

Andriy Burkov

# **THE HUNDRED-PAGE MACHINE LEARNING BOOK**



*“A great introduction to machine learning from a world-class practitioner.”*

— **Karolis Urbonas**, Head of Data Science at Amazon

*“I wish such a book existed when I was a statistics graduate student trying to learn about machine learning.”*

— **Chao Han**, VP, Head of R&D at Lucidworks

*“Andriy’s book does a fantastic job of cutting the noise and hitting the tracks and full speed from the first page.”*

— **Sujeet Varakhedi**, Head of Engineering at eBay

*“A wonderful book for engineers who want to incorporate ML in their day-to-day work without necessarily spending an enormous amount of time.”*

— **Deepak Agarwal**, VP of Artificial Intelligence at LinkedIn

*“Excellent read to get started with Machine Learning.”*

— **Vincent Pollet**, Head of Research at Nuance

with back cover text from **Peter Norvig** and **Aurélien Geron**



# The Hundred-Page Machine Learning Book

Andriy Burkov

Copyright ©2019 Andriy Burkov

All rights reserved. This book is distributed on the “read first, buy later” principle. The latter implies that anyone can obtain a copy of the book by any means available, read it and share it with anyone else. However, if you read the book, liked it or found it helpful or useful in any way, you have to buy it. For further information, please email [author@themlbook.com](mailto:author@themlbook.com).

ISBN 978-1-9995795-0-0

Publisher: Andriy Burkov

To my parents:  
Tatiana and Valeriy

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



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

*“If I had more time, I would have written a shorter letter.”*  
— Blaise Pascal

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





# Contents

<b>3</b>	<b>Fundamental Algorithms</b>	<b>1</b>
3.1	Linear Regression . . . . .	1
3.1.1	Problem Statement . . . . .	1
3.1.2	Solution . . . . .	3
3.2	Logistic Regression . . . . .	5
3.2.1	Problem Statement . . . . .	5
3.2.2	Solution . . . . .	6
3.3	Decision Tree Learning . . . . .	7
3.3.1	Problem Statement . . . . .	7
3.3.2	Solution . . . . .	7
3.4	Support Vector Machine . . . . .	10
3.4.1	Dealing with Noise . . . . .	11
3.4.2	Dealing with Inherent Non-Linearity . . . . .	12
3.5	k-Nearest Neighbors . . . . .	14
	<b>Appendices</b>	<b>15</b>
.1	C4.5 . . . . .	17
<b>A</b>	<b>SVM Dual Derivation</b>	<b>23</b>
A.1	Method of Lagrange Multipliers . . . . .	23
A.2	Lagrange Multipliers in SVM . . . . .	24



# Chapter 3

## Fundamental Algorithms

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

### 3.1 Linear Regression

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

#### 3.1.1 Problem Statement

We have a collection of labeled examples  $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ , where  $N$  is the size of the collection,  $\mathbf{x}_i$  is the  $D$ -dimensional feature vector of example  $i = 1, \dots, N$ ,  $y_i$  is a real-valued<sup>1</sup> target and every feature  $x_i^{(j)}$ ,  $j = 1, \dots, D$ , 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  $\mathbf{x}$ :

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

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

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

---

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

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

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

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

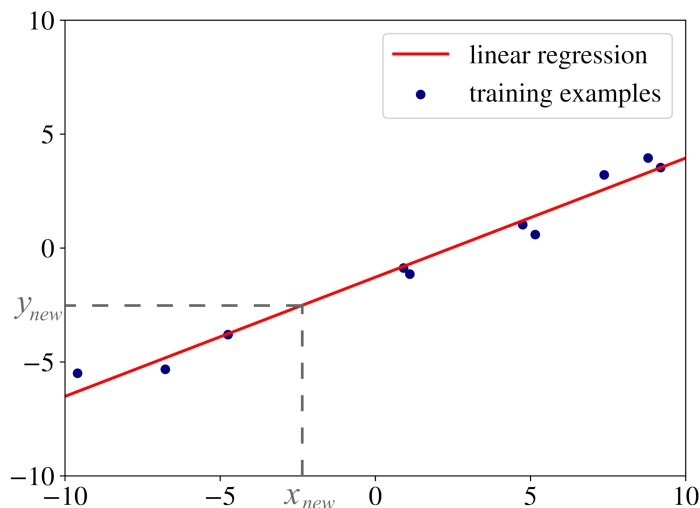


Figure 3.1: Linear Regression for one-dimensional examples.

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

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

### 3.1.2 Solution

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

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

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

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

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

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

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

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

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

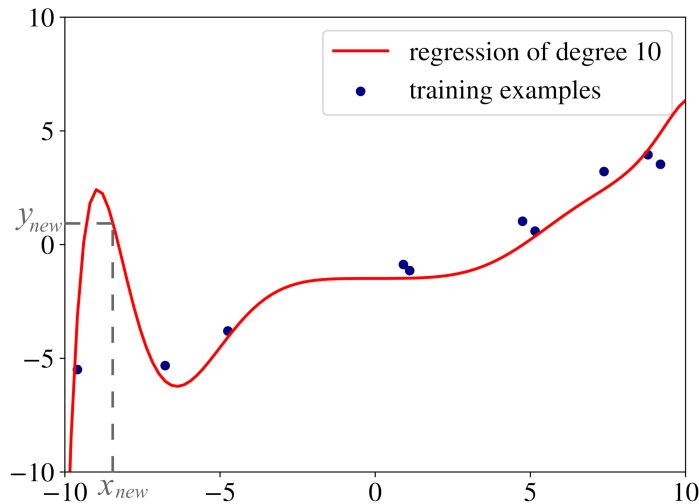


Figure 3.2: Overfitting.

An example of overfitting in regression is shown in Figure 3.2. The data used to build the red regression line is the same as in Figure 3.1. The difference is that this time, this is the polynomial regression with a polynomial of degree 10. The regression line predicts almost perfectly the targets almost all training examples, but will likely make significant errors on new data, as you can see in Figure 3.1 for  $x_{new}$ . We talk more about overfitting and how to avoid it in 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 1805, the French mathematician Adrien-Marie Legendre, who first published the sum of squares method for gauging the quality of the model stated that squaring the error before summing is *convenient*. Why did he say that? The absolute value is not convenient, because it doesn't have a continuous derivative, which makes the function not smooth. Functions that are not smooth create unnecessary difficulties when employing linear algebra to find closed form solutions to optimization problems. Closed form solutions to finding an optimum of a function are simple algebraic expressions and are often preferable to using complex numerical optimization methods, such as **gradient descent** (used, among others, to train neural networks).

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

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

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

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

## 3.2 Logistic Regression

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

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

### 3.2.1 Problem Statement

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

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

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

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

The logistic regression model looks like this:

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

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

By looking at the graph of the standard logistic function, we can see how well it fits our classification purpose: if we optimize the values of  $\mathbf{w}$  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.

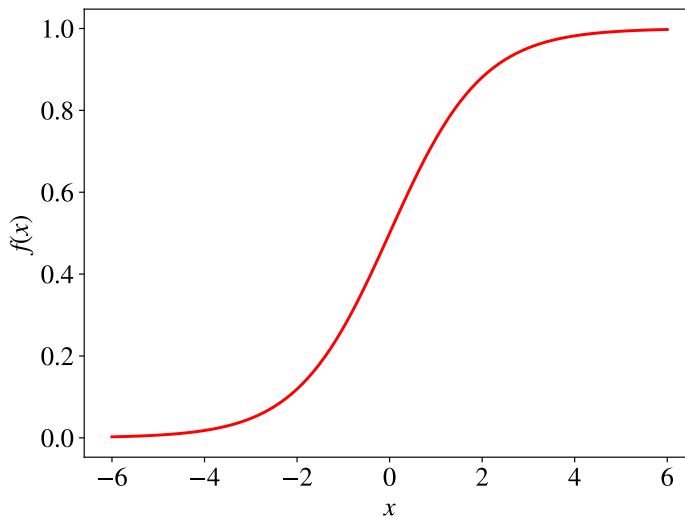


Figure 3.3: Standard logistic function.

### 3.2.2 Solution

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

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

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

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

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

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

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

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

$$\text{Log}L_{\mathbf{w},b} \stackrel{\text{def}}{=} \ln(L_{\mathbf{w},b}(\mathbf{x})) = \sum_{i=1}^N [y_i \ln f_{\mathbf{w},b}(\mathbf{x}) + (1 - y_i) \ln (1 - f_{\mathbf{w},b}(\mathbf{x}))].$$

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:

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

where  $f_{ID3}$  is a decision tree.

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

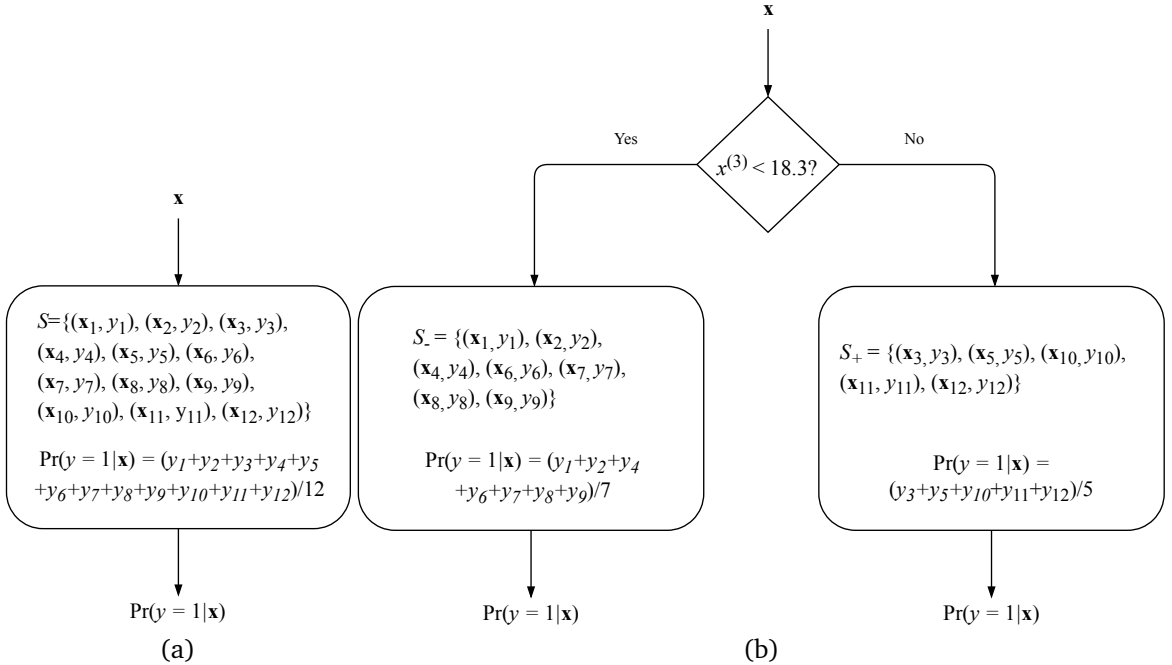


Figure 3.4: An illustration of a decision tree building algorithm. The set  $S$  contains 12 labeled examples. (a) In the beginning, the decision tree only contains the start node; it makes the same prediction for any input. (b) The decision tree after the first split; it tests whether feature 3 is less 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  $S$  denote a set of labeled examples. In the beginning, the decision tree only has a start node that contains all examples:  $S \stackrel{\text{def}}{=} \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ . Start with a constant model  $f_{ID3}^S$  defined as,

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

The prediction given by the above model,  $f_{ID3}^S(\mathbf{x})$ , would be the same for any input  $\mathbf{x}$ . The corresponding decision tree built using a toy dataset of 12 labeled examples is shown in Figure 3.4a.

Then we search through all features  $j = 1, \dots, D$  and all thresholds  $t$ , and split the set  $S$  into two subsets:  $S_- \stackrel{\text{def}}{=} \{(\mathbf{x}, y) \mid (\mathbf{x}, y) \in S, x^{(j)} < t\}$  and  $S_+ \stackrel{\text{def}}{=} \{(\mathbf{x}, y) \mid (\mathbf{x}, y) \in S, x^{(j)} \geq t\}$ . 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  $S_-$  and  $S_+$  is. Finally, we pick the best such values  $(j, t)$ , split  $S$  into  $S_+$  and  $S_-$ , form two new leaf nodes, and continue recursively on  $S_+$  and  $S_-$  (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 Figure 3.4b.

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

$$H(S) \stackrel{\text{def}}{=} -f_{ID3}^S \ln f_{ID3}^S - (1 - f_{ID3}^S) \ln(1 - f_{ID3}^S). \quad (3.7)$$

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

$$H(S_-, S_+) \stackrel{\text{def}}{=} \frac{|S_-|}{|S|} H(S_-) + \frac{|S_+|}{|S|} H(S_+). \quad (3.8)$$

So, in ID3, at each step, at each leaf node, we find a split that minimizes the entropy given by eq. 3.8 or we stop at this leaf node.

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

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

---

<sup>3</sup>In Chapter 5, I show how to do that in the section on hyperparameter tuning.

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

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

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



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

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

criterion. I leave it for further reading.

## 3.4 Support Vector Machine

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

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

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

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

$$\begin{aligned} \mathbf{w}\mathbf{x}_i - b &\geq +1 & \text{if } y_i = +1, \\ \mathbf{w}\mathbf{x}_i - b &\leq -1 & \text{if } y_i = -1. \end{aligned} \tag{3.9}$$

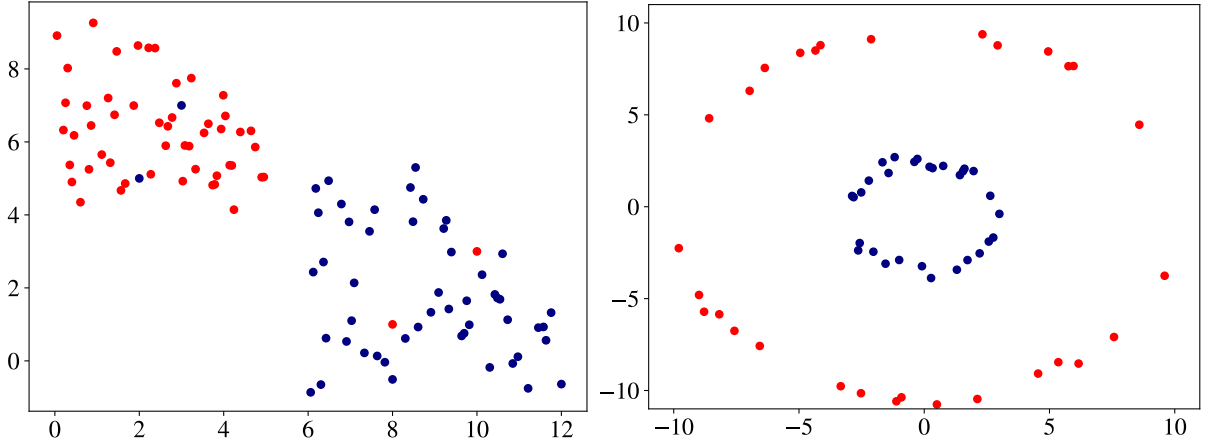


Figure 3.5: Linearly non-separable cases. Left: the presence of noise. Right: inherent nonlinearity.

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

$$\min \frac{1}{2}\|\mathbf{w}\|^2, \text{ such that } y_i(\mathbf{w}\mathbf{x}_i - b) - 1 \geq 0, i = 1, \dots, N. \quad (3.10)$$

### 3.4.1 Dealing with Noise

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

The hinge loss function is zero if the constraints in 3.9 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  $\epsilon$  and  $d$ .

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

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

### 3.4.2 Dealing with Inherent Non-Linearity

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

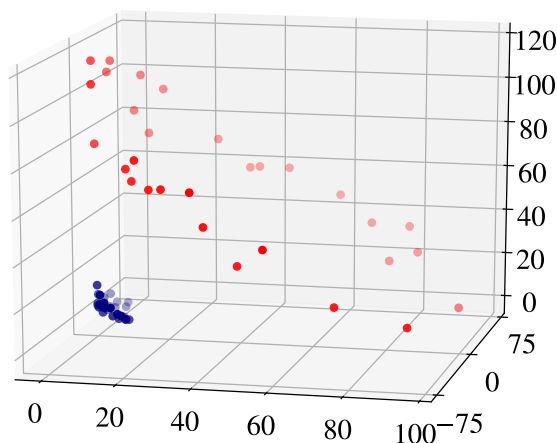


Figure 3.6: The data from Figure 3.5 (right) becomes linearly separable after a transformation into a three-dimensional space.

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

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

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

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

The method traditionally used to solve the optimization problem in eq. 3.10 is the *method of Lagrange multipliers*. Instead of solving the original problem from eq. 3.10, it is convenient to solve an equivalent problem formulated like this:

$$\max_{\alpha_1 \dots \alpha_N} \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{k=1}^N y_i \alpha_i (\mathbf{x}_i \mathbf{x}_k) y_k \alpha_k \text{ subject to } \sum_{i=1}^N \alpha_i y_i = 0 \text{ and } \alpha_i \geq 0, i = 1, \dots, N, \quad (3.11)$$

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

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

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

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



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

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

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

It can be shown that the feature space of the RBF (for “radial basis function”) kernel has an infinite number of dimensions. By varying the hyperparameter  $\sigma$ , 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.

# Appendices



# Appendix A

## C4.5

**C4.5** is one of the most widely used **decision tree** learning algorithms. It can be seen as an improved version of **ID3**, which we saw in Chapter 3. Let's see how C4.5 is different from ID3.

Recall that in ID3, at each leaf node, we looked for a split of the set of examples  $S$  such that the split minimizes the **entropy**  $H(S)$  given by eq. 3.8 (or we stopped at this leaf node). The first difference of C4.5 from ID3 is that, in C4.5, instead of trying to minimize the entropy, we look for a split that maximizes the **information gain**.

What is information gain? Let us have a set  $S$  of examples in the current node of the decision tree, and let us find a feature  $\hat{j}$  that makes the best split in that node. Assume for the moment that we consider only categorical features. Feature  $\hat{j}$  is the one that maximizes the information gain  $G(S, j)$  given by,

$$G(S, j) \stackrel{\text{def}}{=} H(S) - \sum_{k \in V(S, j)} \frac{|\mathcal{S}_{j,k}|}{|S|} H(\mathcal{S}_{j,k}) \quad (\text{A.1})$$

where  $j = 1, \dots, D$  are features,  $H(S)$  is the entropy given by eq. 3.8,  $V(S, j)$  is a function that returns the set of values feature  $j$  takes in examples from set  $S$ , and  $\mathcal{S}_{j,k} \stackrel{\text{def}}{=} \{\mathbf{x} \mid \mathbf{x} \in S, \mathbf{x}^{(j)} = k\}$ . In words,  $\mathcal{S}_{j,k}$  is the subset of examples from set  $S$  in which feature  $j$  has value  $k$ , and  $|\mathcal{S}_{j,k}|$  is the cardinality of that subset.

Once we identified the best feature  $\hat{j}$  to make the split, each of the values of this feature creates a new leaf node in the tree. Let  $k \in V(S, j)$  be a specific value, which feature  $\hat{j}$  may have. To build new leaf nodes,  $S$  is split into several subsets  $\mathcal{S}_{j,k}$ , one for each value of  $k$ .

For example, assume that feature  $\hat{j}$  is “weather” and set  $V(S, j)$  is as follows:  $\{\text{sunny}, \text{clouds}, \text{rain}\}$ . Assume also that there is another feature, “temperature”, in this dataset and, in the current node,  $S$  contains the following examples:

$\{([sunny, 15], 0), ([sunny, 23], 0), ([clouds, 16], 1), ([clouds, 24], 1), ([rain, 18], 0)\}$ ,

where 15, 23, 16, ... are values of “temperature”. After the split, the node would become as shown in Figure 7.

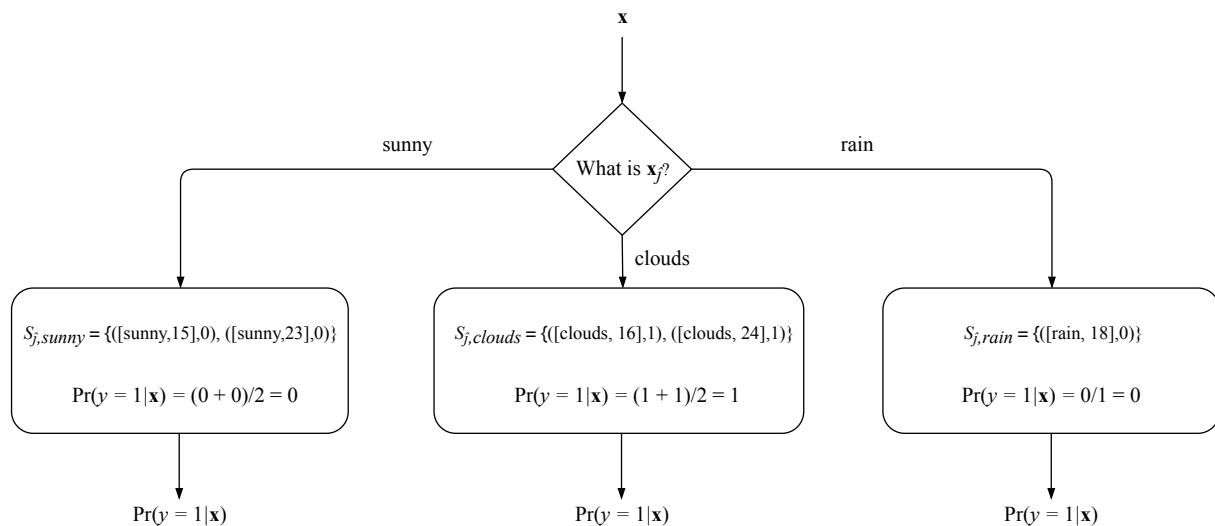


Figure A.1: An example of a split by a categorical feature.

The algorithm might stop after that split, or continue by considering the second feature, “temperature”, for the split in one or several of the three new leaf nodes. (We consider stopping criteria later.)

Let’s now assume that feature  $j$  we consider for the split is numerical like “temperature”, so it has many different values (usually, much more than a categorical feature might have). We want to evaluate the information gain of splitting our current leaf node by this numerical feature.

In C4.5, in order to deal with numerical features, the values of feature  $j$  we consider for the split (for all examples in set  $S$ ) are sorted first. An example of a sorted list of examples by feature  $j$  is shown in Figure 8.

In Figure 8a, the possible splits are shown as vertical red lines. Notice, that splits are binary and only considered in areas of the sorted feature values, in which the label changes. It can be shown that splitting in the areas where the label doesn’t change will not result in an improvement in the information gain. Splitting only in selected areas significantly reduces the amount of computation needed to validate each possible split.

In Figure 8b, one possible split is considered. Then feature  $j$  is considered categorical with two values: “less than or equal to 4.5” (the blue part) and “greater than 4.5” (the green part).

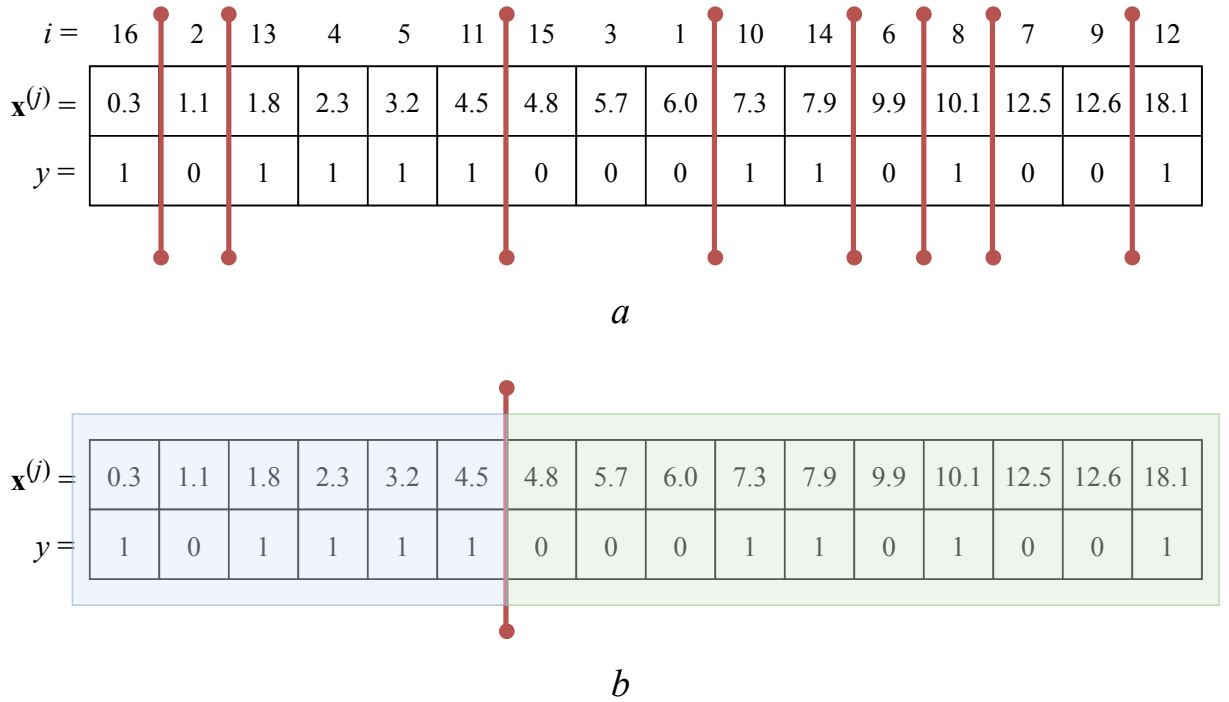


Figure A.2: An example of a split by a numerical feature.

Having two categorical values makes it possible to use eq. 12 to calculate the information gain of the split. In this case,  $V(\mathcal{S}, j)$  returns only two values.

Contrary to ID3, C4.5 can deal with missing values. The idea is quite simple. Assume that we made a split for feature  $j = 3$  as shown in Figure 8b. Then the current leaf node is split into two nodes as shown in Figure 9.

If, in the input example  $\mathbf{x}$ , the value of feature  $j = 3$  is absent then the prediction is made in two leaf nodes at once. In the left leaf node,  $y = 1$  with probability 0.83; in the right leaf node,  $y = 1$  with probability 0.4; the final prediction is  $y = 1$  with probability  $0.83 \cdot \frac{6}{16} + 0.4 \cdot \frac{10}{16} = 0.5625$ . If the decision threshold is 0.5 then the prediction will be  $y = 1$ .

The C4.5 algorithm stops in some leaf node (decides not to split it) in one of the following cases:

- All the examples in the leaf node belong to the same class.
- None of the features provide any information gain.

You could also add additional stopping criteria and use them as hyperparameters. For example, you can decide that the algorithm will stop in a leaf node if:

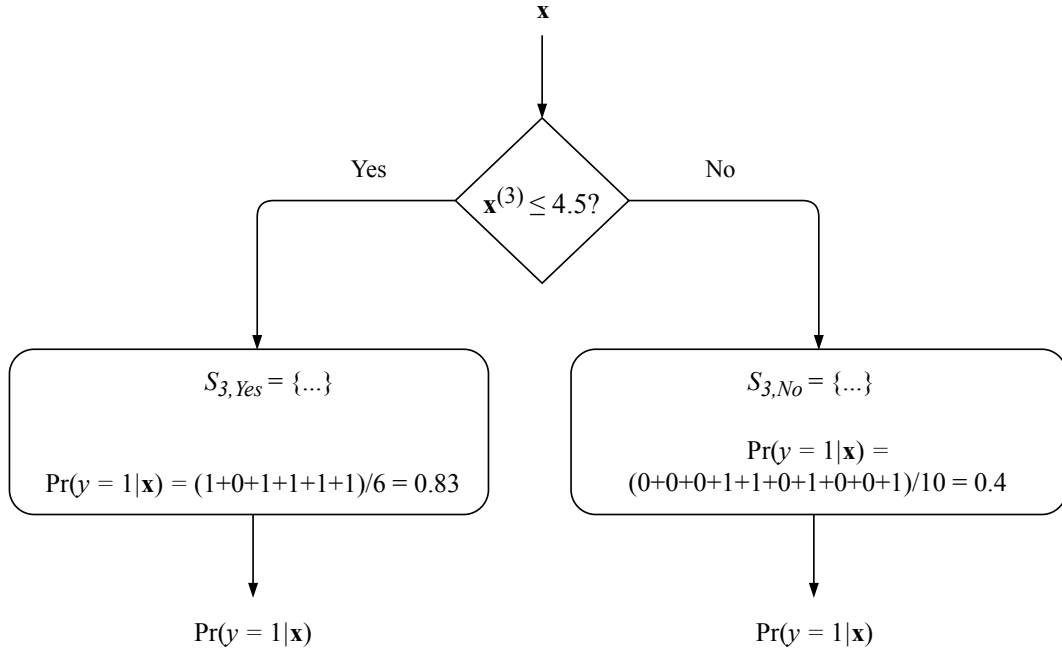


Figure A.3: An decision node for a numerical feature.

- The number of examples in this leaf node is below a certain threshold.
- The tree is already deep enough.

One important feature of C4.5 that differs it from ID3 is that in the former there is a built-in overfitting prevention mechanism. Once the tree is built, C4.5 replaces some branches (also called **subtrees**) by leaf nodes. Doing that reduces **variance** (but inevitably increases **bias** as we remember from the **bias-variance tradeoff** principle).

One possible way to decide whether to keep a subtree or replace it with a leaf is to apply the tree to the examples from the validation set and measure the error made in different leaves. If the weighted sum of errors made in the leaves of some branch is higher than the error that would have been made should the tree stop one level earlier, then the branch is replaced by the leaf.

Consider the part of the tree in Figure 9. Imagine we have 1000 examples in the validation set. Let among those 1000 examples, 50 ended up in the left leaf in Figure 9 and 120 ended up in the right leaf (let the remaining  $1000 - 50 - 120 = 830$  examples end up in some other leaves of the decision tree not shown in Figure 9). Let 10 among 50 predictions were wrong and 20 among 120 predictions were wrong. We then have the following weighted sum of errors:  $\frac{10}{50} \cdot \frac{6}{16} + \frac{20}{120} \cdot \frac{10}{16} = 0.18$ .

Now consider that we replace the branch in Figure 9 by a leaf shown in Figure 10). If among 170 examples from the validation set that ended up in that leaf, for 30 or fewer example the prediction was wrong, then we will replace the branch in Figure 9 by the leaf in Figure 10). Otherwise, we keep the branch.

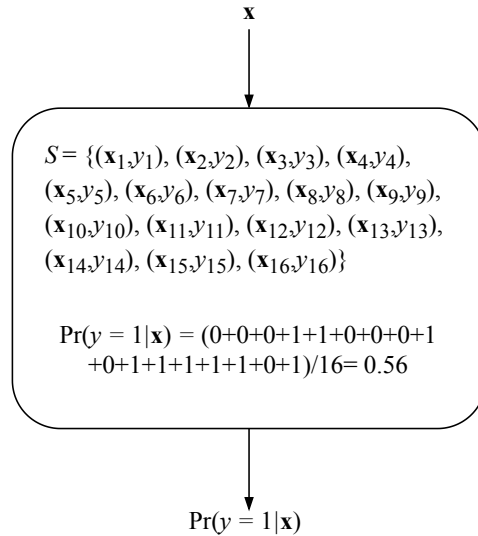


Figure A.4: Example of subtree pruning.

We keep pruning the tree as long as pruning helps to reduce the error on the validation data.





# Appendix B

## SVM Dual Derivation

You probably wonder how did we get to the SVM problem formulation given by eq. 3.11 from the original formulation given by eq. 3.10. As I already mentioned in Chapter 3, we used the method of Lagrange multipliers. Let's see what this method does.

### B.1 Method of Lagrange Multipliers

Let us have a function  $f(\mathbf{x})$  to minimize given some constraints in the form  $g_i(\mathbf{x}) \geq 0$  for  $i = 1, \dots, N$ . The optimal solution to the above problem can be found by optimizing the **Lagrangian** of  $f$  given by,

$$L_{\alpha_1, \dots, \alpha_N}(\mathbf{x}) \stackrel{\text{def}}{=} f(\mathbf{x}) - \sum_{i=1}^N \alpha_i g_i(\mathbf{x}).$$

It can be shown that to find the minimum of the function  $f(\mathbf{x})$  subject to constraints  $g_i(\mathbf{x}) \geq 0$  it is sufficient to find the **stationary point** of the Lagrangian in which we maximize with respect to all  $\alpha_i$  and minimize with respect to  $\mathbf{x}$ . The stationary point of a function is a point in which its gradient is zero (that is, all partial derivatives with respect to all  $\alpha_i$  and  $\mathbf{x}$  are zero). If our constraints were equalities, finding the stationary point such that  $\alpha_i \geq 0$  for  $i = 1, \dots, N$  would suffice. However, for inequalities, the Karush-Kuhn-Tucker (KKT) conditions have to be satisfied for all  $i$ :

$$\begin{aligned} g_i(\mathbf{x}) &\geq 0 \\ \alpha_i &\geq 0 \\ \alpha_i g_i(\mathbf{x}) &= 0 \end{aligned}$$

## B.2 Lagrange Multipliers in SVM

Let's write the original formulation of the SVM problem here once again:

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

In the above formulation,  $f$  is given by  $\frac{1}{2} \|\mathbf{w}\|^2$  while  $g_i$  is given by  $y_i(\mathbf{w}\mathbf{x}_i - b) - 1 \geq 0$ . The Lagrangian is then given by,

$$L_{\alpha_1, \dots, \alpha_N}(\mathbf{w}, b) \stackrel{\text{def}}{=} \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i [y_i(\mathbf{w}\mathbf{x}_i - b) - 1],$$

and the KKT conditions are given by,

$$\begin{aligned} y_i(\mathbf{w}\mathbf{x}_i - b) - 1 &\geq 0 \\ \alpha_i &\geq 0, \\ \alpha_i [y_i(\mathbf{w}\mathbf{x}_i - b) - 1] &= 0 \end{aligned}$$

for  $i = 1, \dots, N$ .

Observe that if we require that  $\alpha_i \geq 0$  for all  $i$ , then in the stationary point,  $y_i(\mathbf{w}\mathbf{x}_i - b) - 1$  is nonnegative for all  $i$ . Why is that? Remember that we minimize the Lagrangian with respect to  $\mathbf{w}$  and  $b$ . For any *fixed* nonnegative values  $\alpha_1, \dots, \alpha_N$ , if  $y_i(\mathbf{w}\mathbf{x}_i - b) - 1$  was negative then the value of the Lagrangian would increase. Therefore, if the constraints of the original SVM formulation are satisfiable (that is, if our data is linearly separable), then nonnegative values of  $\alpha_i$  will push values of  $\mathbf{w}$  and  $b$  towards constraint nonviolation. The constraint nonviolation, in turn, means that  $y_i(\mathbf{w}\mathbf{x}_i - b) - 1$  is nonnegative for all  $i$ .

The optimization problems such as SVM are special in the sense that  $f$  is a **convex function** (that is, it has only one minimum) and constraints  $g_i$  are linear. It can be shown that in such a case, the value of the Lagrangian in the stationary point is exactly the same as the minimum of  $f$ . This means that the term  $\sum_{i=1}^N \alpha_i [y_i(\mathbf{w}\mathbf{x}_i - b) - 1]$ , in the stationary point, equals zero.

Given the above considerations and requiring only one KKT condition satisfied (that is,  $\alpha_i \geq 0$  for all  $i$ ), in the optimal solution to the Lagrangian optimization problem, either  $\alpha_i = 0$  or  $y_i(\mathbf{w}\mathbf{x}_i - b) = 1$ , which satisfies the other two KKT conditions.

Now, how do we obtain eq. 3.11? Remember that in the stationary point, all gradients are zero. Let's set the gradient of the Lagrangian with respect to  $\mathbf{w}$  to zero:

$$\nabla \frac{1}{2} \|\mathbf{w}\|^2 - \nabla \sum_{i=1}^N \alpha_i [y_i(\mathbf{w}\mathbf{x}_i - b) - 1] = 0.$$

By simplifying, we get  $\mathbf{w} - \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i = 0$  or  $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ .

If you wondering how  $\nabla \frac{1}{2} \|\mathbf{w}\|^2$  has become  $\mathbf{w}$ , let's consider a simple example where  $\mathbf{w} \stackrel{\text{def}}{=} (p, q)$ . By using the chain rule, the partial derivative  $\frac{\partial \frac{1}{2} \|\mathbf{w}\|^2}{\partial p} = 2 \frac{1}{2} \sqrt{p^2 + q^2} \cdot \frac{1}{2} (p^2 + q^2)^{-\frac{1}{2}} \cdot 2p = p$ . Similarly,  $\frac{\partial \frac{1}{2} \|\mathbf{w}\|^2}{\partial q} = q$ . By definition of the gradient,  $\nabla \frac{1}{2} \|\mathbf{w}\|^2 = \left( \frac{\partial \frac{1}{2} \|\mathbf{w}\|^2}{\partial p}, \frac{\partial \frac{1}{2} \|\mathbf{w}\|^2}{\partial q} \right) = (p, q)$ . Therefore,  $\nabla \frac{1}{2} \|\mathbf{w}\|^2 = \mathbf{w}$ .

In a similar way, when we set the gradient of the Lagrangian with respect to  $b$  to zero, we obtain  $\sum_{i=1}^N \alpha_i y_i = 0$ .

Now, let's show how we get from eq. 3.10 to eq. 3.11.

$$\begin{aligned} L_{\alpha_1, \dots, \alpha_N}(\mathbf{w}, b) &\stackrel{\text{def}}{=} \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i [y_i (\mathbf{w} \mathbf{x}_i - b) - 1] \\ &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i y_i \mathbf{w} \mathbf{x}_i - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \end{aligned}$$

Because  $\sum_{i=1}^N \alpha_i y_i = 0$ , we can simplify the Lagrangian:

$$L_{\alpha_1, \dots, \alpha_N}(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i y_i \mathbf{w} \mathbf{x}_i + \sum_{i=1}^N \alpha_i.$$

Because  $\mathbf{w}$  is independent of  $i$ , we can write,

$$L_{\alpha_1, \dots, \alpha_N}(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 - \mathbf{w} \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i + \sum_{i=1}^N \alpha_i.$$

Because  $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ , we can simplify the Lagrangian:

$$\begin{aligned} L_{\alpha_1, \dots, \alpha_N}(\mathbf{w}, b) &= \frac{1}{2} \|\mathbf{w}\|^2 - \mathbf{w} \mathbf{w} + \sum_{i=1}^N \alpha_i \\ &= \frac{1}{2} \mathbf{w} \mathbf{w} - \mathbf{w} \mathbf{w} + \sum_{i=1}^N \alpha_i \\ &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \mathbf{w} \mathbf{w} \\ &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{k=1}^N y_i \alpha_i \cdot (\mathbf{x}_i \mathbf{x}_k) y_k \alpha_k. \end{aligned}$$

Notice that the above expression is exactly the same as in eq. 3.11. This form of the SVM problem formulation is called **dual**, while the original form given by equation eq. 3.10 is called **primal**. Because the dual is independent of  $\mathbf{w}$ , we can drop the constraint  $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ , which gives us exactly the formulation of the problem from Chapter 3:

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

By solving the above maximization problem, we find the optimal values of  $\alpha_1 \dots \alpha_N$ . As we dicussed above, they can be either zero or strictly positive. When  $\alpha_i$  is strictly positive, the corresponding  $\mathbf{x}_i$  is called the **support vector**. To make predictions, only support vectors are used, so the predictions in SVM are very fast:

$$y \leftarrow \text{model}(\mathbf{x}) \stackrel{\text{def}}{=} \text{sign}(\mathbf{w}\mathbf{x} - b) = \text{sign} \left( \sum_{i=1}^N \alpha_i y_i \cdot (\mathbf{x}_i \mathbf{x}) - b \right),$$

or, by using a kernel,

$$y \leftarrow \text{model}(\mathbf{x}) \stackrel{\text{def}}{=} \text{sign} \left( \sum_{i=1}^N \alpha_i y_i k(\mathbf{x}_i \mathbf{x}) - b \right).$$

To find the optimal value of  $b$ , we can use any of the support vectors. We know that for a support vector  $\mathbf{x}_i$ , the corresponding  $\alpha_i$  is strictly positive. Because the third KKT constraint is satisfied for all  $i$ , this means that the corresponding  $[y_i(\mathbf{w}\mathbf{x}_i - b) - 1] = 0$ . Therefore,  $b = \mathbf{w}\mathbf{x}_i - \frac{1}{y_i} = \mathbf{w}\mathbf{x}_i - y_i$  (because  $y \in \{-1, +1\}$ ).

Knowing that  $\mathbf{w} = \sum_{i=k}^N \alpha_k y_k \mathbf{x}_k$ , we can write,

$$b = \sum_{k=1}^N \alpha_k y_k \cdot (\mathbf{x}_k \mathbf{x}_i) - y_i,$$

or, by using a kernel,

$$b = \sum_{k=1}^N \alpha_k y_k k(\mathbf{x}_k \mathbf{x}_i) - y_i.$$

The only reason why, to build the SVM model, we solve the dual and not the primal is that in the dual, we can use kernels (by replacing  $(\mathbf{x}_i \mathbf{x}_k)$  by  $k(\mathbf{x}_i \mathbf{x}_k)$  in eq. A.2).

"Burkov has undertaken a very useful but impossibly hard task in reducing all of machine learning to 100 pages. He succeeds well in choosing the topics—both theory and practice—that will be useful to practitioners, and for the reader who understands that this is the first 100 (or actually 150) pages you will read, not the last, provides a solid introduction to the field."

**Peter Norvig**, *Research Director at Google, author of the most popular artificial intelligence textbook in the world* ***Artificial Intelligence: A Modern Approach***

"The breadth of topics the book covers is amazing for just 100 pages (plus few bonus pages!). Burkov doesn't hesitate to go into the math equations: that's one thing that short books usually drop. I really liked how the author explains the core concepts in just a few words. The book can be very useful for newcomers in the field, as well as for old-timers who can gain from such a broad view of the field."

**Aurélien Géron**, *Senior Artificial Intelligence Engineer, author of a #1 Amazon bestseller* ***Hands-On Machine Learning with Scikit-Learn and TensorFlow***