SUPPORT VECTOR MACHINES

SUCCINCTLY

BY ALEXANDRE KOWALCZYK

SUCCINCTLY E-BOOK SERIES

Syncfusion

Support Vector Machines Succinctly

By
Alexandre Kowalczyk

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: John Elderkin

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Alexandre Kowalczyk is a software developer at ABC Arbitrage, a financial company doing automated trading on the stock market, and a certified Microsoft Specialist in C#.

Alexandre first encountered Support Vector Machines (SVMs) while attending the Andrew Ng online course on Machine Learning three years ago. Since then, he has successfully used SVMs on several projects, including real-time news classification.

In his spare time, he participates in Kaggle contests. He has used SVM implementations in C#, R, and Python to classify plankton images, Greek news, and products into categories, and to predict physical and chemical properties of soil using spectral measurements.

Alexandre has spent two years studying SVMs, allowing him to understand how they work. Because it was difficult to find a simple overview of the subject, he started the blog SVM Tutorial, where he explains SVMs as simply as he can.

He hopes this book will help you understand SVMs and provide you with another tool in your machine-learning toolbox.

Acknowledgments

I would like to thank Syncfusion for providing me the opportunity to write this book, Grégory Godin for taking the time to read and review it, and James McCaffrey for his in-depth technical review.

Dedication

I dedicate this book to my mother, Claudine Kowalczyk (1954–2003).

Preface

Who is this book for?

This book's aim is to provide a general overview of Support Vector Machines (SVMs). You will learn what they are, which kinds of problems they can solve, and how to use them. I tried to make this book useful for many categories of readers. Software engineers will find a lot of code examples alongside simple explanations of the algorithms. A deeper understanding of how SVMs work internally will enable you to make better use of the available implementations.

Students looking to take a first look at SVMs will find a large enough coverage of the subject to spike their curiosity. I also tried to include as many references as I could so that the interested reader can dive deeper.

How should you read this book?

Because each chapter is built on the previous one, reading this book sequentially is the preferred method.

References

You will find a bibliography at the end of the book. A reference to a paper or book is made with the name of the author followed by the publication date. For instance, (Bishop, 2006) refers to the following line in the bibliography:

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* Springer.

Code listings

The code listings in this book have been created using the Pycharm IDE, Community Edition 2016.2.3, and executed with WinPython 64-bit 3.5.1.2 version of Python and NumPy.
You can find the code source associated with this book in this Bitbucket.

Introduction

Support Vector Machine is one of the most performant off-the-shelf supervised machine learning algorithms. This means that when you have a problem and you try to run a SVM on it, you will often get pretty good results without many tweaks. Despite this, because it is based on a strong mathematical background, it is often seen as a black box. In this book, we will go under the hood and look at the main ideas behind SVM. There are several Support Vector Machines, which is why I will often refer to SVMs. The goal of this book is to understand how they work.

SVMs are the result of the work of several people over many years. The first SVM algorithm is attributed to Vladimir Vapnik in 1963. He later worked closely with Alexey Chervonenkis on what is known as the VC theory, which attempts to explain the learning process from a statistical point of view, and they both contributed greatly to the SVM. You can find a very detailed history of SVMs here.

In real life, SVMs have been successfully used in three main areas: text categorization, image recognition, and bioinformatics (Cristianini & Shawe-Taylor, 2000). Specific examples include classifying news stories, handwritten digit recognition, and cancer tissue samples.

In the first chapter, we will consider important concepts: vectors, linear separability, and hyperplanes. They are the building blocks that will allow you to understand SVMs. In Chapter 2, instead of jumping right into the subject, we will study a simple algorithm known as the Perceptron. Do not skip it—even though it does not discuss SVMs, this chapter will give you precious insight into why SVMs are better at classifying data.

Chapter 3 will be used to step-by-step construct what is known as the SVM optimization problem. Chapter 4, which is probably the hardest, will show you how to solve this problem—first mathematically, then programmatically. In Chapter 5, we will discover a new support vector machine known as the Soft-margin SVM. We will see how it is a crucial improvement to the original problem.

Chapter 6 will introduce kernels and will explain the so called “kernel trick.” With this trick, we will get the kernelized SVM, which is the most-used nowadays. In Chapter 7, we will learn about SMO, an algorithm specifically created to quickly solve the SVM optimization problem. In Chapter 8, we will see that SVMs can be used to classify more than one class.

Every chapter contains code samples and figures so that you can understand the concepts more easily. Of course, this book cannot cover every subject, and some of them will not be presented. In the conclusion, you will find pointers toward what you can learn next about SVMs.

Let us now begin our journey.

Chapter 1 Prerequisites

This chapter introduces some basics you need to know in order to understand SVMs better. We will first see what vectors are and look at some of their key properties. Then we will learn what it means for data to be linearly separable before introducing a key component: the hyperplane.

Vectors

In Support Vector Machine, there is the word **vector.** It is important to know some basics about vectors in order to understand SVMs and how to use them.

What is a vector?

A vector is a mathematical object that can be represented by an arrow (Figure 1).

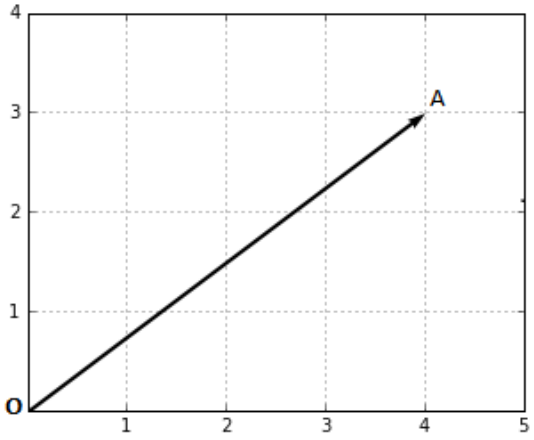

Figure 1: Representation of a vector

When we do calculations, we denote a vector with the coordinates of its endpoint (the point where the tip of the arrow is). In Figure 1, the point A has the coordinates (4,3). We can write:

$$\vec{OA} = (4, 3)$$

If we want to, we can give another name to the vector, for instance, $\mathbf{a}$.

$$\mathbf{a} = (4, 3)$$

From this point, one might be tempted to think that a vector is defined by its coordinates. However, if I give you a sheet of paper with only a horizontal line and ask you to trace the same vector as the one in Figure 1, you can still do it.

You need only two pieces of information:

- What is the length of the vector?
- What is the angle between the vector and the horizontal line?

This leads us to the following definition of a vector:

*A **vector** is an object that has both a magnitude and a direction.*

Let us take a closer look at each of these components.

The magnitude of a vector

The magnitude, or length, of a vector $\mathbf{x}$ is written $\|\mathbf{x}\|$, and is called its **norm**.

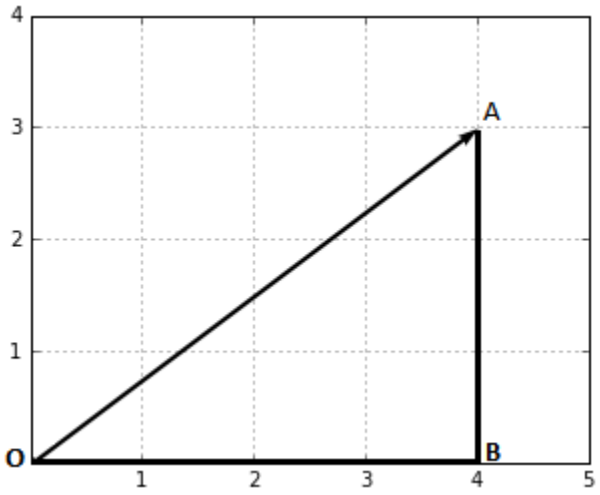

Figure 2: The magnitude of this vector is the length of the segment OA

In Figure 2, we can calculate the norm $\|OA\|$ of vector $\vec{OA}$ by using the Pythagorean theorem:

$$OA^2 = OB^2 + AB^2$$

$$OA^2 = 4^2 + 3^2$$

$$OA^2 = 25$$

$$OA = \sqrt{25}$$

$$\|OA\| = OA = 5$$

In general, we compute the norm of a vector $\mathbf{x} = (x_1, \cdots, x_n)$ by using the **Euclidean norm** formula:

$$\|\mathbf{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}$$

In Python, computing the norm can easily be done by calling the **norm** function provided by the **numpy** module, as shown in Code Listing 1.

Code Listing 1

```
import numpy as np

x = [3,4]
np.linalg.norm(x) # 5.0
```

The direction of a vector

The direction is the second component of a vector. By definition, it is a new vector for which the coordinates are the initial coordinates of our vector divided by its norm.

The **direction** of a vector $\mathbf{u} = (u_1, u_2)$ is the vector:

$$\mathbf{w} = \left(\frac{u_1}{||u||}, \frac{u_2}{||u||} \right)$$

It can be computed in Python using the code in Code Listing 2.

Code Listing 2

```
import numpy as np

# Compute the direction of a vector x.
def direction(x):
    return x/np.linalg.norm(x)
```

Where does it come from? Geometry. Figure 3 shows us a vector $\mathbf{u}$ and its angles with respect to the horizontal and vertical axis. There is an angle θ (theta) between $\mathbf{u}$ and the horizontal axis, and there is an angle α (alpha) between $\mathbf{u}$ and the vertical axis.

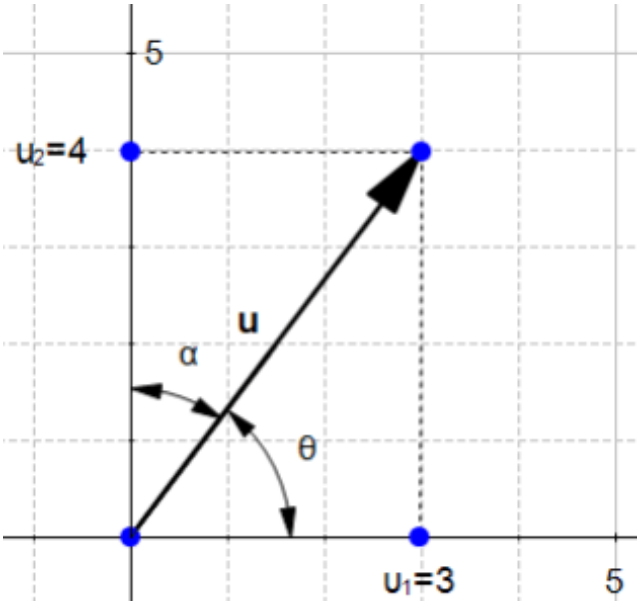

Figure 3: A vector u and its angles with respect to the axis

Using elementary geometry, we see that $cos(\theta) = \frac{u_1}{||u||}$ and $cos(\alpha) = \frac{u_2}{||u||}$, which means that $\mathbf{w}$ can also be defined by:

$$\mathbf{w} = \left(\frac{u_1}{||u||}, \frac{u_2}{||u||}\right) = (cos(\theta), cos(\alpha))$$

The coordinates of $\mathbf{w}$ are defined by cosines. As a result, if the angle between $\mathbf{u}$ and an axis changes, which means the direction of $\mathbf{u}$ changes, $\mathbf{w}$ will also change. That is why we call this vector the direction of vector $\mathbf{u}$. We can compute the value of $\mathbf{w}$ (Code Listing 3), and we find that its coordinates are $(0.6, 0.8)$.

Code Listing 3

```
u = np.array([3,4])
w = direction(u)

print(w) # [0.6 , 0.8]
```

It is interesting to note is that if two vectors have the same direction, they will have the same direction vector (Code Listing 4).

Code Listing 4

```
u_1 = np.array([3,4])
u_2 = np.array([30,40])

print(direction(u_1)) # [0.6 , 0.8]
print(direction(u_2)) # [0.6 , 0.8]
```

Moreover, **the norm of a direction vector is always 1**. We can verify that with the vector $\mathbf{w} = (0.6, 0.8)$ (Code Listing 5).

Code Listing 5

```
np.linalg.norm(np.array([0.6, 0.8])) # 1.0
```

It makes sense, as the sole objective of this vector is to describe the direction of other vectors—by having a norm of 1, it stays as simple as possible. As a result, a direction vector such as $\mathbf{w}$ is often referred to as a **unit vector**.

Dimensions of a vector

Note that the order in which the numbers are written is important. As a result, we say that a n-dimensional vector is a tuple of n real-valued numbers.

For instance, $\mathbf{w} = (0.6, 0.8)$ is a two-dimensional vector; we often write $\mathbf{w} \in \mathbb{R}^2$ ($\mathbf{w}$ belongs to $\mathbb{R}^2$). Similarly, the vector $\mathbf{u} = (5, 3, 2)$ is a three-dimensional vector, and $\mathbf{u} \in \mathbb{R}^3$.

The dot product

The **dot product** is an operation performed on two vectors that returns a number. A number is sometimes called a **scalar**; that is why the dot product is also called a **scalar product**.

People often have trouble with the dot product because it seems to come out of nowhere. What is important is that it is an operation performed on **two vectors** and that its result gives us some insights into how the two vectors relate to each other. There are two ways to think about the dot product: geometrically and algebraically.

Geometric definition of the dot product

Geometrically, the **dot product** is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them.

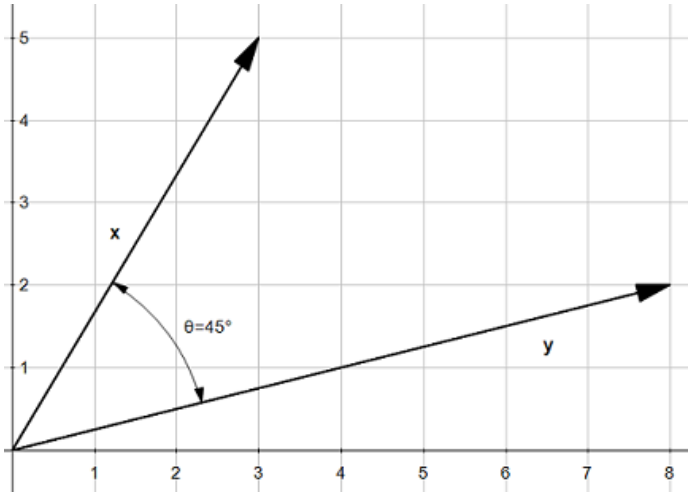

Figure 4: Two vectors x and y

This means that if we have two vectors, $\mathbf{x}$ and $\mathbf{y}$, with an angle θ between them (Figure 4), their dot product is:

$$\mathbf{x} \cdot \mathbf{y} = ||x||\ ||y||\ cos(\theta)$$

By looking at this formula, we can see that the dot product is strongly influenced by the angle θ:

- When $\theta = 0°$, we have $cos(\theta) = 1$ and $\mathbf{x} \cdot \mathbf{y} = ||x||\ ||y||$
- When $\theta = 90°$, we have $cos(\theta) = 0$ and $\mathbf{x} \cdot \mathbf{y} = 0$
- When $\theta = 180°$, we have $cos(\theta) = -1$ and $\mathbf{x} \cdot \mathbf{y} = -||x||\ ||y||$

Keep this in mind—it will be useful later when we study the Perceptron learning algorithm.

We can write a simple Python function to compute the dot product using this definition (Code Listing 6) and use it to get the value of the dot product in Figure 4 (Code Listing 7).

Code Listing 6

```
import math
import numpy as np

def geometric_dot_product(x,y, theta):
    x_norm = np.linalg.norm(x)
    y_norm = np.linalg.norm(y)
    return x_norm * y_norm * math.cos(math.radians(theta))
```

However, we need to know the value of θ to be able to compute the dot product.

Code Listing 7

```
theta = 45

x = [3,5]
y = [8,2]


print(geometric_dot_product(x,y,theta))  # 34.0
```

Algebraic definition of the dot product

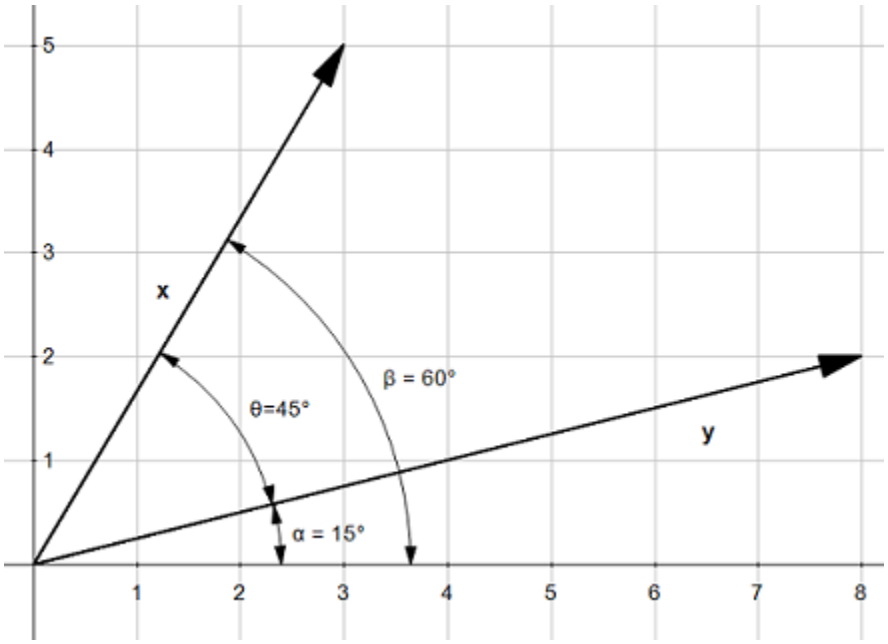

Figure 5: Using these three angles will allow us to simplify the dot product

In Figure 5, we can see the relationship between the three angles θ, β (beta), and α (alpha):

$$\theta = \beta - \alpha$$

This means computing $cos(\theta)$ is the same as computing $cos(\beta - \alpha)$.

Using the difference identity for cosine we get:

$$cos(\theta) = cos(\beta - \alpha) = cos(\beta)cos(\alpha) + sin(\beta)sin(\alpha)$$

$$cos(\theta) = \frac{x_1}{||x||}\frac{y_1}{||y||} + \frac{x_2}{||x||}\frac{y_2}{||y||}$$

$$cos(\theta) = \frac{x_1 y_1 + x_2 y_2}{||x||||y||}$$

If we multiply both sides by $||x||||y||$ we get:

$$||x||||y||cos(\theta) = x_1y_1 + x_2y_2$$

We already know that:

$$||x||||y||cos(\theta) = \mathbf{x} \cdot \mathbf{y}$$

This means the dot product can also be written:

$$\mathbf{x} \cdot \mathbf{y} = x_1y_1 + x_2y_2$$

Or:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{2}(x_iy_i)$$

In a more general way, for n-dimensional vectors, we can write:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n}(x_iy_i)$$

This formula is the **algebraic definition of the dot product**.

Code Listing 8

```
def dot_product(x,y):
    result = 0
    for i in range(len(x)):
        result = result + x[i]*y[i]
    return result
```

This definition is advantageous because we do not have to know the angle θ to compute the dot product. We can write a function to compute its value (Code Listing 8) and get the same result as with the geometric definition (Code Listing 9).

Code Listing 9

```
x = [3,5]
y = [8,2]
print(dot_product(x,y)) # 34
```

Of course, we can also use the dot function provided by **numpy** (Code Listing 10).

Code Listing 10

```
import numpy as np

x = np.array([3,5])
y = np.array([8,2])

print(np.dot(x,y)) # 34
```

We spent quite some time understanding what the dot product is and how it is computed. This is because the dot product is a fundamental notion that you should be comfortable with in order to figure out what is going on in SVMs. We will now see another crucial aspect, linear separability.

Understanding linear separability

In this section, we will use a simple example to introduce linear separability.

Linearly separable data

Imagine you are a wine producer. You sell wine coming from two different production batches:

- One high-end wine costing $145 a bottle.
- One common wine costing $8 a bottle.

Recently, you started to receive complaints from clients who bought an expensive bottle. They claim that their bottle contains the cheap wine. This results in a major reputation loss for your company, and customers stop ordering your wine.

Using alcohol-by-volume to classify wine

You decide to find a way to distinguish the two wines. You know that one of them contains more alcohol than the other, so you open a few bottles, measure the alcohol concentration, and plot it.

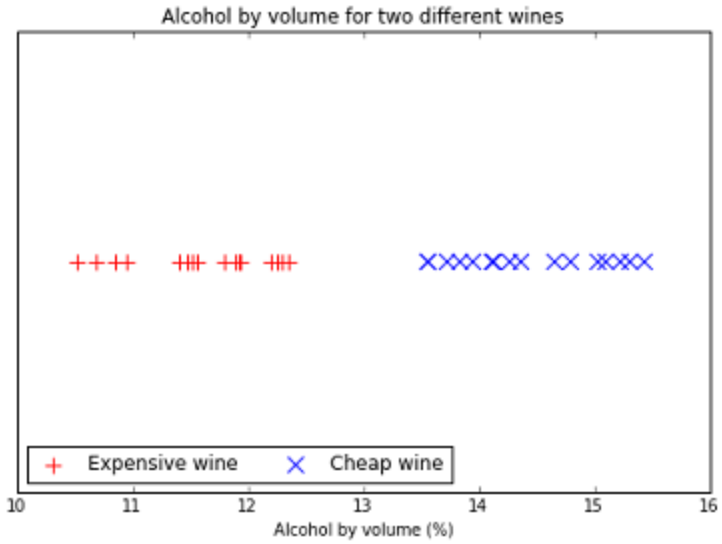

Figure 6: An example of linearly separable data

In Figure 6, you can clearly see that the expensive wine contains less alcohol than the cheap one. In fact, you can find a point that separates the data into two groups. This data is said to be **linearly separable**. For now, you decide to measure the alcohol concentration of your wine automatically before filling an expensive bottle. If it is greater than 13 percent, the production chain stops and one of your employee must make an inspection. This improvement dramatically reduces complaints, and your business is flourishing again.

This example is too easy—in reality, data seldom works like that. In fact, some scientists really measured alcohol concentration of wine, and the plot they obtained is shown in Figure 7. This is an example of non-linearly separable data. Even if most of the time data will not be linearly separable, it is fundamental that you understand linear separability well. In most cases, we will start from the linearly separable case (because it is the simpler) and then derive the non-separable case.

Similarly, in most problems, we will not work with only one dimension, as in Figure 6. Real-life problems are more challenging than toy examples, and some of them can have thousands of dimensions, which makes working with them more abstract. However, its abstractness does not make it more complex. Most examples in this book will be two-dimensional examples. They are simple enough to be easily visualized, and we can do some basic geometry on them, which will allow you to understand the fundamentals of SVMs.

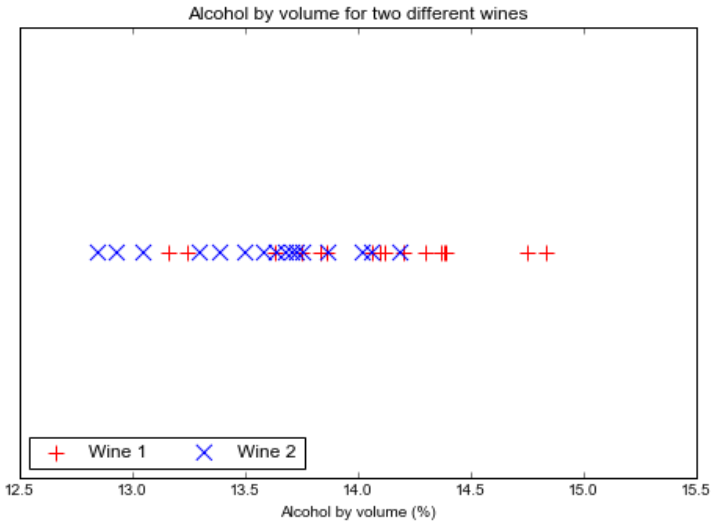

Figure 7: Plotting alcohol by volume from a real dataset

In our example of Figure 6, there is only one dimension: that is, each data point is represented by a single number. When there are more dimensions, we will use vectors to represent each data point. Every time we add a dimension, the object we use to separate the data changes. Indeed, while we can separate the data with a single point in Figure 6, as soon as we go into two dimensions we need a line (a set of points), and in three dimensions we need a plane (which is also a set of points).

To summarize, data is linearly separable when:

- In one dimension, you can find a **point** separating the data (Figure 6).
- In two dimensions, you can find a **line** separating the data (Figure 8).
- In three dimensions, you can find a **plane** separating the data (Figure 9).

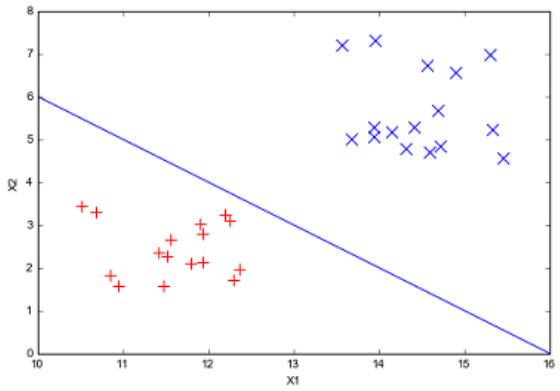

Figure 8: Data separated by a line

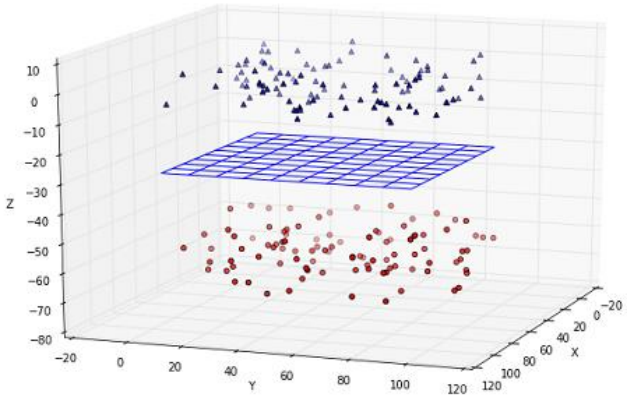

Figure 9: Data separated by a plane

Similarly, when data is non-linearly separable, we **cannot** find a separating point, line, or plane. Figure 10 and Figure 11 show examples of non-linearly separable data in two and three dimensions.

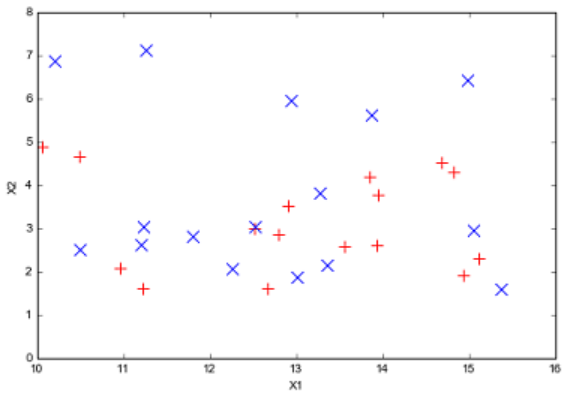

Figure 10: Non-linearly separable data in 2D

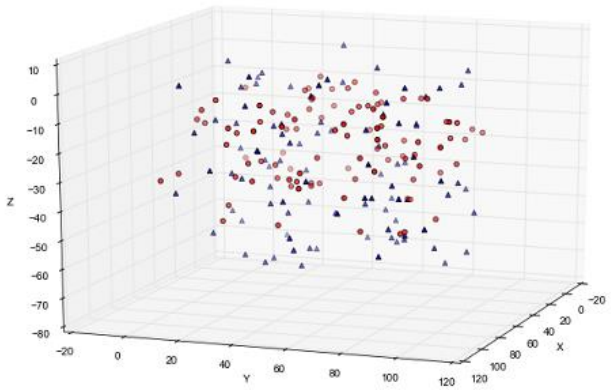

Figure 11: Non-linearly separable data in 3D

Hyperplanes

What do we use to separate the data when there are more than three dimensions? We use what is called a **hyperplane**.

What is a hyperplane?

*In geometry, a **hyperplane** is a subspace of one dimension less than its ambient space.*

This definition, albeit true, is not very intuitive. Instead of using it, we will try to understand what a hyperplane is by first studying what a line is.

If you recall mathematics from school, you probably learned that a line has an equation of the form $y = ax + b$, that the constant a is known as the slope, and that b intercepts the y-axis. There are several values of x for which this formula is true, and we say that the set of the solutions is a line.

What is often confusing is that if you study the function $f(x) = ax + b$ in a calculus course, you will be studying a function with one variable.

However, it is important to note that the linear equation $y = ax + b$ has two variables, respectively y and x, and we can name them as we want.

For instance, we can rename y as x_2 and x as x_1, and the equation becomes: $x_2 = ax_1 + b$.

This is equivalent to $ax_1 - x_2 + b = 0$.

If we define the two-dimensional vectors $\mathbf{x} = (x_1, x_2)$ and $\mathbf{w} = (a, -1)$, we obtain another notation for the equation of a line (where $\mathbf{w} \cdot \mathbf{x}$ is the dot product of $\mathbf{w}$ and $\mathbf{x}$):

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

What is nice with this last equation is that it uses vectors. Even if we derived it by using two-dimensional vectors, it works for vectors of any dimensions. It is, in fact, the equation of a **hyperplane**.

From this equation, we can have another insight into what a hyperplane is: it is the set of points satisfying $\mathbf{w} \cdot \mathbf{x} + b = 0$. And, if we keep just the essence of this definition: **a hyperplane is a set of points**.

If we have been able to deduce the hyperplane equation from the equation of a line, it is because a line *is* a hyperplane. You can convince yourself by reading the definition of a hyperplane again. You will notice that, indeed, a line is a two-dimensional space surrounded by a plane that has three dimensions. Similarly, points and planes are hyperplanes, too.

Understanding the hyperplane equation

We derived the equation of a hyperplane from the equation of a line. Doing the opposite is interesting, as it shows us more clearly the relationship between the two.

Given vectors $\mathbf{w} = (w_0, w_1)$, $\mathbf{x} = (x, y)$ and b , we can define a hyperplane having the equation:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

This is equivalent to:

$$w_0 x + w_1 y + b = 0$$

$$w_1 y = -w_0 x - b$$

We isolate y to get:

$$y = -\frac{w_0}{w_1} x - \frac{b}{w_1}$$

If we define a and c :

$$a = -\frac{w_0}{w_1} \text{ and } c = -\frac{b}{w_1}$$

$$y = ax + c$$

We see that the bias c of the line equation is only equal to the bias b of the hyperplane equation when $w_1 = -1$. So you should not be surprised if b is not the intersection with the vertical axis when you see a plot for a hyperplane (this will be the case in our next example). Moreover, if w_0 and w_1 have the same sign, the slope a will be negative.

Classifying data with a hyperplane

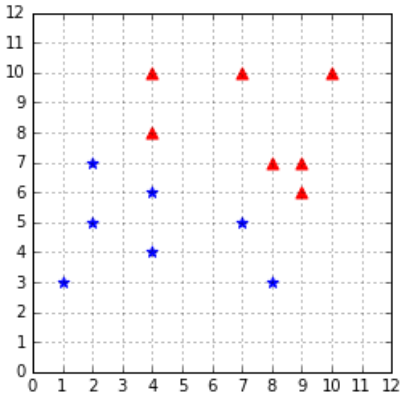

Figure 12: A linearly separable dataset

Given the linearly separable data of Figure 12, we can use a hyperplane to perform binary classification.

For instance, with the vector $\mathbf{w} = (0.4, 1.0)$ and $b = -9$ we get the hyperplane in Figure 13.

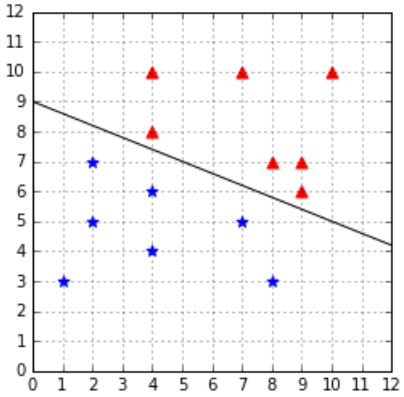

Figure 13: A hyperplane separates the data

We associate each vector $\mathbf{x}_i$ with a label y_i, which can have the value $+1$ or -1 (respectively the triangles and the stars in Figure 13).

We define a hypothesis function h:

$$h(\mathbf{x}_i) = \begin{cases} +1 & \text{if} \quad \mathbf{w} \cdot \mathbf{x}_i + b \geq 0 \\ -1 & \text{if} \quad \mathbf{w} \cdot \mathbf{x}_i + b < 0 \end{cases}$$

which is equivalent to:

$$h(\mathbf{x}_i) = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i + b)$$

It uses the position of $\mathbf{x}$ with respect to the hyperplane to predict a value for the label y. Every data point on one side of the hyperplane will be assigned a label, and every data point on the other side will be assigned the other label.

For instance, for $\mathbf{x} = (8, 7)$, $\mathbf{x}$ is above the hyperplane. When we do the calculation, we get $\mathbf{w} \cdot \mathbf{x} + b = 0.4 \times 8 + 1 \times 7 - 9 = 1.2$, which is positive, so $h(\mathbf{x}) = +1$.

Similarly, for $\mathbf{x} = (1, 3)$, $\mathbf{x}$ is below the hyperplane, and h will return -1 because $\mathbf{w} \cdot \mathbf{x} + b = 0.4 \times 1 + 1 \times 3 - 9 = -5.6$.

Because it uses the equation of the hyperplane, which produces a linear combination of the values, the function h, is called a **linear classifier**.

With one more trick, we can make the formula of h even simpler by removing the b constant. First, we add a component $x_0 = 1$ to the vector $\mathbf{x}_i = (x_1, x_2, ..., x_n)$. We get the vector $\hat{\mathbf{x}}_i = (x_0, x_1, \ldots, x_n)$ (it reads "$\mathbf{x}_i$ hat" because we put a hat on $\mathbf{x}_i$). Similarly, we add a component $w_0 = b$ to the vector $\mathbf{w} = (w_1, w_2, ..., w_n)$, which becomes $\hat{\mathbf{w}} = (w_0, w_1, ..., w_n)$.

Note: In the rest of the book, we will call a vector to which we add an artificial coordinate an augmented vector.

When we use augmented vectors, the hypothesis function becomes:

$$h(\hat{\mathbf{x}}_i) = \text{sign}(\hat{\mathbf{w}} \cdot \hat{\mathbf{x}}_i)$$

If we have a hyperplane that separates the data set like the one in Figure 13, by using the hypothesis function h, we are able to predict the label of every point perfectly.
The main question is: how do we find such a hyperplane?

How can we find a hyperplane (separating the data or not)?

Recall that the equation of the hyperplane is $\mathbf{w} \cdot \mathbf{x} = 0$ in augmented form. It is important to understand that the only value that impacts the shape of the hyperplane is $\mathbf{w}$. To convince you, we can come back to the two-dimensional case when a hyperplane is just a line. When we create the augmented three-dimensional vectors, we obtain $\mathbf{x} = (x_0, x_1, x_2)$ and $\mathbf{w} = (b, a, -1)$. You can see that the vector $\mathbf{w}$ contains both a and b, which are the two main components defining the look of the line. Changing the value of $\mathbf{w}$ gives us different hyperplanes (lines), as shown in Figure 14.

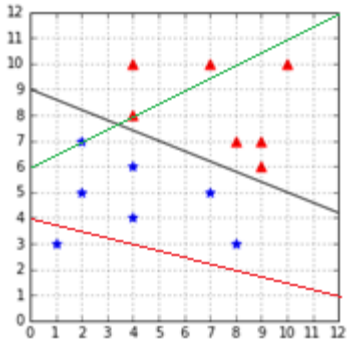

*Figure 14: Different values of **w** will give you different hyperplanes*

Summary

After introducing vectors and linear separability, we learned what a hyperplane is and how we can use it to classify data. We then saw that the goal of a learning algorithm trying to learn a linear classifier is to find a hyperplane separating the data. Eventually, we discovered that finding a hyperplane is equivalent to finding a vector $\mathbf{w}$.

We will now examine which approaches learning algorithms use to find a hyperplane that separates the data. Before looking at how SVMs do this, we will first look at one of the simplest learning models: the Perceptron.

Chapter 2 The Perceptron

Presentation

The Perceptron is an algorithm invented in 1957 by Frank Rosenblatt, a few years before the first SVM. It is widely known because it is the building block of a simple neural network: the multilayer perceptron. The goal of the Perceptron is to find a hyperplane that can separate a linearly separable data set. Once the hyperplane is found, it is used to perform binary classification.

Given augmented vectors $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ and $\mathbf{w} = (w_0, w_1, ..., w_n)$, the Perceptron uses the same hypothesis function we saw in the previous chapter to classify a data point $\mathbf{x}_i$:

$$h(\mathbf{x}_i) = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i)$$

The Perceptron learning algorithm

Given a training set $\mathcal{D}$ of m n-dimensional training examples $(\mathbf{x}_i, y_i)$, the **Perceptron Learning Algorithm** (PLA) tries to find a hypothesis function h that predicts the label y_i of every $\mathbf{x}_i$ correctly.

The hypothesis function of the Perceptron is $h(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x})$, and we saw that $\mathbf{w} \cdot \mathbf{x}$ is just the equation of a hyperplane. We can then say that the set $\mathcal{H}$ of hypothesis functions is the set of $n - 1$ dimensional hyperplanes ($n - 1$ because a hyperplane has one dimension less than its ambient space).

What is important to understand here is that the only unknown value is $\mathbf{w}$. It means that the goal of the algorithm is to find a value for $\mathbf{w}$. You find $\mathbf{w}$; you have a hyperplane. There is an infinite number of hyperplanes (you can give any value to $\mathbf{w}$), so there is an infinity of hypothesis functions.

This can be written more formally this way:

Given a training set: $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{-1, 1\}\}_{i=1}^m$ and a set $\mathcal{H}$ of hypothesis functions.

Find $h \in \mathcal{H}$ such that $h(\mathbf{x}_i) = y_i$ for every $\mathbf{x}_i$.

This is equivalent to:

Given a training set: $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{-1, 1\}\}_{i=1}^m$ and a set $\mathcal{H}$ of hypothesis functions.

Find $\mathbf{w} = (w_0, w_1, ..., w_n)$ such that $\text{sign}(\mathbf{w} \cdot \mathbf{x}_i) = y_i$ for every $\mathbf{x}_i$.

The PLA is a very simple algorithm, and can be summarized this way:

1. Start with a random hyperplane (defined by a vector $\mathbf{w}$) and use it to classify the data.
2. Pick a misclassified example and select another hyperplane by updating the value of $\mathbf{w}$, hoping it will work better at classifying this example (this is called the **update rule**).
3. Classify the data with this new hyperplane.
4. Repeat steps 2 and 3 until there is no misclassified example.

Once the process is over, you have a hyperplane that separates the data.
The algorithm is shown in Code Listing 11.

Code Listing 11

```
import numpy as np

def perceptron_learning_algorithm(X, y):
    w = np.random.rand(3)   # can also be initialized at zero.
    misclassified_examples = predict(hypothesis, X, y, w)

    while misclassified_examples.any():
        x, expected_y = pick_one_from(misclassified_examples, X, y)
        w = w + x * expected_y  # update rule
        misclassified_examples = predict(hypothesis, X, y, w)

    return w
```

Let us look at the code in detail.

The **perceptron_learning_algorithm** uses several functions (Code Listing 12). The **hypothesis** function is just $h(\mathbf{x})$ written in Python code; as we saw before, it is the function that returns the label y_i predicted for an example $\mathbf{x}_i$ when classifying with the hyperplane defined by $\mathbf{w}$. The **predict** function applies the hypothesis for every example and returns the ones that are misclassified.

Code Listing 12

```
def hypothesis(x, w):
    return np.sign(np.dot(w, x))


# Make predictions on all data points
# and return the ones that are misclassified.
def predict(hypothesis_function, X, y, w):
    predictions = np.apply_along_axis(hypothesis_function, 1, X, w)
    misclassified = X[y != predictions]
    return misclassified
```

Once we have made predictions with **predict**, we know which examples are misclassified, so we use the function **pick_one_from** to select one of them randomly (Code Listing 13).

Code Listing 13

```
# Pick one misclassified example randomly
# and return it with its expected label.
def pick_one_from(misclassified_examples, X, y):
    np.random.shuffle(misclassified_examples)
    x = misclassified_examples[0]
    index = np.where(np.all(X == x, axis=1))
    return x, y[index]
```

We then arrive at the heart of the algorithm: the update rule. For now, just remember that it changes the value of $\mathbf{w}$. Why it does this will be explained in detail later. We once again use the **predict** function, but this time, we give it the updated $\mathbf{w}$. It allows us to see if we have classified all data points correctly, or if we need to repeat the process until we do.

Code Listing 14 demonstrates how we can use the **perceptron_learning_algorithm** function with a toy data set. Note that we need the $\mathbf{w}$ and $\mathbf{x}$ vectors to have the same dimension, so we convert every $\mathbf{x}$ vector into an augmented vector before giving it to the function.

Code Listing 14

```
# See Appendix A for more information about the dataset
from succinctly.datasets import get_dataset, linearly_separable as ls

np.random.seed(88)


X, y = get_dataset(ls.get_training_examples)

# transform X into an array of augmented vectors.
X_augmented = np.c_[np.ones(X.shape[0]), X]

w = perceptron_learning_algorithm(X_augmented, y)

print(w) # [-44.35244895   1.50714969   5.52834138]
```

Understanding the update rule

Why do we use this particular update rule? Recall that we picked a misclassified example at random. Now we would like to make the Perceptron correctly classify this example. To do so, we decide to update the vector $\mathbf{w}$. The idea here is simple. Since the sign of the dot product between $\mathbf{w}$ and $\mathbf{x}$ is incorrect, by changing the angle between them, we can make it correct:

- If the predicted label is 1, the angle between $\mathbf{w}$ and $\mathbf{x}$ is smaller than $90°$, and we want to increase it.
- If the predicted label is -1, the angle between $\mathbf{w}$ and $\mathbf{x}$ is bigger than $90°$, and we want to decrease it.

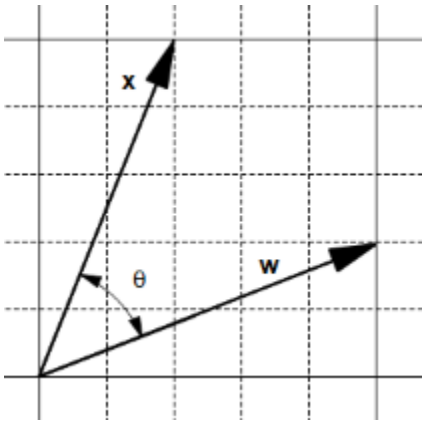

Figure 15: Two vectors

Let's see what happens with two vectors, $\mathbf{w}$ and $\mathbf{x}$, having an angle θ between (Figure 15). On the one hand, adding them creates a new vector $\mathbf{w} + \mathbf{x}$ and the angle β between $\mathbf{x}$ and $\mathbf{w} + \mathbf{x}$ is smaller than θ (Figure 16).

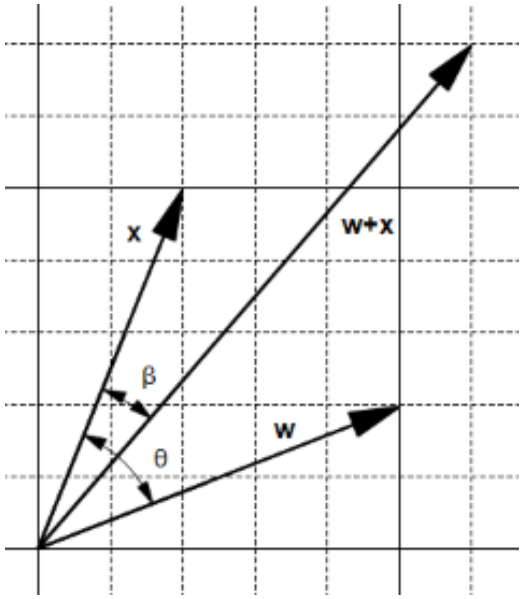

Figure 16: The addition creates a smaller angle

On the other hand, subtracting them creates a new vector $\mathbf{w} - \mathbf{x}$, and the angle β between $\mathbf{x}$ and $\mathbf{w} - \mathbf{x}$ is bigger than θ (Figure 17).

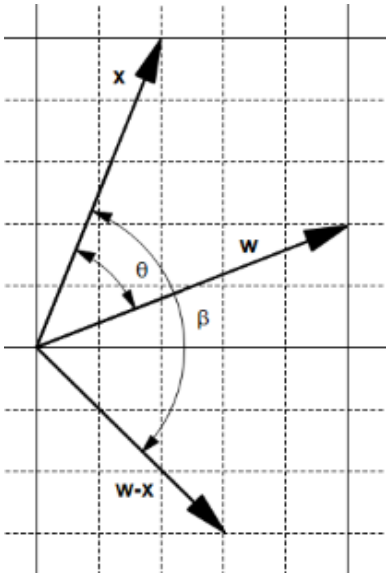

Figure 17: The subtraction creates a bigger angle

We can use these two observations to adjust the angle:

- If the **predicted label** is 1, the angle is smaller than $90°$. We want to increase the angle, so we set $\mathbf{w} = \mathbf{w} - \mathbf{x}$.
- If the **predicted label** is -1, the angle is bigger than $90°$. We want to decrease the angle, so we set $\mathbf{w} = \mathbf{w} + \mathbf{x}$.

As we are doing this only on misclassified examples, when the predicted label has a value, the expected label is the opposite. This means we can rewrite the previous statement:

- If the **expected label** is -1: We want to increase the angle, so we set $\mathbf{w} = \mathbf{w} - \mathbf{x}$.
- If the **expected label** is +1: We want to decrease the angle, so we set $\mathbf{w} = \mathbf{w} + \mathbf{x}$.

When translated into Python it gives us Code Listing 15, and we can see that it is strictly equivalent to Code Listing 16, which is the update rule.

Code Listing 15

```
def update_rule(expected_y, w, x):
    if expected_y == 1:
        w = w + x
    else:
        w = w - x
    return w
```

Code Listing 16

```
def update_rule(expected_y, w, x):
    w = w + x * expected_y
    return w
```

We can verify that the update rule works as we expect by checking the value of the hypothesis before and after applying it (Code Listing 17).

Code Listing 17

```
import numpy as np

def hypothesis(x, w):
    return np.sign(np.dot(w, x))

x = np.array([1, 2, 7])
expected_y = -1
w = np.array([4, 5, 3])

print(hypothesis(w, x))             # The predicted y is 1.

w = update_rule(expected_y, w, x)   # we apply the update rule.

print(hypothesis(w, x))             # The predicted y is -1.
```

Note that the update rule does not necessarily change the sign of the hypothesis for the example the first time. Sometimes it is necessary to apply the update rule several times before it happens as shown in Code Listing 18. This is not a problem, as we are looping across misclassified examples, so we will continue to use the update rule until the example is correctly classified. What matters here is that each time we use the update rule, we change the value of the angle in the right direction (increasing it or decreasing it).

Code Listing 18

```
import numpy as np

x = np.array([1,3])
expected_y = -1
w = np.array([5, 3])

print(hypothesis(w, x))            # The predicted y is 1.

w = update_rule(expected_y, w, x)  # we apply the update rule.

print(hypothesis(w, x))            # The predicted y is 1.

w = update_rule(expected_y, w, x)  # we apply the update rule once again.
```

```
print(hypothesis(w, x))            # The predicted y is -1.
```

Also note that sometimes updating the value of $\mathbf{w}$ for a particular example $\mathbf{x}$ changes the hyperplane in such a way that another example $\mathbf{x}^*$ previously correctly classified becomes misclassified. So, the hypothesis might become worse at classifying after being updated. This is illustrated in Figure 18, which shows us the number of classified examples at each iteration step. One way to avoid this problem is to keep a record of the value of $\mathbf{w}$ before making the update and use the updated $\mathbf{w}$ only if it reduces the number of misclassified examples. This modification of the PLA is known as the **Pocket algorithm** (because we keep $\mathbf{w}$ in our pocket).

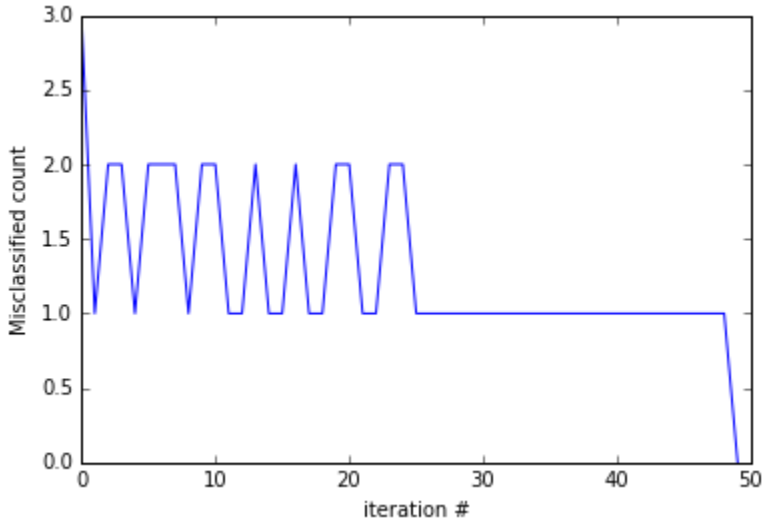

Figure 18: The PLA update rule oscillates

Convergence of the algorithm

We said that we keep updating the vector $\mathbf{w}$ with the update rule until there is no misclassified point. But how can we be so sure that will ever happen? Luckily for us, mathematicians have studied this problem, and we can be very sure because the Perceptron convergence theorem *guarantees* that if the two sets P and N (of positive and negative examples respectively) are linearly separable, the vector $\mathbf{w}$ is updated only a finite number of times, which was first proved by Novikoff in 1963 (Rojas, 1996).

Understanding the limitations of the PLA

One thing to understand about the PLA algorithm is that because weights are randomly initialized and misclassified examples are randomly chosen, it is possible the algorithm will return a different hyperplane each time we run it. Figure 19 shows the result of running the PLA on the same dataset four times. As you can see, the PLA finds four different hyperplanes.

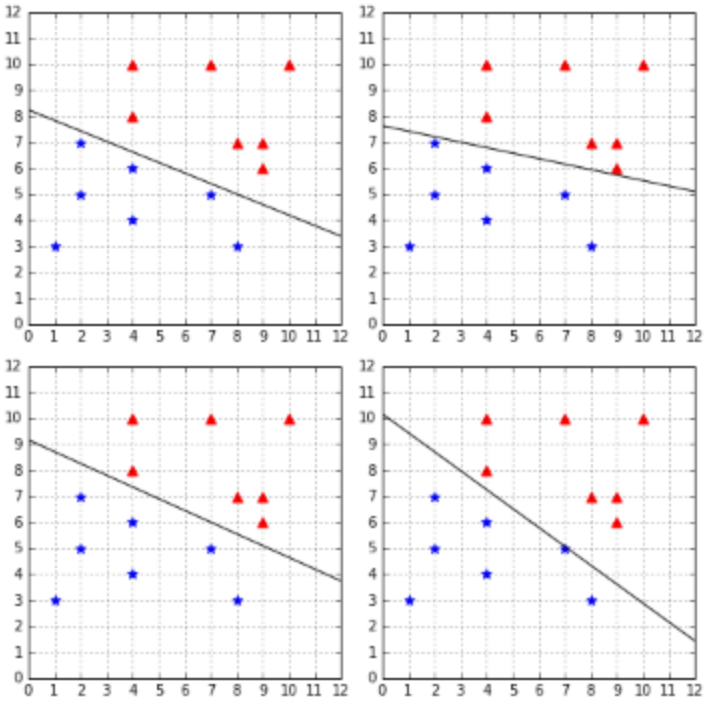

Figure 19: The PLA finds a different hyperplane each time

At first, this might not seem like a problem. After all, the four hyperplanes perfectly classify the data, so they might be equally good, right? However, when using a machine learning algorithm such as the PLA, our goal is not to find a way to classify perfectly the data we have right now. Our goal is to find a way to correctly classify new data we will receive in the future.

Let us introduce some terminology to be clear about this. To train a model, we pick a **sample** of existing data and call it the **training set**. We train the model, and it comes up with a **hypothesis** (a hyperplane in our case). We can measure how well the hypothesis performs on the training set: we call this the **in-sample error** (also called training error). Once we are satisfied with the hypothesis, we decide to use it on unseen data (the **test set**) to see if it indeed learned something. We measure how well the hypothesis performs on the test set, and we call this the **out-of-sample error** (also called the generalization error).

Our goal is to have the smallest out-of-sample error.

In the case of the PLA, all hypotheses in Figure 19 perfectly classify the data: their in-sample error is zero. But we are really concerned about their out-of-sample error. We can use a test set such as the one in Figure 20 to check their out-of-sample errors.

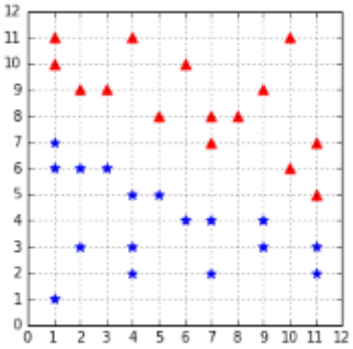

Figure 20: A test dataset

As you can see in Figure 21, the two hypotheses on the right, despite perfectly classifying the training dataset, are making errors with the test dataset.

Now we better understand why it is problematic. When using the Perceptron with a linearly separable dataset, we have the guarantee of finding a hypothesis with zero in-sample error, but we have no guarantee about how well it will **generalize** to unseen data (if an algorithm generalizes well, its out-of-sample error will be close to its in-sample error). How can we choose a hyperplane that generalizes well? As we will see in the next chapter, this is one of the goals of SVMs.

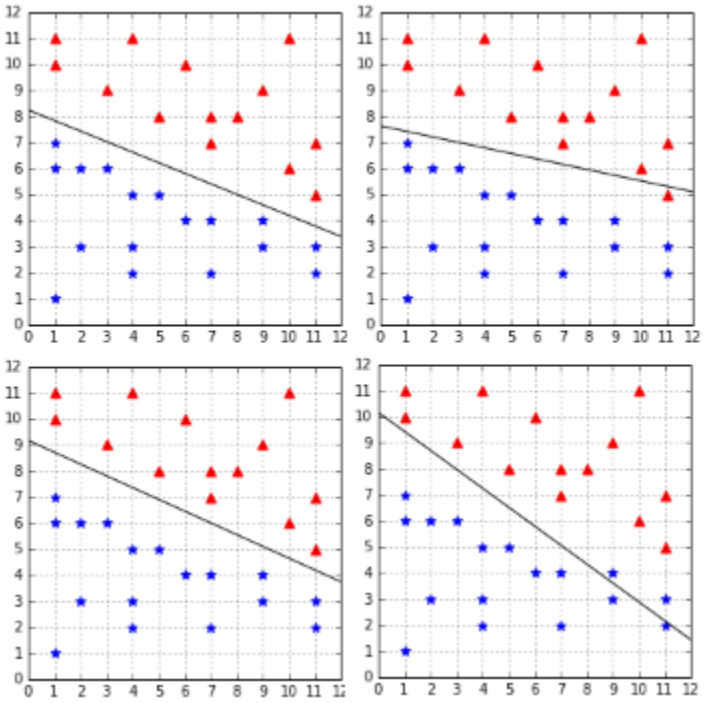

Figure 21: Not all hypotheses have perfect out-of-sample error

Summary

In this chapter, we have learned what a Perceptron is. We then saw in detail how the Perceptron Learning Algorithm works and what the motivation behind the update rule is. After learning that the PLA is guaranteed to converge, we saw that not all hypotheses are equal, and that some of them will generalize better than others. Eventually, we saw that the Perceptron is unable to select which hypothesis will have the smallest out-of-sample error and instead just picks one hypothesis having the lowest in-sample error at random.

Chapter 3 The SVM Optimization Problem

SVMs search for the optimal hyperplane

The Perceptron has several advantages: it is a simple model, the algorithm is very easy to implement, and we have a theoretical proof that it will find a hyperplane that separates the data. However, its biggest weakness is that it will not find the same hyperplane every time. Why do we care? Because not all separating hyperplanes are equals. If the Perceptron gives you a hyperplane that is very close to all the data points from one class, you have a right to believe that it will generalize poorly when given new data.

SVMs do not have this problem. Indeed, instead of looking for *a* hyperplane, SVMs tries to find *the* hyperplane. We will call this the **optimal hyperplane**, and we will say that it is the one that best separates the data.

How can we compare two hyperplanes?

Because we cannot choose the optimal hyperplane based on our feelings, we need some sort of metric that will allow us to compare two hyperplanes and say which one is superior to all others.

In this section, we will try to discover how we can compare two hyperplanes. In other words, we will search for a way to compute a number that allows us to tell which hyperplane separates the data the best. We will look at methods that seem to work, but then we will see why they do not work and how we can correct their limitations. Let us try with a simple attempt to compare two hyperplanes using only the equation of the hyperplane.

Using the equation of the hyperplane

Given an example $(\mathbf{x}, y)$ and a hyperplane, we wish to know how the example relates to the hyperplane.

One key element we already know is that if the value of $\mathbf{x}$ satisfies the equation of a line, then it means it is on the line. It works in the same way for a hyperplane: for a data point $\mathbf{x}$ and a hyperplane defined by a vector $\mathbf{w}$ and bias b, we will get $\mathbf{w} \cdot \mathbf{x} + b = 0$ if $\mathbf{x}$ is on the hyperplane.

But what if the point is not on the hyperplane?

Let us see what happens with an example. In Figure 22, the line is defined by $\mathbf{w} = (-0.4, -1)$ and $b = 9$. When we use the equation of the hyperplane:

- for point $A(1, 3)$, using vector $\mathbf{a} = (1, 3)$ we get $\mathbf{w} \cdot \mathbf{a} + b = 5.6$
- for point $B(3, 5)$, using vector $\mathbf{b} = (3, 5)$ we get $\mathbf{w} \cdot \mathbf{b} + b = 2.8$
- for point $C(5, 7)$, using vector $\mathbf{c} = (5, 7)$ we get $\mathbf{w} \cdot \mathbf{c} + b = 0$

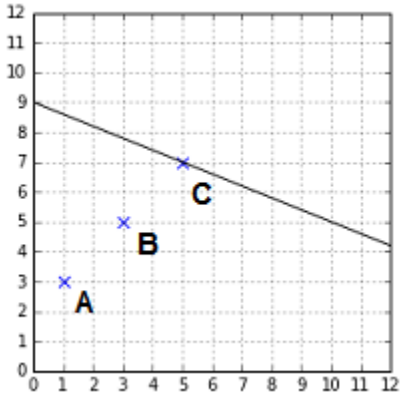

Figure 22: The equation returns a bigger number for A than for B

As you can see, when the point is not on the hyperplane we get a number different from zero. In fact, if we use a point far away from the hyperplane, we will get a bigger number than if we use a point closer to the hyperplane.

Another thing to notice is that the sign of the number returned by the equation tells us where the point stands with respect to the line. Using the equation of the line displayed in Figure 23, we get:

- 2.8 for point $A(3, 5)$
- 0 for point $B(5, 7)$
- -2.8 for point $C(7, 9)$

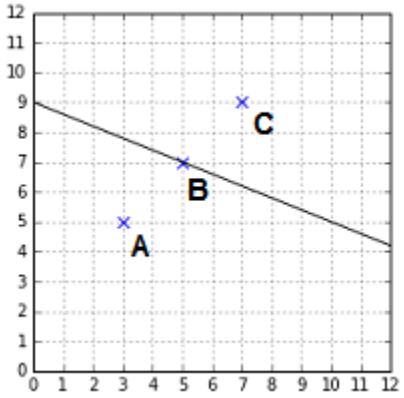

Figure 23: The equation returns a negative number for C

If the equation returns a positive number, the point is below the line, while if it is a negative number, it is above. Note that it is not necessarily visually above or below, because if you have a line like the one in Figure 24, it will be left or right, but the same logic applies. The sign of the number returned by the equation of the hyperplane allows us to tell if two points lie on the same side. In fact, this is exactly what the hypothesis function we defined in Chapter 2 does.

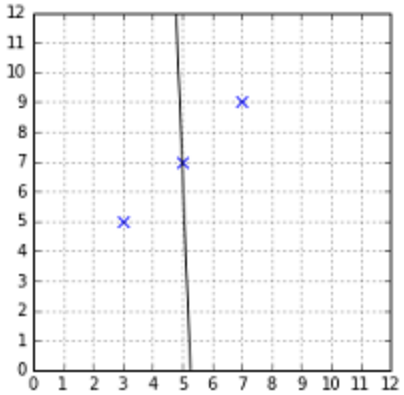

Figure 24: A line can separate the space in different ways

We now have the beginning of a solution for comparing two hyperplanes.

Given a training example $(\mathbf{x}, y)$ and a hyperplane defined by a vector $\mathbf{w}$ and bias b, we compute the number $\beta = \mathbf{w} \cdot \mathbf{x} + b$ to know how far the point is from the hyperplane.

Given a data set $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^n,\ y_i \in \{-1, 1\}\}_{i=1}^{m}$, we compute β for each training example, and say that the number B is the smallest β we encounter.

$$B = \min_{i=1...m} \beta_i$$

If we need to choose between two hyperplanes, we will then select the one from which B is the largest.

To be clear, this means that if we have k hyperplanes, we will compute $\max_{i=1...k} B_i$ and select the hyperplane having this B_i.

Problem with examples on the negative side

Unfortunately, using the result of the hyperplane equation has its limitations. The problem is that taking the minimum value does not work for examples on the negative side (the ones for which the equation returns a negative value).

Remember that we always wish to take the β of the point being the closest to the hyperplane. Computing B with examples on the positive side actually does this. Between two points with $\beta = +5$ and $\beta = +1$, we pick the one having the smallest number, so we choose $+1$. However, between two examples having $\beta = -5$ and $\beta = -1$, this rule will pick -5 because -5 is smaller than -1, but the closest point is actually the one with $\beta = -1$.

One way to fix this problem is to consider the absolute value of β.

Given a data set $\mathcal{D}$, we compute β for each example and say that B is the β having the smallest absolute value:

$$B = \min_{i=1...m} |\beta_i|$$

Does the hyperplane correctly classify the data?

Computing the number B allows us to select a hyperplane. However, using only this value, we might pick the wrong one. Consider the case in Figure 25: the examples are **correctly classified**, and the value of B computed using the last formula is 2.

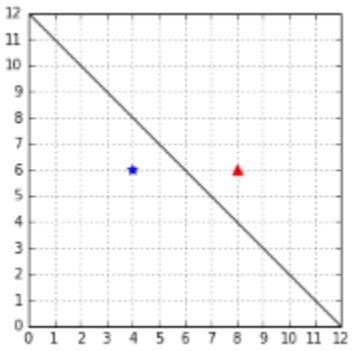

Figure 25: A hyperplane correctly classifying the data

In Figure 26, the examples are **incorrectly classified,** and the value of B is also 2. This is problematic because using B, we do not know which hyperplane is better. In theory, they look equally good, but in reality, we want to pick the one from Figure 25.

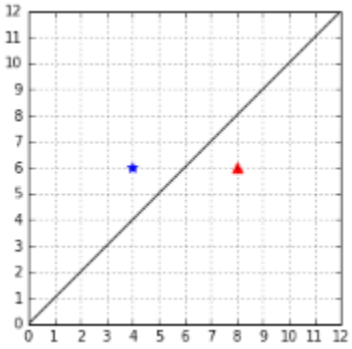

Figure 26: A hyperplane that does not classify the data correctly

How can we adjust our formula to meet this requirement?

Well, there is one component of our training example $(\mathbf{x}, y)$ that we did not use: the y !

If we multiply β by the value of y, we change its sign. Let us call this new number f:

$$f = y \times \beta$$

$$f = y(\mathbf{w} \cdot \mathbf{x} + b)$$

The sign of f will always be:

- Positive if the point is correctly classified
- Negative if the point is incorrectly classified

Given a data set $\mathcal{D}$, we can compute:

$$F = \min_{i=1...m} f_i$$

$$F = \min_{i=1...m} y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$$

With this formula, when comparing two hyperplanes, we will still select the one for which F is the largest. The added bonus is that in special cases like the ones in Figure 25 and Figure 26, we will always pick the hyperplane that classifies correctly (because F will have a positive value, while its value will be negative for the other hyperplane).

In the literature, the number f has a name, it is called the **functional margin** of an example; its value can be computed in Python, as shown in Code Listing 19. Similarly, the number F is known as the **functional margin of the data set** $\mathcal{D}$.

Code Listing 19

```
# Compute the functional margin of an example (x,y)
# with respect to a hyperplane defined by w and b.
def example_functional_margin(w, b, x, y):
    result = y * (np.dot(w, x) + b)
    return result

# Compute the functional margin of a hyperplane
# for examples X with labels y.
def functional_margin(w, b, X, y):
    return np.min([example_functional_margin(w, b, x, y[i])
                   for i, x in enumerate(X)])
```

Using this formula, we find that the functional margin of the hyperplane in Figure 25 is +2, while in Figure 26 it is -2. Because it has a bigger margin, we will select the first one.

Tip: Remember, we wish to choose the hyperplane with the largest margin.

Scale invariance

It looks like we found a good way to compare the two hyperplanes this time. However, there is a major problem with the functional margin: is not scale invariant.

Given a vector $\mathbf{w}_1 = (2, 1)$ and bias $b_1 = 5$, if we multiply them by 10, we get $\mathbf{w}_2 = (20, 10)$ and $b_2 = 50$. We say we **rescaled** them.

The vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, represent the same hyperplane because they have the same unit vector. The hyperplane being a plane orthogonal to a vector $\mathbf{w}$, it does not matter how long the vector is. The only thing that matters is its direction, which, as we saw in the first chapter, is given by its unit vector. Moreover, when tracing the hyperplane on a graph, the coordinate of the intersection between the vertical axis and the hyperplane will be $(0, b/w_1)$, so the hyperplane does not change because of the rescaling of b, either.

The problem, as we can see in Code Listing 20, is that when we compute the functional margin with $\mathbf{w}_2$, we get a number ten times bigger than with $\mathbf{w}_1$. This means that given any hyperplane, we can always find one that will have a larger functional margin, just by rescaling $\mathbf{w}$ and b.

Code Listing 20

```
x = np.array([1, 1])
y = 1

b_1 = 5
w_1 = np.array([2, 1])

w_2 = w_1 * 10
b_2 = b_1 * 10

print(example_functional_margin(w_1, b_1, x, y))  # 8
print(example_functional_margin(w_2, b_2, x, y))  # 80
```

To solve this problem, we only need to make a small adjustment. Instead of using the vector $\mathbf{w}$, we will use its unit vector. To do so, we will divide $\mathbf{w}$ by its norm. In the same way, we will divide b by the norm of $\mathbf{w}$ to make it scale invariant as well.

Recall the formula of the functional margin: $f = y(\mathbf{w} \cdot \mathbf{x}) + b$

We modify it and obtain a new number γ:

$$\gamma = y\left(\frac{\mathbf{w}}{||\mathbf{w}||} \cdot \mathbf{x} + \frac{b}{||\mathbf{w}||}\right)$$

As before, given a data set $\mathcal{D}$, we can compute:

$$M = \min_{i=1...m} \gamma_i$$

$$M = \min_{i=1...m} y_i\left(\frac{\mathbf{w}}{||\mathbf{w}||} \cdot \mathbf{x} + \frac{b}{||\mathbf{w}||}\right)$$

The advantage of γ is that it gives us the same number no matter how large is the vector $\mathbf{w}$ that we choose. The number γ also has a name—it is called the **geometric margin** of a training example, while M is the geometric margin of the dataset. A Python implementation is shown in Code Listing 21.

Code Listing 21

```
# Compute the geometric margin of an example (x,y)
# with respect to a hyperplane defined by w and b.
def example_geometric_margin(w, b, x, y):
    norm = np.linalg.norm(w)
    result = y * (np.dot(w/norm, x) + b/norm)
    return result

# Compute the geometric margin of a hyperplane
# for examples X with labels y.
def geometric_margin(w, b, X, y):
    return np.min([example_geometric_margin(w, b, x, y[i])
                   for i, x in enumerate(X)])
```

We can verify that the geometric margin behaves as expected. In Code Listing 22, the function returns the same value for the vector $\mathbf{w}_1$ or its rescaled version $\mathbf{w}_2$.

Code Listing 22

```
x = np.array([1,1])
y = 1

b_1 = 5
w_1 = np.array([2,1])

w_2 = w_1*10
b_2 = b_1*10

print(example_geometric_margin(w_1, b_1, x, y))  # 3.577708764
print(example_geometric_margin(w_2, b_2, x, y))  # 3.577708764
```

It is called the geometric margin because we can retrieve this formula using simple geometry. It measures the distance between $\mathbf{x}$ and the hyperplane.

In Figure 27, we see that the point X' is the orthogonal projection of X into the hyperplane. We wish to find the distance d between X and X'.

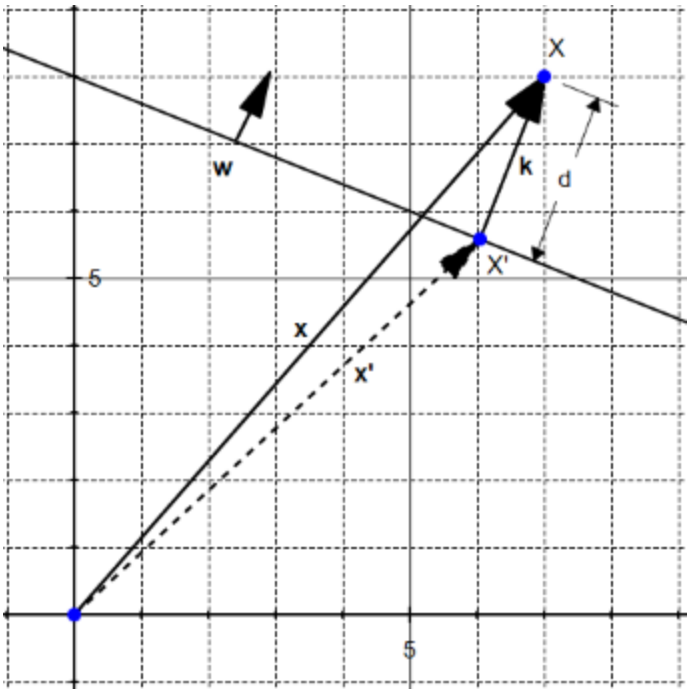

Figure 27: The geometric margin is the distance d between the point X and the hyperplane

The vector $\mathbf{k}$ has the same direction as the vector $\mathbf{w}$, so they share the same unit vector $\frac{\mathbf{w}}{||w||}$. We know that the norm of $\mathbf{k}$ is d, so the vector $\mathbf{k}$ can be defined by $\mathbf{k} = d\frac{\mathbf{w}}{||w||}$.

Moreover, we can see that $\mathbf{x}' = \mathbf{x} - \mathbf{k}$, so if we substitute for $\mathbf{k}$ and do a little bit of algebra, we get:

$$\mathbf{x}' = \mathbf{x} - d\frac{\mathbf{w}}{||w||}$$

Now, the point X' is on the hyperplane. It means that $\mathbf{x}'$ satisfies the equation of the hyperplane, and we have:

$$\mathbf{w} \cdot \mathbf{x}' + b = 0$$

$$\mathbf{w} \cdot \left(\mathbf{x} - d\frac{\mathbf{w}}{||w||}\right) + b = 0$$

$$\mathbf{w} \cdot \mathbf{x} - d\frac{\mathbf{w} \cdot \mathbf{w}}{||w||} + b = 0$$

$$\mathbf{w} \cdot \mathbf{x} - d\frac{||w||^2}{||w||} + b = 0$$

$$\mathbf{w} \cdot \mathbf{x} - d||w|| + b = 0$$

$$d = \frac{\mathbf{w} \cdot \mathbf{x} + b}{||w||}$$

$$d = \frac{\mathbf{w}}{||w||} \cdot \mathbf{x} + \frac{b}{||w||}$$

Eventually, as we did before, we multiply by y to ensure that we select a hyperplane that correctly classifies the data, and it gives us the geometric margin formula we saw earlier:

$$\gamma = y\left(\frac{\mathbf{w}}{||\mathbf{w}||} \cdot \mathbf{x} + \frac{b}{||\mathbf{w}||}\right)$$

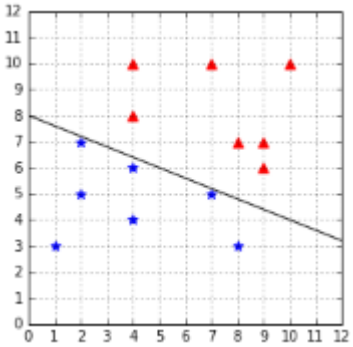

Figure 28: A hyperplane defined by w=(-0.4,-1) and b=8

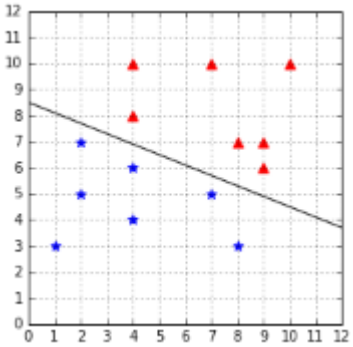

Figure 29: A hyperplane defined by w=(-0.4,-1) and b=8.5

Now that we have defined the geometric margin, let us see how it allows us to compare two hyperplanes. We can see that the hyperplane in Figure 28 is closer to the blue star examples than to the red triangle examples as compared to the one in Figure 29. As a result, we expect its geometric margin to be smaller. Code Listing 23 uses the function defined in Code Listing 21 to compute the geometric margin for each hyperplane. As expected from Figure 29, the geometric margin of the second hyperplane defined by $\mathbf{w} = (-0.4, -1)$ and $b = 8.5$ is larger (0.64 > 0.18). Between the two, we would select this hyperplane.

Code Listing 23

```
# Compare two hyperplanes using the geometrical margin.

positive_x = [[2,7],[8,3],[7,5],[4,4],[4,6],[1,3],[2,5]]
negative_x = [[8,7],[4,10],[9,7],[7,10],[9,6],[4,8],[10,10]]

X = np.vstack((positive_x, negative_x))
y = np.hstack((np.ones(len(positive_x)), -1*np.ones(len(negative_x))))

w = np.array([-0.4, -1])
b = 8

# change the value of b
print(geometric_margin(w, b, X, y))          # 0.185695338177
print(geometric_margin(w, 8.5, X, y))        # 0.64993368362
```

We see that to compute the geometric margin for another hyperplane, we just need to modify the value of $\mathbf{ww}$ or b. We could try to change it by a small increment to see if the margin gets larger, but it is kind of random, and it would take a lot of time. Our objective is to find the optimal hyperplane for a dataset **among all possible hyperplanes**, and there is an infinity of hyperplanes.

Tip: Finding the optimal hyperplane is just a matter of finding the values of w and b for which we get the largest geometric margin.

How can we find the value of $\mathbf{w}$ that produces the largest geometric margin? Luckily for us, mathematicians have designed tools to solve such problems. To find $\mathbf{w}$ and b, we need to solve what is called an **optimization problem**. Before looking at what the optimization problem is for SVMs, let us do a quick review of what an optimization problem is.

What is an optimization problem?

Unconstrained optimization problem

The goal of an **optimization problem** is to minimize or maximize a function with respect to some variable x (that is, to find the value of x for with the function returns its minimum or maximum value). For instance, the problem in which we want to find the minimum of the function $f(x) = x^2$ is written:

$$\underset{x}{\text{minimize}} \quad f(x)$$

Or, alternatively:

$$\min_{x} \quad f(x)$$

In this case, we are free to search amongst all possible values of x. We say that the problem is unconstrained. As we can see in Figure 30, the minimum of the function is zero at $x = 0$.

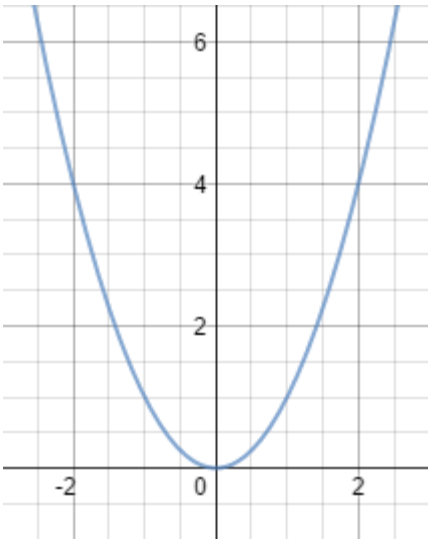

Figure 30: Without constraint, the minimum is zero

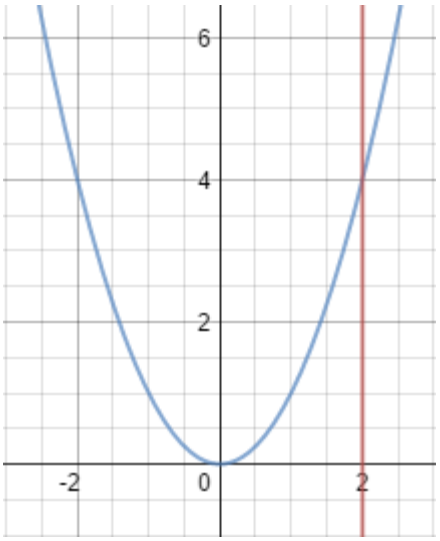

Figure 31: Because of the constraint x-2=0, the minimum is 4

Constrained optimization problem

Single equality constraint

Sometimes we are not interested in the minimum of the function by itself, but rather its minimum when some constraints are met. In such cases, we write the problem and add the constraints preceded by subject to, which is often abbreviated s.t. For instance, if we wish to know the minimum of f but restrict the value of x to a specific value, we can write:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x = 2 \end{aligned}$$

This example is illustrated in Figure 31. In general, constraints are written by keeping zero on the right side of the equality so the problem can be rewritten:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x - 2 = 0 \end{aligned}$$

Using this notation, we clearly see that the constraint is an **affine function** while the objective function f is a **quadratic function**. Thus we call this problem a **quadratic optimization problem** or a Quadratic Programming (QP) problem.

Feasible set

The set of variables that satisfies the problem constraints is called the **feasible set** (or feasible region). When solving the optimization problem, the solution will be picked from the feasible set. In Figure 31, the feasible set contains only one value, so the problem is trivial. However, when we manipulate functions with several variables, such as $f(x, y) = x^2 + y^2$, it allows us to know from which values we are trying to pick a minimum (or maximum).

For example:

$$\begin{aligned} \underset{x,y}{\text{minimize}} \quad & f(x, y) \\ \text{subject to} \quad & x - 2 = 0 \end{aligned}$$

In this problem, the feasible set is the set of all pairs of points (x, y), such as $(x, y) = (2, y)$.

Multiple equality constraints and vector notation

We can add as many constraints as we want. Here is an example of a problem with three constraints for the function $f(x, y, z) = x^2 + y - z^2$:

$$\begin{aligned} \underset{x,y,z}{\text{minimize}} \quad & f(x, y, z) \\ \text{subject to} \quad & x - 2 = 0 \\ & y + 8 = 0 \\ & z + 3 = 0 \end{aligned}$$

When we have several variables, we can switch to vector notation to improve readability. For the vector $\mathbf{x} = (x, y, z)^T$ the function becomes $f(\mathbf{x}) = \mathbf{x}_1^2 - \mathbf{x}_2 + \mathbf{x}_3^2$, and the problem is written:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x}_1 - 2 = 0 \\ & \mathbf{x}_2 + 8 = 0 \end{aligned}$$

When adding constraints, keep in mind that doing so reduces the feasible set. For a solution to be accepted, **all constraints must be satisfied**.

For instance, let us look at the following the problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x^2 \\ \text{subject to} \quad & x - 2 = 0 \\ & x - 8 = 0 \end{aligned}$$

We could think that $x = 2$ and $x = 8$ are solutions, but this is not the case. When $x = 2$, the constraint $x - 8 = 0$ is not met; and when $x = 8$, the constraint $x - 2 = 0$ is not met. The problem is **infeasible**.

Tip: If you add too many constraints to a problem, it can become infeasible.

If you change an optimization problem by adding a constraint, you make the optimum worse, or, at best, you leave it unchanged (Gershwin, 2010).

Inequality constraints

We can also use inequalities as constraints:

$$\begin{aligned} \underset{x,y}{\text{minimize}} \quad & x^2 + y^2 \\ \text{subject to} \quad & x - 2 \geq 0 \\ & y \geq 0 \end{aligned}$$

And we can combine equality constraints and inequality constraints:

$$\begin{aligned} \underset{x,y}{\text{minimize}} \quad & x^2 + y^2 \\ \text{subject to} \quad & x - 2 = 0 \\ & y \geq 0 \end{aligned}$$

How do we solve an optimization problem?

Several methods exist that can solve each type of optimization problem. However, presenting them is outside the scope of this book. The interested reader can see *Optimization Models and Application* (El Ghaoui, 2015) and *Convex Optimization* (Boyd & Vandenberghe, 2004), two good books for starting on the subject and that are available online for free (see Bibliography for details). We will instead focus on the SVMs again and derive an optimization problem allowing us to find the optimal hyperplane. How to solve the SVMs optimization problem will be explained in detail in the next chapter.

The SVMs optimization problem

Given a linearly separable training set $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^p,\ y_i \in \{-1, 1\}\}_{i=1}^m$ and a hyperplane with a normal vector $\mathbf{w}$ and bias b, recall that the **geometric margin** M of the hyperplane is defined by**:**

$$M = \min_{i=1...m} \gamma_i$$

where $\gamma_i = \left(y_i \frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \mathbf{x}_i + \frac{b}{\|\mathbf{w}\|} \right)$ is the geometric margin of a training example $(\mathbf{x}_i, y_i)$.

The **optimal separating hyperplane** is the hyperplane defined by the normal vector $\mathbf{w}$ and bias b for which the geometric margin M is the largest.

To find $\mathbf{w}$ and b, we need to solve the following optimization problem, with the constraint that the margin of each example should be greater or equal to M:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{maximize}} \quad & M \\ \text{subject to} \quad & \gamma_i \geq M, \ i = 1, \ldots, m \end{aligned}$$

There is a relationship between the geometric margin and the functional margin:

$$M = \frac{F}{||\mathbf{w}||}$$

So we can rewrite the problem:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{maximize}} \quad & M \\ \text{subject to} \quad & \frac{f_i}{||\mathbf{w}||} \geq \frac{F}{||\mathbf{w}||}, \ \ i = 1, \ldots, m \end{aligned}$$

We can then simplify the constraint by removing the norm on both sides of the inequality:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{maximize}} \quad & M \\ \text{subject to} \quad & f_i \geq F, \ \ i = 1, \ldots, m \end{aligned}$$

Recall that we are trying to maximize the geometric margin and that the scale of $\mathbf{w}$ and b does not matter. We can choose to rescale $\mathbf{w}$ and b as we want, and the geometric margin will not change. As a result, we decide to scale $\mathbf{w}$ and b so that $F = 1$. It will not affect the result of the optimization problem.

The problem becomes:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{maximize}} \quad & M \\ \text{subject to} \quad & f_i \geq 1, \ \ i = 1, \ldots, m \end{aligned}$$

Because $M = \frac{F}{||\mathbf{w}||}$ it is the same as:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{maximize}} \quad & \frac{F}{||\mathbf{w}||} \\ \text{subject to} \quad & f_i \geq 1, \ \ i = 1, \ldots, m \end{aligned}$$

And because we decided to set $F = 1$, this is equivalent to:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{maximize}} \quad & \frac{1}{||\mathbf{w}||} \\ \text{subject to} \quad & f_i \geq 1, \ \ i = 1, \ldots, m \end{aligned}$$

This maximization problem is equivalent to the following minimization problem:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{minimize}} \quad & ||\mathbf{w}|| \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i) + b \geq 1, \ \ i = 1, \ldots, m \end{aligned}$$

Tip: You can also read an alternate derivation of this optimization problem on this page, where I use geometry instead of the functional and geometric margins.

This minimization problem gives the same result as the following:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i)+b \geq 1, \quad i=1,\ldots,m \end{aligned}$$

The factor $\frac{1}{2}$ has been added for later convenience, when we will use QP solver to solve the problem and squaring the norm has the advantage of removing the square root.

Eventually, here is the optimization problem as you will see it written in most of the literature:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i)+b-1 \geq 0, \quad i=1,\ldots,m \end{aligned}$$

Why did we take the pain of rewriting the problem like this? Because the original optimization problem was difficult to solve. Instead, we now have **convex quadratic optimization problem**, which, although not obvious, is much simpler to solve.

Summary

First, we assumed that some hyperplanes are better than others: they will perform better with unseen data. Among all possible hyperplanes, we decided to call the “best” hyperplane the **optimal hyperplane**. To find the optimal hyperplane, we searched for a way to compare two hyperplanes, and we ended up with a number allowing us to do so. We realized that this number also has a geometrical meaning and is called the **geometric margin**.

We then stated that the optimal hyperplane is the one with the largest geometric margin and that we can find it by **maximizing the margin**. To make things easier, we noted that we could minimize the norm of $\mathbf{w}$, the vector normal to the hyperplane, and we will be sure that it will be the $\mathbf{w}$ of the optimal hyperplane (because if you recall, $\mathbf{w}$ is used in the formula for computing the geometric margin).

Chapter 4 Solving the Optimization Problem

Lagrange multipliers

The Italian-French mathematician **Giuseppe Lodovico Lagrangia**, also known as **Joseph-Louis Lagrange**, invented a strategy for finding the local maxima and minima of a function subject to equality constraint. It is called the method of Lagrange multipliers.

The method of Lagrange multipliers

Lagrange noticed that when we try to solve an optimization problem of the form:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g(\mathbf{x}) = 0 \end{aligned}$$

the minimum of f is found when its gradient point in the same direction as the gradient of g. In other words, when:

$$\nabla f(\mathbf{x}) = \alpha \nabla g(\mathbf{x})$$

So if we want to find the minimum of f under the constraint g, we just need to solve for:

$$\nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x}) = 0$$

Here, the constant α is called a Lagrange multiplier.

To simplify the method, we observe that if we define a function $\mathcal{L}(\mathbf{x}, \alpha) = f(\mathbf{x}) - \alpha g(\mathbf{x})$, then its gradient is $\nabla \mathcal{L}(\mathbf{x}, \alpha) = \nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x})$. As a result, solving for $\nabla \mathcal{L}(\mathbf{x}, \alpha) = 0$ allows us to find the minimum.

The Lagrange multiplier method can be summarized by these three steps:

1. Construct the Lagrangian function $\mathcal{L}$ by introducing one multiplier per constraint
2. Get the gradient $\nabla \mathcal{L}$ of the Lagrangian
3. Solve for $\nabla \mathcal{L}(\mathbf{x}, \alpha) = 0$

The SVM Lagrangian problem

We saw in the last chapter that the SVM optimization problem is:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0, \; i = 1, \ldots, m \end{aligned}$$

Let us return to this problem. We have one objective function to minimize:

$$f(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2$$

and m constraint functions:

$$g_i(\mathbf{w}, b) = y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1, \; i = 1, \ldots, m$$

We introduce the Lagrangian function:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = f(\mathbf{w}) - \sum_{i=1}^{m} \alpha_i g_i(\mathbf{w}, b)$$

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{m} \alpha_i \left[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \right]$$

Note that we introduced one **Lagrange multiplier** α_i for each constraint function.

We could try to solve for $\mathcal{L}(\mathbf{w}, b, \alpha) = 0$, but the problem can only be solved analytically when the number of examples is small (Tyson Smith, 2004). So we will once again rewrite the problem using the duality principle.

To get the solution of the primal problem, we need to solve the following **Lagrangian problem**:

$$\begin{aligned} &\min_{\mathbf{w},b} \quad \max_{\alpha} \quad \mathcal{L}(\mathbf{w}, b, \alpha) \\ &\text{subject to} \quad \alpha_i \geq 0, \;\; i = 1, \ldots, m \end{aligned}$$

What is interesting here is that we need to minimize with respect to $\mathbf{w}$ and b, and to maximize with respect to α at the same time.

Tip: You may have noticed that the method of Lagrange multipliers is used for solving problems with equality constraints, and here we are using them with inequality constraints. This is because the method still works for inequality constraints, provided some additional conditions (the KKT conditions) are met. We will talk about these conditions later.

The Wolfe dual problem

The Lagrangian problem has m inequality constraints (where m is the number of training examples) and is typically solved using its dual form. The **duality principle** tells us that an optimization problem can be viewed from two perspectives. The first one is the primal problem, a minimization problem in our case, and the other one is the dual problem, which will be a maximization problem. What is interesting is that the maximum of the dual problem will always be less than or equal to the minimum of the primal problem (we say it provides a lower bound to the solution of the primal problem).

In our case, we are trying to solve a convex optimization problem, and **Slater's condition** holds for affine constraints (Gretton, 2016), so **Slater's theorem** tells us that **strong duality** holds. This means that the maximum of the dual problem is equal to the minimum of the primal problem. Solving the dual is the same thing as solving the primal, except it is easier.

Recall that the Lagrangian function is:

$$\begin{aligned}\mathcal{L}(\mathbf{w}, b, \alpha) &= \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{m} \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] \\ &= \frac{1}{2}\mathbf{w} \cdot \mathbf{w} - \sum_{i=1}^{m} \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1]\end{aligned}$$

The Lagrangian primal problem is:

$$\begin{aligned}&\min_{\mathbf{w},b} \quad \max_{\alpha} \quad \mathcal{L}(\mathbf{w}, b, \alpha) \\ &\text{subject to} \quad \alpha_i \geq 0, \quad i = 1, \ldots, m\end{aligned}$$

Solving the minimization problem involves taking the partial derivatives of $\mathcal{L}$ with respect to $\mathbf{w}$ and b.

$$\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i = \mathbf{0}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{m} \alpha_i y_i = 0$$

From the first equation, we find that:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$$

Let us substitute $\mathbf{w}$ by this value into $\mathcal{L}$:

$$\begin{aligned}W(\alpha, b) &= \frac{1}{2}\Big(\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i\Big) \cdot \Big(\sum_{j=1}^{m} \alpha_j y_j \mathbf{x}_j\Big) - \sum_{i=1}^{m} \alpha_i \Big[y_i\Big((\sum_{j=1}^{m} \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_i + b\Big) - 1\Big] \\ &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_{i=1}^{m} \alpha_i y_i \Big((\sum_{j=1}^{m} \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_i + b\Big) + \sum_{i=1}^{m} \alpha_i \\ &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - b\sum_{i=1}^{m} \alpha_i y_i + \sum_{i=1}^{m} \alpha_i \\ &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - b\sum_{i=1}^{m} \alpha_i y_i\end{aligned}$$

So we successfully removed $\mathbf{w}$, but b is still used in the last term of the function:

$$W(\alpha, b) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - b \sum_{i=1}^{m} \alpha_i y_i$$

We note that $\frac{\partial \mathcal{L}}{\partial b} = 0$ implies that $\sum_{i=1}^{m} \alpha_i y_i = 0$. As a result, the last term is equal to zero, and we can write:

$$W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

This is the **Wolfe dual Lagrangian function**.

The optimization problem is now called the **Wolfe dual problem**:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ \text{subject to} \quad & \alpha_i \geq 0, \text{ for any } i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

Traditionally the Wolfe dual Lagrangian problem is constrained by the gradients being equal to zero. In theory, we should add the constraints $\nabla_{\mathbf{w}} \mathcal{L} = 0$ and $\frac{\partial \mathcal{L}}{\partial b} = 0$. However, we only added the latter. Indeed, we added $\sum_{i=1}^{m} \alpha_i y_i = 0$ because it is necessary for removing b from the function. However, we can solve the problem without the constraint $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$.

The main advantage of the Wolfe dual problem over the Lagrangian problem is that the objective function W now depends only on the Lagrange multipliers. Moreover, this formulation will help us solve the problem in Python in the next section and will be very helpful when we define kernels later.

Karush-Kuhn-Tucker conditions

Because we are dealing with inequality constraints, there is an additional requirement: the solution must also satisfy the Karush-Kuhn-Tucker (KKT) conditions.

The KKT conditions are first-order **necessary** conditions for a solution of an optimization problem to be optimal. Moreover, the problem should satisfy some regularity conditions. Luckily for us, one of the regularity conditions is **Slater's condition**, and we just saw that it holds for SVMs. Because the primal problem we are trying to solve is a convex problem, the KKT conditions are also **sufficient** for the point to be primal and dual optimal, and there is zero duality gap.

To sum up, if a solution satisfies the KKT conditions, we are guaranteed that it is the optimal solution.

The Karush-Kuhn-Tucker conditions are:

- Stationarity condition:

$$\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i = \mathbf{0}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{m} \alpha_i y_i = 0$$

- Primal feasibility condition:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0 \qquad \text{for all } i = 1, \ldots, m$$

- Dual feasibility condition:

$$\alpha_i \geq 0 \qquad \text{for all } i = 1, \ldots, m$$

- Complementary slackness condition:

$$\alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] = 0 \qquad \text{for all } i = 1, \ldots, m$$

Note: "[...]Solving the SVM problem is equivalent to finding a solution to the KKT conditions." (Burges, 1988)

Note that we already saw most of these conditions before. Let us examine them one by one.

Stationarity condition

The stationarity condition tells us that the selected point must be a **stationary point**. It is a point where the function stops increasing or decreasing. When there is no constraint, the stationarity condition is just the point where the gradient of the objective function is zero. When we have constraints, we use the gradient of the Lagrangian.

Primal feasibility condition

Looking at this condition, you should recognize the constraints of the primal problem. It makes sense that they must be enforced to find the minimum of the function under constraints.

Dual feasibility condition

Similarly, this condition represents the constraints that must be respected for the dual problem.

Complementary slackness condition

From the complementary slackness condition, we see that either $\alpha_i = 0$ or $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0$.

Support vectors are examples having a positive Lagrange multiplier. They are the ones for which the constraint $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0$ is **active**. (We say the constraint is active when $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0$).

Tip: From the complementary slackness condition, we see that support vectors are examples that have a positive Lagrange multiplier.

What to do once we have the multipliers?

When we solve the Wolfe dual problem, we get a vector α containing all Lagrange multipliers. However, when we first stated the primal problem, our goal was to find $\mathbf{w}$ and b. Let us see how we can retrieve these values from the Lagrange multipliers.

Compute w

Computing $\mathbf{w}$ is pretty simple since we derived the formula: $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$ from the gradient $\nabla_{\mathbf{w}} \mathcal{L}$.

Compute b

Once we have $\mathbf{w}$, we can use one of the constraints of the primal problem to compute b:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0$$

Indeed, this constraint is still true because we transformed the original problem in such a way that the new formulations are equivalent. What it says is that the closest points to the hyperplane will have a functional margin of 1 (the value 1 is the value we chose when we decided how to scale $\mathbf{w}$):

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$$

From there, as we know all other variables, it is easy to come up with the value of b. We multiply both sides of the equation by y_i , and because $y_i^2 = 1$, it gives us:

$$\mathbf{w} \cdot \mathbf{x}_i + b = y_i$$

$$b = y_i - \mathbf{w} \cdot \mathbf{x}_i$$

However, as indicated in *Pattern Recognition and Machine Learning* (Bishop, 2006), instead of taking a random support vector $\mathbf{x}_i$, taking the average provides us with a numerically more stable solution:

$$b = \frac{1}{S}\sum_{i=1}^{S}(y_i - \mathbf{w}\cdot\mathbf{x}_i)$$

where S is the number of support vectors.

Other authors, such as (Cristianini & Shawe-Taylor, 2000) and (Ng), use another formula:

$$b = -\frac{max_{y_i=-1}(\mathbf{w}\cdot\mathbf{x}_i) + min_{y_i=1}(\mathbf{w}\cdot\mathbf{x}_i)}{2}$$

They basically take the average of the nearest positive support vector and the nearest negative support vector. This latest formula is the one originally used by *Statistical Learning Theory* (Vapnik V. N., 1998) when defining the optimal hyperplane.

Hypothesis function

The SVMs use the same hypothesis function as the Perceptron. The class of an example $\mathbf{x}_i$ is given by:

$$h(\mathbf{x}_i) = \text{sign}(\mathbf{w}\cdot\mathbf{x}_i + b)$$

When using the dual formulation, it is computed using only the support vectors:

$$h(\mathbf{x}_i) = \text{sign}\left(\sum_{j=1}^{S}\alpha_j y_j(\mathbf{x}_j\cdot\mathbf{x}_i) + b\right)$$

Solving SVMs with a QP solver

A QP solver is a program used to solve quadratic programming problems. In the following example, we will use the Python package called CVXOPT.

This package provides a method that is able to solve quadratic problems of the form:

$$\begin{array}{ll}\underset{x}{\text{minimize}} & \frac{1}{2}x^T Px + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b\end{array}$$

It does not look like our optimization problem, so we will need to rewrite it so that we can solve it with this package.

First, we note that in the case of the Wolfe dual optimization problem, what we are trying to minimize is α, so we can rewrite the quadratic problem with α instead of x to better see how the two problems relate:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}\alpha^T P\alpha + q^T\alpha \\ \text{subject to} \quad & G\alpha \preceq h \\ & A\alpha = b \end{aligned}$$

Here the $\preceq$ symbol represents component-wise vector inequalities. It means that each row of the matrix $G\lambda$ represents an inequality that must be satisfied.

We will change the Wolfe dual problem. First, we transform the maximization problem:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{m}\alpha_i \\ \text{subject to} \quad & \alpha_i \geq 0, \text{ for any } i = 1, \ldots, m \\ & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

into a minimization problem by multiplying by -1.

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_{i=1}^{m}\alpha_i \\ \text{subject to} \quad & -\alpha_i \leq 0, \text{ for any } i = 1, \ldots, m \\ & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

Then we introduce vectors $\alpha = (\alpha_1, \ldots, \alpha_m)^T$ and $\mathbf{y} = (y_1, \cdots, y_m)^T$ and the Gram matrix K of all possible dot products of vectors $\mathbf{x}_i$:

$$K(\mathbf{x}_1, \ldots, \mathbf{x}_m) = \begin{pmatrix} \mathbf{x}_1 \cdot \mathbf{x}_1 & \mathbf{x}_1 \cdot \mathbf{x}_2 & \ldots & \mathbf{x}_1 \cdot \mathbf{x}_m \\ \mathbf{x}_2 \cdot \mathbf{x}_1 & \mathbf{x}_2 \cdot \mathbf{x}_2 & \ldots & \mathbf{x}_2 \cdot \mathbf{x}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_m \cdot \mathbf{x}_1 & \mathbf{x}_m \cdot \mathbf{x}_2 & \ldots & \mathbf{x}_m \cdot \mathbf{x}_m \end{pmatrix}$$

We use them to construct a vectorized version of the Wolfe dual problem where $\mathbf{y}\mathbf{y}^T$ denotes the outer product of $\mathbf{y}$.

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2}\alpha^T(\mathbf{y}\mathbf{y}^T K)\alpha - \alpha \\ \text{subject to} \quad & -\alpha \preceq 0, \\ & \mathbf{y} \cdot \alpha = 0 \end{aligned}$$

We are now able to find out the value for each of the parameters P, q, G, h, A, and b required by the CVXOPT **qp** function. This is demonstrated in Code Listing 24.

Code Listing 24

```
# See Appendix A for more information about the dataset
from succinctly.datasets import get_dataset, linearly_separable as ls
```

```
import cvxopt.solvers


X, y = get_dataset(ls.get_training_examples)
m = X.shape[0]

# Gram matrix - The matrix of all possible inner products of X.
K = np.array([np.dot(X[i], X[j])
              for j in range(m)
              for i in range(m)]).reshape((m, m))

P = cvxopt.matrix(np.outer(y, y) * K)
q = cvxopt.matrix(-1 * np.ones(m))

# Equality constraints
A = cvxopt.matrix(y, (1, m))
b = cvxopt.matrix(0.0)

# Inequality constraints
G = cvxopt.matrix(np.diag(-1 * np.ones(m)))
h = cvxopt.matrix(np.zeros(m))

# Solve the problem
solution = cvxopt.solvers.qp(P, q, G, h, A, b)

# Lagrange multipliers
multipliers = np.ravel(solution['x'])

# Support vectors have positive multipliers.
has_positive_multiplier = multipliers > 1e-7
sv_multipliers = multipliers[has_positive_multiplier]

support_vectors = X[has_positive_multiplier]
support_vectors_y = y[has_positive_multiplier]
```

Code Listing 24 initializes all the required parameters and passes them to the **qp** function, which returns us a solution. The solution contains many elements, but we are only concerned about the **x**, which, in our case, corresponds to the Lagrange multipliers.

As we saw before, we can re-compute $\mathbf{w}$ using all the Lagrange multipliers: $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$. Code Listing 25 shows the code of the function that computes $\mathbf{w}$.

Code Listing 25

```
def compute_w(multipliers, X, y):
    return np.sum(multipliers[i] * y[i] * X[i]
                  for i in range(len(y)))
```

Because Lagrange multipliers for non-support vectors are almost zero, we can also compute **w** using only support vectors data and their multipliers, as illustrated in Code Listing 26.

Code Listing 26

```
w = compute_w(multipliers, X, y)
w_from_sv = compute_w(sv_multipliers, support_vectors, support_vectors_y)

print(w)          # [0.44444446 1.11111114]
print(w_from_sv)  # [0.44444453 1.11111128]
```

And we compute b using the average method:

Code Listing 27

```
def compute_b(w, X, y):
    return np.sum([y[i] - np.dot(w, X[i])
                   for i in range(len(X))])/len(X)
```

Code Listing 28

```
b = compute_b(w, support_vectors, support_vectors_y) # -9.666668268506335
```

When we plot the result in Figure 32, we see that the hyperplane is the optimal hyperplane. Contrary to the Perceptron, the SVM will always return the same result.

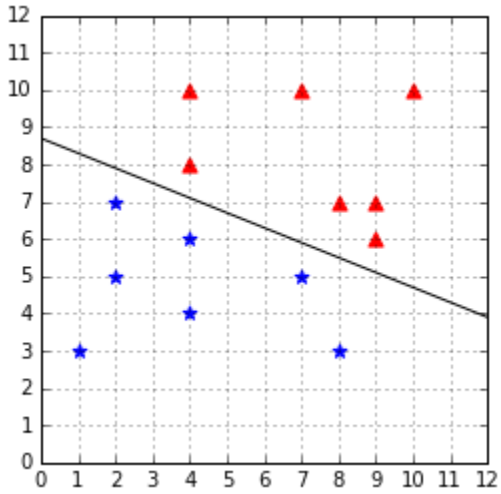

Figure 32: The hyperplane found with CVXOPT

This formulation of the SVM is called the **hard margin SVM**. It cannot work when the data is not linearly separable. There are several Support Vector Machines formulations. In the next chapter, we will consider another formulation called the **soft margin SVM**, which will be able to work when data is non-linearly separable because of outliers.

Summary

Minimizing the norm of $\mathbf{w}$ is a **convex optimization problem**, which can be solved using the Lagrange multipliers method. When there are more than a few examples, we prefer using convex optimization packages, which will do all the hard work for us.

We saw that the original optimization problem can be rewritten using a Lagrangian function. Then, thanks to duality theory, we transformed the Lagrangian problem into the Wolfe dual problem. We eventually used the package CVXOPT to solve the Wolfe dual.

Chapter 5 Soft Margin SVM

Dealing with noisy data

The biggest issue with **hard margin SVM** is that it requires the data to be linearly separable. Real-life data is often noisy. Even when the data is linearly separable, a lot of things can happen before you feed it to your model. Maybe someone mistyped a value for an example, or maybe the probe of a sensor returned a crazy value. In the presence of an outlier (a data point that seems to be out of its group), there are two cases: the outlier can be closer to the other examples than most of the examples of its class, thus reducing the margin, or it can be among the other examples and break linear separability. Let us consider these two cases and see how the hard margin SVM deals with them.

Outlier reducing the margin

When the data is linearly separable, the hard margin classifier does not behave as we would like in the presence of outliers.

Let us now consider our dataset with the addition of an outlier data point at (5, 7), as shown in Figure 33.

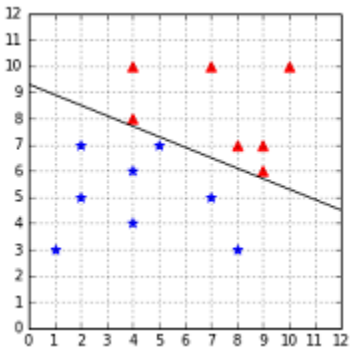

Figure 33: The dataset is still linearly separable with the outlier at (5, 7)

In this case, we can see that the margin is very narrow, and it seems that the outlier is the main reason for this change. Intuitively, we can see that this hyperplane might not be the best at separating the data, and that it will probably generalize poorly.

Outlier breaking linear separability

Even worse, when the outlier breaks the linear separability, as the point (7, 8) does in Figure 34, the classifier is incapable of finding a hyperplane. We are stuck because of a single data point.

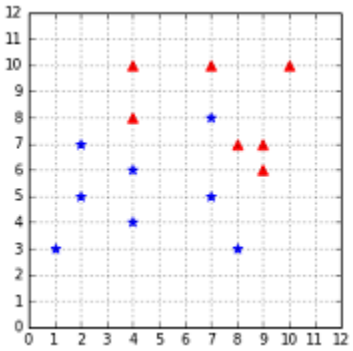

Figure 34: The outlier at (7, 8) breaks linear separability

Soft margin to the rescue

Slack variables

In 1995, Vapnik and Cortes introduced a modified version of the original SVM that allows the classifier to make some mistakes. The goal is now not to make zero classification mistakes, but to make as few mistakes as possible.

To do so, they modified the constraints of the optimization problem by adding a variable ζ (zeta). So the constraint:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$$

becomes:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \zeta_i$$

As a result, when minimizing the objective function, it is possible to satisfy the constraint even if the example does not meet the original constraint (that is, it is too close from the hyperplane, or it is not on the correct side of the hyperplane). This is illustrated in Code Listing 29.

Code Listing 29

```
import numpy as np

w = np.array([0.4, 1])
b = -10

x = np.array([6, 8])
y = -1


def constraint(w, b, x, y):
    return y * (np.dot(w, x) + b)
```

```
def hard_constraint_is_satisfied(w, b, x, y):
    return constraint(w, b, x, y) >= 1


def soft_constraint_is_satisfied(w, b, x, y, zeta):
    return constraint(w, b, x, y) >= 1 - zeta


# While the constraint is not satisfied for the example (6,8).
print(hard_constraint_is_satisfied(w, b, x, y))               # False

# We can use zeta = 2 and satisfy the soft constraint.
print(soft_constraint_is_satisfied(w, b, x, y, zeta=2))       # True
```

The problem is that we could choose a huge value of ζ for every example, and all the constraints will be satisfied.

Code Listing 30

```
# We can pick a huge zeta for every point
# to always satisfy the soft constraint.
print(soft_constraint_is_satisfied(w, b, x, y, zeta=10))   # True
print(soft_constraint_is_satisfied(w, b, x, y, zeta=1000)) # True
```

To avoid this, we need to modify the objective function to penalize the choice of a big ζ_i:

$$\begin{aligned} \underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{m} \zeta_i \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \zeta_i \quad \text{for any } i = 1, \ldots, m \end{aligned}$$

We take the sum of all individual ζ_i and add it to the objective function. Adding such a penalty is called **regularization**. As a result, the solution will be the hyperplane that maximizes the margin while having the smallest error possible.

There is still a little problem. With this formulation, one can easily minimize the function by using negative values of ζ_i. We add the constraint $\zeta_i \geq 0$ to prevent this. Moreover, we would like to keep some control over the soft margin. Maybe sometimes we want to use the hard margin—after all, that is why we add the parameter C, which will help us to determine how important the ζ should be (more on that later).

This leads us to the **soft margin formulation**:

$$\begin{aligned} \underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} \zeta_i \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \zeta_i \\ & \zeta_i \geq 0 \quad \text{for any } i = 1, \ldots, m \end{aligned}$$

As shown by (Vapnik V. N., 1998), using the same technique as for the separable case, we find that we need to maximize **the same Wolfe dual as before, under a slightly different constraint**:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \text{ for any } i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

Here the constraint $\alpha_i \geq 0$ has been changed to become $0 \leq \alpha_i \leq C$. This constraint is often called the **box constraint** because the vector α is constrained to lie inside the box with side length C in the positive orthant. Note that an orthant is the analog n-dimensional Euclidean space of a quadrant in the plane (Cristianini & Shawe-Taylor, 2000). We will visualize the box constraint in Figure 50 in the chapter about the SMO algorithm.

The optimization problem is also called **1-norm soft margin** because we are minimizing the 1-norm of the slack vector ζ.

Understanding what C does

The parameter C gives you control of how the SVM will handle errors. Let us now examine how changing its value will give different hyperplanes.

Figure 35 shows the linearly separable dataset we used throughout this book. On the left, we can see that setting C to $+\inf$ gives us the same result as the hard margin classifier. However, if we choose a smaller value for C like we did in the center, we can see that the hyperplane is closer to some points than others. The hard margin constraint is violated for these examples. Setting $C = 0.01$ increases this behavior as depicted on the right.

What happens if we choose a C very close to zero? Then there is basically no constraint anymore, and we end up with a hyperplane not classifying anything.

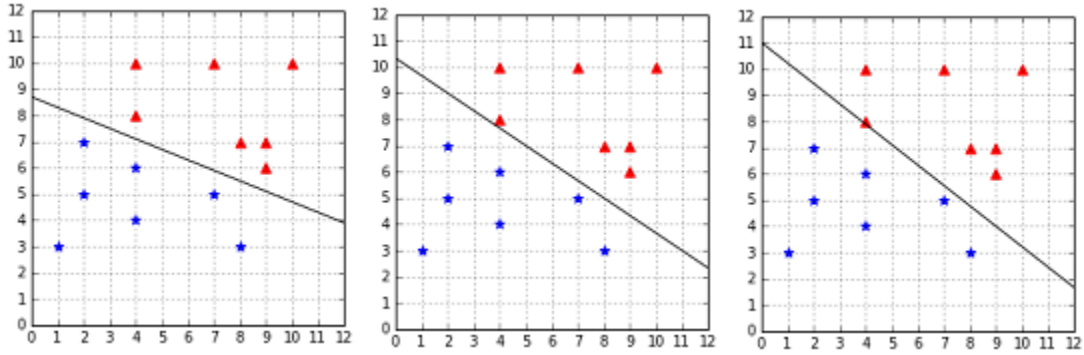

Figure 35: Effect of C=+Infinity, C=1, and C=0.01 on a linearly separable dataset

It seems that when the data is linearly separable, sticking with a big C is the best choice. But what if we have some noisy outlier? In this case, as we can see in Figure 36, using $C = +\infty$ gives us a very narrow margin. However, when we use $C = 1$, we end up with a hyperplane very close to the one of the hard margin classifier without outlier. The only violated constraint is the constraint of the outlier, and we are much more satisfied with this hyperplane. This time, setting $C = 0.01$ ends up violating the constraint of another example, which was not an outlier. This value of C seems to give too much freedom to our soft margin classifier.

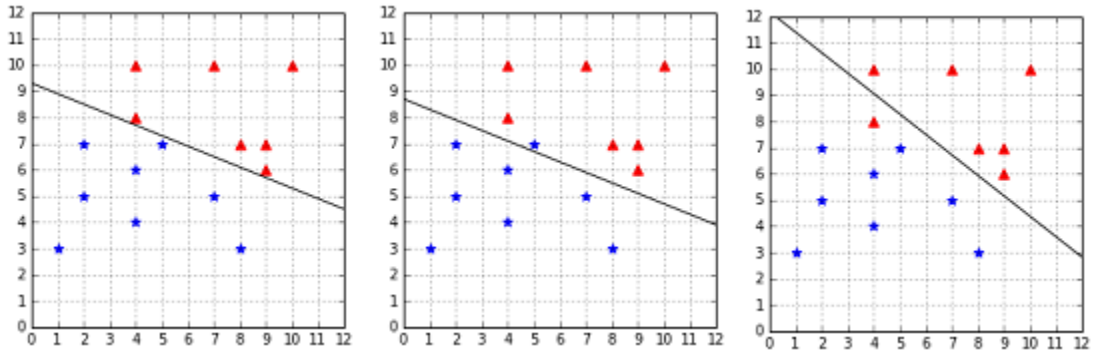

Figure 36: Effect of C=+Infinity, C=1, and C=0.01 on a linearly separable dataset with an outlier

Eventually, in the case where the outlier makes the data non-separable, we cannot use $C = +\infty$ because there is no solution meeting all the hard margin constraints. Instead, we test several values of C, and we see that the best hyperplane is achieved with $C = 3$. In fact, we get the same hyperplane for all values of C greater than or equal to 3. That is because no matter how hard we penalize it, it is necessary to violate the constraint of the outlier to be able to separate the data. When we use a small C, as before, more constraints are violated.

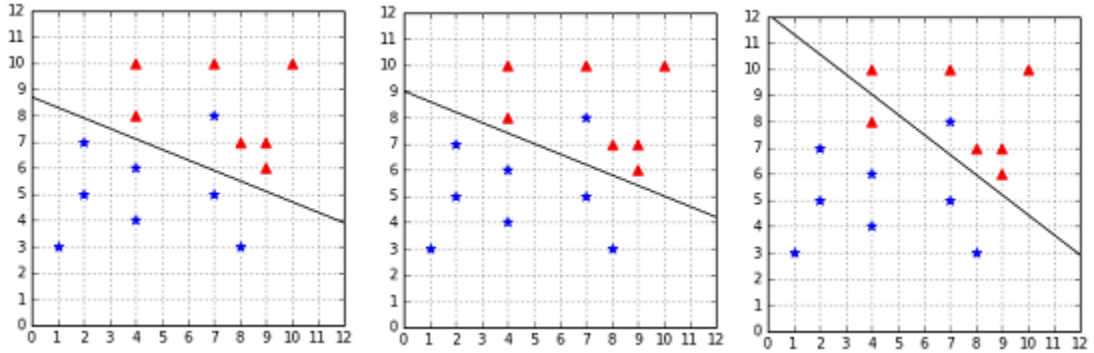

Figure 37: Effect of C=3, C=1, and C=0.01 on a non-separable dataset with an outlier

Rules of thumb:

- A small C will give a wider margin, at the cost of some misclassifications.
- A huge C will give the hard margin classifier and tolerates zero constraint violation.
- The key is to find the value of C such that noisy data does not impact the solution too much.

How to find the best C?

There is no magic value for C that will work for all the problems. The recommended approach to select C is to use grid search with cross-validation (Hsu, Chang, & Lin, *A Practical Guide to Support Vector Classification*). The crucial thing to understand is that the value of C is very specific to the data you are using, so if one day you found that C=0.001 did not work for one of your problems, you should still try this value with another problem, because it will not have the same effect.

Other soft-margin formulations

2-Norm soft margin

There is another formulation of the problem called the **2-norm (or L2 regularized) soft margin** in which we minimize $\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m}\zeta_i^2$. This formulation leads to a Wolfe dual problem without the box constraint. For more information about the 2-norm soft margin, refer to (Cristianini & Shawe-Taylor, 2000).

nu-SVM

Because the scale of C is affected by the feature space, another formulation of the problem has been proposed: the νSVM. The idea is to use a parameter ν whose value is varied between 0 and 1, instead of the parameter C.

> ***Note: "ν gives a more transparent parametrization of the problem, which does not depend on the scaling of the feature space, but only on the noise level in the data." (Cristianini & Shawe-Taylor, 2000)***

The optimization problem to solve is:

$$\begin{aligned}
\underset{\alpha}{\text{maximize}} \quad & -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\
\text{subject to} \quad & 0 \le \alpha_i \le \frac{1}{m}, \\
& \sum_{i=1}^{m}\alpha_i y_i = 0 \\
& \sum_{i=1}^{m}\alpha_i \ge \nu \text{ for any } i = 1, \ldots, m
\end{aligned}$$

Summary

The soft-margin SVM formulation is a nice improvement over the hard-margin classifier. It allows us to classify data correctly even when there is noisy data that breaks linear separability. However, the cost of this added flexibility is that we now have an hyperparameter C, for which we need to find a value. We saw how changing the value of C impacts the margin and allows the classifier to make some mistakes in order to have a bigger margin. This once again reminds us that our goal is to find a hypothesis that will work well on unseen data. A few mistakes on the training data is not a bad thing if the model generalizes well in the end.

Chapter 6 Kernels

Feature transformations

Can we classify non-linearly separable data?

Imagine you have some data that is not separable (like the one in Figure 38), and you would like to use SVMs to classify it. We have seen that it is not possible because the data is not linearly separable. However, this last assumption is not correct. What is important to notice here is that the data is not linearly separable **in two dimensions**.

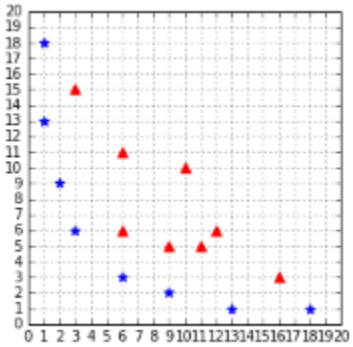

Figure 38: A straight line cannot separate the data

Even if your original data is in two dimensions, nothing prevents you from transforming it before feeding it into the SVM. One possible transformation would be, for instance, to transform every two-dimensional vector (x_1, x_2) into a three-dimensional vector.

For example, we can do what is called a polynomial mapping by applying the function $\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ defined by:

$$\phi(x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

Code Listing 31 shows this transformation implemented in Python.

Code Listing 31

```
# Transform a two-dimensional vector x into a three-dimensional vector.
def transform(x):
    return [x[0]**2, np.sqrt(2)*x[0]*x[1], x[1]**2]
```

If you transform the whole data set of Figure 38 and plot the result, you get Figure 39, which does not show much improvement. However, after some time playing with the graph, we can see that the data is, in fact, separable in three dimensions (Figure 40 and Figure 41)!

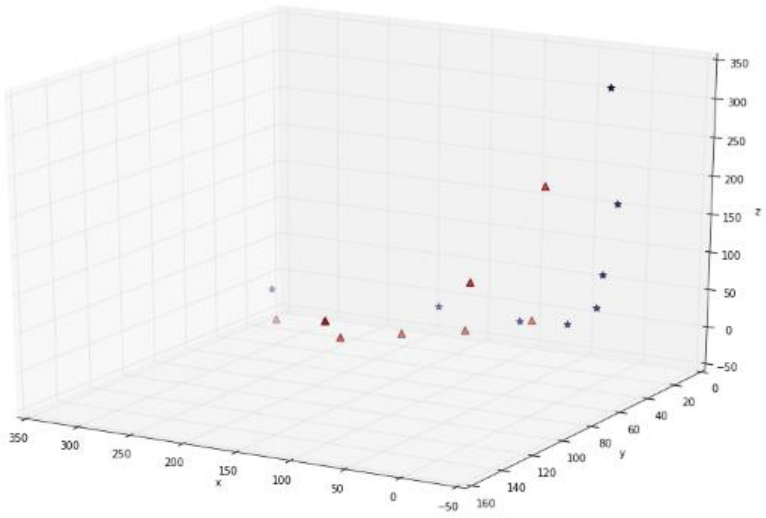

Figure 39: The data does not look separable in three dimensions

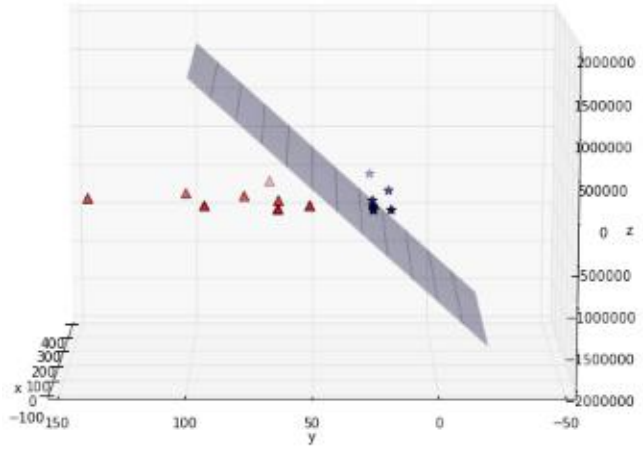

Figure 41: Another view of the data showing the plane from the side

Here is a basic recipe we can use to classify this dataset:

1. Transform every two-dimensional vector into a three-dimensional vector using the transform method of Code Listing 31.
2. Train the SVMs using the 3D dataset.
3. For each new example we wish to predict, transform it using the **`transform`** method before passing it to the **`predict`** method.

Of course, you are not forced to transform the data into three dimensions; it could be five, ten, or one hundred dimensions.

How do we know which transformation to apply?

Choosing which transformation to apply depends a lot on your dataset. Being able to transform the data so that the machine learning algorithm you wish to use performs at its best is probably one key factor of success in the machine learning world. Unfortunately, there is no perfect recipe, and it will come with experience via trial and error. Before using any algorithm, be sure to check if there are some common rules to transform the data detailed in the documentation. For more information about how to prepare your data, you can read the dataset transformation section on the scikit-learn website.

What is a kernel?

In the last section, we saw a quick recipe to use on the non-separable dataset. One of its main drawbacks is that we must transform every example. If we have millions or billions of examples and that transform method is complex, that can take a huge amount of time. This is when kernels come to the rescue.

If you recall, when we search for the KKT multipliers in the Wolfe dual Lagrangian function, we do not need the value of a training example $\mathbf{x}$; we only need the value of the dot product $\mathbf{x}_i \cdot \mathbf{x}_j$ between two training examples:

$$W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

In Code Listing 32, we apply the first step of our recipe. Imagine that when the data is used to learn, the only thing we care about is the value returned by the dot product, in this example 8,100.

Code Listing 32

```
x1 = [3,6]
x2 = [10,10]

x1_3d = transform(x1)
x2_3d = transform(x2)
```

```
print(np.dot(x1_3d,x2_3d))  # 8100
```

The question is this: **Is there a way to compute this value, without transforming the vectors**?

And the answer is: Yes, with a kernel!

Let us consider the function in Code Listing 33:

Code Listing 33

```
def polynomial_kernel(a, b):
    return a[0]**2 * b[0]**2 + 2*a[0]*b[0]*a[1]*b[1] + a[1]**2 * b[1]**2
```

Using this function with the same two examples as before returns the same result (Code Listing 33).

Code Listing 34

```
x1 = [3,6]
x2 = [10,10]

# We do not transform the data.

print(polynomial_kernel(x1, x2)) # 8100
```

When you think about it, this is pretty incredible.

The vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ belong to $\mathbb{R}^2$. The kernel function computes their dot product as if they have been transformed into vectors belonging to $\mathbb{R}^3$, and it does that without doing the transformation, and without computing their dot product!

To sum up: a kernel **is a function** that returns the result of a dot product performed in another space. More formally, we can write:

Definition: *Given a mapping function* $\phi : \mathcal{X} \to \mathcal{V}$*, we call the function* $K : \mathcal{X} \to \mathbb{R}$ *defined by* $K(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle_{\mathcal{V}}$ *, where* $\langle \cdot, \cdot \rangle_{\mathcal{V}}$ *denotes an inner product in* $\mathcal{V}$*, a* ***kernel function****.*

The kernel trick

Now that we know what a kernel is, we will see what the **kernel trick** is.

If we define a kernel as: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$, we can then rewrite the soft-margin dual problem:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \text{ for any } i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

That's it. We have made a single change to the dual problem—we call it the kernel trick.

Tip: Applying the kernel trick simply means replacing the dot product of two examples by a kernel function.

This change looks very simple, but remember that it took a serious amount of work to derive the Wolf dual formulation from the original optimization problem. We now have the power to change the kernel function in order to classify non-separable data.

Of course, we also need to change the hypothesis function to use the kernel function:

$$h(\mathbf{x}_i) = \text{sign}\left(\sum_{j=1}^{S} \alpha_j y_j K(\mathbf{x}_j, \mathbf{x}_i) + b \right)$$

Remember that in this formula S is the set of support vectors. Looking at this formula, we better understand why SVMs are also called **sparse kernel machines**. It is because they only need to compute the kernel function on the support vectors and not on all the vectors, like other kernel methods (Bishop, 2006).

Kernel types

Linear kernel

This is the simplest kernel. It is simply defined by:

$$K(\mathbf{x}, \mathbf{x}') = \mathbf{x} \cdot \mathbf{x}'$$

where $\mathbf{x}$ and $\mathbf{x}'$ are two vectors.

In practice, you should know that a linear kernel works well for text classification.

Polynomial kernel

We already saw the polynomial kernel earlier when we introduced kernels, but this time we will consider the more generic version of the kernel:

$$K(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x}' + c)^d$$

It has two parameters: c, which represents a constant term, and d, which represents the degree of the kernel. This kernel can be implemented easily in Python, as shown in Code Listing 35.

Code Listing 35

```
def polynomial_kernel(a, b, degree, constant=0):
    result = sum([a[i] * b[i] for i in range(len(a))]) + constant
    return pow(result, degree)
```

In Code Listing 36, we see that it returns the same result as the kernel of Code Listing 33 when we use the degree 2. The result of training a SVM with this kernel is shown in Figure 42.

Code Listing 36

```
x1 = [3,6]
x2 = [10,10]
# We do not transform the data.

print(polynomial_kernel(x1, x2, degree=2)) # 8100
```

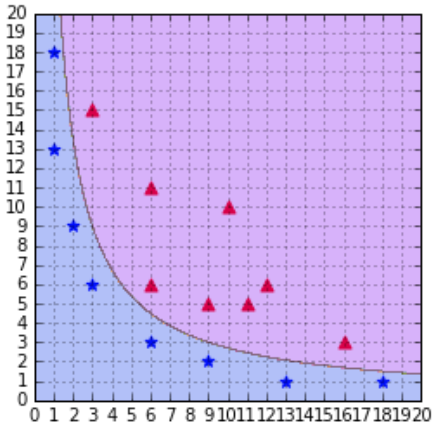

Figure 42: A SVM using a polynomial kernel is able to separate the data (degree=2)

Updating the degree

A polynomial kernel with a degree of 1 and no constant is simply the linear kernel (Figure *43*). When you increase the degree of a polynomial kernel, the decision boundary will become more complex and will have a tendency to be influenced by individual data examples, as illustrated in Figure 44. Using a high-degree polynomial is dangerous because you can often achieve better performance on your test set, but it leads to what is called **overfitting**: the model is too close to the data and does not generalize well.

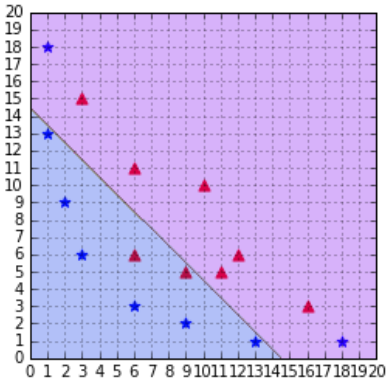

Figure 43: A polynomial kernel with degree = 1

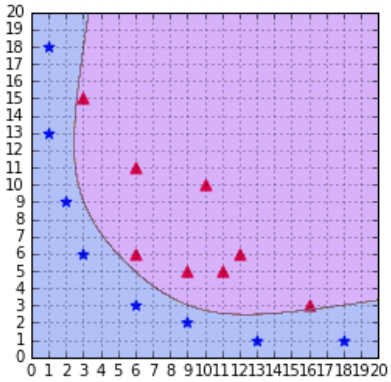

Figure 44: A polynomial kernel with degree = 6

Note: Using a high-degree polynomial kernel will often lead to overfitting.

RBF or Gaussian kernel

Sometimes polynomial kernels are not sophisticated enough to work. When you have a difficult dataset like the one depicted in Figure 45, this type of kernel will show its limitation.

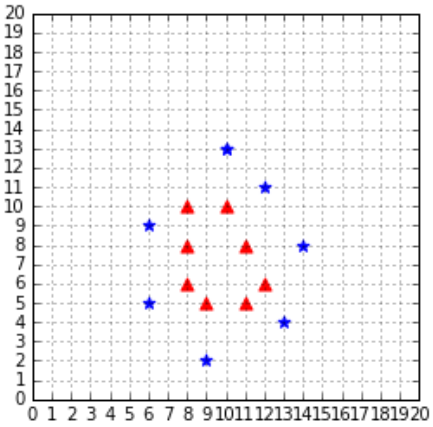

Figure 45: This dataset is more difficult to work with

As we can see in Figure 46, the decision boundary is very bad at classifying the data.

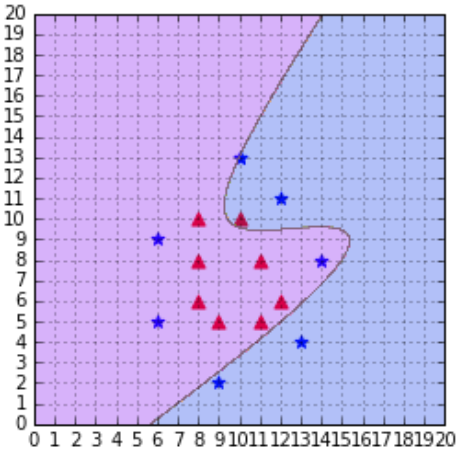

Figure 46: A polynomial kernel is not able to separate the data (degree=3, C=100)

This case calls for us to use another, more complicated, kernel: the **Gaussian kernel**. It is also named RBF kernel, where RBF stands for Radial Basis Function. A radial basis function is a function whose value depends only on the distance from the origin or from some point.

The RBF kernel function is:

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

You will often read that it projects vectors into an infinite dimensional space. What does this mean?

Recall this definition: a kernel is a **function** that returns the result of a dot product performed in another space.

In the case of the polynomial kernel example we saw earlier, the kernel returned the result of a dot product performed in $\mathbb{R}^3$. As it turns out, the RBF kernel returns the result of a dot product performed in $\mathbb{R}^\infty$.

I will not go into details here, but if you wish, you can read this proof to better understand how we came to this conclusion.

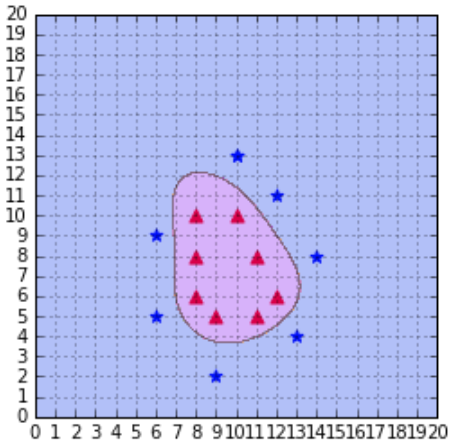

Figure 47: The RBF kernel classifies the data correctly with gamma = 0.1

This video is particularly useful to understand how the RBF kernel is able to separate the data.

Changing the value of gamma

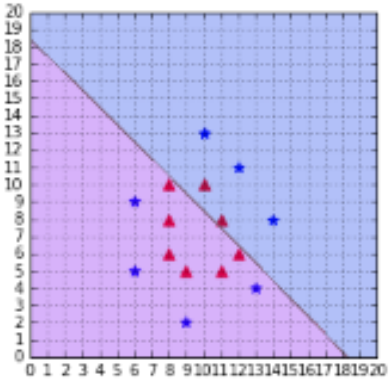

Figure 48: A Gaussian kernel with gamma = 1e-5

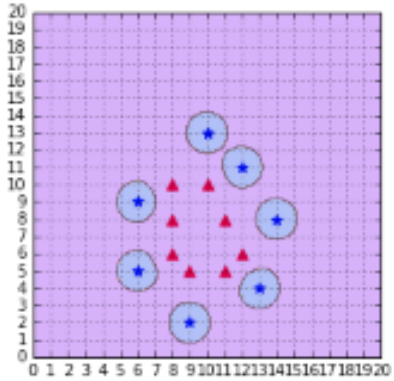

Figure 49: A Gaussian kernel with gamma = 2

When gamma is too small, as in Figure 48, the model behaves like a linear SVM. When gamma is too large, the model is too heavily influenced by each support vector, as shown in Figure 49. For more information about gamma, you can read this scikit-learn documentation page.

Other types

Research on kernels has been prolific, and there are now a lot of kernels available. Some of them are specific to a domain, such as the string kernel, which can be used when working with text. If you want to discover more kernels, this article from César Souza describes 25 kernels.

Which kernel should I use?

The recommended approach is to try a RBF kernel first, because it usually works well. However, it is good to try the other types of kernels if you have enough time to do so. **A kernel is a measure of the similarity between two vectors**, so that is where domain knowledge of the problem at hand may have the biggest impact. Building a custom kernel can also be a possibility, but it requires that you have a good mathematical understanding of the theory behind kernels. You can find more information on this subject in (Cristianini & Shawe-Taylor, 2000).

Summary

The kernel trick is one key component making Support Vector Machines powerful. It allows us to apply SVMs on a wide variety of problems. In this chapter, we saw the limitations of the linear kernel, and how a polynomial kernel can classify non-separable data. Eventually, we saw one of the most used and most powerful kernels: the RBF kernel. Do not forget that there are many kernels, and try looking for kernels created to solve the kind of problems you are trying to solve. Using the right kernel with the right dataset is one key element in your success or failure with SVMs.

Chapter 7 The SMO Algorithm

We saw how to solve the SVM optimization problem using a convex optimization package. However, in practice, we will use an algorithm specifically created to solve this problem quickly: the **SMO (sequential minimal optimization**) **algorithm**. Most machine learning libraries use the SMO algorithm or some variation.

The SMO algorithm will solve the following optimization problem:

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j K(\mathbf{x}_i,\mathbf{x}_j) - \sum_{i=1}^{m}\alpha_i \\ \text{subject to} \quad & 0 \le \alpha_i \le C, \text{ for any } i = 1,\ldots,m \\ & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

It is a kernelized version of the soft-margin formulation we saw in Chapter 5. The objective function we are trying to minimize can be written in Python (Code Listing 37):

Code Listing 37

```
def kernel(x1, x2):
    return np.dot(x1, x2.T)


def objective_function_to_minimize(X, y, a, kernel):
    m, n = np.shape(X)
    return 1 / 2 * np.sum([a[i] * a[j] * y[i] * y[j]* kernel(X[i, :], X[j, :])
                           for j in range(m)
                           for i in range(m)])\
           - np.sum([a[i] for i in range(m)])
```

This is the same problem we solved using CVXOPT. Why do we need another method? Because we would like to be able to use SVMs with big datasets, and using convex optimization packages usually involves matrix operations that take a lot of time as the size of the matrix increases or become impossible because of memory limitations. The SMO algorithm has been created with the goal of being faster than other methods.

The idea behind SMO

When we try to solve the SVM optimization problem, we are free to change the values of α as long as we respect the constraints. Our goal is to modify α so that in the end, the objective function returns the smallest possible value. In this context, given a vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ of Lagrange multipliers, we can change the value of any α_i until we reach our goal.

The idea behind SMO is quite easy: we will solve a simpler problem. That is, given a vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_m)$, we will allow ourselves to change only two values of α, for instance, α_3 and α_7. We will change them until the objective function reaches its minimum given this set of alphas. Then we will pick two other alphas and change them until the function returns its smallest value, and so on. If we continue doing that, we will eventually reach the minimum of the objective function of the original problem.

SMO solves a sequence of several simpler optimization problems.

How did we get to SMO?

This idea of solving several simpler optimization problems is not new. In 1982, Vapnik proposed a method known as “chunking,” which breaks the original problem down into a series of smaller problems (Vapnik V. , 1982). What made things change is that in 1997, Osuna, et al., proved that solving a sequence of sub-problems will be guaranteed to converge as long as we add at least one example violating the KKT conditions (Osuna, Freund, & Girosi, 1997).

Using this result, one year later, in 1998, Platt proposed the SMO algorithm.

Why is SMO faster?

The great advantage of the SMO approach is that we do not need a QP solver to solve the problem for two Lagrange multipliers—it can be solved analytically. As a consequence, it does not need to store a huge matrix, which can cause problems with machine memory. Moreover, SMO uses several heuristics to speed up the computation.

The SMO algorithm

The SMO algorithm is composed of three parts:

- One heuristic to choose the first Lagrange multiplier
- One heuristic to choose the second Lagrange multiplier
- The code to solve the optimization problem analytically for the two chosen multipliers

Tip: A Python implementation of the algorithm is available in Appendix B: The SMO Algorithm. All code listings in this section are taken from this appendix and do not work alone.

The analytical solution

At the beginning of the algorithm, we start with a vector $\alpha(\alpha_1, \alpha_2, \ldots, \alpha_m)$ in which $\alpha_i = 0 \quad \text{for all } i = 1, \ldots, m$. The idea is to pick two elements of this vector, which we will name α_1 and α_2, and to change their values so that the constraints are still respected.

The first constraint $0 \leq \alpha_i \leq C, \text{ for any } i = 1, \ldots, m$ means that $0 \leq \alpha_1 \leq C$ and $0 \leq \alpha_2 \leq C$. That is why we are forced to select a value lying in the blue box of Figure 50 (which displays an example where $C = 5$).

The second constraint is a linear constraint $\sum_{i=1}^{m} \alpha_i y_i = 0$. It forces the values to lie on the red diagonal, and the first couple of selected α_1 and α_2 should have different labels ($y_1 \neq y_2$).

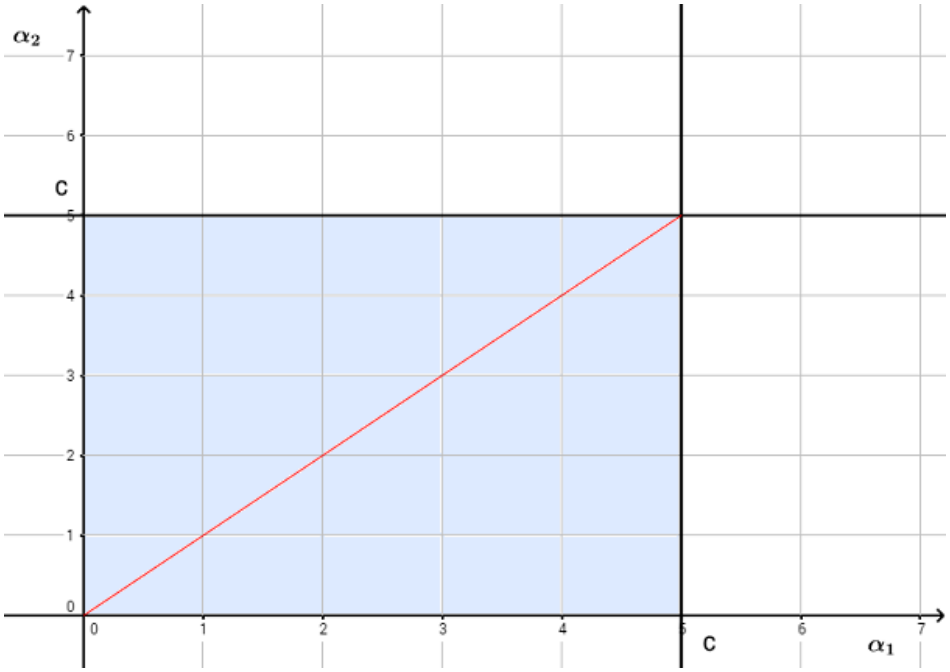

Figure 50: The feasible set is the diagonal of the box

In general, to avoid breaking the linear constraint, we must change the multipliers so that:

$$\alpha_1 y_1 + \alpha_2 y2 = constant = \alpha_1^{old} y_1 + \alpha_2^{old} y2$$

We will not go into the details of how the problem is solved analytically, as it is done very well in (Cristianini & Shawe-Taylor, 2000) and in (Platt J. C., 1998).

Remember that there is a formula to compute the new α_2 :

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{K(\mathbf{x}_1, \mathbf{x}_1) + K(\mathbf{x}_2, \mathbf{x}_2) - 2K(\mathbf{x}_1, \mathbf{x}_2)}$$

with $E_i = f(\mathbf{x}_i) - yi$ being the difference between the output of the hypothesis function and the example label. K is the kernel function. We also compute bounds, which applies to α_2^{new}; it cannot be smaller than the lower bound, or larger than the upper bound, or constraints will be violated. So α_2^{new} is clipped if this is the case.

Once we have this new value, we use it to compute the new α_1 using this formula:

$$\alpha_1^{new} = \alpha_1^{old} + y_1 y_2(\alpha_2^{old} - \alpha_2^{new})$$

Understanding the first heuristic

The idea behind the first heuristic is pretty simple: each time SMO examines an example, it checks whether or not the KKT conditions are violated. Recall that at least one KKT condition must be violated. If the conditions are met, then it tries another example. So if there are millions of examples, and only a few of them violate the KKT conditions, it will spend a lot of time examining useless examples. In order to avoid that, the algorithm concentrates its time on examples in which the Lagrange multiplier is not equal to 0 or C, because they are the most likely to violate the conditions (Code Listing 38).

Code Listing 38

```
def get_non_bound_indexes(self):
    return np.where(np.logical_and(self.alphas > 0,
                                   self.alphas < self.C))[0]

# First heuristic: loop  over examples where alpha is not 0 and not C
# they are the most likely to violate the KKT conditions
# (the non-bound subset).
def first_heuristic(self):
    num_changed = 0
    non_bound_idx = self.get_non_bound_indexes()

    for i in non_bound_idx:
        num_changed += self.examine_example(i)
    return num_changed
```

Because solving the problem analytically involves two Lagrange multipliers, it is possible that a bound multiplier (whose value is between 0 and C) has become KKT-violated. That is why the main routine alternates between all examples and the non-bound subset (Code Listing 39). Note that the algorithm finishes when progress is no longer made.

Code Listing 39

```
def main_routine(self):
    num_changed = 0
    examine_all = True

    while num_changed > 0 or examine_all:
        num_changed = 0

        if examine_all:
            for i in range(self.m):
                num_changed += self.examine_example(i)
        else:
            num_changed += self.first_heuristic()

        if examine_all:
            examine_all = False
        elif num_changed == 0:
            examine_all = True
```

Understanding the second heuristic

The goal of this second heuristic is to select the Lagrange multiplier for which the step taken will be maximal.

How do we update α_2 ? We use the following formula:

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{K(\mathbf{x}_1, \mathbf{x}_1) + K(\mathbf{x}_2, \mathbf{x}_2) - 2K(\mathbf{x}_1, \mathbf{x}_2)}$$

Remember that in this case that we have already chosen the value α_1. Our goal is to pick the α_2 whose α_2^{new} will have the biggest change. This formula can be rewritten as follows:

$$\alpha_2^{new} = \alpha_2 + \text{step}$$

with:

$$\text{step} = \frac{y_2(E_1 - E_2)}{K(\mathbf{x}_1, \mathbf{x}_1) + K(\mathbf{x}_2, \mathbf{x}_2) - 2K(\mathbf{x}_1, \mathbf{x}_2)}$$

So, to pick the best α_2 amongst several α_i, we need to compute the value of step for each α_i and select the one with the biggest step. The problem here is that we need to call the kernel function K three times for each step, and this is costly. Instead of doing that, Platt came with the following approximation:

$$\text{step} \approx |E_1 - E_2|$$

As a result, selecting the biggest step is done by taking the α_i with the smallest error if E_1 is positive, and the α_i with the biggest error if E_1 is negative.

This approximation is visible in the method **second_heuristic** of Code Listing 40.

Code Listing 40

```
def second_heuristic(self, non_bound_indices):
    i1 = -1
    if len(non_bound_indices) > 1:
    max = 0

    for j in non_bound_indices:
        E1 = self.errors[j] - self.y[j]
        step = abs(E1 - self.E2) # approximation
        if step > max:
            max = step
            i1 = j
    return i1

def examine_example(self, i2):
    self.y2 = self.y[i2]
    self.a2 = self.alphas[i2]
    self.X2 = self.X[i2]
    self.E2 = self.get_error(i2)

    r2 = self.E2 * self.y2

    if not((r2 < -self.tol and self.a2 < self.C) or
           (r2 > self.tol and self.a2 > 0)):
        # The KKT conditions are met, SMO looks at another example.
        return 0

    # Second heuristic A: choose the Lagrange multiplier that
    # maximizes the absolute error.
    non_bound_idx = list(self.get_non_bound_indexes())
    i1 = self.second_heuristic(non_bound_idx)

    if i1 >= 0 and self.take_step(i1, i2):
        return 1

    # Second heuristic B: Look for examples making positive
    # progress by looping over all non-zero and non-C alpha,
    # starting at a random point.
    if len(non_bound_idx) > 0:
        rand_i = randrange(len(non_bound_idx))
        for i1 in non_b ound_idx[rand_i:] + non_bound_idx[:rand_i]:
            if self.take_step(i1, i2):
                return 1

    # Second heuristic C: Look for examples making positive progress
    # by looping over all possible examples, starting at a random
```

```
    # point.
    rand_i = randrange(self.m)
    all_indices = list(range(self.m))
    for i1 in all_indices[rand_i:] + all_indices[:rand_i]:
        if self.take_step(i1, i2):
        return 1

    # Extremely degenerate circumstances, SMO skips the first example.
    return 0
```

Summary

Understanding the SMO algorithm can be tricky because a lot of the code is here for performance reasons, or to handle specific degenerate cases. However, at its core, the algorithm remains simple and is faster than convex optimization solvers. Over time, people have discovered new heuristics to improve this algorithm, and popular libraries like LIBSVM use an SMO-like algorithm. Note that even if this is the standard way of solving the SVM problem, other methods exist, such as gradient descent and stochastic gradient descent (SGD), which is particularly used for online learning and dealing with huge datasets.

Knowing how the SMO algorithm works will help you decide if it is the best method for the problem you want to solve. I strongly advise you to try implementing it yourself. In the Stanford CS229 course, you can find the description of a simplified version of the algorithm, which is a good start. Then, in *Sequential Minimal Optimization* (Platt J. C., 1998), you can read the full description of the algorithm. The Python code available in Appendix B has been written from the pseudo-code from this paper and indicates in comments which parts of the code correspond to which equations in the paper.

Chapter 8 Multi-Class SVMs

SVMs are able to generate binary classifiers. However, we are often faced with datasets having more than two classes. For instance, the original wine dataset actually contains data from three different producers. There are several approaches that allow SVMs to work for multi-class classification. In this chapter, we will review some of the most popular multi-class methods and explain where they come from.

For all code examples in this chapter, we will use the dataset generated by Code Listing 41 and displayed in Figure 51.

Code Listing 41

```
import numpy as np

def load_X():
    return np.array([[1, 6], [1, 7], [2, 5], [2, 8],
                     [4, 2], [4, 3], [5, 1], [5, 2],
                     [5, 3], [6, 1], [6, 2], [9, 4],
                     [9, 7], [10, 5], [10, 6], [11, 6],
                     [5, 9], [5, 10], [5, 11], [6, 9],
                     [6, 10], [7, 10], [8, 11]])


def load_y():
    return np.array([1, 1, 1, 1,
                     2, 2, 2, 2, 2, 2, 2,
                     3, 3, 3, 3, 3,
                     4, 4, 4, 4, 4, 4, 4])
```

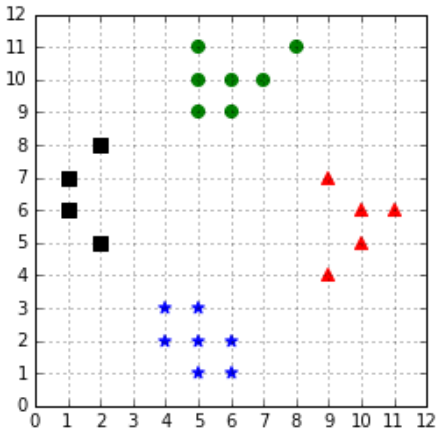

Figure 51: A four classes classification problem

Solving multiple binary problems

One-against-all

Also called “one-versus-the-rest,” this is probably the simplest approach.

In order to classify K classes, we construct K different binary classifiers. For a given class, the positive examples are all the points in the class, and the negative examples are all the points not in the class (Code Listing 42).

Code Listing 42

```
import numpy as np
from sklearn import svm


# Create a simple dataset

X = load_X()
y = load_y()


# Transform the 4 classes problem
# in 4 binary classes problems.
y_1 = np.where(y == 1, 1, -1)
y_2 = np.where(y == 2, 1, -1)
y_3 = np.where(y == 3, 1, -1)
y_4 = np.where(y == 4, 1, -1)
```

We train one binary classifier on each problem (Code Listing 43). As a result, we obtain one decision boundary per classifier (in Figure 52).

Code Listing 43

```
# Train one binary classifier on each problem.
y_list = [y_1, y_2, y_3, y_4]

classifiers = []
for y_i in y_list:
    clf = svm.SVC(kernel='linear', C=1000)
    clf.fit(X, y_i)
    classifiers.append(clf)
```

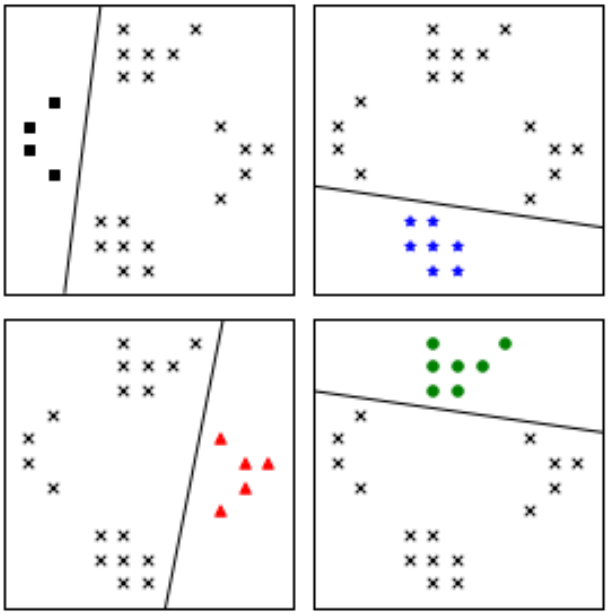

Figure 52: The One-against-all approach creates one classifier per class

In order to make a new prediction, we use each classifier and predict the class of the classifier if it returns a positive answer (Code Listing 44). However, this can give inconsistent results because a label is assigned to multiple classes simultaneously or to none (Bishop, 2006). Figure illustrates this problem; the one-against-all classifier is not able to predict a class for the examples in the blue areas in each corner because two classifiers are making a positive prediction. This would result in the example having two class simultaneously. The same problem occurs in the center because each classifier makes a negative prediction. As a result, no class can be assigned to an example in this region.

Code Listing 44

```
def predict_class(X, classifiers):
    predictions = np.zeros((X.shape[0], len(classifiers)))
    for idx, clf in enumerate(classifiers):
        predictions[:, idx] = clf.predict(X)

    # returns the class number if only one classifier predicted it
    # returns zero otherwise.
    return np.where((predictions == 1).sum(1) == 1,
                    (predictions == 1).argmax(axis=1) + 1,
                    0)
```

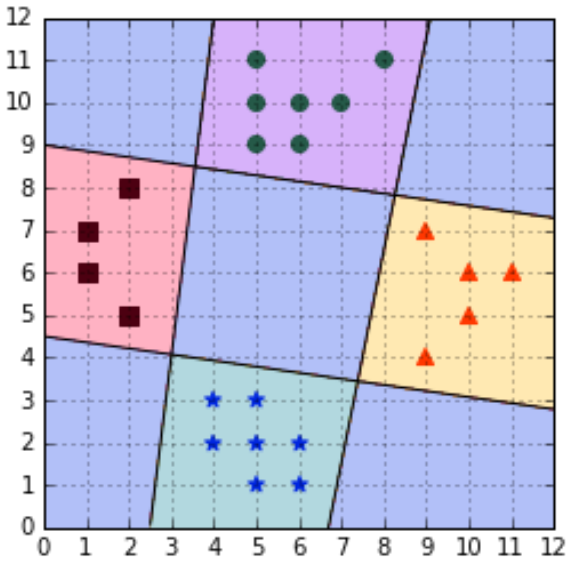

Figure 53: One-against-all leads to ambiguous decisions

As an alternative solution, Vladimir Vapnik suggested using the class of the classifier for which the value of the decision function is the maximum (Vapnik V. N., 1998). This is demonstrated in Code Listing 45. Note that we use the **`decision_function`** instead of calling the **`predict`** method of the classifier. This method returns a real value that will be positive if the example is on the correct side of the classifier, and negative if it is on the other side. It is interesting to note that by taking the maximum of the value, and not the maximum of the absolute value, this approach will choose the class of the hyperplane the closest to the example when all classifiers disagree. For instance, the example point (6,4) in Figure will be assigned the blue star class.

Code Listing 45

```
def predict_class(X, classifiers):
    predictions = np.zeros((X.shape[0], len(classifiers)))
    for idx, clf in enumerate(classifiers):
        predictions[:, idx] = clf.decision_function(X)

    # return the argmax of the decision function as suggested by Vapnik.
    return np.argmax(predictions, axis=1) + 1
```

Applying this heuristic gives us classification results with no ambiguity, as shown in Figure . The major flaw of this approach is that the different classifiers were trained on different tasks, so there is no guarantee that the quantities returned by the **`decision_function`** have the same scale (Bishop, 2006). If one decision function returns a result ten times bigger than results of the others, its class will be assigned incorrectly to some examples.

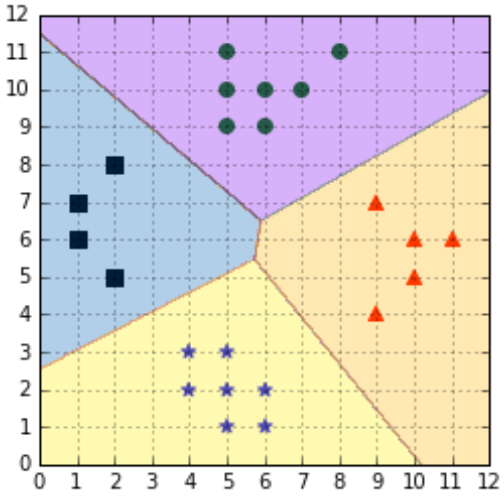

Figure 54: Applying a simple heuristic avoids the ambiguous decision problem

Another issue with the one-against-all approach is that training sets are imbalanced (Bishop, 2006). For a problem with 100 classes, each having 10 examples, each classifier will be trained with 10 positive examples and 990 negative examples. Thus, the negative examples will influence the decision boundary greatly.

Nevertheless, one-against-all remains a popular method for multi-class classification because it is easy to implement and understand.

Note: "[...] In practice the one-versus-the-rest approach is the most widely used in spite of its ad-hoc formulation and its practical limitations." (Bishop, 2006)

When using **sklearn**, **LinearSVC** automatically uses the one-against-all strategy by default. You can also specify it explicitly by setting the **multi_class** parameter to **ovr** (one-vs-the-rest), as shown in Code Listing 46.

Code Listing 46

```
from sklearn.svm import LinearSVC
import numpy as np

X = load_X()
y = load_y()

clf = LinearSVC(C=1000, random_state=88, multi_class='ovr')
clf.fit(X,y)

# Make predictions on two examples.
X_to_predict = np.array([[5,5],[2,5]])
print(clf.predict(X_to_predict)) # prints [2 1]
```

One-against-one

In this approach, instead of trying to distinguish one class from all the others, we seek to distinguish one class from another one. As a result, we train one classifier per pair of classes, which leads to *K(K-1)/2* classifiers for *K* classes. Each classifier is trained on a subset of the data and produces its own decision boundary (Figure).

Predictions are made using a simple **voting strategy**. Each example we wish to predict is passed to each classifier, and the predicted class is recorded. Then, the class having the most votes is assigned to the example (Code Listing 47).

Code Listing 47

```
from itertools import combinations
from scipy.stats import mode
from sklearn import svm
import numpy as np

# Predict the class having the max number of votes.
def predict_class(X, classifiers, class_pairs):
    predictions = np.zeros((X.shape[0], len(classifiers)))
    for idx, clf in enumerate(classifiers):
        class_pair = class_pairs[idx]
        prediction = clf.predict(X)
        predictions[:, idx] = np.where(prediction == 1,
                                       class_pair[0], class_pair[1])
    return mode(predictions, axis=1)[0].ravel().astype(int)

X = load_X()
y = load_y()

# Create datasets.
training_data = []
class_pairs = list(combinations(set(y), 2))
for class_pair in class_pairs:
    class_mask = np.where((y == class_pair[0]) | (y == class_pair[1]))
    y_i = np.where(y[class_mask] == class_pair[0], 1, -1)
    training_data.append((X[class_mask], y_i))

# Train one classifier per class.
classifiers = []
for data in training_data:
    clf = svm.SVC(kernel='linear', C=1000)
    clf.fit(data[0], data[1])
    classifiers.append(clf)

# Make predictions on two examples.
X_to_predict = np.array([[5,5],[2,5]])
print(predict_class(X_to_predict, classifiers, class_pairs))
```

```
# prints [2 1]
```

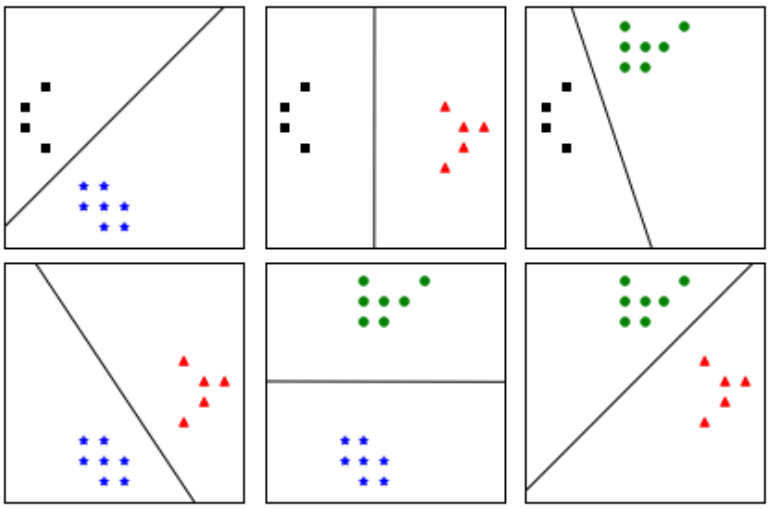

Figure 55: One-against-one construct with one classifier for each pair of classes

With this approach, we are still faced with the ambiguous classification problem. If two classes have an identical number of votes, it has been suggested that selecting the one with the smaller index might be a viable (while probably not the best) strategy (Hsu & Lin, *A Comparison of Methods for Multi-class Support Vector Machines,* 2002*)*.

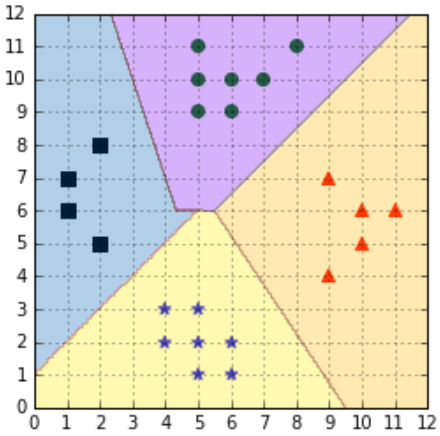

Figure 56: Predictions are made using a voting scheme

Figure shows us that the decision regions generated by the one-against-one strategy are different from the ones generated by one-against-all (Figure). In Figure , it is interesting to note that for regions generated by the one-against-one classifier, a region changes its color only after traversing a hyperplane (denoted by black lines), while this is not the case with one-against-all.

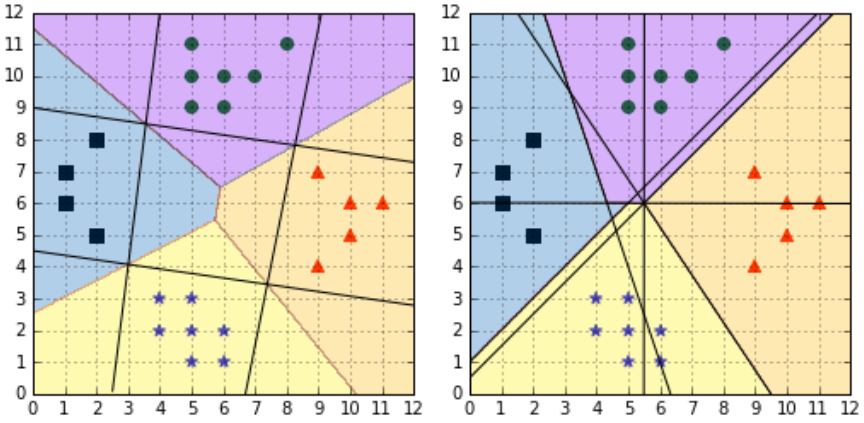

Figure 57: Comparison of one-against-all (left) and one-against-one (right)

The one-against-one approach is the default approach for multi-class classification used in **sklearn**. Instead of Code Listing 47, you will obtain the exact same results using the code of Code Listing 48.

Code Listing 48

```
from sklearn import svm
import numpy as np

X = load_X()
y = load_y()

# Train a multi-class classifier.
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X,y)

# Make predictions on two examples.
X_to_predict = np.array([[5,5],[2,5]])
print(clf.predict(X_to_predict)) # prints [2 1]
```

One of the main drawbacks of the one-against-all method is that the classifier will tend to overfit. Moreover, the size of the classifier grows super-linearly with the number of classes, so this method will be slow for large problems (Platt, Cristianini, & Shawe-Taylor, 2000).

DAGSVM

DAGSVM stands for “Directed Acyclic Graph SVM.” It has been proposed by John Platt et al. in 2000 as an improvement of one-against-one (Platt, Cristianini, & Shawe-Taylor, 2000).

Note: John C. Platt invented the SMO algorithm and Platt Scaling, and proposed the DAGSVM. Quite a contribution to the SVMs world!

The idea behind DAGSVM is to use the same training as one-against-one, but to speed up testing by using a directed acyclic graph (DAG) to choose which classifiers to use.

If we have four classes A, B, C, and D, and six classifiers trained each on a pair of classes: (A, B); (A, C); (A, D); (B, C); (B, D); and (C, D). We use the first classifier, (A, D), and it predicts class A, which is the same as predicting **not** class D, and the second classifier also predits class A (**not** class C). It means that classifiers (B, D), (B, C) or (C, D) can be ignored because we already know the class is neither C nor D. The last "useful" classifier is (A, B), and if it predicts B, we assign the class B to the data-point. This example is illustrated with the red path in Figure . Each node of the graph is a classifier for a pair of class.

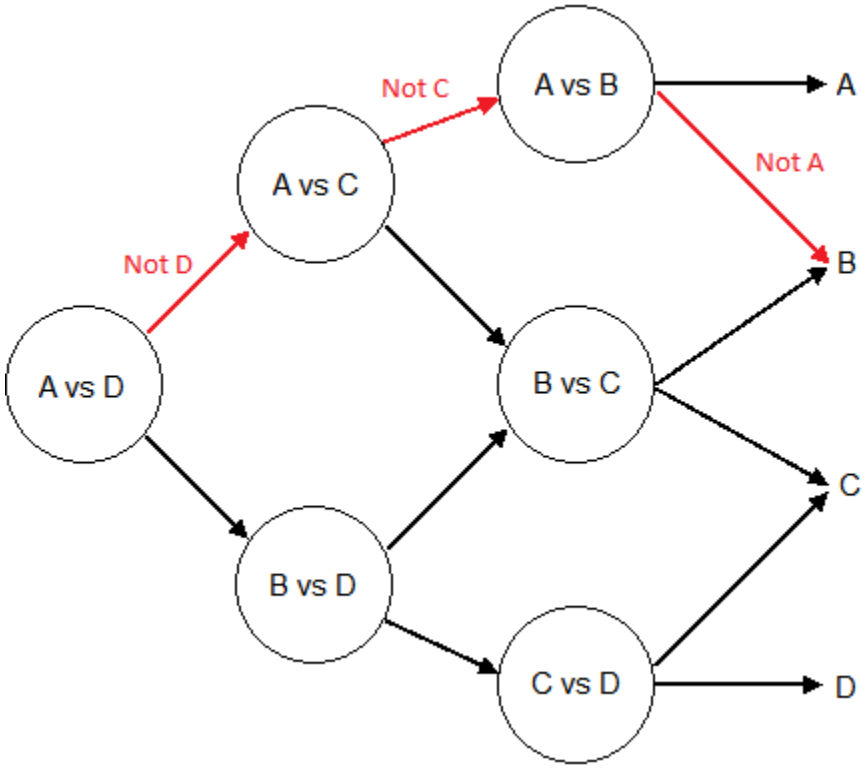

Figure 58: Illustration of the path used to make a prediction along a Directed Acyclic graph

With four classes, we used three classifiers to make the prediction, instead of six with one-against-one. In general, for a problem with *K* classes, *K-1* decision nodes will be evaluated.

Substituting the **`predict_class`** function in Code Listing 47 with the one in Code Listing 49 gives the same result, but with the benefit of using fewer classifiers.

In Code Listing 49, we implement the DAGSVM approach with a list. We begin with the list of possible classes, and after each prediction, we remove the one that has been disqualified. In the end, the remaining class is the one which should be assigned to the example.

Note that Code Listing 49 is here for illustration purposes and should not be used in your production code, as it is not fast when the dataset (X) is large.

Code Listing 49

```
def predict_class(X, classifiers, distinct_classes, class_pairs):
    results = []
    for x_row in X:

        class_list = list(distinct_classes)

        # After each prediction, delete the rejected class
        # until there is only one class.
        while len(class_list) > 1:
            # We start with the pair of the first and
            # last element in the list.
            class_pair = (class_list[0], class_list[-1])
            classifier_index = class_pairs.index(class_pair)
            y_pred = classifiers[classifier_index].predict(x_row)

            if y_pred == 1:
                class_to_delete = class_pair[1]
            else:
                class_to_delete = class_pair[0]

            class_list.remove(class_to_delete)

        results.append(class_list[0])
    return np.array(results)
```

Note: "The DAGSVM is between a factor 1.6 and 2.3 times faster to evaluate than Max Wins." (Platt, Cristianini, & Shawe-Taylor, 2000).

Solving a single optimization problem

Instead of trying to solve several binary optimization problems, another approach is to try to solve a single optimization problem. This approach has been proposed by several people over the years.

Vapnik, Weston, and Watkins

This method is a generalization of the SVMs optimization problem to solve the multi-class classification problem directly. It has been independently discovered by Vapnik (Vapnik V. N., 1998) and Weston & Watkins (Weston & Watkins, 1999). For every class, constraints are added to the optimization problem. As a result, the size of the problem is proportional to the number of classes and can be very slow to train.

Crammer and Singer

Crammer and Singer (C&S) proposed an alternative approach to multi-class SVMs. Like Weston and Watkins, they solve a single optimization problem, but with fewer slack variables (Crammer & Singer, 2001). This has the benefit of reducing the memory and training time. However, in their comparative study, Hsu & Lin found that the C&S method was especially slow when using a large value for the *C* regularization parameter (Hsu & Lin, A Comparison of Methods for Multi-class Support Vector Machines, 2002).

In **sklearn**, when using **LinearSVC** you can choose to use the C&S algorithm (Code Listing 50). In Figure , we can see that the C&S predictions are different from the one-against-all and the one-against-one methods.

Code Listing 50

```
from sklearn import svm
import numpy as np

X = load_X()
y = load_y()

clf = svm.LinearSVC(C=1000, multi_class='crammer_singer')
clf.fit(X,y)

# Make predictions on two examples.
X_to_predict = np.array([[5,5],[2,5]])
print(clf.predict(X_to_predict)) # prints [4 1]
```

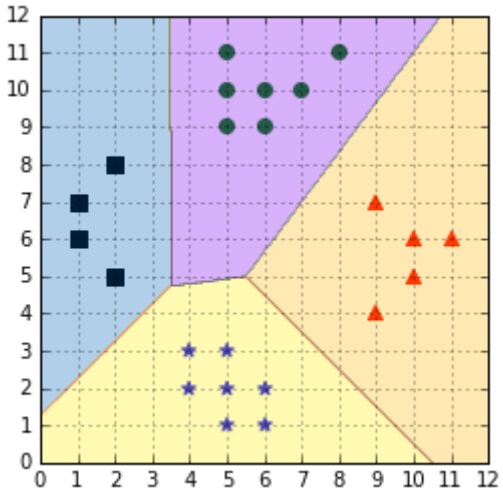

Figure 59: Crammer & Singer algorithm predictions

Which approach should you use?

With so many options available, choosing which multi-class approach is better suited for your problem can be difficult.

Hsu and Lin wrote an interesting paper comparing the different multi-class approaches available for SVMs (Hsu & Lin, *A Comparison of Methods for Multi-class Support Vector Machines*, 2002). They conclude that "the one-against-one and DAG methods are more suitable for practical use than the other methods." The one-against-one method has the added advantage of being already available in **`sklearn`**, so it should probably be your default choice.

Be sure to remember that **`LinearSVC`** uses the one-against-all method by default, and that maybe using the Crammer & Singer algorithm will better help you achieve your goal. On this topic, Dogan et al. found that despite being considerably faster than other algorithms, one-against-all yield hypotheses with a statistically significantly worse accuracy (Dogan, Glasmachers, & Igel, 2011). Table 1 provides an overview of the methods presented in this chapter to help you make a choice.

Table 1: Overview of multi-class SVM methods

Method name	One-against-all	One-against-one	Weston and Watkins	DAGSVM	Crammer and Singer
First SVMs usage	1995	1996	1999	2000	2001
Approach	Use several binary classifiers	Use several binary classifiers	Solve a single optimization problem	Use several binary classifiers	Solve a single optimization problem
Training approach	Train a single classifier for each class	Train a classifier for each pair of classes	Decomposition method	Same as one-against-one	Decomposition method
Number of trained classifiers (K is the number of classes)	K	$\frac{K(K-1)}{2}$	1	$\frac{K(K-1)}{2}$	1
Testing approach	Select the class with the biggest decision function value	"Max-Wins" voting strategy	Use the classifier	Use a DAG to make predictions on K-1 classifiers	Use the classifier
scikit-learn class	LinearSVC	SVC	Not available	Not available	LinearSVC
Drawbacks	Class imbalance	Long training time for large K	Long training time	Not available in popular libraries	Long training time

Summary

Thanks to many improvements over the years, there are now several methods for doing multi-class classification with SVMs. Each approach has advantages and drawbacks, and most of the time you will end up using the one available in the library you are using. However, if necessary, you now know which method can be more helpful to solve your specific problem.

Research on multi-class SVMs is not over. Recent papers on the subject have been focused on distributed training. For instance, Han & Berg have presented a new algorithm called "Distributed Consensus Multiclass SVM," which uses consensus optimization with a modified version of Crammer & Singer's formulation (Han & Berg, 2012).

Conclusion

To conclude, I will quote Stuart Russel and Peter Norvig, who wrote:

"*You could say that SVMs are successful because of one key insight, one neat trick.*"

(Russell & Norvig, 2010)

The key insight is the fact that some examples are more important than others. They are the closest to the decision boundary, and we call them **support vectors**. As a result, we discover that the optimal hyperplane generalizes better than other hyperplanes, and can be constructed using support vectors only. We saw in detail that we need to solve a convex optimization problem to find this hyperplane.

The neat trick is the **kernel trick**. It allows us to use SVMs with non-separable data, and without it, SVMs would be very limited. We saw that this trick, while it can be difficult to grasp at first, is in fact quite simple, and can be reused in other learning algorithms.

That's it. If you have read this book cover to cover, you should now understand *how* SVMs work. Another interesting question is *why* do they work? It is the subject of a field called computational learning theory (SVMs are in fact coming from statistical learning theory). If you wish to learn more about this, you can follow this outstanding course or read *Learning from Data* (Abu-Mostafa, 2012), which provides a very good introduction on the subject.

You should know that SVMs are not used only for classification. One-Class SVM can be used for anomaly detection, and Support Vector Regression can be used for regression. They have not been included in this book in order to keep it succinct, but they are equally interesting topics. Now that you understand the basic SVMs, you should be better prepared to study these derivations.

SVMs will not be the solution to all your problems, but I do hope they will now be a tool in your machine-learning toolbox—a tool that you understand, and that you will enjoy using.

Appendix A: Datasets

Linearly separable dataset

The following code is used to load the simple linearly separable dataset used in most chapters of this book. You can find the code source of the other datasets used in this book in this Bitbucket repository.

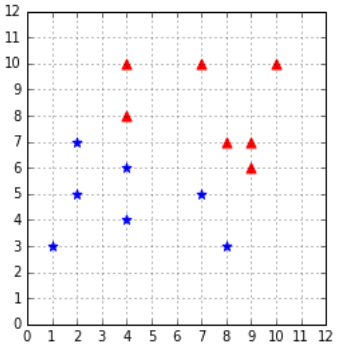

Figure 60: The training set

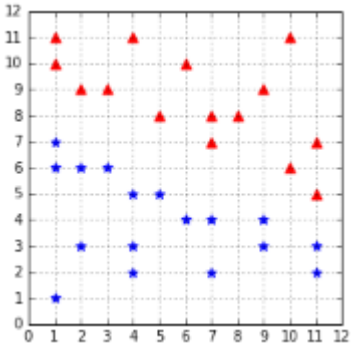

Figure 61: The test set

When a code listing imports the module as in Code Listing 51, it loads the methods displayed in Code Listing 52.

The method **get_training_examples** returns the data shown in Figure 59, while the method **get_test_examples** returns the data of Figure 60.

The method **get_training_examples** returns the data shown in Figure 60, while the method **get_test_examples** returns the data of Figure 61.

Code Listing 51

```
from succinctly.datasets import *
```

The method **get_training_examples** returns the data shown in Figure, while the method **get_test_examples** returns the data of Figure .

Code Listing 52

```
import numpy as np

def get_training_examples():
    X1 = np.array([[8, 7], [4, 10], [9, 7], [7, 10],
                   [9, 6], [4, 8], [10, 10]])
```

```
    y1 = np.ones(len(X1))
    X2 = np.array([[2, 7], [8, 3], [7, 5], [4, 4],
                   [4, 6], [1, 3], [2, 5]])
    y2 = np.ones(len(X2)) * -1
    return X1, y1, X2, y2


def get_test_examples():
    X1 = np.array([[2, 9], [1, 10], [1, 11], [3, 9], [11, 5],
                   [10, 6], [10, 11], [7, 8], [8, 8], [4, 11],
                   [9, 9], [7, 7], [11, 7], [5, 8], [6, 10]])
    X2 = np.array([[11, 2], [11, 3], [1, 7], [5, 5], [6, 4],
                   [9, 4],[2, 6], [9, 3], [7, 4], [7, 2], [4, 5],
                   [3, 6], [1, 6], [2, 3], [1, 1], [4, 2], [4, 3]])
    y1 = np.ones(len(X1))
    y2 = np.ones(len(X2)) * -1
    return X1, y1, X2, y2
```

A typical usage of this code is shown in Code Listing 53. It uses the method **get_dataset** from Code Listing 54, which is loaded with the **datasets** package.

Code Listing 53

```
from succinctly.datasets import get_dataset, linearly_separable as ls

# Get the training examples of the linearly separable dataset.
X, y = get_dataset(ls.get_training_examples)
```

Code Listing 54

```
import numpy as np

def get_dataset(get_examples):
    X1, y1, X2, y2 = get_examples()
    X, y = get_dataset_for(X1, y1, X2, y2)
    return X, y

def get_dataset_for(X1, y1, X2, y2):
    X = np.vstack((X1, X2))
    y = np.hstack((y1, y2))
    return X, y

def get_generated_dataset(get_examples, n):
    X1, y1, X2, y2 = get_examples(n)
    X, y = get_dataset_for(X1, y1, X2, y2)
    return X, y
```

Appendix B: The SMO Algorithm

Code Listing 55

```
import numpy as np
from random import randrange


# Written from the pseudo-code in:
# http://luthuli.cs.uiuc.edu/~daf/courses/optimization/Papers/smoTR.pdf
class SmoAlgorithm:
    def __init__(self, X, y, C, tol, kernel, use_linear_optim):
        self.X = X
        self.y = y
        self.m, self.n = np.shape(self.X)
        self.alphas = np.zeros(self.m)

        self.kernel = kernel
        self.C = C
        self.tol = tol

        self.errors = np.zeros(self.m)
        self.eps = 1e-3 # epsilon

        self.b = 0

        self.w = np.zeros(self.n)
        self.use_linear_optim = use_linear_optim

    # Compute the SVM output for example i
    # Note that Platt uses the convention w.x-b=0
    # while we have been using w.x+b in the book.
    def output(self, i):
        if self.use_linear_optim:
            # Equation 1
            return float(np.dot(self.w.T, self.X[i])) - self.b
        else:
            # Equation 10
            return np.sum([self.alphas[j] * self.y[j]
                           * self.kernel(self.X[j], self.X[i])
                           for j in range(self.m)]) - self.b



    # Try to solve the problem analytically.
    def take_step(self, i1, i2):
```

```
        if i1 == i2:
            return False

        a1 = self.alphas[i1]
        y1 = self.y[i1]
        X1 = self.X[i1]
        E1 = self.get_error(i1)

        s = y1 * self.y2

        # Compute the bounds of the new alpha2.
        if y1 != self.y2:
            # Equation 13
            L = max(0, self.a2 - a1)
            H = min(self.C, self.C + self.a2 - a1)
        else:
            # Equation 14
            L = max(0, self.a2 + a1 - self.C)
            H = min(self.C, self.a2 + a1)

        if L == H:
            return False

        k11 = self.kernel(X1, X1)
        k12 = self.kernel(X1, self.X[i2])
        k22 = self.kernel(self.X[i2], self.X[i2])

        # Compute the second derivative of the
        # objective function along the diagonal.
        # Equation 15
        eta = k11 + k22 - 2 * k12

        if eta > 0:
            # Equation 16
            a2_new = self.a2 + self.y2 * (E1 - self.E2) / eta

            # Clip the new alpha so that is stays at the end of the line.
            # Equation 17
            if a2_new < L:
                a2_new = L
            elif a2_new > H:
                a2_new = H
        else:
            # Under unusual cicumstances, eta will not be positive.
            # Equation 19
            f1 = y1 * (E1 + self.b) - a1 * k11 - s * self.a2 * k12
            f2 = self.y2 * (self.E2 + self.b) - s * a1 * k12 \
                 - self.a2 * k22
            L1 = a1 + s(self.a2 - L)
```

```
            H1 = a1 + s * (self.a2 - H)
            Lobj = L1 * f1 + L * f2 + 0.5 * (L1 ** 2) * k11 \
                   + 0.5 * (L ** 2) * k22 + s * L * L1 * k12
            Hobj = H1 * f1 + H * f2 + 0.5 * (H1 ** 2) * k11 \
                   + 0.5 * (H ** 2) * k22 + s * H * H1 * k12

            if Lobj < Hobj - self.eps:
                a2_new = L
            elif Lobj > Hobj + self.eps:
                a2_new = H
            else:
                a2_new = self.a2

        # If alpha2 did not change enough the algorithm
        # returns without updating the multipliers.
        if abs(a2_new - self.a2) < self.eps * (a2_new + self.a2 \
                                               + self.eps):
            return False

        # Equation 18
        a1_new = a1 + s * (self.a2 - a2_new)

        new_b = self.compute_b(E1, a1, a1_new, a2_new, k11, k12, k22, y1)

        delta_b = new_b - self.b

        self.b = new_b

        # Equation 22
        if self.use_linear_optim:
            self.w = self.w + y1*(a1_new - a1)*X1 \
                     + self.y2*(a2_new - self.a2) * self.X2

        # Update the error cache using the new Lagrange multipliers.
        delta1 = y1 * (a1_new - a1)
        delta2 = self.y2 * (a2_new - self.a2)

        # Update the error cache.
        for i in range(self.m):
            if 0 < self.alphas[i] < self.C:
                self.errors[i] += delta1 * self.kernel(X1, self.X[i]) + \
                                  delta2 * self.kernel(self.X2,self.X[i]) \
                                  - delta_b

        self.errors[i1] = 0
        self.errors[i2] = 0

        self.alphas[i1] = a1_new
        self.alphas[i2] = a2_new
```

```
        return True

    def compute_b(self, E1, a1, a1_new, a2_new, k11, k12, k22, y1):
        # Equation 20
        b1 = E1 + y1 * (a1_new - a1) * k11 + \
             self.y2 * (a2_new - self.a2) * k12 + self.b

        # Equation 21
        b2 = self.E2 + y1 * (a1_new - a1) * k12 + \
             self.y2 * (a2_new - self.a2) * k22 + self.b

        if (0 < a1_new) and (self.C > a1_new):
            new_b = b1
        elif (0 < a2_new) and (self.C > a2_new):
            new_b = b2
        else:
            new_b = (b1 + b2) / 2.0
        return new_b

    def get_error(self, i1):
        if 0 < self.alphas[i1] < self.C:
            return self.errors[i1]
        else:
            return self.output(i1) - self.y[i1]

    def second_heuristic(self, non_bound_indices):
        i1 = -1
        if len(non_bound_indices) > 1:
            max = 0

            for j in non_bound_indices:
                E1 = self.errors[j] - self.y[j]
                step = abs(E1 - self.E2) # approximation
                if step > max:
                    max = step
                    i1 = j
        return i1

    def examine_example(self, i2):
        self.y2 = self.y[i2]
        self.a2 = self.alphas[i2]
        self.X2 = self.X[i2]
        self.E2 = self.get_error(i2)

        r2 = self.E2 * self.y2

        if not((r2 < -self.tol and self.a2 < self.C) or
               (r2 > self.tol and self.a2 > 0)):
```

```
            # The KKT conditions are met, SMO looks at another example.
            return 0

        # Second heuristic A: choose the Lagrange multiplier which
        # maximizes the absolute error.
        non_bound_idx = list(self.get_non_bound_indexes())
        i1 = self.second_heuristic(non_bound_idx)

        if i1 >= 0 and self.take_step(i1, i2):
            return 1

        # Second heuristic B: Look for examples making positive
        # progress by looping over all non-zero and non-C alpha,
        # starting at a random point.
        if len(non_bound_idx) > 0:
            rand_i = randrange(len(non_bound_idx))
            for i1 in non_bound_idx[rand_i:] + non_bound_idx[:rand_i]:
                if self.take_step(i1, i2):
                    return 1

        # Second heuristic C: Look for examples making positive progress
        # by looping over all possible examples, starting at a random
        # point.
        rand_i = randrange(self.m)
        all_indices = list(range(self.m))
        for i1 in all_indices[rand_i:] + all_indices[:rand_i]:
            if self.take_step(i1, i2):
                return 1

        # Extremely degenerate circumstances, SMO skips the first example.
        return 0

    def error(self, i2):
        return self.output(i2) - self.y2

    def get_non_bound_indexes(self):
        return np.where(np.logical_and(self.alphas > 0,
                                       self.alphas < self.C))[0]

    # First heuristic: loop  over examples where alpha is not 0 and not C
    # they are the most likely to violate the KKT conditions
    # (the non-bound subset).
    def first_heuristic(self):
        num_changed = 0
        non_bound_idx = self.get_non_bound_indexes()
        for i in non_bound_idx:
            num_changed += self.examine_example(i)
        return num_changed
```

```
    def main_routine(self):
        num_changed = 0
        examine_all = True

        while num_changed > 0 or examine_all:
            num_changed = 0

            if examine_all:
                for i in range(self.m):
                    num_changed += self.examine_example(i)
            else:
                num_changed += self.first_heuristic()

            if examine_all:
                examine_all = False
            elif num_changed == 0:
                examine_all = True
```

Code Listing 56 demonstrates how to instantiate an SmoAlgorithm object, run the algorithm, and print the result.

Code Listing 56

```
import numpy as np
from random import seed
from succinctly.datasets import linearly_separable, get_dataset
from succinctly.algorithms.smo_algorithm import SmoAlgorithm


def linear_kernel(x1, x2):
    return np.dot(x1, x2)


def compute_w(multipliers, X, y):
    return np.sum(multipliers[i] * y[i] * X[i] for i in range(len(y)))

if __name__ == '__main__':
    seed(5) # to have reproducible results

    X_data, y_data = get_dataset(linearly_separable.get_training_examples)
    smo = SmoAlgorithm(X_data, y_data, C=10, tol=0.001,
kernel=linear_kernel, use_linear_optim=True)

    smo.main_routine()

    w = compute_w(smo.alphas, X_data, y_data)
```

```
print('w = {}'.format(w))

# -smo.b because Platt uses the convention w.x-b=0
print('b = {}'.format(-smo.b))

# w = [0.4443664  1.1105648]
# b = -9.66268641132
```

Bibliography

Abu-Mostafa, Y. S. (2012). *Learning From Data.* AMLBook.

Biernat, E., & Lutz, M. (2016). *Data science: fondamentaux et études de cas.* Eyrolles.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* Springer.

Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization.* Cambridge University Press.

Burges, C. J. (1988). A Tutorial on Support Vector Machines for Pattern. *Data Mining and Knowledge Discovery*, 121-167.

Crammer, K., & Singer, Y. (2001). On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines. *Journal of Machine Learning Research 2*.

Cristianini, N., & Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines.* Cambridge University Press.

Dogan, U., Glasmachers, T., & Igel, C. (2011). *Fast Training of Multi-Class Support Vector Machines.*

El Ghaoui, L. (2015). *Optimization Models and Applications.* Retrieved from http://livebooklabs.com/keeppies/c5a5868ce26b8125

Gershwin, S. B. (2010). *KKT Examples.* Retrieved from MIT Mechanical Engineering Course: http://ocw.mit.edu/courses/mechanical-engineering/2-854-introduction-to-manufacturing-systems-fall-2010/lecture-notes/MIT2_854F10_kkt_ex.pdf

Gretton, A. (2016, 03 05). *Lecture 9: Support Vector Machines .* Retrieved from http://www.gatsby.ucl.ac.uk/~gretton/coursefiles/Slides5A.pdf

Han, X., & Berg, A. C. (2012). *DCMSVM: Distributed Parallel Training For Single-Machine Multiclass Classifiers.*

Hsu, C.-W., & Lin, C.-J. (2002). A Comparison of Methods for Multi-class Support Vector Machines. *IEEE transactions on neural networks*.

Hsu, C.-W., Chang, C.-C., & Lin, C.-J. (2016, 10 02). *A Practical Guide to Support Vector Classification.* Retrieved from LIBSVM website: http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

Ng, A. (n.d.). *CS229 Lecture notes - Part V Support Vector Machines.* Retrieved from http://cs229.stanford.edu/notes/cs229-notes3.pdf

Osuna, E., Freund, R., & Girosi, F. (1997). An Improved Training Algorithm for Support Vector. *Proceedings of IEEE NNSP'97*.

Platt, J. C. (1998). *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines.* Microsoft Research.

Platt, J. C., Cristianini, N., & Shawe-Taylor, J. (2000). *Large margin DAGs for multiclass classification.* MIT Press.

Rojas, R. (1996). *Neural Networks: A Systematic Introduction.* Springer.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach.* Pearson.

Tyson Smith, B. (2004). *Lagrange Multipliers Tutorial in the Context of Support Vector Machines.* Newfoundland.

Vapnik, V. (1982). *Estimation of Dependences Based on Empirical Data.* Springer.

Vapnik, V. N. (1998). *Statistical Learning Theory.* Wiley.

Weston, J., & Watkins, C. (1999). Support Vector Machines for Multi-Class Pattern Recognition. *Proceedings of the Seventh European Symposium on Artificial Neural Networks*.