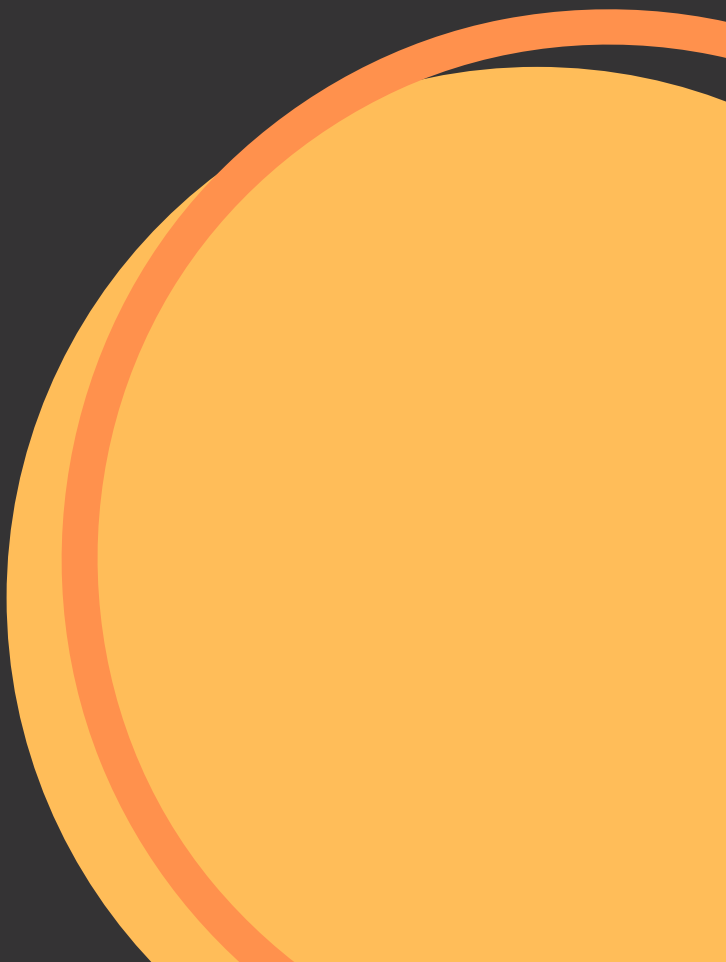
Andriy Burkov's

THE HUNDRED-PAGE MACHINE LEARNING BOOK



The Hundred-Page Machine Learning Book

The Hundred-Page Machine Learning Book

Andriy Burkov

To my parents:
Tatiana and Valeriy

and to my family:
Catherine, Eva, and Dmitriy

“All models are wrong, but some are useful.”
— George Box

The book is distributed on the “read first, buy later” principle.

Preface

Let's start by telling the truth: machines don't learn. What a typical "learning machine" does, is finding a mathematical formula, which, when applied to a collection of inputs (called "training data"), produces the desired outputs. This mathematical formula also generates the correct outputs for most other inputs (distinct from the training data) on the condition that those inputs come from the same or a similar statistical distribution as the one the training data was drawn from.

Why isn't that learning? Because if you slightly distort the inputs, the output is very likely to become completely wrong. It's not how learning in animals works. If you learned to play a video game by looking straight at the screen, you would still be a good player if someone rotates the screen slightly. A machine learning algorithm, if it was trained by "looking" straight at the screen, unless it was also trained to recognize rotation, will fail to play the game on a rotated screen.

So why the name "machine learning" then? The reason, as is often the case, is marketing: Arthur Samuel, an American pioneer in the field of computer gaming and artificial intelligence, coined the term in 1959 while at IBM. Similarly to how in the 2010s IBM tried to market the term "cognitive computing" to stand out from competition, in the 1960s, IBM used the new cool term "machine learning" to attract both clients and talented employees.

As you can see, just like artificial intelligence is not intelligence, machine learning is not learning. However, machine learning is a universally recognized term that usually refers to the science and engineering of building machines capable of doing various useful things without being explicitly programmed to do so. So, the word "learning" in the term is used by analogy with the learning in animals rather than literally.

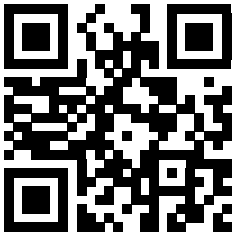
Who This Book is For

This book contains only those parts of the vast body of material on machine learning developed since the 1960s that have proven to have a significant practical value. A beginner in machine learning will find in this book just enough details to get a comfortable level of understanding of the field and start asking the right questions.

Practitioners with experience can use this book as a collection of directions for further self-improvement. The book also comes in handy when brainstorming at the beginning of a project, when you try to answer the question whether a given technical or business problem is "machine-learnable" and, if yes, which techniques you should try to solve it.

How to Use This Book

If you are about to start learning machine learning, you should read this book from the beginning to the end. (It's just a hundred pages, not a big deal.) If you are interested in a specific topic covered in the book and want to know more, most sections have a QR code.



QR Code

By scanning one of those QR codes with your phone, you will get a link to a page on the book's companion wiki theMLbook.com with additional materials: recommended reads, videos, Q&As, code snippets, tutorials, and other bonuses. The book's wiki is continuously updated with contributions from the book's author himself as well as volunteers from all over the world. So this book, like a good wine, keeps getting better after you buy it.

Scan the QR code on the left with your phone to get to the book's wiki.

Some sections don't have a QR code, but they still most likely have a wiki page. You can find it by submitting the section's title to the wiki's search engine.

Should You Buy This Book?

This book is distributed on the "read first, buy later" principle. I firmly believe that paying for the content before consuming it is buying a pig in a poke. You can see and try a car in a dealership before you buy it. You can try on a shirt or a dress in a department store. You have to be able to read a book before paying for it.

The *read first, buy later* principle implies that you can freely download the book, read it and share it with your friends and colleagues. Only if you read and liked the book, or found it helpful or useful in any way, you have to buy it.

Now you are all set. Enjoy your reading!

Contents

Chapter 1

Introduction

1.1 What is Machine Learning

Machine learning is a subfield of computer science that is concerned with building algorithms which, 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) gathering a dataset, and 2) algorithmically building 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.

1.2 Types of Learning

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

1.2.1 Supervised Learning

In **supervised learning**[1], the **dataset** is the collection of **labeled examples** $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$. Each element $\mathbf{x}_i$ among N is called a **feature vector**. A feature vector is a vector in which each dimension $j = 1, \ldots, D$ contains a value that describes the example somehow. That 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

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

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 $\mathbf{x}_k$, $k = 1, \ldots, N$. 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 see 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 $\{spam, not\_spam\}$.

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

1.2.2 Unsupervised Learning

In **unsupervised learning**, the dataset is a collection of **unlabeled examples** $\{\mathbf{x}_i\}_{i=1}^N$. 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. In **dimensionality reduction**, the output of the model is a feature vector that has fewer features than the input $\mathbf{x}$; in **outlier detection**, the output is a real number that indicates how $\mathbf{x}$ is different from a "typical" example in the dataset.

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 using many unlabeled examples can help the learning algorithm to find (we might say "produce" or "compute") a better model.

It could look counter-intuitive that learning could benefit from adding more unlabeled examples. It seems like we add more uncertainty to the problem. However, when you add unlabeled examples, you add more information about your problem: a larger sample reflects better the probability distribution the data we labeled came from. Theoretically, a learning algorithm should be able to leverage this additional information.

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

1.2.4 Reinforcement Learning

Reinforcement learning is a subfield of machine learning where the machine "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 every state. Different actions bring different **rewards** and could also move the machine to another state of the environment. The goal of a reinforcement learning algorithm is to learn a **policy**.

A 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 reward**.

Reinforcement learning solves a particular kind of problem where decision making is sequential, and the goal is long-term, such as game playing, robotics, resource management, or logistics. In this book, I put emphasis on one-shot decision making where input examples are independent of one another and the predictions made in the past. I leave reinforcement learning out of the scope of this book.

1.3 How Supervised Learning Works

In this section, I briefly explain how supervised learning works so that you have the picture of the whole process before we go into detail. I decided to use supervised learning as an example because it's the type of machine learning most frequently used in practice.

The supervised learning process starts with gathering the data. The data for supervised learning is a collection of pairs (input, output). Input could be anything, for example, email messages, pictures, or sensor measurements. Outputs are usually real numbers, or labels (e.g. "spam", "not_spam", "cat", "dog", "mouse", etc). In some cases, outputs are vectors (e.g., four coordinates of the rectangle around a person on the picture), sequences (e.g. ["adjective", "adjective", "noun"] for the input "big beautiful car"), or have some other structure.

Let's say the problem that you want to solve using supervised learning is spam detection. You gather the data, for example, 10,000 email messages, each with a label either "spam" or "not_spam" (you could add those labels manually or pay someone to do that for us). Now, you have to convert each email message into a feature vector.

The data analyst decides, based on their experience, how to convert a real-world entity, such as an email message, into a feature vector. One common way to convert a text into a feature vector, called **bag of words**, is to take a dictionary of English words (let's say it contains 20,000 alphabetically sorted words) and stipulate that in our feature vector:

- the first feature is equal to 1 if the email message contains the word "a"; otherwise, this feature is 0;

- the second feature is equal to 1 if the email message contains the word "aaron"; otherwise, this feature equals 0;
- ...
- the feature at position 20,000 is equal to 1 if the email message contains the word "zulu"; otherwise, this feature is equal to 0.

You repeat the above procedure for every email message in our collection, which gives us 10,000 feature vectors (each vector having the dimensionality of 20,000) and a label ("spam"/"not_spam").

Now you have a machine-readable input data, but the output labels are still in the form of human-readable text. Some learning algorithms require transforming labels into numbers. For example, some algorithms require numbers like 0 (to represent the label "not_spam") and 1 (to represent the label "spam"). The algorithm I use to illustrate supervised learning is called **Support Vector Machine** (SVM). This algorithm requires that the positive label (in our case it's "spam") has the numeric value of $+1$ (one), and the negative label ("not_spam") has the value of -1 (minus one).

At this point, you have a **dataset** and a **learning algorithm**, so you are ready to apply the learning algorithm to the dataset to get the **model**.

SVM sees every feature vector as a point in a high-dimensional space (in our case, space is 20,000-dimensional). The algorithm puts all feature vectors on an imaginary 20,000-dimensional plot and draws an imaginary 19,999-dimensional line (a *hyperplane*) that separates examples with positive labels from examples with negative labels. In machine learning, the boundary separating the examples of different classes is called the **decision boundary**.

The equation of the hyperplane is given by two **parameters**, a real-valued vector $\mathbf{w}$ of the same dimensionality as our input feature vector $\mathbf{x}$, and a real number b like this:

$$\mathbf{wx} - b = 0,$$

where the expression $\mathbf{wx}$ means $w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + \ldots + w^{(D)}x^{(D)}$, and D is the number of dimensions of the feature vector $\mathbf{x}$.

(If some equations aren't clear to you right now, in Chapter 2 we revisit the math and statistical concepts necessary to understand them. For the moment, try to get an intuition of what's happening here. It all becomes more clear after you read the next chapter.)

Now, the predicted label for some input feature vector $\mathbf{x}$ is given like this:

$$y = \text{sign}(\mathbf{wx} - b),$$

where sign is a mathematical operator that takes any value as input and returns $+1$ if the input is a positive number or -1 if the input is a negative number.

The goal of the learning algorithm — SVM in this case — is to leverage the dataset and find the optimal values $\mathbf{w}^*$ and b^* for parameters $\mathbf{w}$ and b. Once the learning algorithm identifies these optimal values, the **model** $f(\mathbf{x})$ is then defined as:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^*\mathbf{x} - b^*)$$

Therefore, to predict whether an email message is spam or not spam using an SVM model, you have to take a text of the message, convert it into a feature vector, then multiply this vector by $\mathbf{w}^*$, subtract b^* and take the sign of the result. This will give us the prediction ($+1$ means "spam", -1 means "not_spam").

Now, how does the machine find $\mathbf{w}^*$ and b^*? It solves an optimization problem. Machines are good at optimizing functions under constraints.

So what are the constraints we want to satisfy here? First of all, we want the model to predict the labels of our 10,000 examples correctly. Remember that each example $i = 1, \ldots, 10000$ is given by a pair $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is the feature vector of example i and y_i is its label that takes values either -1 or $+1$. So the constraints are naturally:

$$\begin{aligned} \mathbf{w}\mathbf{x}_i - b \geq +1 \quad &\text{if } y_i = +1, \\ \mathbf{w}\mathbf{x}_i - b \leq -1 \quad &\text{if } y_i = -1. \end{aligned}$$

We would also prefer that the hyperplane separates positive examples from negative ones with the largest **margin**. The margin is the distance between the closest examples of two classes, as defined by the decision boundary. A large margin contributes to a better **generalization**, that is how well the model will classify new examples in the future. To achieve that, we need to minimize the Euclidean norm of $\mathbf{w}$ denoted by $\|\mathbf{w}\|$ and given by $\sqrt{\sum_{j=1}^{D}(w^{(j)})^2}$.

So, the optimization problem that we want the machine to solve looks like this:

Minimize $\|\mathbf{w}\|$ *subject to* $y_i(\mathbf{w}\mathbf{x}_i - b) \geq 1$ *for* $i = 1, \ldots, N$. The expression $y_i(\mathbf{w}\mathbf{x}_i - b) \geq 1$ is just a compact way to write the above two constraints.

The solution of this optimization problem, given by $\mathbf{w}^*$ and b^*, is called the **statistical model**, or, simply, the **model**. The process of building the model is called **training**.

For two-dimensional feature vectors, the problem and the solution can be visualized as shown in Figure 1.1. The blue and orange circles represent, respectively, positive and negative examples, and the line given by $\mathbf{w}\mathbf{x} - b = 0$ is the decision boundary.

Why, by minimizing the norm of $\mathbf{w}$, do we find the highest margin between the two classes? Geometrically, the equations $\mathbf{w}\mathbf{x} - b = 1$ and $\mathbf{w}\mathbf{x} - b = -1$ define two parallel hyperplanes, as you see in Figure 1.1. The distance between these hyperplanes is given by $\frac{2}{\|\mathbf{w}\|}$, so the smaller the norm $\|\mathbf{w}\|$, the larger the distance between these two hyperplanes.

That's how Support Vector Machines work. This particular version of the algorithm builds the so-called *linear model*. It's called linear because the decision boundary is a straight line

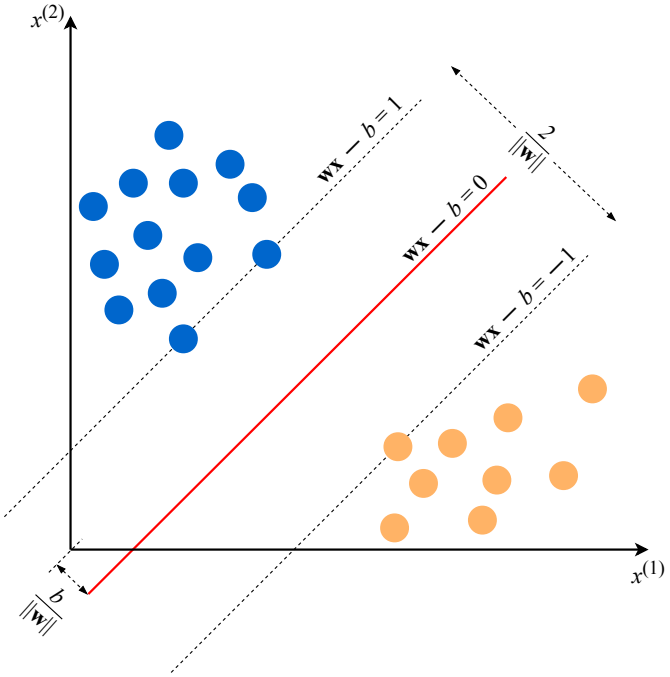


Figure 1.1: An example of an SVM model for two-dimensional feature vectors.

(or a plane, or a hyperplane). SVM can also incorporate **kernels** that can make the decision boundary arbitrarily non-linear. In some cases, it could be impossible to perfectly separate the two groups of points because of noise in the data, errors of labeling, or **outliers** (examples very different from a "typical" example in the dataset). Another version of SVM can also incorporate a penalty hyperparameter[3] for misclassification of training examples of specific classes. We study the SVM algorithm in more detail in Chapter 3.

At this point, you should retain the following: any classification learning algorithm that builds a model implicitly or explicitly creates a decision boundary. The decision boundary can be straight, or curved, or it can have a complex form, or it can be a superposition of some geometrical figures. The form of the decision boundary determines the **accuracy** of the model (that is the ratio of examples whose labels are predicted correctly). The form of the decision boundary, the way it is algorithmically or mathematically computed based on the training data, differentiates one learning algorithm from another.

In practice, there are two other essential differentiators of learning algorithms to consider: speed of model building and prediction processing time. In many practical cases, you would

[3] A hyperparameter is a property of a learning algorithm, usually (but not always) having a numerical value. That value influences the way the algorithm works. Those values aren't learned by the algorithm itself from data. They have to be set by the data analyst before running the algorithm.

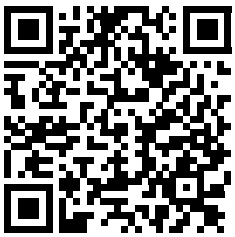
prefer a learning algorithm that builds a less accurate model fast. Additionally, you might prefer a less accurate model that is much quicker at making predictions.

1.4 Why the Model Works on New Data

Why is a machine-learned model capable of predicting correctly the labels of new, previously unseen examples? To understand that, look at the plot in Figure 1.1. If two classes are separable from one another by a decision boundary, then, obviously, examples that belong to each class are located in two different subspaces which the decision boundary creates.

If the examples used for training were selected randomly, independently of one another, and following the same procedure, then, statistically, it is *more likely* that the new negative example will be located on the plot somewhere not too far from other negative examples. The same concerns the new positive example: it will *likely* come from the surroundings of other positive examples. In such a case, our decision boundary will still, *with high probability*, separate well new positive and negative examples from one another. For other, *less likely situations*, our model will make errors, but because such situations are less likely, the number of errors will likely be smaller than the number of correct predictions.

Intuitively, the larger is the set of training examples, the more unlikely that the new examples will be dissimilar to (and lie on the plot far from) the examples used for training.



To minimize the probability of making errors on new examples, the SVM algorithm, by looking for the largest margin, explicitly tries to draw the decision boundary in such a way that it lies as far as possible from examples of both classes.

The reader interested in knowing more about the *learnability* and understanding the close relationship between the model error, the size of the training set, the form of the mathematical equation that defines the model, and the time it takes to build the model is encouraged to read about the *PAC learning*. The PAC (for "probably approximately correct") learning theory helps to analyze whether and under what conditions a learning algorithm will probably output an approximately correct classifier.

Chapter 2

Notation and Definitions

2.1 Notation

Let's start by revisiting the mathematical notation we all learned at school, but some likely forgot right after the prom.

2.1.1 Data Structures

A *scalar* 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 2.1. 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 2.1, $a^{(1)} = 2$ and $a^{(2)} = 3$.

The notation $x^{(j)}$ should not be confused with the power operator, like this x^2 (squared) or x^3 (cubed). If we want to apply a power operator, say square, 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 two columns,

$$\begin{bmatrix} 2 & 4 & -3 \\ 21 & -6 & -1 \end{bmatrix}.$$

Matrices are denoted with bold capital letters, such as $\mathbf{A}$ or $\mathbf{W}$.

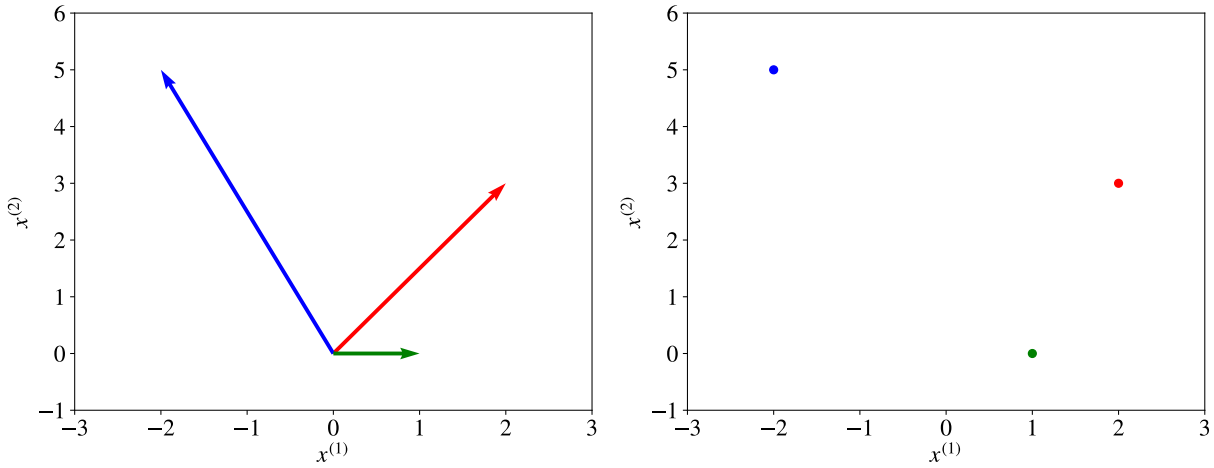


Figure 2.1: Three vectors visualized as directions and as points.

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 $\{x_1, x_2, x_3, x_4, \ldots, x_n\}$. 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 $\{1, 3, 4, 5, 8\}$.

2.1.2 Capital Sigma Notation

The summation over a collection $X = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$ 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".

2.1.3 Capital Pi Notation

A notation analogous to capital sigma is the *capital pi notation*. It denotes a product of elements in a collection or attributes of a vector:

$$\prod_{i=1}^{n} x_i \stackrel{\text{def}}{=} x_1 \cdot x_2 \cdot \ldots \cdot x_{n-1} \cdot x_n,$$

where $a \cdot b$ means a multiplied by b. Where possible, we omit $\cdot$ to simplify the notation, so ab also means a multiplied by b.

2.1.4 Operations on Sets

A derived set creation operator looks like this: $\mathcal{S}' \leftarrow \{x^2 \,|\, x \in \mathcal{S}, x > 3\}$. This notation means that we create a new set $\mathcal{S}'$ by putting into it x squared such that that x is in $\mathcal{S}$, and x is greater than 3.

The cardinality operator $|\mathcal{S}|$ returns the number of elements in set $\mathcal{S}$.

2.1.5 Operations on Vectors

The sum of two vectors $\mathbf{x} + \mathbf{z}$ is defined as the vector $[x^{(1)} + z^{(1)}, x^{(2)} + z^{(2)}, \ldots, x^{(m)} + z^{(m)}]$.

The difference of two vectors $\mathbf{x} - \mathbf{z}$ is defined as $[x^{(1)} - z^{(1)}, x^{(2)} - z^{(2)}, \ldots, x^{(m)} - z^{(m)}]$.

A vector multiplied by a scalar is a vector. For example $\mathbf{x}c \stackrel{\text{def}}{=} [cx^{(1)}, cx^{(2)}, \ldots, cx^{(m)}]$.

A *dot-product* of two vectors is a scalar. For example, $\mathbf{wx} \stackrel{\text{def}}{=} \sum_{i=1}^{m} w^{(i)} x^{(i)}$. In some books, the dot-product is denoted as $\mathbf{w} \cdot \mathbf{x}$. The two vectors must be of the same dimensionality. Otherwise, the dot-product is undefined.

The multiplication of a matrix $\mathbf{W}$ by a vector $\mathbf{x}$ results in another vector. Let our matrix be,

$$\mathbf{W} = \begin{bmatrix} w^{(1,1)} & w^{(1,2)} & w^{(1,3)} \\ w^{(2,1)} & w^{(2,2)} & w^{(2,3)} \end{bmatrix}.$$

When vectors participate in operations on matrices, a vector is by default represented as a matrix with one column. When the vector is on the right of the matrix, it remains a column vector. We can only multiply a matrix by vector if the vector has the same number of rows as the number of columns in the matrix. Let our vector be $\mathbf{x} \stackrel{\text{def}}{=} [x^{(1)}, x^{(2)}, x^{(3)}]$. Then $\mathbf{W}\mathbf{x}$ is a two-dimensional vector defined as,

$$\begin{aligned}\mathbf{W}\mathbf{x} &= \begin{bmatrix} w^{(1,1)} & w^{(1,2)} & w^{(1,3)} \\ w^{(2,1)} & w^{(2,2)} & w^{(2,3)} \end{bmatrix} \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \end{bmatrix} \\ &\stackrel{\text{def}}{=} \begin{bmatrix} w^{(1,1)}x^{(1)} + w^{(1,2)}x^{(2)} + w^{(1,3)}x^{(3)} \\ w^{(2,1)}x^{(1)} + w^{(2,2)}x^{(2)} + w^{(2,3)}x^{(3)} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{w}^{(1)}\mathbf{x} \\ \mathbf{w}^{(2)}\mathbf{x} \end{bmatrix}\end{aligned}$$

If our matrix had, say, five rows, the result of the product would be a five-dimensional vector.

When the vector is on the left side of the matrix in the multiplication, then it has to be *transposed* before we multiply it by the matrix. The transpose of the vector $\mathbf{x}$ denoted as $\mathbf{x}^\top$ makes a row vector out of a column vector. Let's say,

$$\mathbf{x} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix}, \text{ then } \mathbf{x}^\top \stackrel{\text{def}}{=} \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}.$$

The multiplication of the vector $\mathbf{x}$ by the matrix $\mathbf{W}$ is given by $\mathbf{x}^\top\mathbf{W}$,

$$\begin{aligned}\mathbf{x}^\top\mathbf{W} &= \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix} \begin{bmatrix} w^{(1,1)} & w^{(1,2)} & w^{(1,3)} \\ w^{(2,1)} & w^{(2,2)} & w^{(2,3)} \end{bmatrix} \\ &\stackrel{\text{def}}{=} \begin{bmatrix} w^{(1,1)}x^{(1)} + w^{(2,1)}x^{(2)}, w^{(1,2)}x^{(1)} + w^{(2,2)}x^{(2)}, w^{(1,3)}x^{(1)} + w^{(2,3)}x^{(2)} \end{bmatrix}\end{aligned}$$

As you can see, we can only multiply a vector by a matrix if the vector has the same number of dimensions as the number of rows in the matrix.

2.1.6 Functions

A function is a relation that associates each element x of a set $\mathcal{X}$, the *domain* of the function, to a single element y of another set $\mathcal{Y}$, the *codomain* of the function. A function usually has a name. If the function is called f, this relation is denoted $y = f(x)$ (read f of x), the element x is the argument or input of the function, and y is the value of the function or the output.

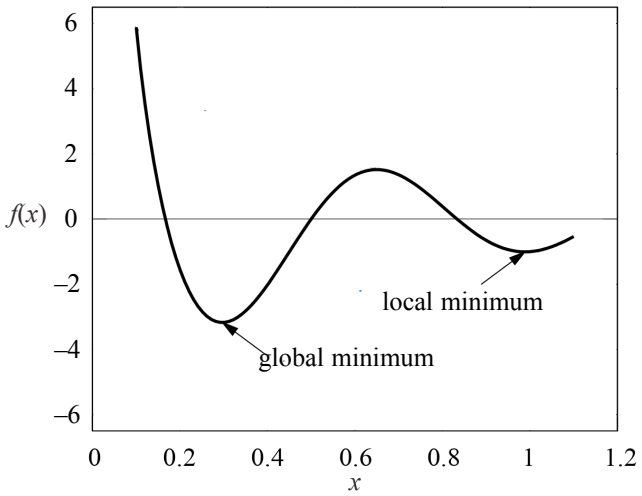


Figure 2.2: A local and a global minima of a function.

The symbol that is used for representing the input is the variable of the function (we often say that f is a function of the variable x).

We say that $f(x)$ has a *local minimum* at $x = c$ if $f(x) \geq f(c)$ for every x in some open interval around $x = c$. An *interval* is a set of real numbers with the property that any number that lies between two numbers in the set is also included in the set. An *open interval* does not include its endpoints and is denoted using parentheses. For example, $(0, 1)$ means "all numbers greater than 0 and less than 1". The minimal value among all the local minima is called the *global minimum*. See illustration in Figure 2.2.

A vector function, denoted as $\mathbf{y} = \boldsymbol{f}(x)$ is a function that returns a vector $\mathbf{y}$. It can have a vector or a scalar argument.

2.1.7 Max and Arg Max

Given a set of values $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, the operator $\max_{a \in \mathcal{A}} f(a)$ returns the highest value $f(a)$ for all elements in the set $\mathcal{A}$. On the other hand, the operator $\arg\max_{a \in \mathcal{A}} f(a)$ returns the element of the set $\mathcal{A}$ that maximizes $f(a)$.

Sometimes, when the set is implicit or infinite, we can write $\max_a f(a)$ or $\arg\max_a f(a)$.

Operators $\min$ and $\arg\min$ operate in a similar manner.

2.1.8 Assignment Operator

The expression $a \leftarrow f(x)$ means that the variable a gets the new value: the result of $f(x)$. We say that the variable a gets assigned a new value. Similarly, $\mathbf{a} \leftarrow [a_1, a_2]$ means that the vector variable $\mathbf{a}$ gets the two-dimensional vector value $[a_1, a_2]$.

2.1.9 Derivative and Gradient

A *derivative* f' of a function f is a function or a value that describes how fast f grows (or decreases). If the derivative is a constant value, like 5 or -3, then the function grows (or decreases) constantly at any point x of its domain. If the derivative f' is a function, then the function f can grow at a different pace in different regions of its domain. If the derivative f' is positive at some point x, then the function f grows at this point. If the derivative of f is negative at some x, then the function decreases at this point. The derivative of zero at x means that the function's slope at x is horizontal.

The process of finding a derivative is called *differentiation*.

Derivatives for basic functions are known. For example if $f(x) = x^2$, then $f'(x) = 2x$; if $f(x) = 2x$ then $f'(x) = 2$; if $f(x) = 2$ then $f'(x) = 0$ (the derivative of any function $f(x) = c$, where c is a constant value, is zero).

If the function we want to differentiate is not basic, we can find its derivative using the *chain rule*. For instance if $F(x) = f(g(x))$, where f and g are some functions, then $F'(x) = f'(g(x))g'(x)$. For example if $F(x) = (5x + 1)^2$ then $g(x) = 5x + 1$ and $f(g(x)) = (g(x))^2$. By applying the chain rule, we find $F'(x) = 2(5x + 1)g'(x) = 2(5x + 1)5 = 50x + 10$.

Gradient is the generalization of derivative for functions that take several inputs (or one input in the form of a vector or some other complex structure). A gradient of a function is a vector of *partial derivatives*. You can look at finding a partial derivative of a function as the process of finding the derivative by focusing on one of the function's inputs and by considering all other inputs as constant values.

For example, if our function is defined as $f([x^{(1)}, x^{(2)}]) = ax^{(1)} + bx^{(2)} + c$, then the partial derivative of function f *with respect to* $x^{(1)}$, denoted as $\frac{\partial f}{\partial x^{(1)}}$, is given by,

$$\frac{\partial f}{\partial x^{(1)}} = a + 0 + 0 = a,$$

where a is the derivative of the function $ax^{(1)}$; the two zeroes are respectively derivatives of $bx^{(2)}$ and c, because $x^{(2)}$ is considered constant when we compute the derivative with respect to $x^{(1)}$, and the derivative of any constant is zero.

Similarly, the partial derivative of function f with respect to $x^{(2)}$, $\frac{\partial f}{\partial x^{(2)}}$, is given by,

$$\frac{\partial f}{\partial x^{(2)}} = 0 + b + 0 = b.$$

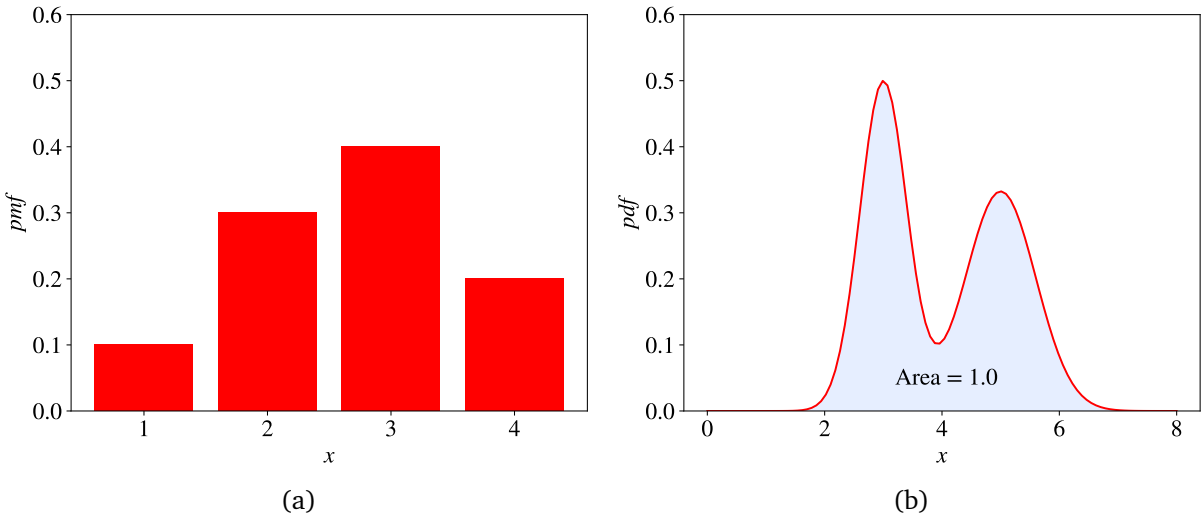


Figure 2.3: A probability mass function and a probability density function.

The gradient of function f, denoted as ∇f is given by the vector $[\frac{\partial f}{\partial x^{(1)}}, \frac{\partial f}{\partial x^{(2)}}]$.

The chain rule works with partial derivatives too, as I illustrate in Chapter 4.

2.2 Random Variable

A *random variable*, usually written as an italic capital letter, like X, is a variable whose possible values are numerical outcomes of a random phenomenon. There are two types of random variables: *discrete* and *continuous*.

A *discrete random variable* takes on only a countable number of distinct values such as red, $yellow$, $blue$ or 1, 2, 3, ...

The *probability distribution* of a discrete random variable is described by a list of probabilities associated with each of its possible values. This list of probabilities is called *probability mass function* (pmf). For example: $\Pr(X = red) = 0.3$, $\Pr(X = yellow) = 0.45$, $\Pr(X = blue) = 0.25$. Each probability in a probability mass function is a value greater than or equal to 0. The sum of probabilities equals 1 (fig. 2.3a).

A *continuous random variable* (CRV) takes an infinite number of possible values in some interval. Examples include height, weight, and time. Because the number of values of a continuous random variable X is infinite, the probability $\Pr(X = c)$ for any c is 0. Therefore, instead of the list of probabilities, the probability distribution of a CRV (a continuous probability distribution) is described by a *probability density function* (pdf). The pdf is a function whose codomain is nonnegative and the area under the curve is equal to 1 (fig. 2.3b).

Let a discrete random variable X have k possible values $\{x_i\}_{i=1}^{k}$. The *expectation* of X denoted as $\mathbb{E}[X]$ is given by,

$$\mathbb{E}[X] \stackrel{\text{def}}{=} \sum_{i=1}^{k} x_i \cdot \Pr(X = x_i) = x_1 \cdot \Pr(X = x_1) + x_2 \cdot \Pr(X = x_2) + \cdots + x_k \cdot \Pr(X = x_k), \quad (2.1)$$

where $\Pr(X = x_i)$ is the probability that X has the value x_i according to the pmf. The expectation of a random variable is also called the *mean*, *average* or *expected value* and is frequently denoted with the letter μ. The expectation is one of the most important *statistics* of a random variable.

Another important statistic is the *standard deviation*, defined as,

$$\sigma \stackrel{\text{def}}{=} \sqrt{\mathbb{E}[(X - \mu)^2]}.$$

Variance, denoted as σ^2 or $var(X)$, is defined as,

$$\sigma^2 = \mathbb{E}[(X - \mu)^2].$$

For a discrete random variable, the standard deviation is given by:

$$\sigma = \sqrt{\Pr(X = x_1)(x_1 - \mu)^2 + \Pr(X = x_2)(x_2 - \mu)^2 + \cdots + \Pr(X = x_k)(x_k - \mu)^2},$$

where $\mu = \mathbb{E}[X]$.

The expectation of a continuous random variable X is given by,

$$\mathbb{E}[X] \stackrel{\text{def}}{=} \int_{\mathbb{R}} x f_X(x)\, dx, \quad (2.2)$$

where f_X is the pdf of the variable X and $\int_{\mathbb{R}}$ is the *integral* of function $x f_X$.

Integral is an equivalent of the summation over all values of the function when the function has a continuous domain. It equals the area under the curve of the function. The property of the pdf that the area under its curve is 1 mathematically means that $\int_{\mathbb{R}} f_X(x)\, dx = 1$.

Most of the time we don't know f_X, but we can observe some values of X. In machine learning, we call these values **examples**, and the collection of these examples is called a **sample** or a **dataset**.

2.3 Unbiased Estimators

Because f_X is usually unknown, but we have a sample $S_X = \{x_i\}_{i=1}^N$, we often content ourselves not with the true values of statistics of the probability distribution, such as expectation, but with their *unbiased estimators*.

We say that $\hat{\theta}(S_X)$ is an unbiased estimator of some statistic θ calculated using a sample S_X drawn from an unknown probability distribution if $\hat{\theta}(S_X)$ has the following property:

$$\mathbb{E}\left[\hat{\theta}(S_X)\right] = \theta,$$

where $\hat{\theta}$ is a *sample statistic*, obtained using a sample S_X and not the real statistic θ that can be obtained only knowing X; the expectation is taken over all possible samples drawn from X. Intuitively, this means that if you can have an unlimited number of such samples as S_X, and you compute some unbiased estimator, such as $\hat{\mu}$, using each sample, then the average of all these $\hat{\mu}$ equals the real statistic μ that you would get computed on X.

It can be shown that an unbiased estimator of an unknown $\mathbb{E}[X]$ (given by either eq. 2.1 or eq. 2.2) is given by $\frac{1}{N}\sum_{i=1}^N x_i$ (called in statistics the *sample mean*).

2.4 Bayes' Rule

The conditional probability $\Pr(X = x|Y = y)$ is the probability of the random variable X to have a specific value x given that another random variable Y has a specific value of y. The **Bayes' Rule** (also known as the **Bayes' Theorem**) stipulates that:

$$\Pr(X = x|Y = y) = \frac{\Pr(Y = y|X = x)\Pr(X = x)}{\Pr(Y = y)}.$$

2.5 Parameter Estimation

Bayes' Rule comes in handy when we have a model of X's distribution, and this model f_θ is a function that has some parameters in the form of a vector θ. An example of such a function could be the Gaussian function that has two parameters, μ and σ, and is defined as:

$$f_\theta(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where $\theta \stackrel{\text{def}}{=} [\mu, \sigma]$.

This function has all the properties of a pdf. Therefore, we can use it as a model of an unknown distribution of X. We can update the values of parameters in the vector θ from the data using the Bayes' Rule:

$$\Pr(\theta = \hat{\theta}|X = x) \leftarrow \frac{\Pr(X = x|\theta = \hat{\theta})\Pr(\theta = \hat{\theta})}{\Pr(X = x)} = \frac{\Pr(X = x|\theta = \hat{\theta})\Pr(\theta = \hat{\theta})}{\sum_{\tilde{\theta}} \Pr(X = x|\theta = \tilde{\theta})}. \tag{2.3}$$

where $\Pr(X = x|\theta = \hat{\theta}) \stackrel{\text{def}}{=} f_{\hat{\theta}}$.

If we have a sample $\mathcal{S}$ of X and the set of possible values for θ is finite, we can easily estimate $\Pr(\theta = \hat{\theta})$ by applying Bayes' Rule iteratively, one example $x \in \mathcal{S}$ at a time. The initial value $\Pr(\theta = \hat{\theta})$ can be guessed such that $\sum_{\hat{\theta}} \Pr(\theta = \hat{\theta}) = 1$. This guess of the probabilities for different $\hat{\theta}$ is called the *prior*.

First, we compute $\Pr(\theta = \hat{\theta}|X = x_1)$ for all possible values $\hat{\theta}$. Then, before updating $\Pr(\theta = \hat{\theta}|X = x)$ once again, this time for $x = x_2 \in \mathcal{S}$ using eq. 2.3, we replace the prior $\Pr(\theta = \hat{\theta})$ in eq. 2.3 by the new estimate $\Pr(\theta = \hat{\theta}) \leftarrow \frac{1}{N} \sum_{x \in \mathcal{S}} \Pr(\theta = \hat{\theta}|X = x)$.

The best value of the parameters θ^* given one example is obtained using the principle of **maximum likelihood**:

$$\theta^* = \arg\max_{\theta} \prod_{i=1}^{N} \Pr(\theta = \hat{\theta}|X = x_i). \tag{2.4}$$

If the set of possible values for θ isn't finite, then we need to optimize eq. 2.4 directly using a numerical optimization routine, such as gradient descent, which we consider in Chapter 4. Usually, we optimize the natural logarithm of the right-hand side expression in eq. 2.4 because the logarithm of a product becomes the sum of logarithms and it's easier for the machine to work with the sum than with a product[1].

2.6 Parameters vs. Hyperparameters

A hyperparameter is a property of a learning algorithm, usually (but not always) having a numerical value. That value influences the way the algorithm works. Hyperparameters aren't learned by the algorithm itself from data. They have to be set by the data analyst before running the algorithm. I show how to do that in Chapter 5.

Parameters are variables that define the model learned 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.

[1] Multiplication of many numbers can give either a very small result or a very large one. It often results in the problem of numerical overflow when the machine cannot store such extreme numbers in memory.

2.7 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.

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[2].

While some learning algorithms naturally allow for more than two classes, others are by nature binary classification algorithms. There are strategies allowing to turn a binary classification learning algorithm into a multiclass one. I talk about one of them in Chapter 7.

Regression is a problem of predicting a real-valued label (often called a *target*) given an unlabeled example. Estimating house price valuation based on house features, such as area, the 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.

2.8 Model-Based vs. Instance-Based Learning

Most supervised learning algorithms are model-based. We have already seen one such algorithm: SVM. Model-based learning algorithms use the training data to create a **model** that has **parameters** learned from the training data. In SVM, the two parameters we saw were $\mathbf{w}^*$ and b^*. After the model was built, 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 the most often in this close neighborhood.

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

2.9 Shallow vs. Deep Learning

A shallow learning algorithm learns the parameters of the model directly from the features of the training examples. Most supervised 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.

Don't worry if you don't understand what that means right now. We look at neural networks more closely in Chapter 6.

Chapter 3

Fundamental Algorithms

In this chapter, I describe five algorithms which are not just the most known but also either very effective on their own or are used as building blocks for the most effective learning algorithms out there.

3.1 Linear Regression

Linear regression is a popular regression learning algorithm that learns a model which is a linear combination of features of the input example.

3.1.1 Problem Statement

We have a collection of labeled examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where N is the size of the collection, $\mathbf{x}_i$ is the D-dimensional feature vector of example $i = 1, \ldots, N$, y_i is a real-valued[1] target and every feature $x_i^{(j)}$, $j = 1, \ldots, D$, is also a real number.

We want to build a model $f_{\mathbf{w},b}(\mathbf{x})$ as a linear combination of features of example $\mathbf{x}$:

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}\mathbf{x} + b, \tag{3.1}$$

where $\mathbf{w}$ is a D-dimensional vector of parameters and b is a real number. The notation $f_{\mathbf{w},b}$ means that the model f is parametrized by two values: $\mathbf{w}$ and b.

We will use the model to predict the unknown y for a given $\mathbf{x}$ like this: $y \leftarrow f_{\mathbf{w},b}(\mathbf{x})$. Two models parametrized by two different pairs $(\mathbf{w}, b)$ will likely produce two different predictions

[1] To say that y_i is real-valued, we write $y_i \in \mathbb{R}$, where $\mathbb{R}$ denotes the set of all real numbers, an infinite set of numbers from minus infinity to plus infinity.

when applied to the same example. We want to find the optimal values $(\mathbf{w}^*, b^*)$. Obviously, the optimal values of parameters define the model that makes the most accurate predictions.

You could have noticed that the form of our linear model in eq. 3.1 is very similar to the form of the SVM model. The only difference is the missing sign operator. The two models are indeed similar. However, the hyperplane in the SVM plays the role of the decision boundary: it's used to separate two groups of examples from one another. As such, it has to be as far from each group as possible.

On the other hand, the hyperplane in linear regression is chosen to be as close to all training examples as possible.

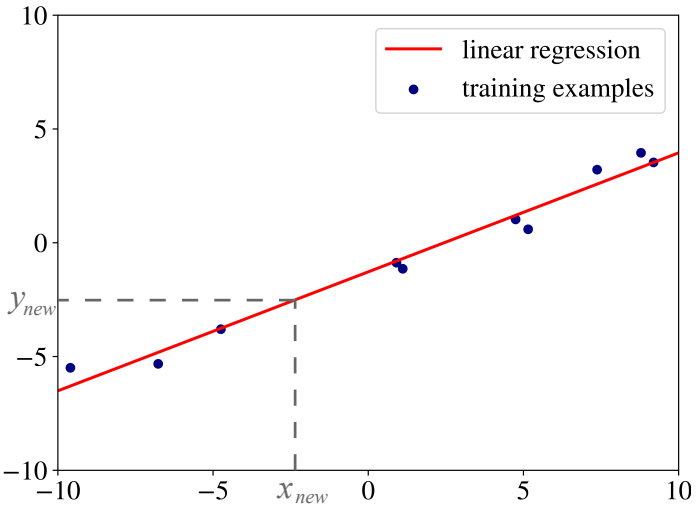


Figure 3.1: Linear Regression for one-dimensional examples.

You can see why this latter requirement is essential by looking at the illustration in Figure 3.1. It displays the regression line (in red) for one-dimensional examples (blue dots). We can use this line to predict the value of the target y_{new} for a new unlabeled input example x_{new}. If our examples are D-dimensional feature vectors (for $D > 1$), the only difference with the one-dimensional case is that the regression model is not a line but a plane (for two dimensions) or a hyperplane (for $D > 2$).

Now you see why it's essential to have the requirement that the regression hyperplane lies as close to the training examples as possible: if the red line in Figure 3.1 was far from the blue dots, the prediction y_{new} would have fewer chances to be correct.

3.1.2 Solution

To get this latter requirement satisfied, the optimization procedure which we use to find the optimal values for $\mathbf{w}^*$ and b^* tries to minimize the following expression:

$$\frac{1}{N}\sum_{i=1\ldots N}(f_{\mathbf{w},b}(\mathbf{x}_i)-y_i)^2. \tag{3.2}$$

In mathematics, the expression we minimize or maximize is called an objective function, or, simply, an objective. The expression $(f_{\mathbf{w},b}(\mathbf{x}_i)-y_i)^2$ in the above objective is called the **loss function**. It's a measure of penalty for misclassification of example i. This particular choice of the loss function is called **squared error loss**. All model-based learning algorithms have a loss function and what we do to find the best model is we try to minimize the objective known as the **cost function**. In linear regression, the cost function is given by the average loss, also called the **empirical risk**. The average loss, or empirical risk, for a model, is the average of all penalties obtained by applying the model to the training data.

Why is the loss in linear regression a quadratic function? Why couldn't we get the absolute value of the difference between the true target y_i and the predicted value $f(\mathbf{x}_i)$ and use that as a penalty? We could. Moreover, we also could use a cube instead of a square.

Now you probably start realizing how many seemingly arbitrary decisions are made when we design a machine learning algorithm: we decided to use the linear combination of features to predict the target. However, we could use a square or some other polynomial to combine the values of features. We could also use some other loss function that makes sense: the absolute difference between $f(\mathbf{x}_i)$ and y_i makes sense, the cube of the difference too; the **binary loss** (1 when $f(\mathbf{x}_i)$ and y_i are different and 0 when they are the same) also makes sense, right?

If we made different decisions about the form of the model, the form of the loss function, and about the choice of the algorithm that minimizes the average loss to find the best values of parameters, we would end up inventing a different machine learning algorithm. Sounds easy, doesn't it? However, do not rush to invent a new learning algorithm. The fact that it's different doesn't mean that it will work better in practice.

People invent new learning algorithms for one of the two main reasons:

1. The new algorithm solves a specific practical problem better than the existing algorithms.
2. The new algorithm has better theoretical guarantees on the quality of the model it produces.

One practical justification of the choice of the linear form for the model is that it's simple. Why use a complex model when you can use a simple one? Another consideration is that linear models rarely overfit. **Overfitting** is the property of a model such that the model predicts very well labels of the examples used during training but frequently makes errors when applied to examples that weren't seen by the learning algorithm during training.

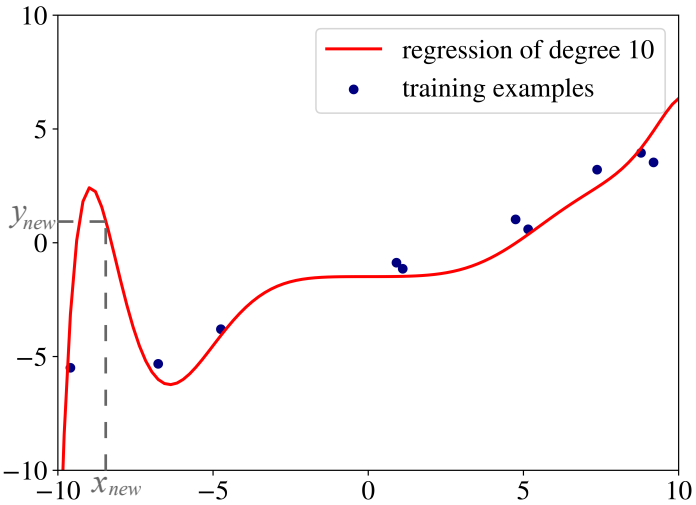


Figure 3.2: Overfitting.

An example of overfitting in regression is shown in Figure 3.2. The data used to build the red regression line is the same as in Figure 3.1. The difference is that this time, this is the polynomial regression with a polynomial of degree 10. The regression line predicts almost perfectly the targets almost all training examples, but will likely make significant errors on new data, as you can see in Figure 3.1 for x_{new}. We talk more about overfitting and how to avoid it Chapter 5.

Now you know why linear regression can be useful: it doesn't overfit much. But what about the squared loss? Why did we decide that it should be squared? In 1705, the French mathematician Adrien-Marie Legendre, who first published the sum of squares method for gauging the quality of the model stated that squaring the error before summing is *convenient*. Why did he say that? The absolute value is not convenient, because it doesn't have a continuous derivative, which makes the function not smooth. Functions that are not smooth create unnecessary difficulties when employing linear algebra to find closed form solutions to optimization problems. Closed form solutions to finding an optimum of a function are simple algebraic expressions and are often preferable to using complex numerical optimization methods, such as **gradient descent** (used, among others, to train neural networks).

Intuitively, squared penalties are also advantageous because they exaggerate the difference between the true target and the predicted one according to the value of this difference. We might also use the powers 3 or 4, but their derivatives are more complicated to work with.

Finally, why do we care about the derivative of the average loss? If we can calculate the gradient of the function in eq. 3.2, we can then set this gradient to zero[2] and find the solution

[2]To find the minimum or the maximum of a function, we set the gradient to zero because the value of the gradient at extrema of a function is always zero. In 2D, the gradient at an extremum is a horizontal line.

to a system of equations that gives us the optimal values $\mathbf{w}^*$ and b^*.

3.2 Logistic Regression

The first thing to say is that logistic regression is not a regression, but a classification learning algorithm. The name comes from statistics and is due to the fact that the mathematical formulation of logistic regression is similar to that of linear regression.

I explain logistic regression on the case of binary classification. However, it can naturally be extended to multiclass classification.

3.2.1 Problem Statement

In logistic regression, we still want to model y_i as a linear function of $\mathbf{x}_i$, however, with a binary y_i this is not straightforward. The linear combination of features such as $\mathbf{w}\mathbf{x}_i + b$ is a function that spans from minus infinity to plus infinity, while y_i has only two possible values.

At the time where the absence of computers required scientists to perform manual calculations, they were eager to find a linear classification model. They figured out that if we define a negative label as 0 and the positive label as 1, we would just need to find a simple continuous function whose codomain is $(0, 1)$. In such a case, if the value returned by the model for input $\mathbf{x}$ is closer to 0, then we assign a negative label to $\mathbf{x}$; otherwise, the example is labeled as positive. One function that has such a property is the **standard logistic function** (also known as the **sigmoid function**):

$$f(x) = \frac{1}{1 + e^{-x}},$$

where e is the base of the natural logarithm (also called *Euler's number*; e^x is also known as the *exp(x)* function in programming languages). Its graph is depicted in Figure 3.3.

The logistic regression model looks like this:

$$f_{\mathbf{w},b}(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{1 + e^{-(\mathbf{w}\mathbf{x}+b)}}. \tag{3.3}$$

You can see the familiar term $\mathbf{w}\mathbf{x} + b$ from linear regression.

By looking at the graph of the standard logistic function, we can see how well it fits our classification purpose: if we optimize the values of $\mathbf{x}$ and b appropriately, we could interpret the output of $f(\mathbf{x})$ as the probability of y_i being positive. For example, if it's higher than or equal to the threshold 0.5 we would say that the class of $\mathbf{x}$ is positive; otherwise, it's negative. In practice, the choice of the threshold could be different depending on the problem. We return to this discussion in Chapter 5 when we talk about model performance assessment.

Now, how do we find optimal $\mathbf{w}^*$ and b^*? In linear regression, we minimized the empirical risk which was defined as the average squared error loss, also known as the **mean squared error** or MSE.

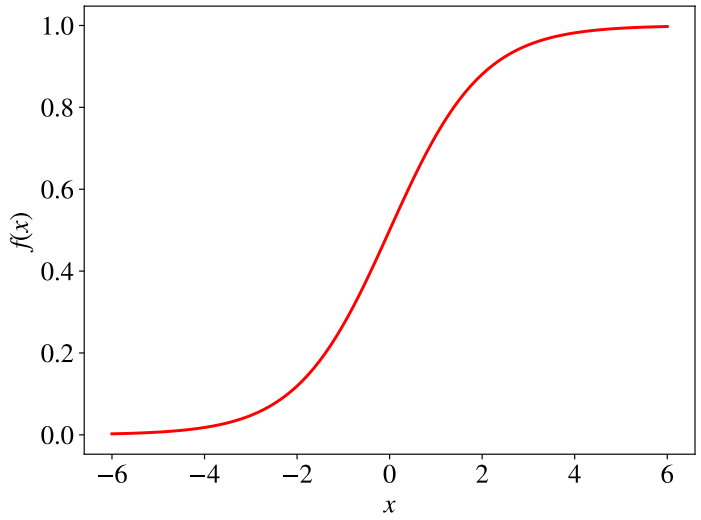


Figure 3.3: Standard logistic function.

3.2.2 Solution

In logistic regression, on the other hand, we maximize the *likelihood* of our training set according to the model. In statistics, the likelihood function defines how likely the observation (an example) is according to our model.

For instance, let's have a labeled example $(\mathbf{x}_i, y_i)$ in our training data. Assume also that we found (guessed) some specific values $\hat{\mathbf{w}}$ and $\hat{b}$ of our parameters. If we now apply our model $f_{\hat{\mathbf{w}},\hat{b}}$ to $\mathbf{x}_i$ using eq. 3.3 we will get some value $0 < p < 1$ as output. If y_i is the positive class, the likelihood of y_i being the positive class, according to our model, is given by p. Similarly, if y_i is the negative class, the likelihood of it being the negative class is given by $1 - p$.

The optimization criterion in logistic regression is called **maximum likelihood**. Instead of minimizing the average loss, like in linear regression, we now maximize the likelihood of the training data according to our model:

$$L_{\mathbf{w},b} \overset{\text{def}}{=} \prod_{i=1\ldots N} f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i}(1 - f_{\mathbf{w},b}(\mathbf{x}_i))^{(1-y_i)}. \tag{3.4}$$

The expression $f_{\mathbf{w},b}(\mathbf{x})^{y_i}(1 - f_{\mathbf{w},b}(\mathbf{x}))^{(1-y_i)}$ may look scary but it's just a fancy mathematical way of saying: "$f_{\mathbf{w},b}(\mathbf{x})$ when $y_i = 1$ and $(1 - f_{\mathbf{w},b}(\mathbf{x}))$ otherwise". Indeed, if $y_i = 1$, then

$(1 - f_{\mathbf{w},b}(\mathbf{x}))^{(1-y_i)}$ equals 1 because $(1 - y_i) = 0$ and we know that anything power 0 equals 1. On the other hand, if $y_i = 0$, then $f_{\mathbf{w},b}(\mathbf{x})^{y_i}$ equals 1 for the same reason.

You may have noticed that we used the product operator $\prod$ in the objective function instead of the sum operator $\sum$ which was used in linear regression. It's because the likelihood of observing N labels for N examples is the product of likelihoods of each observation (assuming that all observations are independent of one another, which is the case). You can draw a parallel with the multiplication of probabilities of outcomes in a series of independent experiments in the probability theory.

Because of the exp function used in the model, in practice, it's more convenient to maximize the *log-likelihood* instead of likelihood. The log-likelihood is defined like follows:

$$LogL_{\mathbf{w},b} \stackrel{\text{def}}{=} \ln(L(_{\mathbf{w},b}(\mathbf{x})) = \sum_{i=1}^{N} y_i \ln f_{\mathbf{w},b}(\mathbf{x}) + (1 - y_i) \ln (1 - f_{\mathbf{w},b}(\mathbf{x})).$$

Because $\ln$ is a *strictly increasing function*, maximizing this function is the same as maximizing its argument, and the solution to this new optimization problem is the same as the solution to the original problem.

Contrary to linear regression, there's no closed form solution to the above optimization problem. A typical numerical optimization procedure used in such cases is **gradient descent**. We talk about it in the next chapter.

3.3 Decision Tree Learning

A decision tree is an acyclic **graph** that can be used to make decisions. In each branching node of the graph, a specific feature j of the feature vector is examined. If the value of the feature is below a specific threshold, then the left branch is followed; otherwise, the right branch is followed. As the leaf node is reached, the decision is made about the class to which the example belongs.

As the title of the section suggests, a decision tree can be learned from data.

3.3.1 Problem Statement

Like previously, we have a collection of labeled examples; labels belong to the set $\{0, 1\}$. We want to build a decision tree that would allow us to predict the class given a feature vector.

3.3.2 Solution

There are various formulations of the decision tree learning algorithm. In this book, we consider just one, called **ID3**.

The optimization criterion, in this case, is the average log-likelihood:

$$\frac{1}{N}\sum_{i=1}^{N} y_i \ln f_{ID3}(\mathbf{x}_i) + (1-y_i)\ln\left(1-f_{ID3}(\mathbf{x}_i)\right), \tag{3.5}$$

where f_{ID3} is a decision tree.

By now, it looks very similar to logistic regression. However, contrary to the logistic regression learning algorithm which builds a **parametric model** $f_{\mathbf{w}^*,b^*}$ by finding an *optimal solution* to the optimization criterion, the ID3 algorithm optimizes it *approximately* by constructing a **nonparametric model** $f_{ID3}(\mathbf{x}) \stackrel{\text{def}}{=} \Pr(y=1|\mathbf{x})$.

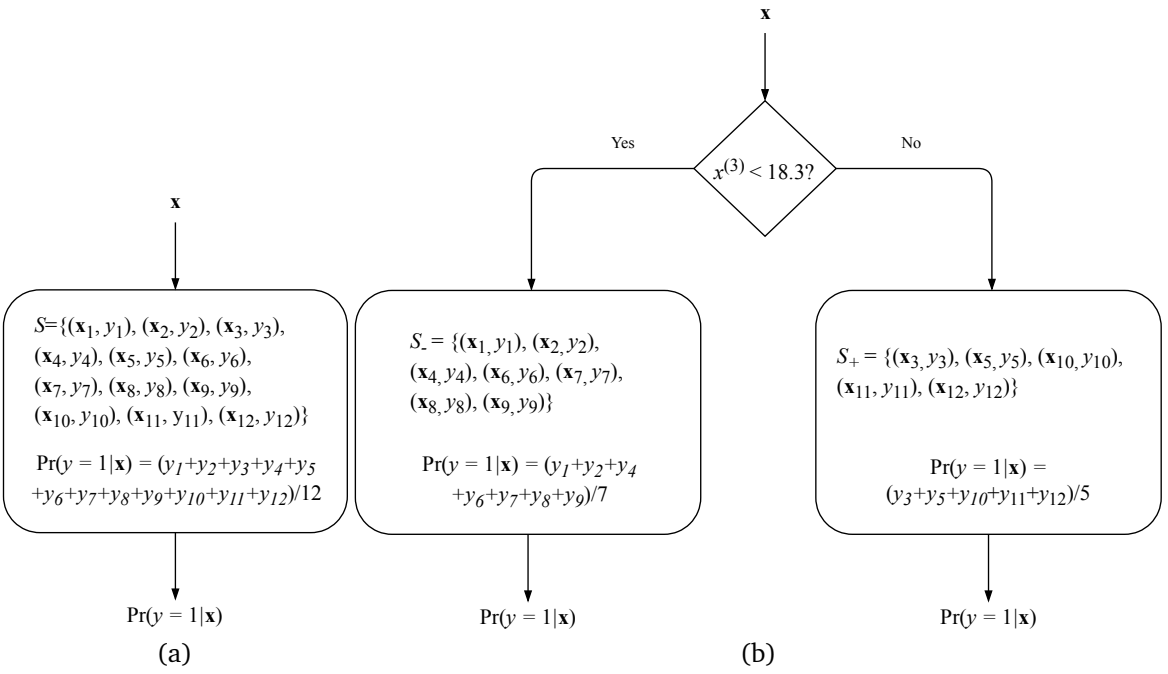


Figure 3.4: An illustration of a decision tree building algorithm. The set $\mathcal{S}$ contains 12 labeled examples. (a) In the beginning, the decision tree only contains the start node; it makes the same prediction for any input. (b) The decision tree after the first split; it tests whether feature 3 is less than 18.3 and, depending on the result, the prediction is made in one of the two leaf nodes.

The ID3 learning algorithm works as follows. Let $\mathcal{S}$ denote a set of labeled examples. In the beginning, the decision tree only has a start node that contains all examples: $\mathcal{S} \stackrel{\text{def}}{=} \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Start with a constant model f_{ID3}^S defined as,

$$f_{ID3}^{\mathcal{S}} \stackrel{\text{def}}{=} \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\in\mathcal{S}} y. \tag{3.6}$$

The prediction given by the above model, $f_{ID3}^{S}(\mathbf{x})$, would be the same for any input $\mathbf{x}$. The corresponding decision tree built using a toy dataset of twelve labeled examples is shown in fig 3.4a.

Then we search through all features $j = 1, \ldots, D$ and all thresholds t, and split the set S into two subsets: $\mathcal{S}_- \stackrel{\text{def}}{=} \{(\mathbf{x}, y) \,|\, (\mathbf{x}, y) \in \mathcal{S}, x^{(j)} < t\}$ and $\mathcal{S}_+ \stackrel{\text{def}}{=} \{(\mathbf{x}, y) \,|\, (\mathbf{x}, y) \in S, x^{(j)} \geq t\}$. The two new subsets would go to two new leaf nodes, and we evaluate, for all possible pairs (j, t) how good the split with pieces $\mathcal{S}_-$ and $\mathcal{S}_+$ is. Finally, we pick the best such values (j, t), split $\mathcal{S}$ into $\mathcal{S}_+$ and $\mathcal{S}_-$, form two new leaf nodes, and continue recursively on $\mathcal{S}_+$ and $\mathcal{S}_-$ (or quit if no split produces a model that's sufficiently better than the current one). A decision tree after one split is illustrated in fig 3.4b.

Now you should wonder what do the words "evaluate how good the split is" mean. In ID3, the goodness of a split is estimated by using the criterion called *entropy*. Entropy is a measure of uncertainty about a random variable. It reaches its maximum when all values of the random variables are equiprobable. Entropy reaches its minimum when the random variable can have only one value. The entropy of a set of examples $\mathcal{S}$ is given by,

$$H(\mathcal{S}) \stackrel{\text{def}}{=} -f_{ID3}^{\mathcal{S}} \ln f_{ID3}^{\mathcal{S}} - (1 - f_{ID3}^{\mathcal{S}}) \ln(1 - f_{ID3}^{\mathcal{S}}).$$

When we split a set of examples by a certain feature j and a threshold t, the entropy of a split, $H(\mathcal{S}_-, \mathcal{S}_+)$, is simply a weighted sum of two entropies:

$$H(\mathcal{S}_-, \mathcal{S}_+) \stackrel{\text{def}}{=} \frac{|\mathcal{S}_-|}{|\mathcal{S}|} H(\mathcal{S}_-) + \frac{|\mathcal{S}_+|}{|\mathcal{S}|} H(\mathcal{S}_+). \tag{3.7}$$

So, in ID3, at each step, at each leaf node, we find a split that minimizes the entropy given by eq. 3.7 or we stop at this leaf node.

The algorithm stops at a leaf node in any of the below situations:

- All examples in the leaf node are classified correctly by the one-piece model (eq. 3.6).
- We cannot find an attribute to split upon.
- The split reduces the entropy less than some ϵ (the value for which has to be found experimentally[3]).
- The tree reaches some maximum depth d (also has to be found experimentally).

[3] In Chapter 5, I show how to do that in the section on hyperparameter tuning.

Because in ID3, the decision to split the dataset on each iteration is local (doesn't depend on future splits), the algorithm doesn't guarantee an optimal solution. The model can be improved by using techniques like *backtracking* during the search for the optimal decision tree at the cost of possibly taking longer to build a model.

The most widely used formulation of a decision tree learning algorithm is called **C4.5**. It has several additional features as compared to ID3:

- it accepts both continuous and discrete features;
- it handles incomplete examples;
- it solves overfitting problem by using a bottom-up technique known as "pruning".

Pruning consists of going back through the tree once it's been created and removing branches that don't contribute significantly enough to the error reduction by replacing them with leaf nodes.

The entropy-based split criterion intuitively makes sense: entropy reaches its minimum of 0 when all examples in $\mathcal{S}$ have the same label; on the other hand, the entropy is at its maximum of 1 when exactly one-half of examples in $\mathcal{S}$ is labeled with 1, making such a leaf useless for classification. The only remaining question is how this algorithm approximately maximizes the average log-likelihood criterion. I leave it for further reading.

3.4 Support Vector Machine

I already presented SVM in the introduction, so this section only fills a couple of blanks. Two critical questions need to be answered:

1. What if there's noise in the data and no hyperplane can perfectly separate positive examples from negative ones?
2. What if the data cannot be separated using a plane, but could be separated by a higher-order polynomial?

You can see both situations depicted in Figure 3.5. In the left case, the data could be separated by a straight line if not for the noise (outliers or examples with wrong labels). In the right case, the decision boundary is a circle and not a straight line.

Remember that in SVM, we want to satisfy the following constraints:

$$\begin{aligned} \mathbf{w}\mathbf{x}_i - b \geq +1 \quad &\text{if } y_i = +1, \\ \mathbf{w}\mathbf{x}_i - b \leq -1 \quad &\text{if } y_i = -1. \end{aligned} \tag{3.8}$$

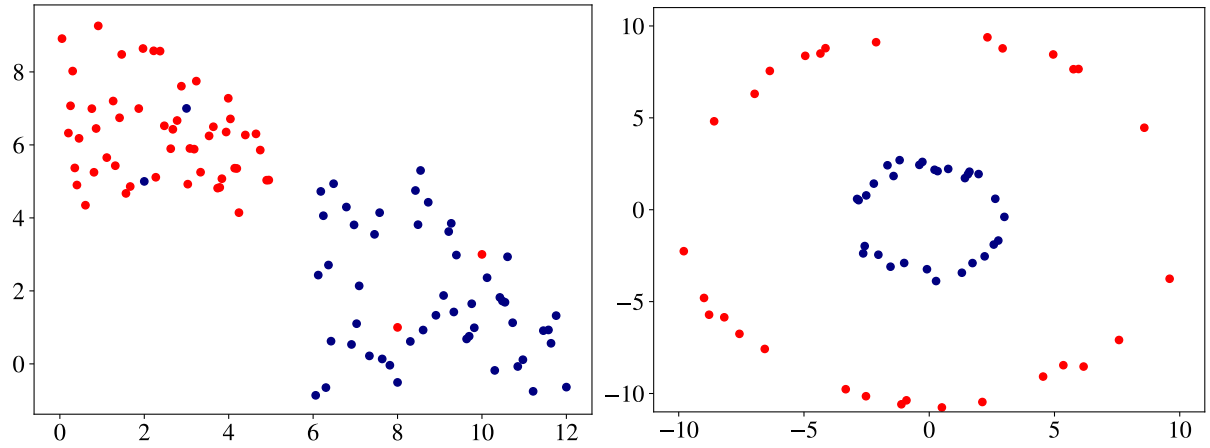


Figure 3.5: Linearly non-separable cases. Left: the presence of noise. Right: inherent nonlinearity.

We also want to minimize $\|\mathbf{w}\|$ so that the hyperplane is equally distant from the closest examples of each class. Minimizing $\|\mathbf{w}\|$ is equivalent to minimizing $\frac{1}{2}||\mathbf{w}||^2$, and the use of this term makes it possible to perform quadratic programming optimization later on. The optimization problem for SVM, therefore, looks like this:

$$\min \frac{1}{2}||\mathbf{w}||^2, \text{ such that } y_i(\mathbf{x}_i\mathbf{w} - b) - 1 \geq 0, i = 1, \ldots, N. \tag{3.9}$$

3.4.1 Dealing with Noise

To extend SVM to cases in which the data is not linearly separable, we introduce the **hinge loss** function: $\max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i - b))$.

The hinge loss function is zero if the constraints in 3.8 are satisfied; in other words, if $\mathbf{w}\mathbf{x}_i$ lies on the correct side of the decision boundary. For data on the wrong side of the decision boundary, the function's value is proportional to the distance from the decision boundary.

We then wish to minimize the following cost function,

$$C\|\mathbf{w}\|^2 + \frac{1}{N}\sum_{i=1}^{N} \max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i - b)),$$

where the hyperparameter C determines the tradeoff between increasing the size of the decision boundary and ensuring that each $\mathbf{x}_i$ lies on the correct side of the decision boundary. The value of C is usually chosen experimentally, just like ID3's hyperparameters ϵ and d.

SVMs that optimize hinge loss are called *soft-margin* SVMs, while the original formulation is referred to as a *hard-margin* SVM.

As you can see, for sufficiently high values of C, the second term in the cost function will become negligible, so the SVM algorithm will try to find the highest margin by completely ignoring misclassification. As we decrease the value of C, making classification errors is becoming more costly, so the SVM algorithm tries to make fewer mistakes by sacrificing the margin size. As we have already discussed, a larger margin is better for generalization. Therefore, C regulates the tradeoff between classifying the training data well (minimizing empirical risk) and classifying future examples well (generalization).

3.4.2 Dealing with Inherent Non-Linearity

SVM can be adapted to work with datasets that cannot be separated by a hyperplane in its original space. Indeed, if we manage to transform the original space into a space of higher dimensionality, we could hope that the examples will become linearly separable in this transformed space. In SVMs, using a function to *implicitly* transform the original space into a higher dimensional space during the cost function optimization is called the **kernel trick**.

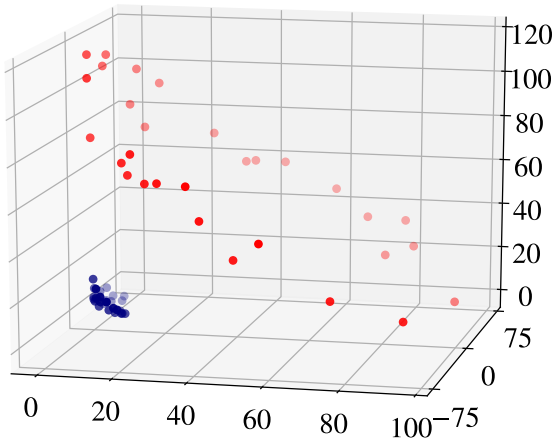


Figure 3.6: The data from Figure 3.5 (right) becomes linearly separable after a transformation into a three-dimensional space.

The effect of applying the kernel trick is illustrated in Figure 3.6. As you can see, it's possible to transform a two-dimensional non-linearly-separable data into a linearly-separable three-dimensional data using a specific mapping $\phi : \mathbf{x} \mapsto \phi(\mathbf{x})$, where $\phi(\mathbf{x})$ is a vector of higher

dimensionality than $\mathbf{x}$. For the example of 2D data in Figure 3.5 (right), the mapping ϕ for that projects a 2D example $\mathbf{x} = [q, p]$ into a 3D space (Figure 3.6) would look like this: $\phi([q, p]) \stackrel{\text{def}}{=} (q^2, \sqrt{2}qp, p^2)$, where $\cdot^2$ means $\cdot$ squared. You see now that the data becomes linearly separable in the transformed space.

However, we don't know a priori which mapping ϕ would work for our data. If we first transform all our input examples using some mapping into very high dimensional vectors and then apply SVM to this data, and we try all possible mapping functions, the computation could become very inefficient, and we would never solve our classification problem.

Fortunately, scientists figured out how to use **kernel functions** (or, simply, **kernels**) to efficiently work in higher-dimensional spaces *without doing this transformation explicitly*. To understand how kernels work, we have to see first how the optimization algorithm for SVM finds the optimal values for $\mathbf{w}$ and b.

The method traditionally used to solve the optimization problem in eq. 3.9 is the *method of Lagrange multipliers*. Instead of solving the original problem from eq. 3.9, it is convenient to solve an equivalent problem formulated like this:

$$\max_{\alpha_1 \ldots \alpha_N} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{N} y_i \alpha_i (\mathbf{x}_i \mathbf{x}_k) y_k \alpha_k \text{ subject to } \sum_{i=1}^{N} \alpha_i y_i = 0 \text{ and } \alpha_i \geq 0, i = 1, \ldots, N,$$

where α_i are called Lagrange multipliers. When formulated like this, the optimization problem becomes a convex quadratic optimization problem, efficiently solvable by quadratic programming algorithms.

Now, you could have noticed that in the above formulation, there is a term $\mathbf{x}_i \mathbf{x}_k$, and this is the only place where the feature vectors are used. If we want to transform our vector space into higher dimensional space, we need to transform $\mathbf{x}_i$ into $\phi(\mathbf{x}_i)$ and $\mathbf{x}_k$ into $\phi(\mathbf{x}_k)$ and then multiply $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_k)$. Doing so would be very costly.

On the other hand, we are only interested in the result of the dot-product $\mathbf{x}_i \mathbf{x}_k$, which, as we know, is a real number. We don't care how this number was obtained as long as it's correct. By using the kernel trick, we can get rid of a costly transformation of original feature vectors into higher-dimensional vectors and avoid computing their dot-product. We replace that by a simple operation on the original feature vectors that gives the same result. For example, instead of transforming (q_1, p_1) into $(q_1^2, \sqrt{2}q_1p_1, p_1^2)$ and (q_2, p_2) into $(q_2^2, \sqrt{2}q_2p_2, p_2^2)$ and then computing the dot-product of $(q_1^2, \sqrt{2}q_1p_1, p_1^2)$ and $(q_2^2, \sqrt{2}q_2p_2, p_2^2)$ to obtain $(q_1^2q_2^2 + 2q_1q_2p_1p_2 + p_1^2p_2^2)$ we could find the dot-product between (q_1, p_1) and (q_2, p_2) to get $(q_1q_2 + p_1p_2)$ and then square it to get exactly the same result $(q_1^2q_2^2 + 2q_1q_2p_1p_2 + p_1^2p_2^2)$.

That was an example of the kernel trick, and we used the quadratic kernel $k(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} (\mathbf{x}_i \mathbf{x}_k)^2$. Multiple kernel functions exist, the most widely used of which is the **RBF kernel**:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right),$$

where $\|\mathbf{x} - \mathbf{x}'\|^2$ is the squared **Euclidean distance** between two feature vectors. The Euclidean distance is given by the following equation:

$$d(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \sqrt{\left(x_i^{(1)} - x_k^{(1)}\right)^2 + \left(x_i^{(2)} - x_k^{(2)}\right)^2 + \cdots + \left(x_i^{(N)} - x_k^{(N)}\right)^2} = \sqrt{\sum_{j=1}^{D}\left(x_i^{(j)} - x_k^{(j)}\right)^2}.$$

It can be shown that the feature space of the RBF (for "radial basis function") kernel has an infinite number of dimensions. By varying the hyperparameter σ, the data analyst can choose between getting a smooth or curvy decision boundary in the original space.

3.5 k-Nearest Neighbors

k-Nearest Neighbors (kNN) is a non-parametric learning algorithm. Contrary to other learning algorithms that allow discarding the training data after the model is built, kNN keeps all training examples in memory. Once a new, previously unseen example $\mathbf{x}$ comes in, the kNN algorithm finds k training examples closest to $\mathbf{x}$ and returns the majority label, in case of classification, or the average label, in case of regression.

The closeness of two examples is given by a distance function. For example, Euclidean distance seen above is frequently used in practice. Another popular choice of the distance function is the negative **cosine similarity**. Cosine similarity defined as,

$$s(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \cos(\angle(\mathbf{x}_i, \mathbf{x}_k)) = \frac{\sum_{j=1}^{D} x_i^{(j)} x_k^{(j)}}{\sqrt{\sum_{j=1}^{D}\left(x_i^{(j)}\right)^2}\sqrt{\sum_{j=1}^{D}\left(x_k^{(j)}\right)^2}},$$

is a measure of similarity of the directions of two vectors. If the angle between two vectors is 0 degrees, then two vectors point to the same direction, and cosine similarity is equal to 1. If the vectors are orthogonal, the cosine similarity is 0. For vectors pointing in opposite directions, the cosine similarity is -1. If we want to use cosine similarity as a distance metric, we need to multiply it by -1. Other popular distance metrics include Chebychev distance, Mahalanobis distance, and Hamming distance. The choice of the distance metric, as well as the value for k, are the choices the analyst makes before running the algorithm. So these are hyperparameters. The distance metric could also be learned from data (as opposed to guessing it). We talk about that in Chapter 10.