Machine Learning Engineering

Andriy Burkov

Copyeditor: Aida E. Roig-Compton

Illustrators: Koen Van den Eeckhout, Andriy Burkov

Cover designer: Cristina Eleutério Alves Augusto

Publisher: True Positive Inc.

ISBN 978-1-7770054-5-0

To my parents:
Tatiana and Valeriy

and to my family:
daughters Catherine and Eva,
and brother Dmitriy

"In theory, there is no difference between theory and practice. But in practice, there is."

— Benjamin Brewster

"The perfect project plan is possible if one first documents a list of all the unknowns."

— Bill Langley

"When you're fundraising, it's AI. When you're hiring, it's ML. When you're implementing, it's linear regression. When you're debugging, it's printf()."

— Baron Schwartz

The book is distributed on the "read-first, buy-later" principle.

Contents

Foreword

Foreword by **Cassie Kozyrkov**, Chief Decision Scientist at Google, author of the course ***Making Friends with Machine Learning*** on Google Cloud Platform.

I'd like to let you in on a secret: when people say "machine learning" it sounds like there's only one discipline here. Surprise! There are actually two machine learnings, and they are as different as innovating in food recipes and inventing new kitchen appliances. Both are noble callings, as long as you don't get them confused; imagine hiring a pastry chef to build you an oven or an electrical engineer to bake bread for you!

The bad news is that almost everyone does mix these two machine learnings up. No wonder so many businesses fail at machine learning as a result. What no one seems to tell beginners is that most machine learning courses and textbooks are about Machine Learning Research — how to build ovens (and microwaves, blenders, toasters, kettles... the kitchen sink!) from scratch, not how to cook things and innovate with recipes at enormous scale. In other words, if you're looking for opportunities to create innovative ML-based solutions to business problems, you want the discipline called Applied Machine Learning, not Machine Learning Research, so most books won't suit your needs.

And now for the good news! You're looking at one of the few true Applied Machine Learning books out there. That's right, you found one! A real applied needle in the haystack of research-oriented stuff. Excellent job, dear reader... unless what you were actually looking for is a book to help you learn the skills to design general purpose algorithms, in which case I hope the author won't be too upset with me for telling you to flee now and go pick up pretty much any other machine learning book. This one is different.

When I created Making Friends with Machine Learning in 2016, Google's Applied Machine Learning course loved by more than ten thousand of our engineers and leaders, I gave it a very similar structure to the one in this book. That's because doing things in the right order is crucial in the applied space. As you use your newfound data powers, tackling certain steps before you've completed others can lead to anything from wasted effort to a project-demolishing kablooie. In fact, the similarity in table of contents between this book and my course is what originally convinced me to give this book a read. In a clear case of convergent evolution, I saw in the author a fellow thinker kept up at night by the lack of available

resources on Applied Machine Learning, one of the most potentially-useful yet horribly-misunderstood areas of engineering, enough to want to do something about it. So, if you're about to close this book, how about you do me a quick favor and at least ponder why the Table of Contents is arranged the way it is. You'll learn something good just from that, I promise.

So, what's in the rest of the book? The machine learning equivalent of a bumper guide to innovating in recipes to make food at scale. Since you haven't read the book yet, I'll put it in culinary terms: you'll need to figure out what's worth cooking / what the objectives are (*decision-making and product management*), understand the suppliers and the customers (*domain expertise and business acumen*), how to process ingredients at scale (*data engineering and analysis*), how to try many different ingredient-appliance combinations quickly to generate potential recipes (*prototype phase ML engineering*), how to check that the quality of the recipe is good enough to serve (*statistics*), how to turn a potential recipe into millions of dishes served efficiently (*production phase ML engineering*), and how to ensure that your dishes stay top notch even if the delivery truck brings you a ton of potatoes instead of the rice you ordered (*reliability engineering*). This book is one of the few to offer perspectives on each step of the end-to-end process.

Now would be a good moment for me to be blunt with you, dear reader. This book *is* pretty good. It is. Really. But it's not perfect. It cuts corners on occasion — just like a professional machine learning engineer is wont to do — though on the whole it gets its message right. And, since it covers an area with rapidly-evolving best practices, it doesn't pretend to offer the last word on the subject. But even if it were terribly sloppy, it would still be worth reading. Given how few comprehensive guides to Applied Machine Learning are out there, a coherent introduction to these topics is worth its weight in gold. I'm so glad this one is here!

One of my favorite things about this book is how fully it embraces the most important thing you need to know about machine learning: mistakes are possible... and sometimes they hurt. As my colleagues in site reliability engineering love to say, "Hope is not a strategy." Hoping that there will be no mistakes is the worst approach you can take. This book does so much better. It promptly shatters any false sense of security you were tempted to have about building an AI system that is more "intelligent" than you are. (Um, no. Just no.) Then it diligently takes you through a survey of all kinds of things that can go wrong in practice and how to prevent/detect/handle them. This book does a great job of outlining the importance of monitoring, how to approach model maintenance, what to do when things go wrong, how to think about fallback strategies for the kinds of mistakes you can't anticipate, how to deal with adversaries who try to exploit your system, and how to manage the expectations of your human users (there's also a section on what to do when your, er, users are machines). These are hugely important topics in practical machine learning, but they're so often neglected in other books. Not here.

If you intend to use machine learning to solve business problems at scale, I'm delighted you got your hands on this book. Enjoy!

Cassie Kozyrkov

September 2020

Preface

During the past several years, machine learning (ML), for many, has become a synonym for artificial intelligence. Even though machine learning, as a field of science, has existed for several decades, only a handful of organizations in the world have fully harnessed its potential. Despite the availability of modern open-source machine learning libraries, packages and frameworks supported by the leading organizations and broad communities of scientists and software engineers, most organizations are still struggling to apply machine learning for solving practical business problems.

One difficulty lies in the scarcity of talent. However, even when they have access to talented machine learning engineers and data analysts, in 2020, most organizations[1] still spend between 31 and 90 days deploying one model, while 18 percent of companies are taking longer than 90 days — some spending more than a year productionizing. The main challenges organizations face when developing ML capabilities, such as model version control, reproducibility, and scaling, are rather engineering than scientific.

There are plenty of good books on machine learning, both theoretical and hands-on. From a typical machine learning book, you can learn the types of machine learning, major families of algorithms, how they work, and how to build models from data using those algorithms.

A typical machine learning book is less concerned with the engineering aspects of implementing machine learning projects. Such questions as data collection, storage, preprocessing, feature engineering, as well as testing and debugging of models, their deployment to and retirement from production, runtime and post-production maintenance, are often left outside the scope of machine learning books.

This book intends to fill that gap.

Who This Book is For

I assume that the reader of this book understands machine learning basics and is capable of building a model, given a properly formatted dataset using a favorite programming

[1] "2020 state of enterprise machine learning", Algorithmia, 2019.

language or a machine learning library. If you don't feel comfortable applying machine learning algorithms to data and don't clearly see the difference between logistic regression, support vector machine, and random forest, I recommend starting your journey with The Hundred-Page Machine Learning Book, and then move to this book.

The target audience of this book is data analysts who lean towards a machine learning engineering role, machine learning engineers who want to bring more structure to their work, machine learning engineering students, as well as software architects who happen to deal with models provided by data analysts and machine learning engineers.

How to Use This Book

This book is a comprehensive review of machine learning engineering best practices and design patterns. I recommend reading it from beginning to end. However, you can read chapters in any order as they cover distinct aspects of the machine learning project lifecycle and do not have direct dependencies.

Should You Buy This Book?

Like its companion and precursor The Hundred-Page Machine Learning Book, this book is distributed on the "read-first, buy-later" principle. I firmly believe that readers must be able to read a book before paying for it; otherwise, they buy a pig in a poke.

The "read-first, buy-later" principle implies that you can freely download the book, read it, and share it with your friends and colleagues. If you read and liked the book, or found it helpful or useful in your work, business, or studies, then buy it.

Now you are all set. Enjoy your reading!

Andriy Burkov

Chapter 1

Introduction

Though the reader of this book should have a basic understanding of machine learning, it is still important to start with definitions, so that we are sure that we have a common understanding of the terms used throughout the book.

Below, I repeat some of the definitions from Chapter 2 of The Hundred-Page Machine Learning Book and also give several new ones. If you read my first book, some parts of this chapter might sound familiar.

After reading this chapter, we will understand the same way such concepts as supervised and unsupervised learning. We will agree on the data terminology, such as data used directly and indirectly, raw and tidy data, training and holdout data.

We will know when to use machine learning, when not to use it, and various forms of machine learning such as model- and instance-based, deep and shallow, classification and regression, and others.

Finally, we will define the scope of machine learning engineering and introduce the machine learning project lifecycle.

1.1 Notation and Definitions

Let's start by stating the basic mathematical notation and define the terms and notions, to which we will often have recourse in this book.

1.1.1 Data Structures

A **scalar**[1] is a simple numerical value, like 15 or -3.25. Variables or constants that take scalar values are denoted by an italic letter, like x or a.

A **vector** is an ordered list of scalar values, called attributes. We denote a vector as a bold character, for example, $\mathbf{x}$ or $\mathbf{w}$. Vectors can be visualized as arrows that point to some directions as well as points in a multi-dimensional space. Illustrations of three two-dimensional vectors, $\mathbf{a} = [2, 3]$, $\mathbf{b} = [-2, 5]$, and $\mathbf{c} = [1, 0]$ are given in Figure 1.1.1. We denote an attribute of a vector as an italic value with an index, like this: $w^{(j)}$ or $x^{(j)}$. The index j denotes a specific **dimension** of the vector, the position of an attribute in the list. For instance, in the vector $\mathbf{a}$ shown in red in Figure 1.1.1, $a^{(1)} = 2$ and $a^{(2)} = 3$.

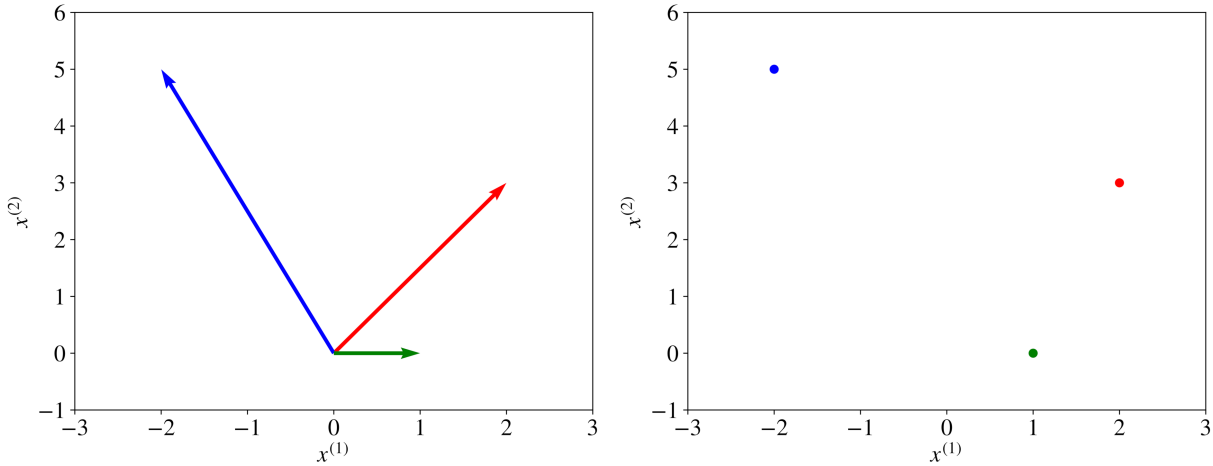


Figure 1.1: Three vectors visualized as directions and as points.

The notation $x^{(j)}$ should not be confused with the power operator, such as the 2 in x^2 (squared) or 3 in x^3 (cubed). If we want to apply a power operator, say squared, to an indexed attribute of a vector, we write like this: $(x^{(j)})^2$.

A variable can have two or more indices, like this: $x_i^{(j)}$ or like this $x_{i,j}^{(k)}$. For example, in neural networks, we denote as $x_{l,u}^{(j)}$ the input feature j of unit u in layer l.

A **matrix** is a rectangular array of numbers arranged in rows and columns. Below is an example of a matrix with two rows and three columns,

$$\mathbf{A} = \begin{bmatrix} 2 & -2 & 1 \\ 3 & 5 & 0 \end{bmatrix}.$$

[1] If a term is **in bold**, that means that the term can be found in the index at the end of the book.

Matrices are denoted with bold capital letters, such as $\mathbf{A}$ or $\mathbf{W}$. You can notice from the above example of matrix $\mathbf{A}$ that matrices can be seen as regular structures composed of vectors. Indeed, the columns of matrix $\mathbf{A}$ above are vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ illustrated in Figure .

A **set** is an unordered collection of unique elements. We denote a set as a calligraphic capital character, for example, $\mathcal{S}$. A set of numbers can be finite (include a fixed amount of values). In this case, it is denoted using accolades, for example, $\{1, 3, 18, 23, 235\}$ or $\{x_1, x_2, x_3, x_4, \ldots, x_n\}$. Alternatively, a set can be infinite and include all values in some interval. If a set includes all values between a and b, including a and b, it is denoted using brackets as $[a, b]$. If the set doesn't include the values a and b, such a set is denoted using parentheses like this: (a, b). For example, the set $[0, 1]$ includes such values as 0, 0.0001, 0.25, 0.784, 0.9995, and 1.0. A special set denoted $\mathbb{R}$ includes all numbers from minus infinity to plus infinity.

When an element x belongs to a set $\mathcal{S}$, we write $x \in \mathcal{S}$. We can obtain a new set $\mathcal{S}_3$ as an **intersection** of two sets $\mathcal{S}_1$ and $\mathcal{S}_2$. In this case, we write $\mathcal{S}_3 \leftarrow \mathcal{S}_1 \cap \mathcal{S}_2$. For example $\{1, 3, 5, 8\} \cap \{1, 8, 4\}$ gives the new set $\{1, 8\}$.

We can obtain a new set $\mathcal{S}_3$ as a **union** of two sets $\mathcal{S}_1$ and $\mathcal{S}_2$. In this case, we write $\mathcal{S}_3 \leftarrow \mathcal{S}_1 \cup \mathcal{S}_2$. For example $\{1, 3, 5, 8\} \cup \{1, 8, 4\}$ gives the new set $\{1, 3, 5, 8, 4\}$.

The notation $|\mathcal{S}|$ means the size of set $\mathcal{S}$, that is, the number of elements it contains.

1.1.2 Capital Sigma Notation

The summation over a collection $\mathcal{X} = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$ or over the attributes of a vector $\mathbf{x} = [x^{(1)}, x^{(2)}, \ldots, x^{(m-1)}, x^{(m)}]$ is denoted like this:

$$\sum_{i=1}^{n} x_i \stackrel{\text{def}}{=} x_1 + x_2 + \ldots + x_{n-1} + x_n, \text{ or else: } \sum_{j=1}^{m} x^{(j)} \stackrel{\text{def}}{=} x^{(1)} + x^{(2)} + \ldots + x^{(m-1)} + x^{(m)}.$$

The notation $\stackrel{\text{def}}{=}$ means "is defined as".

The **Euclidean norm** of a vector $\mathbf{x}$, denoted by $\|\mathbf{x}\|$, characterizes the "size" or the "length" of the vector. It's given by $\sqrt{\sum_{j=1}^{D} \left(x^{(j)}\right)^2}$.

The distance between two vectors $\mathbf{a}$ and $\mathbf{b}$ is given by the **Euclidean distance**:

$$\|a - b\| \stackrel{\text{def}}{=} \sqrt{\sum_{i=1}^{N} \left(a^{(i)} - b^{(i)}\right)^2}.$$

1.2 What is Machine Learning

Machine learning is a subfield of computer science that is concerned with building algorithms that, to be useful, rely on a collection of examples of some phenomenon. These examples can come from nature, be handcrafted by humans, or generated by another algorithm.

Machine learning can also be defined as the process of solving a practical problem by,

1) collecting a dataset, and
2) algorithmically training a **statistical model** based on that dataset.

That statistical model is assumed to be used somehow to solve the practical problem. To save keystrokes, I use the terms "learning" and "machine learning" interchangeably. For the same reason, I often say "model" referring to a statistical model.

Learning can be supervised, semi-supervised, unsupervised, and reinforcement.

1.2.1 Supervised Learning

In **supervised learning**, the data analyst works with a collection of **labeled examples** $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$. Each element $\mathbf{x}_i$ among N is called a **feature vector**. In computer science, a vector is a one-dimensional array. A one-dimensional array, in turn, is an ordered and indexed sequence of values. The length of that sequence of values, D, is called the vector's **dimensionality**.

A feature vector is a vector in which each dimension j from 1 to D contains a value that describes the example. Each such value is called a **feature** and is denoted as $x^{(j)}$. For instance, if each example $\mathbf{x}$ in our collection represents a person, then the first feature, $x^{(1)}$, could contain height in cm, the second feature, $x^{(2)}$, could contain weight in kg, $x^{(3)}$ could contain gender, and so on. For all examples in the dataset, the feature at position j in the feature vector always contains the same kind of information. It means that if $x_i^{(2)}$ contains weight in kg in some example $\mathbf{x}_i$, then $x_k^{(2)}$ will also contain weight in kg in every example $\mathbf{x}_k$, for all k from 1 to N. The **label** y_i can be either an element belonging to a finite set of **classes** $\{1, 2, \ldots, C\}$, or a real number, or a more complex structure, like a vector, a matrix, a tree, or a graph. Unless otherwise stated, in this book y_i is either one of a finite set of classes or a real number.[2] You can think of a class as a category to which an example belongs.

For instance, if your examples are email messages and your problem is spam detection, then you have two classes: spam and not_spam. In supervised learning, the problem of predicting a class is called **classification**, while the problem of predicting a real number is called **regression**. The value that has to be predicted by a supervised model is called a **target**. An example of regression is a problem of predicting the salary of an employee given their work experience

[2] A real number is a quantity that can represent a distance along a line. Examples: 0, -256.34, 1000, 1000.2.

and knowledge. An example of classification is when a doctor enters the characteristics of a patient into a software application, and the application returns the diagnosis.

The difference between classification and regression is shown in Figure 1.2. In classification, the learning algorithm looks for a line (or, more generally, a hypersurface) that separates examples of different classes from one another. In regression, on the other hand, the learning algorithm looks to find a line or a hypersurface that closely follows the training examples.

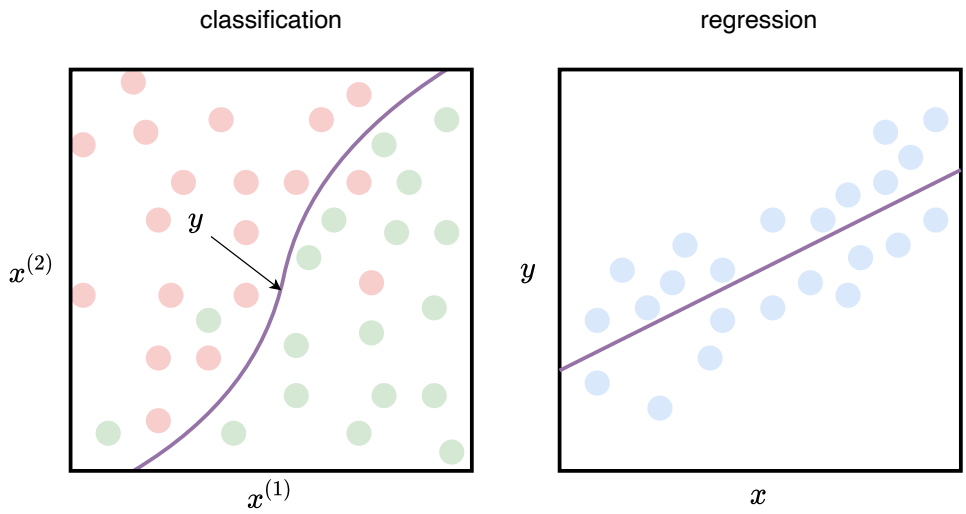


Figure 1.2: Difference between classification and regression.

The goal of a **supervised learning algorithm** is to use a dataset to produce a model that takes a feature vector $\mathbf{x}$ as input and outputs information that allows deducing a label for this feature vector. For instance, a model created using a dataset of patients could take as input a feature vector describing a patient and output a probability that the patient has cancer.

Even if the model is typically a mathematical function, when thinking about what the model does with the input, it is convenient to think that the model "looks" at the values of some features in the input and, based on experience with similar examples, outputs a value. That output value is a number or a class "the most similar" to the labels seen in the past in the examples with similar values of features. It looks simplistic, but the decision tree model and the k-nearest neighbors algorithm work almost like that.

1.2.2 Unsupervised Learning

In **unsupervised learning**, the dataset is a collection of **unlabeled examples** $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$. Again, $\mathbf{x}$ is a feature vector, and the goal of an **unsupervised learning algorithm** is to create

a model that takes a feature vector $\mathbf{x}$ as input and either transforms it into another vector or into a value that can be used to solve a practical problem. For example, in **clustering**, the model returns the ID of the cluster for each feature vector in the dataset. Clustering is useful for finding groups of similar objects in a large collection of objects, such as images or text documents. By using clustering, for example, the analyst can sample a sufficiently representative yet small subset of unlabeled examples from a large collection of examples for manual labeling: a few examples are sampled from each cluster instead of sampling directly from the large collection and risking only sampling examples very similar to one another.

In **dimensionality reduction**, the model's output is a feature vector with fewer dimensions than the input. For example, the scientist has a feature vector that is too complex to visualize (it has more than three dimensions). The dimensionality reduction model can transform that feature vector into a new feature vector (by preserving the information up to some extent) with only two or three dimensions. This new feature vector can be plotted on a graph.

In **outlier detection**, the output is a real number that indicates how the input feature vector is different from a "typical" example in the dataset. Outlier detection is useful for solving a network intrusion problem (by detecting abnormal network packets that are different from a typical packet in "normal" traffic) or detecting novelty (such as a document different from the existing documents in a collection).

1.2.3 Semi-Supervised Learning

In **semi-supervised learning**, the dataset contains both labeled and unlabeled examples. Usually, the quantity of unlabeled examples is much higher than the number of labeled examples. The goal of a **semi-supervised learning algorithm** is the same as the goal of the supervised learning algorithm. The hope here is that, by using many unlabeled examples, a learning algorithm can find (we might say "produce" or "compute") a better model.

1.2.4 Reinforcement Learning

Reinforcement learning is a subfield of machine learning where the machine (called an agent) "lives" in an environment and is capable of perceiving the state of that environment as a vector of features. The machine can execute actions in non-terminal states. Different actions bring different rewards and could also move the machine to another state of the environment. A common goal of a reinforcement learning algorithm is to learn an optimal **policy**.

An optimal policy is a function (similar to the model in supervised learning) that takes the feature vector of a state as input and outputs an optimal action to execute in that state. The action is optimal if it maximizes the expected average long-term reward.

Reinforcement learning solves a particular problem where decision making is sequential, and the goal is long-term, such as game playing, robotics, resource management, or logistics.

In this book, for simplicity, most explanations are limited to supervised learning. However, all the material presented in the book is applicable to other types of machine learning.

1.3 Data and Machine Learning Terminology

Now let's introduce the common data terminology (such as data used directly and indirectly, raw and tidy data, training and holdout data) and the terminology related to machine learning (such as baseline, hyperparameter, pipeline, and others).

1.3.1 Data Used Directly and Indirectly

The data you will work with in your machine learning project can be used to form the examples $\mathbf{x}$ **directly** or **indirectly**.

Imagine that we build a named entity recognition system. The input of the model is a sequence of words; the output is the sequence of labels[3] of the same length as the input. To make the data readable by a machine learning algorithm, we have to transform each natural language word into a machine-readable array of attributes, which we call a feature vector.[4] Some features in the feature vector may contain the information that distinguishes that specific word from other words in the dictionary. Other features can contain additional attributes of the word in that specific sequence, such as its shape (lowercase, uppercase, capitalized, and so on). Or it can be binary attributes indicating whether this word is the first word of some human name or the last word of the name of some location or organization. To create these latter binary features, we may decide to use some dictionaries, lookup tables, gazetteers, or other machine learning models making predictions about words.

You could already have noticed that the collection of word sequences is the data used to form training examples directly, while the data contained in dictionaries, lookup tables, and gazetteers is used indirectly: we can use it to extend feature vectors with additional features, but we cannot use it to create new feature vectors.

1.3.2 Raw and Tidy Data

As we just discussed, directly used data is a collection of entities that constitute the basis of a dataset. Each entity in that collection can be transformed into a training example. **Raw data** is a collection of entities in their natural form; they cannot always be directly employable for

[3] Labels can be, for example, values from the set {"Location", "Organization", "Person", "Other"}.

[4] The terms "attribute" and "feature" are often used interchangeably. In this book, I use the term "attribute" to describe a specific property of an example, while the term "feature" refers to value $x^{(j)}$ at position j in the feature vector $\mathbf{x}$ used by a machine learning algorithm.

machine learning. For instance, a Word document or a JPEG file are pieces of raw data; they cannot be directly used by a machine learning algorithm.[5]

To be employable in machine learning, a necessary (but not sufficient) condition for the data is to be tidy. **Tidy data** can be seen as a spreadsheet, in which each row represents one example, and columns represent various **attributes** of an example, as shown in Figure 1.3. Sometimes raw data can be tidy, e.g., provided to you in the form of a spreadsheet. However, in practice, to obtain tidy data from raw data, data analysts often resort to the procedure called **feature engineering**, which is applied to the direct and, optionally, indirect data with the goal to transform each raw example into a feature vector $\mathbf{x}$. Chapter 4 is devoted entirely to feature engineering.

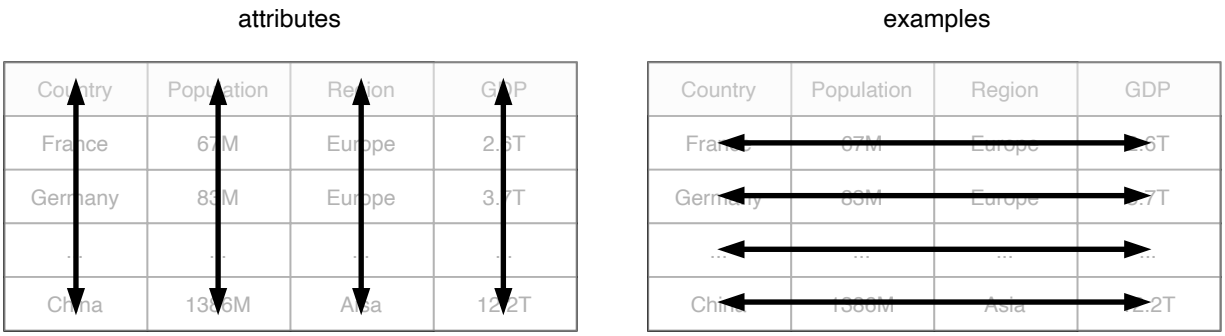


Figure 1.3: Tidy data: examples are rows and attributes are columns.

It's important to note here that for some tasks, an example used by a learning algorithm can have a form of a sequence of vectors, a matrix, or a sequence of matrices. The notion of data tidiness for such algorithms is defined similarly: you only replace "row of fixed width in a spreadsheet" by a matrix of fixed width and height, or a generalization of matrices to a higher dimension called a **tensor**.

The term "tidy data" was coined by Hadley Wickham in his paper with the same title.[6]

As I mentioned at the beginning of this subsection, data can be tidy, but still not usable by a particular machine learning algorithm. Most machine learning algorithms, in fact, only accept training data in the form of a collection of numerical feature vectors. Consider the data shown in Figure 1.3. The attribute "Region" is categorical and not numerical. The decision tree learning algorithm can work with categorical values of attributes, but most learning

[5]The term "unstructured data" is often used to designate a data element that contains information whose type was not formally defined. Examples of unstructured data are photos, images, videos, text messages, social media posts, PDFs, text documents, and emails. The term "semi-structured data" refers to data elements whose structure helps deriving types of some information encoded in those data elements. Examples of semi-structured data include log files, comma- and tab-delimited text files, as well as documents in JSON and XML formats.

[6]Wickham, Hadley. "Tidy data." Journal of Statistical Software 59.10 (2014): 1-23.

algorithms cannot. In Section **??** of Chapter 4, we will see how to transform a categorical attribute into a numerical feature.

Note that in the academic machine learning literature, the word "example" typically refers to a tidy data example with an optionally assigned label. However, during the stage of data collection and labeling, which we consider in the next chapter, examples can still be in the raw form: images, texts, or rows with categorical attributes in a spreadsheet. In this book, when it's important to highlight the difference, I will say **raw example** to indicate that a piece of data was not transformed into a feature vector yet. Otherwise, assume that examples have the form of feature vectors.

1.3.3 Training and Holdout Sets

In practice, data analysts work with three distinct sets of examples:

1) training set,
2) validation set,[7] and
3) test set.

Once you have got the data in the form of a collection of examples, the first thing you do in your machine learning project is shuffle the examples and split the dataset into three distinct sets: **training**, **validation**, and **test**. The training set is usually the biggest one; the learning algorithm uses the training set to produce the model. The validation and test sets are roughly the same size, much smaller than the size of the training set. The learning algorithm is not allowed to use examples from the validation or test sets to train the model. That is why those two sets are also called **holdout sets**.

The reason to have three sets, and not one, is simple: when we train a model, we don't want the model to only do well at predicting labels of examples the learning algorithm has already seen. A trivial algorithm that simply memorizes all training examples and then uses the memory to "predict" their labels will make no mistakes when asked to predict the labels of the training examples. However, such an algorithm would be useless in practice. What we really want is a model that is good at predicting examples that the learning algorithm didn't see. In other words, we want good performance on a holdout set.[8]

We need two holdout sets and not one because we use the validation set to 1) choose the learning algorithm, and 2) find the best configuration values for that learning algorithm (known as **hyperparameters**). We use the test set to assess the model before delivering it

[7] In some literature, the validation set can also be called "development set." Sometimes, when the labeled examples are scarce, analysts can decide to work without a validation set, as we will see in Chapter 5 in the section on **cross-validation**.

[8] To be precise, we want the model to do well on most random samples from the statistical distribution to which our data belongs. We assume that if the model demonstrates good performance on a holdout set, randomly drawn from the unknown distribution of our data, there are high chances that our model will do well on other random samples of our data.

to the client or putting it in production. That is why it's important to make sure that no information from the validation or test sets is exposed to the learning algorithm. Otherwise, the validation and test results will most likely be too optimistic. This can indeed happen due to **data leakage**, an important phenomenon we consider in Section 3.2.8 of Chapter 3 and subsequent chapters.

1.3.4 Baseline

In machine learning, a **baseline** is a simple algorithm for solving a problem, usually based on a heuristic, simple summary statistics, randomization, or very basic machine learning algorithm. For example, if your problem is classification, you can pick a baseline classifier and measure its performance. This baseline performance will then become what you compare any future model to (usually, built using a more sophisticated approach).

1.3.5 Machine Learning Pipeline

A machine learning **pipeline** is a sequence of operations on the dataset that goes from its initial state to the model.

A pipeline can include, among others, such stages as data partitioning, missing data imputation, feature extraction, data augmentation, class imbalance reduction, dimensionality reduction, and model training.

In practice, when we deploy a model in production, we usually deploy an entire pipeline. Furthermore, an entire pipeline is usually optimized when hyperparameters are tuned.

1.3.6 Parameters vs. Hyperparameters

Hyperparameters are inputs of machine learning algorithms or pipelines that influence the performance of the model. They don't belong to the training data and cannot be learned from it. For example, the maximum depth of the tree in the decision tree learning algorithm, the misclassification penalty in support vector machines, k in the k-nearest neighbors algorithm, the target dimensionality in dimensionality reduction, and the choice of the missing data imputation technique are all examples of hyperparameters.

Parameters, on the other hand, are variables that define the model trained by the learning algorithm. Parameters are directly modified by the learning algorithm based on the training data. The goal of learning is to find such values of parameters that make the model optimal in a certain sense. Examples of parameters are w and b in the equation of linear regression $y = wx + b$. In this equation, x is the input of the model, and y is its output (the prediction).

1.3.7 Classification vs. Regression

Classification is a problem of automatically assigning a **label** to an **unlabeled example**. Spam detection is a famous example of classification.

In machine learning, the classification problem is solved by a **classification learning algorithm** that takes a collection of **labeled examples** as inputs and produces a **model** that can take an unlabeled example as input and either directly output a label or output a number that can be used by the analyst to deduce the label. An example of such a number is a probability of an input data element to have a specific label.

In a classification problem, a label is a member of a finite set of **classes**. If the size of the set of classes is two ("sick"/"healthy", "spam"/"not_spam"), we talk about **binary classification** (also called **binomial** in some sources). **Multiclass classification** (also called **multinomial**) is a classification problem with three or more classes.[9]

While some learning algorithms naturally allow for more than two classes, others are by nature binary classification algorithms. There are strategies to turn a binary classification learning algorithm into a multiclass one. I talk about one of them, **one-versus-rest**, in Section **??** of Chapter 6.

Regression is a problem of predicting a real-valued quantity given an unlabeled example. Estimating house price valuation based on house features, such as area, number of bedrooms, location, and so on, is a famous example of regression.

The regression problem is solved by a **regression learning algorithm** that takes a collection of labeled examples as inputs and produces a model that can take an unlabeled example as input and output a target.

1.3.8 Model-Based vs. Instance-Based Learning

Most supervised learning algorithms are **model-based**. A typical **model** is a **support vector machine** (**SVM**). Model-based learning algorithms use the training data to create a model with **parameters** learned from the training data. In SVM, the two parameters are $\mathbf{w}$ (a vector) and b (a real number). After the model is trained, it can be saved on disk while the training data can be discarded.

Instance-based learning algorithms use the whole dataset as the model. One instance-based algorithm frequently used in practice is **k-Nearest Neighbors** (kNN). In classification, to predict a label for an input example, the kNN algorithm looks at the close neighborhood of the input example in the space of feature vectors and outputs the label that it saw most often in this close neighborhood.

[9] There's still one label per example, though.

1.3.9 Shallow vs. Deep Learning

A **shallow learning** algorithm learns the parameters of the model directly from the features of the training examples. Most machine learning algorithms are shallow. The notorious exceptions are **neural network** learning algorithms, specifically those that build neural networks with more than one **layer** between input and output. Such neural networks are called **deep neural networks**. In deep neural network learning (or, simply, **deep learning**), contrary to shallow learning, most model parameters are learned not directly from the features of the training examples, but from the outputs of the preceding layers.

1.3.10 Training vs. Scoring

When we apply a machine learning algorithm to a dataset in order to obtain a model, we talk about **model training** or simply training.

When we apply a trained model to an input example (or, sometimes, a sequence of examples) in order to obtain a prediction (or, predictions) or to somehow transform an input, we talk about **scoring**.

1.4 When to Use Machine Learning

Machine learning is a powerful tool for solving practical problems. However, like any tool, it should be used in the right context. Trying to solve all problems using machine learning would be a mistake.

You should consider using machine learning in one of the following situations.

1.4.1 When the Problem Is Too Complex for Coding

In a situation where the problem is so complex or big that you cannot hope to write all the rules to solve it and where a partial solution is viable and interesting, you can try to solve the problem with machine learning.

One example is spam detection: it's impossible to write the code that will implement such a logic that will effectively detect spam messages and let genuine messages reach the inbox. There are just too many factors to consider. For instance, if you program your spam filter to reject all messages from people who are not in your contacts, you risk losing messages from someone who has got your business card at a conference. If you make an exception for messages containing specific keywords related to your work, you will probably miss a message from your child's teacher, and so on.

If you still decide to directly program a solution to that complex problem, with time, you will have in your programming code so many conditions and exceptions from those conditions

that maintaining that code will eventually become infeasible. In this situation, training a classifier on examples “spam”/“not_spam” seems logical and the only viable choice.

Another difficulty for writing code to solve a problem lies in the fact that humans have a hard time with prediction problems based on input that has too many parameters; it’s especially true when those parameters are **correlated** in unknown ways. For example, take the problem of predicting whether a borrower will repay a loan. Hundreds of numbers represent each borrower: age, salary, account balance, frequency of past payments, married or not, number of children, make and year of the car, mortgage balance, and so on. Some of those numbers may be important to make the decision, some may be less important alone, but become more important if considered in combination with some other numbers.

Writing code that will make such decisions is hard because, even for an expert, it’s not clear how to combine, in an optimal way, all the attributes describing a person into a prediction.

1.4.2 When the Problem Is Constantly Changing

Some problems may continuously change with time so that the programming code must be regularly updated. That results in the frustration of software engineers working on the problem, an increased chance of introducing errors, difficulties of combining “previous” and “new” logic, and significant overhead of testing and deploying updated solutions.

For example, you can have a task of scraping specific data elements from a collection of webpages. Let’s say that for each webpage in that collection, you write a set of fixed data extraction rules in the following form: “pick the third <p> element from <body> and then pick the data from the second inside that <p>.” If a website owner changes the design of a webpage, the data you scrape may end up in the second or the fourth <p> element, making your extraction rule wrong. If the collection of webpages you scrape is large (thousands of URLs), every day you will have rules that become wrong; you will end up endlessly fixing those rules. Needless to say that very few software engineers would love to do such work on a daily basis.

1.4.3 When It Is a Perceptive Problem

Today, it’s hard to imagine someone trying to solve **perceptive problems** such as speech, image, and video recognition without using machine learning. Consider an image. It’s represented by millions of pixels. Each pixel is given by three numbers: the intensity of red, green, and blue channels. In the past, engineers tried to solve the problem of image recognition (detecting what’s on the picture) by applying handcrafted “filters” to square patches of pixels. If one filter, for example, the one that was designed to “detect” grass, generates a high value when applied to many pixel patches, while another filter, designed to detect brown fur, also returns high values for many patches, then we can say that there are high chances that the image represents a cow in a field (I’m simplifying a bit).

Today, perceptive problems are effectively solved using machine learning models, such as neural networks. We consider the problem of training neural networks in Chapter 6.

1.4.4 When It Is an Unstudied Phenomenon

If we need to be able to make predictions of some phenomenon that is not well-studied scientifically, but examples of it are observable, then machine learning might be an appropriate (and, in some cases, the only available) option. For example, machine learning can be used to generate personalized mental health medication options based on the patient's genetic and sensory data. Doctors might not necessarily be able to interpret such data to make an optimal recommendation, while a machine can discover patterns in data by analyzing thousands of patients and predicting which molecule has the highest chance to help a given patient.

Another example of observable but unstudied phenomena are logs of a complex computing system or a network. Such logs are generated by multiple independent or interdependent processes. For a human, it's hard to make predictions about the future state of the system based on logs alone without having a model of each process and their interdependency. If the number of examples of historical log records is high enough (which is often the case), the machine can learn patterns hidden in logs and be able to make predictions without knowing anything about each process.

Finally, making predictions about people based on their observed behavior is hard. In this problem, we obviously cannot have a model of a person's brain, but we have readily available examples of expressions of the person's ideas (in the form of online posts, comments, and other activities). Based on those expressions alone, a machine learning model deployed in a social network can recommend the content or other people to connect with.

1.4.5 When the Problem Has a Simple Objective

Machine learning is especially suitable for solving problems that you can formulate as a problem with a simple objective: such as yes/no decisions or a single number. In contrast, you cannot use machine learning to build a model that works as a general video game, like Mario, or a word processing software, like Word. This is due to too many different decisions to make: what to display, where and when, what should happen as a reaction to the user's input, what to write to or read from the hard drive, and so on; getting examples that illustrate all (or even most) of those decisions is practically infeasible.

1.4.6 When It Is Cost-Effective

Three major sources of cost in machine learning are:

- collecting, preparing, and cleaning the data,

- training the model,
- building and running the infrastructure to serve and and monitor the model, as well as labor resources to maintain it.

The cost of training the model includes human labor and, in some cases, the expensive hardware needed to train deep models. Model maintenance includes continuously monitoring the model and collecting additional data to keep the model up to date.

1.5 When Not to Use Machine Learning

There are plenty of problems that cannot be solved using machine learning; it's hard to characterize all of them. Here we only consider several hints.

You probably should not use machine learning when:

- every action of the system or a decision made by it must be explainable,
- every change in the system's behavior compared to its past behavior in a similar situation must be explainable,
- the cost of an error made by the system is too high,
- you want to get to the market as fast as possible,
- getting the right data is too hard or impossible,
- you can solve the problem using traditional software development at a lower cost,
- a simple heuristic would work reasonably well,
- the phenomenon has too many outcomes while you cannot get a sufficient amount of examples to represent them (like in video games or word processing software),
- you build a system that will not have to be improved frequently over time,
- you can manually fill an exhaustive lookup table by providing the expected output for any input (that is, the number of possible input values is not too large, or getting outputs is fast and cheap).

1.6 What is Machine Learning Engineering

Machine learning engineering (MLE) is the use of scientific principles, tools, and techniques of machine learning and traditional software engineering to design and build complex computing systems. MLE encompasses all stages from data collection, to model training, to making the model available for use by the product or the customers.

Typically, a data analyst[10] is concerned with understanding the business problem, building a model to solve it, and evaluating the model in a restricted development environment. A

[10] Since circa 2013, data scientist has become a popular job title. Unfortunately, companies and experts don't have an agreement on the definition of the term. Instead, I use the term "data analyst" by referring to a person capable of applying numerical or statistical analysis to data ready for analysis.

machine learning engineer, in turn, is concerned with sourcing the data from various systems and locations and preprocessing it, programming features, training an effective model that will run in the production environment, coexist well with other production processes, be stable, maintainable, and easily accessible by different types of users with different use cases.

In other words, MLE includes any activity that lets machine learning algorithms be implemented as a part of an effective production system.

In practice, machine learning engineers might be employed in such activities as rewriting a data analyst's code from rather slow R and Python[11] into more efficient Java or C++, scaling this code and making it more robust, packaging the code into an easy-to-deploy versioned package, optimizing the machine learning algorithm to make sure that it generates a model compatible with, and running correctly in, the organization's production environment.

In many organizations, data analysts execute some of the MLE tasks, such as data collection, transformation, and feature engineering. On the other hand, machine learning engineers often execute some of the data analysis tasks, including learning algorithm selection, hyperparameter tuning, and model evaluation.

Working on a machine learning project is different from working on a typical software engineering project. Unlike traditional software, where a program's behavior usually is deterministic, machine learning applications incorporate models whose behavior may naturally degrade over time, or they can start behaving abnormally. Such abnormal behavior of the model might be explained by various reasons, including a fundamental change in the input data or an updated feature extractor that now returns a different distribution of values or values of a different type. They often say that machine learning systems "fail silently." A machine learning engineer must be capable of preventing such failures or, when it's impossible to prevent them entirely, know how to detect and handle them when they happen.

1.7 Machine Learning Project Life Cycle

A machine learning project starts with understanding the business objective. Usually, a business analyst works with the client[12] and the data analyst to transform a business problem into an engineering project. The engineering project may or may not have a machine learning part. In this book, we, of course, consider engineering projects that have some machine learning involved.

Once an engineering project is defined, this is where the scope of the machine learning engineering starts. In the scope of a broader engineering project, machine learning must first have a well-defined **goal**. The goal of machine learning is a specification of what a

[11] Many scientific modules in Python are indeed implemented in fast C/C++; however, data analyst's own Python code can still be slow.

[12] If the machine learning project supports a product developed and sold by the organization, then the business analyst works with the product owner.

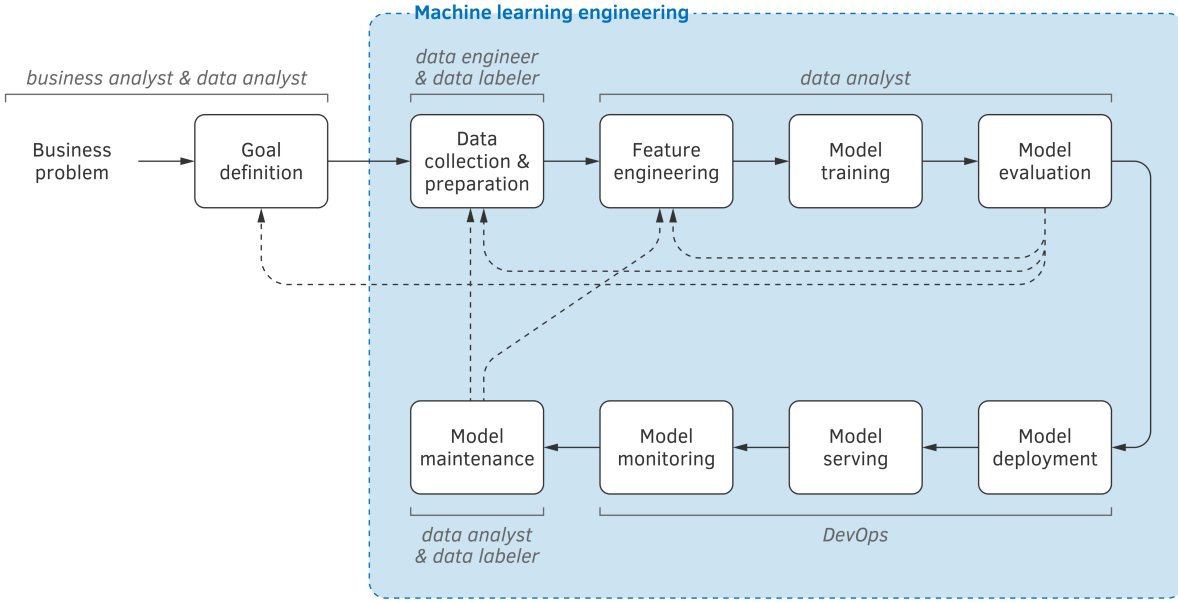


Figure 1.4: Machine learning project life cycle.

statistical model receives as input, what it generates as output, and the criteria of acceptable (or unacceptable) behavior of the model.

The goal of machine learning is not necessarily the same as the business objective. The business objective is what the organization wants to achieve. For example, the business objective of Google with Gmail can be to make Gmail the most-used email service in the world. Google might create multiple machine learning engineering projects to achieve that business objective. The goal of one of those machine learning projects can be to distinguish Primary emails from Promotions with accuracy above 90%.

Overall, a machine learning project life cycle, illustrated in Figure 1.4, consists of the following stages: 1) goal definition, 2) data collection and preparation, 3) feature engineering, 4) model training, 5) model evaluation, 6) model deployment, 7) model serving, 8) model monitoring, and 9) model maintenance.

In Figure 1.4, the scope of machine learning engineering (and the scope of this book) is limited by the blue zone. The solid arrows show a typical flow of the project stages. The dashed arrows indicate that at some stages, a decision can be made to go back in the process and either collect more data or collect different data, and revise features (by decommissioning some of them and engineering new ones).

Every stage mentioned above will be considered in one of the book's chapters. But first, let's discuss how to prioritize machine learning projects, define the project's goal, and structure a machine learning team. The next chapter is devoted to these three questions.

1.8 Summary

A model-based machine learning algorithm takes a collection of training examples as input and outputs a model. An instance-based machine learning algorithm uses the entire training dataset as a model. The training data is exposed to the machine learning algorithm, while holdout data isn't.

A supervised learning algorithm builds a model that takes a feature vector and outputs a prediction about that feature vector. An unsupervised learning algorithm builds a model that takes a feature vector as input and transforms it into something useful.

Classification is the problem of predicting, for an input example, one of a finite set of classes. Regression, in turn, is a problem of predicting a numerical target.

Data can be used directly or indirectly. Directly-used data is a basis for forming a dataset of examples. Indirectly-used data is used to enrich those examples.

The data for machine learning must be tidy. A tidy dataset can be seen as a spreadsheet where each row is an example, and each column is one of the properties of an example. In addition to being tidy, most machine learning algorithms require numerical data, as opposed to categorical. Feature engineering is the process of transforming data into a form that machine learning algorithms can use.

A baseline is essential to make sure that the model works better than a simple heuristic.

In practice, machine learning is implemented as a pipeline that contains chained stages of data transformation, from data partitioning to missing-data imputation, to class imbalance and dimensionality reduction, to model training. The hyperparameters of the entire pipeline are usually optimized; the entire pipeline can be deployed and used for predictions.

Parameters of the model are optimized by the learning algorithm based on the training data. The values of hyperparameters cannot be learned by the learning algorithm and are, in turn, tuned by using the validation dataset. The test set is only used to assess the model's performance and report it to the client or product owner.

A shallow learning algorithm trains a model that makes predictions directly from the input features. A deep learning algorithm trains a layered model, in which each layer generates outputs by taking the outputs of the preceding layer as inputs.

You should consider using machine learning to solve a business problem when the problem is too complex for coding, the problem is constantly changing, it is a perceptive problem, it is an unstudied phenomenon, the problem has a simple objective, and it is cost-effective.

There are many situations when machine learning should, probably, not be used: when explainability is needed, when errors are intolerable, when traditional software engineering is a less expensive option, when all inputs and outputs can be enumerated and saved in a database, and when data is hard to get or too expensive.

Machine learning engineering (MLE) is the use of scientific principles, tools, and techniques of machine learning and traditional software engineering to design and build complex computing systems. MLE encompasses all stages from data collection, to model training, to making the model available for use by the product or the consumers.

A machine learning project life cycle consists of the following stages: 1) goal definition, 2) data collection and preparation, 3) feature engineering, 4) model training, 5) model evaluation, 6) model deployment, 7) model serving, 8) model monitoring, and 9) model maintenance.

Every stage will be considered in one of the book's chapters.

Chapter 2

Before the Project Starts

Before a machine learning project starts, it must be prioritized. Prioritization is inevitable: the team and equipment capacity is limited, while the organization's backlog of projects could be very long.

To prioritize a project, one has to estimate its complexity. With machine learning, accurate complexity estimation is rarely possible because of major unknowns, such as whether the required model quality is attainable in practice, how much data is needed, and what, and how many features are necessary.

Furthermore, a machine learning project must have a well-defined goal. Based on the goal of the project, the team could be adequately adjusted and resources provisioned.

In this chapter, we consider these and related activities that must be taken care of before a machine learning project starts.

2.1 Prioritization of Machine Learning Projects

The key considerations in the prioritization of a machine learning project, are impact and cost.

2.1.1 Impact of Machine Learning

The impact of using machine learning in a broader engineering project is high when, 1) machine learning can replace a complex part in your engineering project or 2) there's a great benefit in getting inexpensive (but probably imperfect) predictions.

For example, a complex part of an existing system can be rule-based, with many nested rules and exceptions. Building and maintaining such a system can be extremely difficult,

time-consuming, and error-prone. It can also be a source of significant frustration for software engineers when they are asked to maintain that part of the system. Can the rules be learned instead of programming them? Can an existing system be used to generate labeled data easily? If yes, such a machine learning project would have a high impact and low cost.

Inexpensive and imperfect predictions can be valuable, for example, in a system that dispatches a large number of requests. Let's say many such requests are "easy" and can be solved quickly using some existing automation. The remaining requests are considered "difficult" and must be addressed manually.

A machine learning-based system that recognizes "easy" tasks and dispatches them to the automation will save a lot of time for humans who will only concentrate their effort and time on difficult requests. Even if the dispatcher makes an error in prediction, the difficult request will reach the automation, the automation will fail on it, and the human will eventually receive that request. If the human gets an easy request by mistake, there's no problem either: that easy request can still be sent to the automation or processed by the human.

2.1.2 Cost of Machine Learning

Three factors highly influence the cost of a machine learning project:

- the difficulty of the problem,
- the cost of data, and
- the need for accuracy.

Getting the right data in the right amount can be very costly, especially if manual labeling is involved. The need for high accuracy can translate into the requirement of getting more data or training a more complex model, such as a unique **architecture** of a deep neural network or a nontrivial **ensembling** architecture.

When you think about the problem's difficulty, the primary considerations are:

- whether an implemented algorithm or a software library capable of solving the problem is available (if yes, the problem is greatly simplified),
- whether significant computation power is needed to build the model or to run it in the production environment.

The second driver of the cost is data. The following considerations have to be made:

- can data be generated automatically (if yes, the problem is greatly simplified),
- what is the cost of manual **annotation** of the data (i.e., assigning labels to unlabeled examples),
- how many examples are needed (usually, that cannot be known in advance, but can be estimated from known published results or the organization's own experience).

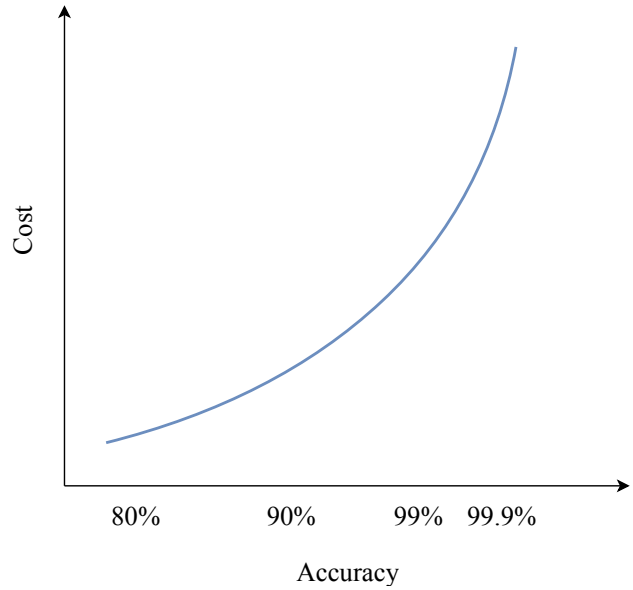


Figure 2.1: Superlinear growth of the cost as a function of accuracy requirement.

Finally, one of the most influential cost factors is the desired accuracy of the model. The machine learning project's cost grows superlinearly with the accuracy requirement, as illustrated in Figure 2.1. Low accuracy can also be a source of significant loss when the model is deployed in the production environment. The considerations to make:

- how costly is each wrong prediction, and
- what is the lowest accuracy level below which the model becomes impractical.

2.2 Estimating Complexity of a Machine Learning Project

There is no standard complexity estimation method for a machine learning project, other than by comparison with other projects executed by the organization or reported in the literature.

2.2.1 The Unknowns

There are several major unknowns that are almost impossible to guess with confidence unless you worked on a similar project in the past or read about such a project. The unknowns are:

- whether the required quality is attainable in practice,
- how much data you will need to reach the required quality,

- what features and how many features are necessary so that the model can learn and generalize sufficiently,
- how large the model should be (especially relevant for neural networks and ensemble architectures), and
- how long will it take to train one model (in other words, how much time is needed to run one **experiment**) and how many experiments will be required to reach the desired level of performance.

One thing you can almost be sure of: if the required level of model **accuracy** (one of the popular model quality metrics we consider in Section **??** of Chapter 5) is above 99%, you can expect complications related to an insufficient quantity of labeled data. In some problems, even 95% accuracy is considered very hard to reach. (Here we assume, of course, that the data is balanced, that is, there's no **class imbalance**. We will discuss class imbalance in Section 3.9 of the next chapter.)

Another useful reference is the human performance on the task. This is typically a hard problem if you want your model to perform as well as a human.

2.2.2 Simplifying the Problem

One way to make a more educated guess is to simplify the problem and solve a simpler problem first. For example, assume that the problem is that of classifying a set of documents into 1000 topics. Run a pilot project by focusing on 10 topics first, by considering documents belonging to other 990 topics as "Other."[1] Manually label the data for these 11 classes (10 real topics, plus "Other"). The logic here is that it's much simpler for a human to keep in mind the definitions of only 10 topics compared to memorizing the difference between 1000 topics[2].

Once you have simplified your problem to 11 classes, solve it, and measure time on every stage. Once you see that the problem for 11 classes is solvable, you can reasonably hope that it will be solvable for 1000 classes as well. Your saved measurements can then be used to estimate the time required to solve the full problem, though you cannot simply multiply this time by 100 to get an accurate estimate. The quantity of data needed to learn to distinguish between more classes usually grows superlinearly with the number of classes.

An alternative way of obtaining a simpler problem from a potentially complex one is to split the problem into several simple ones by using the natural slices in the available data. For example, let an organization have customers in multiple locations. If we want to train a model that predicts something about the customers, we can try to solve that problem only for one location, or for customers in a specific age range.

[1] Putting examples belonging to 990 classes in one class will likely create a highly imbalanced dataset. If it's the case, you would prefer to **undersample** the data in the class "Other." We consider data undersampling in Section 3.9 of the next chapter.

[2] To save even more time, apply clustering to the whole collection of unlabeled documents and only manually label documents belonging to one or a few clusters.

2.2.3 Nonlinear Progress

The progress of a machine learning project is nonlinear. The prediction error usually decreases fast in the beginning, but then the progress gradually slows down.[3] Sometimes you see no progress and decide to add additional features that could potentially depend on external databases or knowledge bases. While you are working on a new feature or labeling more data (or outsourcing this task), no progress in model performance is happening.

Because of this nonlinearity of progress, you should make sure that the product owner (or the client) understands the constraints and risks. Carefully log every activity and track the time it took. This will help not only in reporting, but also in the estimation of the complexity of similar projects in the future.

2.3 Defining the Goal of a Machine Learning Project

The **goal** of a machine learning project is to build a model that solves, or helps solve, a business problem. Within a project, the model is often seen as a black box described by the structure of its input (or inputs) and output (or outputs), and the minimum acceptable level of performance (as measured by accuracy of prediction or another **performance metric**).

2.3.1 What a Model Can Do

The model is typically used as a part of a system that serves some purpose. In particular, the model can be used within a broader system to:

- automate (for example, by taking action on the user's behalf or by starting or stopping a specific activity on a server),
- alert or prompt (for example, by asking the user if an action should be taken or by asking a system administrator if the traffic seems suspicious),
- organize, by presenting a set of items in an order that might be useful for a user (for example, by sorting pictures or documents in the order of similarity to a query or according to the user's preferences),
- annotate (for instance, by adding contextual annotations to displayed information, or by highlighting, in a text, phrases relevant to the user's task),
- extract (for example, by detecting smaller pieces of relevant information in a larger input, such as named entities in the text: proper names, companies, or locations),
- recommend (for example, by detecting and showing to a user highly relevant items in a large collection based on item's content or user's reaction to the past recommendations),
- classify (for example, by dispatching input examples into one, or several, of a predefined set of distinctly-named groups),

[3] The $80/20$ rule of thumb often applies: 80% of progress is made using the first 20% of resources.

- quantify (for example, by assigning a number, such as a price, to an object, such as a house),
- synthesize (for example, by generating new text, image, sound, or another object similar to the objects in a collection),
- answer an explicit question (for example, "Does this text describe that image?" or "Are these two images similar?"),
- transform its input (for example, by reducing its dimensionality for visualization purposes, paraphrasing a long text as a short abstract, translating a sentence into another language, or augmenting an image by applying a filter to it),
- detect a novelty or an anomaly.

Almost any business problem solvable with machine learning can be defined in a form similar to one from the above list. If you cannot define your business problem in such a form, likely, machine learning is not the best solution in your case.

2.3.2 Properties of a Successful Model

A successful model has the following four properties:

- it respects the input and output specifications and the performance requirement,
- it benefits the organization (measured via cost reduction, increased sales or profit),
- it helps the user (measured via productivity, engagement, and sentiment),
- it is scientifically rigorous.

A scientifically rigorous model is characterized by a predictable behavior (for the input examples that are similar to the examples that were used for training) and is reproducible. The former property (predictability) means that if input feature vectors come from the same distribution of values as the training data, then the model, on average, has to make the same percentage of errors as observed on the holdout data when the model was trained. The latter property (reproducibility) means that a model with similar properties can be easily built once again from the same training data using the same algorithm and values of hyperparameters. The word "easily" means that no additional analysis, labeling, or coding is necessary to rebuild the model, only the compute power.

When defining the goal of machine learning, make sure you solve the right problem. To give an example of an incorrectly defined goal, imagine your client has a cat and a dog and needs a system that lets their cat in the house but keeps their dog out. You might decide to train the model to distinguish cats from dogs. However, this model will also let *any cat* in and not just *their* cat. Alternatively, you may decide that because the client only has two animals, you will train a model that distinguishes between those two. In this case, because your classification model is binary, a raccoon will be classified as either the dog or the cat. If it's classified as the cat, it will be let in the house.[4]

[4]This is why having the class "Other" your classification problems is almost always a good idea.

2.4 Structuring a Machine Learning Team

There are two cultures of structuring a machine learning team, depending on the organization.

2.4.1 Two Cultures

One culture says that a machine learning team has to be composed of data analysts who collaborate closely with software engineers. In such a culture, a software engineer doesn't need to have deep expertise in machine learning, but has to understand the vocabulary of their fellow data analysts.

According to other culture, all engineers in a machine learning team must have a combination of machine learning and software engineering skills.

There are pros and cons in each culture. The proponents of the former say that each team member must be the best in what they do. A data analyst must be an expert in many machine learning techniques and have a deep understanding of the theory to come up with an effective solution to most problems, fast and with minimal effort. Similarly, a software engineer must have a deep understanding of various computing frameworks and be capable of writing efficient and maintainable code.

The proponents of the latter say that scientists are hard to integrate with software engineering teams. Scientists care more about how accurate their solution is and often come up with solutions that are impractical and cannot be effectively executed in the production environment. Also, because scientists don't usually write efficient, well-structured code, the latter has to be rewritten into production code by a software engineer; depending on the project, that can turn out to be a daunting task.

2.4.2 Members of a Machine Learning Team

Besides machine learning and software engineering skills, a machine learning team may include experts in data engineering (also known as data engineers) and experts in data labeling.

Data engineers are software engineers responsible for ETL (for Extract, Transform, Load). These three conceptual steps are part of a typical data pipeline. Data engineers use ETL techniques and create an automated pipeline, in which raw data is transformed into analysis-ready data. Data engineers design how to structure the data and how to integrate it from various resources. They write on-demand queries on that data, or wrap the most frequent queries into fast application programming interfaces (APIs) to make sure that the data is easily accessible by analysts and other data consumers. Typically, data engineers are not expected to know any machine learning.

In most big companies, data engineers work separately from machine learning engineers in a data engineering team.

Experts in data labeling are responsible for four activities:

- manually or semi-automatically assign labels to unlabeled examples according to the specification provided by data analysts,
- build labeling tools,
- manage outsourced labelers, and
- validate labeled examples for quality.

A **labeler** is person responsible for assigning labels to unlabeled examples. Again, in big companies, data labeling experts may be organized in two or three different teams: one or two teams of labelers (for example, one local and one outsourced) and a team of software engineers, plus a user experience (UX) specialist, responsible for building labeling tools.

When possible, invite domain experts to work closely with scientists and engineers. Employ domain experts in your decision making about the inputs, outputs, and features of your model. Ask them what they think your model should predict. Just the fact that the data you can get access to can allow you to predict some quantity doesn't mean the model will be useful for the business.

Discuss with the domain experts what they look for in the data to make a specific business decision; that will help you with feature engineering. Discuss also what clients pay for and what is a deal-breaker for them; that will help you to translate a business problem into a machine learning problem.

Finally, there are DevOps engineers. They work closely with machine learning engineers to automate model deployment, loading, monitoring, and occasional or regular model maintenance. In smaller companies and startups, a DevOps engineer may be part of the machine learning team, or a machine learning engineer could be responsible for the DevOps activities. In big companies, DevOps engineers employed in machine learning projects usually work in a larger DevOps team. Some companies introduced the MLOps role, whose responsibility is to deploy machine learning models in production, upgrade those models, and build data processing pipelines involving machine learning models.

2.5 Summary

Before a machine learning project starts, it must be prioritized, and the team working on the project must be built. The key considerations in the prioritization of a machine learning project, are impact and cost.

The impact of using machine learning is high when, 1) machine learning can replace a complex part in your engineering project, or 2) there's a great benefit in getting inexpensive (but probably imperfect) predictions.

The cost of the machine learning project is highly influenced by three factors: 1) the difficulty of the problem, 2) the cost of data, and 3) the needed model performance quality.

There is no standard method of estimation of how complex a machine learning project is other than by comparison with other projects executed by the organization or reported in the literature. There are several major unknowns that are almost impossible to guess: whether the required level of model performance is attainable in practice, how much data you will need to reach that level of performance, what features and how many features are needed, how large the model should be, and how long will it take to run one experiment and how many experiments will be needed to reach the desired level of performance.

One way to make a more educated guess is to simplify the problem and solve a simpler one.

The progress of a machine learning project is nonlinear. The error usually decreases fast in the beginning, but then the progress slows down. Because of this nonlinearity of progress, it's better to make sure that the client understands the constraints and the risks. Carefully log every activity and track the time it took. This will help not only in reporting but also in estimating complexity for similar projects in the future.

The goal of a machine learning project is to build a model that solves some business problem. In particular, the model can be used within a broader system to automate, alert or prompt, organize, annotate, extract, recommend, classify, quantify, synthesize, answer an explicit question, transform its input, and detect novelty or an anomaly. If you cannot frame the goal of machine learning in one of these forms, likely, machine learning is not the best solution.

A successful model 1) respects the input and output specifications and the minimum performance requirement, 2) benefits the organization and the user, and 3) is scientifically rigorous.

There are two cultures of structuring a machine learning team, depending on the organization. One culture says that a machine learning team has to be composed of data analysts who collaborate closely with software engineers. In such a culture, a software engineer doesn't need to have profound expertise in machine learning but has to understand the vocabulary of their fellow data analysts or scientists. According to the other culture, all engineers in a machine learning team must have a combination of machine learning and software engineering skills.

Besides having machine learning and software engineering skills, a machine learning team may include experts in data labeling and data engineering experts. DevOps engineers work closely with machine learning engineers to automate model deployment, loading, monitoring, and occasional or regular model maintenance.

Chapter 3

Data Collection and Preparation

Before any machine learning activity can start, the analyst must collect and prepare the data. The data available to the analyst is not always "right" and is not always in a form that a machine learning algorithm can use. This chapter focuses on the second stage in the machine learning project life cycle, as shown below:

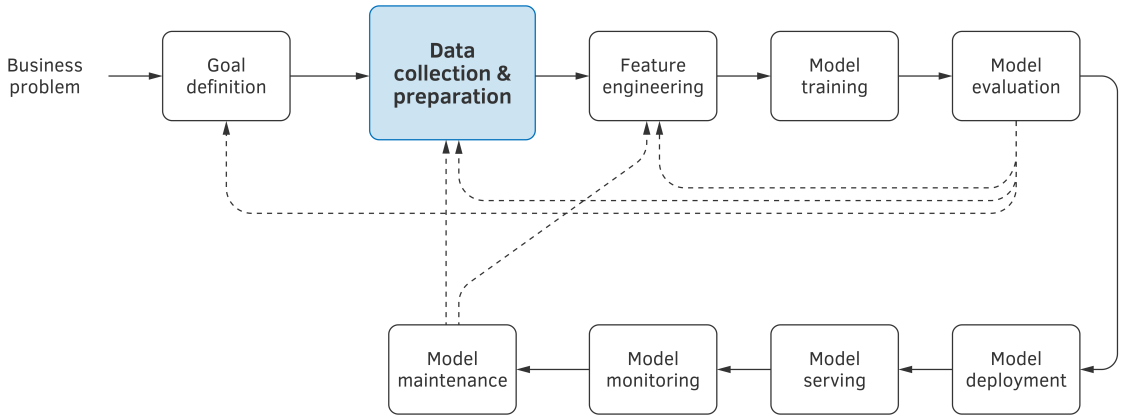


Figure 3.1: Machine learning project life cycle.

In particular, we talk about the properties of good quality data, typical problems a dataset can have, and ways to prepare and store data for machine learning.

3.1 Questions About the Data

Now that you have a machine learning goal with well-defined model input, output, and success criteria, you can start collecting the data needed to train your model. However, before you start collecting the data, there are some questions to answer.

3.1.1 Is the Data Accessible?

Does the data you need already exist? If yes, is it accessible (physically, contractually, ethically, or from a cost perspective)? If you are purchasing or re-using someone else's data sources, have you considered how that data might be used or shared? Do you need to negotiate a new licensing agreement with the original supplier?

If the data is accessible, is it protected by copyright or other legal norms? If so, have you established who owns the copyright in your data? Might there be joint copyright?

Is the data sensitive (e.g., concerning your organization's projects, clients, or partners, or it is classified by the government), and are there any potential privacy issues? If so, have you discussed data sharing with the respondents from whom you collected the data? Can you preserve personal information for a long-term so that it can be used in the future?

Do you need to share the data along with the model? If so, do you need to get written consent from owners or respondents?

Do you need to anonymize data,[1] for example, to remove **personally identifiable information** (**PII**), during analysis or in preparation for sharing?

Even if it's physically possible to get the data you need, don't work with it until all the above questions are resolved.

3.1.2 Is the Data Sizeable?

The question for which you would like to have a definitive answer is whether there's enough data. However, as we already found out, it's usually not known how much data is needed to reach your goal, especially if the minimum model quality requirement is stringent.

If you have doubts about the immediate availability of sufficient data, find out how frequently new data gets generated. For some projects, you can start with what's initially available and, while you are working on feature engineering, modeling, and solving other relevant technical problems, new data might gradually come in. It can come in naturally, as the result of some

[1] An illustrative example is the content redistribution policy of Twitter. The policy restricts the sharing of tweet information other than tweet IDs and user IDs. Twitter wants the analysts always to pull fresh data using Twitter API. One possible explanation of such a restriction is that some users might want to delete a particular tweet because they changed their mind or found it too controversial. If that tweet has already been pulled and is shared on a public domain, then it might make that user vulnerable.

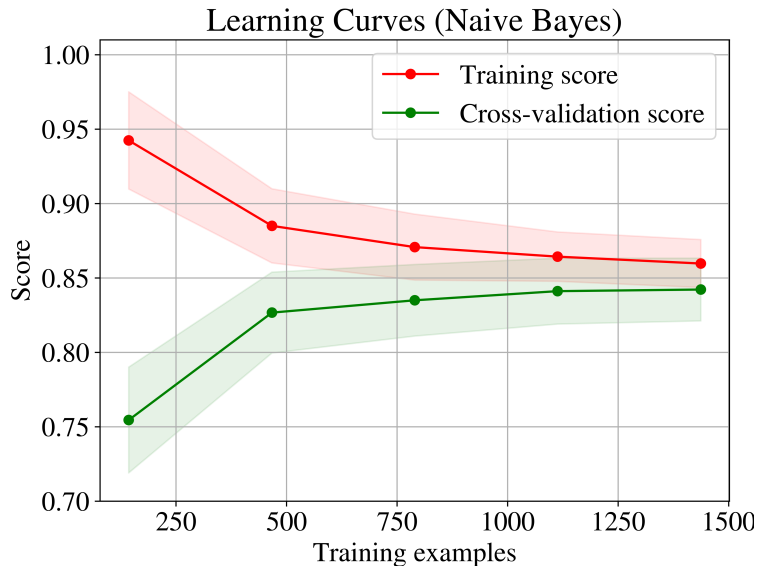


Figure 3.2: Learning curves for the Naïve Bayes learning algorithm applied to the standard "digits" dataset of scikit-learn.

observable or measurable process, or progressively be provided by your data labeling experts or a third-party data provider.

Consider the estimated time needed to accomplish the project. Will a sufficiently large dataset[2] be gathered during this time? Base your answer on the experience working on similar projects or results reported in the literature.

One practical way to find out if you have collected sufficient data is to plot **learning curves**. More specifically, plot the training and validation scores of your learning algorithm for varying numbers of training examples, as shown in Figure 3.2.

By looking at the learning curves, you will see your model's performance will plateau after you reach a certain number of training examples. Upon reaching that number of training examples, you will begin to experience diminishing returns from additional examples.

If you observe the performance of the learning algorithm plateaued, it might be a sign that collecting more data will not help in training a better model. I used the expression "might be" because two other explanations are possible:

- you didn't have enough informative features that your learning algorithm can leverage to build a more performant model, or

[2]Don't forget, in your estimates, that you need not just training but also holdout data to validate the model performance on the examples it wasn't trained on. That holdout data also has to be sizeable to provide reliable, in a statistical sense, model quality estimates.

- you used a learning algorithm incapable of training a complex enough model using the data you have.

In the former case, you might think about engineering additional features by combining the existing features in some clever ways, or by using information from indirect data sources, such as lookup tables and gazetteers. We consider techniques for synthesizing features in Section **??** of Chapter 4.

In the latter case, one possible approach would be to use an ensemble learning method or train a deep neural network. However, deep neural networks usually require more training data compared to shallow learning algorithms.

Some practitioners use rules of thumb to estimate the number of training examples needed for a problem. Usually, they are using scaling factors applied to either,

- number of features, or
- number of classes, or
- number of trainable parameters in the model.

Such rules of thumb often work, but they are different for different problem domains. Each analyst adjusts the numbers based on experience. While you would discover by experience those "magical" scaling factors that work for you, the most frequently cited numbers in various online sources are:

- 10 times the amount of features (this often exaggerates the size of the training set, but works well as an upper bound),
- 100 or 1000 times the number of classes (this often underestimates the size), or
- ten times the number of trainable parameters (usually applied to neural networks).

Keep in mind that just because you have big data does not mean that you should use all of it. A smaller sample of big data can give good results in practice and accelerate the search for a better model. It's important to ensure, though, that the sample is representative of the whole big dataset. Sampling strategies such as **stratified** and **systematic sampling** can lead to better results. We consider data sampling strategies in Section 3.10.

3.1.3 Is the Data Useable?

The data quality is one of the major factors affecting the model performance. Imagine that you want to train a model that predicts a person's gender, given their name. You might acquire a dataset of people that contains gender information. However, if you use this dataset blindly, you might realize that no matter how hard you try to improve the quality of your model, its performance on new data is low. What is the reason for such a weak performance?

The answer could be that the gender information was not factual but obtained using a rather low-quality statistical classifier. In this case, the best you can achieve with your model is the performance of that low-quality classifier.

If the dataset comes in the form of a spreadsheet, the first thing to check is if the data in the spreadsheet is tidy. As discussed in the introduction, the dataset used for machine learning must be tidy. If it's not the case for your data, you must transform it into tidy data using, as already mentioned, feature engineering.

A tidy dataset can have **missing values**. Consider **data imputation** techniques to fill the missing values. We will discuss several such techniques in Section 3.7.

One frequent problem with datasets compiled by a human is that people can decide to indicate missing values with some **magic number** like 9999 or -1. Such situations must be spotted during the visual analysis of the data, and those magic numbers have to be replaced using an appropriate data imputation technique.

Another property to validate is whether the dataset contains **duplicates**. Usually, duplicates are removed, unless you added them on purpose to balance an **imbalanced problem**. We consider this problem and methods to alleviate it in Section 3.9.

Data can be **expired** or be significantly not up to date. For example, let your goal be to train a model that recognizes abnormality in the behavior of a complex piece of electronic appliance, such as a printer. You have measurements taken during the normal and abnormal functioning of a printer. However, these measurements have been recorded for a previous generation of printers, while the new generation has received several significant upgrades since then. The model trained using such expired data from an older printer generation might perform worse when deployed on the new generation of printers.

Finally, data can be **incomplete** or **unrepresentative** of the phenomenon. For example, a dataset of animal photos might contain pictures taken only during the summer or in a specific geography. A dataset of pedestrians for self-driving car systems might be created with engineers posing as pedestrians; in such a dataset, most situations would include only younger men, while children, women, and the elderly would be underrepresented or entirely absent.

A company working on facial expression recognition might have the research and development office in a predominantly white location, so the dataset would only show faces of white men and women, while black or Asian people would be underrepresented. Engineers developing a posture recognition model for a camera might build the training dataset by taking pictures of people indoors, while the customers would typically use the camera outdoors.

In practice, data can only become useable for modeling after preprocessing; hence the importance of visual analysis of the dataset before you start modeling. Let's say you work on a problem of predicting the topic in news articles. It's likely you will scrape your data from news websites. It's also likely that download dates would be saved in the same document as the news article text. Imagine also that the data engineer decides to loop over news topics mentioned on the websites and scrape one topic at a time. So, on Monday the arts-related articles were scraped, on Tuesday — sports, on Wednesday — technology, and so on.

If you don't preprocess such data by removing the dates, the model can learn the date-topic correlation, and such a model will be of no practical use.

3.1.4 Is the Data Understandable?

As demonstrated in gender prediction, it's crucial to understand from where each attribute in the dataset came. It is equally important to understand what each attribute exactly represents. One frequent problem observed in practice is when the variable that the analyst tries to predict is found among the features in the feature vector. How could that happen?

Imagine that you work on the problem of predicting the price of a house from its attributes such as the number of bedrooms, surface, location, year of construction, and so on. The attributes of each house were provided to you by the client, a large online real estate sales platform. The data has the form of an Excel spreadsheet. Without spending too much time analyzing each column, you remove only the transaction price from the attributes and use that value as the target you want to learn to predict. Very quickly you realize that the model is almost perfect: it predicts the transaction price with accuracy near 100%. You deliver the model to the client, they deploy it in production, and the tests show that the model is wrong most of the time. What happened?

What did happen is called **data leakage** (also known as **target leakage**). After a more careful examination of the dataset, you realize that one of the columns in the spreadsheet contained the real estate agent's commission. Of course, the model easily learned to convert this attribute into the house price perfectly. However, this information is not available in the production environment before the house is sold, because the commission depends on the selling price. In Section 3.2.8, we will consider the problem of data leakage in more detail.

3.1.5 Is the Data Reliable?

The reliability of a dataset varies depending on the procedure used to gather that dataset. Can you trust the labels? If the data was produced by the workers on Mechanical Turk (so-called "turkers"), then the reliability of such data might be very low. In some cases, the labels assigned to feature vectors might be obtained as a majority vote (or an average) of several turkers. If that's the case, the data can be considered more reliable. However, it's better to do additional validation of quality on a small random sample of the dataset.

On the other hand, if the data represents measurements made by some measuring devices, you can find the details of each measurement's accuracy in the technical documentation of the corresponding measuring device.

The reliability of labels can also be affected by the **delayed** or **indirect** nature of the label. The label is considered delayed when the feature vector to which the label was assigned represents something that happened significantly earlier than the time of label observation.

To be more concrete, take the **churn prediction** problem. Here, we have a feature vector describing a customer, and we want to predict whether the customer will leave at some point in the future (typically six months to one year from now). The feature vector represents what we know about the customer now, but the label ("left" or "stayed") will be assigned in the

future. This is an important property, because between now and the future, many events, not reflected in our feature vector might happen which would affect the customer's decision to stay or leave. Therefore delayed labels make our data less reliable.

Whether a label is direct or indirect also affects reliability, depending, of course, on what we are trying to predict. For example, let's say our goal is to predict whether the website visitor will be interested in a webpage. We might acquire a certain dataset containing information about users, webpages, and labels "interested"/"not_interested" reflecting whether a specific user was interested in a particular webpage. A direct label would indeed indicate interest, while an indirect label could suggest some interest. For example, if the user pressed the "Like" button, we have the direct indicator of interest. However, if the user only clicked on the link, this could be an indicator of some interest, but it's an indirect indicator. The user could have clicked by mistake, or because the link text was a clickbait, we cannot know for sure. If the label is indirect, this also makes such data less reliable. Of course, it's less reliable for predicting the interest, but can be perfectly reliable for predicting clicks.

Another source of unreliability in the data is feedback loops. A **feedback loop** is a property in the system design when the data used to train the model is obtained using the model itself. Again, imagine that you work on a problem of predicting whether a specific user of a website will like the content, and you only have indirect labels – clicks. If the model is already deployed on the website and the users click on links recommended by the model, this means that the new data indirectly reflects not only the interest of users to the content, but also how intensively the model recommended that content. If the model decided that a specific link is important enough to recommend to many users, more users would likely click on that link, especially if the recommendation was made repeatedly during several days or weeks.

3.2 Common Problems With Data

As we have just seen, the data you will work with can have problems. In this section, we cite the most important of these problems and what you can do to alleviate them.

3.2.1 High Cost

Getting unlabeled data can be expensive; however, labeling data is the most expensive work, especially if the work is done manually.

Getting unlabeled data becomes expensive when it nust be gathered specifically for your problem. Let's say your goal is to know where different types of commerce are located in a city. The best solution would be to buy this data from a government agency. However, for various reasons it can be complicated or even impossible: the government database may be incomplete or outdated. To get up-to-date data, you may decide to send cars equipped with cameras on the streets of a given city. They would take pictures of all buildings on the streets.

Original photo Labeled photo

Figure 3.3: The unlabeled and labeled aerial photo. Photo credit: Tom Fisk.

As you might imagine, such an enterprise is not cheap. Collecting pictures of the buildings is not enough. We need the type of commerce in every building. Now we need labeled data: "coffee house," "bank," "grocery," "drug store," "gas station," etc. These must be assigned manually, and paying someone to do that work is expensive. By the way, Google has a clever technique outsourcing the labeling to random people with its free reCAPTCHA service. reCAPTCHA thus solves two problems: reducing spam on the Web and providing cheap labeled data to Google.

In Figure 3.3, you can see the work needed to label one image. The goal here is to segment a picture by assigning labels to every pixel from the following: "heavy truck," "car or light truck," "boat," "building," "container," "other." Labeling image in Figure 3.3 took me about 30 minutes. If there were more types, for example "motorcycle," "tree," "road," it would take longer, and the labeling cost would be higher.

Well-designed labeling tools will minimize mouse use (including menus activated by mouse clicks), maximize hotkeys, and reduce costs by increasing the speed of data labeling.

Whenever possible, reduce decision-making to a yes/no answer. Instead of asking "Find all prices in this text", extract all numbers from the text and then display each number, one by one, asking, "Is this number a price?" as shown in 3.4. If the labeler clicks "Not Sure," you can save this example to analyze later or simply not use such examples for training the model.

Another trick allowing for accelerated labeling is **noisy pre-labeling** consisting of pre-labeling the example using the current best model. In this scenario, you start by labeling a certain quantity of examples "from scratch" (that is, without using any support). Then you build the first model that works reasonably well, using this initial set of labeled examples. Next, use the current model and label each new example in place of the human labeler.[3] Ask whether the

[3]This is why it's called a "noisy" pre-labeling: the labels assigned to examples using a sub-optimal model would

automatically assigned label is correct. If the labeler clicks "Yes," save this example as usual. If they click "No," then ask to label this example manually. See the workflow chart illustrating this process in Figure 3.5. The goal of a good labeling process design is to make the labeling as streamlined as possible. Keeping the labeler engaged is also key. Show progress in the number of labels added, as well as the quality of the current best model. This engages the labeler and adds purpose to the labeling task.

Figure 3.4: An example of simple labeling interface.

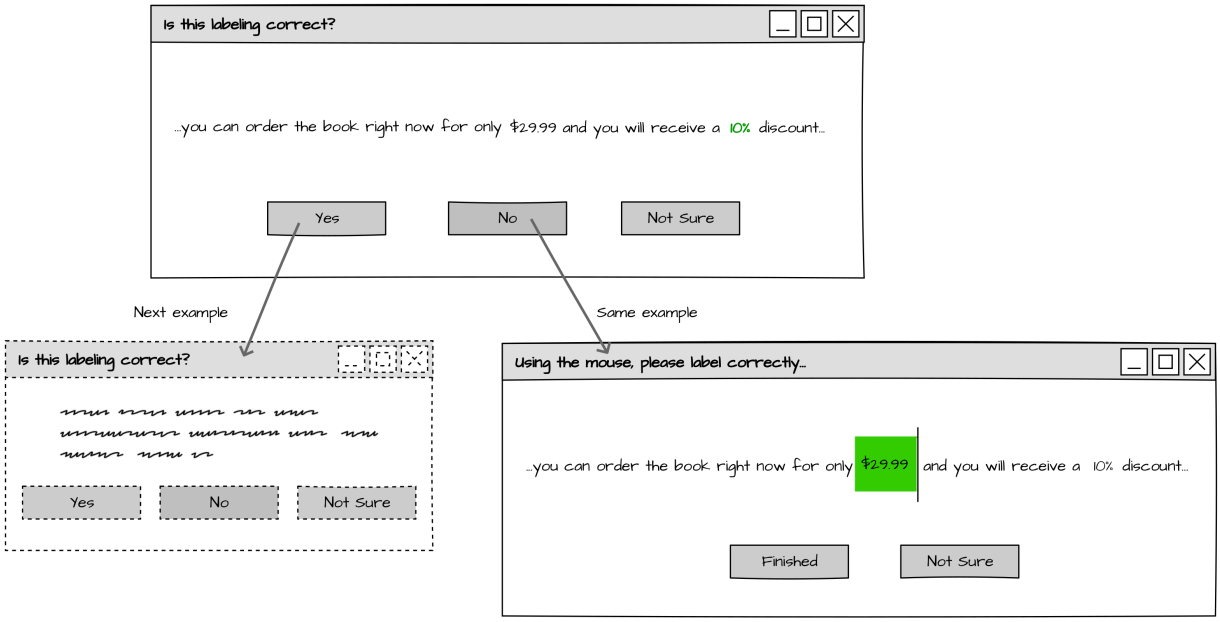


Figure 3.5: An example of noisy pre-labeling workflow.

not all be accurate and require a human validation.

3.2.2 Bad Quality

Remember that data quality is one of the major factors affecting the performance of the model. I cannot stress it strongly enough.

Data quality has two components: raw data quality and labeling quality.

Some common problems with raw data are noise, bias, low predictive power, outdated examples, outliers, and leakage.

3.2.3 Noise

Noise in data is a corruption of examples. Images can be blurry or incomplete. Text can lose formatting, which makes some words concatenated or split. Audio data can have noise in the background. Poll answers can be incomplete or have missing attributes, such as the responder's age or gender. Noise is often a random process that corrupts each example independently of other examples in the collection.

If tidy data has missing attributes, **data imputation** techniques can help in guessing values for those attributes. We will consider data imputation techniques in Section 3.7.1.

Blurred images can be deblurred using specific image deblurring algorithms, though deep machine learning models, such as neural networks, can learn to deblur if needed. The same can be said about noise in audio data: it can be algorithmically suppressed.

Noise is more a problem when the dataset is relatively small (thousands of examples or less), because the presence of noise can lead to **overfitting**: the algorithm may learn to model the noise contained in the training data, which is undesirable. In the big data context, on the other hand, noise, if it's randomly applied to each example independently of other examples in the dataset, is typically "averaged out" over multiple examples. In that latter context, noise can bring a regularization effect as it prevents the learning algorithm from relying too much on a small subset of input features.[4]

3.2.4 Bias

Bias in data is an inconsistency with the phenomenon that data represents. This inconsistency may occur for a number of reasons (which are not mutually exclusive).

Types of Bias

Selection bias is the tendency to skew your choice of data sources to those that are easily available, convenient, and/or cost-effective. For example, you might want to know the opinion of the readers on your new book. You decide to send several initial chapters to the

[4]This is, by the way, the rationale behind the increase in performance brought by the **dropout** regularization technique in **deep learning**.

mailing list of your previous book's readers. It's very likely this select group will like your new book. However, this information doesn't tell you much about a general reader.

A real-life example of selection bias is an image generated by the Photo Upsampling via Latent Space Exploration (**PULSE**) algorithm that uses a neural network model to upscale (increase the resolution) of images. When Internet users tested it, they discovered that an upscaled image of a black person could, in some cases, represent a white person, as illustrated by Barack Obama's upscaled photo shown in Figure 3.6.

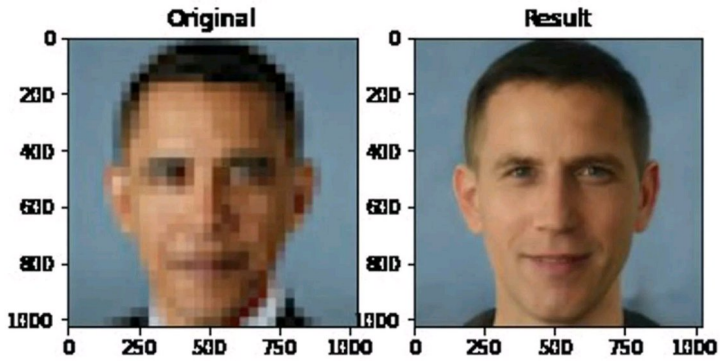


Figure 3.6: An illustration of the effect the selection bias can have on the trained model. Image: Twitter / @Chicken3gg.

The above example shows that it cannot be assumed that machine learning model is correct, simply because machine learning algorithms are impartial and the trained models are based on data. If the data has a bias, it will most likely be reflected in the model.

Self-selection bias is a form of selection bias where you get the data from sources that "volunteered" to provide it. Most poll data has this type of bias. For example, you want to train a model that predicts the behavior of successful entrepreneurs. You decide to first ask entrepreneurs whether they are successful or not. Then you only keep the data obtained from those who declared themselves successful. The problem here is that most likely, really successful entrepreneurs don't have time to answer your questions, while those who claim themselves successful can be wrong on that matter.

Here's another example. Let's say, you want to train a model that predicts whether a book will be liked by the readers. You can use the rating users gave to similar books in the past. However, it turns out that unhappy users tend to provide disproportionally low ratings. The data will be biased towards too many very low ratings as compared to the quantity of mid-range ratings, as shown in Figure 3.7. The bias is compounded by the fact that we tend to rate only when the experience was either very good or very bad.

Figure 3.7: The distribution of ratings given by the readers to a popular AI book on Amazon.

Omitted variable bias happens when your featurized data doesn't have a feature necessary for accurate prediction. For example, let's assume that you are working on a churn prediction model and you want to predict whether a customer cancels their subscription within six months. You train a model, and it's accurate enough; however, several weeks after deployment you see many unexpected false negatives. You investigate the decreased model performance and discover a new competitor now offers a very similar service for a lower price. This feature wasn't initially available to your model, therefore important information for accurate prediction was missing.

Sponsorship or **funding bias** affects the data produced by a sponsored agency. For example, let a famous video game company sponsor a news agency to provide news about the video game industry. If you try to make a prediction about the video game industry, you might include in your data the story produced by this sponsored agency.

However, sponsored news agencies tend to suppress bad news about their sponsor and exaggerate their achievements. As a result, the model's performance will be suboptimal.

Sampling bias (also known as **distribution shift**) occurs when the distribution of examples used for training doesn't reflect the distribution of the inputs the model will receive in production. This type of bias is frequently observed in practice. For example, you are working on a system that classifies documents according to a taxonomy of several hundred topics. You might decide to create a collection of documents in which an equal amount of documents represents each topic. Once you finish the work on the model, you observe 5% error. Soon after deployment, you see the wrong assignment to about 30% of documents. Why did this happen?

One of the possible reasons is sampling bias: one or two frequent topics in production data might account for 80% of all input. If your model doesn't perform well for these frequent topics, then your system will make more errors in production than you initially expected.

Prejudice or **stereotype bias** is often observed in data obtained from historical sources, such as books or photo archives, or from online activity such as social media, online forums, and comments to online publications.

Using a photo archive to train a model that distinguishes men from women might show, for example, men more frequently in work or outdoor contexts, and women more often at home indoors. If we use such biased data, our model will have more difficulty recognizing a woman outdoors or a man at home.

A famous example of this type of bias is looking for associations for words using **word embeddings** trained with an algorithm like **word2vec**. The model predicts that king $-$ man $+$ woman $\approx$ queen, but at the same time, that programmer $-$ man $+$ woman $\approx$ homemaker.

Systematic value distortion is bias usually occurring with the device making measurements or observations. This results in a machine learning model making suboptimal predictions when deployed in the production environment.

For example, the training data is gathered using a camera with a white balance which makes white look yellowish. In production, however, engineers decide to use a higher-quality camera which "sees" white as white. Because your model was trained on lower-quality pictures, the predictions using higher-quality input will be suboptimal.

This should not be confused with noisy data. Noise is the result of a random process that distorts the data. When you have a sufficiently large dataset, noise becomes less of a problem because it might average out. On the other hand, if the measurements are consistently skewed in one direction, then it damages training data, and ultimately results in a poor-quality model.

Experimenter bias is the tendency to search for, interpret, favor, or recall information in a way that affirms one's prior beliefs or hypotheses. Applied to machine learning, experimenter bias often occurs when each example in the dataset is obtained from the answers to a survey given by a particular person, one example per person.

Usually, each survey contains multiple questions. The form of those questions can significantly affect the responses. The simplest way for a question to affect the response is to provide limited response options: "Which kind of pizza do you like: pepperoni, all meats, or vegetarian?" This doesn't leave the choice of giving a different answer, or even "Other."

Alternatively, a survey question might be constructed with a built-in slant. Instead of asking, "Do you recycle?" an analyst with a experimenter bias might ask, "Do you dodge from recycling?" In the former case, the respondent is more likely to give an honest answer, compared to the latter.

Furthermore, experimenter bias might happen when the analyst is briefed in advance to support a particular conclusion (for example, the one in favor of doing "business as usual"). In that situation, they can exclude specific variables from the analysis as unreliable or noisy.

Labeling bias happens when labels are assigned to unlabeled examples by a biased process or person. For example, if you ask several labelers to assign a topic to a document by reading the document, some labelers can indeed read the document entirely and assign well-thought

labels. In contrast, others could just try to quickly "scan" the text, spot some keyphrases and choose the topic that corresponds the best to the selected keyphrases. Because each person's brain pays more attention to keyphrases from a specific domain or domains and less to others, the labels assigned by labelers who scan the text without reading will be biased.

Alternatively, some labelers would be more interested in reading documents on some topics that they personally prefer. If it's the case, a labeler might skip uninteresting documents, and the latter will be underrepresented in your data.

Ways to Avoid Bias

It is usually impossible to know exactly what biases are present in a dataset. Furthermore, even knowing there are biases, avoiding them is a challenging task. First of all, be prepared.

A good habit is to question everything: who created the data, what were their motivations and quality criteria, and more importantly, how and why the data was created. If the data is a result of some research, question the research method and make sure that it doesn't contribute to any of the biases described above.

Selection bias can be avoided by systematically questioning the reason why a specific data source was chosen. If the reason is simplicity or low cost, then pay careful attention. Recall the example whether a specific customer would subscribe to your new offering. Training the model using only the data about your current customers is likely a bad idea, because your existing customers are more loyal to your brand than a random potential customer. Your estimates of model's quality will be overly optimistic.

Self-selection bias cannot be completely eliminated. It usually appears in surveys; the mere consent of the responder to answer the questions represents self-selection bias. The longer the survey, the less likely the respondent will answer with a high degree of attention. Therefore, keep your survey short and provide an incentive to give quality answers.

Pre-select responders to reduce self-selection. Don't ask entrepreneurs whether they consider themselves successful. Rather, build a list based on references from experts or publications, and only contact those individuals.

It's tough to avoid the **omitted variable bias** completely, because, as they say, "we don't know what we don't know." One approach is to use all available information, that is, to include in your feature vector as many features as possible, even those you deem unnecessary. This could make your feature vector very wide (i.e., of many dimensions) and sparse (i.e., when the values in most dimensions are zero). Still, if you use a well-tuned regularization, your model will "decide" which features are important, and which ones aren't.

Alternatively, let us suspect that a particular variable would be important for accurate predictions, and leaving it out of our model could result in an omitted variable bias. Suppose getting that data is problematic. Try using a proxy variable in lieu of the omitted variable. For instance, if we want to train a model that predicts the price of a used car, and we cannot get the car's age, use, instead, the length of ownership by its current owner. The amount of time the current owner owned the car can be taken as a proxy for the age of the vehicle.

Sponsorship bias can be reduced by carefully investigating the data source, specifically the source owner's incentive to provide the data. For example, it's known that publications on tobacco and pharmaceutical drugs are very often sponsored by tobacco and pharmaceutical companies, or their opponents. The same can be said about news companies, especially those that depend on the advertisement revenue or have an undisclosed business model.

Sampling bias can be avoided by researching the real proportion of various properties in the data that will be observed in production, and then sampling the training data by keeping similar proportions.

Prejudice or **stereotype bias** can be controlled. When developing the training model to distinguish pictures of women from men, a data analyst could choose to under-sample the number of women indoors, or oversample the number of men at home. In other words, prejudice or stereotype bias is reduced by exposing the learning algorithm to a more even-handed distribution of examples.

Systematic value distortion bias can be alleviated by having multiple measuring devices, or hiring humans trained to compare the output of measuring or observing devices.

Experimenter bias can be avoided by letting multiple people validate the questions asked in the survey. Ask yourself: "Do I feel uncomfortable or constrained answering this question?"

Furthermore, despite more difficulties in analysis, opt for open-ended questions rather than yes/no or multiple-choice questions. If you still prefer to give responders a choice of answers, include the option "Other" and a place to write a different answer.

Labeling bias can be avoided by asking several labelers to identify the same example. Ask the labelers why they decided to assign a specific label to examples that produced different results. If you see that some labelers refer to certain keyphrases, rather than trying to paraphrase the entire document, you can identify those who are quickly scanning instead of reading.

You can also compare the frequency of skipped documents for different labelers. If you see that a labeler skips documents more often than the average, ask if they encountered technical problems, or simply were not interested in some topics.

You cannot entirely avoid bias in data. There's no silver bullet. As a general rule, keep a human in the loop, especially if your model affects people's lives.

Recall that there is a temptation among the data analysts to assume that machine learning models are inherently fair because they make decisions based on evidence and math, as opposed to often messy or irrational human judgments. This is, unfortunately, not always the case: inevitably, a model trained on biased data will produce biased results.

It is the duty of people training the model to ensure that the output is fair. But what's fair, you may ask? Unfortunately, again, there is no silver bullet measurement that would always detect unfairness. Choosing an appropriate definition of model fairness is always problem-specific and requires human judgment. In Section **??** of Chapter 7, we consider several definitions of **fairness** in machine learning.

Human involvement in all stages of data gathering and preparation is the best approach to make sure that the possible damage caused by machine learning is minimized.

3.2.5 Low Predictive Power

Low predictive power is an issue that you often don't consider until you have spent fruitless energy trying to train a good model. Does the model underperform because it is not expressive enough? Does the data not contain enough information from which to learn? You don't know.

Suppose the goal is to predict whether a listener will like a new song on a music streaming service. Your data is the name of the artist, the song title, lyrics, and whether that song is in their playlist. The model you train with this data will be far from perfect.

Artists who are not in the listener's playlist are unlikely to receive a high score from the model. Furthermore, many users will only add some songs of a specific artist to their playlist. Their musical preferences are significantly influenced by the song arrangement, choice of instruments, sound effects, tone of voice, and subtle changes in tonality, rhythm, and beat. These are properties of songs that cannot be found in lyrics, title, or the artist's name; they have to be extracted from the sound file.

On the other hand, extracting these relevant features from an audio file is challenging. Even with modern neural networks, recommending songs based on how they sound is considered a hard task for artificial intelligence. Typically, song recommendations are developed by comparing playlists of different listeners and finding those with similar compositions.

Consider a different example of low predictive power. Let's say we want to train a model that will predict where to point the telescope and observe something interesting. Our data are photos of various regions of the sky where something unusual was captured in the past. Based on these photos alone, it's very unlikely that we will be able to train a model that accurately predicts such an event. However, if we add to this data the measurements of various sensors, such as those measuring radiofrequency signals from different zones, or particle bursts, it is more likely we will be able to make better predictions.

Your work may be especially challenging the first time working with the dataset. If you cannot obtain acceptable results, no matter how complex the model becomes, it may be time to consider the problem of low predictive power. Engineer as many additional features as possible (apply your creativity!). Consider indirect data sources to enrich feature vectors.

3.2.6 Outdated Examples

Once you build the model and deploy it in production, the model usually performs well for some time. This period depends entirely on the phenomenon you are modeling.

Typically, as we will discuss in Section **??** of Chapter 9, a certain model quality monitoring procedure is deployed in the production environment. Once an erratic behavior is detected, new training data is added to adjust the model; the model is then retrained and redeployed.

Often, the cause of an error is explained by the finiteness of the training set. In such cases, additional training examples will solidify the model. However, in many practical scenarios, the model starts to make errors because of **concept drift**. Concept drift is a fundamental change in the statistical relationship between the features and the label.

Imagine your model predicts whether a user will like certain content on a website. Over time, the preferences of some users may start to change, perhaps due to aging, or because a user discovers something new (I didn't listen to jazz three years ago, now I do!). The examples added to the training data in the past no longer reflect some user's preferences and start hurting the model performance, rather than contributing to it. This is concept drift. Consider it if you see a decreasing trend in model performance on new data.

Correct the model by removing the outdated examples from the training data. Sort your training examples, most recent first. Define an additional hyperparameter — what percentage of the most recent examples to use to retrain the model — and tune it using **grid search**, or another hyperparameter tuning technique.

Concept drift is an example of a broader problem known as **distribution shift**. We consider hyperparameter tuning and other types of distribution shift in Sections **??** and **??**.

3.2.7 Outliers

Outliers are examples that look dissimilar to the majority of examples from the dataset. It's up to the data analyst to define "dissimilar." Typically, dissimilarity is measured by some distance metric, such as **Euclidean distance**.

In practice, however, what seems to be an outlier in the original feature vector space can be a typical example in a feature vector space transformed using tools such as a **kernel function**. Feature space transformation is often explicitly done by a kernel-based model, such as **support vector machine** (SVM), or implicitly by a deep neural network.

Shallow algorithms, such as linear or logistic regression, and some ensemble methods, such as AdaBoost, are particularly sensitive to outliers. SVM has one definition that is less sensitive to outliers: a special penalty hyperparameter regulates the influence of misclassified examples (which often happen to be outliers) on the **decision boundary**. If this penalty value is low, the SVM algorithm may completely ignore outliers from consideration when drawing the decision boundary (an imaginary hyperplane separating positive and negative examples). If it's too low, even some regular examples can end up on the wrong side of the decision boundary. The best value for that hyperparameter should be found by the analyst using a hyperparameter tuning technique.

A sufficiently complex neural network can learn to behave differently for each outlier in the dataset and, at the same time, still work well for the regular examples. It's not the desired outcome, as the model becomes unnecessarily complex for the task. More complexity results in longer training and prediction time, and poorer generalization after production deployment.

Whether to exclude outliers from the training data, or to use machine learning algorithms and models robust to outliers, is debatable. Deleting examples from a dataset is not considered scientifically or methodologically sound, especially in small datasets. In the big data context, on the other hand, outliers don't typically have a significant influence on the model.

From a practical standpoint, if excluding some training examples results in better performance of the model on the holdout data, the exclusion may be justified. Which examples to consider for exclusion can be decided based on a certain similarity measure. A modern approach to getting such a measure is to build an **autoencoder** and use the reconstruction error[5] as the measure of (dis)similarity: the higher the reconstruction error for a given example, the more dissimilar it is to the dataset.

3.2.8 Data Leakage

Data leakage, also called **target leakage**, is a problem affecting several stages of the machine learning life cycle, from data collection to model evaluation. In this section, I will only describe how this problem manifests itself at the data collection and preparation stages. In the subsequent chapters, I will describe its other forms.

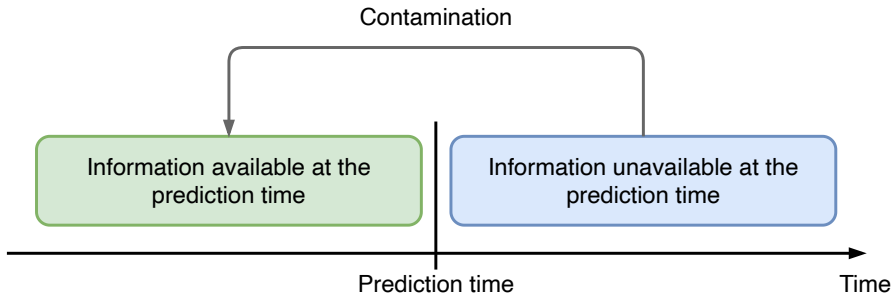


Figure 3.8: Data leakage in a nutshell.

Data leakage in supervised learning is the unintentional introduction of information about the target that should not be made available. We call it "contamination" (Figure 3.8). Training on contaminated data leads to overly optimistic expectations about the model performance.

3.3 What Is Good Data

We already considered questions to answer about the data before to start collecting it and the common problems with the data an analyst might encounter. But what constitutes good data

[5] An autoencoder model is trained to reconstruct its input from an **embedding** vector. The hyperparameters of the autoencoder are tuned to minimize the reconstruction error of the holdout data.

for a machine learning project? Below we take a look at several properties of good data.

3.3.1 Good Data Is Informative

Good data contains enough information that can be used for modeling. For example, if you want to train a model that predicts whether the customer will buy a specific product, you will need to possess both the properties of the product in question and the properties of the products customers purchased in the past. If you only have the properties of the product and a customer's location and name, then the predictions will be the same for all users from the same location.

If you have enough training examples, then the model can potentially derive the gender and ethnicity from the name and make different predictions for men, women, locations, and ethnicities, but not to each customer individually.

3.3.2 Good Data Has Good Coverage

Good data has good coverage of what you want to do with the model. For example, if you're going to use the model to classify web pages by topic and you have a thousand topics of interest, then your data has to contain examples of documents on each of the thousand topics in quantity sufficient for the algorithm to be able to learn the difference between topics.

Imagine a different situation. Let's say that for a particular topic, you only have one or a couple of documents. Let each document contain a unique ID in the text. In such a scenario, the learning algorithm will not be sure what it must look at in each document to understand to which topic it belongs. Maybe the IDs? They look like good differentiators. If the algorithm decides to use IDs to separate these couple examples from the rest of the dataset, then the learned model will not be able to generalize: it will not see any of those IDs ever again.

3.3.3 Good Data Reflects Real Inputs

Good data reflects real inputs that the model will see in production. For example, if you build a system that recognizes cars on the road and all pictures you have were taken during the working hours, then it's unlikely that you will have many examples of night pictures. Once you deploy the model in production, pictures will start coming from all times of the day, and your model will more frequently make errors on night pictures. Also, remember the problem of a cat, a dog, and a raccoon: if your model doesn't know anything about raccoons, it will predict their pictures as either dogs or cats.

3.3.4 Good Data Is Unbiased

Good data is as unbiased as possible. This property can look similar to the previous one. Still, bias can be present in both the data you use for training and the data that the model is applied to in the production environment.

We discussed several sources of bias in data and how to deal with it in Section 3.2. A user interface can also be a source of bias. For example, you want to predict the popularity of a news article, and use the click rate as a feature. If some news article was displayed on the top of the page, the number of clicks it got would often be higher compared to another news article displayed on the bottom, even if the latter is more engaging.

3.3.5 Good Data Is Not a Result of a Feedback Loop

Good data is not a result of the model itself. This echoes the problem of the **feedback loop** discussed above. For example, you cannot train a model that predicts the gender of a person from their name, and then use the prediction to label a new training example.

Alternatively, if you use the model to decide which email messages are important to the user and highlight those important messages, you should not directly take the clicks on those emails as a signal that the email is important. The user might have clicked on them because the model highlighted them.

3.3.6 Good Data Has Consistent Labels

Good data has consistent labels. Inconsistency in labeling can come from several sources:

- Different people do labeling according to different criteria. Even if people believe that they use the same criteria, different people often interpret the same criteria differently.[6]
- The definition of some classes evolved over time. This results in a situation when two very similar feature vectors receive two different labels.
- Misinterpretation of user's motives. For example, assume that the user ignored a recommended news article. As a consequence, this news article receives a negative label. However, the motive of the user for ignoring this recommendation might be that they already knew the story and not that they are uninterested in the topic of the story.

3.3.7 Good Data Is Big Enough

Good data is big enough to allow generalization. Sometimes, nothing can be done to increase the accuracy of the model. No matter how much data you throw on the learning algorithm:

[6] Recall the example of Mechanical Turk we considered in Section 3.1. To improve the reliability of labels assigned by different people, one can use a majority vote (or an average) of several labelers.

the information contained in the data has low predictive power for your problem. However, more often, you can get a very accurate model if you pass from thousands of examples to millions or hundreds of millions. You cannot know how much data you need before you start working on your problem and see the progress.

3.3.8 Summary of Good Data

For the convenience of future reference, let me once again repeat the properties of good data:

- it contains enough information that can be used for modeling,
- it has good coverage of what you want to do with the model,
- it reflects real inputs that the model will see in production,
- it is as unbiased as possible,
- it is not a result of the model itself,
- it has consistent labels, and
- it is big enough to allow generalization.

3.4 Dealing With Interaction Data

Interaction data is the data you can collect from user interactions with the system your model supports. You are considered lucky if you can gather good data from interactions of the user with the system.

Good interaction data contains information on three aspects:

- context of interaction,
- action of the user in that context, and
- outcome of interaction.

As an example, assume that you build a search engine, and your model reranks search results for each user individually. A reranking model takes as input the list of links returned by the search engine, based on keywords provided by the user and outputs another list in which the items change order. Usually, a reranked model "knows" something about the user and their preferences and can reorder the generic search results for each user individually according to that user's learned preferences. The context here is the search query and the hundred documents presented to the user in a specific order. The action is a click of the user on a particular document link. The outcome is how much time the user spent reading the document and whether the user hit "back." Another action is the click on the "next page" link.

The intuition is that the ranking was good if the user clicked on some link and spent significant time reading the page. The ranking was not so good if the user clicked on a link to a result and then hit "back" quickly. The ranking was bad if the user clicked on the "next page" link. This data can be used to improve the ranking algorithm and make it more personalized.

Country	Population	Region	...	GDP per capita	GDP
France	67M	Europe	...	38,800	2.6T
Germany	83M	Europe	...	44,578	3.7T
...	...	...	...	...	...
China	1386M	Asia	...	8,802	12.2T

Figure 3.9: An example of the target (GDP) being a simple function of two features: Population and GDP per capita.

3.5 Causes of Data Leakage

Let's discuss the three most frequent causes of **data leakage** that can happen during data collection and preparation: 1) target being a function of a feature, 2) feature hiding the target, and 3) feature coming from the future.

3.5.1 Target is a Function of a Feature

Gross Domestic Product (GDP) is defined as the monetary measure of all finished goods and services in a country within a specific period. Let our goal be to predict a country's GDP based on various attributes: area, population, geographic region, and so on. An example of such data is shown in Figure 3.9. If you don't do a careful analysis of each attribute and its relation to GDP, you might let a leakage happen: in the data in Figure 3.9, two columns, Population and GDP per capita, multiplied, equal GDP. The model you will train will perfectly predict GDP by looking at these two columns only. The fact that you let GDP be one of the features, though in a slightly modified form (devised by the population), constitutes contamination and, therefore, leads to data leakage.

A simpler example is when you have a copy of the target among features, just put in a different format. Imagine you train a model to predict the yearly salary, given the attributes of an employee. The training data is a table that contains both monthly and yearly salary, among many other attributes. If you forget to remove the monthly salary from the list of features, that attribute alone will perfectly predict the yearly salary, making you believe your model is perfect. Once the model is put in production, it will likely stop receiving information about a person's monthly salary: otherwise, the modeling would not be needed.

Customer ID	Group	Yearly Spendings	Yearly Pageviews	...	Gender
1	M18-25	1350	11,987	...	M
2	F25-35	2365	8,543	...	F
...	...	...	...	...	...
18879	F65+	3653	6,775	...	F

Figure 3.10: An example of the target being hidden in one of the features.

3.5.2 Feature Hides the Target

Sometimes the target is not a function of one or more features, but rather is "hidden" in one of the features. Consider the dataset in Figure 3.10.

In this scenario, you use customer data to predict their gender. Look at the column Group. If you closely investigate the data in the column Group, you will see that it represents a demographic value to which each existing customer was related in the past. If the data about a customer's gender and age is factual (as opposed to being guessed by another model that might be available in production), then the column Group constitutes a form of data leakage, when the value you want to predict is "hidden" in the value of a feature.

On the other hand, if the Group values are predictions provided by another, possibly less accurate model, then you can use this attribute to build a potentially stronger model. This is called **model stacking**, and we will consider this topic in Section **??** in Chapter 6.

3.5.3 Feature From the Future

Feature from the future is a kind of data leakage that is hard to catch if you don't have a clear understanding of the business goal. Imagine a client asked you to train a model that predicts whether a borrower will pay back the loan, based on attributes such as age, gender, education, salary, marital status, and so on. An example of such data is shown in Figure 3.11.

If you don't make an effort to understand the business context in which your model will be used, you might decide to use all available attributes to predict the value in the column Will Pay Loan, including the data from the column Late Payment Reminders. Your model will look accurate at testing time and you send it to the client, who will later report that the model doesn't work well in the production environment.

After investigation, you find out that, in the production environment, the value of Late Payment Reminders is always zero. This makes sense because the client uses your model before the borrower gets the credit, so no reminders have yet been made! However, your

Borrower ID	Demographic Group	Education	...	Late Payment Reminders	Will Pay Loan
1	M35-50	High school	...	0	Y
2	F25-35	Master's	...	1	N
...	...	...	...	...	...
65723	M25-35	Master's	...	3	N

Figure 3.11: A feature unavailable at the prediction time: Late Payment Reminders.

model most likely learned to make the "No" prediction when Late Payment Reminders is 1 or more and pays less attention to the other features.

Here is another example. Let's say you have a news website and you want to predict the ranking of news you serve to the user, so as to maximize the number of clicks on stories. If in your training data, you have positional features for each news item served in the past (e.g., the $x - y$ position of the title, and the abstract block on the webpage), such information will not be available on the serving time, because you don't know the positions of articles on the page before you rank them.

Understanding the business context in which the model will be used is, thus, crucial to avoid data leakage.

3.6 Data Partitioning

As discussed in Section 1.3.3 of the first chapter, in practical machine learning, we typically use three disjoint sets of examples: training set, validation set, and test set.

Figure 3.12: The entire dataset partitioned into a training, validation and test sets.

The **training set** is used by the machine learning algorithm to train the model.

The **validation set** is needed to find the best values for the hyperparameters of the machine learning pipeline. The analyst tries different combinations of hyperparameter values one by one, trains a model by using each combination, and notes the model performance on the validation set. The hyperparameters that maximize the model performance are then used to train the model for production. We consider techniques of hyperparameter tuning in more detail in Section **??** of Chapter 5.

The **test set** is used for reporting: once you have your best model, you test its performance on the test set and report the results.

Validation and test sets are often referred to as **holdout sets**: they contain the examples that the learning algorithm is not allowed to see.

To obtain good partitions of your entire dataset into these three disjoint sets, as schematically illustrated in Figure 3.12, partitioning has to satisfy several conditions.

Condition 1: Split was applied to raw data.
Once you have access to raw examples, and before everything else, do the split. This will allow avoiding data leakage, as we will see later.

Condition 2: Data was randomized before the split.
Randomly shuffle your examples first, then do the split.

Condition 3: Validation and test sets follow the same distribution.
When you select the best values of hyperparameters using the validation set, you want that this selection yields a model that works well in production. The examples in the test set are your best representatives of the production data. Hence the need for the validation and test sets to follow the same distribution.

Condition 4: Leakage during the split was avoided.
Data leakage can happen even during the data partitioning. Below, we will see what forms of leakage can happen at that stage.

There is no ideal ratio for the split. In older literature (pre-big data), you might find the recommended splits of either $70\%/15\%/15\%$ or $80\%/10\%/10\%$ (for training, validation, and test sets, respectively, in proportion to the entire dataset).

Today, in the era of the Internet and cheap labor (e.g., Mechanical Turk or crowdsourcing), organizations, scientists, and even enthusiasts at home can get access to millions of training examples. That makes it wasteful only to use 70% or 80% of the available data for training.

The validation and test data are only used to calculate statistics reflecting the performance of the model. Those two sets just need to be large enough to provide reliable statistics. How much is debatable. As a rule of thumb, having a dozen examples per class is a desirable minimum. If you can have a hundred examples per class in each of the two holdout sets, you have a solid setup and the statistics calculated based on such sets are reliable.

The percentage of the split can also be dependent on the chosen machine learning algorithm or model. Deep learning models tend to significantly improve when exposed to more training data. This is less true for shallow algorithms and models.

Your proportions may depend on the size of the dataset. A small dataset of less than a thousand examples would do best with 90% of the data used for training. In this case, you might decide to not have a distinct validation set, and instead simulate with the **cross-validation** technique. We will talk more about that in Section **??** in Chapter 5.

It's worth mentioning that when you split **time-series data** into the three datasets, you must execute the split so that the order of observations in each example is preserved during the shuffling. Otherwise, for most predictive problems, your data will be broken, and no learning will be possible. We talk more about time series in Section **??** in Chapter 4.

3.6.1 Leakage During Partitioning

As you already know, data leakage may happen at any stage, from data collection to model evaluation. The data partitioning stage is no exception.

Group leakage may occur during partitioning. Imagine you have magnetic resonance images of the brains of multiple patients. Each image is labeled with certain brain disease, and the same patient may be represented by several images taken at different times. If you apply the partitioning technique discussed above (shuffle, then split), images of the same patient might appear in both the training and holdout data.

The model might learn from the particularities of the patient rather than the disease. The model would remember that patient A's brain has specific brain convolutions, and if they have a specific disease in the training data, the model successfully predicts this disease in the validation data by recognizing patient A from just the brain convolutions.

The solution to group leakage is **group partitioning**. It consists of keeping all patient examples together in one set: either training or holdout. Once again, you can see how important it is for the data analyst to know as much as possible about the data.

3.7 Dealing with Missing Attributes

Sometimes, the data comes to the analyst in a tidy form, such as an Excel spreadsheet,[7] but you might find some attributes missing. This often happens when the dataset was handcrafted, and the person forgot to fill some values or didn't get them measured.

The list of typical approaches of dealing with missing values for an attribute include:

[7] The fact that your raw dataset is contained in an Excel spreadsheet doesn't guarantee that the data is tidy. One property of tidiness is that each row represents one example.

- removing the examples with missing attributes from the dataset (this can be done if your dataset is big enough to safely sacrifice some data);
- using a learning algorithm that can deal with missing attribute values (such as the decision tree learning algorithm);
- using a **data imputation** technique.

3.7.1 Data Imputation Techniques

To impute the value of a missing numerical attribute, one technique consists in replacing the missing value by the average value of this attribute in the rest of the dataset. Mathematically it looks as follows. Let j be an attribute that is missing in some examples in the original dataset, and let $\mathcal{S}^{(j)}$ be the set of size $N^{(j)}$ that contains only those examples from the original dataset in which the value of the attribute j is present. Then the missing value $\hat{x}^{(j)}$ of the attribute j is given by,

$$\hat{x}^{(j)} \leftarrow \frac{1}{N^{(j)}} \sum_{i \in \mathcal{S}^{(j)}} x_i^{(j)},$$

where $N^{(j)} < N$ and the summation is made only over those examples where the value of the attribute j is present. An illustration of this technique is given in Figure 3.13, where two examples (at row 1 and 3) have the Height attribute missing. The average value, 177, will be imputed in the empty cells.

Row	Age	Weight	Height	Salary
1	18	70		35,000
2	43	65	175	26,900
3	34	87		76,500
4	21	66	187	94,800
5	65	60	169	19,000

$$\hat{Height} \leftarrow \frac{1}{3}(175 + 187 + 169) = 177$$

Figure 3.13: Replacing the missing value by an average value of this attribute in the dataset.

Another technique is to replace the missing value with a value outside the normal range of values. For example, if the regular range is $[0, 1]$, you can set the missing value to 2 or -1; if the attribute is categorical, such as days of the week, then a missing value can be replaced by the value "Unknown." Here, the learning algorithm learns what to do when the attribute has a value different from regular values. If the attribute is numerical, another technique is

replacing the missing value with a value in the middle of the range. For example, if the range for an attribute is $[-1, 1]$, you can set the missing value to be equal to 0. Here, the idea is that the value in the middle of the range will not significantly affect the prediction.

A more advanced technique is to use the missing value as the target variable for a regression problem. (In this case, we assume all attributes are numerical.) You can use the remaining attributes $[x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(j-1)}, x_i^{(j+1)}, \ldots, x_i^{(D)}]$ to form a feature vector $\hat{\mathbf{x}}_i$, set $\hat{y}_i \leftarrow x_i^{(j)}$, where j is the attribute with a missing value. Then you build a regression model to predict $\hat{y}$ from $\hat{\mathbf{x}}$. Of course, to build training examples $(\hat{\mathbf{x}}, \hat{y})$, you only use those examples from the original dataset, in which the value of attribute j is present.

Finally, if you have a significantly large dataset and just a few attributes with missing values, you can add a synthetic binary indicator attribute for each original attribute with missing values. Let's say that examples in your dataset are D-dimensional, and attribute at position $j = 12$ has missing values. For each example $\mathbf{x}$, you then add the attribute at position $j = D + 1$, which is equal to 1 if the value of the attribute at position 12 is present in $\mathbf{x}$ and 0 otherwise. The missing value then can be replaced by 0 or any value of your choice.

At prediction time, if your example is not complete, you should use the same data imputation technique to fill the missing values as the technique you used to complete the training data.

Before you start working on the learning problem, you cannot tell which data imputation technique will work best. Try several techniques, build several models, and select the one that works best (using the validation set to compare models).

3.7.2 Leakage During Imputation

If you use the imputation techniques that compute some statistic of one attribute (such as average) or several attributes (by solving the regression problem), the leakage happens if you use the whole dataset to compute this statistic. Using all available examples, you contaminate the training data with information obtained from the validation and test examples.

This type of leakage is not as significant as other types discussed earlier. However, you still have to be aware of it and avoid it by partitioning first, and then computing the imputation statistic only on the training set.

3.8 Data Augmentation

For some types of data, it's quite easy to get more labeled examples without additional labeling. The strategy is called **data augmentation**, and it's most effective when applied to images. It consists of applying simple operations, such as crop or flip, to the original images to obtain new images.

Figure 3.14: Examples of data augmentation techniques. Photo credit: Alfonso Escalante.

3.8.1 Data Augmentation for Images

In Figure 3.14, you can see examples of operations that can be easily applied to a given image to obtain one or more new images: flip, rotation, crop, color shift, noise addition, perspective change, contrast change, and information loss.

Flipping, of course, has to be done only with respect to the axis for which the meaning of the image is preserved. If it's a football, you can flip with respect to both axes,[8] but if it's a car or a pedestrian, then you should only flip with respect to the vertical axis.

Rotation should be applied with slight angles to simulate an incorrect horizon calibration. You can rotate an image in both directions.

Crops can be randomly applied multiple times to the same image by keeping a significant part of the object(s) of interest in the cropped images.

[8]Unless the context, like grass, makes flipping according to the horizontal axis irrelevant.

In color shift, nuances of red-green-blue (RGB) are slightly changed to simulate different lighting conditions. Contrast change (both decreasing and increasing) and **Gaussian noise** of different intensity can also be applied multiple times to the same image.

By randomly removing parts of an image, we can simulate situations when an object is recognizable but not entirely visible because of an obstacle.

Another popular technique of data augmentation that seems counterintuitive, but works very well in practice, is **mixup**. As the name suggests, the technique consists of training the model on a mix of the images from the training set. More precisely, instead of training the model on the raw images, we take two images (that could be of the same class or not) and use for training their linear combination:

$$\text{mixup\_image} = t \times \text{image}_1 + (1 - t) \times \text{image}_2,$$

where t is a real number between 0 and 1. The target of that mixup image is a combination of the original targets obtained using the same value of t:

$$\text{mixup\_target} = t \times \text{target}_1 + (1 - t) \times \text{target}_2.$$

Experiments[9] on the **ImageNet-2012**, **CIFAR-10**, and several other datasets showed that mixup improves the generalization of neural network models. The authors of the mixup also found that it increases the robustness to **adversarial examples** and stabilizes the training of **generative adversarial networks** (GANs).

In addition to the techniques shown in Figure 3.14, if you expect the input images in your production system will come overcompressed, you can simulate overcompression by using some frequently used lossy compression methods and file formats, such as JPEG or GIF.

Only training data undergoes augmentation. Of course, it's impractical to generate all these additional examples in advance and store them. In practice, the data augmentation techniques are applied to the original data on-the-fly during training.

3.8.2 Data Augmentation for Text

When it comes to text data augmentations, it is not as straightforward. We need to use appropriate transformation techniques to preserve the contextual and grammatical structure of natural language texts.

One technique involves replacing random words in a sentence with their close **synonyms**. For the sentence, "The car stopped near a shopping mall." some equivalent sentences are:

> "The automobile stopped near a shopping mall."

[9] More details on the mixup technique can be found in Zhang, Hongyi, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. "mixup: Beyond empirical risk minimization." arXiv preprint arXiv:1710.09412 (2017).

"The car stopped near a shopping center."

"The auto stopped near a mall."

A similar technique uses **hypernyms** instead of synonyms. A hypernym is a word that has more general meaning. For example, "mammal" is a hypernym for "whale" and "cat"; "vehicle" is a hypernym for "car" and "bus." From our example above, we could create the following sentences:

"The vehicle stopped near a shopping mall."

"The car stopped near a building."

If you represent words or documents in your dataset using **word** or **document embeddings**, you can apply slight Gaussian noise to randomly chosen embedding features to make a variation of the same word or document. You can tune the number of features to modify and the noise intensity as hyperparameters by optimizing the performance on validation data.

Alternatively, to replace a given word w in the sentence, you can find k nearest neighbors to the word w in the word embedding space and generate k new sentences by replacing the word w with its respective neighbor. The nearest neighbors can be found using a measure such as **cosine similarity** or **Euclidean distance**. The choice of the measure and the value of k, can be tuned as hyperparameters.

A modern alternative to the k-nearest-neighbors approach described above is to use a deep pre-trained model such as Bidirectional Encoder Representations from Transformers (**BERT**). Models like BERT are trained to predict a masked word given other words in a sentence. One can use BERT to generate k most likely predictions for a masked word and then use them as synonyms for data augmentation.

Similarly, if your problem is document classification, and you have a large corpus of unlabeled documents, but only a small corpus of labeled documents, you can do as follows. First, build document embeddings for all documents in your large corpus. Use **doc2vec** or any other technique of document embedding. Then, for each labeled document d in your dataset, find k closest unlabeled documents in the document embedding space and label them with the same label as d. Again, tune k on the validation data.

Another useful text data augmentation technique is **back translation**. To create a new example from a text written in English (it can be a sentence or a document), first translate it into another language l using a machine translation system. Then translate it back from l into English. If the text obtained through back translation is different from the original text, you add it to the dataset by assigning the same label as the original text.

There are also data augmentation techniques for other data types, such as audio and video: addition of noise, shifting an audio or a video clip in time, slowing it down or accelerating, changing pitch for audio and color balance for video, to name a few. Describing these techniques in detail is out of the scope of this book. You should just be aware that data augmentation can be applied to any media data, and not only images and text.

3.9 Dealing With Imbalanced Data

Class imbalance is a condition in the data that can significantly affect the performance of the model, independently of the chosen learning algorithm. The problem is a very uneven distribution of labels in the training data.

This is the case, for example, when your classifier has to distinguish between genuine and fraudulent e-commerce transactions: the examples of genuine transactions are much more frequent. Typically, a machine learning algorithm tries to classify most training examples correctly. The algorithm is pushed to do so because it needs to minimize a cost function that typically assigns a positive loss value to each misclassified example. If the loss is the same for the misclassification of a minority class example as it is for the misclassification of a majority class, then it's very likely that the learning algorithm decides to "give up" on many minority class examples in order to make fewer mistakes in the majority class.

While there is no formal definition of **imbalanced data**, consider the following rule of thumb. If there are two classes, then balanced data would mean half of the dataset representing each class. A slight class imbalance is usually not a problem. So, if 60% examples belong to one class and 40% belong to the other, and you use a popular machine learning algorithm in its standard formulation, it should not cause any significant performance degradation. However, when the class imbalance is high, for example when 90% examples are of one class, and 10% are of the other, using the standard formulation of the learning algorithm that usually equally weights errors made in both classes may not be as effective and would need modification.

3.9.1 Oversampling

A technique used frequently to mitigate class imbalance is **oversampling**. By making multiple copies of minority class examples, it increases their weight, as illustrated in Figure 3.15a. You might also create synthetic examples by sampling feature values of several examples of the minority class and combining them to obtain a new example of that class. Two popular algorithms that oversample the minority class by creating synthetic examples: Synthetic Minority Oversampling Technique (**SMOTE**) and Adaptive Synthetic Sampling Method (**ADASYN**).

SMOTE and ADASYN work similarly in many ways. For a given example $\mathbf{x}_i$ of the minority class, they pick k nearest neighbors. Let's denote this set of k examples as $\mathcal{S}_k$. The synthetic example $\mathbf{x}_{new}$ is defined as $\mathbf{x}_i + \lambda(\mathbf{x}_{zi} - \mathbf{x}_i)$, where $\mathbf{x}_{zi}$ is an example of the minority class chosen randomly from $\mathcal{S}_k$. The interpolation hyperparameter λ is an arbitrary number in the range $[0, 1]$. (See an illustration for $\lambda = 0.5$ in Figure 3.16.)

Both SMOTE and ADASYN randomly pick among all possible $\mathbf{x}_i$ in the dataset. In ADASYN, the number of synthetic examples generated for each $\mathbf{x}_i$ is proportional to the number of examples in $\mathcal{S}_k$, which are not from the minority class. Therefore, more synthetic examples are generated in the area where the minority class examples are rare.

Figure 3.15: Undersampling (left) and oversampling (right).

3.9.2 Undersampling

An opposite approach, **undersampling**, is to remove from the training set some examples of the majority class (Figure 3.15b).

The undersampling can be done randomly; that is, the examples to remove from the majority class can be chosen at random. Alternatively, examples to withdraw from the majority class can be selected based on some property. One such property is **Tomek links**. A Tomek link exists between two examples $\mathbf{x}_i$ and $\mathbf{x}_j$ belonging to two different classes if there's no other example $\mathbf{x}_k$ in the dataset closer to either $\mathbf{x}_i$ or $\mathbf{x}_j$ than the latter two are to each other. The closeness can be defined using a metric such as **cosine similarity** or **Euclidean distance**.

In Figure 3.17, you can see how removing examples from the majority class based on Tomek links helps to establish a clear margin between examples of two classes.

Cluster-based undersampling works as follows. Decide on the number of examples you want to have in the majority class resulting from undersampling. Let that number be k. Run a **centroid-based clustering algorithm** on the majority examples only with k being the desired number of clusters. Then replace all examples in the majority classes with the k centroids. An example of a centroid-based clustering algorithm is **k-nearest neighbors**.

3.9.3 Hybrid Strategies

You can develop your hybrid strategies (by combining both over- and undersampling) and possibly get better results. One such strategy consists of using ADASYN to oversample, and then Tomek links to undersample.

Another possible strategy consists of combining cluster-based undersampling with SMOTE.

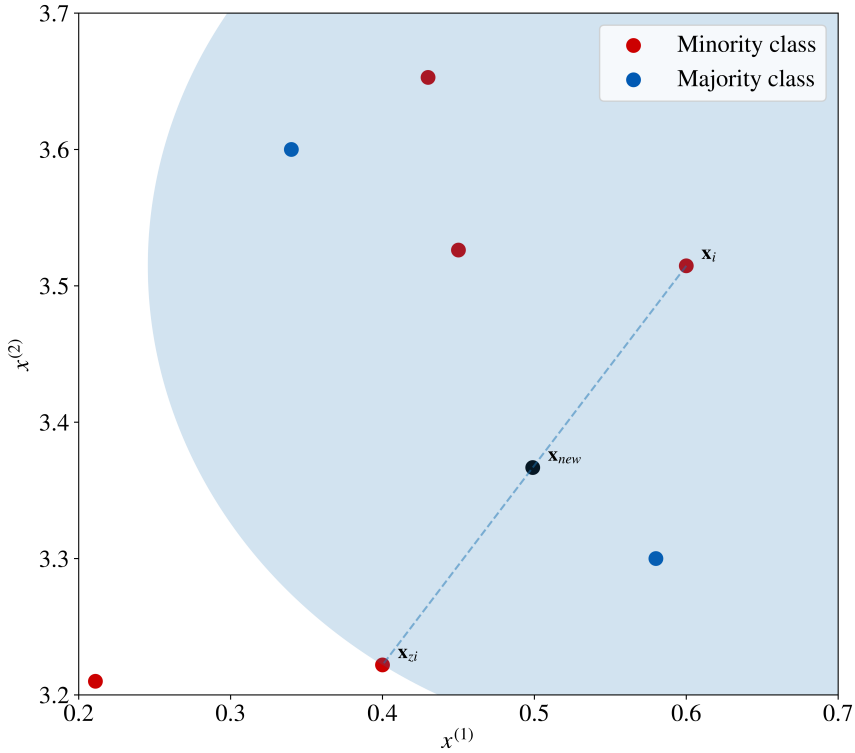


Figure 3.16: An illustration of a synthetic example generation for SMOTE and ADASYN. (Built using a script adapted from Guillaume Lemaitre.)

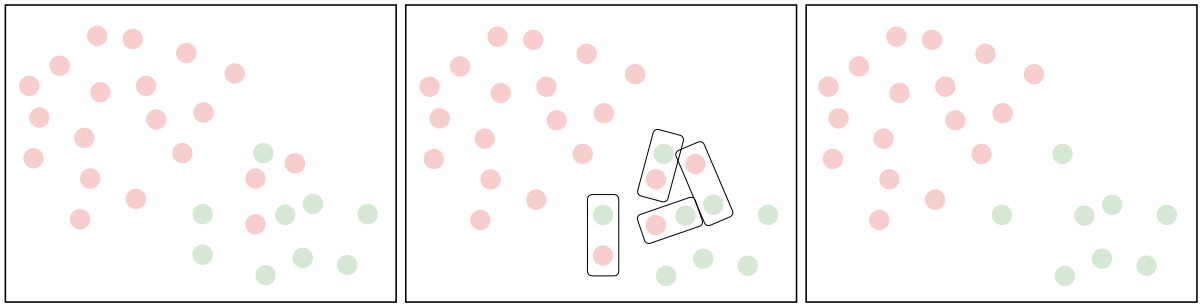
3.10 Data Sampling Strategies

When you have a large data asset, so-called big data, it's not always practical or necessary to work with the entire data asset. Instead, you can draw a smaller data sample that contains enough information for learning.

Similarly, when you undersample the majority class to adjust for data imbalance, the smaller data sample should be representative of the entire majority class. In this section, we discuss several sampling strategies, their properties, advantages, and drawbacks.

There are two main strategies: probability sampling and nonprobability sampling. In **probability sampling**, all examples have a chance to be selected. These techniques involve randomness.

Nonprobability sampling is not random. To build a sample, it follows a fixed deterministic sequence of heuristic actions. This means that some examples don't have a chance of being



(a) original data (b) Tomek links (c) undersampled data

Figure 3.17: Undersampling with Tomek links.

selected, no matter how many samples you build.

Historically, nonprobability methods were more manageable for a human to execute manually. However, nowadays this advantage is not significant. Data analysts use computers and software that greatly simplify sampling, even from big data. The main drawback of nonprobability sampling methods is that they include non-representative samples and might systematically exclude important examples. These drawbacks outweigh the possible advantages of nonprobability sampling methods. Therefore, in this book I will only present probability sampling methods.

3.10.1 Simple Random Sampling

Simple random sampling is the most straightforward method, and the one I refer to when I say "sample randomly." Here, each example from the entire dataset is chosen purely by chance; each example has an equal chance of being selected.

One way of obtaining a simple random sample is to assign a number to each example, and then use a random number generator to decide which examples to select. For example, if your entire dataset contains 1000 examples, tagged from 0 to 999, use groups of three digits from the random number generator to select an example. So, if the first three numbers from the random number generator were 0, 5, and 7, choose the example numbered 57, and so on.

Simplicity is the great advantage of this sampling method, and it can be easily implemented as any programming language can serve as a random number generator. A disadvantage of simple random sampling is that you may not select enough examples that would have a particular property of interest. Consider the situation where you extract a sample from a large imbalanced dataset. In so doing, you accidentally fail to capture a sufficient number of examples from the minority class - or any at all.

3.10.2 Systematic Sampling

To implement **systematic sampling** (also known as **interval sampling**), you create a list containing all examples. From that list, you randomly select the first example $\mathbf{x}_{start}$ from the first k elements on the list. Then, you select every k^{th} item on the list starting from $\mathbf{x}_{start}$. You choose such a value of k that will give you a sample of the desired size.

An advantage of the systematic sampling over the simple random sampling is that it draws examples from the whole range of values. However, systematic sampling is inappropriate if the list of examples has periodicity or repetitive patterns. In the latter case, the obtained sample can exhibit a bias. However, if the list of examples is randomized, then systematic sampling often results in a better sample than simple random sampling.

3.10.3 Stratified Sampling

If you know about the existence of several groups (e.g., gender, location, or age) in your data, you should have examples from each of those groups in your sample. In **stratified sampling**, you first divide your dataset into groups (called strata) and then randomly select examples from each stratum, like in simple random sampling. The number of examples to select from each stratum is proportional to the size of the stratum.

Stratified sampling often improves the representativeness of the sample by reducing its bias; in the worst of cases, the resulting sample is of no less quality than the results of simple random sampling. However, to define strata, the analyst has to understand the properties of the dataset. Furthermore, it can be difficult to decide which attributes will define the strata.

If you don't know how to best define the strata, you can use **clustering**. The only decision you have to make is how many clusters you need. This technique is also useful to choose the unlabeled examples to send for labeling to a human labeler. It often happens that we have millions of unlabeled examples, and few resources available for labeling. Choose examples carefully, so that each stratum or cluster is represented in our labeled data.

Stratified sampling is the slowest of the three methods due to the additional overhead of working with several independent strata. However, its potential benefit of producing a less biased sample typically outweighs its drawbacks.

3.11 Storing Data

Keeping data safe is insurance for your organization's business: if you lose a business-critical model for any reason, such as a disaster or human mistake (the file with the model was accidentally erased or overwritten), having the data will allow you to rebuild that model easily.

When sensitive data or personally identifiable information (PII) is provided by customers or business partners, it must be stored in not just a safe but also a secure location. Jointly with

the DBA or DevOps engineers, access to sensitive data can be restricted by username and, if needed, IP address. Access to a relational database might also be limited on the per row and per column basis.

It's also recommended to limit access to read-only and add-only operations, by restricting write and erase operations to specific users.

If the data is collected on mobile devices, it might be necessary to store it on the mobile device until the owner connects to wifi. This data might need to be encrypted so that other applications cannot access it. Once the user is connected to wifi, the data has to be synchronized with a secure server by using cryptographic protocols, such as Transport Layer Security (TLS). Each data element on the mobile device has to be marked with a timestamp to allow its proper synchronization with the data on the server.

3.11.1 Data Formats

Data for machine learning can be stored in various formats. Data used indirectly, such as dictionaries or gazetteers, may be stored as a table in a relational database, a collection in a key-value store, or a structured text file.

The tidy data is usually stored as comma-separated values (CSV) or tab-separated values (TSV) files. In this case, all examples are stored in one file. Alternatively, collection of XML (Extensible Markup Language) files or JSON (JavaScript Object Notation) files can contain one example per file.

In addition to general-purpose formats, certain popular machine learning packages use proprietary data formats to store tidy data. Other machine learning packages often provide application programming interfaces (APIs) to one or several such proprietary data formats. The most frequently supported formats are **ARFF** (Attribute-Relation File Format used in the Weka machine learning package) and the **LIBSVM** (Library for Support Vector Machines) format, which is the default format used by the LIBSVM and **LIBLINEAR** (Library for Large Linear Classification) machine learning libraries.

The data in the LIBSVM format consists of one file containing all examples. Each line of that file represents a labeled feature vector using the following format:

```
label index1:value1 index2:value2 ...
```

where index$X$:value$Y$ specifies the value Y of the feature at position (dimension) X. If the value at some position is zero, it can be omitted. This data format is especially convenient for **sparse data** consisting of examples in which the values of most features are zero.

Furthermore, different programming languages come with **data serialization** capabilities. The data for a specific machine learning package can be persisted on the hard drive using a serialization object or function provided by the programming language or library. When

needed, the data can be deserialized in its original form. For example, in Python, a popular general-purpose serialization module is **Pickle**; R has built-in `saveRDS` and `readRDS` functions. Different data analysis packages can also offer their own serialization/deserialization tools.

In Java, any object that implements the `java.io.Serializable` interface can be serialized into a file and then deserialized when needed.

3.11.2 Data Storage Levels

Before deciding how and where to store the data, it's essential to choose the appropriate **storage level**. Storage can be organized in different levels of abstraction: from the lowest level, the filesystem, to the highest level, such as data lake.

Filesystem is the foundational level of storage. The fundamental unit of data on that level is a **file**. A file can be text or binary, is not versioned, and can be easily erased or overwritten.

A filesystem can be local or networked. A networked filesystem can be simple or distributed.

A **local filesystem** can be as simple as a locally mounted disk containing all the files needed for your machine learning project.

A **distributed filesystem**, such as **NFS** (Network File System), **CephFS** (Ceph File System) or **HDFS**, can be accessed over the network by multiple physical or virtual machines. Files in a distributed filesystem are stored and accessed over multiple machines in the network.

Despite its simplicity, filesystem-level storage is appropriate for a many use cases, including:

File sharing
: The simplicity of filesystem-level storage and support for standard protocols allows you to store and share data with a small group of colleagues with minimal effort.

Local archiving
: Filesystem-level storage is a cost-effective option for archiving data, thanks to the availability and accessibility of scale-out NAS solutions.

Data protection
: Filesystem-level storage is a viable data protection solution thanks to built-in redundancy and replication.

Parallel access to the data on the filesystem level is fast for retrieval access but slow for storage, so it's an appropriate storage level for smaller teams and data.

Object storage is an application programming interface (API) defined over a filesystem. Using an API, you can programmatically execute such operations on files as GET, PUT, or DELETE without worrying where the files are actually stored. The API is typically provided by an **API service** available on the network and accessible by **HTTP** or, more generally, **TCP/IP** or a different communication protocol suite.

The fundamental unit of data in an object storage level is an **object**. Objects are usually binary: images, sound, or video files, and other data elements having a particular format.

Such features as versioning and redundancy can be built into the API service. The access to the data stored on the object storage level can often be done in parallel, but the access is not as fast as on the filesystem level.

Canonical examples of object storage are **Amazon S3** and **Google Cloud Storage** (GCS). Alternatively, **Ceph** is a storage platform that implements object storage on a single distributed computer cluster and provides interfaces for both object- and filesystem-level storage. It's often used as an alternative to S3 and GCS in on-premises computing systems.

The **database** level of data storage allows persistent, fast, and scalable storage of **structured data** with fast parallel access for both storage and retrieval.

A modern database management system (**DBMS**) stores data in random-access memory (**RAM**), but software ensures that data is persisted (and operations on data are logged) to disk and never lost.

The fundamental unit of data at this level is a **row**. A row has a unique ID and contains values in columns. In a relational database, rows are organized in **tables**. Rows can have references to other rows in the same or different tables.

Databases are not exceptionally well suited for storing binary data, though rather small binary objects can sometimes be stored in a column in the form of a **blob** (for Binary Large OBject). Blob is a collection of binary data stored as a single entity. More often, though, a row stores references to binary objects stored elsewhere — in a filesystem or object storage.

The four most frequently used DBMS in the industry are Oracle, MySQL, Microsoft SQL Server, and PostgresSQL. They all support SQL (Structured Query Language), an interface for accessing and modifying data stored in the databases, as well as creating, modifying, and erasing databases.[10]

A **data lake** is a repository of data stored in its natural or raw format, usually in the form of object blobs or files. A data lake is typically an unstructured aggregation of data from multiple sources, including databases, logs, or intermediary data obtained as a result of expensive transformations of the original data.

The data is saved in the data lake in its raw format, including the structured data. To read data from a data lake, the analyst needs to write the programming code that reads and parses the data stored in a file or a blob. Writing a script to parse the data file or a blob is an approach called **schema on read**, as opposed to the **schema on write** in DBMS. In a DBMS, the schema of data is defined beforehand, and, at each write, the DBMS makes sure that the data corresponds to the schema.

[10]The SQL Server uses its proprietary Transact SQL (T-SQL) while Oracle uses Procedural Language SQL (PL/SQL).

3.11.3 Data Versioning

If data is held and updated in multiple places, you might need to keep track of versions. Versioning the data is also needed if you frequently update the model by collecting more data, especially in an automated way. This happens when you work on automated driving, spam detection, or personalized recommendations, for example. The new data comes from a human driving a car, or the user cleaning up their electronic mail, or recent video streaming. Sometimes, after an update of the data, the new model performs worse, and you would like to investigate why by switching from one version of the data to another.

Data versioning is also critical in supervised learning when the labeling is done by multiple labelers. Some labelers might assign very different labels to similar examples, which typically hurts the performance of the model. You would like to keep the examples annotated by different labelers separately and only merge them when you build the model. Careful analysis of the model performance may show that labelers didn't provide quality or consistent labels. Exclude such data from the training data, or relabel it, and data versioning will allow this with minimal effort.

Data versioning can be implemented in several levels of complexity, from the most basic to the most elaborate.

Level 0: data is unversioned.
: At this level, data may reside on a local filesystem, object storage, or in a database. The advantage of having unversioned data is the speed and simplicity of dealing with the data. Still, that advantage is outweighed by potential problems you might encounter when working on your model. Most likely, your first problem will be the inability to make versioned deployments. As we will discuss in Chapter 8, model deployments must be versioned. A deployed machine learning model is a mix of code and data. If the code is versioned, the data must be too. Otherwise, the deployment will be unversioned.

  If you don't version deployments, you will not be able to get back to the previous level of performance in case of any problem with the model. Therefore, unversioned data is not recommended.

Level 1: data is versioned as a snapshot at training time.
: At this level, data is versioned by storing, at training time, a snapshot of everything needed to train a model. Such an approach allows you to version deployed models and get back to past performance. You should keep track of each version in some document, typically an Excel spreadsheet. That document should describe the location of the snapshot of both code and data, hyperparameter values, and other metadata needed to reproduce the experiment if needed. If you don't have many models and don't update them too frequently, this level of versioning could be a viable strategy. Otherwise, it's not recommended.

Level 2: both data and code are versioned as one asset.
: At this level of versioning, small data assets, such as dictionaries, gazetteers, and small

datasets, are stored jointly with the code in a version control system, such as **Git** or **Mercurial**. Large files are stored in object storage, such as **S3** or **GCS**, with unique IDs. The training data is stored as JSON, XML, or another standard format, and includes relevant metadata such as labels, the identity of the labeler, time of labeling, the tool used to label the data, and so on.

Tools like **Git Large File Storage** (LFS) automatically replace large files such as audio samples, videos, large datasets, and graphics with text pointers, inside Git, while storing the file contents on a remote server.

The version of the dataset is defined by the **git signatures** of the code and the data file. It can also be helpful to add a timestamp to identify a needed version easily.

Level 3: using or building a specialized data versioning solution.
Data versioning software such as **DVC** and **Pachyderm** provide additional tools for data versioning. They typically interoperate with code versioning software, such as Git.

Level 2 of versioning is a recommended way of implementing versioning for most projects. If you feel like Level 2 is not sufficient for your needs, explore Level 3 solutions, or consider building your own. Otherwise, that approach is not recommended, as it adds complexity to what is already a complex engineering project.

3.11.4 Documentation and Metadata

While you are actively working on a machine learning project, you are often capable of remembering important details about the data. However, once the project goes to production and you switch to another project, this information will eventually become less detailed.

Before you switch to another project, you should make sure that others can understand your data and use it properly.

If the data is self-explanatory, then you might leave it undocumented. However, it's rather rare that someone who didn't create a dataset can easily understand it and know how to use it just by looking at it.

Documentation has to accompany any data asset that was used to train a model. This documentation has to contain the following details:

- what that data means,
- how it was collected, or methods used to create it (instructions to labelers and methods for quality control),
- the details of train-validation-test splits,
- details of all pre-processing steps,
- an explanation of any data that were excluded,
- what format is used to store the data,
- types of attributes or features (which values are allowed for each attribute or feature),

- number of examples,
- possible values for labels or the allowable range for a numerical target.

3.11.5 Data Lifecycle

Some data can be stored indefinitely. However, in some business contexts, you might be allowed to store some data for a specific time, and then you might have to erase it. If such restrictions apply to the data you work with, you have to make sure that a reliable alerting system is in place. That alerting system has to contact the person responsible for the data erasure and have a backup plan in case that person is not available. Don't forget that the consequences for not erasing data can sometimes be very serious for the organization.

For every sensitive data asset, a **data lifecycle document** has to describe the asset, the circle of persons who have access to that data asset, both during and after the project development. The document has to describe how long the data asset will be stored and whether it has to be explicitly destroyed.

3.12 Data Manipulation Best Practices

To conclude this chapter, we consider two remaining best practices: reproducibility and "data first, algorithm second."

3.12.1 Reproducibility

Reproducibility should be an important concern in everything you do, including data collection and preparation. You should avoid transforming data manually, or using powerful tools included in text editors or command line shells, such as regular expressions, "quick and dirty" ad hoc awk or sed commands, and piped expressions.

Usually, the data collection and transformation activities consist of multiple stages. These include downloading data from web APIs or databases, replacing multiword expressions by unique tokens, removing stop-words and noise, cropping and unblurring images, imputation of missing values, and so on. Each step in this multistage process has to be implemented as a software script, such as Python or R script with their inputs and outputs. If you are organized like that in your work, it will allow you to keep track of all changes in the data. If during any stage something wrong happens to the data, you can always fix the script and run the entire data processing pipeline from scratch.

On the other hand, manual interventions can be hard to reproduce. These are difficult to apply to updated data, or scale for much more data (once you can afford getting more data or a different dataset).

3.12.2 Data First, Algorithm Second

Remember that in the industry, contrary to academia, "data first, algorithm second," so focus most of your effort and time on getting more data of wide variety and high quality, instead of trying to squeeze the maximum out of a learning algorithm.

Data augmentation, when implemented well, will most likely contribute more to the quality of the model than the search for the best hyperparameter values or model architecture.

3.13 Summary

Before you start collecting the data, there are five questions to answer: is the data you will work with accessible, sizeable, useable, understandable, and reliable.

Common problems with data are high cost, bias, low predictive power, outdated examples, outliers, and leakage.

Good data contains enough information that can be used for modeling, has good coverage of what you want to do with the model, and reflects real inputs that the model will see in production. It is as unbiased as possible and not a result of the model itself, has consistent labels, and is big enough to allow generalization.

Good interaction data contains information on three aspects: context of interaction, action of the user in that context, and outcome of the interaction.

To obtain a good partition of your entire dataset into training, validation and test sets, the process of partitioning has to satisfy several conditions: 1) data was randomized before the split, 2) split was applied to raw data, 3) validation and test sets follow the same distribution, and 4) leakage was avoided.

Data imputation techniques can be used to deal with missing attributes in the data.

Data augmentation techniques are often used to get more labeled examples without additional manual labeling. The techniques usually apply to image data, but could also be applied to text and other types of perceptive data.

Class imbalance can significantly affect the performance of the model. Learning algorithms perform suboptimally when the training data suffers from class imbalance. Such techniques as over- and undersampling can help to overcome the class imbalance problem.

When you work with big data, it's not always practical and necessary to work with the entire data asset. Instead, draw a smaller sample of data that contains enough information for learning. Different data sampling strategies can be used for that, in particular simple random sampling, systematic sampling, stratified sampling, and cluster sampling.

Data can be stored in different data formats and on several data storage levels. Data versioning is a critical element in supervised learning when the labeling is done by multiple labelers.

Different labelers may provide labels of varying quality, so it's important to keep track of who created which labeled example. Data versioning can be implemented with several levels of complexity, from the most basic to the most elaborate: unversioned (level 0), versioned as a snapshot at training time (level 1), versioned as one asset containing both data and code (level 3), and versioned by using or building a specialized data versioning solution (level 4).

Level 2 is recommended for most projects.

Documentation has to accompany any data asset that was used to train a model. That documentation has to contain the following details: what that data means, how it was collected, or methods used to create it (instructions to labelers and methods for quality control), the details of train-validation-test splits and of all pre-processing steps. It must also contain an explanation of any data that were excluded, what format is used to store the data, types of attributes or features, number of examples, and possible values for labels or the allowable range for a numerical target.

For every sensitive data asset, a data lifecycle document has to describe the asset, the circle of persons who have access to that data asset both during and after the project development.