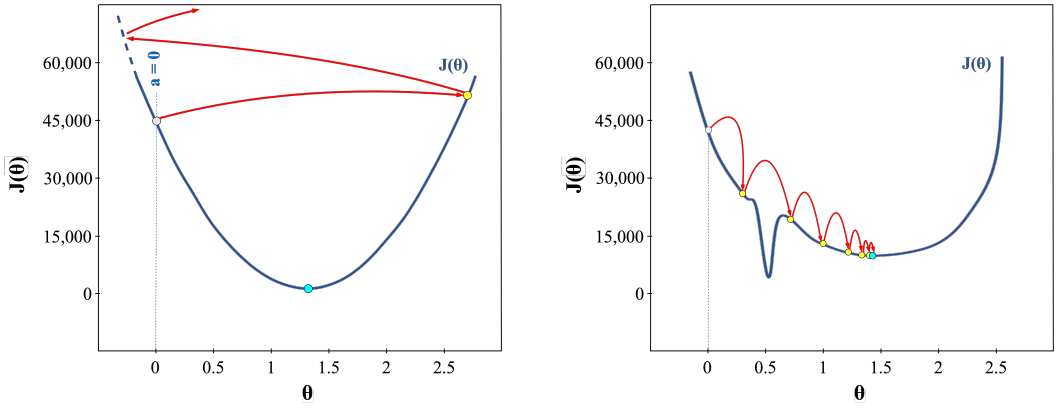
Figure 3.14a shows a convex curve with a single global minimum. This means that the gradient descent algorithm will reach the global minimum for any value of a that we start at (as we saw in Section 3.1.3). Figure 3.14b shows a non-convex curve and an example run of the gradient descent algorithm initialized at $a = 0$. In this case we *get stuck* at a local minimum and fail to find the global minimum (represented by the blue dot in the figures).

Getting stuck in a local minimum is a big problem for many learning algorithms with non-convex cost functions. It seems that one of the solutions is to increase the learning rate, since it ultimately increases the step-size, hence leading to lower probability for the gradient to stuck in local minimum. However, you should be careful with tuning the learning rate: large learning rates can lead to another problem - overshooting global minimum.

3.2.2 Overshooting Global Minimum

Larger learning rates produces larger step size, and we don't like large steps because the gradient can accidentally skip or even diverge from the global optimum.

For instance, Figure 3.15 shows how we can overshoot the global minimum point with a big learning rate on (a) a convex cost function, and (b) a non-convext cost function.



(a) Gradient descent missing global minimum on a convex cost function due to a very large learning rate.

(b) Gradient Descent missing global minimum on a non-convex cost function due to a very large learning rate.

Figure 3.15: Comparison of the gradient descent algorithm missing a global minimum on a convex cost function (left) and a non-convex cost function (right). If the learning rate l (and the step size with it) is too large, for a convex cost function we can (plausibly) "jump over" the minima we are trying to reach, while for a non-convex cost function we can skip a global optimum.

Possible Solutions

In general, there is no silver bullet solution to extremum problems, but several techniques have been shown to work well in practice to avoid them and move closer to the global minimum (or a better local minimum). These techniques include:

1. Use different variations of gradient descent (such as Stochastic Gradient Descent, Mini Batch Gradient Descent, Momentum-based Gradient Descent, Nesterov Accelerated Gradient Descent)
2. Use different step sizes by adjusting the learning rate.

These techniques are not covered by this book, but are something you can discover in your free time.

3.2.3 Non-differentiable Cost Functions

As we learned in Section 3.1.3, gradient descent requires taking the derivative of a cost function. In other words, gradient descent is only applicable if the cost function is differentiable. Unfortunately, not all the functions are differentiable. This is a slightly more advanced topic, but generally the most common forms of non-differentiable behavior involve:

1. Discontinuous functions
2. Continuous but non-differentiable functions

If you need a small refresher for discontinuous / continuous non-differentiable functions, you could have a look at Appendix B.

Try It Now in Python

Scan the following QR code to get a git repository that provides a step-by-step guide to coding the following topics in Python:

1. **Gradient Descent**: fitting linear regression with gradient descent.
   : `linear_regression_and_gradient_descent.ipynb`

Key concepts

- Goodness-of-Fit and SSR
- Cost Function
- Gradient Descent

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Explain how to quantitatively evaluate linear algorithms.
- Explain what is gradient descent and how to visualize it on a linear regression model.

4. Basis Expansion and Regularization

Aims and Objectives

The aim of this chapter is to explain:

1. Basis expansion
2. Regularization

Learning Outcomes

By the end of this chapter, you will be able to understand:

- How we can use basis expansion to increase model's complexity.
- Why we need regularization and when it is helpful.

An overview of linear regression and gradient descent was presented in the last chapter. This chapter focuses on how we can modify linear regression and its cost function as a way to change its complexity.

4.1 Basis Expansion

You may be thinking that linear regression is too weak of a model for any practical purposes. Sometimes, this is true: the data features have strong non-linear relationships that are not captured by the linear model. But does this mean that everything we just learned about linear regression in the previous chapter is useless in this case? Not at all! This section discusses a powerful technique known as *basis expansion* that effectively adds non-linear features into the model. Then, linear regression can be applied directly on the dataset to learn the coefficients of the non-linear terms. This section will discuss basis expansion into polynomial features, which is a very general technique. There are many other possibilities, some of which are more appropriate for different problems; we discuss the choice in more detail in the chapter on Feature Engineering.

Training Dataset		
$n.$	Area (m^2)	Price (€10,000)
A	30	31
B	46	30
C	60	80
D	65	49
E	77	70
F	95	118

(a) Training dataset

Test Dataset		
$n.$	Area (m^2)	Price (€10,000)
G	17	19
H	40	50
I	55	60
J	57	32
K	70	90
L	85	110

(b) Test dataset

Table 4.1: Amsterdam housing dataset. (a) Training set identical to the one we used in Section 3.1, (b) Test dataset used to evaluate our model.

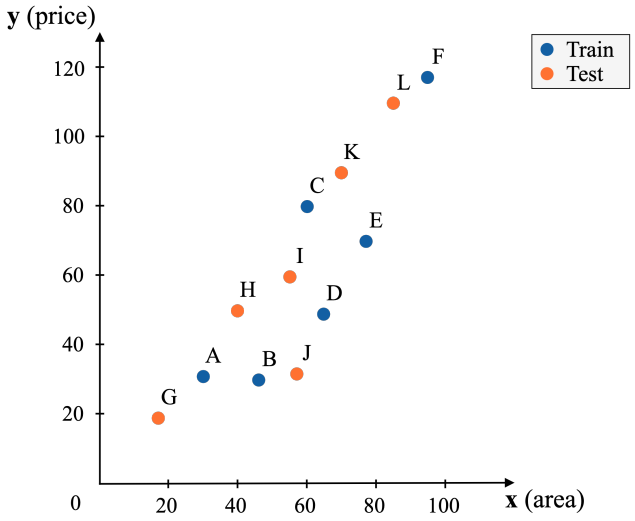


Figure 4.1: Plot of the training and test data points in our Amsterdam housing example.

4.1.1 Polynomial Basis Expansion

We will show how to fit a model with polynomial basis expansion in the context of an example. Consider the Amsterdam housing prices dataset shown in Table 4.1a which will serve as a training set, and the table next to it which will serve as a test set to evaluate our model. These two datasets are plotted in Figure 4.1. As in the previous section, we would like to build a model that predicts the price of an apartment given its floor area (m^2). But now, rather than learning a linear model, we will learn a *polynomial model*.

Recall from your introductory math classes that the degree of a polynomial $f(x)$ is equal to the largest power of x in the definition of $f(x)$. For example:

- A first-degree polynomial function is a simple linear function and can be represented as:

$$f(x) = w_0 + w_1 x$$

- A second-degree polynomial function can be represented as:

$$f(x) = w_0 + w_1 x + w_2 x^2$$

- A third-degree polynomial function can be represented as:

$$f(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$

- In general, an *n-th* degree polynomial function can be represented as:

$$f(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_n x^n$$

where each w_k is a (real-valued) coefficient (e.g. $w_0 = 5$, $w_1 = -3.3$, and so on). A polynomial of a higher degree is said to be more *complex* than a polynomial of a lower degree.

Let's build and fit three different polynomial functions to the training set - a second-degree, a fourth-degree and a fifth-degree polynomial, and see how each function performs. For each polynomial, we will:

1. Fit the function to the training set, that is, to the ● points in Figure 4.1, ignoring the ● points.
2. Use the function to predict values for training and test sets
3. Use the predicted values to calculate $SSR_{training}$ and SSR_{test}

Note that if we have 6 data points and want to perform polynomial regression, the maximum degree of our polynomial is n-1, where n is the number of data points. In this case, the maximum degree would be 5.

Second-degree polynomial

Let's start with fitting a second-degree polynomial function $\hat{f}(x_i) = w_0 + w_1 x_i + w_2 x_i^2$ to the training set. As in the last section we want to find the values of parameters w_0, w_1, and w_2 that produce the minimum SSR:

$$\min_{w_0, w_1, w_2} \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_1 + w_2 x_2^2))^2$$

We can estimate the weights that give the minimum SSR using gradient descent as in the last section. Because we covered that part, I will not show the exact calculations, but only the final weights $w_0 = 39.04$, $w_1 = -1.65$, $w_2 = 0.04$, and $w_3 = -0.0002$.

Again, this is our model that predicts the price of an apartment located in Amsterdam, given its area. For instance, we know that the real price of an apartment A from Table 4.2b is €310,000. However, if we did not know that price, we could use our model to predict it:

$$\begin{aligned}\hat{f}(x_A) &= w_0 + w_1 x_A + w_2 x_A^2 + w_3 x_A^3 \\ &= 31.9 - 0.5 \cdot 30 + 0.014 \cdot 30^2 \\ &= 29.5\end{aligned}$$

Hence, €295,000 is the predicted price for a 30-squared meters apartment. This is a pretty good prediction, because the actual price of that apartment is 310,000. So the model was wrong by a small

Training Set			
n.	*area* (m^2)	*actual price* (€10,000)	*predicted price* (€10,000)
A	30	31	29.5
B	46	30	38.5
C	60	80	52.3
D	65	49	58.6
E	77	70	76.4
F	95	118	110.8

(a) Training set predictions

Test Set			
n.	*area* (m^2)	*actual price* (€10,000)	*predicted price* (€10,000)
G	17	19	27.4
H	40	50	34.3
I	55	60	46.8
J	57	32	48.9
K	70	90	65.5
L	85	110	90.6

(b) Test set predictions

Table 4.2: Predictions on the Amsterdam housing prices dataset for a second-degree polynomial

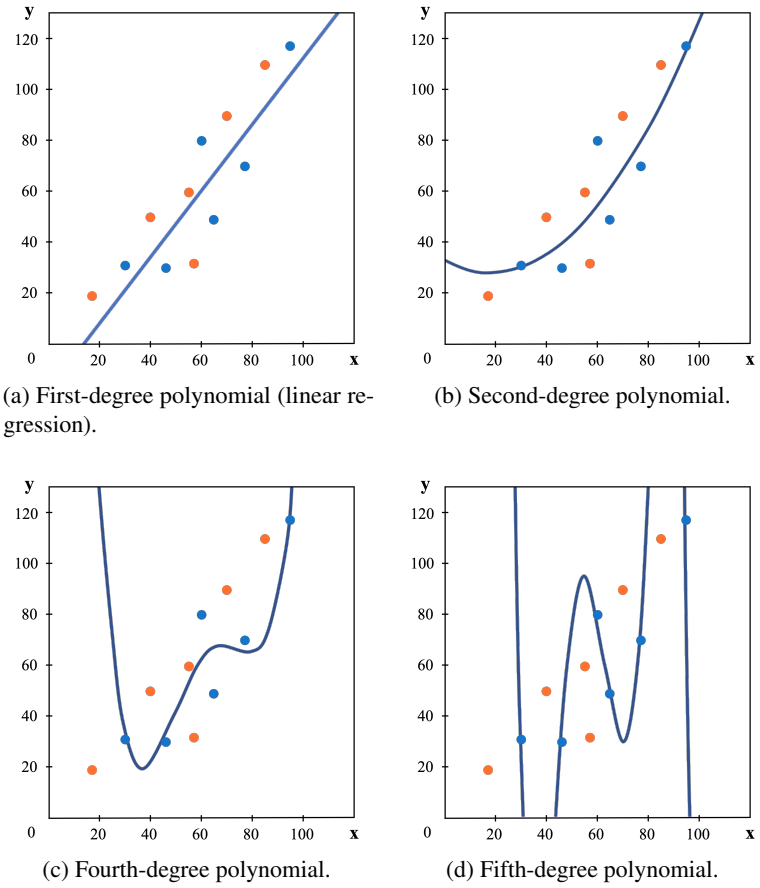
amount of €15,000.[1] Let's predict the prices of the remaining apartments in the training set (I will leave actual calculations as a homework to you).

Now that we have both predicted and actual apartment prices, we can measure how good this model performs overall on a training set by calculating the Sum of Squared Residuals (SSR), that you learned in the previous chapter.

$$\begin{aligned} SSR_{training} &= \sum_{i \in S}^{n} \Big(y_i - \hat{f}(x_i) \Big)^2 \\ &= (y_A - \hat{f}(x_A))^2 + (y_C - \hat{f}(x_C))^2 + \ldots + (y_F - \hat{f}(x_F))^2 \\ &= (31 - 29.5)^2 + (30 - 38.5)^2 + \ \ldots \ + (118 - 110.5)^2 \\ &= 1027.0004 \end{aligned}$$

We've measured $SSR_{training}$ - the total error of a model. It shows how good the model performs on a training set. Remember that $SSR_{training}$ alone does not tell us anything by now. It becomes relevant only when we compare it with other models' $SSR_{training}$. Let's now evaluate how good the model performs on the dataset it has not seen before, i.e. on the test set. For that, we similarly need to predict the prices of the apartments in the test set (again, I will leave actual math as homework to you).

[1] The calculated weights that got us the answer were actually rounded to avoid writing very long numbers, but if we didn't, the answer would actually be 29.96.



(a) First-degree polynomial (linear regression).

(b) Second-degree polynomial.

(c) Fourth-degree polynomial.

(d) Fifth-degree polynomial.

Figure 4.2: Fit of polynomials of different degrees to the Amsterdam housing prices dataset.

Now that we have both predicted and actual apartment prices, we can measure how good this model performs on a test set by calculating Sum of Squared Residuals (SSR):

$$\begin{aligned} SSR_{test} &= \sum_{i=1}^{n} \left(y_i - \hat{f}(x_i)\right)^2 \\ &= (y_G - \hat{f}(x_G))^2 + (y_H - \hat{f}(x_H))^2 + \ldots + (y_L - \hat{f}(x_L))^2 \\ &= (19 - 27.4)^2 + (50 - 34.3)^2 + \ \ldots \ + (110 - 90.6)^2 \\ &= 1757.08 \end{aligned}$$

By comparing $SSR_{training}$ and SSR_{test}, we can say that the model performs approximately five times worse for the test set than for the training set. Let's now leave everything as it is and move to a fifth-degree polynomial model.

Fourth-degree and Fifth-degree Polynomial

The procedure for fitting the fourth and fifth degree polynomial are similar. I will not show the detailed calculations for the sake of clarity. The learned functions of the fourth-degree and fifth-degree polynomial are shown in Figure 4.2c and Figure 4.2d respectively (and learned weights are

Model Comparisons			
polynomial degree	$SSR_{training}$	SSR_{test}	$\sum \lvert w_i \rvert$
2	1027.00	1757.08	32.41
4	688.66	29379.05	945.20
5	0.6	6718669.7	21849.22

Table 4.3: Training and test error on Amsterdam housing prices dataset

detailed below). The training and test error rates for all polynomial functions are collected in Table 4.3. We see that even though a fifth-degree polynomial model has the lowest $SSR_{training}$ compared to the previous two models, it also has the largest SSR_{test}. The huge gap between $SSR_{training}$ and SSR_{test} is a sign of overfitting.

The learned weights for the fourth-degree and fifth-degree polynomial are as follows. For the fourth degree polynomial we learn the model

$$\hat{f}(x_i) = w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3 + w_4 x_i^4 \tag{4.1}$$

where $w_0 = 876.9$, $w_1 = -66.46$, $w_2 = 1.821$, $w_3 = -0.02076$, and $w_4 = 8.49e-05$. We obtain a train error rate of $SSR_{training} = 688.66$ and a test error rate of $SSR_{test} = 29379.05$. For the fifth degree polynomial we learn the model

$$\hat{f}(x_i) = w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3 + w_4 x_i^4 + w_5 x_i^5 \tag{4.2}$$

where $w_0 = 19915.1$, $w_1 = -1866.21$, $w_2 = 66.7535$, $w_3 = -1.144326$, $w_4 = 0.009449443$, and $w_5 = -3.017709e-05$. We obtain a train error rate of $SSR_{training} = 0.6$ and a test error rate of $SSR_{test} = 6718669.714$.

4.1.2 Comparison of Model Weights

Another sign of overfitting can be noticed from the absolute sum of coefficients: the higher the sum, the more the model tends to overfit. This is quite important to grasp to understand further why we need regularization that controls the sum of weights (discussed in the next Section 4.2). Therefore, let's showcase this theory by calculating the absolute sum of the weights for three aforementioned polynomials and see if the absolute sum of weights increase with model complexity.

- For a second-degree polynomial, an absolute sum of weights is:

$$\begin{aligned}\sum_{i=0}^{n=2} |w_i| &= |w_0| + |w_1| + |w_2| \\ &= 31.9 + 0.5 + 0.014 \\ &= 32.41\end{aligned}$$

- For a fourth-degree polynomial, an absolute sum of weights is:

$$\begin{aligned}\sum_{i=0}^{n=4} |w_i| &= |w_0| + |w_1| + |w_2| + |w_3| + |w_4| \\ &= 876.9 + 66.46 + 1.821 + \ldots \\ &= 945.20\end{aligned}$$

- For a fifth-degree polynomial, an absolute sum of weights is:

$$\begin{aligned}\sum_{i=0}^{n=6} |w_i| &= |w_0| + |w_1| + |w_2| + |w_3| + |w_4| + |w_5| \\ &= 19915.1 + 1866.21 + 66.7535 + \dots \\ &= 21849.22\end{aligned}$$

  The weights are recorded in Table 4.3

As we can see, the sum of coefficients increase with the increase in model complexity. In the context of polynomial regression, this makes perfect sense even without calculations: the higher-degree polynomial would have more weight terms added that the lower-degree.

4.2 Regularization

The basis expansion strategy presented in the last section produces a more complex model. Also, we proved that a more complex model has the higher sum of its weights, as shown in section 4.1.2. As discussed in that section, this may overfit the data. One technique to *decrease* the complexity of a model is known as *regularization*. At a high level, regularization puts a constraint on the sum of weights in order to keep the weights small. In other words, regularization constructs a *penalized loss function* of the form

$$L_\lambda(w;X,Y) = \underbrace{L_D(w;X,Y)}_{\text{fit data well}} + \underbrace{\lambda}_{\text{strength}} \cdot \underbrace{R(w)}_{\text{penalize complex models}}$$

where L_D is the data loss function that measures the goodness-of fit, $R(w)$ is the *penalty* term that penalizes complex models, and $\lambda \geq 0$ is a parameter that controls the *strength* of the penalty (relative to the goodness-of-fit). For regression problems, the data loss can be the SSR

$$L_D(w;X,Y) = \sum_{i=1}^{n} (y_i - \hat{f}_w(x_i))^2$$

as we used above. (Note that when $\lambda = 0$, the penalty is zero, so we recover the ordinary least square solution.) We discuss two popular choices of the penalty term R, known as ridge regression (or $L2$ regularized regression) and lasso regression (or $L1$ regularized regression), in the following sections

4.2.1 Ridge Regression

Ridge regression is probably the most popular regularized regression method. It corresponds to using an L2 penalty term

$$R(w) = \sum_{j=1}^{d} w_j^2$$

that computes the sum of squared values of the weights. This penalizes weight vectors whose components are large.

Let's see how ridge regression works in the context of an example. Let's assume we are learning a third degree polynomial: $\hat{f}_w(x_i) = w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3$. The penalized objective is:

$$SSR_{L2} = \arg\min_{\lambda \geq 0} \sum_i (y_i - \hat{f}(x_i))^2 + \underbrace{\lambda \cdot \sum_{j=1}^{k} w_j^2}_{\text{L2 penalty term}}$$

$$= \arg\min_{\lambda \geq 0} \underbrace{\sum_i \left(y_i - (w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3)\right)^2}_{\text{fit the training data well}} + \underbrace{\lambda \cdot (w_0^2 + w_1^2 + w_2^2 + w_3^2)}_{\text{keep weights small}}$$

We can see how λ affects the learned model by looking at the model's behavior at its extremes:

- When $\lambda = 0$, the penalty has no effect and we obtain the ordinary least squares (OLS) solution. When λ is close to 0, we obtain a model that is close to the OLS solution. In other words, as $\lambda \to 0$, $w_i^{regularized} \to w_i$.
- As $\lambda \to \infty$, the penalty term dominates the data term. This forces the weights to be exactly equal to zero, leaving us with just the intercept $\hat{f}(x) = w_0$. When λ is large, the weights are encouraged to be close to 0. In other words, as $\lambda \to \infty$, $w_i^{regularized} \to 0$, and $\hat{f}(x) \to w_0$

4.2.2 Choosing Regularization Strength λ

$$\lambda^* = \arg\min_{\lambda} \sum_{i=1}^{n'} \mathscr{L}(f_{\theta_\lambda^*}(x_i'),\, y_i')$$

The learning is then performed with a *restricted loss function*:

$$\theta_\lambda^* = \arg\min_{\theta \in S_\lambda} \sum_{i=1}^{n} \mathscr{L}(f_\theta(x_i),\, y_i) \tag{4.3}$$

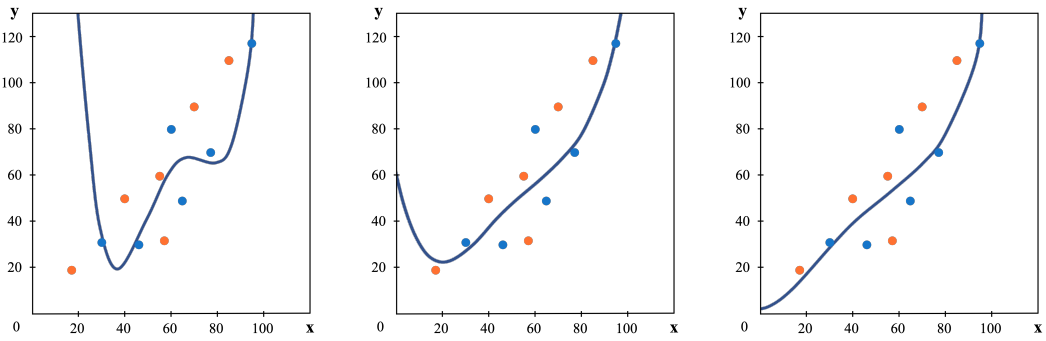
where λ stands for the set of hyper-parameters and S_λ is the set of allowable models restricted to these hyper-parameters. (In this chapter, we use this notation at a very high level; don't worry about the details.) How do we select the best hyper-parameters to use? Ideally, we'd evaluate the learned model on an independent validation set, for each different setting of the hyper-parameter(s) λ. We'd then select the λ giving the best validation performance. Mathematically, we would solve

$$\lambda^* = \arg\min_{\lambda} \sum_{i=1}^{n'} \mathscr{L}(f_{\theta_\lambda^*}(x_i'),\, y_i')$$

In practice, we usually don't have an independent test set and must resort to cross-validation or similar methods to evaluate the quality of the candidate model (as discussed in the previous section.)

4.2.3 Lasso Regression

L1 penalty term puts a constraint on the *sum of the absolute values* of the model weights in order to keep the weights small. That makes the model try to balance between *fitting the training data well and keeping the weights small* to avoid overfitting. We can observe different values for lambda on Figure 4.4a.



(a) No regularization.

(b) with L1 regularization term, $\lambda = 1$.

(c) with L2 regularization term, $\lambda = 1$.

Figure 4.3: Degree 4 polynomial fit for different regularization methods: (a) unregularized fit, (b) L1-regularization with $\alpha = 1$, (c) L2 Regularization with $\alpha = 1$

L1 regularization differs from L2 regularization in its penalty term

$$R(w) = \sum_{j=1}^{d} |w_j|$$

For our example third degree polynomial we have:

$$SSR_{L1} = \underset{\lambda \geq 0}{\arg\min} \; (y_i - \hat{f}(x_i))^2 + \underbrace{\lambda \sum_{j=0}^{k} | \, w_j \, |}_{\text{L1 penalty term}}$$

$$= \underset{\lambda \geq 0}{\arg\min} \; \underbrace{\left(y_i - (w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3)\right)^2}_{\text{fit the training data well}} + \underbrace{\lambda \cdot \left(\, | \, w_0 \, | + | \, w_1 \, | + | \, w_2 \, | + | \, w_3 \, | \, \right)}_{\text{keep weights small}}$$

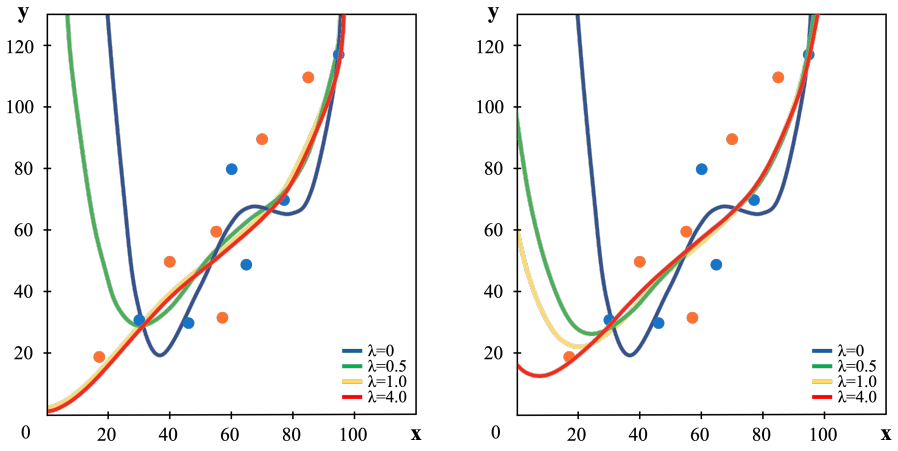
where $\lambda \geq 0$ controls the strength of the penalty. The $L1$-regularized linear regression is also known *Lasso Regression*. The $L1$-regularized model behaves the same as the $L2$-regularized model at the extremes of λ, that is:

- As $\lambda \to 0$, $w_{regularized} \to w_i$
- As $\lambda \to \infty$, $w_{regularized} \to 0$, $\hat{f}(x) \to w_0$

4.2.4 Comparison between L1 and L2 Regularization

If both L1 and L2 regularization behave the same at the extremes, what's the difference between them? The main difference is that L1-regularization shrinks many coefficients to be *exactly* 0. This produces a sparse model, which can be attractive in many problems, especially those that can benefit from features elimination.

Figure 4.3 shows the difference between an L1 and L2 fit.



(a) L1 regularization.

(b) L2 regularization.

Figure 4.4: Degree 4 polynomial with different lambda values for L1 (left) and L2 (right) regularization.

Try It Now in Python

Scan the following QR code to get a git repository that provides a step-by-step guide to coding the following topics in Python:

1. **Basis expansion and regularization**: a second-/third-degree polynomial functions; L1 and L2 regularization terms;
   : `basis_expansion_and_regularization.ipynb`

Key concepts

- Basis expansion
- Regularization, L1 and L2 penalty terms

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Understand how to increase model's complexity through basis expansion
- Understand how to use regularization penalty terms to decrease overfitting.
- Understand the difference between L1 and L2 penalty terms.

5. Model Selection

Aims and Objectives

The aim of this chapter is to explain:

1. Bias error, variance error, irreducible error
2. Validation methods

Learning Outcomes

By the end of this chapter, you will be able to understand:

- What types of errors any model consists of, and how to decrease/minimize them.
- How to mathematically decompose bias and variance errors from a cost function.

In the previous chapter we showed how a ML algorithm learns a model from training data. In this chapter we learn about another important part of the ML pipeline: selecting a model that will perform well on unseen (test) data. The challenge is to select a model that is complex enough to capture the details of the training data but not too complex that it (nearly) memorizes the data – in other words, we want to select a model that neither *underfits* nor *overfits* the training data. In this chapter, we discuss how to select a model that balances these two competing desires from both a theoretical and a practical point of view:

- From a theoretical perspective, we discuss the *bias-variance decomposition* which provides insight into the problems of underfitting and overfitting.
- From a practical perspective, we discuss methods like *cross-validation* which provide estimates of a model's generalization performance which can be used to select a model that will perform well on unseen data.

5.1 Bias-Variance Decomposition

Recall that we saw the problems of over-fitting and underfitting with our fruit dataset in Section 2.1.3. Typical curves illustrating the error as a function of model complexity are shown by the solid lines in Figure 5.1. As the model complexity increases (left to right on the graph), the training error (black

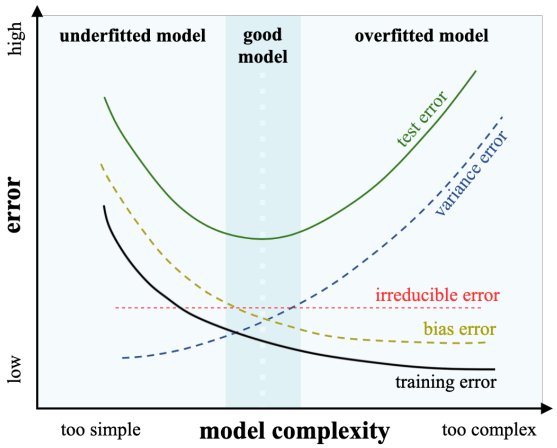


Figure 5.1: Model error as a function of model complexity. The solid black and green curves show training and test error, respectively; as model complexity increases, training error decreases, while the test error hits a minimal value in the middle then begins to increase again. The test error can be decomposed into three theoretical error sources: bias error, which decreases as model complexity increases; variance error, which increases as model complexity increases; and irreducible error which is constant for all models. These error sources are unobserved and hence illustrated with dashed lines.[5-1]

curve) monotonically decreases, meaning that we continually fit the data better. On the other hand, the test error (green curve) is initially large (resulting from a low complexity model that underfits the data), decreases until it hits a good model (which adequately balances overfitting and underfitting), then begins to increase monotonically (resulting from a high complexity model that overfits the data).

In this chapter, we analyze the overfitting and underfitting problems in more detail using a mathematical decomposition of the error known as the *bias-variance decomposition*. At a high level, the bias-variance decomposition decomposes the error as follows:

$$\begin{aligned} \text{Error} &= \text{Irreducible Error} + \text{Reducible Error} \\ &= \text{Irreducible Error} + (\text{Bias Error} + \text{Variance Error}). \end{aligned}$$

That is, the error is first decomposed into an irreducible component, which represents error inherent in the problem, like noise in the labels, that no model can reduce, and a reducible component, which represents error resulting from our choice of model (including its hyperparameters). The reducible error is further decomposed into a bias error that measures the average error over different training sets, and a variance error that measures how sensitive the model is to randomness in the training set. In summary:

Irreducible Error: The irreducible error (red curve) is flat, indicating that it is a constant source of error inherent in the problem. In other words, it is an error source that no model can reduce.

Bias Error: The bias error (yellow curve) is large when the model has low complexity then monotonically decreases as the model complexity increases. The bias error indicates the extent to which the model *underfits* the data.

Variance Error: The variance error (blue curve) is small when the model has low complexity then monotonically increases as the model complexity increases. The variance error indicates how

sensitive the learned model is to perturbations in the training data. The variance error indicates the extent to which the model *overfits* the data.

These three error components are illustrated by the dashed lines in Figure 5.1. Note that the sum of the three components of error (dashed curves) equals the test error (green curve) for each model complexity (indexed on the x-axis).

Figure **??** contains example models that illustrate over-fitting and underfitting. In each subfigure, the training data is shown as black dots and the learned model is shown as a blue line. The model in the first figure *underfits* the data: it predicts the same value (the average value on the training set) for each data point. On the other hand, the model in the last figure *overfits* the data: it interpolates the function at the training data points, producing a perfect prediction at each training point (zero error); however, we expect this rough curve to perform poorly on new data. The model in the middle figure balances the two and seems to be a preferable model. The next section will dissect over-fitting and underfitting in mathematical detail and will use these models as examples.

5.1.1 Mathematical Definition

We now present the mathematical definition of the bias-variance decomposition. We assume the response variable y can be written as $y = f(x) + \varepsilon$ for a given input vector x where f is a *true* deterministic function and ε is a noise random variable with mean zero and variance σ (representing the irreducible error).

The bias-variance decomposition analyzes the mean-squared error of a function $\hat{f}$ fit to a finite sample of n training data points. It measures the error by taking an average, or *expectation*, of the fit $\hat{f}$ over random datasets of size n. We write this as

$$MSE = E\left[\left(y - \hat{f}(x)\right)^2\right] \tag{5.1}$$

where the expectation operator E averages over everything that is random in the fit: all possible training sets of size N and the noise in the response variable.[1] We then write

$$E\left[\left(y - \hat{f}(x)\right)^2\right] = E\left[\left((f(x) + \varepsilon) - \hat{f}(x)\right)^2\right] = E\left[\left(\left(f(x) - \hat{f}(x)\right) + \varepsilon\right)^2\right] \tag{5.2}$$

where we substituted the definition for y using the true function f and noise variable ε, and then regrouped the terms. We then isolate reducible and irreducible components of the error by rewriting it as:

$$\begin{aligned} E\left[\left(\left(f(x) - \hat{f}(x)\right) + \varepsilon\right)^2\right] &= E\left[\left(f(x) - \hat{f}(x)\right)^2\right] + 2E\left[\left(f(x) - \hat{f}(x)\right)\varepsilon\right] + E[\varepsilon^2] \\ &= E\left[\left(f(x) - \hat{f}(x)\right)^2\right] + 2E\left[\left(f(x) - \hat{f}(x)\right)\right]\underbrace{E[\varepsilon]}_{=0} + \underbrace{E[\varepsilon^2]}_{=\sigma^2} \\ &= \underbrace{E\left[\left(f(x) - \hat{f}(x)\right)^2\right]}_{\text{reducible error}} + \underbrace{\sigma^2}_{\text{irreducible error}} \end{aligned} \tag{5.3}$$

[1] Recall from basic probability theory that the expectation operator $E[\cdot]$ averages the value of the operand with respect to an implicitly defined probability distribution. For example, the expectation of a function g with respect to a probability distribution p is defined as $E[g(x)] = \int p(x) \cdot g(x)\, dx$.

The reducible error can be further decomposed into bias and variance components as follows. First, we write

$$E\left[\left(f(x)-\hat{f}(x)\right)^2\right] = E\left[\left(\left(f(x)-E\left[\hat{f}(x)\right]\right)-\left(\hat{f}(x)-E\left[\hat{f}(x)\right]\right)\right)^2\right] \tag{5.4}$$

where we subtracted $E\left[\hat{f}(x)\right]$ from one term and added it to the other inside the parentheses. We then rewrite this equation as follows:

$$\begin{aligned}
&E\left[\left(\left(f(x)-E\left[\hat{f}(x)\right]\right)-\left(\hat{f}(x)-E\left[\hat{f}(x)\right]\right)\right)^2\right] \\
&= \underbrace{E\left[\left(f(x)-E\left[\hat{f}(x)\right]\right)^2\right]}_{\text{Bias}(\hat{f}(x))^2} + \underbrace{E\left[\left(\hat{f}(x)-E\left[\hat{f}(x)\right]\right)^2\right]}_{\text{Var}(\hat{f}(x))} - 2E\left[\left(f(x)-E\left[\hat{f}(x)\right]\right)\cdot\left(\hat{f}(x)-E\left[\hat{f}(x)\right]\right)\right] \\
&= \text{Bias}(\hat{f}(x))^2 + \text{Var}(\hat{f}(x)) - 2\left(f(x)-E\left[\hat{f}(x)\right]\right)\cdot\underbrace{\left(E\left[\hat{f}(x)\right]-E\left[\hat{f}(x)\right]\right)}_{=0} \\
&= \text{Bias}(\hat{f}(x))^2 + \text{Var}(\hat{f}(x))
\end{aligned} \tag{5.5}$$

where the first equality follows from the simple algebraic identity $(a-b)^2 = a^2+b^2-2\cdot a\cdot b$ where $a = f(x)-E\left[\hat{f}(x)\right]$ and $b = \hat{f}(x)-E\left[\hat{f}(x)\right]$, the second equality moves the expectation operator inside on the third term. Putting together the decomposition of error into reducible and irreducible components from Eq. (5.3) and the decomposition of reducible error into bias error and variance error from Eq. (5.5), we have:

$$\begin{aligned}
MSE &= \underbrace{E\left[\left(f(x)-\hat{f}(x)\right)^2\right]}_{\text{reducible error}} + \underbrace{\sigma^2}_{\text{irreducible error}} \\
&= \text{Bias}(\hat{f}(x))^2 + \text{Var}(\hat{f}(x)) + \sigma^2
\end{aligned}$$

We see that the bias-variance decomposition of the error has the same form that we specified in the previous section, consisting of bias error, variance error, and an irreducible error.

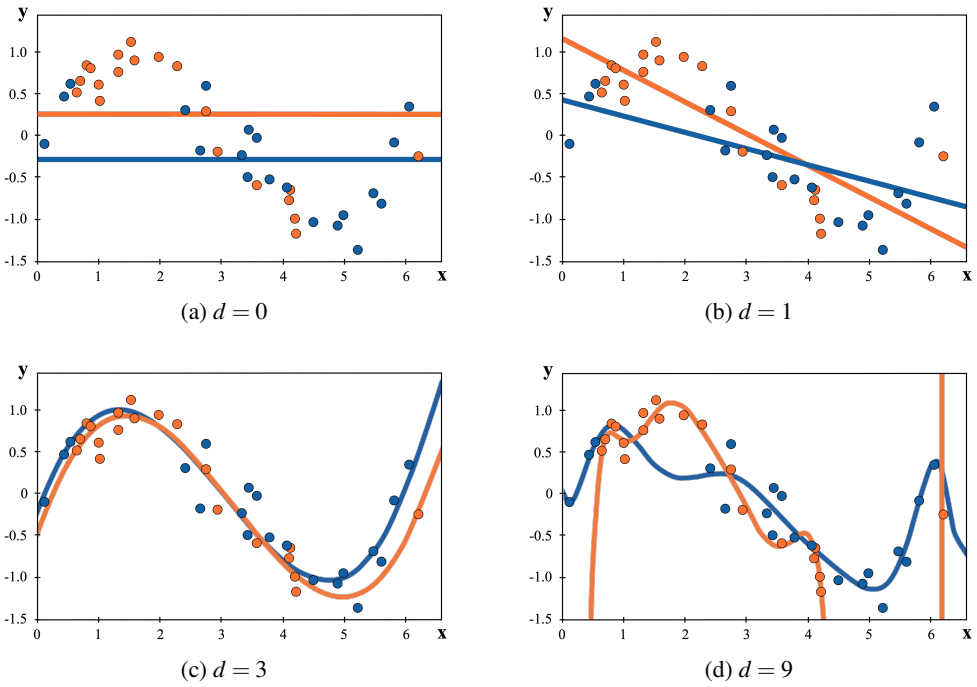
Illustrations

We now illustrate the bias error and variance error using graphs. A model with high variance means that the fit of the function $\hat{f}$ varies a lot given the data. That is, there is randomness in the training set (x values) that we observe; This is illustrated in Figure 5.2.

5.1.2 Diagnosing Bias and Variance Error Sources

It's important to remember that bias and variance are theoretical error sources – they can't be measured in a real problem since the true function f is unknown (of course, if it were known we would have no reason to learn the estimate $\hat{f}$ from data). However, when diagnosing the error of a model, we often try to *qualitatively* diagnose if its dominant error source is bias error or variance error. For example, it is common to hear phrases like:

- "Increasing the model's accuracy on the training set" which means we want to decreasing bias and increase variance. We do this when we suspect we have a high-bias and low-variance model (meaning we are underfitting).



(a) $d = 0$ (b) $d = 1$

(c) $d = 3$ (d) $d = 9$

Figure 5.2: Relation of variance error to model complexity. Each subfigure illustrates a model with fixed complexity (controlled by polynomial degree d) that is fit to two different training data sets, one to the orange data points, and the other to the blue data points. The subfigures represent models of increasing complexity. Note how the fits on each training set, $\hat{f}^{orange}$ and $\hat{f}^{blue}$, are similar for low-complexity models such as (a), (b) and (c) (indicating low variance), while they are very different for high-complexity models, such as (d) (indicating high variance).

- "Making the model more generic" which means decreasing variance and increasing bias. You can hear it when there is a low-bias-high-variance model, or overfitting.

Intuitively, bias can be decreased by making the model more complex using one of the following actions (each of which will be covered in more detail later in the book):

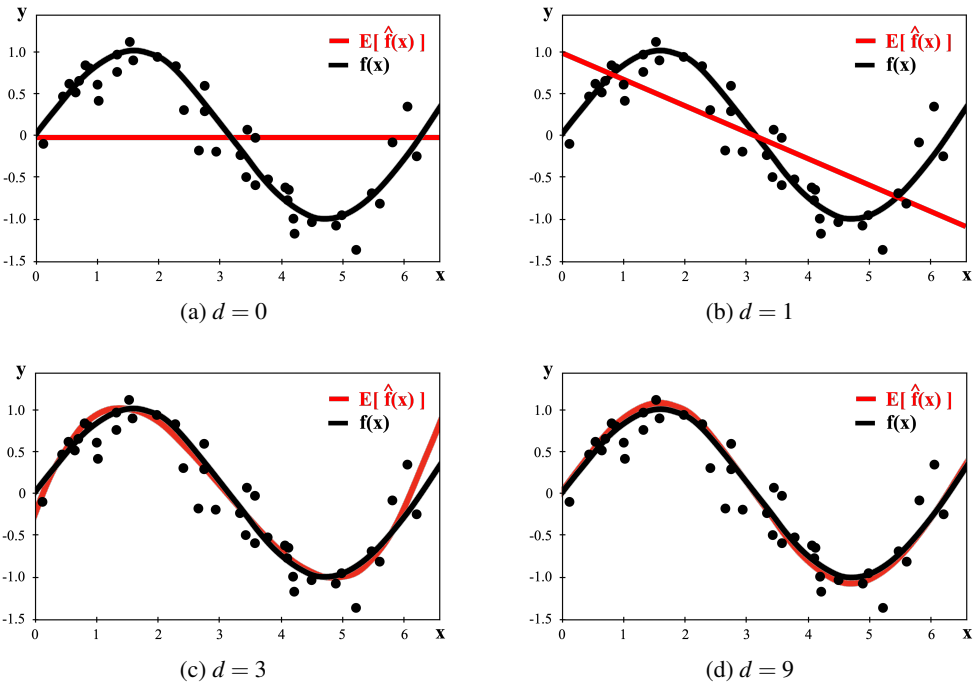
- Choose a more complex functional form for $\hat{f}$ (e.g., a higher degree polynomial)
- Engineer better features from existing feature measurements
- Measure new features

The variance of a model can be decreased by:

- Collecting and labeling more training data
- Feature selection (identify and drop irrelevant features)

5.2 Validation Methods

In the last section we discussed the bias-variance decomposition which shed light on the problems of over-fitting and underfitting from a theoretical perspective. However, in practice we can't directly measure the bias and variance components of the error. Instead, it is often necessary to use *empirical* methods to assess your model's performance and to select a model. We turn our attention to empirical



(a) $d = 0$ (b) $d = 1$

(c) $d = 3$ (d) $d = 9$

Figure 5.3: Relation of bias error, $f(x) - E[\hat{f}(x)]$ to model complexity. Each subfigure shows the true function $f(x)$ in orange (which is known because this is a synthetic example), and the learned model averaged over random samples of the dataset $E_D[\hat{f}(x)]$. (The red curve is approximated by building random samples of the dataset D, fitting $\hat{f}$ to the dataset, and averaging the curves.) The true function f is shown in black in each subfigure (f is known because this is a synthetic dataset; in real datasets f is not known). Note how the fit model matches the true function better as model complexity increases - that is, the bias

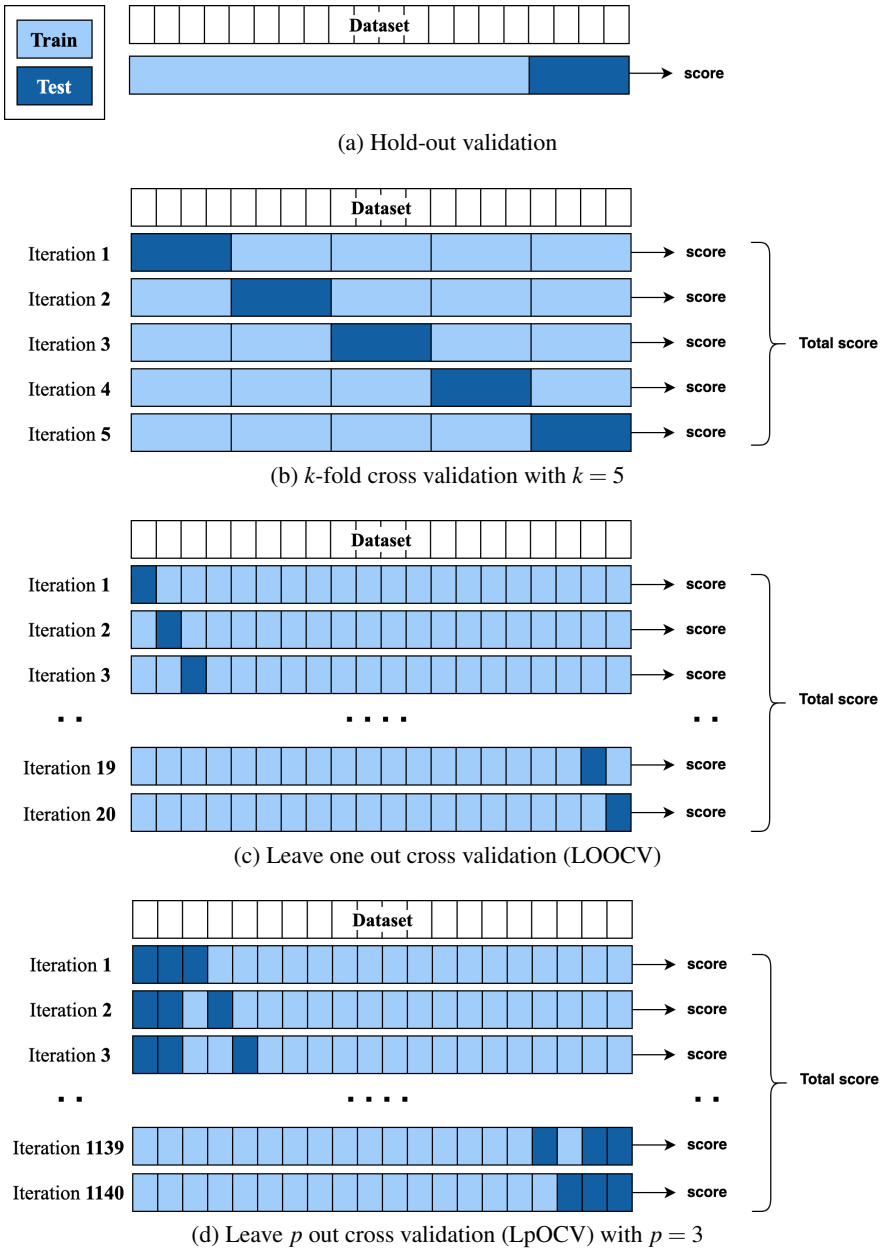
methods in this section.

5.2.1 Hold-out Validation

In Section 2.1.3, we estimated the performance of a model on unseen data using a method known as hold-out validation. In hold-out validation, we first split our data into a training set and a test, or hold-out, set (usually, 80% of the data is used for a train set and 20% for a test set). The model is then trained on the training set, and its performance is evaluated on the hold-out set. Figure 5.4a illustrates how a hypothetical dataset of twenty data points is split using hold-out validation.

The hold-out validation estimator of test performance has two important shortcomings, which are especially prominent when working with small datasets:

- **Losing data for model training:** Setting aside a chunk of data as the hold-out set means you won't be able to use it to train your algorithm. For instance, splitting 20 observations into 16 for a training set and 4 for a test set means losing 4 observations for model training. This loss of data often corresponds to a loss in model accuracy.
- **Skewed training and test sets:** Since the training and test sets are sampled at random, they might be skewed, meaning that they don't accurately represent the whole dataset. Suppose,



(a) Hold-out validation

(b) k-fold cross validation with $k = 5$

(c) Leave one out cross validation (LOOCV)

(d) Leave p out cross validation (LpOCV) with $p = 3$

Figure 5.4: Different ways to split a dataset of $n = 20$ data points into training and validation sets (each data point corresponds to a rectangle in the figures). (a) hold-out validation with 80% train set size and 20% test set size, (b) 5-fold cross-validation, which trains five separate models on different splits of the data, (c) leave one out cross validation (LOOCV) which trains n models each on $n-1$ data points and tests the model on the remaining data point. (d) leave-p-Out Cross Validation (LpOCV) which trains $\binom{n}{p}$ models each on $n-p$ data points and tests the model on the remaining p data points, for all subsets of data of size p. The cross-validation methods generally produce better estimates of generalization error at the expense of increased computational cost.

for example, you use a hold-out set of four out of the twenty data points chosen at random. What if those four points happened to be the four highest values in your dataset? Your testing set would not properly represent the range of values. Such an extreme split may be unlikely in practice, but it is not too unlikely for the split to have an abnormally large concentration of high-valued (or low-valued) data points.

5.2.2 Cross Validation

Cross-validation, or CV, remedies the shortcomings associated with standard hold-out validation for estimating the accuracy of a learned model. Unlike the hold-out method that puts a chunk of the data aside for testing, CV allows the *entire* dataset to participate in both the training and testing process. I know, you are thinking – wait a minute, we've just learnt that training and testing the model using the same dataset is not methodologically correct! So, why are we doing just that now? Well, CV uses a trick such that no data point is used to train and test the same model. The most popular CV technique is known as K-Fold Cross Validation (KfCV) which we discuss in the next section.

K-Fold Cross Validation

Algorithm 5.2.2 summarizes the kFCV process. First, kFCV randomly splits the dataset into k groups, called *folds*. One of the folds is chosen to serve as the test data set, and the other $k-1$ folds are concatenated to form a training set. A model is trained on the training folds and evaluated on the test fold. This process is repeated k times where each fold serves as the test set exactly once. This produces k different performance scores that can be averaged together to produce an estimate on the model's validation performance.

Definition 5.2.1: K-Fold Cross Validation Algorithm

1. Randomly shuffle the dataset and split it into K folds.
2. For each unique fold do the following steps:
   (a) Make one fold a test set (holdout set)
   (b) Concatenate the data in the remaining folds into a training data set.
   (c) Fit a model on the training set
   (d) Evaluate the model on the test fold, store its score, and discard the learned model.
3. Estimate the generalization performance of the model as $\hat{err} = \sum_{k=1}^{K} \hat{err}_k$, the average of the errors on each of the folds.

Figure 5.4b illustrates how the k-fold cross validation splits a dataset with twenty observations into five equal folds, each fold contains four observations. It then uses four folds (sixteen observations) to train the model, and the remaining fold (four observations) to evaluate it. This process is iterated five times, where a different fold serves as the test set at each iteration.

Leave-one-out Cross Validation

In the extreme case, we can perform *leave one out cross validation (LOOCV)* which is equivalent to n-fold cross validation. That is, each data point is its own fold; we train n models, each on $n-1$ data points and test on the remaining data point. The data splitting for LOOCV is illustrated in Figure 5.4c.

Leave-p-out Cross Validation

Leave-p-Out Cross Validation Method (LpOCV) is similar to LOOCV but has a critical difference: at each iteration of LpOCV, p observations, rather than 1 as for LOOCV, are used for validation. In other words, for a dataset with n observations, for every iteration $n-p$ observations will be used as a training set, while p will be used as a test set. LpOCV is exhaustive, meaning that it trains and tests on *all possible* test sets of size p; clearly, this is computationally expensive for larger values of p. For instance, let's say we have a dataset with 20 observations (i.e., $n = 20$), and we want to perform LpOCV. If we set $p = 3$, we get $n - p = 17$ observations for training, and $p = 3$ observations for validation on all possible combinations. Figure 5.4d illustrates all train-validation set splits for a dataset of this size.

Computational Cost and Data Set Size

Cross-validation methods generally provide a more accurate estimate of performance than hold-out validation. However, the obvious downside of cross-validation methods is the increased computational cost that arises from training multiple models on different folds. In the examples used here, hold-out validation is the cheapest but least accurate estimate of generalization performance, 5-fold validation balances cost and accuracy, and leave one out cross validation is the most expensive and most accurate. As a general rule of thumb, 5-fold or 10-fold cross-validation is used. However, for extremely large datasets, hold-out validation may be most appropriate due to the small computational cost, while for extremely small datasets, LOOCV may be appropriate since each data point is crucial for training an accurate model; LpOCV may also be used for small datasets, but we note that it is rarely used in practice due to the high computational cost, requiring $\binom{n}{p}$ models to be trained. Figure 5.5 summarizes the guidelines for choosing a cross-validation method based on the data set size and the computational resources available.

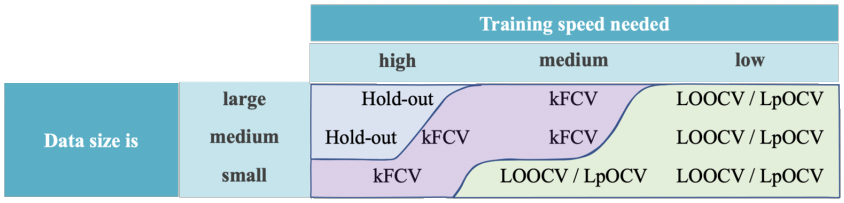


Figure 5.5: Matrix summarizing guidelines for choosing a cross-validation method based on the size of the data set and the computational resources available.

5.3 Unrepresentative Data

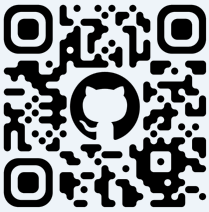
Most machine learning algorithms (and all the ones we cover in this book) assume the data is independently and identically distributed (i.i.d.). This is implicitly assumed in both the bias variance decomposition formulas and the cross-validation methods discussed in this chapter. However, this i.i.d. assumption often does not hold in practice. Sometimes a dataset that violates the i.i.d. assumption is said to be *unrepresentative*. Sometimes an unrepresentative dataset is said to be a *biased* dataset, but this terminology should not be confused with the terminology used in the bias-variance decomposition above – they are distinct problems which unfortunately share the same terminology. In this section, we give some examples where unrepresentative datasets might arise in practice.

One common cause of am unrepresentative dataset occurs when the data is sampled from a subset of the total population. As an example, consider the problem of predicting which US political party, Democrats or Republicans, will win a specific county's midterm election of 2018. We go to a specific neighborhood and start asking people if they would vote for a Democrat or a Republican. Of the 100 people we interview, 32 say they will vote for Democrats, 57 say they will vote for Republican, and 11 are undecided. Based on this data we build a model to make a prediction of who will win the midterm elections. Will this model be accurate? Not even remotely, because the dataset might look way different if we go to a different neighborhood or a different county or state. A single neighborhood is not a good baseline for any kind of political model – unless the model is on who will run the homeowners association or something else on a micro level. We have a very poor assumption about the national results based on the data gathered in one specific neighbourhood of a single city in one state. The data is simply insufficient to make accurate predictions. Our model has an incredibly high bias due to the insufficiently collected data.

Try It Now in Python

Scan the following QR code to get a git repository that provides a step-by-step guide to coding the following topics in Python:

1. **Bias-variance decomposition**: bias error, variance error;
   : `bias_variance_decomposition.ipynb`
2. **Validation methods**: hold-out and cross validation;
   : `validation_methods.ipynb`



Key concepts

- Bias-variance decomposition
- Hold-out and cross validation methods

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Mathematically and conceptually understand bias-variance decomposition
- Understand different validation methods and when they become relevant/applicable.

6. Feature Selection

Aims and Objectives

The aim of this chapter is to explain:

1. The main purpose of Feature Selection.
2. Three groups of feature selection techniques.
3. How main methods of each group work.

Learning Outcomes

By the end of this chapter, you will be able to understand:

- Three main feature selection families.
- Different feature selection procedures and philosophies.

In many ML problems, the dataset contains a large number of features but some of the features are irrelevant to, or are only weakly relevant to, the prediction task. This chapter discusses *feature selection* techniques that seek to identify and remove such features. Feature selection is useful for two reasons: first, it can prevent overfitting and hence increase the performance of the model – i.e., it can prevent the ML algorithm from learning correlations with irrelevant features that are present in training data but not in the test data; second, it leads to models that are easier for a human to interpret – it is difficult to understand a model that uses, say, one hundred features, but if the dataset contains only, say, five relevant features, we can build a model on those five features that is easy to understand. As is the case with data preprocessing, there is no "right" way to perform feature selection – it often requires a lot of trial and error for each problem. This chapter presents some feature selection techniques that are the most popular and useful in practice.

6.1 Introduction

Consider the dataset in Table 6.1 and the task of predicting if a car is a luxury car or not – based on its other features. This dataset contains several features that are either irrelevant to, or are only weakly relevant to, the prediction task. An example of an *irrelevant* feature is the *type* feature. This

doors	horsepower	wheel	type	interior	manufacturer	luxury car
2	500	right	sedan	1	BMW M5	1
4	150	right	sedan	1	Toyota Yaris	0
4	382	left	sedan	1	Toyota Supra	1
2	154	left	sedan	2	Audi A1	0
4	444	right	sedan	1	Audi RS5	1
4	165	right	sedan	2	Volkswagen Polo	0

Table 6.1: Example dataset of car features and car price (target variable).

feature has the same value, *sedan*, for each data point and clearly cannot help distinguish between a luxury and non-luxury car. An example of a feature with weak predictive power is the *wheel* feature (which represents the side of the car that the driving wheel is on). We see that many cars of both classes, luxury and not luxury, have driver's wheel on both the left and right side of the car. Hence, such a feature likely has weak predictive power. Although we discovered this statistically (i.e., by looking at the data), we might have suspected that this was the case *a priori* since we know each manufacturer makes cars with the driver wheel on each side.[1]

The goal of *feature selection* is to systematically identify the features that are the most important, or have the *highest predictive power*, and then train the model only on those features. Feature selection is useful for two reasons:

Prevent Overfitting: Removing irrelevant or weakly relevant features can prevent our ML model from overfitting to the training data set. In the above example, there may be some correlation between driver wheel side and car class in the training data set, but if this correlation is not present in the test set, then whatever we learned about this feature will cause errors at test time.

Interpretability: Removing irrelevant or weakly relevant features can help us interpret, or understand, our ML model better. If a model is built on all of the features, it would be difficult to understand exactly how the predictions behave based on the interactions among features. However, if we remove many features, such as the driver wheel side feature, and leave, let's say, just the manufacturer and horsepower, it is much easier for humans to understand. Removing features improves interpretability even more in large datasets if we can, say, reduce a dataset with millions of features to a dataset with hundreds of features.

So how do we perform feature selection? Sometimes we can identify irrelevant features *a priori*, based on our knowledge of the problem. But many other times we will identify irrelevant features using the properties of the dataset or prediction task. Roughly speaking, there are three different groups of feature selection techniques: filter methods, search methods, and embedded methods. We discuss each of these methods in the following sections, then discuss the similarities, differences, strengths, and weaknesses of each approach in the final section.

[1] It is important to note that although it might not "make sense" *a priori* for a feature to possess predictive power, the feature may in fact have high predictive power. For example, if the relative wealth of people in countries with right-handed cars is higher that that of people in countries with left-handed cars, more luxury cars with right-handed wheels will probably be made.

6.2 Filter Methods

The simplest family of feature selection techniques is known as *filter methods* which remove features before performing any learning. The basic approach of filter methods is to compute a statistical measure of the *relevance* of each feature. The features are then ranked by their relevance score and either the top K features or all features with score larger than a threshold τ are kept, where K or τ is a user-specified hyper-parameter. Filter methods differ from one another by their choice of *relevance* score. Broadly speaking, there are two types of relevance scores:

Univariate score: A relevance score can be computed for each feature *individually*, i.e., without looking at other features. Some techniques examine the statistics of the single feature, while others examine the correlation between the feature and the class label. The basic idea for the first group of techniques is that features with small difference in value provide little differentiation between data points. The basic idea for the second group of techniques is that features that are nearly independent of the target variable are of little use for classification.

Multivariate score: Another class of methods looks at *interactions* among features. The underlying idea is to remove features that are highly correlated with each other. In the extreme case, two features have identical values for all data points, i.e., one feature is a duplicate of another. Clearly, such the duplicate adds no new information to help the classification task. In practice, features will not be exactly identical, but we may be able to identify and remove highly correlated features.

A filter method combines one of the qualitative ideas above with a quantitative computation to produce a relevance score. Some filter methods use heuristic scores while others use more rigorous statistical methods which test, for example, the independence of a feature variable and the target variable. There are too many filter methods to discuss in detail in this section. Instead, we will describe and show a detailed calculation of one method from each group as a representative of that group.

6.2.1 Univariate Selection

We first examine filter methods that examine features individually, i.e., without examining interaction between features. There are multiple filter methods, such as Chi-squared score, or ... that you can explore in your free time. In this section, we discuss the easiest one among them - variance score.

Variance score

The simplest method for univariate feature selection uses a simple variance score of the feature. Simply put, you calculate the *variance* of all the values for a particular feature. Recall that variance, or mean square, can be computed in general as the average of the squares of the deviations from the mean:

$$\text{variance} = s^2 = \frac{\sum y^2}{n}$$

where each y is the difference between a value and the mean of all the values. The variance for feature j would thus be:

$$Var(x_j) = \frac{\sum_{i=1}^{n} x_j}{n}$$

You can see that if all the values for a particular feature were identical, the variance in them would be zero, and they would provide no information at all in the ability to discern a feature that is useful for model creation from one that it not useful. For example, consider the features in Table 6.1. We have already reasoned that the *type* feature has no value, because all its values are the same in the dataset. Now let's look at the *horsepower* feature. If we compute the variance for this feature, we first compute the mean, which in this case is 299.2. Then we compute the average squared deviation from the mean, which is the variance, and find that it is 25,899. This number can be used as the variance score and compared to other scores for data that span similar scales.

Of course we can't legitimately compare this variance with that obtained from the *interior* feature, because that latter variance is about 0.27, several orders of magnitude smaller. But what if we had another feature, say *torque*. Here we might find the values in our dataset range from 200 to 500, and give a variance of 20,000. Then our top two features in terms of variance would be *interior* and *torque*, and we might find that these are the most powerful in our model to predict luxury.

Chi-squared score

A chi-square test is used in statistics to test the independence of two events. For our feature selection task, we are interested in testing the independence between a specific feature variable and the target variable. A feature that is completely independent of the target variable is irrelevant to the prediction task and can be dropped. In practice, the Chi-square test measures the degree of dependence between the feature and the target variable, and we drop features with the worst score.

The Chi-square score assumes that both the input variables and the target variable are categorical. For example, you might want to know if the location of the wheel affects the status of a cat. If Chi-squared score shows that it doesn't affect the car status in any way, we can drop this feature from the dataset.

For example, you might want to know if the location of the wheel affects the price. If it does not affect the price in any way, we can drop this feature from the dataset. Note that the Chi-square score assumes that both the input variables and the target variable are categorical.

6.2.2 Multivariate Selection

Sometimes it's useful to know if one feature affects another feature. We now turn our attention to methods that take interaction between features into account.

Correlation-based Feature Selection

Correlation is one of the easiest filter methods. It calculates the correlation between each feature in the dataset and a target variable.

Correlation-based Feature Selection (CFS) is a simple filter algorithm that ranks feature subsets according to a correlation based heuristic evaluation function. The central idea is to identify subsets of featues that are highly correlated with the class but are uncorrelated with each other. Irrelevant features should be ignored because they will have low correlation with the class. Redundant features should be screened out as they will be highly correlated with one or more of the remaining features. The acceptance of a feature will depend on the extent to which it predicts classes in areas of the instance space not already predicted by other features.

Fisher Score

The key idea of Fisher score is to find a subset of features, such that in the data space spanned by the selected features, the distances between data points in different classes are as large as possible, while the distances between data points in the same class are as small as possible. In other words, the between-class variance of the feature should be large, while the within-class variance of the feature should be small.

In particular, given dataset $\{x_i, y_i\}_{i=1}^N$, where $x_i \in R^M$ and $y_i \in \{1, 2, .., c\}$ represents the class which x_i belongs to. N and M are the number of samples and features, respectively. $f_1, f_2, .., f_M$ denote the M features. We are interested in selecting a good subset from the M features in order to make the dataset more productive. Then the Fisher score F of the i-th feature (f_i) is computed below:

$$F(f_i) = \frac{\sum_{k=1}^{c} n_k(\mu_i^k - \mu_i)^2}{\sum_{k=1}^{c} \sum_{y_i=k}(f_{i,j} - \mu_i^k - \mu_i^k)^2} = \frac{\sum_{k=1}^{c} n_k(\mu_i^k - \mu_i)^2}{\sum_{k=1}^{c} n_k(\sigma_i^k)^2} = \frac{\sum_{k=1}^{c} n_k(\mu_i^k - \mu_i)^2}{\sigma_i^2} \tag{6.1}$$

where n_k is the number of samples in class k, μ_i^k and σ_i^k is the mean and standard deviation of a feature f_i in a certain k-th class, μ_i and σ_i denote the mean and standard deviation of a feature f_i, and $f_{i,j}$ is the value of feature f_i in sample (*or observation*) x_j.

Definition 6.2.1: Find best subset of features of size K

1. Let *algo* be a learning algorithm, X, y be the training set and K be the desired number of features.

2. For each subset s of features of size K:
   2a. Learn a model with parameters θ using the proposed subset of feature s with data matrix $X[:, s]$, and store the score in an array indexed by s:

$$score[s] \leftarrow \min_{\theta} L(\theta;\, X[:, s],\, y)$$

3. Return the subset s with the best score from our array.

Definition 6.2.2: Step forward feature selection

1. Let *algo* be a learning algorithm, X, y be the training set, and K be the desired number of features.

2. Let $s \leftarrow \emptyset$ represent the set of features we have selected so far (empty at first).
3. For $i = 1 \ldots K$
   // Identify the best feature to add
   3a. For $j \in \{1, \ldots, p\} \setminus s$
      3a.i. Let $s' \leftarrow s \cup \{j\}$ be the proposed feature set with feature j added.
      3a.ii Fit a model using *algo* to the feature-selected training set $X[:, s']$, y
      3a.ii Store the training error in an array *score* indexed by s, i.e., $score[s] \leftarrow error$.
   3b. Identify the j with the best score, and add j to the feature set, i.e., let $s \leftarrow s \cup \{j\}$.
4. Return s, the best subset of features of size k chosen via greedy step forward selection.

6.3 Search Methods

Search methods perform an explicit search for the best subset of features of size K for a particular learning algorithm. For example, "out of all subsets of features of size K, find the best subset of features to use with a decision tree algorithm." The algorithm box in Alg. 6.2.1 details the procedure. One of the main problems with this approach is computational: there are an exponential number, $\binom{p}{K}$, of subsets of size K for a model with p features. For example, when $p = 1000$ and $K = 10$, there are $\binom{p}{K} \approx 2.6 \cdot 10^{23}$ subsets of features of size K.

$$\min_{s:\, s \subset \{1,\ldots,p\}, |s|=K} \min_{\theta} L(\theta; X[:,s], y)$$

For most practical problems, performing the exact search is infeasible, and we must resort to approximate search methods. The two most popular approximate methods are the forward-selection and backward-selection algorithms. Each of these algorithms performs the approximate search in a greedy fashion as follows:

- Step forward feature selection starts with an empty set of features. At each iteration, it identifies the *most relevant* feature and adds it to the set. It identifies this feature by brute force: for each feature it fits a model with that feature added to the current features and records the model's score; the most relevant feature is the feature whose addition causes the largest improvement to the score. Step forward feature selection is presented in Algorithm 6.2.1.
- Step backwards feature selection starts with the set of all features. At each iteration, it identifies the *least relevant* feature and remove it from the set. It identifies this feature by brute force: for each feature it fits a model with that feature removed from the current features and records the model's score; the least relevant feature is the feature whose removal had the smallest decline in the score. Similar to the algorithm for forward feature selection in Alg. 6.2.1, we can write down an algorithm for backward feature selection, but this is omitted for space.

Recursive Feature Elimination

Recursive feature elimination (RFE) is an approach to feature selection that is similar to step-backwards feature selection. At a high level, RFE begins with the set of all features, then iteratively fits a model and removes the weakest features. However, unlike step-backwards feature selection, RFE may remove multiple features at each iteration and it does not use the training objective to identify weak features. Instead, RFE allows an arbitrary measure of *feature importance* to be plugged-in to identify weak features. For example, in linear regression, one measure of feature importance can be the magnitude of the learned linear coefficients. If we used this metric, then we would eliminate all features whose linear coefficients are below some threshold at each iteration of RFE.

6.4 Embedded Methods

Embedded methods learn which features best contribute to the accuracy of the model **while** the model is being created. The most common type of embedded feature selection methods are regularization methods. For example, adding L1/L2 penalty score against complexity to reduce the degree of overfitting or variance of a model by adding more bias. Here, we add a penalty term directly to the cost function.

Many learning algorithms have their own built-in feature selection methods, which is why we call them **embedded** methods. Embedded methods are used by the algorithms and performed during model training.

6.5 Comparison

In this chapter we saw three different types of feature selection methods. We now provide a brief comparison between them. Broadly speaking, the methods differ in their accuracy, their computational cost, and their generality

They are differentiated in three ways:

Accuracy: How does each feature selection technique affect the final prediction accuracy of the ML classifier? In general, filter methods are at a disadvantage because they perform feature selection without looking at the model class.

Computational Cost: Filter methods are often very fast. Univariate filter methods operate in time $O(n_{classes} \cdot n_{data})$. Correlation based methods must examine all pairs of features and run in time quadratic in the number of features $O(n^2_{classes} \cdot n_{data})$. Search methods, on the other hand are much slower. The step-forward and step-backward

Generality: What learning algorithms is each method compatible with? Both the filter and search methods can be combined with any learning algorithm. However, the embedded methods only apply to learning algorithms that can be modified with a penalized loss function.

These methods are differentiated by the type of information they use, their computational cost, and their accuracy. Filter methods are very fast since they only require computing simple statistics rather than running a full ML training algorithm. However, since they perform feature selection without running a ML algorithm, they can often be inaccurate.

Filter methods: Filter methods identify irrelevant features based solely on the features, without examining the classification task. (Some use class label, but they don't build or assume any particular model.) As an extreme example, a feature that has the same value for each data point clearly has no predictive value. More realistically, a feature with many similar values (e.g., with low variance) will probably have low predictive power. Filter methods are often very fast – they allow you to eliminate a large number of features with a single computation of feature score – however, since they don't take into account the model that will use these features, they are often inaccurate.

Search methods: Search method identify features that are directly relevant to the prediction problem. The basic idea is to define a search for the K best features *outside* of the specific learning algorithm. For example, "out of all possible sets of K features, find the set of K features that gives the best performance using a decision tree classifier." Since there are an exponential number of sets of K features, these methods are often computationally expensive.

Embedded Methods: Embedded methods also utilize prediction accuracy to select features, however they do so *within* the learning algorithm itself. Embedded methods define the learning problem with a *penalized* learning objective that *automatically* does feature selection – in other words, although the penalized objective does not *explicitly* enforce sparsity, it turns out that the optimized solution is in fact sparse. The most popular example of this procedure is $L1$-penalized linear regression.

Try It Now in Python

Scan the following QR code to get a git repository that provides a step-by-step guide to coding the following topics in Python:

1. **Filter methods**: chi-square, fisher score, relief, correlation-based feature selection
   : `filter_methods.ipynb`
2. **Search methods**: step forward feature selection, step backward feature selection, recursive feature elimination
   : `search_methods.ipynb`

Key concepts

- Filter methods, search methods, embedded methods
- Variance score, Chi-squared score, Correlation-based feature selection, Fisher score
- Step Forward Feature Selection, Step Backward Feature Selection, Recursive Feature Elimination

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Understand different groups of feature selection methods, how they differ from each other and where each of them fit best
- Understand few methods from each group

7. Data Preparation

Aims and Objectives

The aim of this chapter is to explain:

1. Data Cleaning
2. Feature Transformation
3. Feature Engineering
4. Class Label Imbalance

Learning Outcomes

By the end of this chapter, you will be able to understand:

- What procedures in data preparation exist.
- Why we need them and how they are performed.

The previous chapters discussed the core elements of the ML pipeline, which assumed the data was in an "ideal" form. Unfortunately, in practice, we are often confronted with datasets with incorrect data, or with data that is correct but is not in a format that can be processed by ML algorithms. Before applying ML algorithms we often need to preprocess the data. While there is no "correct" way to preprocess the data, a number of methods are widely used in practice. This chapter discusses the following methods:

Data Cleaning: Data cleaning seeks to correct that appears to be incorrect. Incorrect data may arise due to human input error, such as misspellings or improper formatting, or data may be missing, duplicated, or irrelevant to the prediction task.

Encoding: ML algorithms require numeric data. However, many datasets contain unstructured data (like strings) or categorical variables (like color) which must be encoded numerically.

Feature Engineering: The goal of feature engineering is to create new features by combining several features that we expect to be important based on our human knowledge of the problem. For example, if a dataset contains features for the total sales per day and number of customers per day, we expect a feature that measures the average sale per customer per day (by dividing

Customer ID	State	City	Postal Code	Ship Date	Purchase ($)
383	Pennsylvania	Drexel Hill	19026	23/08/2020	190
1997	Californai	Sacramento	94229	07/03/2020	243
698	California	Los Angeles	90058	14/09/2020	
1314	Iowa	Fort Dodge	50501	29/02/2020	193
1314	Iowa	Fort Dodge	50501	29/02/2020	193
333	New York	Brooklyn	11249	14-09-2020	298
1996	Washington		98101	19/05/2020	1

(a) Initial dirty dataset. The first row (data point) is clean, while the rest are dirty: the second row contains *incorrect data* (misspelling the state California), the third row contains *missing data* (purchase value missing), the fourth and fifth rows contain *duplicated data* (same value for each feature), the sixth row contains *improperly formatted data* (ship date format), and the seventh row contains a likely *outlier* (purchase value about one hundred times lower than others).

Customer ID	State	City	Postal Code	Ship Date	Purchase ($)
383	Pennsylvania	Drexel Hill	19026	23/08/2020	190
1997	California	Sacramento	94229	07/03/2020	243
698	California	Los Angeles	90058	14/09/2020	220
1314	Iowa	Fort Dodge	50501	29/02/2020	193
~~1314~~	~~Iowa~~	~~Fort Dodge~~	~~50501~~	~~29/02/2020~~	~~193~~
333	New York	Brooklyn	11249	14/09/2020	298
1996	Washington	Seattle	98101	19/05/2020	220

(b) A cleaned dataset with modified values highlighted in red.

Table 7.1: Hypothetical dataset of product orders showing (a) initial 'dirty' dataset, and (b) a (potential) cleaned dataset. The initial dataset in Table (a) cannot be consumed by ML algorithms, while the cleaned dataset in Table (b) can be.

these two raw features) to be useful.

Feature Scaling: Features with very different scales can affect the regularization of ML models, and can also make the learning procedure itself slow. The goal of normalization is to transform the feature values into a similar (or identical) range.

Class Label Imbalance: For some classification problems, datasets contain many more examples of one class than the others, e.g., in detecting credit card fraud probably 0.1% of transactions may be fraudulent while 99.9% may be legitimate.

7.1 Data Cleaning

The first, and perhaps most important, step in any ML project is to carefully examine and understand your data. In practice, you will often find that your dataset is "dirty," meaning that it contains incorrect, missing, duplicated, irrelevant, or improperly formatted data. In addition to dirty data, many datasets contain data points that are legitimate measurements but are *outliers*, meaning that they differ substantially from the other data points (this will be defined more precisely later). The data quality has an enormous influence on the quality of any ML model.[1] Consequently, it is often

[1] You may have heard the phrase "garbage in, garbage out" that is sometimes used to describe this phenomenon.

necessary to preprocess the data to correct or delete the dirty or outlier data. We can then run our ML algorithms on the corrected data set. The following subsection discusses ways to deal with dirty data and the subsequent subsection discusses ways to deal with outlier data. (We emphasise again that data cleaning is often a subjective procedure – there are no hard set rules. It requires a lot of empirical experience to understand datasets and how ML algorithms will be affected by various cleaning procedures.)

7.1.1 Dirty Data

Table 7.1 contains a hypothetical dirty dataset of online product orders. This dataset has a number of issues, such as incorrect data, missing data, duplicated data, irrelevant data, and improperly formatted data, that make it impossible to apply ML algorithms right away. This section discusses methods that can be used to clean this data set such that ML algorithms can be applied to it.

Incorrect Data

Datasets may contain data that is clearly incorrect, such as spelling or syntax errors. In some cases, however, it may be difficult to tell if the data is incorrect or if it is correct but simply unexpected (to us, as humans). The data point in the second row of Table 7.1 has value “Californai” for its state feature, which is clearly a misspelling of the state “California”. If this mistake were left uncorrected, any ML algorithm built on this dataset would treat the two strings “Californai” and “California” differently.

How can we identify incorrect data? Perhaps the most exhaustive way is to hire a human being to go through the data manually to correct it, such as identifying and correcting spelling errors. One way to check whether a particular column has misspelled values is to look at its set of unique values, which is often much smaller than the set of all values itself. You can see how this is done in Python by following "Try It Now" box at the end of this chapter.

Improperly Formatted Data

In some cases, we might have improperly formatted values. For instance, the *Ship Date* column in Table 7.1a includes dates are improperly formatted, leading to misaligned date format. We need to standardize the format for all the dates, since an algorithm would treat the date 19-05-2020 and the date 19/05/2020 as two different dates, even though they are the same date in different formats.

Duplicated Data

Duplicated data is another common problem that arises in practice. For example, Table 7.1a has duplicate observations in rows two and three, and in rows four and five. Duplicate data effectively doubles the weight that an ML algorithm gives to the data point and has the effect of incorrectly prioritizing some data points over others which can lead to a poor model. In some cases, however, the duplicate data are in fact genuine. For example, if two purchases for the exact same amount were made on the same day from the exact same location. In most scenarios, genuine duplicates are very unlikely, but there is no way to know for certain simply by looking at the data. To resolve the issue for certain you would need to use external sources of information (for example, verifying with another department that two identical purchases were in fact made).

There are different methods in python to spot duplicated data. You can learn about them in "Try It Now" box.

Irrelevant Features or Data

Oftentimes, you are presented with a very large dataset where many of the features (columns) or data points (rows) are irrelevant to the prediction task. Some potential examples of irrelevant features and data for our running example are as follows:

Irrelevant features (columns): In addition to customer purchase information, our database may contain a column that tells which internet browser the customer used to make the purchase. It seems likely (though there is no way to know for certain) that this information is irrelevant to the prediction task and its inclusion might cause our learning algorithm to overfit. Hence, we delete this feature column from our data set.

Irrelevant data (rows): Suppose our task is to predict purchase history for customers in the United States. If our dataset contains purchase history from customers in other countries, such as the Netherlands, then those points should be removed from the dataset.

Missing Data

Missing data arises for a variety of reasons. For example, if the data is entered by a human being, he may have forgotten to input one or more values. Alternatively, data may be missing because it is genuinely unknown or unmeasured, such as, for example, a set of survey questions that were answered by some, but not all, customers. A missing value occurs in our running example for the purchase column in row three of Table 7.1.

Some ML algorithms have built-in ways to handle missing data, but most do not. So how should we deal with missing data? One approach is to simply *delete* all data points that have any missing features. If there are not many such data points, this may be an acceptable solution. But if there are many such data points, then a large part of the dataset will be removed and the ML algorithm will suffer significantly. Instead, it is desirable to maintain all of the data points, but fill in the missing values with a 'good' value. There are two different types of data filling procedures, discussed below.

Exact: Sometimes the missing value can be determined exactly. For example, if the US State of an order is missing, but we have its zip code, we can determine its state exactly (assuming we have another table which maps zip codes to states) and fill it into the table.

Imputed: Many times, the missing data cannot be determined exactly and we need to make an educated guess at its value. For numeric data, one popular guess is to choose the mean or median value of the non-missing values of the features. For example, to impute a missing product order, we take the median order total. For categorical data, we can impute the value as the mode, or most likely, value. In cases where an imputed value of a feature is used, it is sometimes useful to create a binary feature that indicates if the input data contained the missing feature or not (i.e., a new feature 'order total was missing') – this provides more information to the learning algorithm which may help it learn a good predictive model.

7.1.2 Outliers

An outlier is an observation that differs significantly from other observations. Outliers may be problematic for one of two reasons: first, an outlier may simply not be representative of data that we will see at test time (in a new dataset); second, many ML algorithms are sensitive to severe outliers and often learn models which focuses too heavily on outliers and consequently make poor predictions on the rest of the data points. On the other hand, outliers sometimes reveal insights into important, though unexpected, properties of our dataset that we might not otherwise notice. There are no hard and fast rules about how to classify a point as an outlier and whether or not to remove it

from the dataset. Usually, you will build ML models several times, both with and without outliers, and with different methods of outlier categorization. This subsection discusses two ways that outlier detection is commonly performed in practice: the first is to use common sense or domain knowledge; the second is to use statistical tests that measure how far a point is from a 'typical' data point.

How can common sense or domain knowledge be used to identify outliers? Consider the purchase value of \$1 in the final row of Table 7.1a. If you know, for example, that the cheapest product in your shop is a \$24 cowboy hat, then clearly the data point is erroneous and does not represent a valid purchase value. In Table 7.1b we fix this erroneous value by filling it with the mean purchase value (i.e., it is treated as if it were a missing purchase value).

How can we use statistical metrics to determine if a data point is an outlier? The simplest way is to identify if a datapoint is too far away from the average value. For example, if $\mu^{(j)}$ is the mean and $\sigma^{(j)}$ is the standard deviation of the j^{th} feature in the dataset, we may want to classify values that are further than k standard deviations from the mean as outliers. That is, a feature value x_j^i with value $x_j^i < \mu^{(j)} - k \cdot \sigma^{(j)}$ or $x_j^i > \mu^{(j)} + k \cdot \sigma^{(j)}$ is considered an outlier. Typically, a value of $k = 3$ standard deviations is chosen.

Let's show how to use statistical metrics to identify outliers in Table **??**, column *Purchase*. The mean of the column's observations is:

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{190+243+193+193+298+1}{6} = 186.3$$

and its standard deviation is

$$\begin{aligned}\sigma &= \sqrt{\frac{\sum(x_i - \bar{x})^2}{n}} \\ &= \sqrt{\frac{(190-186.3)^2+(243-186.3)^2+(193-186.3)^2+(193-186.3)^2+(298-186.3)^2+(1-186.3)^2}{6}} \\ &= 91.41\end{aligned}$$

Suppose we set a range of acceptable values of $k = 3$ standard deviations. Then, any data points below $\mu - 3 \cdot \sigma = 186.3 - 3 \cdot 91.41 = -87.93$ or above $\mu + 3 \cdot \sigma = 186.3 + 3 \cdot 91.41 = 460.53$ is considered an outlier. But since we cannot have a purchase with a negative sum, outlier would be above 460.53. In this data set, there are no outliers present.

7.2 Feature Transformation

In this section, we will explain how we transform feature values in an understandable for an algorithm form.

7.2.1 Feature Encoding

After having cleaned your data, you must encode it in a way such that the ML algorithm can consume it. One important thing you must do is encode complex data types, like strings or categorical variables, in a numeric format. We will illustrate feature encoding on the dataset in Table 7.2, where the three independent variables are *Income*, *Vehicle*, and *Kids*, each of which are *categorical* variables, and the target variable is a person's *Residence* (whether a person lives in downtown or in suburbs).

Amsterdam Demographics				
Age	Income (€)	Vehicle	Kids	Residence
32	95,000	none	no	downtown
46	210,000	car	yes	downtown
25	75,000	truck	yes	suburbs
36	30,000	car	yes	suburbs
29	55,000	none	no	suburbs
54	430,000	car	yes	downtown

Table 7.2: Amsterdam housing dataset.

Amsterdam Demographics				
Age	Income (€)	Vehicle	Kids	Residence
32	95,000	0	0	1
46	210,000	1	1	1
25	75,000	2	1	0
36	30,000	1	1	0
29	55,000	0	0	0
54	430,000	1	1	1

(a) Substitute categorical with numeric variables

Amsterdam Demographics						
Age	Income (€)	Vehicle_none	Vehicle_car	Vehicle_truck	Kids	Residence
32	95,000	1	0	0	0	1
46	210,000	0	1	0	1	1
25	75,000	0	0	1	0	0
36	30,000	0	1	0	1	0
29	55,000	1	0	0	0	0
54	430,000	0	1	0	1	1

(b) *Vehicle* categorical variable with one-hot encoding

Table 7.3: Numerical encoding of categorical features in the Amsterdam housing dataset. (a) uses a direct encoding, while (b) uses a one-hot encoding of the ternary *Vehicle* feature.

How can we convert categorical variables to numeric variables? For binary categorical variables we can simply substitute the values 0 and 1 for each category. For example, for the *Kids* feature we can map value *No* to 0 and *Yes* to 1, and for the *Residence* feature we can map value *Suburbs* to 0 and *Downtown* to 1. For categorical variables with more than two categories, we can perform a similar numeric substitution. For example, for the *Vehicle* feature we can map value *none* to 0, *car* to 1, and *truck* to 2. The substitutions of each categorical variables for numeric variables transform the original dataset in Table 7.2 to the dataset in Table 7.3a.

The direct substitution for multiple categories is valid, though potentially problematic: it implies that there is an order among the values, and that this order is important for classification. In the encoding above, it implies that a vehicle feature of *none* (with encoding 0) is somehow more similar to a vehicle feature of *car* (with encoding 1) than it is to a vehicle feature of *truck* (with encoding

2). If the order of a feature's values is not important, it is better to encode the categorical variable using *one-hot encoding*.[2] One-hot encoding transforms a categorical feature with K categories into K features, one for each category, taking value 1 for the data's category and 0 in all other feature categories. For example, one-hot encoding would split the categorical column *Vehicle* into three columns: *Vehicle_none*, *Vehicle_car*, and *Vehicle_truck*, as shown in Table 7.3.

It is important to keep in mind that the more categories that a categorical variable has, the more columns one-hot encoding will add to the dataset. This can cause your dataset to blow up in size unexpectedly and cause severe computational problems if you are not careful.

7.2.2 Feature Scaling

Many datasets contain numeric features with significantly different numeric scales. For example, the *Age* feature ranges from 27 to 54 (years), while the *Income* feature ranges from €30,000 to €430,000, while the features *Vehicle_none*, *Vehicle_car*, *Vehicle_truck*, and *Kids* all have the range from 0 to 1. Unscaled data will, technically, not prohibit the ML algorithm from running, but can often lead to problems in the learning algorithm. For example, since the *Income* feature has much larger value than the other features, it will influence the target variable much more.[3] But we don't necessarily want this to be the case. To ensure that the measurement scale doesn't adversely affect our learning algorithm, we *scale*, or *normalize*, each feature to a common range of values. The two most popular approaches to data scaling are *feature standardization (or z-score normalization)* and *feature normalization*, described below.

Feature Standardization: In feature standardization, the feature values are rescaled to have a mean of $\mu = 0$ and a standard deviation of $\sigma = 1$. That is, the standardized features are calculated as:

$$x_i^{(j)} = \frac{x_i^{(j)} - \mu^{(j)}}{\sigma^{(j)}} \tag{7.1}$$

where $x_i^{(j)}$ is an observation i of the feature j, $x_i^{(j)}$ is a standardized value for an observation i of the feature j, $\mu^{(j)}$ is a mean value of the feature j, and $\sigma^{(j)}$ is a standard deviation of the feature j.

Feature Normalization: In *feature normalization*, the feature values are converted into a specific range, typically in the interval $[0,1]$. That is, the normalized features are calculated as:

$$\hat{x}_i^{(j)} = \frac{x_i^{(j)} - min^{(j)}}{max^{(j)} - min^{(j)}} \tag{7.2}$$

where $x_i^{(j)}$ is an observation i of the feature j, $\hat{x}_i^{(j)}$ is a normalized value for an observation i of the feature j, $min^{(j)}$ is a minimum value of the feature j, and $max^{(j)}$ is a maximum value of the feature j.

[2] When the ordering of a specific feature is important, we can substitute that order with numbers. For example, if we had a feature that showed a credit score of inhabitants and the values were {bad, satisfactory, good, excellent}, we could replace those categories with numbers {1, 2, 3, 4}. Such features are called *ordinal* features.

[3] This is the case for many ML models, such as linear models we discussed in the last section. However, some ML models like decision trees are invariant to feature scaling.

Age	Income (€)	Vehicle_none	Vehicle_car	Vehicle_truck	Kids	Residence
-0.50	-0.39	1.43	-2	-1.20	-2	1
0.90	0.44	-0.70	2	-1.20	2	1
-1.20	-0.54	-0.70	-2	5.99	-2	0
-0.10	-0.86	-0.70	2	-1.20	2	0
-0.80	-0.68	1.43	-2	-1.20	-2	0
1.70	2.04.	-0.70	2	-1.20	2	1

(a) Standardized Features

Age	Income	Vehicle_none	Vehicle_car	Vehicle_truck	Kids	Residence
0.24	0.16	1	0	0	0	1
0.72	0.45	0	1	0	1	1
0	0.11	0	0	1	0	0
0.38	0	0	1	0	1	0
0.14	0.06	1	0	0	0	0
1	1	0	1	0	1	1

(b) Normalized Features

Table 7.4: Amsterdam demographics dataset from Table 7.2 where the features have been transformed via (a) standardization and (b) normalization.

When should you use standardization and when is it better to use normalization? In general, there's no definitive answer. Usually the best thing to do is to try both and see which one performs better for your task. A good default (and first attempt in many projects) is to use standardization, especially when a feature has extremely high or low values (outliers) since this will cause normalization to "squeeze" the typical data values into a very small range.

Example

We now illustrate the computation of standardized features and normalized features on the Amsterdam demographics dataset in Table 7.3a Let's start by standardizing the *Age* feature. We first compute its mean:

$$\begin{aligned}\mu^{age} &= \frac{\sum_{i=1}^{n} x_i^{age}}{n} \\ &= \frac{32+46+25+36+29+54}{6} \\ &= 37\end{aligned}$$

and then its standard deviation:

$$\begin{aligned}\sigma^{age} &= \sqrt{\frac{\sum_{i=1}^{n}(x_i^{age}-\mu^{age})^2}{n}} \\ &= \sqrt{\frac{(32-37)^2+(46-37)^2+(25-37)^2+(36-37)^2+(29-37)^2+(54-37)^2}{6}} \\ &= 10.03\end{aligned}$$

Each *Age* measurement is then standardized by subtracting the feature's mean and dividing by its standard deviation, as in Equation 7.1:

$$x_1^{age} = \frac{32-37}{10.03} = -0.50 \quad x_2^{age} = \frac{46-37}{10.03} = 0.90$$
$$x_3^{age} = \frac{25-37}{10.03} = -1.20 \quad x_4^{age} = \frac{36-37}{10.03} = -0.10$$
$$x_5^{age} = \frac{29-37}{10.03} = -0.80 \quad x_6^{age} = \frac{54-37}{10.03} = 1.70$$

The remaining features can be standardized using the same logic. Table 7.4a shows the result. (I'll leave the actual computation to you as homework).

The normalized features for the same data set are computed as follows. First, let's go back to Table 7.2 where the natural range of the *Age* feature is 25 to 54. By subtracting 25 from every value, then dividing the result by 54, you can normalize those values into the range [0, 1]. Let's scale *Age* using the normalization method:

$$x_1^{age} = \frac{32-25}{54-25} = 0.24 \quad x_2^{age} = \frac{46-25}{54-25} = 0.72$$
$$x_3^{age} = \frac{25-25}{54-25} = 0 \quad x_4^{age} = \frac{36-25}{54-25} = 0.38$$
$$x_5^{age} = \frac{29-25}{54-25} = 0.14 \quad x_6^{age} = \frac{54-25}{54-25} = 1$$

Following the same logic, we should do the same for all the remaining features (I'll let you do the rest of the math to get a bit more experience). After normalization, we get the dataset on Table 7.4b. Note that the feature columns with binary values [0, 1] do not change – that's because according to the Normalization formula in Equation 7.2, the scaled value of 0 is 0, and 1 is 1.

7.3 Feature Engineering

The previous sections showed the basic preprocessing steps – data cleaning and feature encoding and scaling – that need to be done for nearly all ML problems. After these two phases, generic ML algorithms can, in theory be run on the data sets. However, the performance of the ML algorithm on this raw pre-processed data will often not be very good. Often, it is necessary to design or engineer *features* that will help the prediciton task. Good features should encode some important aspect of the problem; often, a lot of trial and error as well as utilizing expert domain knowledge goes into feature engineering.

This section discusses some of the most popular 'generic' feature engineering approaches – by 'generic' we mean that they are applicable in many domains. This is by no means an exhaustive list (indeed it cannot be, as new applications and new features are constantly being designed). Instead, it is meant to show you some popular techniques, illustrate why feature engineering is so powerful, and give you insight into creating custom features for your problem.

Amsterdam Demographics						
Age	Income (€)	Vehicle_none	Vehicle_car	Vehicle_truck	Kids	Residence
young	95,000	1	0	0	0	1
older	210,000	0	1	0	1	1
young	75,000	0	0	1	0	0
middle	30,000	0	1	0	1	0
young	55,000	1	0	0	0	0
older	430,000	0	1	0	1	1

Table 7.5: Transformed dataset with the age feature binned.

7.3.1 Feature Binning

Feature binning is the process that converts a numerical (either continuous and discrete) feature into a categorical feature represented by a set of ranges, or *bins*. For example, instead of representing age as a single real-valued feature, we chop ranges of age into 3 discrete bins:

$$young \in [ages\ 25-34], \qquad middle \in [ages\ 35-44], \qquad old \in [ages\ 45-54]$$

We can substitute the *Age* integers with the bins in Table 7.5. (And just to be clear, this isn't saying that people are young until 34, middle age until 44, and old at 45; these are just the initial assessment comparing the three data sets.)

Clearly, the binning process throws away information: in the example above we throw away the exact age of a person. So how can binning help an ML algorithm? The answer lies in preventing overfitting: the algorithm is not able to overfit by learning to distinguish between values that are in the same bin. A good designed binning strategy must trade off between appropriately restricting the model (e.g., choosing which bins of ages should be treated the same) while minimizing the amount of relevant information that is lost, where the definition of "relevant" depends on the prediction task. For example, modeling consumer behavior in the U.S. may have a bin for age 18-20 since an important commodity, alcohol, cannot legally be purchased by anyone under the age of 21. On the other hand, for modeling car buying habits, for example, this precise age range may not be important.

The number of bins and the precise range of bins that you use will have a significant impact on the prediction model. So how do we define the bin ranges? In some problems, a set of good bin ranges are be specified beforehand via expert *domain knowledge*. In other problems it is up to us, the data scientists, to define good bin ranges. This is often done with *equal width* binning or *quantile* binning, which we discuss in the following paragraphs. These latter two binning methods utilize a hyperparameter that specifies the number of bins; often this hyper-parameter is tuned just as other hyper-parameters as discussed in Chapter **??**. (Other binning methods are less popular and not covered in this book; but I strongly recommend that you do a bit of research into them when you get a chance. When you start to actually work on your own model training, you'll have a good collection of binning methods to try.)

Domain Knowledge Binning

Sometimes there are standard cut-offs used within a field for a continuous variable. For example, blood pressure measurements may be categorized as low, medium, or high based on the established ranges that define them. However, this is more often the exception than the rule. If you are lucky

enough to need to bin something that has defined categories, consider yourself lucky as you quickly split the data. Otherwise, you'll need to take some time to seriously consider the right way to divide the data before you get started.

Equal Width Binning

Equal width binning divides the range of values of a feature into bins with equal width. Usually, we specify the number of bins as a hyper-parameter K, and then compute the width of each bin as

$$w = \left\lceil \frac{max^{(j)} - min^{(j)}}{K} \right\rceil \tag{7.3}$$

where $max^{(j)}$ and $min^{(j)}$ are the j^{th} feature's maximum and minimum values, respectively. The ranges of the K bins are then

$$\begin{aligned} &Bin\ 1 : [min,\ min + w - 1] \\ &Bin\ 2 : [min + w,\ min + 2 \cdot w - 1] \\ &\quad ... \\ &Bin\ K : [min + (K-1) \cdot w,\ max] \end{aligned} \tag{7.4}$$

As an example of equal width binning, consider splitting the *Age* feature in the Amsterdam demographics dataset into $K = 3$ bins. The bin's width is:

$$w = \left\lceil \frac{max - min}{x} \right\rceil = \left\lceil \frac{54 - 25}{3} \right\rceil = 9.7 \approx 10$$

which we rounded to the nearest integer because Age values are always integers (in this dataset). To calculate each bin's range, we plug the bin width into equation (7.4) and obtain

$$\begin{aligned} &Bin\ 1 : [min,\ min + w - 1] \rightarrow [25,\ 25 + 10 - 1] \rightarrow [25, 34] \\ &Bin\ 2 : [min + w,\ min + 2w - 1] \rightarrow [25 + 10,\ 25 + (2 \cdot 10) - 1] \rightarrow [35, 44] \\ &Bin\ 3 : [min + (x-1) \cdot w,\ max] \rightarrow [25 + (3-1) \cdot 10,\ 54] \rightarrow [45, 54] \end{aligned}$$

7.3.2 Ratio Features

Sometimes it is useful to engineer new features from the existing features in the dataset. For instance, check out Table 7.6 with the information on six different online ads:

1. An ad number (feature *Ad no.*)
2. The total number of people who clicked on an ad (feature *Visitors*)
3. The total number of people who actually purchased something after clicking (feature *Customers*)
4. The total return generated from an ad (feature *Total Return ($)*)

Ads Return Prediction Data Set			
Ad no.	Visitors	Customers	Total Return ($)
1	2360	230	1,301
2	12400	2235	8,439
3	35938	4302	14,422
4	1299	143	870
5	4200	298	3,280
6	3218	689	3,607

Table 7.6: Ads Return Prediction

If we divide the *Customers* feature by the *Visitors* feature, we can obtain a new feature that represents the conversion ratio of a specific ad (Table 7.7).

Ads Return Prediction Data Set				
Ad no.	Visitors	Customers	Conversion Ratio	Total Return ($)
1	2360	230	0.10	1,301
2	12400	2235	0.18	8,439
3	35938	4302	0.12	14,422
4	1299	143	0.11	870
5	4200	298	0.07	3,280
6	3218	689	0.21	3,607

Table 7.7: Ads Return Prediction

This new feature can improve the performance of ML models. Since *Conversion Ratio* is based on both *Visitors* and *Customers*, you might think that we can technically exclude these two. But actually keeping both of these variables can achieve higher accuracy.

For the sake of simplicity, I used the dataset with just a few features to show what Feature Engineering is. In reality, expect the datasets to have 10+, 20+, 100+ columns, or even more.

7.4 Handling Class Label Imbalance

Imagine you obtained the following dataset (Table 7.8) from Silver Suchs Bank. This dataset contains 55 observations of bank transaction over a certain period of time.

Bank Transactions				
#	*date*	*time*	*location*	**Status**
1	21/08/2020	02:00	Amsterdam	Legit
2	24/12/2020	05:19	Dusseldorf	Fraud
3	10/04/2020	18:06	Berlin	Legit
...	...	...	...	...
53	13/03/2020	19:01	Belgium	Legit
54	08/10/2020	15:34	Paris	Legit
55	02/04/2020	23:58	Amsterdam	Fraud

Table 7.8: Bank Transactions Dataset

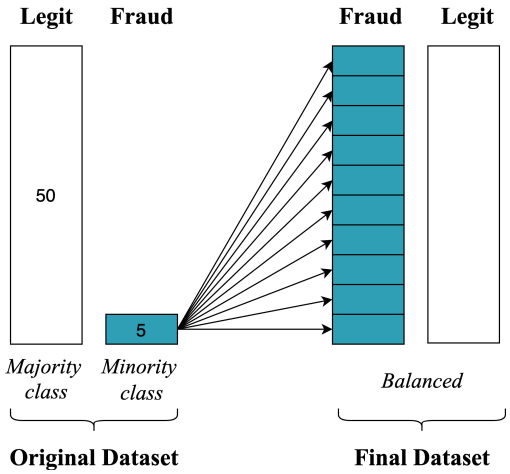


Figure 7.1: Oversampling the Minority Class

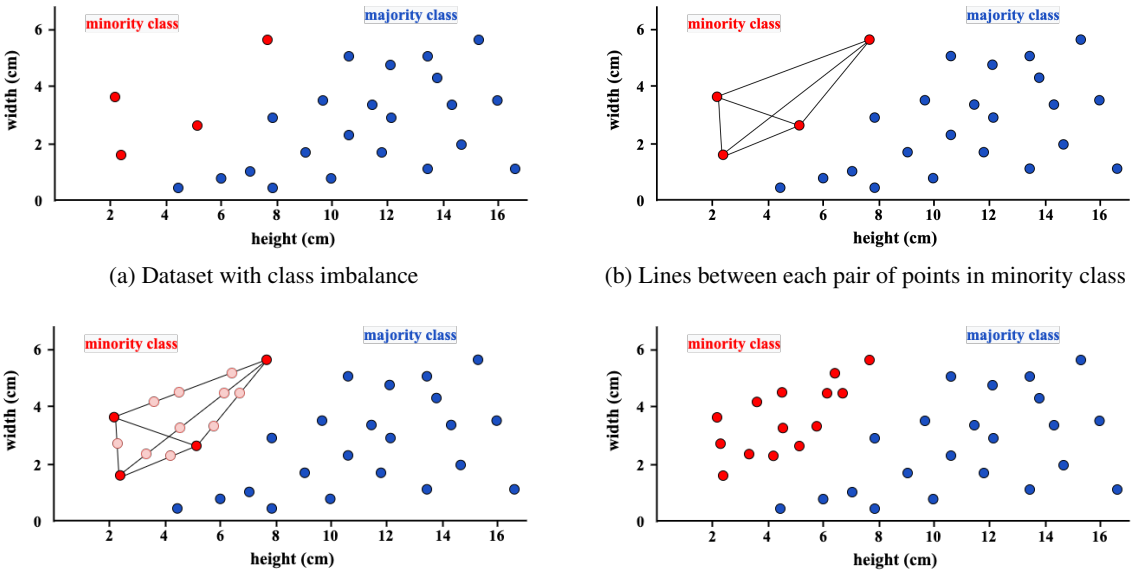
The target column *Status* has two classes: *Fraud* for fraudulent transactions and *Legit* for legal transactions. Imagine that out of 55 observations in the dataset, there are 50 legal transactions (class *Legit*) and only 5 fraudulent transactions (class *Fraud*). These two classes are **imbalanced**.

When we have a disproportionate ratio of observations in each class, the class with the smaller number of observations is called the minority class, while the class with the larger number of observations is called the majority class. In our current example, the class *Fraud* is a minority class and the class *Legit* is a majority class.

Imbalanced classes can create problems in **ML classification** if the difference between the minority and majority classes are significant. When we have a very few observations in one class and a lot of observations in another, we try to minimize the gap. One of the ways to do so is by using oversampling techniques.

7.4.1 Oversampling

Many ML classifiers produce predictions that are biased towards the class with the largest number of samples in the training set. When the classes have wildly different numbers of observations, this can cause the ML algorithm to learn a poor model. For instance, imagine we happen to collect a dataset with 1,000 credit card transactions, where there is only one fraudulent transaction and 999



(a) Dataset with class imbalance

(b) Lines between each pair of points in minority class

(c) Interpolation of data points

(d) Synthetic dataset with balanced classes

Figure 7.2: Illustration of SMOTE algorithm to create a synthetic dataset with balanced classes

non-fraudulent transactions. We use that dataset to train the algorithm. It's likely that the algorithm will almost always predict a transaction to be non-fraudulent.

Oversampling techniques try to balance a dataset by *artificially increasing the number of observations in the minority class*. For our dataset in Table 7.8, 5 out of 55 (or 9%) of the transactions are found to be fraudulent. We might want to increase those 5 fraudulent transactions by 25 (=0.45) or even 50 to avoid discarding the rare class, as shown in Figure 7.1. Another method for augmenting the dataset is with the Synthetic Minority Oversampling Technique (SMOTE), which doesn't simply replicate data points but produces new synthetic data points. The SMOTE algorithm is discussed in the next section.

7.4.2 Synthetic Minority Oversampling Technique (SMOTE)

SMOTE is a statistical technique for artificially increasing the number of observations in the minority class (without affecting the number of observations in the majority class). SMOTE generates artificial observations by combining existing observations from the minority class.

Consider the following example. You went to the supermarket and purchased 4 mandarins and 22 lemons. It's been a very long day with little mental stimulation, so you decide to give yourself a little mental work. Sitting down, you measure the width and length of each fruit, and plot everything on a graph (Figure 7.2a).

Curiosity in what you can get out of this random data collection, you build a binary classifier - an algorithm that would classify mandarins and lemons based on their properties (their width and height). Because there were 22 lemons and only 4 mandarins in the supermarket, you have imbalanced classes, with mandarins in the minority class and lemons in the majority class. Since the number of observations in a mandarin class might be insufficient to train a binary classifier, you decide to use SMOTE to artificially increase observations in that class.

SMOTE synthesises new minority observations between existing minority observations. SMOTE draws lines between existing minority observations similar to what is shown in Figure 7.2b. SMOTE then **randomly** generates new, synthetic minority observations, as shown in Figure 7.2c. Figure 7.2d shows the results. We generated 10 new observations, increasing the number of observations in the minority class from 4 to 14. Now the difference between classes is not so significant.

That is a high level overview of SMOTE. I am going to skip the mathematical definition of SMOTE because it is somewhat complicated. If you are interested, I urge you to test SMOTE in Python. The "How To Code" box below will help you find some very helpful resources.

Try It Now in Python

Scan the following QR code to get a git repository that provides a step-by-step guide to coding the following topics in Python:

1. **Data cleaning**: incorrect data, incomplete data, irrelevant data, duplicated data, and improperly formatted data
   : `data_cleaning.ipynb`
2. **Feature encoding and binning**: one-hot encoding and binning
   : `feature_encoding_binning.ipynb`
3. **Feature scaling**: normalization and standardization
   : `feature_scaling.ipynb`
4. **Data augmentation**: synthetic minority oversampling technique
   : `data_augmentation.ipynb`

Key concepts

- Data Cleaning
- Feature Transformation
- Feature Engineering
- Data Augmentation

A reminder of your learning outcomes

Having completed this chapter, you should be able to:

- Understand what kind of procedures we should/can perform in the data preparation phase.
- Understand how to transform categorical data into numerical, and why.
- Identify outliers in the dataset.

ACKNOWLEDGEMENTS

Without the support and help from a few key contributors, this book would not be possible. I am deeply thankful to the following people in particular:

Alikber Alikberov, Elena Siamashvili, George Ionitsa, Victor Zhou, Anastasiia Tupitsyna

Much appreciation to many other contribute directly and indirectly:

1. **Joshua Starmer**, Assistant Professor at UNC-Chapel Hill
2. **Josh Tenenbaum**, Professor at MIT, Department of Brain and Cognitive Sciences
3. **Guy Bresler**, Associate Professor at MIT, Department of Electrical Engineering and Computer Science

END NOTES

For space considerations, I'm presenting copious (but not comprehensive) citations. I intend these notes as both a trail of the sources used for this book and a detailed entry point into primary sources for anyone interested in some Friday night (or Saturday morning) exploration.

Chapter 1

1-1. Theodoros Evgeniou; *inseaddataanalytics.github.io/INSEADAnalytics*

Chapter 5

5-1. Giorgos Papachristoudis; *towardsdatascience.com/the-bias-variance-tradeoff-8818f41e39e9*

A. Unsupervised Learning

This Appendix provides a brief overview of **unsupervised learning**.

Unlike supervised learning, unsupervised learning algorithms are working with **unlabeled** datasets. That means, we **do not have** a target variable and are no longer able to train or "supervise" our models (hence, the name "unsupervised"). Instead, unsupervised models partition an unlabeled dataset into a certain number of distinct groups. While supervised learning has "classes," unsupervised learning has groups called **clusters**.

The goal is to create **distinctive groups** of data points from an **uncategorized** dataset, so that points in different clusters are dissimilar and points within a cluster are similar (Figure A.1).

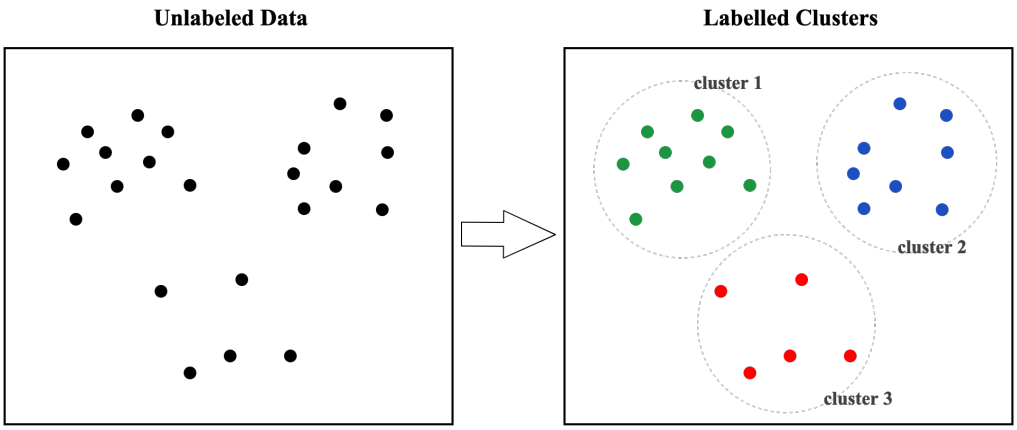


Figure A.1: Clustering Unlabelled Dataset

Let's go back to the example where someone removed your table's last column so that you no longer had the different types of fruit labelled (Table **??**). Let's say I've found your graph (Figure **??**) representing the unlabelled data of fruits you measured, and want to use it to "restore" the fruit class

label from the supervised learning section. In other words, I want to partition this dataset into a chosen number of clusters.

My decision causes several interesting challenges. First, because I do not know how many types of fruits were purchased, I have to consider the number of clusters needed. Second, since I do not know what types of fruits were measured (even if I do manage to partitioning the dataset into the correct number of clusters), I won't be able to identify which cluster represents which fruit. We'll need to tackle these two problems separately.

Because I do not know how many fruit types were measured, I'll start by splitting my algorithm into clusters of 2, 3, 4, 5, and 6 (Figure A.2).

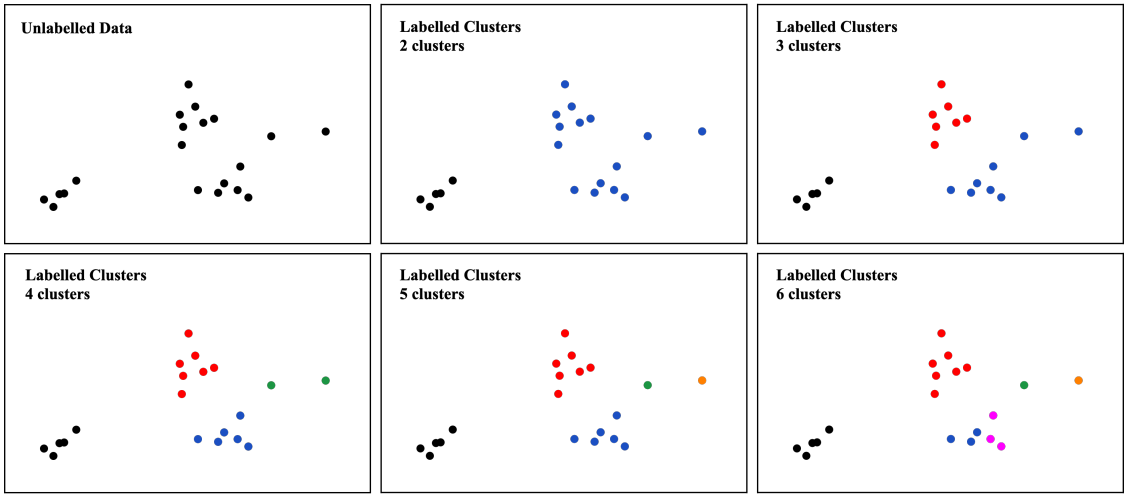


Figure A.2: Selecting Clusters to Partition the Dataset

After careful observations of my output graphs, I noticed that partitioning the dataset into three clusters appears to be the best option. As a result, I concluded that only three fruit types were measured. My decision was based purely on the graphs' appearances, which means I also accepted the risk of being wrong - I can assume there were only three fruits, but I will never know for certain.

Despite having decided how many fruits I think were measured, I can't say what *types* of fruits were measured. They could have been watermelon, kiwi, apple, banana, orange, lemon, or something else entirely. However, since the height (x-axis) and the width (y-axis) are known, if I go to the nearest fruit market and show this graph to a farmer, I'd probably get my answers. Again, I have to accept the risk that some of those answers might be wrong.

Unsupervised learning has two incredibly important aspects to consider when clustering an unlabeled dataset.

1. We need to be careful about determining the number of clusters for the algorithm to partition the dataset.
2. We have to know the market/business to successfully identify each cluster.

B. Non-differentiable Cost Functions

As we learned in Section 3.1.3, gradient descent requires taking the derivative of a cost function. In other words, gradient descent is only applicable if the cost function is differentiable. Unfortunately, not all the functions are differentiable. Generally the most common forms of non-differentiable behavior involve:

1. Discontinuous functions
2. Continuous but non-differentiable functions

B.0.1 Discontinuous Functions

Let's consider the following function $f(x)$:

$$f(x) = x^3 - 2x + 1 \tag{B.1}$$

Is this function differentiable? Yes: we can easily find its derivative $f'(x)$:

$$f'(x) = 3x^2 - 2 \tag{B.2}$$

In mathematics, a continuous function is a function that exists for every value of its domain. In other words, for any value of x, there will be the only one corresponding value of y. Based on this information, we know that the function is **continuous**.

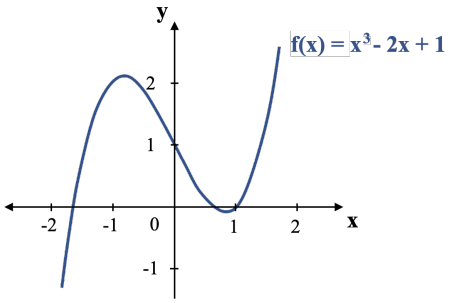


Figure B.1: Continuous Function

What is a discontinuous function then? Consider the following:

$$f(x) = \begin{cases} 3x^2 - 2, & x < 1 \\ 2x - 1, & x > 1 \end{cases}$$

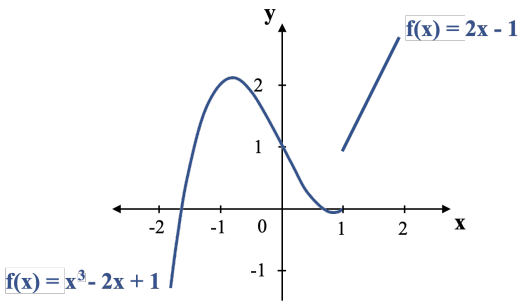


Figure B.2: Discontinous "Jump" Function

This function is not differentiable at $x = 1$ because there is a "jump" in the value of the function: the function is not defined at x=1, so it is not continuous.

Discontinuous functions can also be without any "jumps" in its domain. Let's have a look at the following functions.

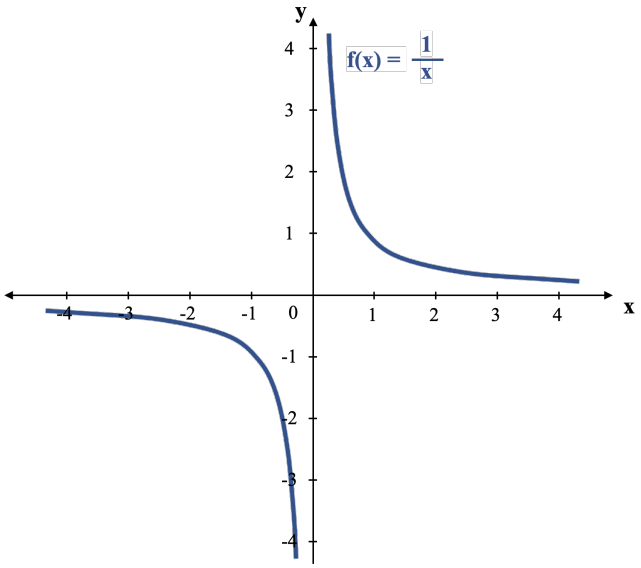


Figure B.3: Discontinous Function

The function on Figure B.3 is not defined at $x = 0$, so it makes no sense to ask if they are differentiable at that point.

Many students assume that a continuous function is also a differentiable function, but that is not always the case: while a differentiable function is always continuous, a continuous function is not always differentiable. In other words, there are continuous but non-differentiable functions. Let's go ahead and check out some examples of these surprising functions.

B.0.2 Continuous Non-differentiable Functions

Consider the following function $f(x)$:

$$f(x) = x^{\frac{1}{3}} \tag{B.3}$$

If we look at this function, we see that it is continuous – there are no jumps.

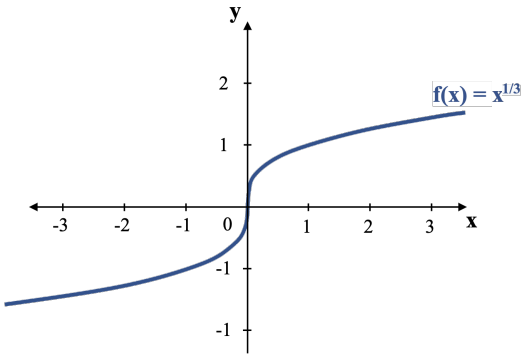


Figure B.4: Continuous Non-differentiable Function

Let's now try to take the derivative of that function:

$$f'(x) = \frac{1}{3}x^{-\frac{2}{3}} \tag{B.4}$$

Now that we have the function's derivative, let's determine the derivative when $x = 0$

$$\begin{aligned} f'(0) &= \frac{1}{3} \cdot 0^{-\frac{2}{3}} \\ &= \frac{1}{3} \cdot \frac{1}{0^{\frac{2}{3}}} \\ &= \frac{1}{0} \end{aligned} \tag{B.5}$$

Since division by 0 is undefined, at $x = 0$ the derivative is undefined, so $f(x) = x^{1/3}$ is not a differentiable function even though it is continuous.

Root functions are not differentiable. We could pick $f(x) = x^{1/2}$ or $f(x) = x^{2/5}$, the result would still be the same – as long as there is a fraction in the power of x, we will always end up dividing by 0.

You can actually spot non-differentiable functions just by looking at the curve. If we go back to the Figure B.4, we can see that at $x = 0$ the curve has a vertical lift. Because the gradient is the tangent that touches the curve, at that lift the gradient will be vertical, as shown in Figure B.5:

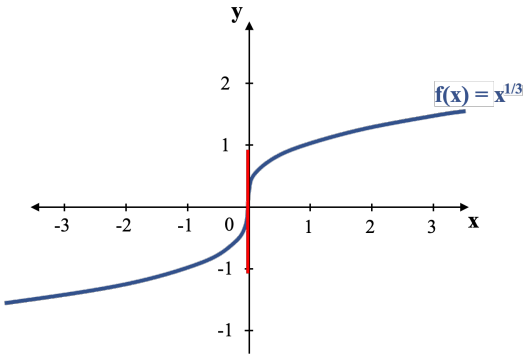


Figure B.5: Non-differentiable at x=0

Since vertical gradients are undefined, any curve with a vertical lift/drop would be non-differentiable. So you can check the function on the presence of these vertical lifts to understand if it is differentiable.

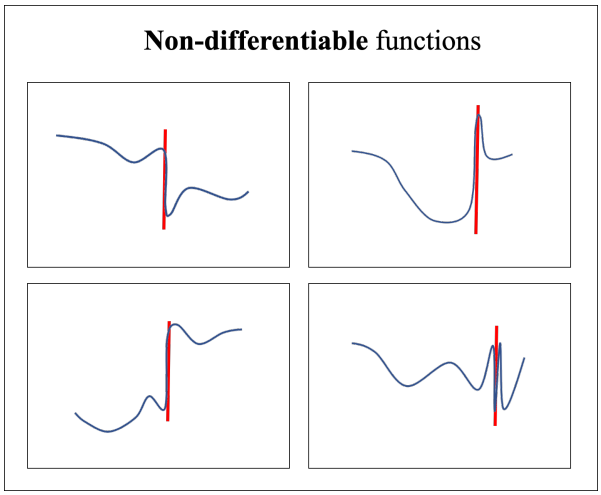


Figure B.6: Continuous Non-differentiable Functions

This was a very brief introduction to non-differentiable functions. Because this topic is slightly more advanced, this book does not cover more than basic fundamentals. I suggest you discover non-differentiable functions in your free time. For now it's enough to know that those functions exist, and you should keep that in mind when working with gradient.