Prutor

AI ML Notes

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

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

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.

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:

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

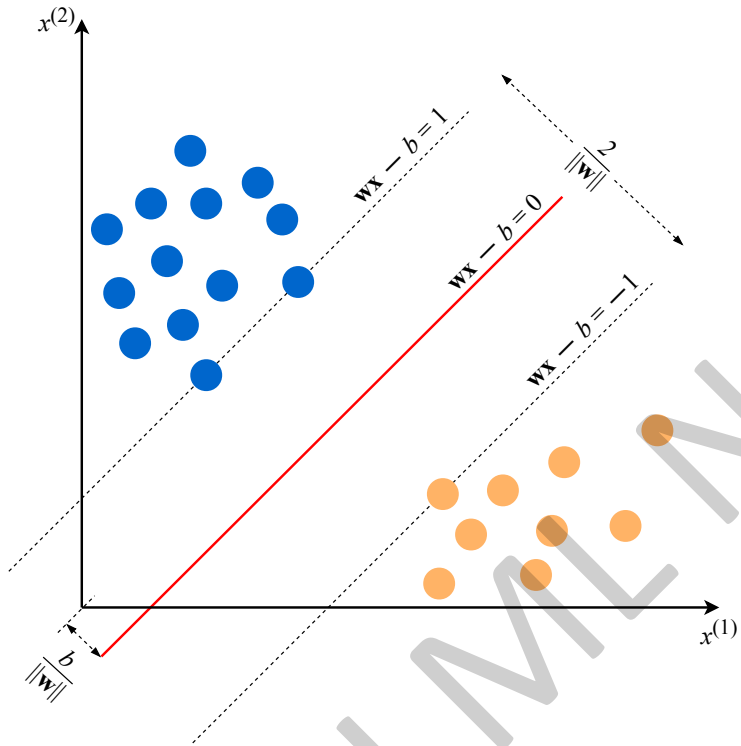


Figure 1: An example of an SVM model for two-dimensional feature vectors.

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* $y_i(\mathbf{wx}_i - b) \geq 1$ *for* $i = 1, \ldots, N$. The expression $y_i(\mathbf{wx}_i - b) \geq 1$ 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. The blue and orange circles represent, respectively, positive and negative examples, and the line given by $\mathbf{wx} - b = 0$ 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 $\mathbf{wx} - b = 1$ and $\mathbf{wx} - b = -1$ define two parallel hyperplanes, as you see in Figure 1. The distance between these hyperplanes is given by $\frac{2}{\|\mathbf{w}\|}$, so the smaller

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.

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[3].

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.

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

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 $\mathbf{x}$:

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}\mathbf{x} + b, \tag{1}$$

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 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. 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.

You can see why this latter requirement is essential by looking at the illustration in Figure 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

[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.

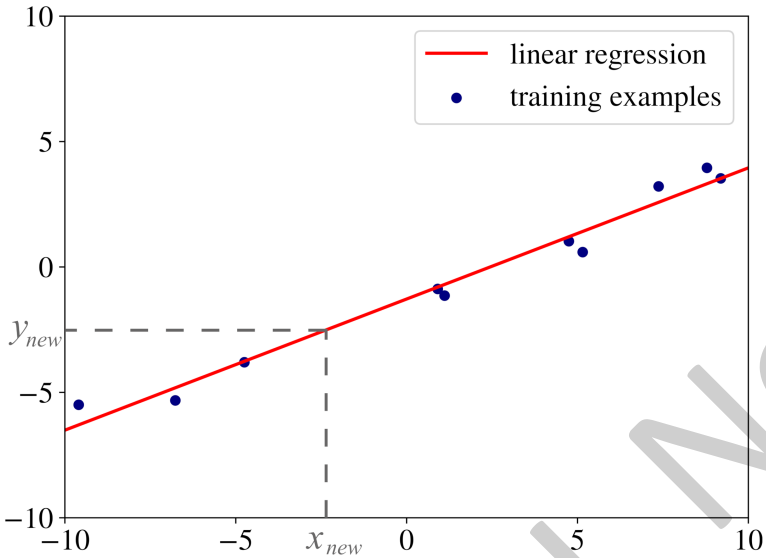


Figure 1: Linear Regression for one-dimensional examples.

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 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:

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

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.

An example of overfitting in regression is shown in Figure 2. The data used to build the red regression line is the same as in Figure 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 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

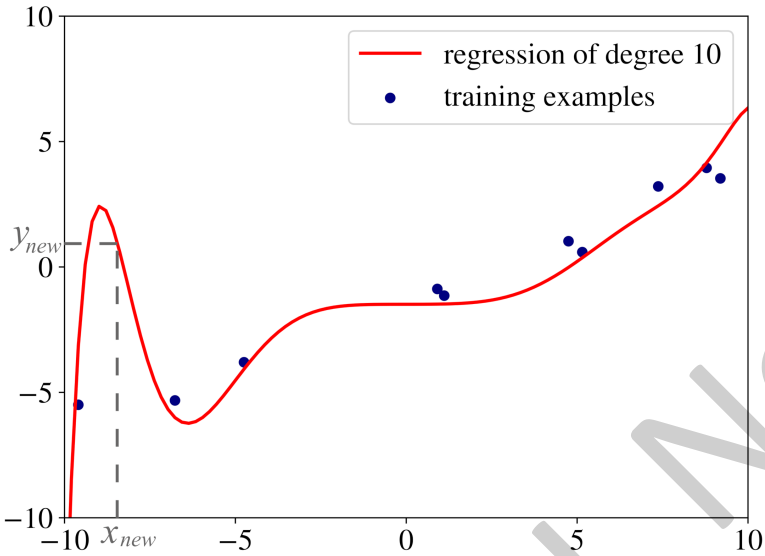


Figure 2: Overfitting.

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. 2, we can then set this gradient to zero[2] and find the solution 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.

[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.

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.

The logistic regression model looks like this:

$$f_{\mathbf{w},b}(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{1 + e^{-(\mathbf{w}\mathbf{x}+b)}}. \tag{3}$$

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 $\mathbf{w}$ 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.

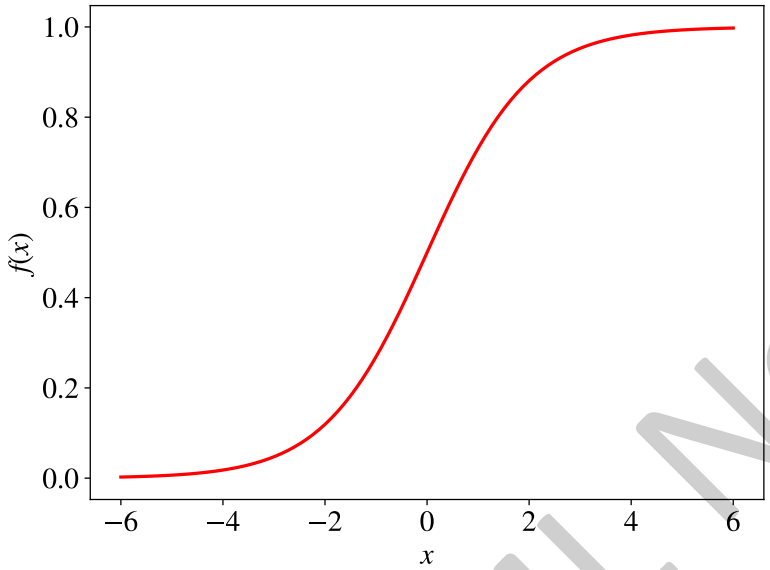


Figure 3: Standard logistic function.

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 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:

$$L_{\mathbf{w},b} \overset{\text{def}}{=} \prod_{i=1\ldots N} f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i}(1 - f_{\mathbf{w},b}(\mathbf{x}_i))^{(1-y_i)}. \tag{4}$$

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:

$$\frac{1}{N}\sum_{i=1}^{N} y_i \ln f_{ID3}(\mathbf{x}_i) + (1-y_i)\ln\left(1-f_{ID3}(\mathbf{x}_i)\right), \tag{5}$$

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** $f_{ID3}(\mathbf{x}) \stackrel{\text{def}}{=} \Pr(y=1|\mathbf{x})$.

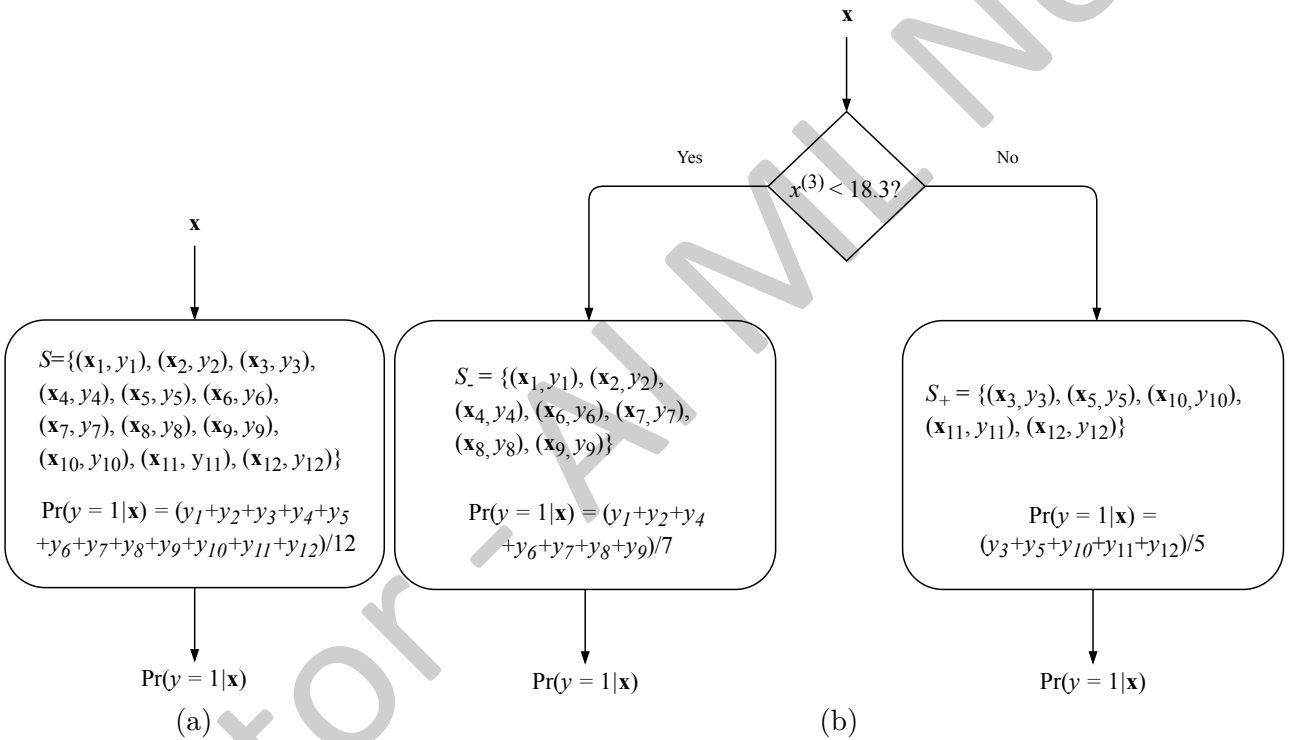


Figure 4: An illustration of a decision tree building algorithm. The set $\mathcal{S}$ contains 12 labeled examples. (a) In the beginning, the decision tree only contains the start node; it makes the same prediction for any input. (b) The decision tree after the first split; it tests whether feature 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,

$$f^{\mathcal{S}}_{ID3} \stackrel{\text{def}}{=} \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\in\mathcal{S}} y. \tag{6}$$

The prediction given by the above model, $f^{S}_{ID3}(\mathbf{x})$, 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 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 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,

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

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:

$$H(\mathcal{S}_-, \mathcal{S}_+) \stackrel{\text{def}}{=} \frac{|\mathcal{S}_-|}{|\mathcal{S}|} H(\mathcal{S}_-) + \frac{|\mathcal{S}_+|}{|\mathcal{S}|} H(\mathcal{S}_+). \tag{7}$$

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

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

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

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 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:

$$\begin{aligned} \mathbf{w}\mathbf{x}_i - b \geq +1 \quad &\text{if } y_i = +1, \\ \mathbf{w}\mathbf{x}_i - b \leq -1 \quad &\text{if } y_i = -1. \end{aligned} \tag{8}$$

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:

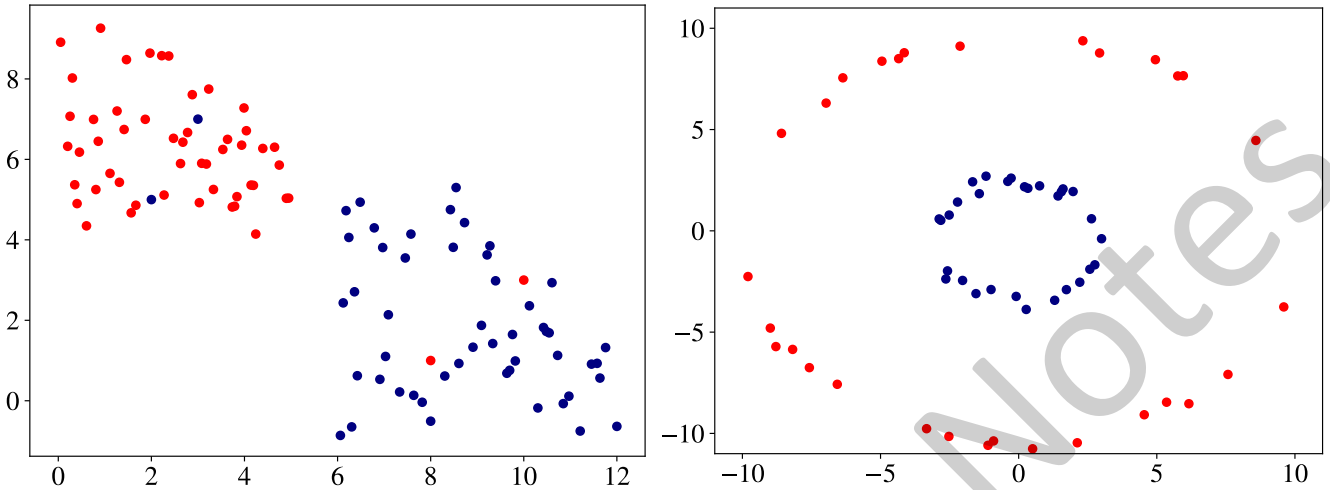


Figure 5: Linearly non-separable cases. Left: the presence of noise. Right: inherent nonlinearity.

$$\min \frac{1}{2}||\mathbf{w}||^2, \text{ such that } y_i(\mathbf{x}_i\mathbf{w} - b) - 1 \geq 0, i = 1, \ldots, N. \tag{9}$$

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 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**.

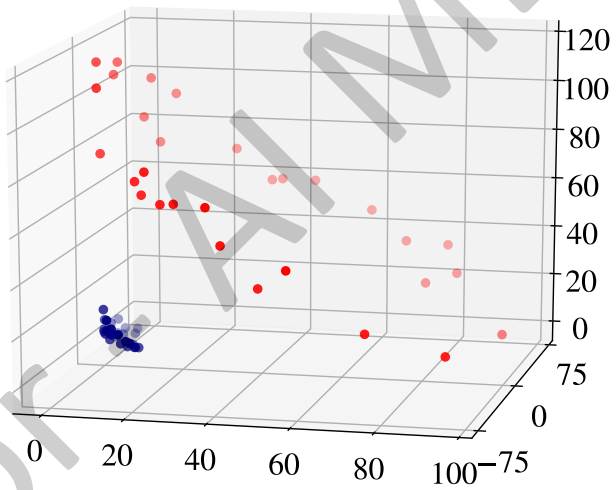


Figure 6: The data from Figure 5 (right) becomes linearly separable after a transformation into a three-dimensional space.

The effect of applying the kernel trick is illustrated in Figure 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 5 (right), the mapping ϕ for that projects a 2D example $\mathbf{x} = [q, p]$ into a 3D space (Figure 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. 9 is the *method of Lagrange multipliers*. Instead of solving the original problem from eq. 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 $(q_1^2, \sqrt{2} q_1 p_1, p_1^2)$ and (q_2, p_2) into $(q_2^2, \sqrt{2} q_2 p_2, p_2^2)$ and then computing the dot-product of $(q_1^2, \sqrt{2} q_1 p_1, p_1^2)$ and $(q_2^2, \sqrt{2} q_2 p_2, p_2^2)$ to obtain $(q_1^2 q_2^2 + 2 q_1 q_2 p_1 p_2 + p_1^2 p_2^2)$ we could find the dot-product between (q_1, p_1) and (q_2, p_2) to get $(q_1 q_2 + p_1 p_2)$ and then square it to get exactly the same result $(q_1^2 q_2^2 + 2 q_1 q_2 p_1 p_2 + p_1^2 p_2^2)$.

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.

4 Anatomy of a Learning Algorithm

4.1 Building Blocks of a Learning Algorithm

You may have noticed by reading the previous chapter that each learning algorithm we saw consisted of three parts:

1) a loss function;
2) an optimization criterion based on the loss function (a cost function, for example); and
3) an optimization routine leveraging training data to find a solution to the optimization criterion.

These are the building blocks of any learning algorithm. You saw in the previous chapter that some algorithms were designed to explicitly optimize a specific criterion (both linear and logistic regressions, SVM). Some others, including decision tree learning and kNN, optimize the criterion implicitly. Decision tree learning and kNN are among the oldest machine learning algorithms and were invented experimentally based on intuition, without a specific global optimization criterion in mind, and (like it often happened in scientific history) the optimization criteria were developed later to explain why those algorithms work.

By reading modern literature on machine learning, you often encounter references to **gradient descent** or **stochastic gradient descent**. These are two most frequently used optimization algorithms used in cases where the optimization criterion is differentiable.

Gradient descent is an iterative optimization algorithm for finding the minimum of a function. To find a *local* minimum of a function using gradient descent, one starts at some random point and takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

Gradient descent can be used to find optimal parameters for linear and logistic regression, SVM and also neural networks which we consider later. For many models, such as logistic regression or SVM, the optimization criterion is *convex*. Convex functions have only one minimum, which is global. Optimization criteria for neural networks are not convex, but in practice even finding a local minimum suffices.

Let's see how gradient descent works.

4.2 Gradient Descent

In this section, I demonstrate how gradient descent finds the solution to a linear regression problem[1]. I illustrate my description with Python code as well as with plots that show how the solution improves after some iterations of gradient descent. I use a dataset with only

[1]As you know, linear regression has a closed form solution. That means that gradient descent is not needed to solve this specific type of problem. However, for illustration purposes, linear regression is a perfect problem to explain gradient descent.

one feature. However, the optimization criterion will have two parameters: w and b. The extension to multi-dimensional training data is straightforward: you have variables $w^{(1)}$, $w^{(2)}$, and b for two-dimensional data, $w^{(1)}$, $w^{(2)}$, $w^{(3)}$, and b for three-dimensional data and so on.

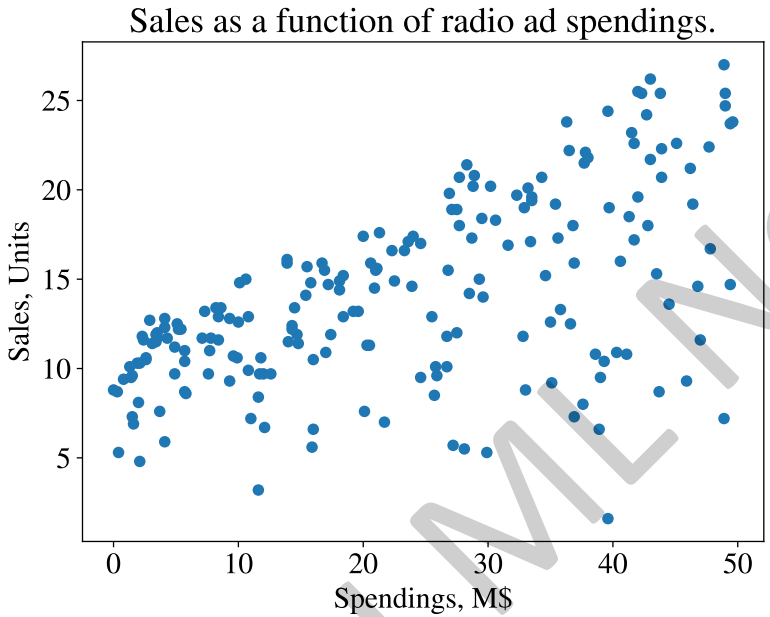


Figure 1: The original data. The Y-axis corresponds to the sales in units (the quantity we want to predict), the X-axis corresponds to our feature: the spendings on radio ads in M$.

To give a practical example, I use the real dataset (can be found on the book's wiki) with the following columns: the Spendings of various companies on radio advertising each year and their annual Sales in terms of units sold. We want to build a regression model that we can use to predict units sold based on how much a company spends on radio advertising. Each row in the dataset represents one specific company:

Company	Spendings, M$	Sales, Units
1	37.8	22.1
2	39.3	10.4
3	45.9	9.3
4	41.3	18.5
..	..	..

We have data for 200 companies, so we have 200 training examples in the form $(x_i, y_i) = (Spendings_i, Sales_i)$. Figure 1 shows all examples on a 2D plot.

Remember that the linear regression model looks like this: $f(x) = wx + b$. We don't know what the optimal values for w and b are and we want to learn them from data. To do that,

we look for such values for w and b that minimize the mean squared error:

$$l \stackrel{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2.$$

Gradient descent starts with calculating the partial derivative for every parameter:

$$\begin{aligned}\frac{\partial l}{\partial w} &= \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (wx_i + b));\\ \frac{\partial l}{\partial b} &= \frac{1}{N}\sum_{i=1}^{N} -2(y_i - (wx_i + b)).\end{aligned} \tag{1}$$

To find the partial derivative of the term $(y_i - (wx + b))^2$ with respect to w I applied the *chain rule*. Here, we have the chain $f = f_2(f_1)$ where $f_1 = y_i - (wx + b)$ and $f_2 = f_1^2$. To find a partial derivative of f with respect to w we have to first find the partial derivative of f with respect to f_2 which is equal to $2(y_i - (wx + b))$ (from calculus, we know that the derivative $\frac{\partial}{\partial x}x^2 = 2x$) and then we have to multiply it by the partial derivative of $y_i - (wx + b)$ with respect to w which is equal to $-x$. So overall $\frac{\partial l}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (wx_i + b))$. In a similar way, the partial derivative of l with respect to b, $\frac{\partial l}{\partial b}$, was calculated.

Gradient descent proceeds in **epochs**. An epoch consists of using the training set entirely to update each parameter. In the beginning, the first epoch, we initialize[2] $w \leftarrow 0$ and $b \leftarrow 0$. The partial derivatives, $\frac{\partial l}{\partial w}$ and $\frac{\partial l}{\partial b}$ given by eq. 1 equal, respectively, $\frac{-2}{N}\sum_{i=1}^{N} x_i y_i$ and $\frac{-2}{N}\sum_{i=1}^{N} y_i$. At each epoch, we update w and b using partial derivatives. The learning rate α controls the size of an update:

$$\begin{aligned} w &\leftarrow w - \alpha\frac{\partial l}{\partial w};\\ b &\leftarrow b - \alpha\frac{\partial l}{\partial b}.\end{aligned} \tag{2}$$

We subtract (as opposed to adding) partial derivatives from the values of parameters because derivatives are indicators of growth of a function. If a derivative is positive at some point[3], then the function grows at this point. Because we want to minimize the objective function,

[2] In complex models, such as neural networks, which have thousands of parameters, the initialization of parameters may significantly affect the solution found using gradient descent. There are different initialization methods (at random, with all zeroes, with small values around zero, and others) and it is an important choice the data analyst has to make.

[3] A point is given by the current values of parameters.

when the derivative is positive we know that we need to move our parameter in the opposite direction (to the left on the axis of coordinates). When the derivative is negative (function is decreasing), we need to move our parameter to the right to decrease the value of the function even more. Subtracting a negative value from a parameter moves it to the right.

At the next epoch, we recalculate partial derivatives using eq. 1 with the updated values of w and b; we continue the process until convergence. Typically, we need many epochs until we start seeing that the values for w and b don't change much after each epoch; then we stop.

It's hard to imagine a machine learning engineer who doesn't use Python. So, if you waited for the right moment to start learning Python, this is that moment. Below, I show how to program gradient descent in Python.

The function that updates the parameters w and b during one epoch is shown below:

```
def update_w_and_b(spendings, sales, w, b, alpha):
    dl_dw = 0.0
    dl_db = 0.0
    N = len(spendings)

    for i in range(N):
        dl_dw += -2*spendings[i]*(sales[i] - (w*spendings[i] + b))
        dl_db += -2*(sales[i] - (w*spendings[i] + b))

    # update w and b
    w = w - (1/float(N))*dl_dw*alpha
    b = b - (1/float(N))*dl_db*alpha

    return w, b
```

The function that loops over multiple epochs is shown below:

```
def train(spendings, sales, w, b, alpha, epochs):
    for e in range(epochs):
        w, b = update_w_and_b(spendings, sales, w, b, alpha)

        # log the progress
        if e % 400 == 0:
            print("epoch:", e, "loss: ", avg_loss(spendings, sales, w, b))

    return w, b
```

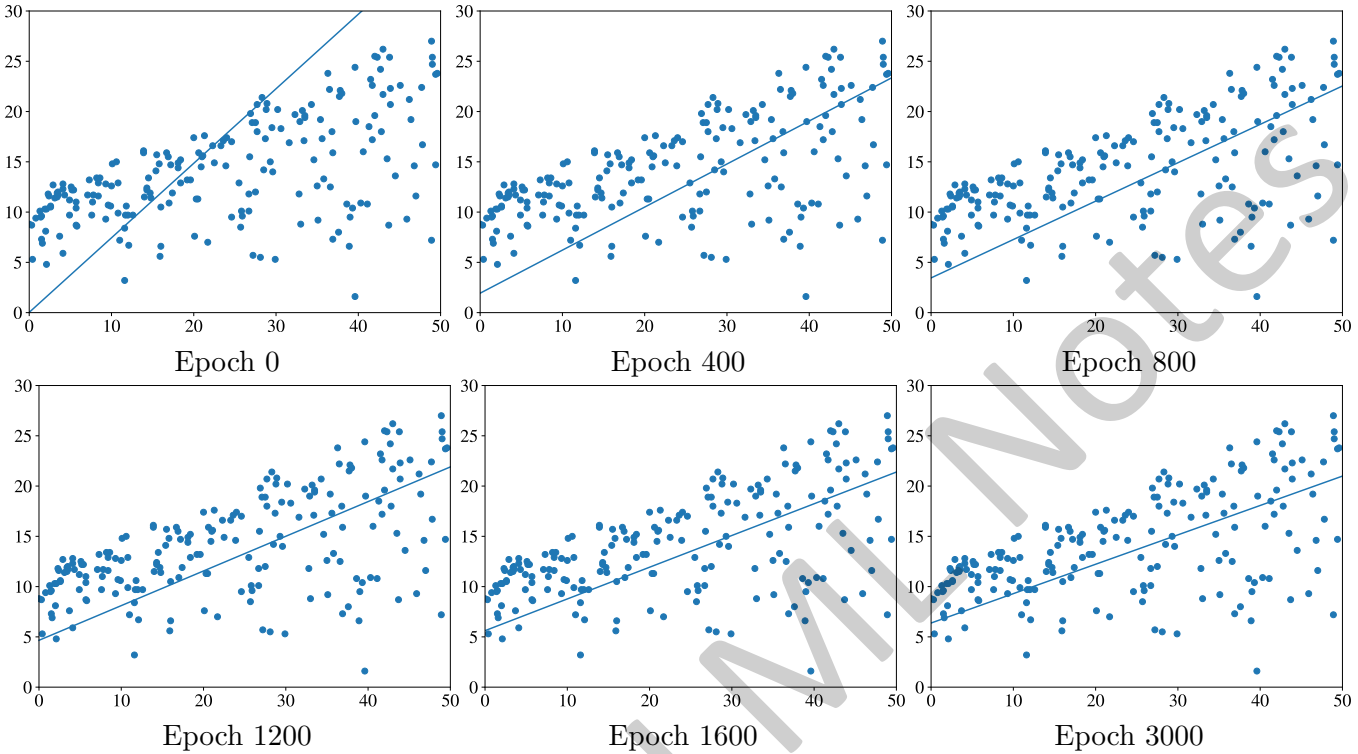


Figure 2: The evolution of the regression line through gradient descent epochs.

The function *avg_loss* in the above code snippet is a function that computes the mean squared error. It is defined as:

```
25 def avg_loss(spendings, sales, w, b):
26     N = len(spendings)
27     total_error = 0.0
28     for i in range(N):
29         total_error += (sales[i] - (w*spendings[i] + b))**2
30     return total_error / float(N)
```

If we run the *train* function for $\alpha = 0.001$, $w = 0.0$, $b = 0.0$, and 15,000 epochs, we will see the following output (shown partially):

```
epoch:  0 loss: 92.32078294903626
epoch:  400 loss: 33.79131790081576
epoch:  800 loss: 27.9918542960729
epoch:  1200 loss: 24.33481690722147
epoch:  1600 loss: 22.028754937538633
...
epoch:  2800 loss: 19.07940244306619
```

You can see that the average loss decreases as the *train* function loops through epochs. Figure 2 shows the evolution of the regression line through epochs.

Finally, once we have found the optimal values of parameters w and b, the only missing piece is a function that makes predictions:

```
31 def predict(x, w, b):
32     return w*x + b
```

Try to execute the following code:

```
33 w, b = train(x, y, 0.0, 0.0, 0.001, 15000)
34 x_new = 23.0
35 y_new = predict(x_new, w, b)
36 print(y_new)
```

The output is 13.97.

Gradient descent is sensitive to the choice of the learning rate α. It is also slow for large datasets. Fortunately, several significant improvements to this algorithm have been proposed.

Minibatch stochastic gradient descent (minibatch SGD) is a version of the algorithm that speeds up the computation by approximating the gradient using smaller batches (subsets) of the training data. SGD itself has various "upgrades". **Adagrad** is a version of SGD that scales α for each parameter according to the history of gradients. As a result, α is reduced for very large gradients and vice-versa. **Momentum** is a method that helps accelerate SGD by orienting the gradient descent in the relevant direction and reducing oscillations. In neural network training, variants of SGD such as **RMSprop** and **Adam**, are very frequently used.

Notice that gradient descent and its variants are not machine learning algorithms. They are solvers of minimization problems in which the function to minimize has a gradient (in most points of its domain).

4.3 How Machine Learning Engineers Work

Unless you are a research scientist or work for a huge corporation with a large R&D budget, you usually don't implement machine learning algorithms yourself. You don't implement gradient descent or some other solver either. You use libraries, most of which are open source. A library is a collection of algorithms and supporting tools implemented with stability and efficiency in mind. The most frequently used in practice open-source machine learning library is *scikit-learn*. It's written in Python and C. Here's how you do linear regression in scikit-learn:

```
1 def train(x, y):
2     from sklearn.linear_model import LinearRegression
3     model = LinearRegression().fit(x,y)
4     return model
```

```
5
6  model = train(x,y)
7
8  x_new = 23.0
9  y_new = model.predict(x_new)
10 print(y_new)
```

The output will, again, be 13.97. Easy, right? You can replace LinearRegression with some other type of regression learning algorithm without modifying anything else. It just works. The same can be said about classification. You can easily replace *LogisticRegression* algorithm with *SVC* algorithm (this is scikit-learn's name for the Support Vector Machine algorithm), *DecisionTreeClassifier*, *NearestNeighbors* or many other classification learning algorithms implemented in scikit-learn.

4.4 Learning Algorithms' Particularities

Here, I outline some practical particularities that can differentiate one learning algorithm from another. You already know that different learning algorithms can have different hyperparameters (C in SVM, ϵ and d in ID3). Solvers such as gradient descent can also have hyperparameters, like α for example.

Some algorithms, like decision tree learning, can accept categorical features. For example, if you have a feature "color" that can take values "red", "yellow", or "green", you can keep this feature as is. SVM, logistic and linear regression, as well as kNN (with cosine similarity or Euclidean distance metrics), expect numerical values for all features. All algorithms implemented in scikit-learn expect numerical features. In the next chapter, I show how to convert categorical features into numerical ones.

Some algorithms, like SVM, allow the data analyst to provide weightings for each class. These weightings influence how the decision boundary is drawn. If the weight of some class is high, the learning algorithm tries to not make errors in predicting training examples of this class (typically, for the cost of making an error elsewhere). That could be important if instances of some class are in the minority in your training data, but you would like to avoid misclassifying examples of that class as much as possible.

Some classification models, like SVM and kNN, given a feature vector only output the class. Others, like logistic regression or decision trees, can also return the score between 0 and 1 which can be interpreted as either how confident the model is about the prediction or as the probability that the input example belongs to a certain class[4].

Some classification algorithms (like decision tree learning, logistic regression, or SVM) build the model using the whole dataset at once. If you have got additional labeled examples, you have

[4]If it's really necessary, the score for SVM and kNN predictions could be synthetically created using simple techniques.

5 Basic Practice

Until now, I only mentioned in passing some issues that a data analyst needs to consider when working on a machine learning problem: feature engineering, overfitting, and hyperparameter tuning. In this chapter, we talk about these and other challenges that have to be addressed before you can type model = LogisticRegression().fit(x,y) in scikit-learn.

5.1 Feature Engineering

When a product manager tells you "We need to be able to predict whether a particular customer will stay with us. Here are the logs of customers' interactions with our product for five years." you cannot just grab the data, load it into a library and get a prediction. You need to build a **dataset** first.

Remember from the first chapter that 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 $x^{(j)}$.

The problem of transforming raw data into a dataset is called **feature engineering**. For most practical problems, feature engineering is a labor-intensive process that demands from the data analyst a lot of creativity and, preferably, domain knowledge.

For example, to transform the logs of user interaction with a computer system, one could create features that contain information about the user and various statistics extracted from the logs. For each user, one feature would contain the price of the subscription; other features would contain the frequency of connections per day, week and year. Another feature would contain the average session duration in seconds or the average response time for one request, and so on. Everything measurable can be used as a feature. The role of the data analyst is to create *informative* features: those would allow the learning algorithm to build a model that predicts well labels of the data used for training. Highly informative features are also called features with high *predictive power*. For example, the average duration of a user's session has high predictive power for the problem of predicting whether the user will keep using the application in the future.

We say that a model has a **low bias** when it predicts the training data well. That is, the model makes few mistakes when we use it to predict labels of the examples used to build the model.

5.1.1 One-Hot Encoding

Some learning algorithms only work with numerical feature vectors. When some feature in your dataset is categorical, like "colors" or "days of the week," you can transform such a categorical feature into several binary ones.

5.1.3 Normalization

Normalization is the process of converting an actual range of values which a numerical feature can take, into a standard range of values, typically in the interval $[-1, 1]$ or $[0, 1]$.

For example, suppose the natural range of a particular feature is 350 to 1450. By subtracting 350 from every value of the feature, and dividing the result by 1100, one can normalize those values into the range $[0, 1]$.

More generally, the normalization formula looks like this:

$$\bar{x}^{(j)} = \frac{x^{(j)} - min^{(j)}}{max^{(j)} - min^{(j)}},$$

where $min^{(j)}$ and $max^{(j)}$ are, respectively, the minimum and the maximum value of the feature j in the dataset.

Why do we normalize? Normalizing the data is not a strict requirement. However, in practice, it can lead to an increased speed of learning. Remember the gradient descent example from the previous chapter. Imagine you have a two-dimensional feature vector. When you update the parameters of $w^{(1)}$ and $w^{(2)}$, you use partial derivatives of the mean squared error with respect to $w^{(1)}$ and $w^{(2)}$. If $x^{(1)}$ is in the range $[0, 1000]$ and $x^{(2)}$ the range $[0, 0.0001]$, then the derivative with respect to a larger feature will dominate the update.

Additionally, it's useful to ensure that our inputs are roughly in the same relatively small range to avoid problems which computers have when working with very small or very big numbers (known as numerical overflow).

5.1.4 Standardization

Standardization (or **z-score normalization**) is the procedure during which the feature values are rescaled so that they have the properties of a *standard normal distribution* with $\mu = 0$ and $\sigma = 1$, where μ is the mean (the average value of the feature, averaged over all examples in the dataset) and σ is the standard deviation from the mean.

Standard scores (or z-scores) of features are calculated as follows:

$$\hat{x}^{(j)} = \frac{x^{(j)} - \mu^{(j)}}{\sigma^{(j)}}.$$

You may ask when you should use normalization and when standardization. There's no definitive answer to this question. Usually, if your dataset is not too big and you have time, you can try both and see which one performs better for your task.

If you don't have time to run multiple experiments, as a rule of thumb:

- unsupervised learning algorithms, in practice, more often benefit from standardization than from normalization;
- standardization is also preferred for a feature if the values this feature takes are distributed close to a normal distribution (so-called bell curve);
- again, standardization is preferred for a feature if it can sometimes have extremely high or low values (outliers); this is because normalization will "squeeze" the normal values into a very small range;
- in all other cases, normalization is preferable.

Feature rescaling is usually beneficial to most learning algorithms. However, modern implementations of the learning algorithms, which you can find in popular libraries, are robust to features lying in different ranges.

5.1.5 Dealing with Missing Features

In some cases, the data comes to the analyst in the form of a dataset with features already defined. In some examples, values of some features can be missing. That often happens when the dataset was handcrafted, and the person working on it forgot to fill some values or didn't get them measured at all.

The typical approaches of dealing with missing values for a feature include:

- removing the examples with missing features from the dataset (that can be done if your dataset is big enough so you can sacrifice some training examples);
- using a learning algorithm that can deal with missing feature values (depends on the library and a specific implementation of the algorithm);
- using a **data imputation** technique.

5.1.6 Data Imputation Techniques

One data imputation technique consists in replacing the missing value of a feature by an average value of this feature in the dataset:

$$\hat{x}^{(j)} \leftarrow \frac{1}{N} x^{(j)}.$$

Another technique is to replace the missing value with a value outside the normal range of values. For example, if the normal range is $[0, 1]$, then you can set the missing value to 2 or -1. The idea is that the learning algorithm will learn what is best to do when the feature has a value significantly different from regular values. Alternatively, you can replace the missing value by a value in the middle of the range. For example, if the range for a feature 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.

How many training examples do you have in your dataset? How many features does each example have? Some algorithms, including **neural networks** and **gradient boosting** (we consider both later), can handle a huge number of examples and millions of features. Others, like SVM, can be very modest in their capacity.

- Categorical vs. numerical features

Is your data composed of categorical only, or numerical only features, or a mix of both? Depending on your answer, some algorithms cannot handle your dataset directly, and you would need to convert your categorical features into numerical ones.

- Nonlinearity of the data

Is your data linearly separable or can it be modeled using a linear model? If yes, SVM with the linear kernel, logistic or linear regression can be good choices. Otherwise, deep neural networks or ensemble algorithms, discussed in Chapters 6 and 7, might work better.

- Training speed

How much time is a learning algorithm allowed to use to build a model? Neural networks are known to be slow to train. Simple algorithms like logistic and linear regression or decision trees are much faster. Specialized libraries contain very efficient implementations of some algorithms; you may prefer to do research online to find such libraries. Some algorithms, such as random forests, benefit from the availability of multiple CPU cores, so their model building time can be significantly reduced on a machine with dozens of cores.

- Prediction speed

How fast does the model have to be when generating predictions? Will your model be used in production where very high throughput is required? Algorithms like SVMs, linear and logistic regression, and (some types of) neural networks, are extremely fast at the prediction time. Others, like kNN, ensemble algorithms, and very deep or recurrent neural networks, are slower[2].

If you don't want to guess the best algorithm for your data, a popular way to choose one is by testing it on the **validation set**. We talk about that below. Alternatively, if you use scikit-learn, you could try their algorithm selection diagram shown in Figure 1.

5.3 Three Sets

Until now, I used the expressions "dataset" and "training set" interchangeably. However, in practice data analysts work with three distinct sets of labeled examples:

1) training set,
2) validation set, and

[2]The prediction speed of kNN and ensemble methods implemented in the modern libraries are still pretty fast. Don't be afraid of using these algorithms in your practice.

3) test set.

Once you have got your annotated dataset, the first thing you do is you shuffle the examples and split the dataset into three subsets: training, validation, and test. The training set is usually the biggest one; you use it to build the model. The validation and test sets are roughly the same sizes, much smaller than the size of the training set. The learning algorithm *cannot* use examples from these two subsets to build the model. That is why those two sets are often called **holdout sets**.

There's no optimal proportion to split the dataset into these three subsets. In the past, the rule of thumb was to use 70% of the dataset for training, 15% for validation and 15% for testing. However, in the age of big data, datasets often have millions of examples. In such cases, it could be reasonable to keep 95% for training and 2.5%/2.5% for validation/testing.

You may wonder, what is the reason to have three sets and not one. The answer is simple: when we build a model, what we do not want is for the model to only do well at predicting labels of examples the learning algorithms 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, but 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: we want good performance on a holdout set.

Why do we need two holdout sets and not one? We use the validation set to 1) choose the learning algorithm and 2) find the best values of hyperparameters. We use the test set to assess the model before delivering it to the client or putting it in production.

5.4 Underfitting and Overfitting

I mentioned above the notion of **bias**. I said that a model has a low bias if it predicts well the labels of the training data. If the model makes many mistakes on the training data, we say that the model has a **high bias** or that the model **underfits**. So, underfitting is the inability of the model to predict well the labels of the data it was trained on. There could be several reasons for underfitting, the most important of which are:

- your model is too simple for the data (for example a linear model can often underfit);
- the features you engineered are not informative enough.

The first reason is easy to illustrate in the case of one-dimensional regression: the dataset can resemble a curved line, but our model is a straight line. The second reason can be illustrated like this: let's say you want to predict whether a patient has cancer, and the features you have are height, blood pressure, and heart rate. These three features are clearly not good predictors for cancer so our model will not be able to learn a meaningful relationship between these features and the label.

The solution to the problem of underfitting is to try a more complex model or to engineer features with higher predictive power.

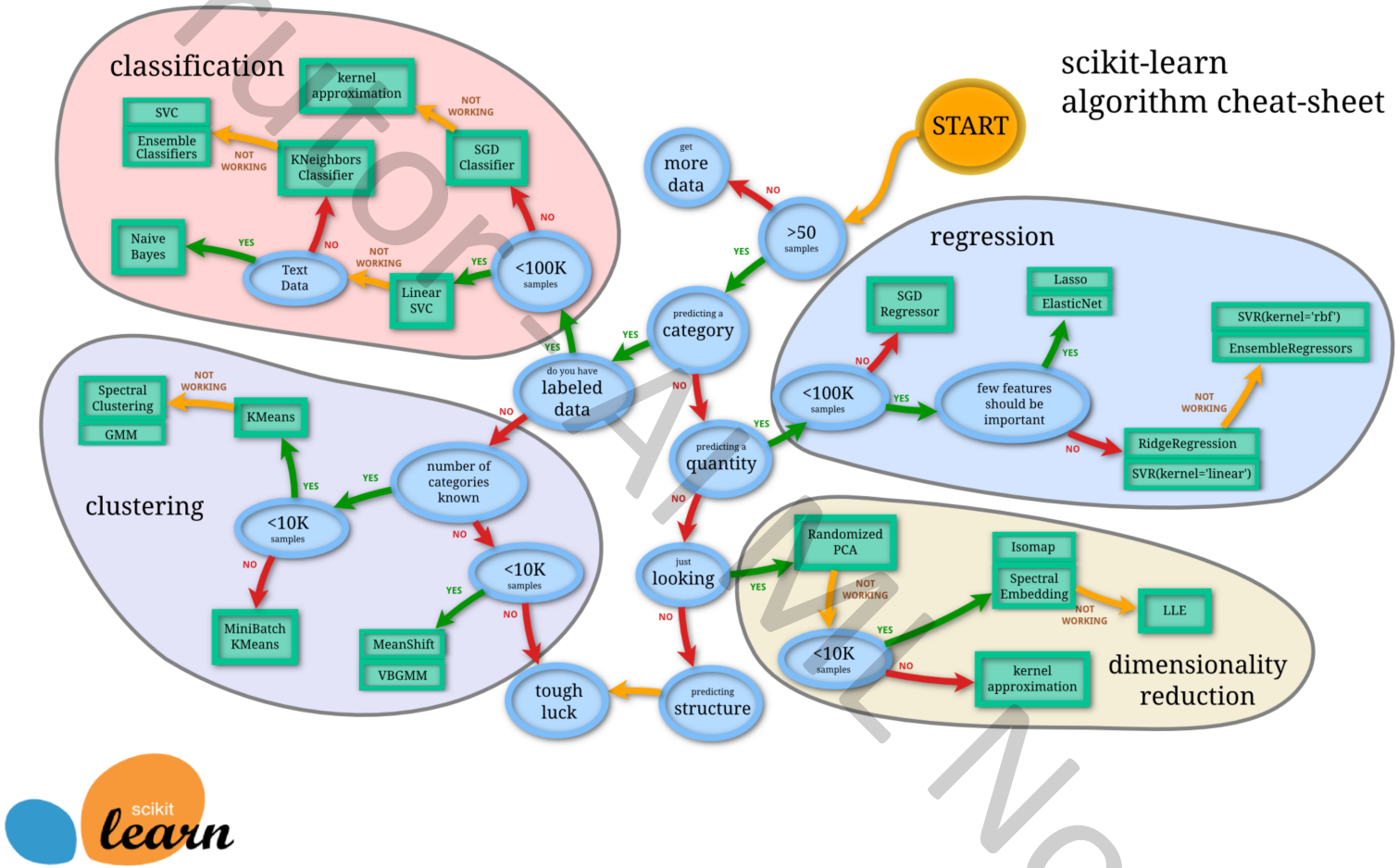


Figure 1: Machine learning algorithm selection diagram for scikit-learn.

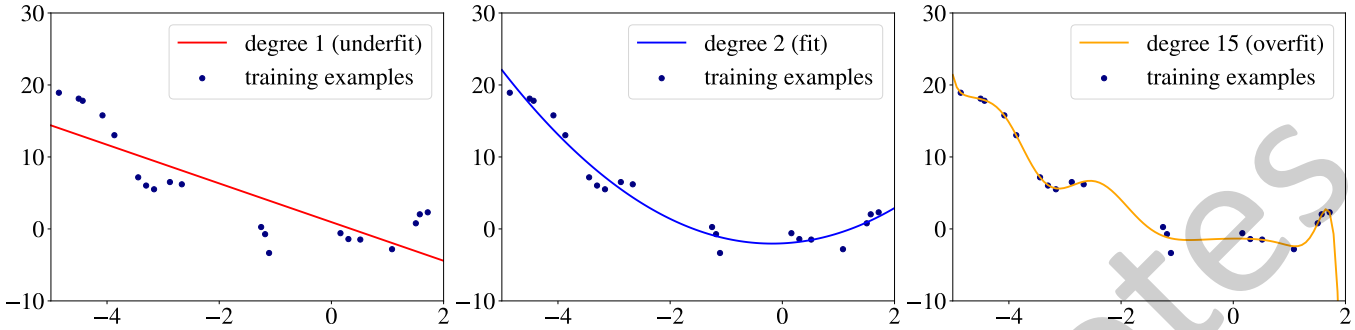


Figure 2: Examples of underfitting (linear model), good fit (quadratic model), and overfitting (polynomial of degree 15).

Overfitting is another problem a model can exhibit. The model that overfits predicts very well the training data but poorly the data from at least one of the two holdout sets. I already gave an illustration of overfitting in Chapter 3. Several reasons can lead to overfitting, the most important of which are:

- your model is too complex for the data (for example a very tall decision tree or a very deep or wide neural network often overfit);
- you have too many features but a small number of training examples.

In the literature, you can find another name for the problem of overfitting: the problem of **high variance**. This term comes from statistics. The variance is an error of the model due to its sensitivity to small fluctuations in the training set. It means that if your training data was sampled differently, the learning would result in a significantly different model. Which is why the model that overfits performs poorly on the test data: test and training data are sampled from the dataset independently of one another.

Even the simplest model, such as linear, can overfit the data. That usually happens when the data is high-dimensional, but the number of training examples is relatively low. In fact, when feature vectors are very high-dimensional, the linear learning algorithm can build a model that assigns non-zero values to most parameters $w^{(j)}$ in the parameter vector $\mathbf{w}$, trying to find very complex relationships between all available features to predict labels of training examples perfectly.

Such a complex model will most likely predict poorly the labels of the holdout examples. This is because by trying to perfectly predict labels of all training examples, the model will also learn the idiosyncrasies of the training set: the noise in the values of features of the training examples, the sampling imperfection due to the small dataset size, and other artifacts extrinsic to the decision problem at hand but present in the training set.

Figure 2 illustrates a one-dimensional dataset for which a regression model underfits, fits well and overfits the data.

Several solutions to the problem of overfitting are possible:

1. Try a simpler model (linear instead of polynomial regression, or SVM with a linear kernel instead of RBF, a neural network with fewer layers/units).
2. Reduce the dimensionality of examples in the dataset (for example, by using one of the dimensionality reduction techniques discussed in Chapter 9).
3. Add more training data, if possible.
4. Regularize the model.

Regularization is the most widely used approach to prevent overfitting.

5.5 Regularization

Regularization is an umbrella-term that encompasses methods that force the learning algorithm to build a less complex model. In practice, that often leads to slightly higher bias but significantly reduces the variance. This problem is known in the literature as the **bias-variance tradeoff**.

The two most widely used types of regularization are called **L1** and **L2 regularization**. The idea is quite simple. To create a regularized model, we modify the objective function by adding a penalizing term whose value is higher when the model is more complex.

For simplicity, I illustrate regularization using the example of linear regression. The same principle can be applied to a wide variety of models.

Recall the linear regression objective:

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

An L1-regularized objective looks like this:

$$\min_{\mathbf{w},b} \left[C|\mathbf{w}| + \frac{1}{N}\sum_{i=1}^{N}(f_{\mathbf{w},b}(\mathbf{x}_i) - y_i)^2 \right], \tag{3}$$

where $|\mathbf{w}| \overset{\text{def}}{=} \sum_{j=1}^{D} |w^{(j)}|$ and C is a hyperparameter that controls the importance of regularization. If we set C to zero, the model becomes a standard non-regularized linear regression model. On the other hand, if we set to C to a high value, the learning algorithm will try to set most $w^{(j)}$ to a very small value or zero to minimize the objective, the model will become very simple which can lead to underfitting. Your role as the data analyst is to find such a value of the hyperparameter C that doesn't increase the bias too much but reduces the variance to a level reasonable for the problem at hand. In the next section, I will show how to do that.

An L2-regularized objective looks like this:

$$\min_{\mathbf{w},b}\left[C\|\mathbf{w}\|^2+\frac{1}{N}\sum_{i=1}^{N}(f_{\mathbf{w},b}(\mathbf{x}_i)-y_i)^2\right],\ \text{where}\ \|\mathbf{w}\|^2\overset{\text{def}}{=}\sum_{j=1}^{D}(w^{(j)})^2. \tag{4}$$

In practice, L1 regularization produces a **sparse model**, a model that has most of its parameters (in case of linear models, most of $w^{(j)}$) equal to zero, provided the hyperparameter C is large enough. So L1 makes **feature selection** by deciding which features are essential for prediction and which are not. That can be useful in case you want to increase model explainability. However, if your only goal is to maximize the performance of the model on the holdout data, then L2 usually gives better results. L2 also has the advantage of being differentiable, so gradient descent can be used for optimizing the objective function.

L1 and L2 regularization methods were also combined in what is called **elastic net regularization** with L1 and L2 regularizations being special cases. You can find in the literature the name **ridge regularization** for L2 and **lasso** for L1.

In addition to being widely used with linear models, L1 and L2 regularization are also frequently used with neural networks and many other types of models, which directly minimize an objective function.

Neural networks also benefit from two other regularization techniques: **dropout** and **batch-normalization**. There are also non-mathematical methods that have a regularization effect: **data augmentation** and **early stopping**. We talk about these techniques in Chapter 8.

5.6 Model Performance Assessment

Once you have a model which our learning algorithm has built using the training set, how can you say how good the model is? You use the test set to assess the model.

The test set contains the examples that the learning algorithm has never seen before, so if our model performs well on predicting the labels of the examples from the test set, we say that our model *generalizes well* or, simply, that it's good.

To be more rigorous, machine learning specialists use various formal metrics and tools to assess the model performance. For regression, the assessment of the model is quite simple. A well-fitting regression model results in predicted values close to the observed data values. The **mean model**, which always predicts the average of the labels in the training data, generally would be used if there were no informative features. The fit of a regression model being assessed should, therefore, be better than the fit of the mean model. If this is the case, then the next step is to compare the performances of the model on the training and the test data.

To do that, we compute the mean squared error[3] (MSE) for the training, and, separately, for the test data. If the MSE of the model on the test data is *substantially higher* than

[3]Or any other type of average loss function that makes sense.

the MSE obtained on the training data, this is a sign of overfitting. Regularization or a better hyperparameter tuning could solve the problem. The meaning of "substantially higher" depends on the problem at hand and has to be decided by the data analyst jointly with the decision maker/product owner who ordered the model.

For classification, things are a little bit more complicated. The most widely used metrics and tools to assess the classification model are:

- confusion matrix,
- accuracy,
- cost-sensitive accuracy,
- precision/recall, and
- area under the ROC curve.

To simplify the illustration, I use a binary classification problem. Where necessary, I show how to extend the approach to the multiclass case.

5.6.1 Confusion Matrix

The **confusion matrix** is a table that summarizes how successful the classification model is at predicting examples belonging to various classes. One axis of the confusion matrix is the label that the model predicted, and the other axis is the actual label. In a binary classification problem, there are two classes. Let's say, the model predicts two classes: "spam" and "not_spam":

	spam (predicted)	not_spam (predicted)
spam (actual)	23 (TP)	1 (FN)
not_spam (actual)	12 (FP)	556 (TN)

The above confusion matrix shows that of the 24 examples that actually were spam, the model correctly classified 23 as spam. In this case, we say that we have 23 **true positives** or TP = 23. The model incorrectly classified 1 example as not_spam. In this case, we have 1 **false negative**, or FN = 1. Similarly, of 568 examples that actually were not spam, 556 were correctly classified (556 **true negatives** or TN = 556), and 12 were incorrectly classified (12 **false positives**, FP = 12).

The confusion matrix for multiclass classification has as many rows and columns as there are different classes. It can help you to determine mistake patterns. For example, a confusion matrix could reveal that a model trained to recognize different species of animals tends to mistakenly predict "cat" instead of "panther," or "mouse" instead of "rat." In this case, you can decide to add more labeled examples of these species to help the learning algorithm to "see" the difference between them. Alternatively, you might add additional features the learning algorithm can use to build a model that would better distinguish between these species.

Confusion matrix is used to calculate two other performance metrics: **precision** and **recall**.

5.6.2 Precision/Recall

The two most frequently used metrics to assess the model are **precision** and **recall**. Precision is the ratio of correct positive predictions to the overall number of positive predictions:

$$\text{precision} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Recall is the ratio of correct positive predictions to the overall number of positive examples in the dataset:

$$\text{recall} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

To understand the meaning and importance of precision and recall for the model assessment it is often useful to think about the prediction problem as the problem of research of documents in the database using a query. The precision is the proportion of relevant documents in the list of all returned documents. The recall is the ratio of the relevant documents returned by the search engine to the total number of the relevant documents that could have been returned.

In the case of the spam detection problem, we want to have high precision (we want to avoid making mistakes by detecting that a legitimate message is spam) and we are ready to tolerate lower recall (we tolerate some spam messages in our inbox).

Almost always, in practice, we have to choose between a high precision or a high recall. It's usually impossible to have both. We can achieve either of the two by various means:

- by assigning a higher weighting to the examples of a specific class (the SVM algorithm accepts weightings of classes as input);
- by tuning hyperparameters to maximize precision or recall on the validation set;
- by varying the decision threshold for algorithms that return probabilities of classes; for instance, if we use logistic regression or decision tree, to increase precision (at the cost of a lower recall), we can decide that the prediction will be positive only if the probability returned by the model is higher than 0.9.

Even if precision and recall are defined for the binary classification case, you can always use it to assess a multiclass classification model. To do that, first select a class for which you want to assess these metrics. Then you consider all examples of the selected class as positives and all examples of the remaining classes as negatives.

5.6.3 Accuracy

Accuracy is given by the number of correctly classified examples divided by the total number of classified examples. In terms of the confusion matrix, it is given by:

$$\text{accuracy} \stackrel{\text{def}}{=} \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \tag{5}$$

Accuracy is a useful metric when errors in predicting all classes are equally important. In case of the spam/not spam, this may not be the case. For example, you would tolerate false positives less than false negatives. A false positive in spam detection is the situation in which your friend sends you an email, but the model labels it as spam and doesn't show you. On the other hand, the false negative is less of a problem: if your model doesn't detect a small percentage of spam messages, it's not a big deal.

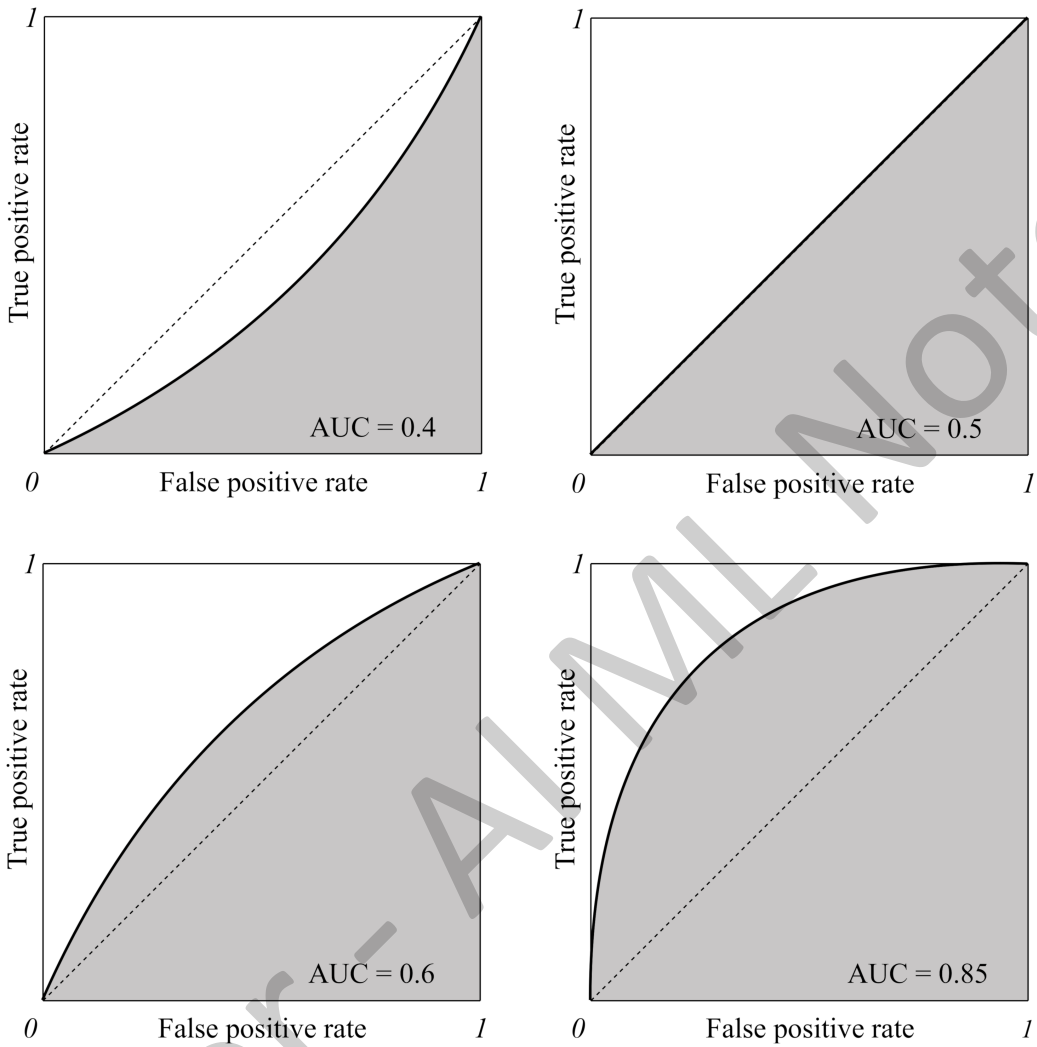


Figure 3: The area under the ROC curve (shown on gray).

5.6.4 Cost-Sensitive Accuracy

For dealing with the situation in which different classes have different importance, a useful metric is **cost-sensitive accuracy**. To compute a cost-sensitive accuracy, you first assign a cost (a positive number) to both types of mistakes: FP and FN. You then compute the counts TP, TN, FP, FN as usual and multiply the counts for FP and FN by the corresponding cost before calculating the accuracy using eq. 5.

5.6.5 Area under the ROC Curve (AUC)

The ROC curve (stands for "receiver operating characteristic," the term comes from radar engineering) is a commonly used method to assess the performance of classification models. ROC curves use a combination of the **true positive rate** (defined exactly as **recall**) and false positive rate (the proportion of negative examples predicted incorrectly) to build up a summary picture of the classification performance.

The true positive rate (TPR) and the false positive rate (FPR) are respectively defined as,

$$\text{TPR} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FN}} \text{ and } \text{FPR} \stackrel{\text{def}}{=} \frac{\text{FP}}{\text{FP} + \text{TN}}.$$

ROC curves can only be used to assess classifiers that return some confidence score (or a probability) of prediction. For example, logistic regression, neural networks, and decision trees (and ensemble models based on decision trees) can be assessed using ROC curves.

To draw a ROC curve, you first discretize the range of the confidence score. If this range for a model is $[0, 1]$, then you can discretize it like this: $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$. Then, you use each discrete value as the prediction threshold and predict the labels of examples in your dataset using the model and this threshold. For example, if you want to compute TPR and FPR for the threshold equal to 0.7, you apply the model to each example, get the score, and, if the score is higher than or equal to 0.7, you predict the positive class; otherwise, you predict the negative class.

Look at the illustration in Figure 3. It's easy to see that if the threshold is 0, all our predictions will be positive, so both TPR and FPR will be 1 (the upper right corner). On the other hand, if the threshold is 1, then no positive prediction will be made, both TPR and FPR will be 0 which corresponds to the lower left corner.

The higher the **area under the ROC curve** (AUC), the better the classifier. A classifier with an AUC higher than 0.5 is better than a random classifier. If AUC is lower than 0.5, then something is wrong with your model. A perfect classifier would have an AUC of 1. Usually, if your model behaves well, you obtain a good classifier by selecting the value of the threshold that gives TPR close to 1 while keeping FPR near 0.

ROC curves are popular because they are relatively simple to understand, they capture more than one aspect of the classification (by taking both false positives and negatives into account) and allow visually and with low effort comparing the performance of different models.

5.7 Hyperparameter Tuning

When I presented learning algorithms, I mentioned that you as a data analyst have to select good values for the algorithm's hyperparameters, such as ϵ and d for ID3, C for SVM, or α

6 Neural Networks and Deep Learning

First of all, you already know what a neural network is, and you already know how to build such a model. Yes, it's logistic regression! As a matter of fact, the logistic regression model, or rather its generalization for multiclass classification, called the softmax regression model, is a standard unit in a neural network.

6.1 Neural Networks

If you understood linear regression, logistic regression, and gradient descent, understanding neural networks should not be a problem.

A neural network (NN), just like a regression or an SVM model, is a mathematical function:

$$y = f_{NN}(\mathbf{x}).$$

The function f_{NN} has a particular form: it's a *nested function*. You have probably already heard of neural network **layers**. So, for a 3-layer neural network that returns a scalar, f_{NN} looks like this:

$$y = f_{NN}(\mathbf{x}) = f_3(\boldsymbol{f}_2(\boldsymbol{f}_1(\mathbf{x}))).$$

In the above equation, $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are vector functions of the following form:

$$\boldsymbol{f}_l(\mathbf{z}) \stackrel{\text{def}}{=} \boldsymbol{g}_l(\mathbf{W}_l\mathbf{z} + \mathbf{b}_l), \tag{1}$$

where l is called the layer index and can span from 1 to any number of layers. The function $\boldsymbol{g}_l$ is called an **activation function**. It is a fixed, usually nonlinear function chosen by the data analyst before the learning is started. The parameters $\mathbf{W}_l$ (a matrix) and $\mathbf{b}_l$ (a vector) for each layer are learned using the familiar gradient descent by optimizing, depending on the task, a particular cost function (such as MSE). Compare eq. 1 with the equation for logistic regression, where you replace $\boldsymbol{g}_l$ by the sigmoid function, and you will not see any difference. The function f_3 is a scalar function for the regression task, but can also be a vector function depending on your problem.

You may probably wonder why a matrix $\mathbf{W}_l$ is used and not a vector $\mathbf{w}_l$. The reason is that $\boldsymbol{g}_l$ is a vector function. Each row $\mathbf{w}_{l,u}$ (u for unit) of the matrix $\mathbf{W}_l$ is a vector of the same dimensionality as $\mathbf{z}$. Let $a_{l,u} = \mathbf{w}_{l,u}\mathbf{z} + b_{l,u}$. The output of $\boldsymbol{f}_l(\mathbf{z})$ is a vector $[g_l(a_{l,1}), g_l(a_{l,2}), \ldots, g_l(a_{l,size_l})]$, where g_l is some scalar function[1], and $size_l$ is the number of units in layer l. To make it more concrete, let's consider one architecture of neural networks called **multilayer perceptron** and often referred to as a **vanilla neural network**.

[1] A scalar function outputs a scalar, that is a simple number and not a vector.

6.1.1 Multilayer Perceptron Example

We have a closer look at one particular configuration of neural networks called **feed-forward neural networks** (FFNN), and more specifically the architecture called a **multilayer perceptron** (MLP). As an illustration, let's consider an MLP with three layers. Our network takes a two-dimensional feature vector as input and outputs a number. This FFNN can be a regression or a classification model, depending on the activation function used in the third, output layer.

Our MLP is depicted in Figure 1. The neural network is represented graphically as a connected combination of **units** logically organized into one or more **layers**. Each unit is represented by either a circle or a rectangle. The inbound arrow represents an input of a unit and indicates where this input came from. The outbound arrow indicates the output of a unit.

The output of each unit is the result of the mathematical operation written inside the rectangle. Circle units don't do anything with the input; they just send their input directly to the output.

The following happens in each rectangle unit. Firstly, all inputs of the unit are joined together to form an input vector. Then the unit applies a linear transformation to the input vector, exactly like linear regression model does with its input feature vector. Finally, the unit applies an activation function g to the result of the linear transformation and obtains the output value, a real number. In a vanilla FFNN, the output value of a unit of some layer becomes an input value of each of the units of the subsequent layer.

In Figure 1, the activation function g_l has one index: l, the index of the layer the unit belongs to. Usually, all units of a layer use the same activation function, but it's not a rule. Each layer can have a different number of units. Each unit has its parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$, where u is the index of the unit, and l is the index of the layer. The vector $\mathbf{y}_{l-1}$ in each unit is defined as $[y_{l-1}^{(1)}, y_{l-1}^{(2)}, y_{l-1}^{(3)}, y_{l-1}^{(4)}]$. The vector $\mathbf{x}$ in the first layer is defined as $[x^{(1)}, \ldots, x^{(D)}]$.

As you can see in Figure 1, in multilayer perceptron all outputs of one layer are connected to each input of the succeeding layer. This architecture is called **fully-connected**. A neural network can contain **fully-connected layers**. Those are the layers whose units receive as inputs the outputs of each of the units of the previous layer.

6.1.2 Feed-Forward Neural Network Architecture

If we want to solve a regression or a classification problem discussed in previous chapters, the last (the rightmost) layer of a neural network usually contains only one unit. If the activation function g_{last} of the last unit is linear, then the neural network is a regression model. If the g_{last} is a logistic function, the neural network is a binary classification model.

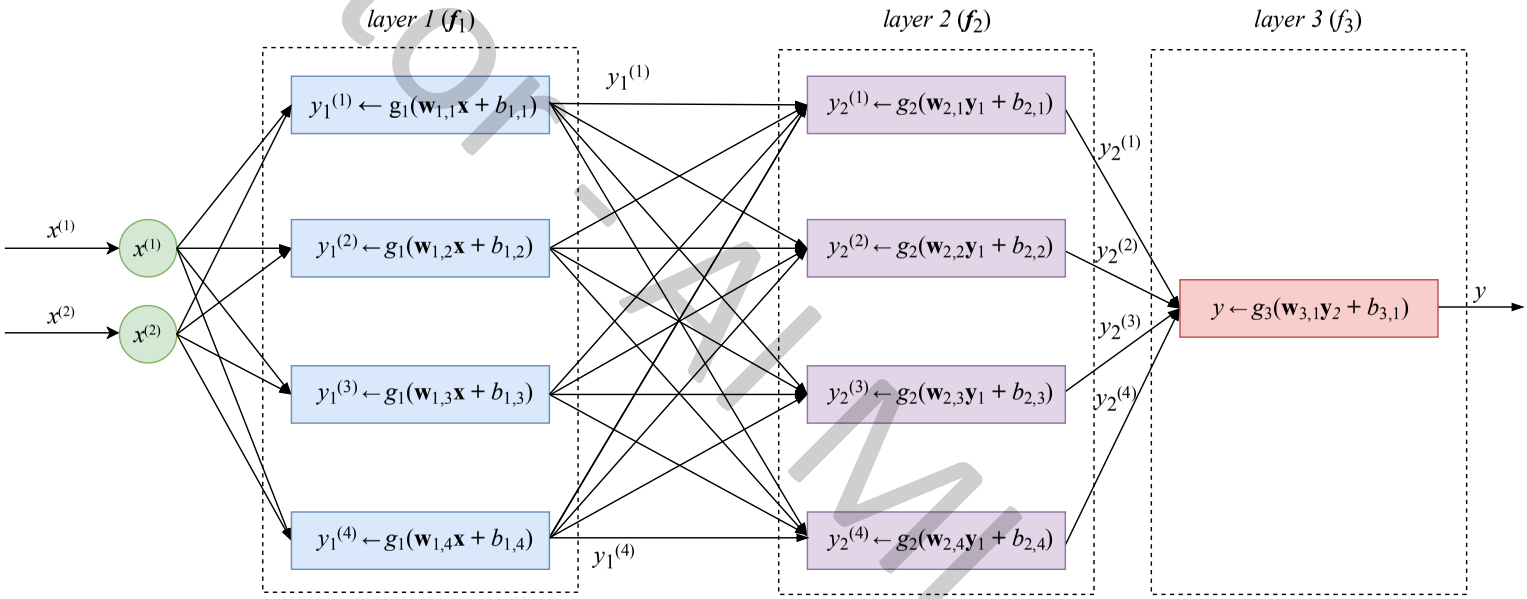


Figure 1: A multilayer perceptron with two-dimensional input, two layers with four units and one output layer with one unit.

The data analyst can choose any mathematical function as $g_{l,u}$, assuming it's differentiable[2]. The latter property is essential for gradient descent used to find the values of the parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$ for all l and u. The primary purpose of having nonlinear components in the function f_{NN} is to allow the neural network to approximate nonlinear functions. Without nonlinearities, f_{NN} would be linear, no matter how many layers it has. The reason is that $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$ is a linear function and a linear function of a linear function is also linear.

Popular choices of activation functions are the logistic function, already known to you, as well as **TanH** and **ReLU**. The former is the hyperbolic tangent function, similar to the logistic function but ranging from -1 to 1 (without reaching them). The latter is the rectified linear unit function, which equals to zero when its input z is negative and to z otherwise:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

$$relu(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}.$$

As I said above, $\mathbf{W}_l$ in the expression $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$, is a matrix, while $\mathbf{b}_l$ is a vector. That looks different from linear regression's $\mathbf{w}\mathbf{z} + b$. In matrix $\mathbf{W}_l$, each row u corresponds to a vector of parameters $\mathbf{w}_{l,u}$. The dimensionality of the vector $\mathbf{w}_{l,u}$ equals to the number of units in the layer $l - 1$. The operation $\mathbf{W}_l\mathbf{z}$ results in a vector $\mathbf{a}_l \stackrel{\text{def}}{=} [\mathbf{w}_{l,1}\mathbf{z}, \mathbf{w}_{l,2}\mathbf{z}, \ldots, \mathbf{w}_{l,size_l}\mathbf{z}]$. Then the sum $\mathbf{a}_l + \mathbf{b}_l$ gives a $size_l$-dimensional vector $\mathbf{c}_l$. Finally, the function $\boldsymbol{g}_l(\mathbf{c}_l)$ produces the vector $\mathbf{y}_l \stackrel{\text{def}}{=} [y_l^{(1)}, y_l^{(2)}, \ldots, y_l^{(size_l)}]$ as output.

6.2 Deep Learning

Deep learning refers to training neural networks with more than two non-output layers. In the past, it became more difficult to train such networks as the number of layers grew. The two biggest challenges were referred to as the problems of **exploding gradient** and **vanishing gradient** as gradient descent was used to train the network parameters.

While the problem of exploding gradient was easier to deal with by applying simple techniques like **gradient clipping** and L1 or L2 regularization, the problem of vanishing gradient remained intractable for decades.

What is vanishing gradient and why does it arise? To update the values of the parameters in neural networks the algorithm called **backpropagation** is typically used. Backpropagation is an efficient algorithm for computing gradients on neural networks using the chain rule. In Chapter 4, we have already seen how the chain rule is used to calculate partial derivatives of a complex function. During gradient descent, the neural network's parameters receive an

[2]The function has to be differentiable across its whole domain or in the majority of the points of its domain. For example, ReLU is not differentiable at 0.

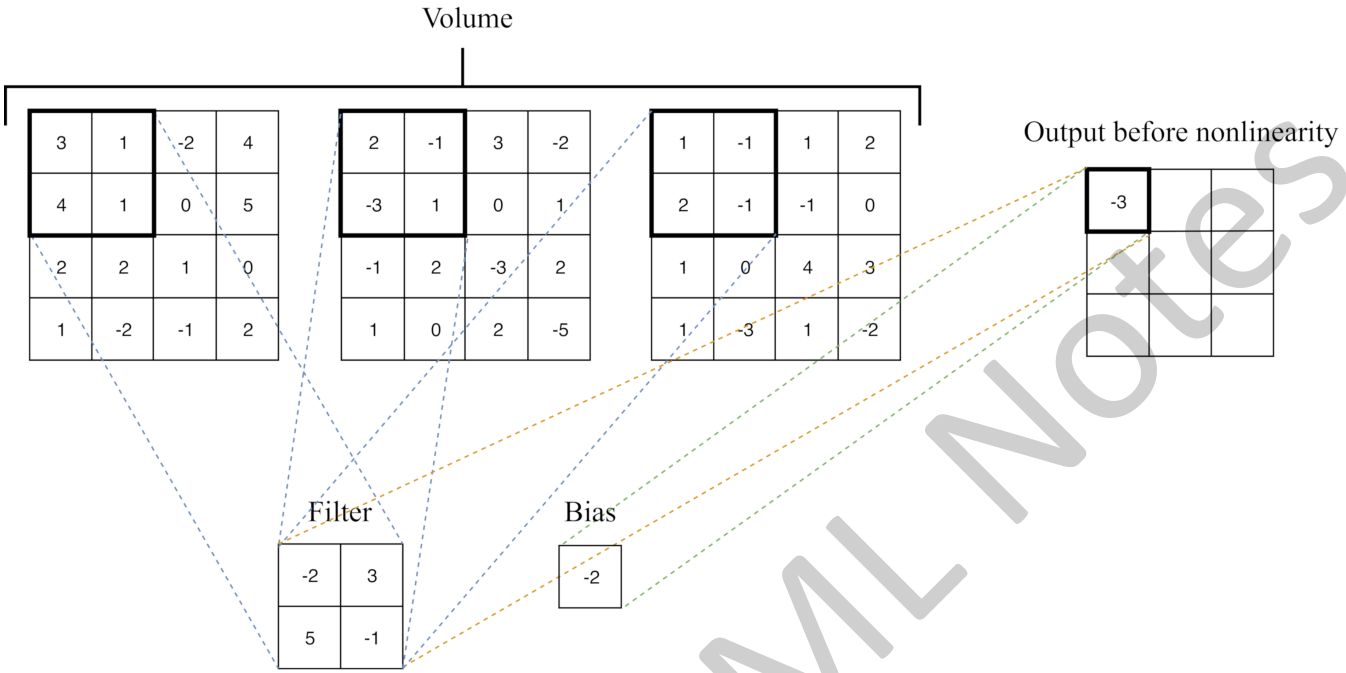


Figure 4: Convolution of a volume consisting of three matrices.

An example of a convolution of a patch of a volume consisting of depth 3 is shown in Figure 4. The value of the convolution, -3, was obtained as $(-2 \cdot 3 + 3 \cdot 1 + 5 \cdot 4 + -1 \cdot 1) + (-2 \cdot 2 + 3 \cdot (-1) + 5 \cdot (-3) + -1 \cdot 1) + (-2 \cdot 1 + 3 \cdot (-1) + 5 \cdot 2 + -1 \cdot (-1)) + (-2)$.

In computer vision, CNNs often get volumes as input, since an image is usually represented by three channels: R, G, and B, each channel being a monochrome picture.

Two important properties of convolution are **stride** and **padding**. Stride is the step size of the moving window. In Figure 3, the stride is 1, that is the filter slides to the right and to the bottom by one cell at a time. In Figure 5, you can see a partial example of convolution with stride 2. You can see that the output matrix is smaller when stride is bigger.

Padding allows getting a larger output matrix; it's the width of the square of additional cells with which you surround the image (or volume) before you convolve it with the filter. The cells added by padding usually contain zeroes. In Figure 3, the padding is 0, so no additional cells are added to the image. In Figure 6, on the other hand, the stride is 2 and padding is 1, so a square of width 1 of additional cells are added to the image. You can see that the output matrix is bigger when padding is bigger[5].

[5]To save space, in Figure 6, only the first two of the nine convolutions are shown.

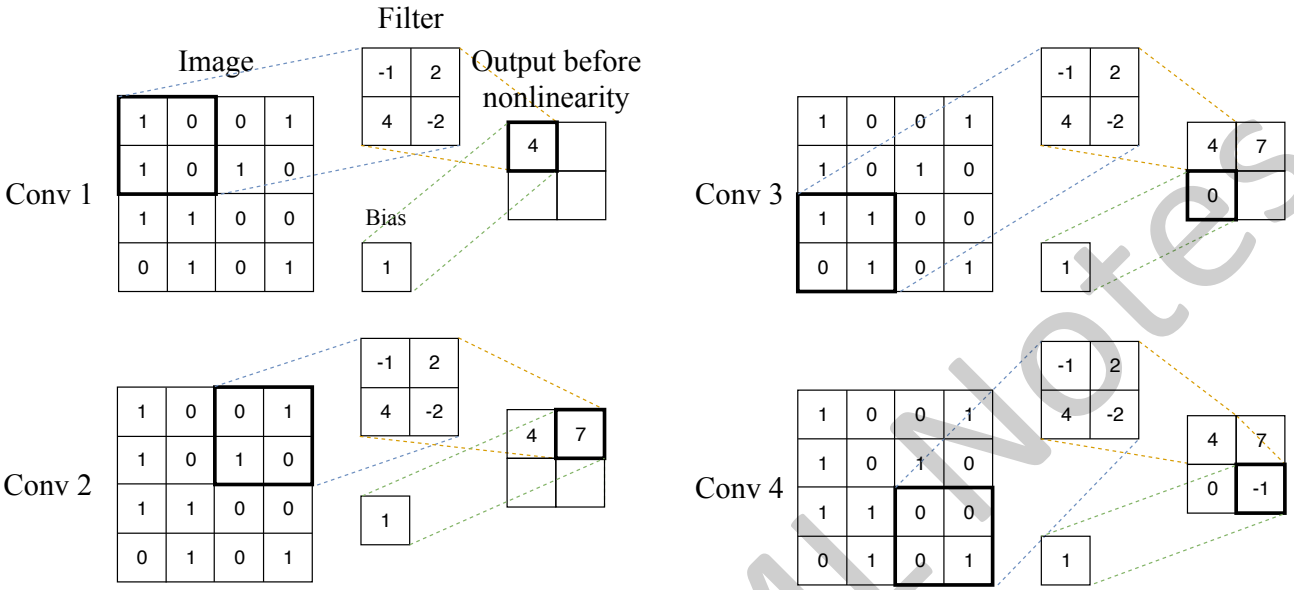


Figure 5: Convolution with stride 2.

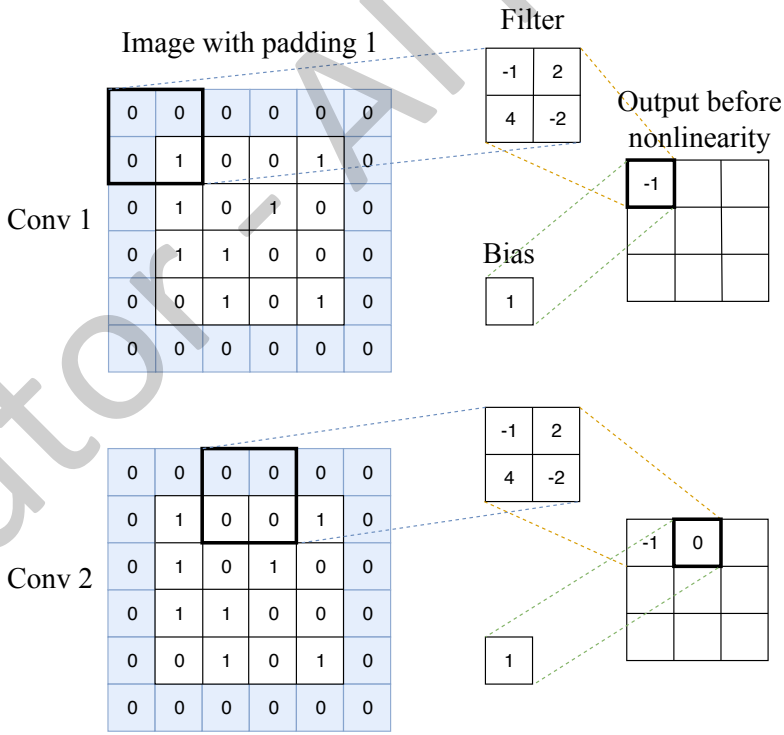


Figure 6: Convolution with stride 2 and padding 1.

An example of an image with padding 2 is shown in Figure 7. Padding is helpful with larger filters because it allows them to better "scan" the boundaries of the image.

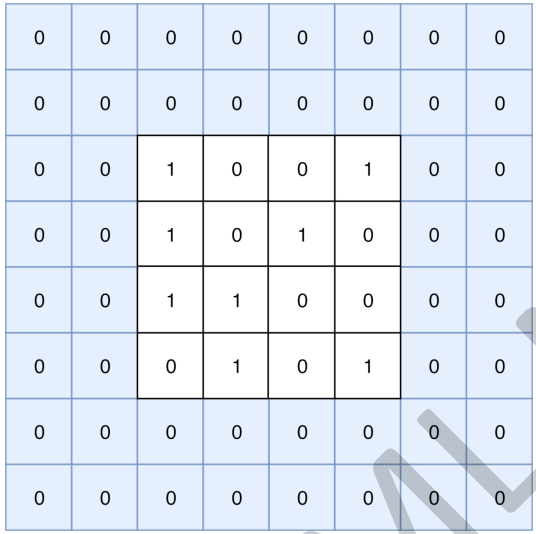


Figure 7: Image with padding 2.

This section would not be complete without presenting **pooling**, a technique very often used in CNNs. Pooling works in a way very similar to convolution, as a filter applied using a moving window approach. However, instead of applying a trainable filter to an input matrix or a volume, pooling layer applies a fixed operator, usually either max or average. Similarly to convolution, pooling has hyperparameters: the size of the filter and the stride. An example of max pooling with filter of size 2 and stride 2 is shown in Figure 8.

Usually, a pooling layer follows a convolution layer, and it gets the output of convolution as input. When pooling is applied to a volume, each matrix in the volume is processed independently of others. Therefore, the output of the pooling layer applied to a volume is a volume of the same depth as the input.

As you can see, pooling only has hyperparameters and doesn't have parameters to learn. Typically, the filter of size 2 or 3 and stride 2 are used in practice. Max pooling is more popular than average and often gives better results.

Typically pooling contributes to the increased accuracy of the model. It also improves the speed of training by reducing the number of parameters of the neural network. (As you can see in Figure 8, with filter size 2 and stride 2 the number of parameters is reduced to 25%, that is to 4 parameters instead of 16.)

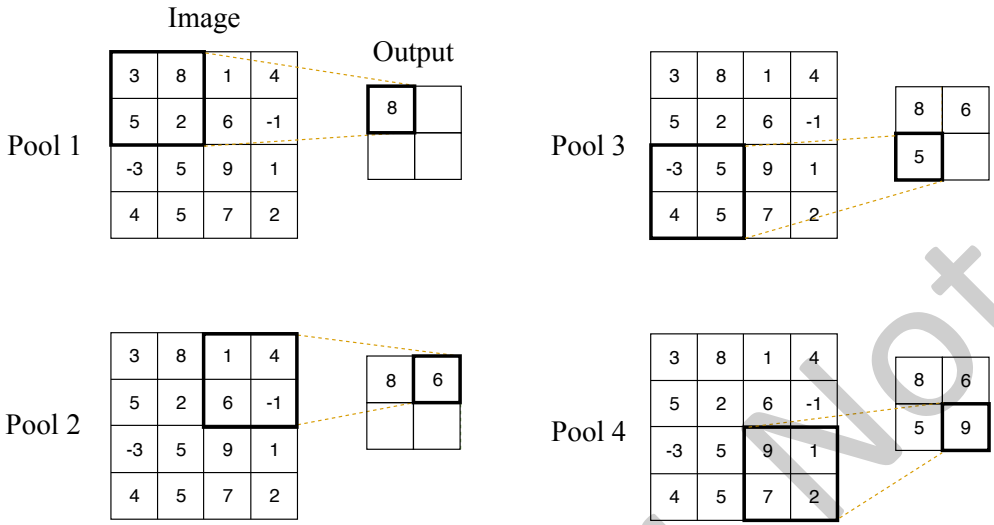


Figure 8: Pooling with filter of size 2 and stride 2.

6.2.2 Recurrent Neural Network

Recurrent neural networks (RNNs) are used to label, classify, or generate sequences. A sequence is a matrix, each row of which is a feature vector and the order of rows matters. To label a sequence is to predict a class for each feature vector in a sequence. To classify a sequence is to predict a class for the entire sequence. To generate a sequence is to output another sequence (of a possibly different length) somehow relevant to the input sequence.

RNNs are often used in text processing because sentences and texts are naturally sequences of either words/punctuation marks or sequences of characters. For the same reason, recurrent neural networks are also used in speech processing.

A recurrent neural network is not feed-forward: it contains loops. The idea is that each unit u of recurrent layer l has a real-valued **state** $h_{l,u}$. The state can be seen as the memory of the unit. In RNN, each unit u in each layer l receives two inputs: a vector of states from the previous layer $l-1$ and the vector of states from this same layer l from *the previous time step*.

To illustrate the idea, let's consider the first and the second recurrent layers of an RNN. The first (leftmost) layer receives a feature vector as input. The second layer receives the output of the first layer as input.

This situation is schematically depicted in Figure 9. As I said above, each training example is a matrix in which each row is a feature vector. For simplicity, let's illustrate this matrix as a sequence of vectors $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{t-1}, \mathbf{x}^t, \mathbf{x}^{t+1}, \ldots, \mathbf{x}^{length_\mathbf{X}}]$, where $length_\mathbf{X}$ is the length of the input sequence. If our input example $\mathbf{X}$ is a text sentence, then feature vector $\mathbf{x}^t$ for each $t = 1, \ldots, length_\mathbf{X}$ represents a word in the sentence at position t.

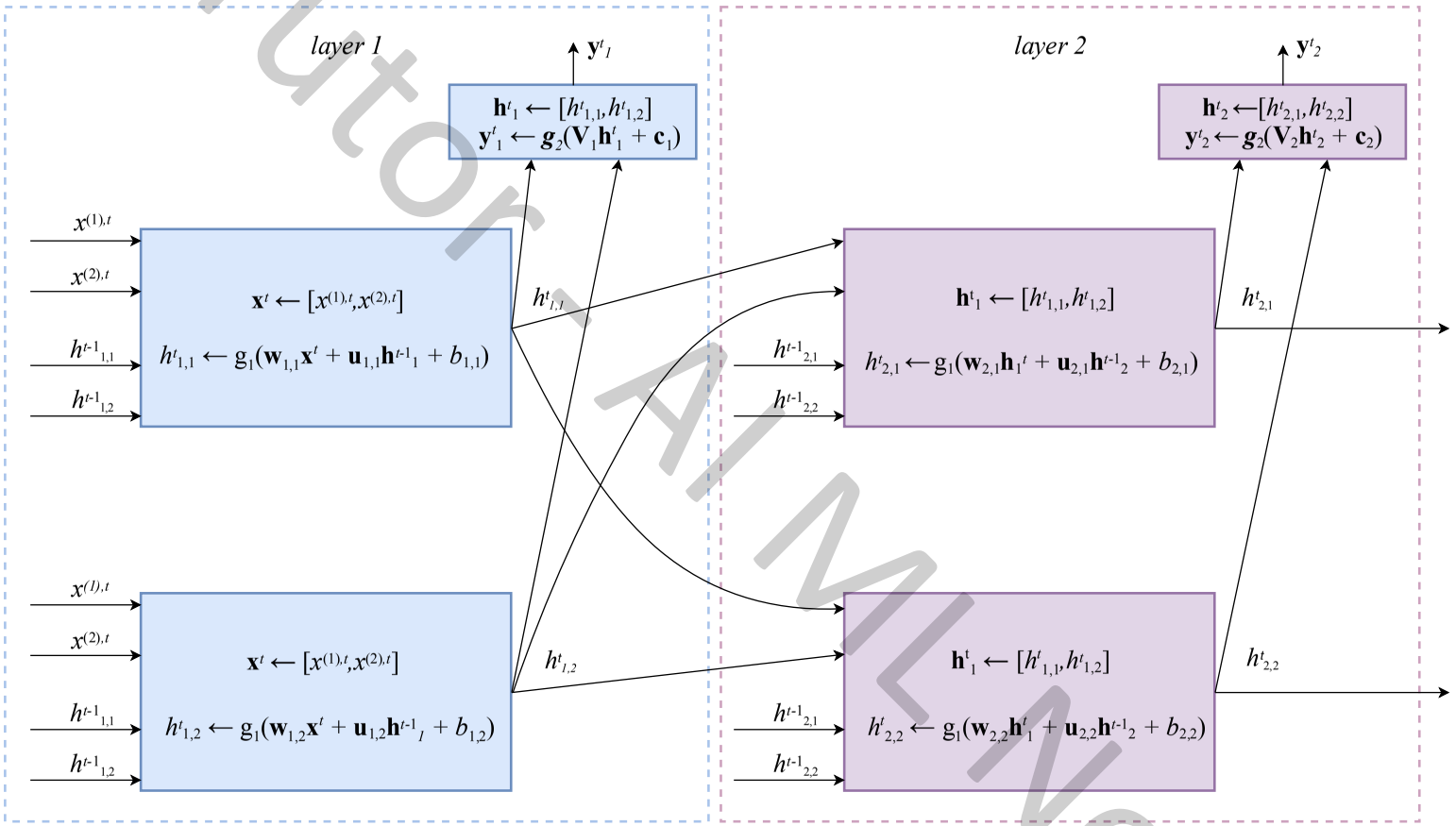


Figure 9: The first two layers of an RNN. The input feature vector is two-dimensional; each layer has two units.

As depicted in Figure 9, in an RNN, the feature vectors from an input example are "read" by the neural network sequentially in the order of the timesteps. The index t denotes a timestep. To update the state $h_{l,u}^t$ at each timestep t in each unit u of each layer l we first calculate a linear combination of the input feature vector with the state vector $\mathbf{h}_{l,u}^{t-1}$ of this same layer from the previous timestep, $t-1$. The linear combination of two vectors is calculated using two parameter vectors $\mathbf{w}_{l,u}$, $\mathbf{u}_{l,u}$ and a parameter $b_{l,u}$. The value of $h_{l,u}^t$ is then obtained by applying activation function g_1 to the result of the linear combination. A typical choice for function g_1 is *tanh*. The output $\mathbf{y}_l^t$ is typically a vector calculated for the whole layer l at once. To obtain $\mathbf{y}_l^t$, we use activation function $\boldsymbol{g}_2$ that takes a vector as input and returns a different vector of the same dimensionality. The function $\boldsymbol{g}_2$ is applied to a linear combination of the state vector values $\mathbf{h}_{l,u}^t$ calculated using a parameter matrix $\mathbf{V}_l$ and a parameter vector $\mathbf{c}_{l,u}$. In classification, a typical choice for $\boldsymbol{g}_2$ is the **softmax function**:

$$\boldsymbol{\sigma}(\mathbf{z}) \stackrel{\text{def}}{=} [\sigma^{(1)}, \ldots, \sigma^{(D)}], \text{ where } \sigma^{(j)} \stackrel{\text{def}}{=} \frac{\exp\left(z^{(j)}\right)}{\sum_{k=1}^{D} \exp\left(z^{(k)}\right)}.$$

The softmax function is a generalization of the sigmoid function to multidimensional outputs. It has the property that $\sum_{j=1}^{D} \sigma^{(j)} = 1$ and $\sigma^{(j)} > 0$ for all j.

The dimensionality of $\mathbf{V}_l$ is chosen by the data analyst such that multiplication of matrix $\mathbf{V}_l$ by the vector $\mathbf{h}_l^t$ results in a vector of the same dimensionality as that of the vector $\mathbf{c}_l$. This choice depends on the dimensionality for the output label $\mathbf{y}$ in your training data. (Until now we only saw one-dimensional labels, but we will see in the future chapters that labels can be multidimensional as well.)

The values of $\mathbf{w}_{l,u}$, $\mathbf{u}_{l,u}$, $b_{l,u}$, $\mathbf{V}_{l,u}$, and $\mathbf{c}_{l,u}$ are computed from the training data using gradient descent with backpropagation. To train RNN models, a special version of backpropagation is used called **backpropagation through time**.

Both *tanh* and *softmax* suffer from the vanishing gradient problem. Even if our RNN has just one or two recurrent layers, because of the sequential nature of the input, backpropagation has to "unfold" the network over time. From the point of view of the gradient calculation, in practice this means that the longer is the input sequence, the deeper is the unfolded network.

Another problem RNNs have is that of handling long-term dependencies. As the length of the input sequence grows, the feature vectors from the beginning of the sequence tend to be "forgotten," because the state of each unit, which serves as network's memory, becomes significantly affected by the feature vectors read more recently. Therefore, in text or speech processing, the cause-effect link between distant words in a long sentence can be lost.

The most effective recurrent neural network models used in practice are **gated RNNs**. These include the **long short-term memory** (LSTM) networks and networks based on the **gated recurrent unit** (GRU).

The beauty of using gated units in RNNs is that such networks can store information in their units for future use, much like bits in a computer's memory. The difference with the real

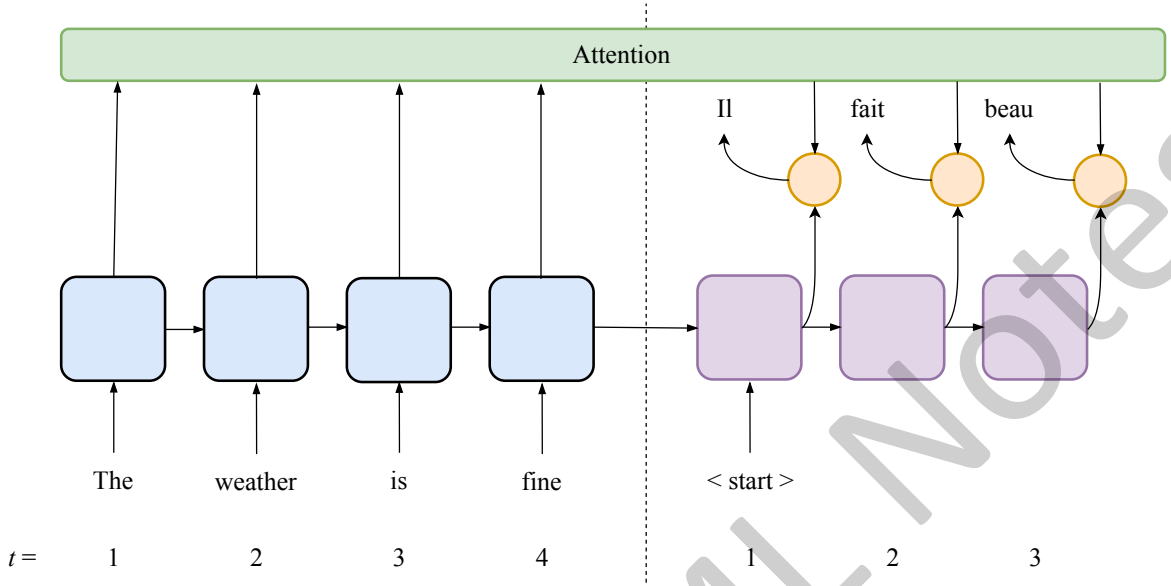


Figure 5: A seq2seq architecture with attention.

A seq2seq architecture with attention is illustrated in Figure 5.

seq2seq learning is a relatively new research domain. Novel network architectures are regularly discovered and published. Training such architectures can be challenging as the number of hyperparameters to tune and other architectural decisions can be overwhelming. Consult the book's wiki for the state of the art material, tutorials and code samples.

7.8 Active Learning

Active learning is an interesting supervised learning paradigm. It is usually applied when obtaining labeled examples is costly. That is often the case in the medical or financial domains, where the opinion of an expert may be required to annotate patients' or customers' data. The idea is to start learning with relatively few labeled examples, and a large number of unlabeled ones, and then label only those examples that contribute the most to the model quality.

There are multiple strategies of active learning. Here, we discuss only the following two:

1) data density and uncertainty based, and
2) support vector-based.

The former strategy applies the current model f, trained using the existing labeled examples, to each of the remaining unlabelled examples (or, to save the computing time, to some

random sample of them). For each unlabeled example $\mathbf{x}$, the following importance score is computed: $density(\mathbf{x}) \cdot uncertainty_f(\mathbf{x})$. Density reflects how many examples surround $\mathbf{x}$ in its close neighborhood, while $uncertainty_f(\mathbf{x})$ reflects how uncertain the prediction of the model f is for $\mathbf{x}$. In binary classification with sigmoid, the closer the prediction score is to 0.5, the more uncertain is the prediction. In SVM, the closer the example is to the decision boundary, the most uncertain is the prediction.

In multiclass classification, *entropy* can be used as a typical measure of uncertainty:

$$\mathrm{H}_f(\mathbf{x}) = -\sum_{c=1}^{C} \Pr(y^{(c)}; f(\mathbf{x})) \ln \left[\Pr(y^{(c)}; f(\mathbf{x}))\right],$$

where $\Pr(y^{(c)}; f(\mathbf{x}))$ is the probability score the model f assigns to class $y^{(c)}$ when classifying $\mathbf{x}$. You can see that if for each $y^{(c)}$, $f(y^{(c)}) = \frac{1}{C}$ then the model is the most uncertain and the entropy is at its maximum of 1; on the other hand, if for some $y^{(c)}$, $f(y^{(c)}) = 1$, then the model is certain about the class $y^{(c)}$ and the entropy is at its minimum of 0.

Density for the example $\mathbf{x}$ can be obtained by taking the average of the distance from $\mathbf{x}$ to each of its k nearest neighbors (with k being a hyperparameter).

Once we know the importance score of each unlabeled example, we pick the one with the highest importance score and ask the expert to annotate it. Then we add the new annotated example to the training set, rebuild the model and continue the process until some stopping criterion is satisfied. A stopping criterion can be chosen in advance (the maximum number of requests to the expert based on the available budget) or depend on how well our model performs according to some metric.

The support vector-based active learning strategy consists in building an SVM model using the labeled data. We then ask our expert to annotate the unlabeled example that lies the closest to the hyperplane that separates the two classes. The idea is that if the example lies closest to the hyperplane, then it is the least certain and would contribute the most to the reduction of possible places where the true (the one we look for) hyperplane could lie.

Some active learning strategies can incorporate the cost of asking an expert for a label. Others *learn* to ask expert's opinion. The "query by committee" strategy consists of training multiple models using different methods and then asking an expert to label example on which those models disagree the most. Some strategies try to select examples to label so that the variance or the bias of the model are reduced the most.

8 Advanced Practice

This chapter contains the description of techniques that you could find useful in your practice in some contexts. It's called "Advanced Practice" not because the presented techniques are more complex, but rather because they are applied in some very specific contexts. In many practical situations, you will most likely not need to resort to using these techniques, but sometimes they are very helpful.

8.1 Handling Imbalanced Datasets

Often in practice, examples of some class will be underrepresented in your 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. If you use SVM with soft margin, you can define a cost for misclassified examples. Because noise is always present in the training data, there are high chances that many examples of genuine transactions would end up on the wrong side of the decision boundary by contributing to the cost.

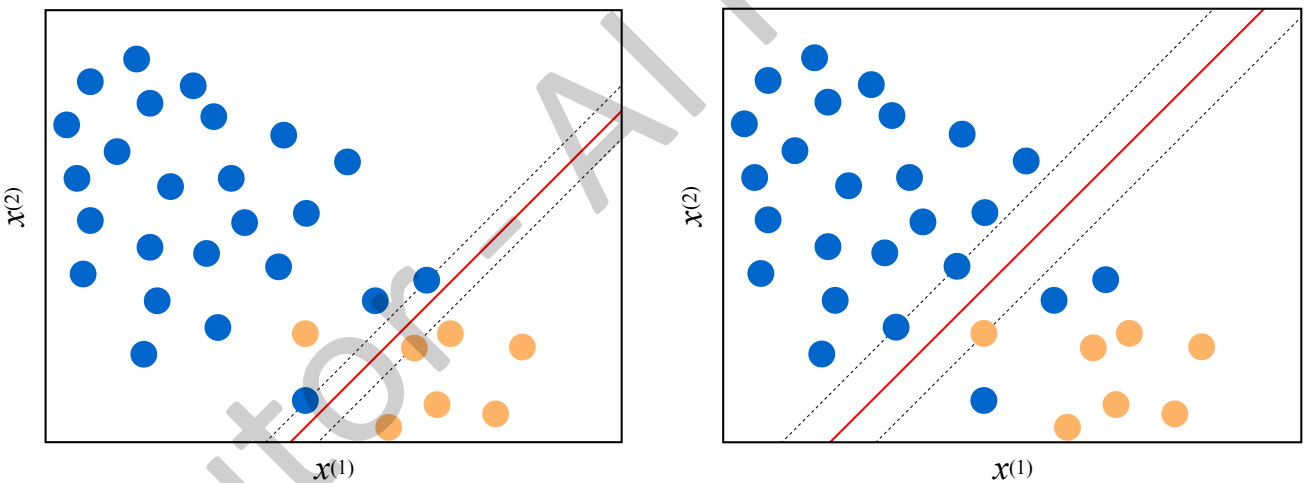


Figure 1: An illustration of an imbalanced problem. (a) Both classes have the same weight; (b) examples of the minority class have a higher weight.

The SVM algorithm will try to move the hyperplane to avoid as much as possible misclassified examples. The "fraudulent" examples, which are in the minority, risk being misclassified in order to classify more numerous examples of the majority class correctly. This situation is illustrated in Figure 1a. This problem is observed for most learning algorithms applied to imbalanced datasets.

If you set the cost of misclassification of examples of the minority class higher, then the model will try harder to avoid misclassifying those examples, obviously for the cost of misclassification of some examples of the majority class, as illustrated in Figure 1b.

Some SVM implementations allow you to provide weights for every class. The learning algorithm takes this information into account when looking for the best hyperplane.

If a learning algorithm doesn't allow weighting classes, you can try the technique of **oversampling**. It consists of increasing the importance of examples of some class by making multiple copies of the examples of that class.

An opposite approach, **undersampling**, is to randomly remove from the training set some examples of the majority class.

You might also try to create synthetic examples by randomly sampling feature values of several examples of the minority class and combining them to obtain a new example of that class. There are two popular algorithms that oversample the minority class by creating synthetic examples: the **synthetic minority oversampling technique** (SMOTE) and the **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 of this example (let's denote this set of k examples $\mathcal{S}_k$) and then create a synthetic example $\mathbf{x}_{new}$ 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 a random number in the range $[0, 1]$.

Both SMOTE and ADASYN randomly pick 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 examples of the minority class are rare.

Some algorithms are less sensitive to the problem of an imbalanced dataset. Decision trees, as well as random forest and gradient boosting, often perform well on imbalanced datasets.

8.2 Combining Models

Ensemble algorithms, like Random Forest, typically combine models of the same nature. They boost performance by combining hundreds of weak models. In practice, we can sometimes get an additional performance gain by combining strong models made with different learning algorithms. In this case, we usually use only two or three models.

Three typical ways to combine models are 1) averaging, 2) majority vote and 3) stacking.

Averaging works for regression as well as those classification models that return classification scores. You simply apply all your models, let's call them *base models*, to the input $\mathbf{x}$ and then average the predictions. To see if the averaged model works better than each individual algorithm, you test it on the validation set using a metric of your choice.

Majority vote works for classification models. You apply all your base models to the input $\mathbf{x}$ and then return the majority class among all predictions. In the case of a tie, you either randomly pick one of the classes, or, you return an error message (if the fact of misclassifying would incur a significant cost).

Stacking consists of building a meta-model that takes the output of base models as input. Let's say you want to combine classifiers f_1 and f_2, both predicting the same set of classes. To create a training example $(\hat{\mathbf{x}}_i, \hat{y}_i)$ for the stacked model, set $\hat{\mathbf{x}}_i = [f_1(\mathbf{x}), f_2(\mathbf{x})]$ and $\hat{y}_i = y_i$.

If some of your base models return not just a class, but also a score for each class, you can use these values as features too.

To train the stacked model, it is recommended to use examples from the training set and tune the hyperparameters of the stacked model using cross-validation.

Obviously, you have to make sure that your stacked model performs better on the validation set than each of the base models you stacked.

The reason that combining multiple models can bring better performance is the observation that when several uncorrelated strong models agree they are more likely to agree on the correct outcome. The keyword here is "uncorrelated." Ideally, base models should be obtained using different features or using algorithms of a different nature — for example, SVMs and Random Forest. Combining different versions of decision tree learning algorithm, or several SVMs with different hyperparameters may not result in a significant performance boost.

8.3 Training Neural Networks

In neural network training, one challenging aspect is to convert your data into the input the network can work with. If your input is images, first of all, you have to resize all images so that they have the same dimensions. After that, pixels are usually first standardized and then normalized to the range $[0, 1]$.

Texts have to be tokenized (that is split into pieces, such as words, punctuation marks, and other symbols). For CNN and RNN, each token is converted into a vector using the one-hot encoding, so the text becomes a list of one-hot vectors. Another, often a better way to represent tokens is by using **word embeddings**. For multilayer perceptron, to convert texts to vectors the bag of words approach may work well, especially for larger texts (larger than SMS messages and tweets).

The choice of specific neural network architecture is a difficult one. For the same problem, like seq2seq learning, there is a variety of architectures, and new ones are proposed almost every year. I recommend researching state of the art solutions for your problem using Google Scholar or Microsoft Academic search engines that allow searching for scientific publications using keywords and time range. If you don't mind working with less modern architecture, I recommend looking for implemented architectures on GitHub and finding one that could be applied to your data with minor modifications.

In practice, the advantage of modern architecture over an older one becomes less significant as you preprocess, clean and normalize your data, and create a larger training set. Modern neural network architectures are a result of the collaboration of scientists from several labs and companies; such models could be very complex to implement on your own and usually require much computational power to train. Time spent trying to replicate results from a recent scientific paper may not be worth it. This time could better be spent on building the solution around a less modern but stable model and getting more training data.

Once you decided on the architecture of your network, you have to decide on the number of layers, their type, and size. It is recommended to start with one or two layers, train a model and see if it fits the training data well (has a low bias). If not, gradually increase the size of each layer and the number of layers until the model perfectly fits the training data. Once this is the case, if the model doesn't perform well on the validation data (has a high variance), you should add regularization to your model. If, after adding regularization, the model doesn't fit the training data anymore, slightly increase the size of the network. Continue iteratively until the model fits both training and validation data well enough according to your metric.

8.4 Advanced Regularization

In neural networks, besides L1 and L2 regularization, you can use neural network specific regularizers: **dropout**, **early stopping**, and **batch normalization**. The latter is technically not a regularization technique, but it often has a regularization effect on the model.

The concept of dropout is very simple. Each time you run a training example through the network, you temporarily exclude at random some units from the computation. The higher the percentage of units excluded the higher the regularization effect. Neural network libraries allow you to add a dropout layer between two successive layers, or you can specify the dropout parameter for the layer. The dropout parameter is in the range $[0, 1]$ and it has to be found experimentally by tuning it on the validation data.

Early stopping is the way to train a neural network by saving the preliminary model after every epoch and assessing the performance of the preliminary model on the validation set. As you remember from the section about gradient descent in Chapter 4, as the number of epochs increases, the cost decreases. The decreased cost means that the model fits the training data well. However, at some point, after some epoch e, the model can start overfitting: the cost keeps decreasing, but the performance of the model on the validation data deteriorates. If you keep, in a file, the version of the model after each epoch, you can stop the training once you start observing a decreased performance on the validation set. Alternatively, you can keep running the training process for a fixed number of epochs and then, in the end, you pick the best model. Models saved after each epoch are called *checkpoints*. Some machine learning practitioners rely on this technique very often; others try to properly regularize the model to avoid such an undesirable behavior.

Batch normalization (which rather has to be called batch standardization) is a technique that consists of standardizing the outputs of each layer before the units of the subsequent

layer receive them as input. In practice, batch normalization results in faster and more stable training, as well as some regularization effect. So it's always a good idea to try to use batch normalization. In neural network libraries, you can often insert a batch normalization layer between two layers.

Another regularization technique that can be applied not just to neural networks, but to virtually any learning algorithm, is called **data augmentation**. This technique is often used to regularize models that work with images. Once you have your original labeled training set, you can create a synthetic example from an original example by applying various transformations to the original image: zooming it slightly, rotating, flipping, darkening, and so on. You keep the original label in these synthetic examples. In practice, this often results in increased performance of the model.

8.5 Handling Multiple Inputs

Often in practice, you will work with multimodal data. For example, your input could be an image and text and the binary output could indicate whether the text describes this image.

It's hard to adapt shallow learning algorithms to work with multimodal data. However, it's not impossible. You could train one shallow model on the image and another one on the text. Then you can use a model combination technique we discussed above.

If you cannot divide your problem into two independent subproblems, you can try to vectorize each input (by applying the corresponding feature engineering method) and then simply concatenate two feature vectors together to form one wider feature vector. For example, if your image has features $[i^{(1)}, i^{(2)}, i^{(3)}]$ and your text has features $[t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}]$ your concatenated feature vector will be $[i^{(1)}, i^{(2)}, i^{(3)}, t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}]$.

With neural networks, you have more flexibility. You can build two subnetworks, one for each type of input. For example, a CNN subnetwork would read the image while an RNN subnetwork would read the text. Both subnetworks have as their last layer an embedding: CNN has an embedding of the image, while RNN has an embedding of the text. You can now concatenate two embeddings and then add a classification layer, such as softmax or sigmoid, on top of the concatenated embeddings. Neural network libraries provide simple to use tools that allow concatenating or averaging layers from several subnetworks.

8.6 Handling Multiple Outputs

In some problems, you would like to predict multiple outputs for one input. We considered multi-label classification in the previous chapter. Some problems with multiple outputs can be effectively converted into a multi-label classification problem. Especially those that have labels of the same nature (like tags) or fake labels can be created as a full enumeration of combinations of original labels.

However, in some cases the outputs are multimodal, and their combinations cannot be effectively enumerated. Consider the following example: you want to build a model that detects an object on an image and returns its coordinates. In addition, the model has to return a tag describing the object, such as "person," "cat," or "hamster." Your training example will be a feature vector that represents an image. The label will be represented as a vector of coordinates of the object and another vector with a one-hot encoded tag.

To handle a situation like that, you can create one subnetwork that would work as an encoder. It will read the input image using, for example, one or several convolution layers. The encoder's last layer would be the embedding of the image. Then you add two other subnetworks on top of the embedding layer: one that takes the embedding vector as input and predicts the coordinates of an object. This first subnetwork can have a ReLU as the last layer, which is a good choice for predicting positive real numbers, such as coordinates; this subnetwork could use the mean squared error cost C_1. The second subnetwork will take the same embedding vector as input and predict the probabilities for each tag. This second subnetwork can have a softmax as the last layer, which is appropriate for the probabilistic output, and use the averaged negative log-likelihood cost C_2 (also called cross-entropy cost).

Obviously, you are interested in both accurately predicted coordinates and the tags. However, it is impossible to optimize the two cost functions at the same time. By trying to optimize one, you risk hurting the second one and the other way around. What you can do is add another hyperparameter γ in the range $(0, 1)$ and define the combined cost function as $\gamma C_1 + (1-\gamma)C_2$. Then you tune the value for γ on the validation data just like any other hyperparameter.

8.7 Transfer Learning

Transfer learning is probably where neural networks have a unique advantage over the shallow models. In transfer learning, you pick an existing model trained on some dataset, and you adapt this model to predict examples from another dataset, different from the one the model was built on. This second dataset is not like holdout sets you use for validation and test. It may represent some other phenomenon, or, as machine learning scientists say, it may come from another statistical distribution.

For example, imagine you have trained your model to recognize (and label) wild animals on a big labeled dataset. After some time, you have another problem to solve: you need to build a model that would recognize domestic animals. With shallow learning algorithms, you do not have many options: you have to build another big labeled dataset, now for domestic animals.

With neural networks, the situation is much more favorable. Transfer learning in neural networks works like this.

1. You build a deep model on the original big dataset (wild animals).
2. You compile a much smaller labeled dataset for your second model (domestic animals).
3. You remove the last one or several layers from the first model. Usually, these are layers responsible for the classification or regression; they usually follow the embedding layer.

4. You replace the removed layers with new layers adapted for your new problem.
5. You "freeze" the parameters of the layers remaining from the first model.
6. You use your smaller labeled dataset and gradient descent to train the parameters of only the new layers.

Usually, there is an abundance of deep models for visual problems available online. You can find one that has high chances to be of use for your problem, download that model, remove several last layers (the quantity of layers to remove is a hyperparameter), put your own prediction layers and train your model.

Even if you don't have an existing model, transfer learning can still help you in situations when your problem requires a labeled dataset very costly to obtain, but you can get another dataset for which labels are more readily available. Let's say you build a document classification model. You got the taxonomy of labels from your employer, and it contains a thousand categories. In this case, you would need to pay someone to a) read, understand and memorize the differences between categories and b) read up to a million documents and annotate them.

To save on labeling so many examples, you could consider using Wikipedia pages as the dataset to build your first model. The labels for a Wikipedia page can be obtained automatically by taking the category the Wikipedia page belongs to. Once your first model has learned to predict Wikipedia categories, you can "fine tune" this model to predict the categories of your employer's taxonomy. You will need much fewer annotated examples for your employer's problem than you would need if you started solving your original problem from scratch.

8.8 Algorithmic Efficiency

Not all algorithms capable of solving a problem are practical. Some can be too slow. Some problems can be solved by a fast algorithm, for others, no fast algorithms can exist.

The subfield of computer science called *analysis of algorithms* is concerned with determining and comparing the complexity of algorithms. The **big O notation** is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

For example, let's say we have the problem of finding the two most distant one-dimensional examples in the set of examples $\mathcal{S}$ of size N. One algorithm we could craft to solve this problem would look like this (here and below, in Python):

```
def find_max_distance(S):
    result = None
    max_distance = 0
    for x1 in S:
        for x2 in S:
            if abs(x1 - x2) >= max_distance:
                max_distance = abs(x1 - x2)
```

```
                result = (x1, x2)
    return result
```

In the above algorithm, we loop over all values in $\mathcal{S}$, and at every iteration of the first loop, we loop over all values in $\mathcal{S}$ once again. Therefore, the above algorithm makes N^2 comparisons of numbers. If we take as a unit time the time the comparison, abs and assignment operations take, then the time complexity (or, simply, complexity) of this algorithm is at most $5N^2$. (At each iteration, we have one comparison, two abs and two assignment operations.) When the complexity of an algorithm is measured in the worst case, the big O notation is used. For the above algorithm, using the big O notation, we write that the algorithm's complexity is $O(N^2)$; the constants, like 5, are ignored.

For the same problem, we can craft another algorithm like this:

```
def find_max_distance(S):
    result = None
    min_x = float("inf")
    max_x = float("-inf")
    for x in S:
        if x < min_x:
            min_x = x
        elif x > max_x:
            max_x = x
    result = (max_x, min_x)
    return result
```

In the above algorithm, we loop over all values in $\mathcal{S}$ only once, so the algorithm's complexity is $O(N)$. In this case, we say that the latter algorithm is *more efficient* than the former.

An algorithm is called efficient when its complexity is polynomial in the size of the input. Therefore both $O(N)$ and $O(N^2)$ are efficient because N is a polynomial of degree 1 and N^2 is a polynomial of degree 2. However, for very large inputs, an $O(N^2)$ algorithm can be slow. In the big data era, scientists often look for $O(\log N)$ algorithms.

From a practical standpoint, when you implement your algorithm, you should *avoid using loops whenever possible*. For example, you should use operations on matrices and vectors, instead of loops. In Python, to compute $\mathbf{wx}$, you should write,

```
import numpy
wx = numpy.dot(w,x)
```

and not,

```
wx = 0
for i in range(N):
    wx += w[i]*x[i]
```

Use appropriate data structures. If the order of elements in a collection doesn't matter, use set instead of list. In Python, the operation of verifying whether a specific example $\mathbf{x}$ belongs

9 Unsupervised Learning

Unsupervised learning deals with problems in which data doesn't have labels. That property makes it very problematic for many applications. The absence of labels representing the desired behavior for your model means the absence of a solid reference point to judge the quality of your model. In this book, I only present unsupervised learning methods that allow building models that can be evaluated based on data as opposed to human judgment.

9.1 Density Estimation

Density estimation is a problem of modeling the probability density function (pdf) of the unknown probability distribution from which the dataset has been drawn. It can be useful for many applications, in particular for novelty or intrusion detection. In Chapter 7, we already estimated the pdf to solve the one-class classification problem. To do that, we decided that our model would be **parametric**, more precisely a multivariate normal distribution (MVN). This decision was somewhat arbitrary because if the real distribution from which our dataset was drawn is different from the MVN, our model will be very likely far from perfect. We also know that models can be nonparametric. We used a **nonparametric model** in kernel regression. It turns out that the same approach can work for density estimation.

Let $\{x_i\}_{i=1}^N$ be a one-dimensional dataset (a multi-dimensional case is similar) whose examples were drawn from a distribution with an unknown pdf f with $x_i \in \mathbb{R}$ for all $i = 1, \ldots, N$. We are interested in modeling the shape of f. Our kernel model of f, denoted as $\hat{f}_b$, is given by,

$$\hat{f}_b(x) = \frac{1}{Nb} \sum_{i=1}^{N} k\left(\frac{x - x_i}{b}\right), \tag{1}$$

where b is a hyperparameter that controls the tradeoff between bias and variance of our model and k is a kernel. Again, like in Chapter 7, we use a Gaussian kernel:

$$k(z) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-z^2}{2}\right).$$

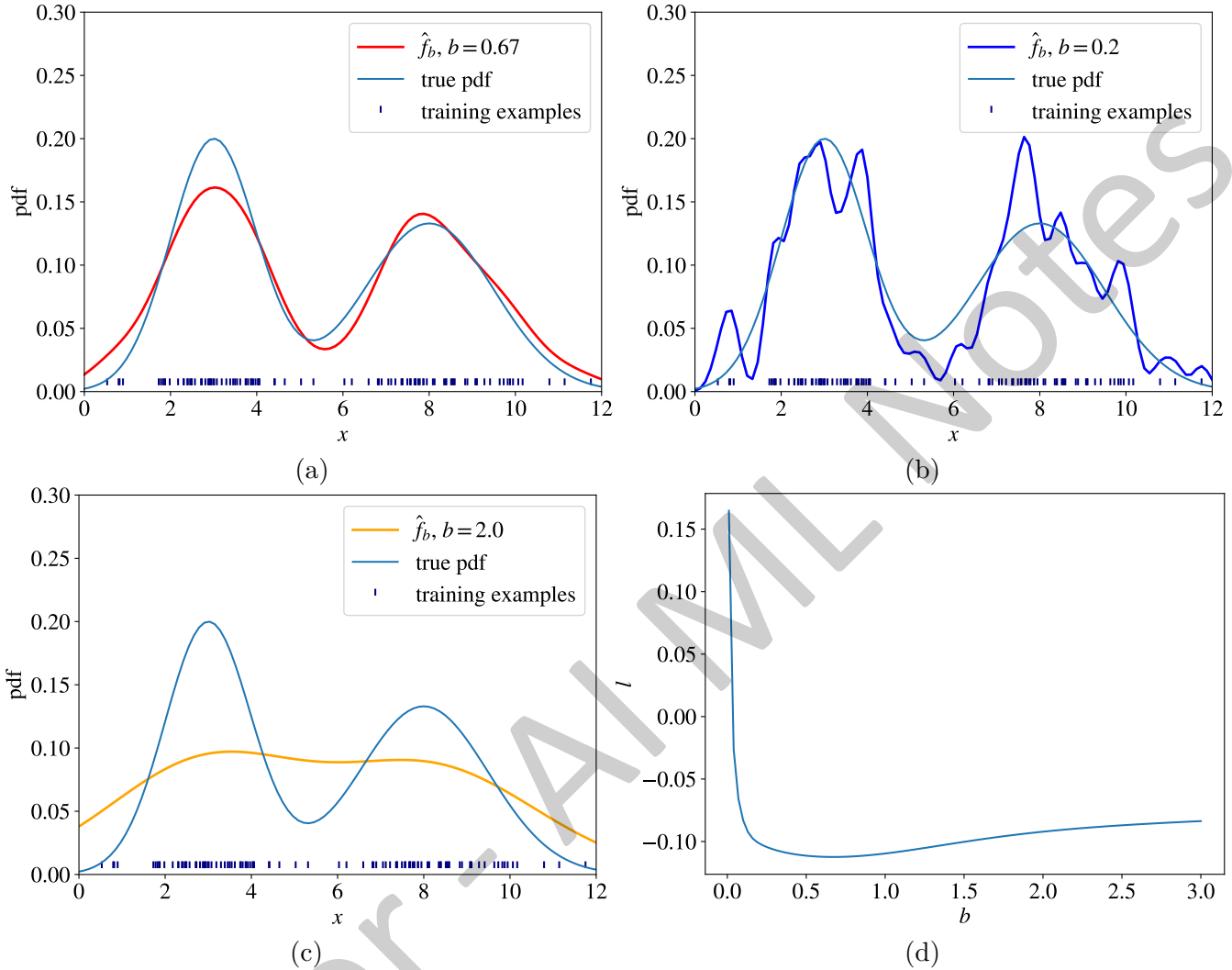


Figure 1: Kernel density estimation: (a) good fit; (b) overfitting; (c) underfitting; (d) the curve of grid search for the best value for b.

We look for such a value of b that minimizes the difference between the real shape of f and the shape of our model $\hat{f}_b$. A reasonable choice of measure of this difference is called the **mean integrated squared error** (MISE):

$$\text{MISE}(b) = \mathbb{E}\left[\int_{\mathbb{R}} (\hat{f}_b(x) - f(x))^2 \, dx\right]. \tag{2}$$

Intuitively, you see in eq. 2 that we square the difference between the real pdf f and our model of it $\hat{f}_b$. The integral $\int_{\mathbb{R}}$ replaces the summation $\sum_{i=1}^{N}$ we employed in the mean

squared error, while the expectation operator $\mathbb{E}$ replaces the average $\frac{1}{N}$.

Indeed, when our loss is a function with a continuous domain, such as $(\hat{f}_b(x) - f(x))^2$, we have to replace the summation with the integral. The expectation operation $\mathbb{E}$ means that we want b to be optimal for all possible realizations of our training set $\{x_i\}_{i=1}^N$. That is important because $\hat{f}_b$ is defined on a *finite* sample of some probability distribution, while the real pdf f is defined on an infinite domain (the set $\mathbb{R}$).

Now, we can rewrite the right-hand side term in eq. 2 like this:

$$\mathbb{E}\left[\int_{\mathbb{R}} \hat{f}_b^2(x)dx\right] - 2\mathbb{E}\left[\int_{\mathbb{R}} \hat{f}_b(x)f(x)dx\right] + \mathbb{E}\left[\int_{\mathbb{R}} f(x)^2 dx\right].$$

The third term in the above summation is independent of b and thus can be ignored. An unbiased estimator of the first term is given by $\int_{\mathbb{R}} \hat{f}_b^2(x)dx$ while the unbiased estimator of the second term can be approximated by **cross-validation** $-\frac{2}{N}\sum_{i=1}^N \hat{f}_b^{(i)}(x_i)$, where $\hat{f}_b^{(i)}$ is a kernel model of f computed on our training set with the example x_i excluded.

The term $\sum_{i=1}^N \hat{f}_b^{(i)}(x_i)$ is known in statistics as the *leave one out estimate*, a form of cross-validation in which each fold consists of one example. You could have noticed that the term $\int_{\mathbb{R}} \hat{f}_b(x)f(x)dx$ (let's call it a) is the expected value of the function $\hat{f}_b$, because f is a pdf. It can be demonstrated that the leave one out estimate is an unbiased estimator of $\mathbb{E}[a]$.

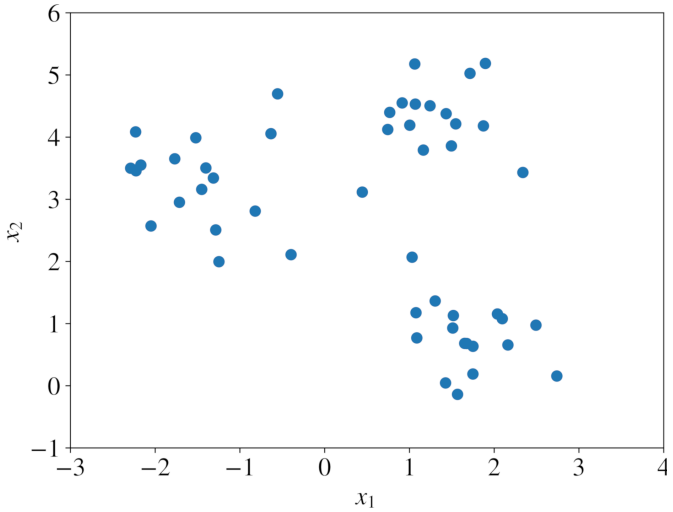
Now, to find the optimal value b^* for b, we minimize the cost defined as,

$$\int_{\mathbb{R}} \hat{f}_b^2(x)dx - \frac{2}{N}\sum_{i=1}^N \hat{f}_b^{(i)}(x_i).$$

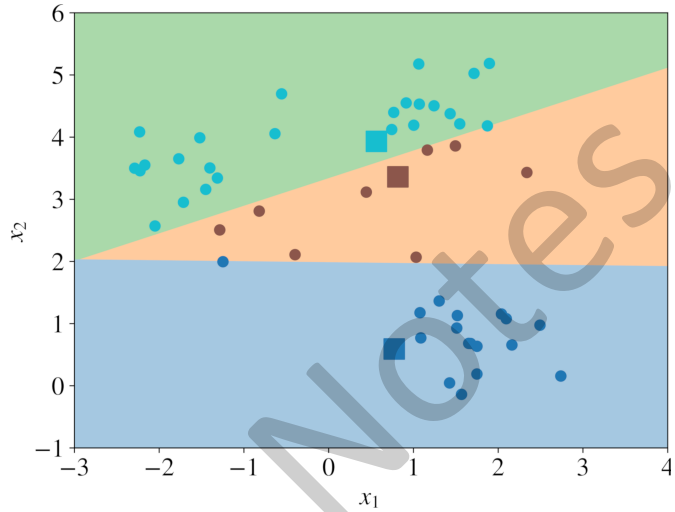
We can find b^* using grid search. For D-dimensional feature vectors $\mathbf{x}$, the error term $x - x_i$ in eq. 1 can be replaced by the Euclidean distance $\|\mathbf{x} - \mathbf{x}_i\|$. In Figure 1 you can see the estimates for the same pdf obtained with three different values of b from a 100-example dataset, as well as the grid search curve. We pick b^* at the minimum of the grid search curve.

9.2 Clustering

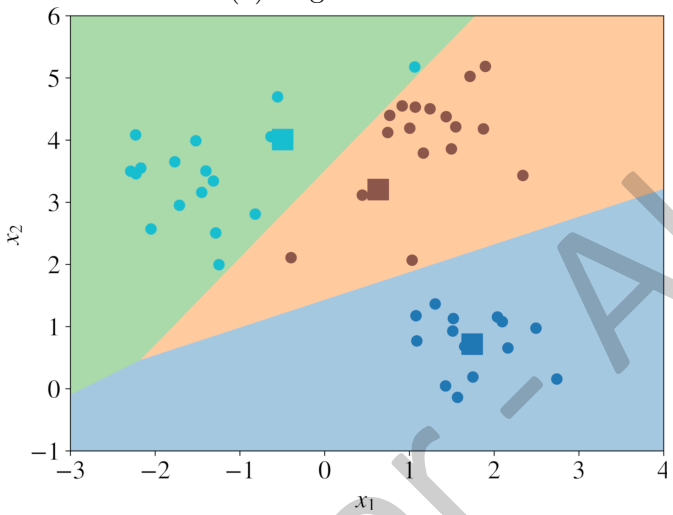
Clustering is a problem of learning to assign a label to examples by leveraging an unlabeled dataset. Because the dataset is completely unlabeled, deciding on whether the learned model is optimal is much more complicated than in supervised learning.



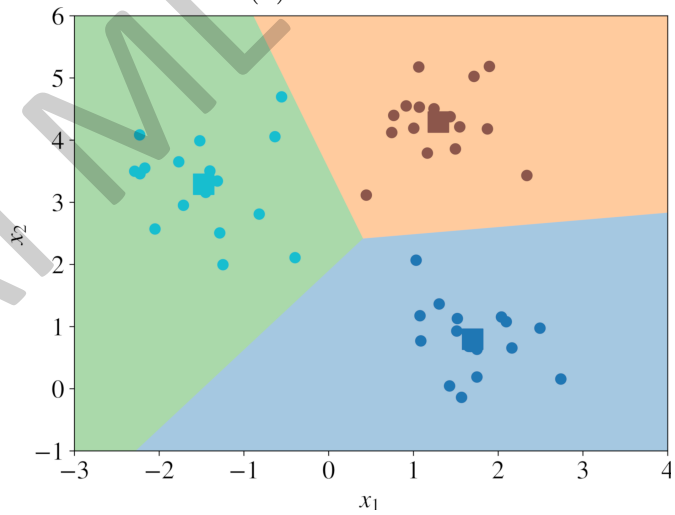
(a) original data



(b) iteration 1



(c) iteration 3



(d) iteration 5

Figure 2: The progress of the k-means algorithm for $k = 3$.

There is a variety of clustering algorithms, and, unfortunately, it's hard to tell which one is better in quality for your dataset. Usually, the performance of each algorithm depends on the unknown properties of the probability distribution the dataset was drawn from. In this Chapter, I outline the most useful and widely used clustering algorithms.

9.2.1 K-Means

The **k-means** clustering algorithm works as follows. First, you choose k — the number of clusters. Then you randomly put k feature vectors, called **centroids**, to the feature space.

We then compute the distance from each example $\mathbf{x}$ to each centroid $\mathbf{c}$ using some metric, like the Euclidean distance. Then we assign the closest centroid to each example (like if we labeled each example with a centroid id as the label). For each centroid, we calculate the average feature vector of the examples labeled with it. These average feature vectors become the new locations of the centroids.

We recompute the distance from each example to each centroid, modify the assignment and repeat the procedure until the assignments don't change after the centroid locations were recomputed. The model is the list of assignments of centroids IDs to the examples.

The initial position of centroids influence the final positions, so two runs of k-means can result in two different models. Some variants of k-means compute the initial positions of centroids based on some properties of the dataset.

One run of the k-means algorithm is illustrated in Figure 2. The circles in Figure 2 are two-dimensional feature vectors; the squares are moving centroids. Different background colors represent regions in which all points belong to the same cluster.

The value of k, the number of clusters, is a hyperparameter that has to be tuned by the data analyst. There are some techniques for selecting k. None of them is proven optimal. Most of those techniques require the analyst to make an "educated guess" by looking at some metrics or by examining cluster assignments visually. In this chapter, I present one approach to choose a reasonably good value for k without looking at the data and making guesses.

9.2.2 DBSCAN and HDBSCAN

While k-means and similar algorithms are centroid-based, **DBSCAN** is a density-based clustering algorithm. Instead of guessing how many clusters you need, by using DBSCAN, you define two hyperparameters: ϵ and n. You start by picking an example $\mathbf{x}$ from your dataset at random and assign it to cluster 1. Then you count how many examples have the distance from $\mathbf{x}$ less than or equal to ϵ. If this quantity is greater than or equal to n, then you put all these ϵ-neighbors to the same cluster 1. You then examine each member of cluster 1 and find their respective ϵ-neighbors. If some member of cluster 1 has n or more ϵ-neighbors, you expand cluster 1 by putting those ϵ-neighbors to the cluster. You continue expanding cluster 1 until there are no more examples to put in it. In the latter case, you pick from the dataset another example not belonging to any cluster and put it to cluster 2. You continue like this until all examples either belong to some cluster or are marked as outliers. An outlier is an example whose ϵ-neighborhood contains less than n examples.

The advantage of DBSCAN is that it can build clusters that have an arbitrary shape, while k-means and other centroid-based algorithms create clusters that have a shape of a hypersphere.

An obvious drawback of DBSCAN is that it has two hyperparameters and choosing good values for them (especially ϵ) could be challenging. Furthermore, having ϵ fixed, the clustering algorithm cannot effectively deal with clusters of varying density.

HDBSCAN is the clustering algorithm that keeps the advantages of DBSCAN, by removing the need to decide on the value of ϵ. The algorithm is capable of building clusters of varying density. HDBSCAN is an ingenious combination of multiple ideas and describing the algorithm in full is beyond the scope of this book.

HDBSCAN only has one important hyperparameter: n, the minimum number of examples to put in a cluster. This hyperparameter is relatively simple to choose by intuition. HDBSCAN has very fast implementations: it can deal with millions of examples effectively. Modern implementations of k-means are much faster than HDBSCAN though, but the qualities of the latter may outweigh its drawbacks for many practical tasks. I recommend to always try HDBSCAN on your data first.

9.2.3 Determining the Number of Clusters

The most important question is how many clusters does your dataset have? When the feature vectors are one-, two- or three-dimensional, you can look at the data and see "clouds" of points in the feature space. Each cloud is a potential cluster. However, for D-dimensional data, with $D > 3$, looking at the data is problematic[1].

One way of determining the reasonable number of clusters is based on the concept of **prediction strength**. The idea is to split the data into training and test set, similarly to how we do in supervised learning. Once you have the training and test sets, $\mathcal{S}_{tr}$ of size N_{tr} and $\mathcal{S}_{te}$ of size N_{te} respectively, you fix k, the number of clusters, and run a clustering algorithm C on sets $\mathcal{S}_{tr}$ and $\mathcal{S}_{te}$ and obtain the clustering results $C(\mathcal{S}_{tr}, k)$ and $C(\mathcal{S}_{te}, k)$.

Let A be the clustering $C(\mathcal{S}_{tr}, k)$ built using the training set. The clusters in A can be seen as regions. If an example falls within one of those regions, then that example belongs to some specific cluster. For example, if we apply the k-means algorithm to some dataset, it results in a partition of the feature space into k polygonal regions, as we saw in Figure 2.

Define the $N_{te} \times N_{te}$ *co-membership matrix* $\mathbf{D}[A, \mathcal{S}_{te}]$ as follows: $\mathbf{D}[A, \mathcal{S}_{te}]^{(i,i')} = 1$ if and only if examples $\mathbf{x}_i$ and $\mathbf{x}_{i'}$ from the test set belong to the same cluster according to the clustering A. Otherwise $D[A, \mathcal{S}_{te}]^{(i,i')} = 0$.

Let's take a break and see what we have here. We have built, *using the training set* of examples, a clustering A that has k clusters. Then we have built the co-membership matrix that indicates whether two examples *from the test set* belong to the same cluster in A.

[1] Some analysts look at multiple two-dimensional plots, in which only a pair of features are present at the same time. It might give an intuition about the number of clusters. However, such an approach suffers from subjectivity, is prone to error and counts as an educated guess rather than a scientific method.

Intuitively, if the quantity k is the reasonable number of clusters, then two examples that belong to the same cluster in clustering $C(\mathcal{S}_{te}, k)$ will most likely belong to the same cluster in clustering $C(\mathcal{S}_{tr}, k)$. On the other hand, if k is not reasonable (too high or too low), then training data-based and test data-based clusterings will likely be less consistent.

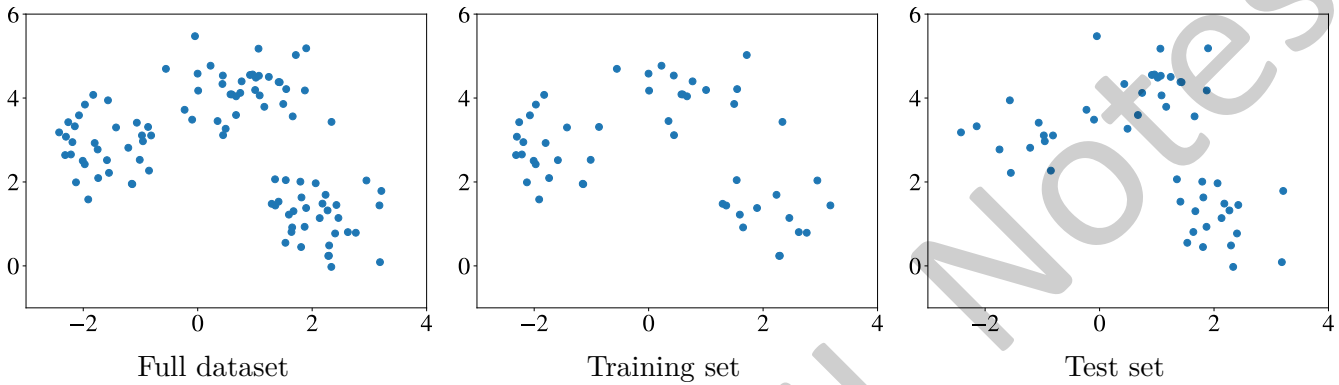


Figure 3: Data used for clustering illustrated in Figure 4

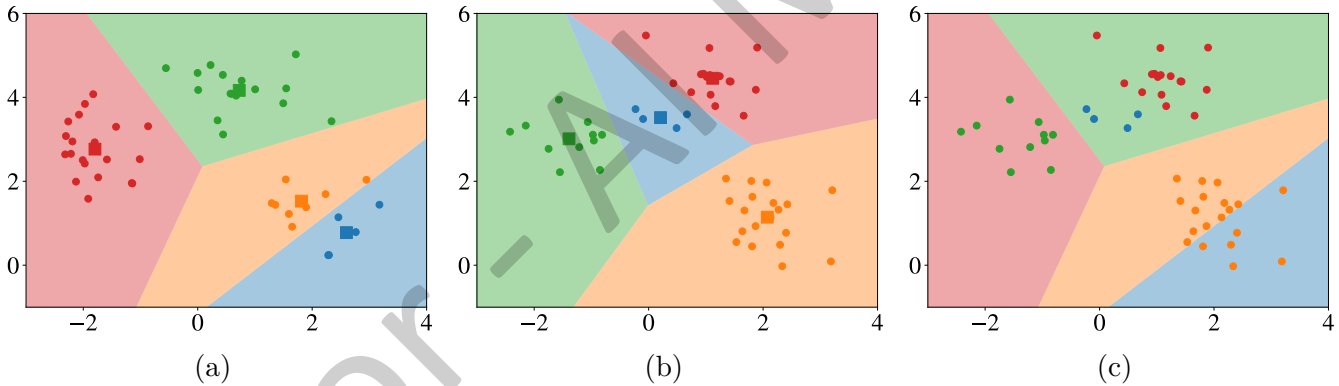


Figure 4: The clustering for $k = 4$: (a) training data clustering; (b) test data clustering; (c) test data plotted over the training clustering.

Using the data shown in Figure 3, the idea is illustrated in Figure 4. The plots in Figure 4a and 4b show respectively $C(\mathcal{S}_{tr}, 4)$ and $C(\mathcal{S}_{te}, 4)$ with their respective cluster regions. Figure 4c shows the test examples plotted over the training data cluster regions. You can see in 4c that orange test examples don't belong anymore to the same cluster according to the clustering regions obtained from the training data. This will result in many zeroes in the matrix $\mathbf{D}[A, \mathcal{S}_{te}]$ which, in turn, is an indicator that $k = 4$ is likely not the best number of clusters.

More formally, the prediction strength for the number of clusters k is given by,

$$\text{ps}(k) \stackrel{\text{def}}{=} \min_{j=1,\ldots,k} \frac{1}{|A_j|(|A_j|-1)} \sum_{i,i' \in A_j} \mathbf{D}[A, \mathcal{S}_{te}]^{(i,i')},$$

where $A \stackrel{\text{def}}{=} C(\mathcal{S}_{tr}, k)$, A_j is j^{th} cluster from the clustering $C(\mathcal{S}_{te}, k)$ and $|A_j|$ is the number of examples in cluster A_j.

Given a clustering $C(\mathcal{S}_{tr}, k)$, for each test cluster, we compute the proportion of observation pairs in that cluster that are also assigned to the same cluster by the training set centroids. The prediction strength is the minimum of this quantity over the k test clusters.

Experiments suggest that a reasonable number of clusters is the largest k such that $\text{ps}(k)$ is above 0.8. You can see in Figure 5 examples of predictive strength for different values of k for two-, three- and four-cluster data.

For non-deterministic clustering algorithms, such as k-means, which can generate different clusterings depending on the initial positions of centroids, it is recommended to do multiple runs of the clustering algorithm for the same k and compute the average prediction strength $\bar{\text{ps}}(k)$ over multiple runs.

Another effective method to estimate the number of clusters is the **gap statistic** method. Other, less automatic methods, which some analysts still use, include the **elbow method** and the **average silhouette method**.

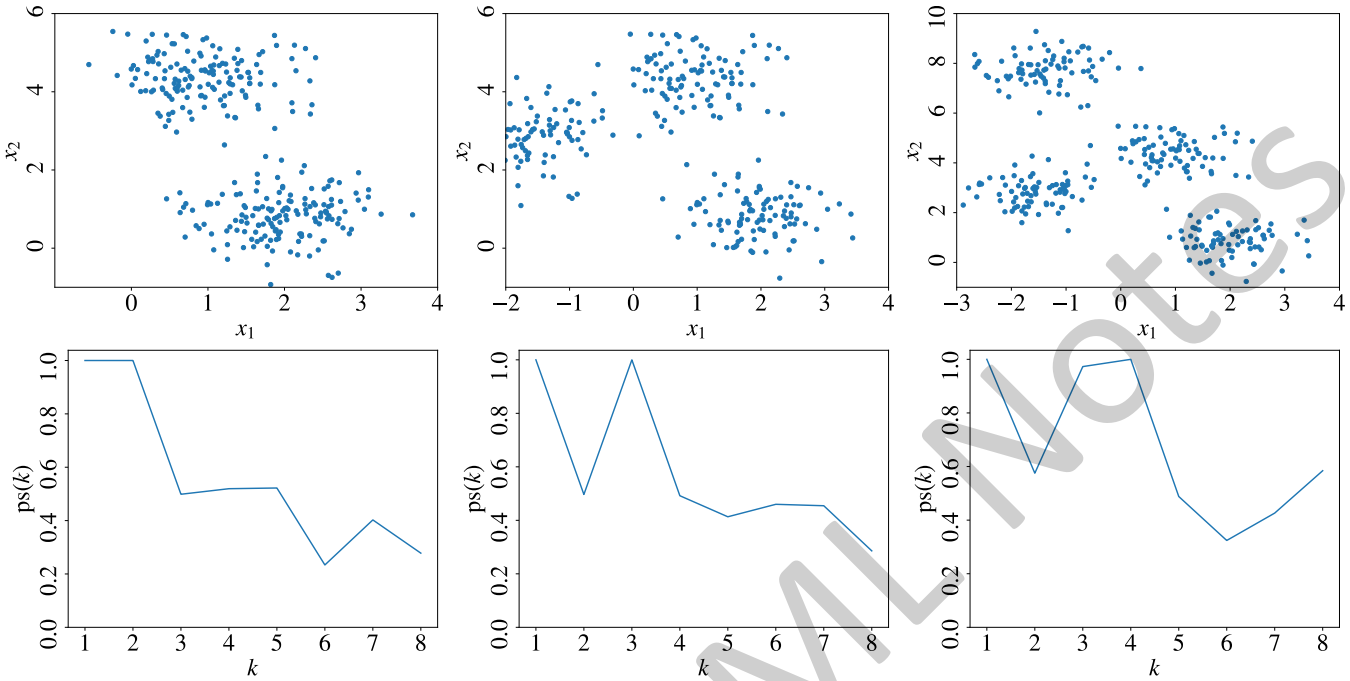


Figure 5: Predictive strength for different values of k for two-, three- and four-cluster data.

9.2.4 Other Clustering Algorithms

DBSCAN and k-means compute so-called **hard clustering**, in which each example can belong to only one cluster. **Gaussian mixture model** (GMM) allow each example to be a member of several clusters with different **membership score** (HDBSCAN allows that too). Computing a GMM is very similar to doing model-based density estimation. In GMM, instead of having just one multivariate normal distribution (MND), we have a weighted sum of several MNDs:

$$f_X = \sum_{j=1}^{k} \phi_j f_{\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j},$$

where $f_{\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j}$ is a MND j, and ϕ_j is its weight in the sum. The values of parameters $\boldsymbol{\mu}_j$, $\boldsymbol{\Sigma}_j$, and ϕ_j, for all $j = 1, \ldots, k$ are obtained using the **expectation maximization algorithm** (EM) to optimize the **maximum likelihood** criterion.

Again, for simplicity, let us look at the one-dimensional data. Also assume that there are two clusters: $k = 2$. In this case, we have two Gaussian distributions,

$$f(x \mid \mu_1, \sigma_1^2) = \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp -\frac{(x-\mu_1)^2}{2\sigma_1^2} \text{ and } f(x \mid \mu_2, \sigma_2^2) = \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp -\frac{(x-\mu_2)^2}{2\sigma_2^2}, \quad (3)$$

where $f(x \mid \mu_1, \sigma_1^2)$ and $f(x \mid \mu_2, \sigma_2^2)$ are two pdf defining the likelihood of $X = x$.

We use the EM algorithm to estimate μ_1, σ_1^2, μ_2, σ_2^2, ϕ_1, and ϕ_2. The parameters ϕ_1 and ϕ_2 are useful for the density estimation and less useful for clustering, as we will see below.

EM works like follows. In the beginning, we guess the initial values for μ_1, σ_1^2, μ_2, and σ_2^2, and set $\phi_1 = \phi_2 = \frac{1}{2}$ (in general, it's $\frac{1}{k}$ for each ϕ_j, $j \in 1, \ldots, k$).

At each iteration of EM, the following four steps are executed:

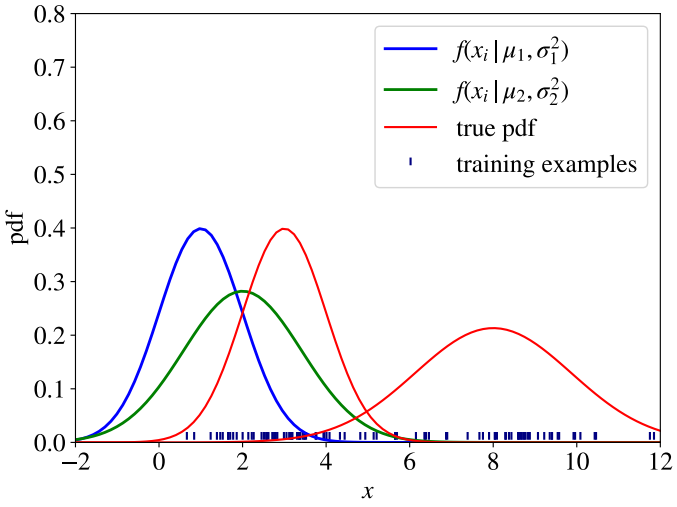
1. For all $i = 1, \ldots, N$, calculate the likelihood of each x_i using eq. 3:

$$f(x_i \mid \mu_1, \sigma_1^2) \leftarrow \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp -\frac{(x_i-\mu_1)^2}{2\sigma_1^2} \text{ and } f(x_i \mid \mu_2, \sigma_2^2) \leftarrow \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp -\frac{(x_i-\mu_2)^2}{2\sigma_2^2}.$$

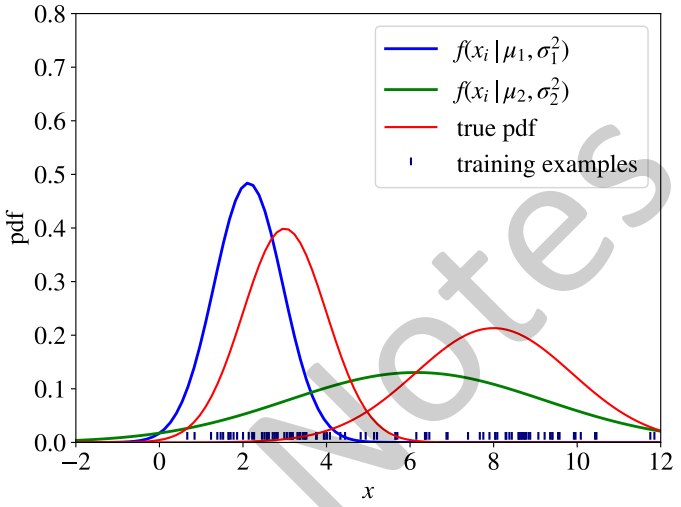
2. Using **Bayes' Rule**, for each example x_i, calculate the likelihood $b_i^{(j)}$ that the example belongs to cluster $j \in \{1, 2\}$ (in other words, the likelihood that the example was drawn from the Gaussian j):

$$b_i^{(j)} \leftarrow \frac{f(x_i \mid \mu_j, \sigma_j^2)\phi_j}{f(x_i \mid \mu_1, \sigma_1^2)\phi_1 + f(x_i \mid \mu_2, \sigma_2^2)\phi_2}.$$

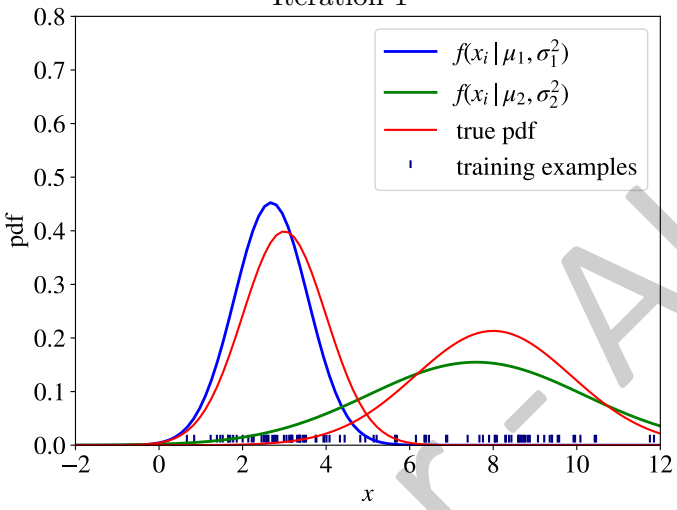
The parameter ϕ_j reflects how likely is that our Gaussian distribution j with parameters μ_j and σ_j^2 may have produced our dataset. That is why in the beginning we set $\phi_1 = \phi_2 = \frac{1}{2}$: we don't know how each of the two Gaussians is likely, and we reflect our ignorance by setting the likelihood of both to one half.



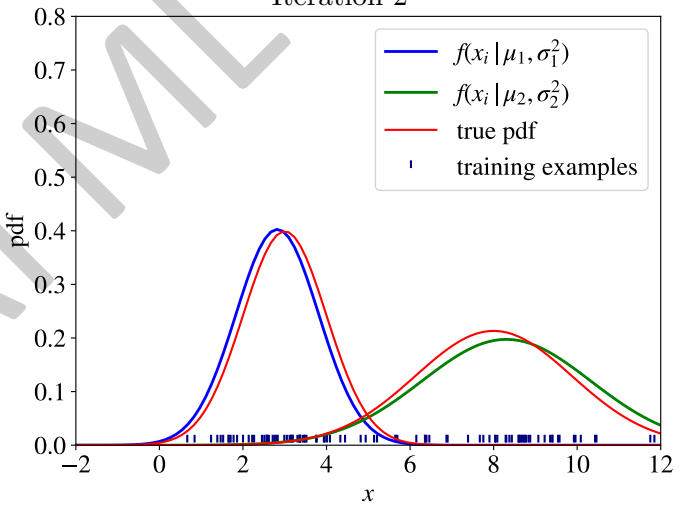
Iteration 1



Iteration 2



Iteration 10



Iteration 40

Figure 6: The progress of the Gaussian mixture model estimation using the EM algorithm for two clusters ($k = 2$).

3. Compute the new values of μ_j and σ_j^2, $j \in \{1, 2\}$ as,

$$\mu_j \leftarrow \frac{\sum_{i=1}^{N} b_i^{(j)} x_i}{\sum_{i=1}^{N} b_i^{(j)}} \text{ and } \sigma_j^2 \leftarrow \frac{\sum_{i=1}^{N} b_i^{(j)} (x_i - \mu_j)^2}{\sum_{i=1}^{N} b_i^{(j)}}. \tag{4}$$

4. Update ϕ_j, $j \in \{1, 2\}$ as,

$$\phi_j \leftarrow \frac{1}{N}\sum_{i=1}^{N} b_i^{(j)}.$$

The steps $1-4$ are executed iteratively until the values μ_j and σ_j^2 don't change much: for example, the change is below some threshold ϵ. Figure 6 illustrates this process.

You may have noticed that the EM algorithm is very similar to the k-means algorithm: start with random clusters, then iteratively update each cluster's parameters by averaging the data that is assigned to that cluster. The only difference in the case of the GMM is that the assignment of an example x_i to the cluster j is **soft**: x_i belongs to cluster j with probability $b_i^{(j)}$. This is why we calculate the new values for μ_j and σ_j^2 in eq. 4 not as an average (used in k-means) but as a *weighted average* with weights $b_i^{(j)}$.

Once we have learned the parameters μ_j and σ_j^2 for each cluster j, the membership score of example x in cluster j is given by $f(x \mid \mu_j, \sigma_j^2)$.

The extension to D-dimensional data ($D > 1$) is straightforward. The only difference is that instead of the variance σ^2, we now have the covariance matrix $\mathbf{\Sigma}$ that parametrizes the multinomial normal distribution (MND).

Contrary to k-means where clusters can only be circular, the clusters in GMM have a form of an ellipse that can have an arbitrary elongation and rotation. The values in the covariance matrix control these properties.

There's no universally recognized method to choose the right k in GMM. I recommend to first split the dataset into training and test set. Then you try different k and build a different model f_{tr}^k for each k on the training data. You pick the value of k that maximizes the likelihood of examples in the test set:

$$\arg\max_k \prod_{i=1}^{|N_{te}|} f_{tr}^k(\mathbf{x}_i),$$

where $|N_{te}|$ is the size of the test set.

There is a variety of clustering algorithms described in the literature. Worth mentioning are **spectral clustering** and **hierarchical clustering**. For some datasets, you may find those more appropriate. However, in most practical cases, kmeans, HDBSCAN and the Gaussian mixture model would satisfy your needs.

9.3 Dimensionality Reduction

Modern machine learning algorithms, such as ensemble algorithms and neural networks handle well very high-dimensional examples, up to millions of features. With modern computers and graphical processing units (GPUs), dimensionality reduction techniques are used less in practice than in the past. The most frequent use case for dimensionality reduction is data visualization: humans can only interpret on a plot the maximum of three dimensions.

Another situation in which you could benefit from dimensionality reduction is when you have to build an interpretable model and to do so you are limited in your choice of learning algorithms. For example, you can only use decision tree learning or linear regression. By reducing your data to lower dimensionality and by figuring out which quality of the original example each new feature in the reduced feature space reflects, one can use simpler algorithms. Dimensionality reduction removes redundant or highly correlated features; it also reduces the noise in the data — all that contributes to the interpretability of the model.

Three widely used techniques of dimensionality reduction are **principal component analysis** (PCA), **uniform manifold approximation and projection** (UMAP), and **autoencoders**.

I already explained autoencoders in Chapter 7. You can use the low-dimensional output of the **bottleneck layer** of the autoencoder as the vector of reduced dimensionality that represents the high-dimensional input feature vector. You know that this low-dimensional vector represents the essential information contained in the input vector because the autoencoder is capable of reconstructing the input feature vector based on the bottleneck layer output alone.

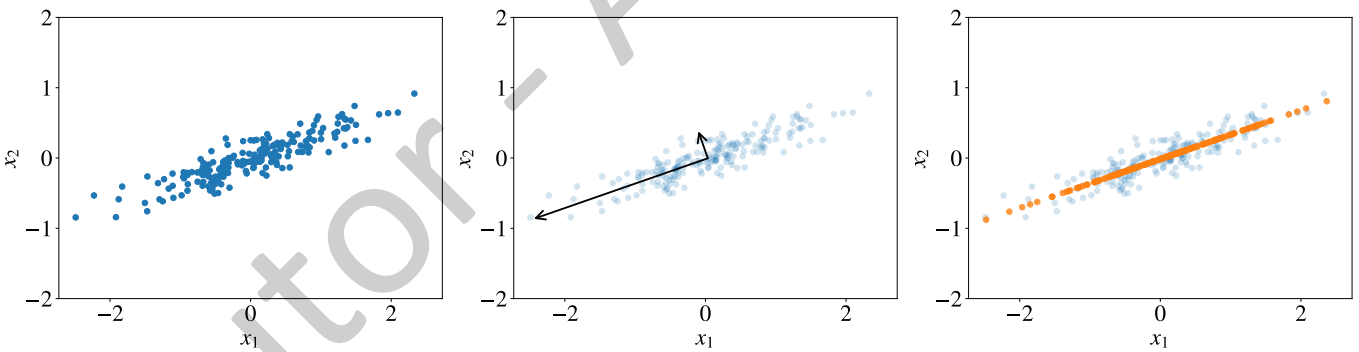


Figure 7: PCA: (a) the original data; (b) two principal components displayed as vectors; (c) the data projected on the first principal component.