2.1.9 Derivative and Gradient

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

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

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

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

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

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

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

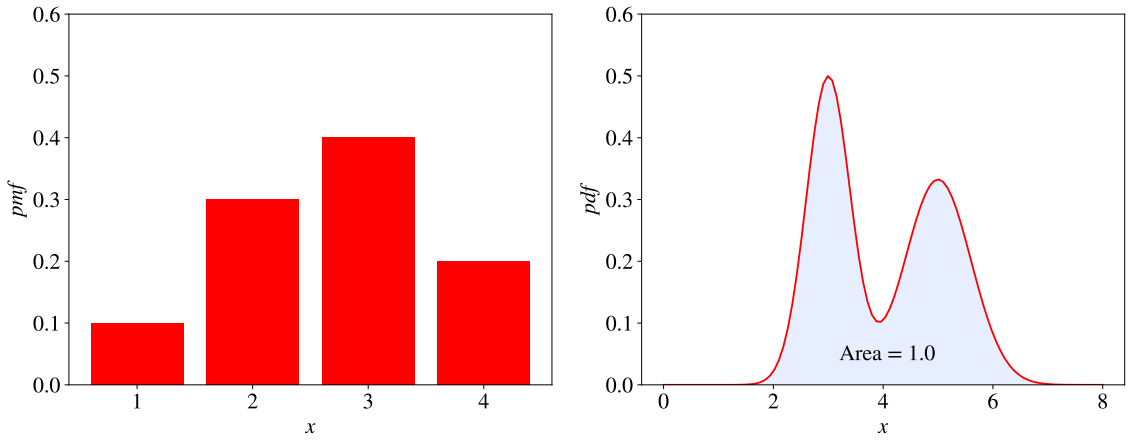
where a is the derivative of the function $ax^{(1)}$; the two zeroes are respectively derivatives of $bx^{(2)}$ and c, because $x^{(2)}$ is considered constant when we compute the derivative with respect to $x^{(1)}$, and the derivative of any constant is zero.

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

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

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

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



(a) (b)

Figure 3: A probability mass function and a probability density function.

2.2 Random Variable

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

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

The *probability distribution* of a discrete random variable is described by a list of probabilities associated with each of its possible values. This list of probabilities is called *probability mass function* (pmf). For example: $\Pr(X = red) = 0.3$, $\Pr(X = yellow) = 0.45$, $\Pr(X = blue) = 0.25$. Each probability in a probability mass function is a value greater than or equal to 0. The sum of probabilities equals 1 (fig. 3a).

A *continuous random variable* takes an infinite number of possible values in some interval. Examples include height, weight, and time. Because the number of values of a continuous random variable X is infinite, the probability $\Pr(X = c)$ for any c is 0. Therefore, instead of the list of probabilities, the probability distribution of a continuous random variable (a continuous probability distribution) is described by a *probability density function* (pdf). The pdf is a function whose codomain is nonnegative and the area under the curve is equal to 1 (fig. 3b).

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

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

where $\Pr(X = x_i)$ is the probability that X has the value x_i according to the pmf. The expectation of a random variable is also called the *mean*, *average* or *expected value* and is frequently denoted with the letter μ. The expectation is one of the most important *statistics* of a random variable. Another important statistic is the *standard deviation*. For a discrete random variable, the standard deviation usually denoted as σ is given by:

$$\sigma \stackrel{\text{def}}{=} \sqrt{\mathbb{E}[(X-\mu)^2]} = \sqrt{\Pr(X = x_1)(x_1 - \mu)^2 + \Pr(X = x_2)(x_2 - \mu)^2 + \cdots + \Pr(X = x_k)(x_k - \mu)^2},$$

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

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

$$\mathbb{E}[X] \stackrel{\text{def}}{=} \int_{\mathbb{R}} x f_X(x)\, dx, \quad (2)$$

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

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

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

2.3 Unbiased Estimators

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

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

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

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

It can be shown that an unbiased estimator of an unknown $\mathbb{E}[X]$ (given by either eq. 1 or eq. 2) is given by $\frac{1}{N}\sum_{i=1}^{N} x_i$ (called in statistics the *sample mean*).

2.4 Bayes' Rule

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

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

2.5 Parameter Estimation

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

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

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

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

$$\Pr(\theta = \hat{\theta}|X = x) \leftarrow \frac{\Pr(X = x|\theta = \hat{\theta})\Pr(\theta = \hat{\theta})}{\Pr(X = x)} = \frac{\Pr(X = x|\theta = \hat{\theta})\Pr(\theta = \hat{\theta})}{\sum_{\tilde{\theta}} \Pr(X = x|\theta = \tilde{\theta})}. \quad (3)$$

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

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

First, we compute $\Pr(\theta = \hat{\theta}|X = x_1)$ for all possible values $\hat{\theta}$. Then, before updating $\Pr(\theta = \hat{\theta}|X = x)$ once again, this time for $x = x_2 \in \mathcal{S}$ using eq. 3, we replace the prior $\Pr(\theta = \hat{\theta})$ in eq. 3 by the new estimate $\Pr(\theta = \hat{\theta}) \leftarrow \frac{1}{N}\sum_{x \in \mathcal{S}} \Pr(\theta = \hat{\theta}|X = x)$.

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

$$\theta^* = \arg\max_{\theta} \prod_{i=1}^{N} \Pr(\theta = \hat{\theta}|X = x_i). \tag{4}$$

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

2.6 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 data analyst to deduce the label easily. 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 books).

Multiclass classification (also called **multinomial**) is a classification problem with three or more classes[2].

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

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

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

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

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.

2.7 Model-Based vs. Instance-Based Learning

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

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

2.8 Shallow vs. Deep Learning

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

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

The Hundred-Page Machine Learning Book

Andriy Burkov

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

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

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 fig. 1. It displays the regression line (in light-blue) for one-dimensional examples (dark-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.

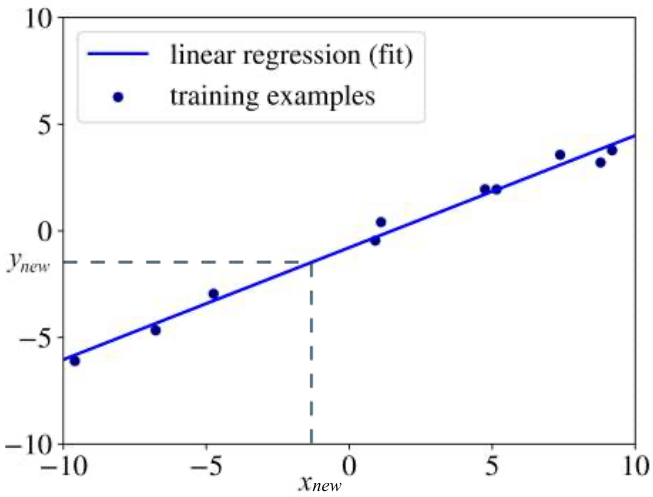


Figure 1: Linear Regression for one-dimensional examples.

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 blue line in fig. 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 $(f(\mathbf{x}_i) - y_i)^2$ 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 fig. 2. The data used to build the red regression line is the same as in fig. 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 fig. 1 for x_{new}. We talk more about overfitting and how to avoid it Chapter 5.

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

Intuitively, squared penalties are also advantageous because they exaggerate the difference between the true target and the predicted one according to the value of this difference. We

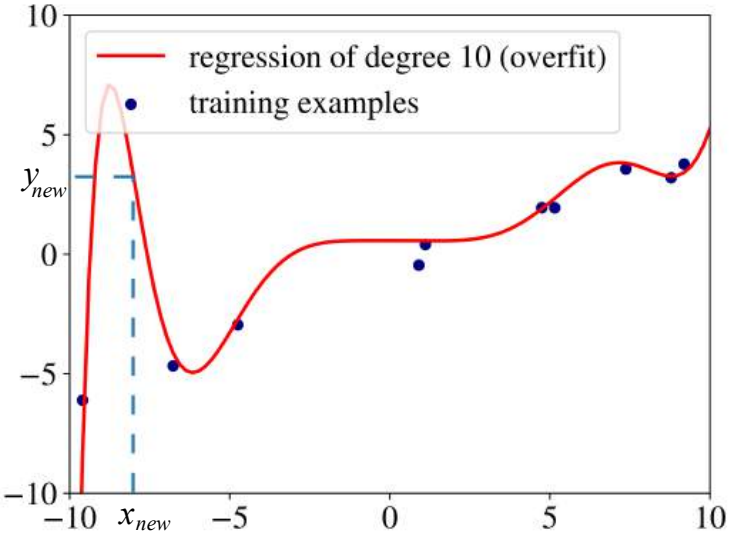


Figure 2: Overfitting.

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? Remember from algebra that 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^*. You can spend several minutes and check it yourself.

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.

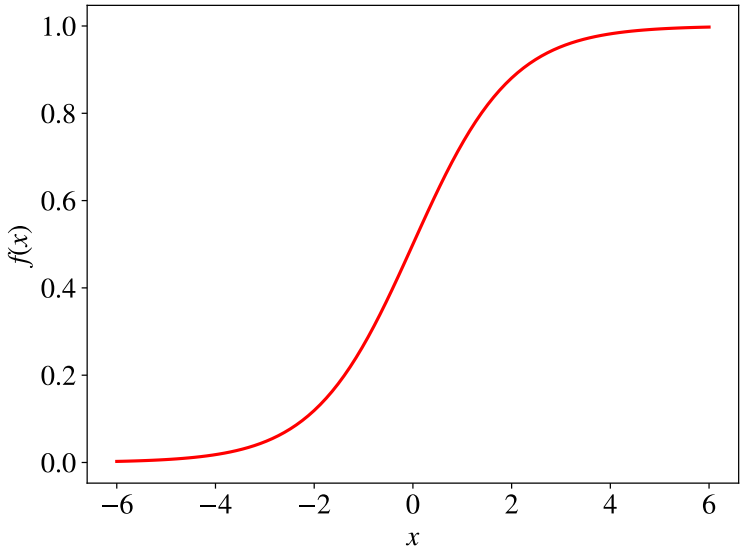


Figure 3: Standard logistic function.

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 Excel and many programming languages). Its graph is depicted in fig. 3.

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

So our 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. Now, how do we find the best values $\mathbf{w}^*$ and b^* for our model? 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.

3.2.2 Solution

In logistic regression, instead of using a squared loss and trying to minimize the empirical risk, 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, assume that we have a labeled example $(\mathbf{x}_i, y_i)$ in our training data. Assume also that we have 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:

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

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**. I 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 of an example 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:

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

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

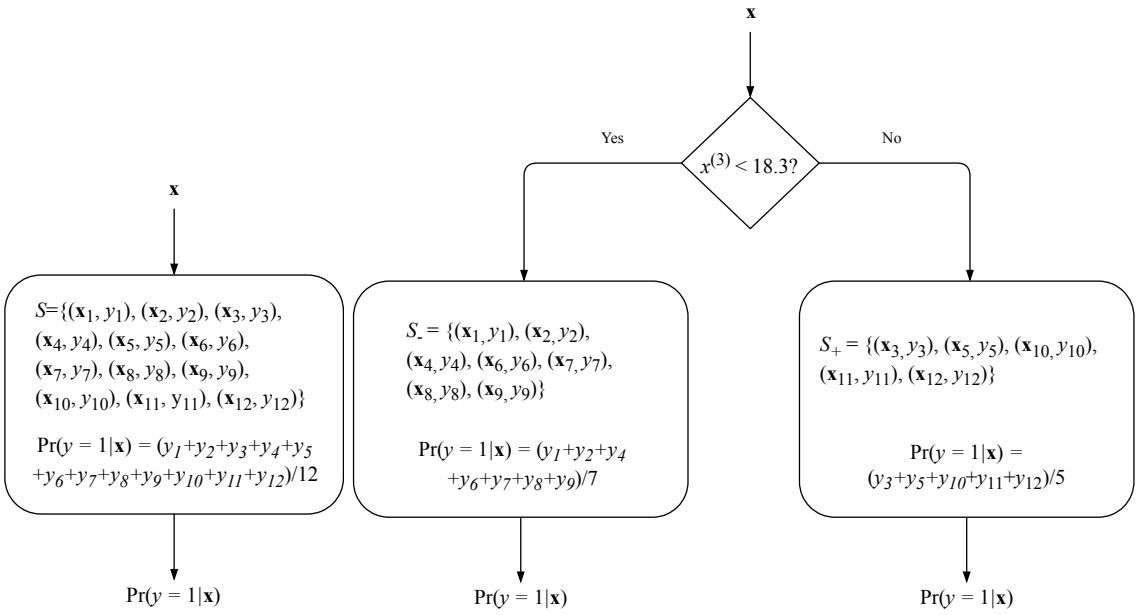


Figure 4: An illustration of a decision tree building algorithm. The set $\mathcal{S}$ contains 12 labeled examples. (a) In the beginning, the decision tree only contains the start node; it makes the same prediction for any input. (b) The decision tree after the first split; it tests whether feature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

$$f_{ID3}^{\mathcal{S}} = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\in\mathcal{S}} y. \tag{6}$$

The prediction given by the above model, $f_{ID3}^{S}(\mathbf{x})$, would be the same for any input $\mathbf{x}$. The corresponding decision tree 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 $\mathcal{S}_+ = \{(\mathbf{x}, y) \,|\, (\mathbf{x}, y) \in S, x^{(j)} \geq t\}$. 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:

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

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

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

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 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 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]In Chapter 5, we will see how to do that when we talk about hyperparameter tuning.

3.4 Support Vector Machine

We already considered 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?

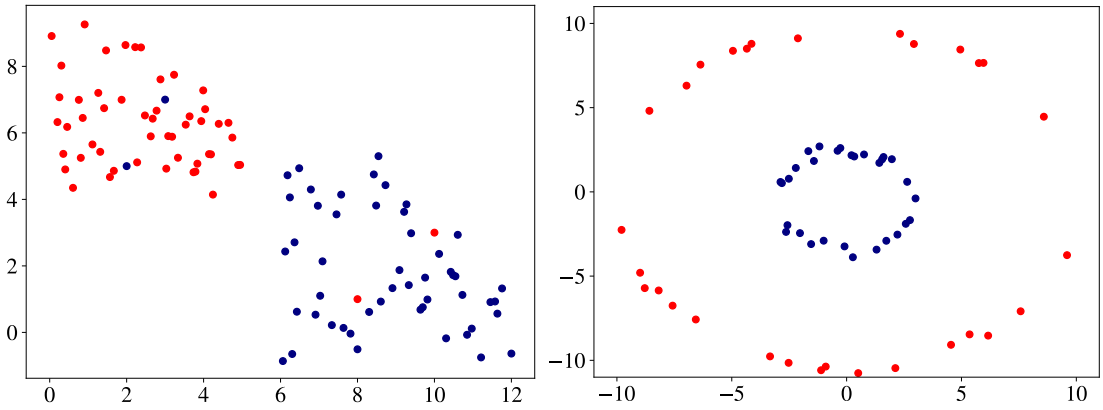


Figure 5: Linearly non-separable cases. Left: the presence of noise. Right: inherent nonlinearity.

You can see both situations depicted in fig 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:

a) $\mathbf{w}\mathbf{x}_i - b \geq 1$ if $y_i = +1$, and
b) $\mathbf{w}\mathbf{x}_i - b \leq -1$ if $y_i = -1$

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

$$\min \frac{1}{2}||\mathbf{w}||^2, \text{ such that } y_i(\mathbf{x}_i\mathbf{w} - b) - 1 \geq 0, i = 1, \ldots, N. \tag{8}$$

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 a) and b) 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 will try 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. However, 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**.

The effect of applying the kernel trick is illustrated in fig. 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 fig. 5 (right), the mapping ϕ for example $\mathbf{x} = [q, p]$ that projects this example into a 3D space (fig. 6) would look like this $\phi([q, p]) \stackrel{\text{def}}{=} (q^2, \sqrt{2}qp, p^2)$, where q^2 means q squared. You see now that the data becomes linearly separable in the transformed space.

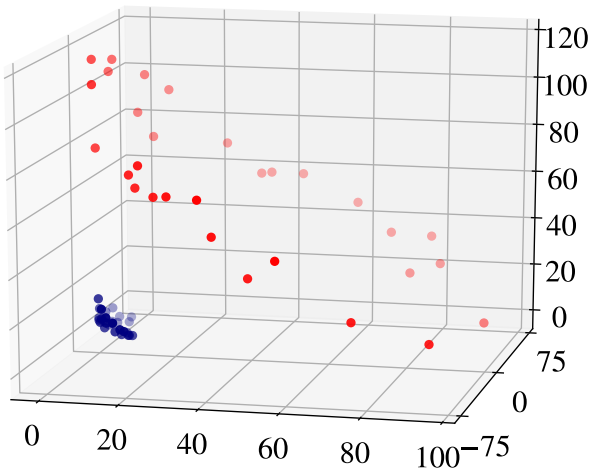


Figure 6: The data from fig. 5 (right) becomes linearly separable after a transformation into a three-dimensional space.

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. 8 is the *method of Lagrange multipliers*. Instead of solving the original problem from eq. 8, 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.