Understanding Support Vector Machines

Arvind Saraswat

2023

Contents

List of Figures

Listings

Chapter 1

Introduction

The Support Vector Machine (SVM) stands out as one of the most effective off-the-shelf supervised machine learning algorithms. It's capable of delivering impressive results without requiring extensive adjustments when applied to a given problem. However, it is sometimes perceived as a black box due to the underlying strong mathematical foundation. In this book, we'll delve into the inner workings of SVM, exploring its core concepts. Since there are multiple types of SVMs, I will frequently refer to them while aiming to grasp their functionality. The ultimate goal of this book is to gain a comprehensive understanding of how SVMs operate.

SVMs have been developed over many years through the collaborative efforts of several individuals. The initial SVM algorithm was introduced by Vladimir Vapnik back in 1963. Subsequently, he collaborated closely with Alexey Chervonenkis on the VC theory, which offers a statistical perspective on the learning process and significantly influenced the advancement of SVM. If you're interested in a comprehensive history of SVMs, you can find detailed information here.

In real-world applications, SVMs have demonstrated successful usage in three main domains: text categorization, image recognition, and bioinformatics. Notable examples include classifying news stories, recognizing handwritten digits, and analyzing cancer tissue samples.

The first chapter of this book will explore fundamental concepts such as vectors, linear separability, and hyperplanes. These building blocks are essential for grasping the workings of SVMs. Moving on to Chapter 2, instead of directly delving into SVMs, we will study the Perceptron, a simple algorithm. This chapter offers valuable insights into why SVMs excel at data classification, so it should not be skipped.

In Chapter 3, we will gradually construct the SVM optimization problem, while Chapter 4 will delve into solving this problem, first mathematically and then programmatically. Chapter 5 will introduce the Soft-margin SVM, a significant improvement over the original formulation.

In Chapter 6, we'll introduce kernels and explain the "kernel trick." This leads to the kernelized SVM, which is widely used today. Chapter 7 will focus on SMO, an algorithm specifically designed to efficiently solve the SVM optimization problem. Chapter 8 will showcase how SVMs can be applied to classify multiple classes.

Throughout each chapter, you'll find code samples and figures to aid in better understanding the concepts. Although this book cannot cover every topic, it aims to present the most important aspects of SVMs. The conclusion will provide pointers on further learning about SVMs.

Now, let's embark on this exciting journey of understanding Support Vector Machines.

Chapter 2

Prerequisites

In this chapter, we will cover essential fundamentals to enhance your comprehension of SVMs. We'll start by exploring the concept of vectors and examining their important characteristics. Afterward, we'll delve into the notion of linear separability in data and introduce a critical element known as the **hyperplane**.

2.1 Vectors

In the context of Support Vector Machine, the term "vector" is significant. To grasp SVMs and their application, it's essential to have a foundational understanding of vectors and their fundamentals.

2.1.1 What is a Vector?

A vector is a mathematical entity that can be visualized as an arrow, as demonstrated in the below figure.

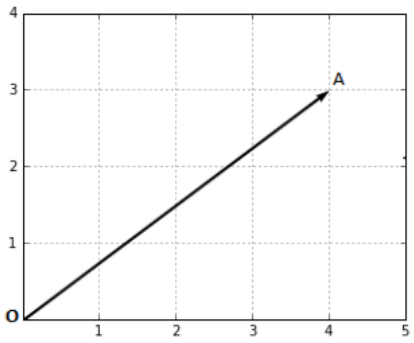

Figure 2.1: Representation of a Vector

During calculations, we represent a vector by using the coordinates of its endpoint. In the above figure, point A has the coordinates (4, 3). Consequently, we can express it as follows:

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

Alternatively, we can give another name to the vector, for instance, **a**.

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

Although it might seem that a vector is solely defined by its coordinates, there's more to it. For instance, if I were to give you a sheet of paper with only a horizontal line and ask you to trace the same vector as shown in the above figure, you would still be able to do it. This highlights that vectors have additional properties beyond their coordinate representation.

To fully describe a vector, you only require two essential details:

- The length of the vector.
- The angle between the vector and the horizontal line.

With these two pieces of information, we arrive at a concise definition of a **vector**: It is an entity characterized by both magnitude and direction.

Now, let's delve deeper into each of these components for a better understanding.

The Magnitude of a Vector

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

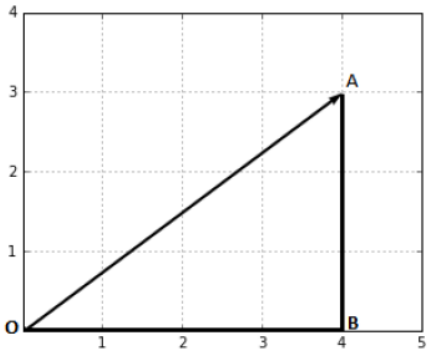

Figure 2.2: Calculating Norm of a Vector

In the above figure, we can calculate the norm $\|\mathbf{OA}\|$ of vector $\overrightarrow{OA}$ using the Pythagorean Theorem:

$$OA^2 = OB^2 + OA^2$$
$$OA^2 = 4^2 + 3^2$$
$$OA^2 = 25$$
$$OA = \sqrt{25}$$
$$\|\mathbf{OA}\| = OA = 5$$

In general, for a n-dimensional vector $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, we calculate the norm using the **Euclidean norm** formula: $\|\mathbf{x}\| = \sqrt{{x_1}^2 + {x_2}^2 + \cdots + {x_n}^2}$

In Python, we calculate the norm by using the **norm** function from the **numpy** module.

Listing 2.1: Calculating norm of a vector

```
1 import numpy as np
2
3 x = [3, 4]
4 np.linalg.norm(x)       # 5.0
```

The Direction of a Vector

The direction represents the second component of a vector. Simply put, it is a distinct vector whose coordinates are calculated by dividing the initial coordinates of our vector by its norm.

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

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

In Python, it can be computed using the below code.

Listing 2.2: Calculating direction of a vector

```
import numpy as np

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

The concept of direction in vectors is rooted in geometry. In the below figure, we observe a vector **u** and its angles concerning both the horizontal and vertical axes. One angle, labeled as theta (θ), signifies the angle between the vector and the horizontal axis, while another angle, labeled as alpha (α), represents the angle between the vector and the vertical axis.

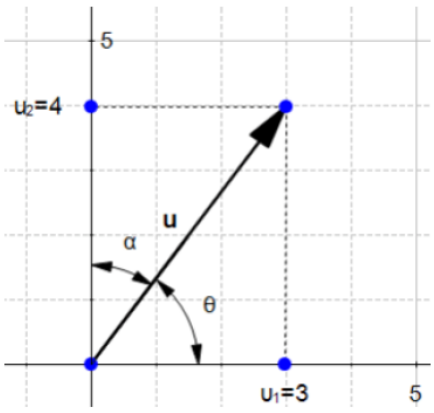

Figure 2.3: Vector and its angles with respect to the axes

Elementary geometry will show that $\cos\theta = \frac{u_1}{\|u\|}$, and $\cos\alpha = \frac{u_2}{\|u\|}$. This means, that $\mathbf{w}$ can also be written as:

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

The coordinates of the vector $\mathbf{w}$ are determined by cosines. Consequently, if the angle between the vector $\mathbf{u}$ and an axis alters, meaning a change in the direction of vector $\mathbf{u}$, it causes a change in the direction of the vector $\mathbf{w}$. This is the reason why we refer to vector $\mathbf{w}$ as the direction of the vector $\mathbf{u}$. We can use the below code snippet to calculate the vector $\mathbf{w}$.

Listing 2.3: Example 1 - Calculating direction of a vector

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

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

An interesting observation is that when two vectors share the same direction, they will also have the same direction vector. The next listing makes it clear with an example.

Listing 2.4: Example 2 - Calculating direction of a vector

```
u_1 = np.array([3, 4])
u_2 = np.array([30, 40])    # this is 10x scaled version of u_1

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

In addition, the **norm or magnitude of a direction vector is constantly equal to 1**. We can verify this with $\mathbf{w} = (0.6, 0.8)$ as an example.

Listing 2.5: Calculating norm of a vector

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

It's logical since the main purpose of this vector is to indicate the direction of other vectors. By having a norm of 1, it remains as simple as possible. Consequently, a direction vector like $\mathbf{w}$ is commonly known as a **unit vector**.

The Dimensions of a Vector

Note that the sequence in which the numbers are recorded holds significance. Therefore, we define a n-dimensional vector as a collection of n real numbers presented in the form of a tuple.

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 $\mathbf{u} \in \mathbb{R}^3$.

2.1.2 The Dot Product

The **dot product** is a mathematical operation involving two vectors that produce a numerical value. This numerical value is often referred to as a **scalar**, leading to the dot product being called a **scalar product**.

The dot product might appear challenging at first since its origin might seem unclear. However, it is crucial to understand that the dot product operates on two vectors, and its outcome provides valuable insights into the relationship between those vectors. There are two main perspectives to consider when thinking about the dot product: the geometric interpretation and the algebraic interpretation.

Geometric Definition

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

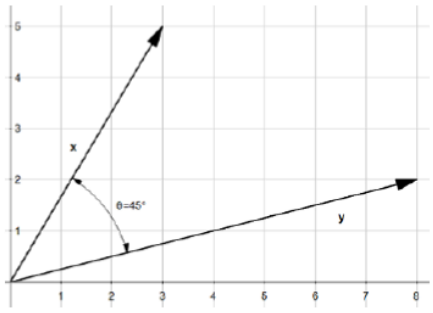

Figure 2.4: Geometric Definition of Dot Product

If we have two vectors $\mathbf{x}$ and $\mathbf{y}$ with angle θ between them, their dot product is defined as:

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

From the definition, it's clear that the dot product is dependent on the angle θ. Specifically:

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

These will come in handy when we discuss the Perceptron Algorithm later on.

We can use the following code to calculate the dot product of two vectors.

Listing 2.6: Geometric Dot Product of Two Vectors

```
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))
```

As an example:

Listing 2.7: Example - Geometric Dot Product of Two Vectors

```
theta = 45
x = [3, 5]
y = [8, 2]
print(geometric_dot_product(x, y, theta)) # 34.0
```

Algebraic Definition

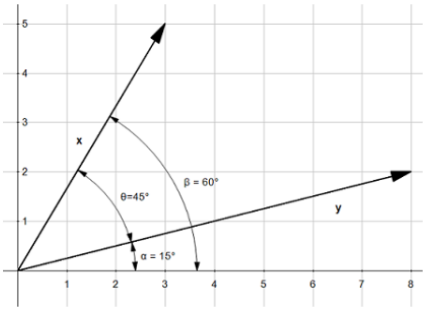

Figure 2.5: Algebraic Definition of Dot Product

From the above figure, we have $\theta = \beta - \alpha$. Further:

$$\begin{aligned}
\cos(\theta) &= \cos(\beta - \alpha) \\
\cos(\theta) &= \cos(\beta)\cos(\alpha) + \sin(\beta)\sin(\alpha) \\
\cos(\theta) &= \frac{x_1}{\|x\|}\frac{y_1}{\|y\|} + \frac{x_2}{\|x\|}\frac{y_2}{\|y\|} \\
\cos(\theta) &= \frac{x_1 y_1 + x_2 y_2}{\|x\|\|y\|} \\
\implies \|x\|\|y\|\cos(\theta) &= x_1 y_1 + x_2 y_2
\end{aligned}$$

From an earlier definition, we know that:

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

This means, we can write dot product as:

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 = \sum_{i=1}^{2} (x_i y_i)$$

In general, for n-domensional vectors, we can write as:

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

This gives us the **algebraic definition of dot product**. This definition has the advantage that it doesn't require the knowledge of the angle θ between the vectors. Programmatically, we see:

Listing 2.8: Algebraic Dot Product of Two Vectors

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

Listing 2.9: Example - Algebraic Dot Product of Two Vectors

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

We can alternatively use the **dot** method of **numpy** library.

Listing 2.10: Example - Algebraic Dot Product of Two Vectors using numpy

```
import numpy as np

x = np.array([3, 5])
y = np.array([8, 2])
print(np.dot(x, y))     # 34
```

Next, we visit linear separability which is a crucial aspect in understanding SVMs.

2.1.3 Linear Separability

We use an example to introduce the concept. Put yourself in the shoes of a wine producer. Your wine business offers two types of wine produced from different batches:

- A high-end wine priced at $145 per bottle
- A common wine priced at $8 per bottle

Lately, you have been facing complaints from customers who purchased the expensive wine, claiming that they received the cheaper one instead. This has caused significant damage to your company's reputation, leading to a decline in wine orders.

To address this issue, you decide to find a way to distinguish between the two wines. You decide to use the alcohol-by-volume metric to classify the wines. You recall that one of the wines has a higher alcohol concentration than the other. In an attempt to differentiate them, you open a few bottles from each batch, measure the alcohol content, and create a plot to visualize the differences. It looks something like this.

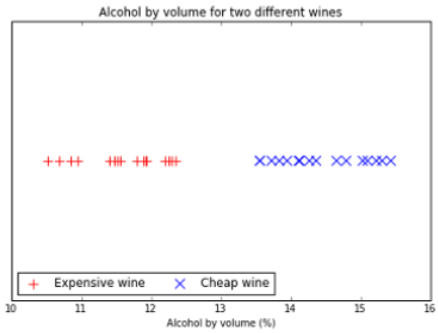

Figure 2.6: Alcohol by Volume Classification

In the above figure, it is evident that the expensive wine has a lower alcohol content compared to the cheaper one. This distinction creates two separate groups of data points, making it **linearly separable**. To address the issue

of mislabeling expensive bottles, you implement an automated alcohol concentration measurement before filling the high-end bottles. If the alcohol content exceeds 13 percent, production is halted for inspection by an employee. This improvement significantly reduces complaints, leading to a flourishing business once again.

While this example portrays an easily distinguishable scenario, real-world data is rarely that simple. The next figure shows an actual plot of wine data, where the data points are **not** linearly separable. Despite the common non-linear separability of data, it is crucial to grasp the concept of linear separability well. In most cases, we begin with the linearly separable case because it is simpler, then extend our understanding to non-separable cases.

In real-life problems, we often deal with more than one dimension, unlike the one-dimensional example. Real-world problems are complex, with some having thousands of dimensions, making them more abstract to work with. However, their abstractness doesn't necessarily imply increased complexity. Throughout this book, most examples will be two-dimensional, allowing for easy visualization and basic geometric analysis, facilitating a solid understanding of Support Vector Machines (SVMs) fundamentals.

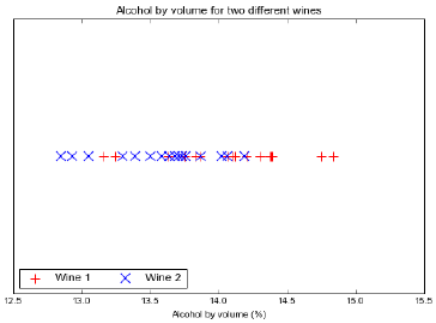

Figure 2.7: Real World Example of Alcohol by Volume Classification

In our example shown in Figure 6, we are dealing with only one dimension, where each data point is represented by a single number. However, when we work with data in higher dimensions, we use vectors to represent each data point. As the number of dimensions increases, the method of separating the data also changes.

In Figure 6, where we have just one dimension, we can separate the data using a single point. But as we move into two dimensions, we require a line (a set of points) to separate the data, as illustrated in Figure 8. Similarly, in three dimensions, we need a plane (which is also a set of points) to achieve separation, as shown in Figure 9.

In summary, data is considered linearly separable under the following conditions:

- In one dimension, a single point can separate the data (Figure 6).
- In two dimensions, a line is needed to separate the data (Figure 8).
- In three dimensions, a plane is required to separate the data (Figure 9).

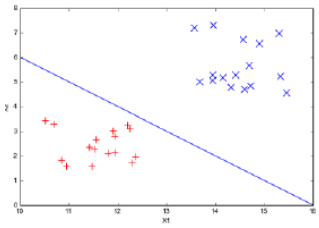

Figure 2.8: Data Seperated by Line

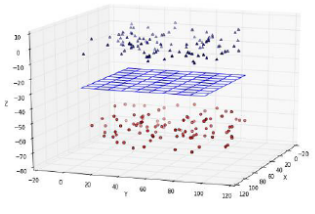

Figure 2.9: Data Seperated by Plane

Likewise, in cases where data is non-linearly separable, we encounter situations where we cannot find a single point, line, or plane to effectively separate the data. Figure 10 and Figure 11 provide examples of such non-linearly separable data in two and three dimensions, respectively.

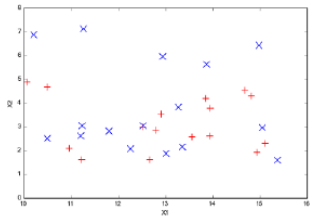

Figure 2.10: Non-Linearly seperable data in 2D

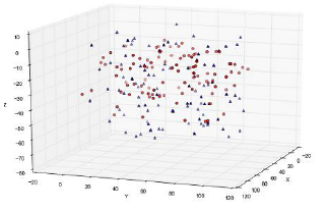

Figure 2.11: Non-Linearly seperable data in 3D

Next, we look at Hyperplanes.

2.1.4 Hyperplanes

To separate data in spaces with more than three dimensions, we use what is known as a **hyperplane**.

What is a hyperplane?

In geometry, a **hyperplane** is a *subspace with one dimension less than the ambient space it exists in.* While this definition is accurate, it might not be immediately intuitive. To better understand a hyperplane, let's start by looking at what a line is.

In school mathematics, you likely learned that a line has an equation of the form $y = ax + b$, where a represents the slope, and b is the y-intercept. This equation holds true for several values of x, and the set of solutions forms a line.

Confusion often arises because when studying this function in a calculus course, you deal with a function with only one variable. However, it's important to note that the linear equation has two variables, which we can name as we please. For instance, we can rename x as x_1 and y as x_2, and the equation becomes $x_2 = a * x_1 + b$, which is equivalent to $x_2 - a * x_1 - b = 0$.

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

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

What's great about this final equation is that it involves vectors. Although we derived it using two-dimensional vectors, it can be applied to vectors of any dimension. This equation represents a hyperplane, which means it allows us to handle data in spaces with any number of dimensions effectively.

With this equation, we gain another perspective on what a hyperplane is: it is the set of points that satisfy the equation $\mathbf{w} \cdot \mathbf{x} + b = 0$. In essence, a hyperplane is a set of points.

The reason we can derive the hyperplane equation from the equation of a line is that a line is, in fact, a type of hyperplane. You can convince yourself of this by revisiting the definition of a hyperplane. When you do so, you'll observe that a line can be seen as a two-dimensional space enclosed within a plane, which exists in three dimensions. This realization applies to other shapes as well, like points and planes, which are also considered hyperplanes.

Understanding the hyperplane equation

It's fascinating how we derived the equation of a hyperplane from the equation of a line. But what's even more intriguing is reversing the process, as it allows us to better understand the connection between the two concepts.

Given two vectors, $\mathbf{x} = (x, y)$, and $\mathbf{w} = (w_0, w_1)$ we can define the hyperplane as:

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

This is the same as:

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

We evaluate for y:

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

If we define $a = -\frac{w_0}{w_1}$ and $c = -\frac{b}{w_1}$, we get $y = ax + c$.

Let's take a closer look at the bias c in the line equation and how it relates to the bias b in the hyperplane equation. We observe that the two biases are equal when the coefficient w_1 in the line equation is -1. So, it's essential to understand that b may not always represent the intersection with the vertical axis when we plot a hyperplane (you'll notice this in our upcoming example).

Additionally, when both w_0 and w_1 have the same sign, the slope a will be negative. This means that the hyperplane will have a downward slant.

Calssifying Data using hyperplane

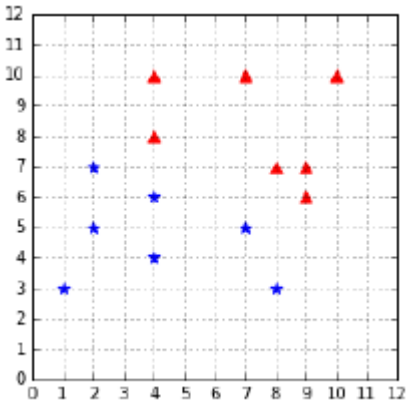

Figure 2.12: A Linearly Seperable Dataset

Let us see how we can use a hyperplane to perform binary classification.

For example, with the vector $\mathbf{w} = (0.4, 1.0)$ and $b = -9$, we get the hyperplane as follows.

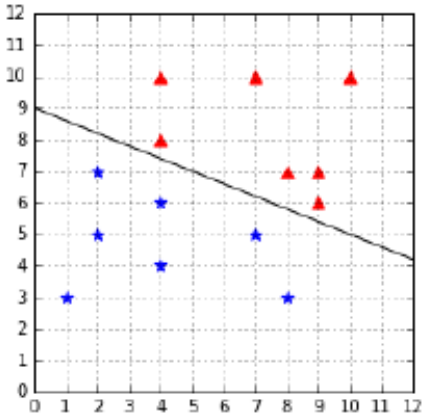

Figure 2.13: Hyperplane seperating data

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 the above figure).

Next we define a *hypothesis* function h:

$$h(\mathbf{x_i}) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \mathbf{x_i} + b \geq 0 \\ +1 & \text{if } \mathbf{w} \cdot \mathbf{x_i} + b < 0 \end{cases} \tag{2.1}$$

This is equivalent to: $h(\mathbf{x_i}) = sign(\mathbf{w} \cdot \mathbf{x_i} + b)$

The way it works is that we utilize the position of each data point (represented as $\mathbf{x}$) about the hyperplane to make predictions about its label. When we have a hyperplane, data points lying on one side of it will be assigned one label, while those on the other side will receive a different label. This way, we can effectively classify and distinguish between different groups or categories based on their positions in relation to the hyperplane.

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

Similarly, for $\mathbf{x} = (1, 3)$, $\mathbf{x}$ is below the hyperplane. When we compute, we get $\mathbf{w} \cdot \mathbf{x} + b = 0.4 \times 1 + 1 \times 3 - 9 = -5.6$, which is negastive, and hence $h(\mathbf{x}) = -1$.

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

We can further simplify the expression of h by eliminating the constant b. First, we add a component $x_0 = 1$ to the vector $\mathbf{x_i} = (x_1, x_2, \ldots, x_n)$ to get $\hat{\mathbf{x}}_\mathbf{i} = (x_0, x_1, \ldots, x_n)$. Similarly, we add component $w_0 = b$ to the vector $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ to get $\hat{\mathbf{w}} = (w_0, w_1, \ldots, w_n)$. These new vectors are called **argumented vecotrs** and will be extensively used going further.

In terms of the augmented vectors, the hypothesis function becomes:

$$\hat{\mathbf{x}}_\mathbf{i} = sign(\hat{\mathbf{w}} \cdot \hat{\mathbf{x}}_\mathbf{i})$$

Imagine we have a hyperplane that perfectly separates the data points in a dataset, just like the one shown in Figure 13. By using a hypothesis function that takes advantage of this hyperplane, we can accurately predict the label of each data point.

Now, the big question is: How do we actually find such a hyperplane?

Finding the hyperplane

Let's revisit the equation of the hyperplane, which is written as $\mathbf{w} \cdot \mathbf{x} = 0$ in an augmented form. It's crucial to understand that the key factor that influences the shape of the hyperplane is the vector $\mathbf{w}$. To illustrate this point, let's go back to the two-dimensional case when a hyperplane is simply a line. By creating augmented three-dimensional vectors, we have $\mathbf{x} = (x_0, x_1, x_2)$ and $\mathbf{w} = (b, a, -1)$. In this vector $\mathbf{w}$, you'll find both a and b which are the two main components that determine the appearance of the line.

Changing the value of $\mathbf{w}$ allows us to obtain different hyperplanes (lines), as demonstrated in the next figure. So, essentially, it's the vector $\mathbf{w}$ that defines the orientation and position of the hyperplane, just like it does for a line in two-dimensional space.

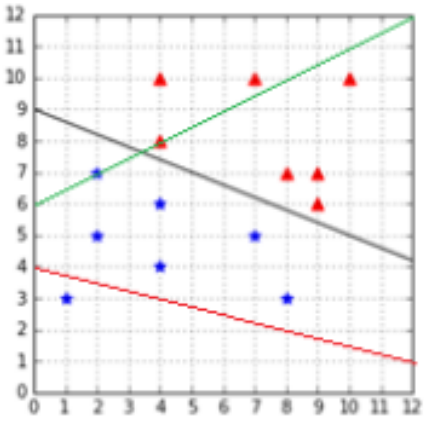

Figure 2.14: Different Hyperplanes for different values of **w**

2.1.5 Summary

Once we covered vectors and the concept of linear separability, we delved into understanding what a hyperplane is and how it becomes instrumental in classifying data. Our main objective is to find a hyperplane that effectively separates the data, and in the process, we realized that this is equivalent to finding a vector **w**.

Now, let's explore the different methods that learning algorithms employ to discover such a separating hyperplane. Before we dive into how Support Vector Machines (SVMs) achieve this, we'll start by examining one of the most basic learning models: the Perceptron.

Chapter 3

The Perceptron

The Perceptron is an algorithm created by Frank Rosenblatt in 1957, predating the first Support Vector Machine (SVM) by a few years. It gained significant popularity because it serves as the foundation for a basic type of neural network called the multilayer perceptron. The main objective of the Perceptron is to discover a hyperplane that can effectively separate a linearly separable dataset. Once this hyperplane is identified, it can be utilized to perform binary classification tasks. In simpler terms, the Perceptron is a key tool for solving classification problems where data points can be separated by a straight line or a flat surface.

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

$$h(\mathbf{x_i}) = sign(\mathbf{w} \cdot \mathbf{x_i})$$

3.1 The Perceptron Learning Algorithm

Let's break down the concept of the Perceptron Learning Algorithm (PLA) in simpler terms.

Imagine we have a training set $\mathcal{D}$ containing m examples, where each example consists of n-dimensional data points $\mathbf{x_i}$ and their corresponding labels y_i. The goal of the PLA is to find a hypothesis function h that correctly predicts the label y_i for each data point $\mathbf{x_i}$.

In the case of the Perceptron, the hypothesis function $h(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x})$ is represented by the equation of a hyperplane. This hyperplane can be thought of as a flat surface in an n-dimensional space. So, the set of all possible hypothesis functions $\mathcal{H}$ is the set of $(n-1)$-dimensional hyperplanes. The reason it's $(n-1)$ dimensional is that a hyperplane has one less dimension than the overall space it exists in.

Now, the key thing to understand is that the only unknown value in this equation is represented by the vector $\mathbf{w}$. The main objective of the algorithm is to find an appropriate value for $\mathbf{w}$. Once you find the right value for $\mathbf{w}$, you essentially determine a specific hyperplane. However, there are infinitely many hyperplanes since you can assign any valid value to $\mathbf{w}$. And this, in turn, results in an infinite number of hypothesis functions.

To put it more formally, the task of the PLA is to determine the appropriate value for $\mathbf{w}$ that defines the hyperplane separating the data, allowing it to predict the correct labels for all the given examples.

Given a training set $\mathcal{D} = \{ (\mathbf{x_i}, y_i); \mathbf{x_i} \in \mathbb{R}^n, y_i \in -1, 1 \}_{i=1}^m$, and a set $\mathcal{H}$ of hypothesis functions. Find $h \in \mathcal{H}$ such that $h(\mathbf{x_i} = y_i) \forall \mathbf{x_i}$.

This is same as, given a training set $\mathcal{D} = \{ (\mathbf{x_i}, y_i); \mathbf{x_i} \in \mathbb{R}^n, y_i \in -1, 1 \}_{i=1}^m$, and a set $\mathcal{H}$ of hypothesis functions, find $\mathbf{w} = (w_0, w_1, \ldots, w_n)$ such that $sign(\mathbf{w} \cdot \mathbf{x_i}) = y_i \forall \mathbf{x_i}$.

The Perceptron Learning Algorithm (PLA) is a straightforward and easy-to-understand algorithm, and its steps can be summarized as follows:

1. We start by initializing a random hyperplane, represented by a vector $\mathbf{w}$, and use it to classify the data points.
2. Next, we identify a misclassified data point, and to improve the classification, we update the hyperplane by adjusting the value of $\mathbf{w}$. This step is known as the **update rule**.
3. Once we have the updated hyperplane, we use it to classify the entire dataset again.
4. We repeat steps 2 and 3 until there are no more misclassified data points.

By the end of this process, we will have a hyperplane that effectively separates the data points into distinct categories. The algorithm's code listing is shown below.

Listing 3.1: Perceptron Learning Algorithm

```
import numpy as np

def perceptron_learning_algorithm(X, y):
    w = np.random.rand(42) # 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
```

Now, let's take a closer look at the code. The **perceptron_learning_algorithm** makes use of various functions. The **hypothesis** function, denoted as $h\mathbf{x}$ in Python code, is responsible for predicting the label y_i for a given example $\mathbf{x_i}$ when using the hyperplane defined by $\mathbf{w}$ for classification. We've seen this function before. On the other hand, the **predict** function applies the hypothesis to each example and identifies those that are misclassified.

Listing 3.2: PLA - hypothesis() and predict()

```
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
```

After making predictions with the **predict** function, we can identify the misclassified examples, and then we utilize the **pick_one_from** function to randomly choose one of those misclassified examples.

Listing 3.3: PLA - pick_one_from()

```
# 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]
```

Next, we delve into the core of the algorithm: **the update rule**. At this point, simply remember that it modifies the value of **w**. We will provide a more detailed explanation later on. We once again utilize the **predict** function, but this time, we supply it with the updated **w**. This enables us to assess whether all data points are correctly classified or if we must repeat the process until they are.

The next piece of code showcases how the **perceptron_learning_algorithm** function can be applied to a simple data set. It's worth noting that for **w** and **x** vectors to be compatible, we convert each **x** vector into an augmented vector before passing it to the function.

Listing 3.4: Example - Perceptron Learning Algorithm

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

np.random.seed(42)

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]
```

3.1.1 Understanding the update rule

Why do we choose this specific update rule? Remember that we randomly selected a misclassified example. Now, our goal is to adjust the Perceptron so

that it classifies this example correctly. To achieve this, we decide to update the vector $\mathbf{w}$. The concept behind this update is straightforward. Since the dot product between $\mathbf{w}$ and $\mathbf{x}$ has the wrong sign, we need to change the angle between them to correct it. We apply the below rules:

- When the predicted label is $+1$, the angle between $\mathbf{w}$ and $\mathbf{x}$ is less than $90°$ degrees, so we aim to increase it.
- Conversely, when the predicted label is -1, the angle between $\mathbf{w}$ and $\mathbf{x}$ is greater than $90°$ degrees, so we aim to decrease it.

Consider two vectors $\mathbf{w}$ and $\mathbf{x}$ having an angle θ between them.

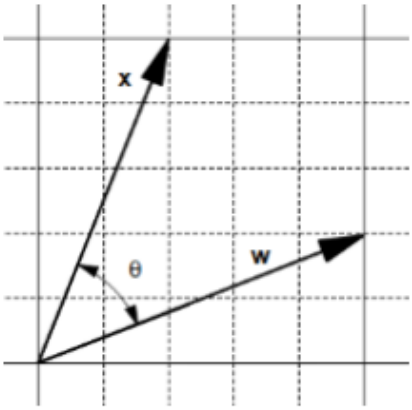

Figure 3.1: Two Vectors

If we add these two vectors, we get a new vector $\mathbf{w} + \mathbf{x}$, and the angle β between $\mathbf{x}$ and $\mathbf{w} + \mathbf{x}$ is smaller than θ.

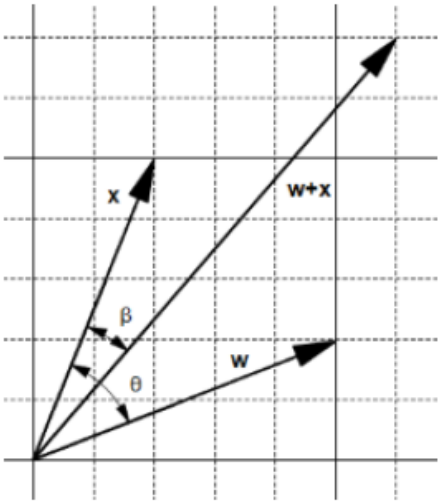

Figure 3.2: Adding Two Vectors

If we subtract these vectors, a new vector **w - x** is created and the angle β between **x** and **w - x** is larger than θ.

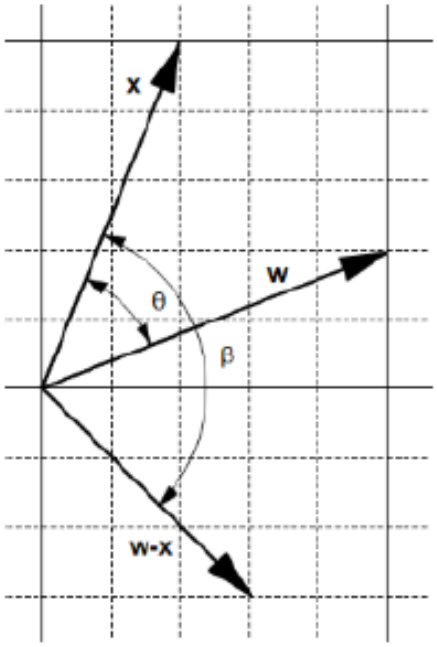

Figure 3.3: Subtracting Two Vectors

We can now use these two observations to adjust the angle:

- When the predicted label is $+1$, the angle between $\mathbf{w}$ and $\mathbf{x}$ is less than $90°$ degrees, so we aim to increase it.
- Conversely, when the predicted label is -1, the angle between $\mathbf{w}$ and $\mathbf{x}$ is greater than $90°$ degrees, so we aim to decrease it.

Since we only make these adjustments for misclassified examples, the predicted label and the expected label have opposite values. To simplify, we can state the following:

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

When this is translated into Python code, it corresponds following code snippets for the update rule.

Listing 3.5: Update Rule

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

Listing 3.6: Update Rule Optimized

```
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.

Listing 3.7: Example 1 - Verifying Update Rule

```
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.
```

Please be aware that the update rule may not alter the hypothesis sign for the first instance. It might be necessary to apply the update rule multiple times, as demonstrated in the next code listing. However, this is not an issue since we iterate through misclassified examples, persistently using the update rule until the example is classified correctly. The crucial aspect is that with each application of the update rule, we modify the angle's value in the correct direction (either increasing or decreasing it).

Listing 3.8: Example 2 - Verifying Update Rule

```
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.
```

Sometimes, when we update the value of **w** for a specific example, it can alter the hyperplane in a manner that causes another example **x*** which was previously classified correctly, to be misclassified. As a result, the hypothesis might become less effective at classifying after the update. This situation is depicted in the next figure, where the number of classified examples is shown for each iteration step.

To address this issue, one approach is to keep track of the value of **w** before making the update and only accept the updated value if it reduces the number of misclassified examples. This modification of the Perceptron Learning Algorithm (PLA) is referred to as the **Pocket algorithm**, as it involves keeping the value of **w** in our pocket for comparison purposes.

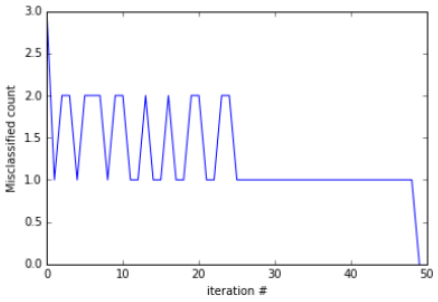

Figure 3.4: PLA rule updates and Oscillations

3.1.2 Convergence of the algorithm

We stated that we continuously update the vector $\mathbf{w}$ using a specific rule until no misclassified points remain. However, a valid concern arises: how can we be certain that this process will ever lead to a solution? Fortunately, mathematicians have extensively studied this matter, and we can find reassurance in the Perceptron convergence theorem. According to this theorem, if the sets $\mathcal{P}$ and $\mathcal{N}$, which consist of positive and negative examples, respectively, can be separated by a straight line, then the vector $\mathbf{w}$ will undergo a finite number of updates. This result was first proven by Novikoff in 1963 (Rojas, 1996).

3.1.3 Limitations of the PLA

It's important to grasp that the PLA algorithm behaves unpredictably due to its random weight initialization and the random selection of misclassified examples. This can lead to different hyperplanes being obtained each time the algorithm is run. The next figure illustrates this phenomenon, showing four different hyperplanes discovered by PLA when applied to the same dataset in four separate runs.

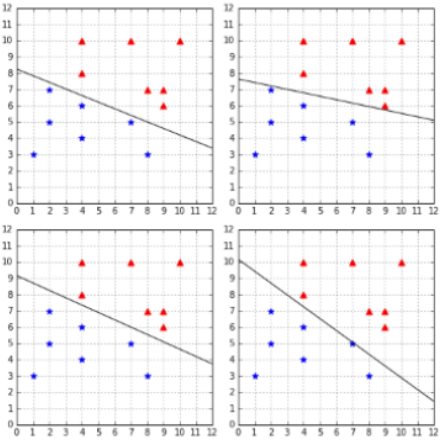

Figure 3.5: PLA identifying different Hyperplanes

Initially, this may not appear to be an issue. After all, the four hyperplanes flawlessly classify the data, so one might assume they are equally effective, right? However, in machine learning, particularly with the PLA algorithm, our objective isn't merely to achieve perfect classification on the current dataset. Instead, we aim to discover a method that will accurately classify new, unseen data we encounter in the future.

Let's clarify some terms. To train a model, we select a **subset** of existing data and refer to it as the **training set**. The model is trained, resulting in a **hypothesis** (in this case, a hyperplane). We can assess how well this hypothesis performs on the training set, which is called the **in-sample error** or training error. Once we are satisfied with the hypothesis, we intend to apply it to unseen data, known as the **test set**, to determine if it has truly learned meaningful patterns. We assess its performance on the test set, which is known as the **out-of-sample** error or generalization error.

Our ultimate goal is to minimize the out-of-sample error.

In the context of the PLA, all the hypotheses shown in the previous figure achieve perfect classification of the given data, resulting in zero in-sample error. However, our primary concern lies in their out-of-sample error. To evaluate this, we can utilize a test set, like the one shown in the next figure, to assess their performance on unseen data.

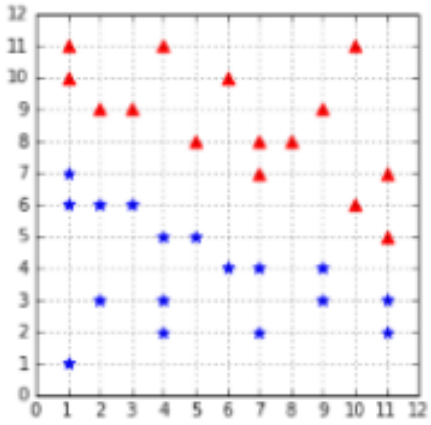

Figure 3.6: A Test Dataset

As you can observe in the following diagram, the two hypotheses on the right, despite achieving perfect classification on the training dataset, are making errors when applied to the test dataset.

Now we have a better understanding of why this is problematic. When using the Perceptron with a linearly separable dataset, we can guarantee to find a hypothesis with zero error on the training data. However, there is no assurance of how well it will perform on unseen data, as **generalization** to new data is uncertain (if an algorithm generalizes well, its error on new data should be close to its error on the training data). **How can we select a hyperplane that generalizes effectively**? In the upcoming chapter, we will explore how Support Vector Machines (SVMs) address this issue as one of their primary objectives.

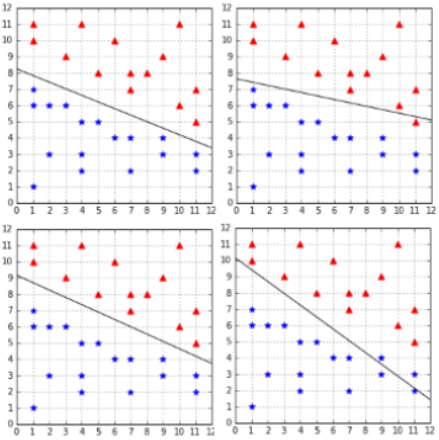

Figure 3.7: Test Dataset Performance

3.2 Summary

In this chapter, we gained an understanding of the Perceptron, its structure, and how it operates. We delved into the intricacies of the Perceptron Learning Algorithm, exploring the reasoning behind its update rule. We discovered that the PLA is guaranteed to converge, but not all hypotheses are equally effective; some will generalize better than others. Surprisingly, the Perceptron cannot determine which hypothesis will yield the smallest out-of-sample error, so it simply selects a hypothesis with the lowest in-sample error randomly.

Chapter 4

The SVM Optimization Problem

The Perceptron offers several advantages due to its simplicity, easy algorithm implementation, and theoretical guarantee of finding a hyperplane to separate the data. However, its main drawback lies in its inconsistency, as it may not find the same hyperplane every time. This inconsistency matters because not all separating hyperplanes are equally effective. If the Perceptron produces a hyperplane very close to one class of data points, it is reasonable to expect poor generalization when confronted with new data.

On the other hand, Support Vector Machines (SVMs) overcome this issue. Instead of merely seeking a **hyperplane**, SVMs aim to find the **optimal hyperplane**. This optimal hyperplane is the one that best separates the data, and it mitigates the problem of inconsistent results encountered in the Perceptron.

4.1 COmparing Hyperplanes

Since we can't rely on our intuition or feelings to choose the best hyperplane, we need a measurable way to compare different hyperplanes and determine which one is the most effective.

In this section, we'll explore how we can compare two hyperplanes. Essentially, we want to find a method to calculate a number that tells us which hyperplane separates the data the best. We'll examine some methods that initially appear promising, but we'll also identify their shortcomings and figure out how to overcome them. Let's start by attempting a basic comparison of two hyperplanes using just their equations.

4.1.1 Using the equation of the hyperplane

When we have an example and a hyperplane, our goal is to understand the relationship between the example and the hyperplane. We already know that if a value of $\mathbf{x}$ satisfies the equation of a line, it means the point lies on that line. The same principle applies to a hyperplane: if we have a data point $\mathbf{x}$ and a hyperplane defined by a vector $\mathbf{w}$ and bias b, we can determine if $\mathbf{x}$ is on the hyperplane by checking if $\mathbf{w} \cdot \mathbf{x} + b = 0$.

However, what happens when the point is not on the hyperplane?

An example would make it clear. In the next figure, the line is defined by $\mathbf{w} = (-0.4, -1)$ and $b = 9$.

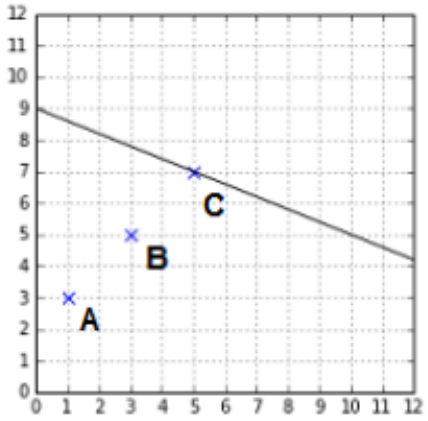

Figure 4.1: Distances from Line

Using the equation of 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 $\mathbf{w} \cdot \mathbf{a} + c = 0$.

When the point is not on the hyperplane, the result we get from the equation is not zero. If we use a point that is far from the hyperplane, the result will be larger than if we use a point closer to the hyperplane.

Also, the sign of the number we get from the equation tells us whether the point is above or below the line. Positive numbers mean the point is above, while negative numbers mean it's below the line. This is evident from the next figure.

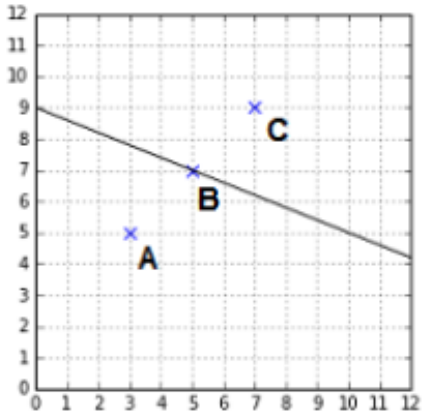

Figure 4.2: Positive and Negative Distances from Line

Using the same equation for the line as earlier, we get:

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

When we plug in the values of points into the equation, a positive result indicates that the point lies below the line. Conversely, a negative result means the point is above the line. However, it's important to note that "above" and "below" doesn't always refer to a visual representation, as shown in the next figure where it could be left or right. Nevertheless, the sign of the number obtained from the equation helps us determine if two points are on the same side. This aligns with the hypothesis function described in the previous chapter.

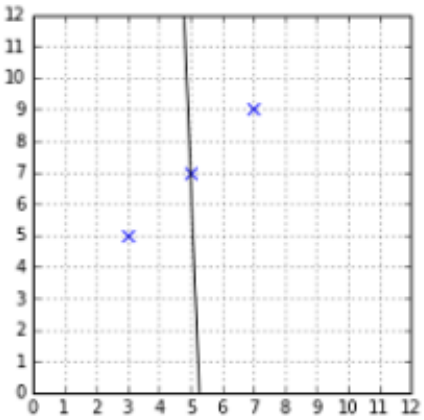

Figure 4.3: Different way of seperating points

We now have the initial contours of our solution to compare 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 dataset $\mathcal{D} = \{ (\mathbf{x_i}, y_i); \mathbf{x_i} \in \mathbb{R}^n, y_i \in -1, 1 \}_{i=1}^{m}$, we compute β for each training example, and define number $\mathcal{B}$ as the smallest β we encounter:

$$\mathcal{B} = \min_{1 \ldots m} \beta_i$$

If we have to decide between two hyperplanes, we'll choose the one with the largest value of $\mathcal{B}$. Further, if we have k hyperplanes, we will compute $\min_{1 \ldots k} \mathcal{B}_i$ and then select the hyperplane having this $\mathcal{B}_i$.

4.1.2 Problem with Negatives

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). Let us see why.

Remember that we always wish to take the β of the point being the closest to the hyperplane. Computing $\mathcal{B}$ 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 $\beta = -1$.

This can be fixed by considering the absolute value of β. We modify the algorithm accordingly.

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

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

4.1.3 Is the Hyperplane classifying correctly?

When we calculate the number $\mathcal{B}$, it helps us choose a hyperplane. But relying solely on this number might lead us to make a mistake. Let's take a look at the next figure as an example: the examples are currently **classified correctly**, and the computed value using the last formula is $\mathcal{B} = 2$.

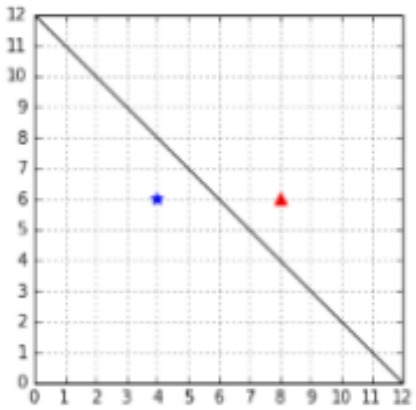

Figure 4.4: Correct Classification

The example in the next figure shows that the samples are **incorrectly classified** even though the value of $\mathcal{B}$ is still 2. They both look equally good, but we should pick the correct hyperplane as in the first figure.

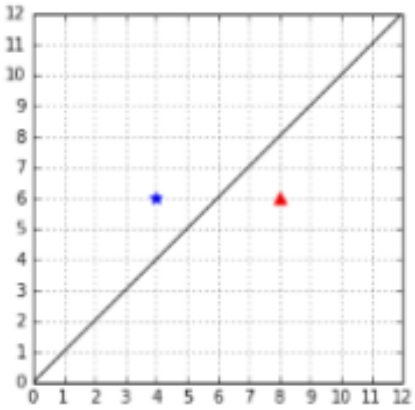

Figure 4.5: Incorrect Classification

How can we modify our formula to fulfill this condition?

Notice that we haven't considered one part of our training example, which is represented by the variable y. If we multiply β by the value of y, it will result in a change of its sign. Let's denote this new value as f:

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

The sign of f will always be:

- Positive, if the sample is correctly classified.
- Negative, if the sample is incorrectly classified.

We again modify the algorithm. Given a dataset $\mathcal{D}$, we compute:

$$\mathcal{F} = \min_{1...m} f_i$$
$$\mathcal{F} = \min_{1...m} y_i(\mathbf{w} \cdot \mathbf{x_i} + b)$$

When using this particular formula, when we compare two hyperplanes, we will choose the one with the largest value of $\mathcal{F}$. An additional advantage is that in specific scenarios, such as those depicted in the first two figures in this section,

we will always select the hyperplane that classifies correctly (as it will have a positive value), while the other hyperplane will have a negative value.

In the relevant literature, the number f has a specific term associated with it - it is referred to as the **functional margin** of an example. Similarly, the number $\mathcal{F}$ is known as the **functional margin** of the entire dataset $\mathcal{D}$. The code is shown next.

Listing 4.1: Functional Margins

```
# 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 for the first figure is $+2$, while for the next figure, it is -2. Because it has a bigger margin, we will select the first one.

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

4.1.4 Scale Invariance

It seems like we've come across an effective method to compare two hyperplanes this time. However, there is a significant issue with the functional margin — it lacks **scale invariance**.

Let's consider two sets of parameters: vector $\mathbf{w_1}$ and bias b_1, and their scaled counterparts $\mathbf{w_2}$ and bias b_2, which result from multiplying them by 10. We refer to this process as **rescaling**.

The vectors $\mathbf{w_1}$ and $\mathbf{w_2}$ represent the same hyperplane because they share the same unit vector. Since a hyperplane is a plane orthogonal to a vector $\mathbf{w}$, its length doesn't matter; only its direction matters. As we discussed earlier, the direction is determined by its unit vector. Furthermore, when we plot the

hyperplane on a graph, the intersection with the vertical axis will remain the same (intersection point $(0, \frac{b}{w_1})$), meaning the hyperplane remains unchanged even after rescaling the parameter b.

The problem, illustrated in the next code snippet, arises when we calculate the functional margin with $\mathbf{w_2}$ — it becomes ten times larger than with $\mathbf{w_1}$. This means that for any given hyperplane, we can always find another hyperplane that has a larger functional margin simply by rescaling $\mathbf{w}$ and b.

Listing 4.2: Example of Rescaling

```
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.

Startr by recalling that functional margin is: $f = y(\mathbf{w} \cdot \mathbf{x} + b)$

We normalize it and get a new number, γ:

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

And finally, given the dataset $\mathcal{D}$, we compute:

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

The advantage of γ is that it gives us the same number no matter how large 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 $\mathcal{M}$ is the geometric margin of the dataset. A sample Python code implementation is shown next.

Listing 4.3: Geometric Margin

```
# 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 see the behavior of the geometric margin in the next code and observe that it gives the same value for the original and scaled vectors.

Listing 4.4: Example - Geometric Margin

```
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
```

Next, we delve into the origins of geometric margin. One way is to derive it using simple geometry, which we do next. We will see that it measures the Distances between $\mathbf{x}$ and the hyperplane.

In the next figure, we see that point X' is the orthogonal projection of X into the hyperplane. We aim to find the distance d between X and X'.

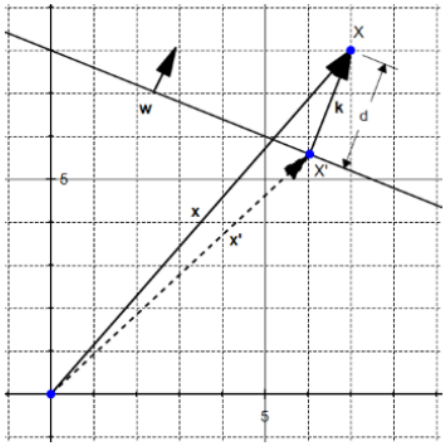

Figure 4.6: Visualizing Geometric Margin

We can see that the vectors $\mathbf{k}$ and $\mathbf{w}$ have the same direction. Thus, we can say that they share the same unit vector $\frac{\mathbf{w}}{\|w\|}$. Further, know that the norm of $\mathbf{k}$ is d, so we can define vector $\mathbf{k}$ as $\mathbf{k} = d\frac{\mathbf{w}}{\|w\|}$.

Further, from vector subtraction we see that $\mathbf{x}' = \mathbf{x} - \mathbf{k}$. Using the definition of $\mathbf{k}$, we get:

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

Since the point X' lies on the hyperplane, it means that $\mathbf{x}'$ satisfies the equation of the hyperplane. This gives us:

$$
\begin{aligned}
\mathbf{w} \cdot \mathbf{x}' + b &= 0 \\
\implies \mathbf{w} \cdot \left(\mathbf{x} - d\frac{\mathbf{w}}{\|w\|} \right) + b &= 0 \\
\implies \mathbf{w} \cdot \mathbf{x} - d\frac{\mathbf{w} \cdot \mathbf{w}}{\|w\|} + b &= 0 \\
\implies \mathbf{w} \cdot \mathbf{x} - d\frac{\|w\|^2}{\|w\|} + b &= 0 \\
\implies \mathbf{w} \cdot \mathbf{x} - d\|w\| + b &= 0 \\
\implies d &= \frac{\mathbf{w} \cdot \mathbf{x} + b}{\|w\|} \\
\implies d &= \frac{\mathbf{w}}{\|w\|} \cdot \mathbf{x} + \frac{b}{\|w\|}
\end{aligned}
$$

In the end, just like we've done previously, we multiply by y to guarantee the selection of a hyperplane that accurately classifies the data. This process yields the same geometric margin formula we encountered earlier.

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

Now that we have established the concept of the geometric margin, let's explore how it helps us compare two hyperplanes. Look at the two figures next. By examining the left figure, we can observe that the hyperplane is closer to the blue star examples compared to the red triangle examples, unlike the situation in the right figure. Consequently, we anticipate that its geometric margin will be smaller. To calculate the geometric margin for each hyperplane, refer to the code snippet below. It's abundantly clear that the second hyperplane defined by $\mathbf{w} = (-0.4, -1)$ and $b = 8.5$ has a larger geometric margin ($0.64 > 0.18$). Given these results, we would choose this hyperplane over the other one.

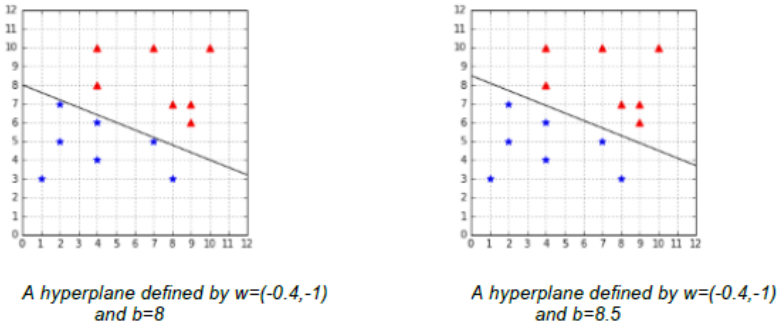

Figure 4.7: Comparing two hyperplanes

Listing 4.5: Comparing Geometric Margin

```
# 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 can compute the geometric margin for different hyperplanes by adjusting the values of $\mathbf{w}$ or b. However, trying random adjustments could be time-consuming and not very effective. Our goal is to find the **optimal hyperplane** that maximizes the margin **among all possible hyperplanes**, but there are infinitely many of them.

To find the optimal hyperplane, we need to determine the values of $\mathbf{w}$ and b that result in the largest geometric margin. Fortunately, mathematicians have created tools to solve such problems. The process involves dealing with an optimization problem. Before delving into the specifics of the optimization problem for Support Vector Machines (SVMs), let's quickly review what an optimization problem entails.

4.2 What is an optimization problem?

4.2.1 Unconstrained Optimization

The objective of an **optimization problem** is to find the best possible value for a variable x that either minimizes or maximizes a given function. In simpler terms, we want to determine the value of x that makes the function produce its minimum or maximum value. For example, the problem where we want to find the minimum value of the function $f(x) = x^2$ is written as:

$$\min_x f(x)$$

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 the next figure, the minimum of the function is zero at $x = 0$.

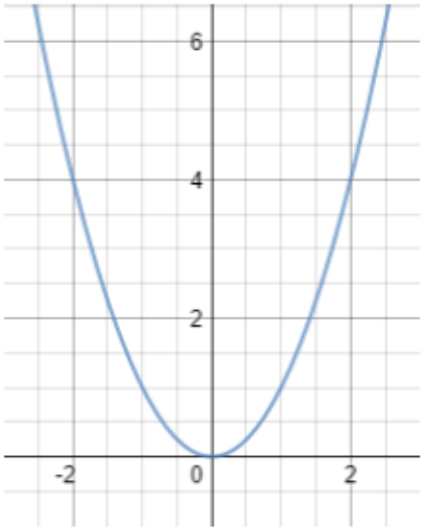

Figure 4.8: Unconstrained Optimization

4.2.2 Constrained Optimization

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. For instance, if we wish to know the minimum of f but restrict the value to a specific value, we can write:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & x = 2 \end{aligned}$$

In general, constraints are written by keeping zero on the right side of the equality so the problem can be rewritten as:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & x - 2 = 0 \end{aligned}$$

This can be visualized as:

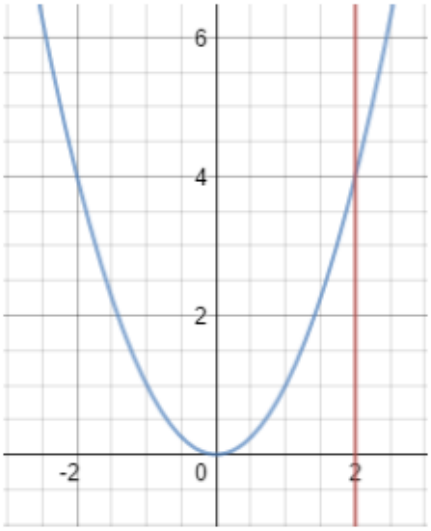

Figure 4.9: Constrained Optimization

Using this notation, we 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 meets the problem's constraints is referred to as the **feasible set** or feasible region. When addressing the optimization problem, the solution will be chosen from this feasible set. In the previous section, the feasible set contains just one value, making the problem trivial. However, when dealing with functions involving multiple variables, like $f(x, y) = x^2 + y^2$, it enables us to determine the values we are seeking to find a minimum or maximum.

As an example:

$$\begin{aligned} \min_{x,y} \quad & f(x,y) \\ \text{s.t.} \quad & x-2=0 \end{aligned}$$

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

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

$$\begin{aligned} \min_{x,y,z} \quad & f(x,y,z) \\ \text{s.t.} \quad & x-2=0 \\ \text{s.t.} \quad & y+8=0 \\ \text{s.t.} \quad & z+3=0 \end{aligned}$$

When we have several variables, we can switch to vector notation to improve readability. For the vector $\mathbf{x} = (x, y, z)^\top$, the function becomes $f(\mathbf{x}) = \mathbf{x_1}^2 + \mathbf{x_2} - \mathbf{x_3}^2$, and it can be written as:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{x_1}-2=0 \\ \text{s.t.} \quad & \mathbf{x_2}+8=0 \\ \text{s.t.} \quad & \mathbf{x_3}+3=0 \end{aligned}$$

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 example, take a look at the next optimization problem:

$$\begin{aligned} \min_{x} \quad & x^2 \\ \text{s.t.} \quad & x-2=0 \\ \text{s.t.} \quad & x-8=0 \end{aligned}$$

We may be tempted to think that $x = 2$ and $x = 8$ are solutions, but that is not the case. When $x = 2$, the constraint $x - 8 = 0$ is not satisfied. Similarly, when $x = 8$, the constraint $x - 2 = 0$ is not satisfied. Such problems can be called **infeasible**. We should remember that if we add too many constraints, the problem may become infeasible.

Inequality Constraints

We can also use inequalities as constraints. For example:

$$\begin{aligned} \min_{x,y} \quad & x^2 + y^2 \\ \text{s.t.} \quad & x - 2 \geq 0 \\ \text{s.t.} \quad & y \geq 0 \end{aligned}$$

And we can combine equality constraints and inequality constraints as:

$$\begin{aligned} \min_{x,y} \quad & x^2 + y^2 \\ \text{s.t.} \quad & x - 2 = 0 \\ \text{s.t.} \quad & y \geq 0 \end{aligned}$$

4.2.3 Solving an Optimization Problem

There are various methods available for solving different types of optimization problems. However, discussing them in this short book is beyond its scope. If you're interested, you can refer to two excellent books on the subject: "Optimization Models and Applications" (El Ghaoui, 2015) and "Convex Optimization" (Boyd & Vandenberghe, 2004), both of which can be accessed online for free (details provided in the Bibliography). Instead, in the next chapter, we will concentrate on SVMs again and establish an optimization problem to find the optimal hyperplane. A detailed explanation of how to solve the SVMs optimization problem will be presented there.

4.3 The SVM Optimization Problem

Given a linearly separable training dataset $\mathcal{D} = \{ (\mathbf{x_i}, y_i); \mathbf{x_i} \in \mathbb{R}^n, y_i \in -1, 1 \}_{i=1}^m$ and a hyperplane with parameters $(\mathbf{w}, b)$, recall that the **geometric marging** $\mathcal{M}$ of the hyperplane is defined as:

$$\mathcal{M} = \min_{1 \ldots m} \gamma_i$$

where $\gamma_i = y_i \left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \mathbf{x} + \frac{b}{\|\mathbf{w}\|} \right)$ is the **geometric margin** of the 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 $\mathcal{M}$ 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 $\mathcal{M}$. More specifically:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \mathcal{M} \\ \text{s.t.} \quad & \gamma_i \geq \mathcal{M} \end{aligned}$$

By recalling the relationship between the geometric and functional margin:

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

The optimization problem can now be rewritten as:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \mathcal{M} \\ \text{s.t.} \quad & \frac{f_i}{\|\mathbf{w}\|} \geq \frac{\mathcal{F}}{\|\mathbf{w}\|}, \qquad\qquad \forall i = 1 \ldots m \end{aligned}$$

We can further simplify the constraint by eliminating the norm from both sides of the inequality:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \mathcal{M} \\ \text{s.t.} \quad & f_i \geq \mathcal{F}, \qquad \forall i = 1 \dots m \end{aligned}$$

Remember, our goal is to maximize the geometric margin. And the good news is, the actual size of $\mathbf{w}$ and b doesn't matter. We have the freedom to scale them up or down as we like, and it won't change the geometric margin. So, to make things easier, we decide to scale $\mathbf{w}$ and b in a way that $\mathcal{F} = 1$. This won't impact the optimization outcome, but it will make our calculations more convenient.

The optimization problem can now be rewritten as:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \mathcal{M} \\ \text{s.t.} \quad & f_i \geq 1, \qquad \forall i = 1 \dots m \end{aligned}$$

Since $\mathcal{M} = \dfrac{\mathcal{F}}{\|\mathbf{w}\|}$, it can be written as:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \frac{\mathcal{F}}{\|\mathbf{w}\|} \\ \text{s.t.} \quad & f_i \geq 1, \qquad \forall i = 1 \dots m \end{aligned}$$

And since we have set $\mathcal{F} = 1$, it can be rewritten as:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \frac{1}{\|\mathbf{w}\|} \\ \text{s.t.} \quad & f_i \geq 1, \qquad \forall i = 1 \dots m \end{aligned}$$

This maximization problem is equivalent to the following minimization problem:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \|\mathbf{w}\| \\ \text{s.t.} \quad & y_i \left(\mathbf{w} \cdot \mathbf{x}_i\right) + b \geq 1, \qquad \forall i = 1 \dots m \end{aligned}$$

We rewrite this minimization problem as follows:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i\left(\mathbf{w}\cdot\mathbf{x}_i\right)+b \geq 1, \qquad\qquad \forall i = 1\ldots m \end{aligned}$$

We include the factor of $\frac{1}{2}$ to make things easier for us when we use the QP solver to solve the problem. Squaring the norm is beneficial because it eliminates the need for square root calculations. This simplifies the process and aids in later steps.

Finally, with one small adjustment, we rewrite it as:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i\left(\mathbf{w}\cdot\mathbf{x}_i\right)+b-1 \geq 0, \qquad\qquad \forall i = 1\ldots m \end{aligned}$$

Why did we bother rephrasing the problem this way? Well, it was because the original optimization problem was quite challenging to solve. However, by transforming it into a **convex quadratic optimization problem**, we've made it much easier to tackle, even though it might not seem apparent at first.

4.4 Summary

Initially, we assumed that certain hyperplanes outperform others when dealing with new, unseen data. Among the various hyperplanes, we referred to the one that performs the best as the "optimal hyperplane." To identify this optimal hyperplane, we sought a method to compare different hyperplanes. Ultimately, we arrived at a numerical value, known as the **geometric margin**, which also holds a geometrical interpretation.

We asserted that the optimal hyperplane is the one with the largest geometric margin, and to discover it, we aimed to **maximize the margin**. To simplify the process, we realized that by minimizing the norm of the vector $\mathbf{w}$ (the normal vector of the hyperplane), we can ensure that it corresponds to the $\mathbf{w}$ of the optimal hyperplane. This conclusion follows from the fact that $\mathbf{w}$ is involved in the formula used to compute the geometric margin.

Chapter 5

Solving the Optimization Problem

5.1 Lagrange Multipliers

Lagrange multipliers are a mathematical technique used to find the extrema (maxima or minima) of a function subject to equality constraints. In other words, they help solve optimization problems where we need to find the highest or lowest value of a function while satisfying certain constraints.

The general idea behind Lagrange multipliers is to convert a constrained optimization problem into an unconstrained one by introducing new variables called "Lagrange multipliers" or "dual variables." These multipliers are used to incorporate the constraints into the objective function.

5.1.1 The method of Lagrange Multipliers

While solving optimization problems of the below type:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g(\mathbf{x}) = 0 \end{aligned}$$

Lagrange noticed that the minima of f are found when its gradient points in the same direction as the gradient of g. In other words, the minimum is found when:

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

Thus, to find the minimum of f under the constraints of g, we need to solve:

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

Here, the constant α is called the **Lagrange Multiplier**.

Further, let us define a function $\mathcal{L}(\mathbf{x}, \alpha) = f(\mathbf{x}) - \alpha g(\mathbf{x})$, then its gradient will be $\nabla \mathcal{L}(\mathbf{x}, \alpha) = \nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x})$. Thus, solving $\nabla \mathcal{L}(\mathbf{x}, \alpha) = 0$ will allow us to find the minimum.

Now, we are ready to define the Lagrange multiplier method:

- Construct the Lagrangian function $\mathcal{L}$ with one multiplier per constriant
- Find the gradient of the Lagrangian, $\nabla \mathcal{L}$
- Solve for $\nabla \mathcal{L}(\mathbf{x}, \alpha) = 0$

5.1.2 The SVM Lagrangian

Recall from the last chapter that the SVM optimization problem is:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i \left(\mathbf{w} \cdot \mathbf{x}_i\right) + b - 1 \geq 0, \qquad \forall i = 1 \ldots m \end{aligned}$$

Our function to minimize is:

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

and the m constraints are:

$$g_i(\mathbf{w}, b) = y_i \left(\mathbf{w} \cdot \mathbf{x}_i\right) + b - 1 \geq 0, \quad \forall i = 1 \ldots m$$

We define the Lagrangian as:

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

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

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 to the primal problem, we need to solve the following **Lagrangian problem**:

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

We should note carefully that we have to simultaneously minimize with respect to $\mathbf{w}$ and b, and maximize with respect to α.

At the beginning of the chapter, it was mentioned that the technique of Lagrange multipliers is employed to tackle problems involving equality constraints. However, in our current context, we are utilizing them to handle inequality constraints. This adaptation is feasible since the method remains effective for inequality constraints, given that certain extra conditions (known as the Karush-Kuhn-Tucker conditions) are satisfied. We will delve into these conditions at a later point in our discussion.

5.2 Wolfe Dual Problem

The Lagrangian problem involves m inequality constraints, where m represents the number of training examples. It is commonly solved using its dual form. According to the **duality principle**, an optimization problem can be seen from two perspectives: the primal problem, which is a minimization problem in this context, and the dual problem, which is a maximization problem. Interestingly, the maximum value of the dual problem will always be less than or equal to the minimum value of the primal problem, providing a lower bound to the solution of the primal problem.

In our specific case, we are dealing with a convex optimization problem, and **Slater's condition** holds for affine constraints (Gretton, 2016). As a result,

Slater's theorem guarantees that **strong duality** holds. This means that the maximum value of the dual problem is equal to the minimum value of the primal problem. Solving the dual problem is essentially the same as solving the primal problem, but it is easier to handle.

We start by recalling the Lagrangian:

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

The primary Lagrangian problem is:

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

To solve the minimization problem, we need to take the partial derivative of $\mathcal{L}$ with respect to $\mathbf{w}$ and b.

$$\begin{aligned}\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i = \mathbf{0} \\ \frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{m} \alpha_i y_i = 0\end{aligned}$$

From the first equation, we get:

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

Substituing this value of $\mathbf{w}$ in $\mathcal{L}$, we get:

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

We have elimintaed $\mathbf{w}$, but b still exist in the last term:

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

Recall:

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

This means that the last term equals 0. And we can write:

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

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

The optimization problem is now called the **Wolfe dual problem**, and is written as:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ \text{s.t.} \quad & \alpha_i \geq 0, \forall i = 1 \dots m \\ \text{s.t.} \quad & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

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 $\mathcal{W}$ 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.

5.3 Karush-Kuhn-Tucker conditions

Since we are dealing with inequality constraints, there's an additional requirement: the solution must meet the Karush-Kuhn-Tucker (KKT) conditions.

The KKT conditions are necessary conditions for an optimization problem's solution to be optimal. Additionally, the problem must satisfy certain regularity conditions. Fortunately, one of these regularity conditions is Slater's condition, which we have already seen holds for SVMs. Since the primal problem we are working on is convex, the KKT conditions are also sufficient for the point to be both primal and dual optimal, resulting in a duality gap of zero.

In summary, **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

$$y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b - 1\right) \geq 0, \forall i = 1 \ldots m$$

- Dual feasibility condition

$$\alpha_i \geq 0, \forall i = 1 \ldots m$$

- Complementary slackness condition

$$\alpha_i \left[y_i \left(\mathbf{w} \cdot \mathbf{x}_i + b - 1\right)\right] = 0, \forall i = 1 \ldots m$$

"[...]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.

5.3.1 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.

5.3.2 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.

5.3.3 Dual feasibility condition

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

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

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

5.4 What to do once we have the multipliers?

When we work on solving the Wolfe dual problem, we obtain a vector α that holds all the Lagrange multipliers. But initially, our objective in formulating the primal problem was to find the values of $\mathbf{w}$ and b. Now, we'll explore how we can extract these specific values of w and b from the Lagrange multipliers.

5.4.1 Computing w

We already derived the formula for $\mathbf{w}$: $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$ from the gradient $\nabla_{\mathbf{w}} \mathcal{L}$

5.4.2 Computing 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 $\mathbf{w}$):

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

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:

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

where S is the number of Support Vectors.

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

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

They 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.

5.4.3 Computing Hypothesis Function

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

$$h(\mathbf{x_i}) = sign(\mathbf{w} \cdot \mathbf{x_i} + b)$$

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

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

5.5 Solving SVMs with a QP solver

A quadratic programming (QP) solver is a software tool designed to tackle quadratic programming problems. In this case, we'll utilize a Python package named CVXOPT as an example.

This package provides a method that can solve quadratic problems of the form:

$$\begin{aligned} \min_{x} \quad & \frac{1}{2} x^T P x + q^T x \\ \text{s.t.} \quad & Gx \preccurlyeq h \\ \text{s.t.} \quad & Ax = b \end{aligned}$$

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 x of to better see how the two problems relate:

$$\begin{aligned} \min_{x} \quad & \frac{1}{2}\alpha^T P\alpha + q^T\alpha \\ \text{s.t.} \quad & G\alpha \preccurlyeq h \\ \text{s.t.} \quad & A\alpha = b \end{aligned}$$

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

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{m}\alpha_i \\ \text{s.t.} \quad & \alpha_i \geq 0, \forall i = 1 \dots m \\ \text{s.t.} \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

into a minimization problem by multiplying by -1:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_{i=1}^{m}\alpha_i \\ \text{s.t.} \quad & -\alpha_i \leq 0, \forall i = 1 \dots m \\ \text{s.t.} \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

Next we introduce vectors α and $\mathbf{y}$ and the Gram Matrix $\mathbf{K}$ consisting of all possible dot products of vectors $\mathbf{x}_i$:

$$\alpha = (\alpha_1, \alpha_2, \dots \alpha_m)^\top$$

$$\mathbf{y} = (y_1, y_2, \dots y_m)^\top$$

$$\mathbf{K}(\mathbf{x}_1, \mathbf{x}_2 \dots \mathbf{x}_m) = \begin{bmatrix} \mathbf{x}_1 \cdot \mathbf{x}_1 & \mathbf{x}_1 \cdot \mathbf{x}_2 & \dots & \mathbf{x}_1 \cdot \mathbf{x}_m \\ \mathbf{x}_2 \cdot \mathbf{x}_1 & \mathbf{x}_2 \cdot \mathbf{x}_2 & \dots & \mathbf{x}_2 \cdot \mathbf{x}_m \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{x}_m \cdot \mathbf{x}_1 & \mathbf{x}_m \cdot \mathbf{x}_2 & \dots & \mathbf{x}_m \cdot \mathbf{x}_m \end{bmatrix}$$

Now we construct a vectorized version of the Wolfe dual problem where $\mathbf{y}\mathbf{y}^\top$ denotes the outer product of $\mathbf{y}$:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^\top \left(\mathbf{y}\mathbf{y}^\top \mathbf{K}\right) \alpha - \alpha \\ \text{s.t.} \quad & -\alpha \preccurlyeq 0 \\ \text{s.t.} \quad & \mathbf{y} \cdot \alpha = 0 \end{aligned}$$

We are now in a position 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 the next code snippet.

Listing 5.1: CVXOPT qp function

```
from mySVM.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]
```

The code 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.

We can nbow compute $\mathbf{w}$ using all the Lagrange Multipliers: $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$.

Listing 5.2: Computing **w**

```
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:

Listing 5.3: Example - Computing **w**

```
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 finally compute b:

Listing 5.4: Example - Computing b

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

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

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

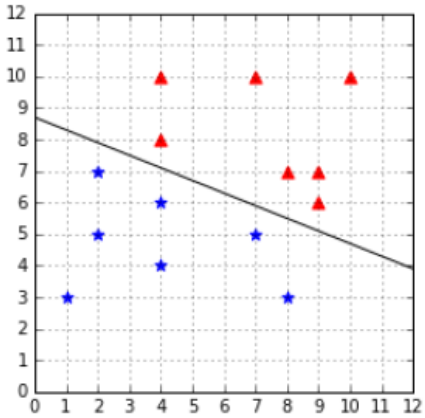

Figure 5.1: The hyperplane using CVXOPT

The version of the Support Vector Machine (SVM) described here is known as the **hard margin SVM**. It is only effective when the data can be perfectly separated by a straight line. However, in situations where the data is not linearly separable, this method fails. There are various other formulations of Support Vector Machines, and in the upcoming chapter, we will explore the **soft margin SVM** which can handle non-linearly separable data by considering outliers.

5.6 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 6

Soft Marging SVM

6.1 Noisy Data

The main challenge with the **hard margin** Support Vector Machine (SVM) is that it necessitates the data to be separable by a straight line boundary. However, real-world data tends to be imperfect, often containing noise. Even when the data appears separable by a straight line, various factors might affect it before using it for modeling. For instance, there could be typos in data values or erroneous readings from sensors. When an outlier, which is a data point deviating significantly from its expected group, is present, two scenarios emerge: the outlier might be closer to other examples from a different group, thus shrinking the separation margin, or it could be situated within the group of other examples, causing the linear separability to break down. Let's examine these two situations and observe how the hard margin SVM addresses them.

6.1.1 Outlier reducing margin

When the data points can be cleanly divided with a straight line, the strict boundary classifier doesn't work well if there are extreme values that don't fit the pattern. Now, let's think about our data collection, but this time we've included an outlier at the coordinates $(5, 7)$, as illustrated in the next figure.

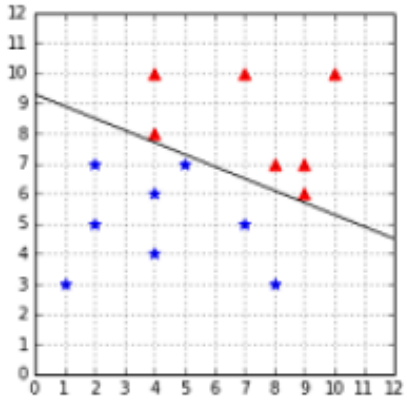

Figure 6.1: Outlier reducing the margin

In this situation, it's evident that the margin is quite small, and it appears that the unusual data point is primarily responsible for this shift. Common sense suggests that this specific plane might not effectively distinguish the data, and it's likely to generalize poorly.

6.1.2 Outlier breaking liner seperability

Consider what happens if we add point $(7, 8)$ as in the next figure.

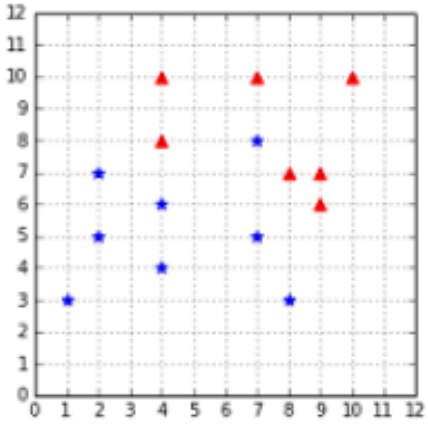

Figure 6.2: Outlier breaking linear seperability

Now we are faced with a problem where cannot find any linearly separable hyperplane that will classify all the data points. We are stuck because of a single outlier data point.

6.2 Help at Hand

In 1995, Vapnik and Cortes made a change to the original Support Vector Machine. This alteration permitted the classifier to have a certain degree of error. Instead of aiming for perfect classification, the new objective was to minimize the number of errors. To achieve this, they adjusted the optimization problem's restrictions by introducing a new variable called ζ (zeta). This adjustment resulted in a change to the previous constraint:

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

The modified constraint becomes:

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

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 to the hyperplane, or it is not on the correct side of the hyperplane). This is shown in the next code example:

Listing 6.1: Adding constraints to hyperplanes

```
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 for every example, and all the constraints will be satisfied.

Listing 6.2: Adding LARGE constraints to hyperplanes

```
# 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
```

This problem can be fixed by modifying the objective function penalizing for large values of ζ_i:

$$\begin{aligned} \min_{\mathbf{w},b,\zeta} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{m} \zeta_i \\ \text{s.t.} \quad & y_i\left(\mathbf{w}\cdot\mathbf{x}_i + b\right) \geq 1 - \zeta_i, \forall 1 \ldots m \end{aligned}$$

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 small 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).

We now define $softmargin$ as:

$$\begin{aligned} \min_{\mathbf{w},b,\zeta} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{m} \zeta_i \\ \text{s.t.} \quad & y_i\left(\mathbf{w}\cdot\mathbf{x}_i + b\right) \geq 1 - \zeta_i, \forall 1 \ldots m \\ \text{s.t.} \quad & \zeta_i \geq 0, \forall 1 \ldots m \end{aligned}$$

Following the same method as earlier, we derive the **Wolfe Dual** conditions:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \forall i = 1 \ldots m \\ \text{s.t.} \quad & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

We see that the constraint $\alpha_i \geq 0$ is modified to $0 \leq \alpha_i \leq C$. This limitation is frequently referred to as the **box constrasint** since the vector α must stay within a box situated in the **positive orthant** with a side length of C. It's worth noting that an orthant is akin to a quadrant in a two-dimensional plane but in n-dimensional Euclidean space. We will revisit box constraint when we discuss SMO Algorithm in a later chapter.

This optimization problem is also called $1 - normsoftmarging$ since we are minimizing the $1 - norm$ of the slack vector ζ.

6.3 What does C do?

The parameter C allows you to influence how the SVM handles errors. Now, let's explore how modifying its value leads to different hyperplanes.

In the next figure, we have the dataset that's been separable by a straight line, a scenario we've consistently used in this book. On the left, setting C to a very high value ($+\infty$) produces the same outcome as the hard margin classifier. However, opting for a smaller C, as demonstrated in the middle, shifts the hyperplane closer to certain points, thereby disregarding the hard margin constraint for these specific instances. This effect intensifies when C is set to a smaller value like 0.01, as shown on the right.

What if we were to choose an extremely low value for C? In this case, constraints practically disappear, resulting in a hyperplane that fails to classify anything effectively.

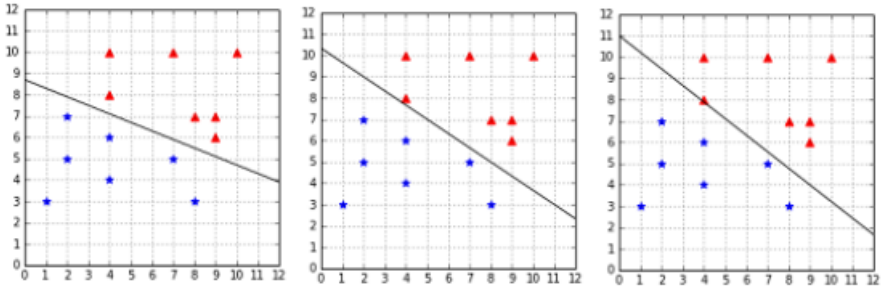

Figure 6.3: Effect of C on a Linearly Seperable Dataset

It appears that when the data can be separated in a straight line, opting for a larger C is the optimal decision. But what if there are some noisy exceptions? In this scenario, as depicted next, selecting $C = +\infty$ results in an extremely narrow gap. However, by choosing $C = 1$, we achieve a hyperplane that closely resembles the one used in the hard margin classifier when outliers are absent. The sole violated condition is that of the outlier, and we find this hyperplane to be more satisfactory. On this occasion, choosing $C = 0.01$ leads to a violation of a constraint for another instance, which was not an outlier. This value of C seems to grant excessive flexibility to our soft margin classifier.

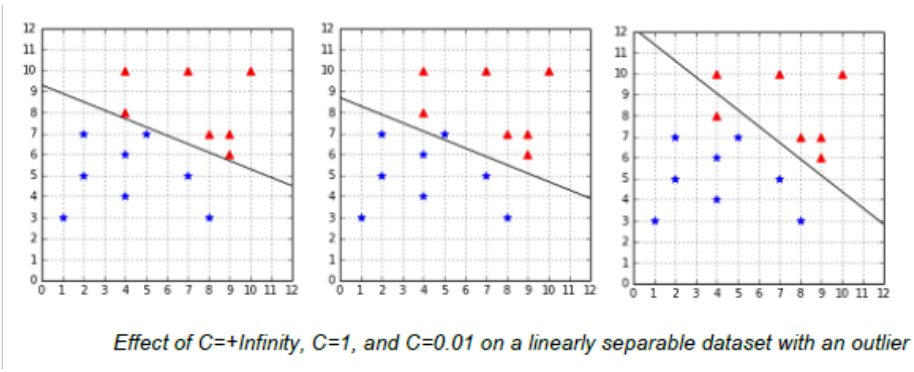

Figure 6.4: Effect of C on a Linearly Seperable Dataset with Outlier

In situations where the data is not separable, the option of setting $C = +\infty$ becomes impractical due to the inability to satisfy all strict margin requirements. Instead, we explore different C values and find that the optimal hyperplane occurs when C is set to 3. Interestingly, this same hyperplane holds true for all C values greater than or equal to 3. This outcome arises because, regardless of the level of penalization applied, it remains essential to violate the outlier constraint in order to achieve data separation. With lower C values, similar to what was done previously, more of these constraints end up being violated.

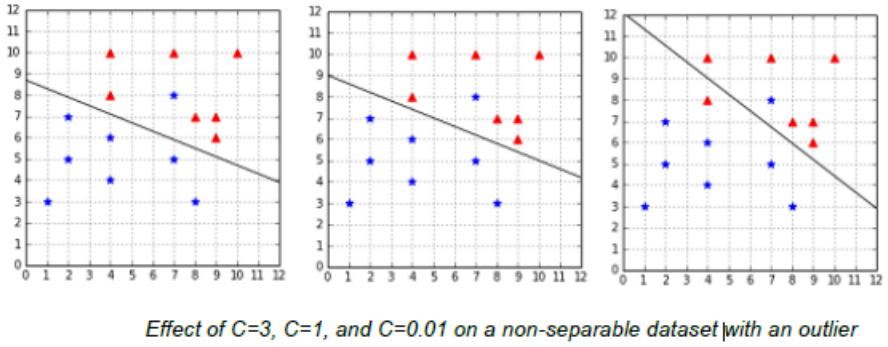

Figure 6.5: Effect of C on a non-Seperable Dataset

We can now summarize the following heuristic:

- A lower value of C will provide a large margin, although it might result in a few incorrect classifications.
- A very high value of C will yield a hard margin classifier that doesn't allow any violations of constraints.

- The crucial part is determining the right C value, ensuring that noisy data doesn't overly influence the solution.

6.4 Finding the best value of C

There isn't a one-size-fits-all C value that solves every problem. The best method is to employ grid search along with cross-validation, as suggested by Hsu, Chang, & Lin in their work *A Practical Guide to Support Vector Classification*. It's important to grasp that the optimal C value depends heavily on the particular dataset you're working with. Therefore, if you find that $C = 0.001$ wasn't effective for one problem, don't discard it entirely; it might still prove useful for a different problem since its impact can vary.

6.5 2-norm Soft Margin

There is another formulation of the problem called the **2-norm (or L2 regularized)** soft margin in which we minimize $\frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{m} {\zeta_i}^2$. 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*.

6.6 ν SVM

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

Under these conditions, the optimization problem is:

$$\begin{aligned}
\max_{\alpha} \quad & -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\
\text{s.t.} \quad & 0 \leq \alpha_i \leq \frac{1}{m}, \forall i = 1 \ldots m \\
\text{s.t.} \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \\
\text{s.t.} \quad & \sum_{i=1}^{m}\alpha_i \geq \nu
\end{aligned}$$

6.7 Summary

The soft-margin SVM formulation represents a notable advancement over the hard-margin classifier. This improvement enables us to effectively categorize data, even in the presence of noisy data that disrupts straightforward separation. However, this enhanced adaptability comes at the cost of introducing a new parameter, C, which requires careful tuning. We observed how adjusting the value of C influences the margin and permits the classifier to make certain errors in exchange for a larger margin. This underscores the notion that our objective is to discover a hypothesis that performs effectively with new, unseen data. Making a few errors during the training phase is acceptable if the model can ultimately generalize effectively.

Chapter 7

Kernels

7.1 Feature transformations

7.1.1 Can non-linearly separable data be classified?

Picture this: you've got some information that can't be neatly divided into categories, just like the example shown in the next figure. Now, let's say you want to employ SVMs to sort this data into groups. Initially, it seems like this won't work because the data can't be separated with straight lines. But hold on, that last idea isn't entirely accurate. What really matters is that the data can't be separated by straight lines when looking at it in **two dimensions**.

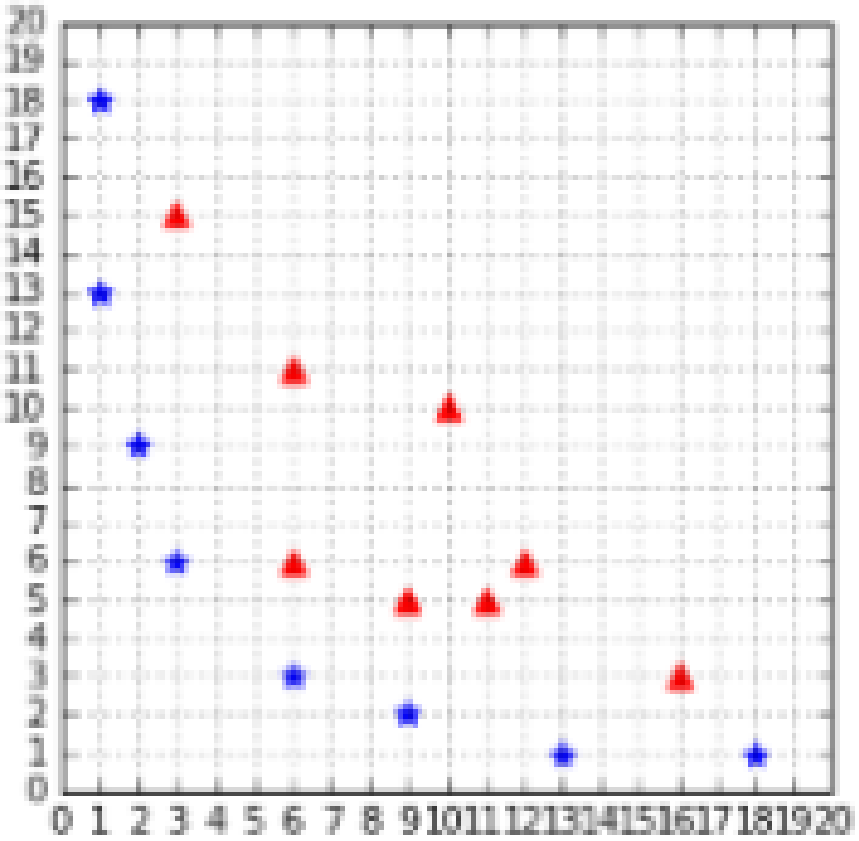

Figure 7.1: A Straight Line cannot separate the data

Even if your initial information exists in a two-dimensional form, you can still modify it before inputting it into the SVM. For example, you could change each two-dimensional set of data into a three-dimensional set.

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

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

Programatically it can be done like:

Listing 7.1: Transforming Vectors

```
# 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 the earlier figure and plot the result, you get below, 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 as shown next.

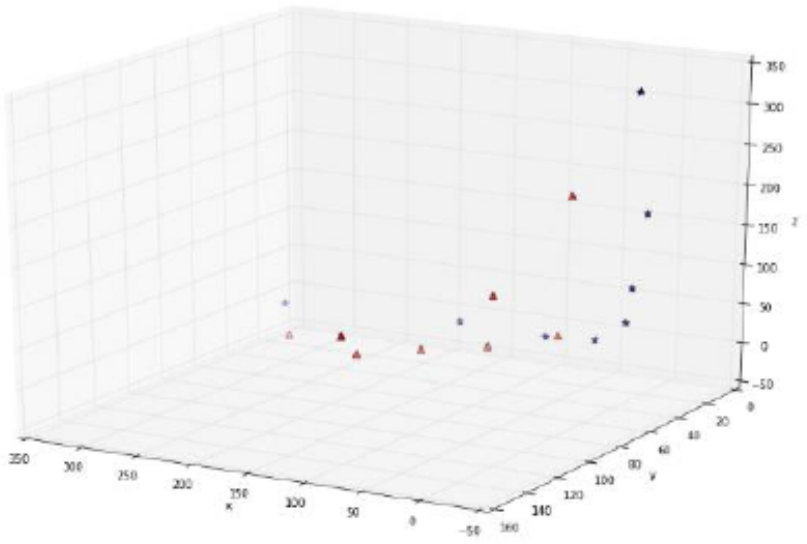

Figure 7.2: Not looks seperable in 3D

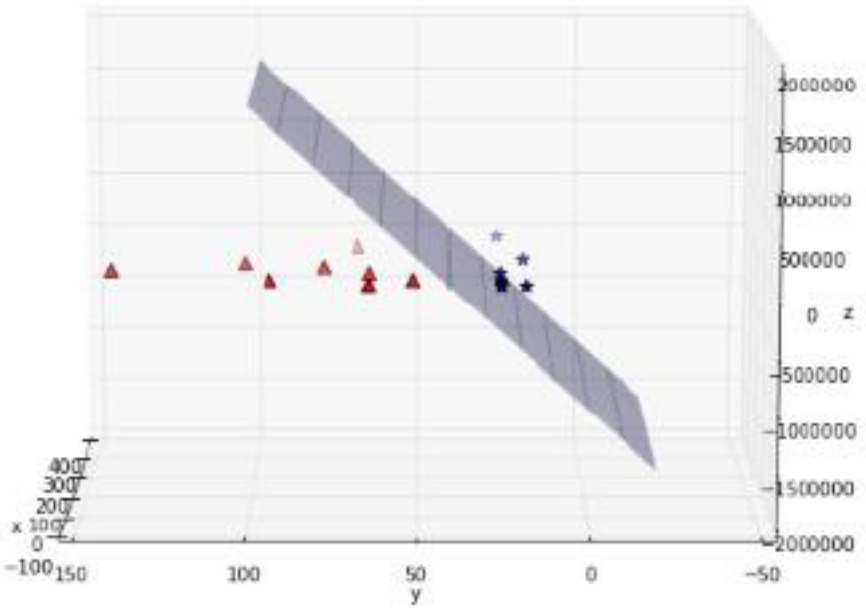

Figure 7.3: Data Seperable by plane in 3D

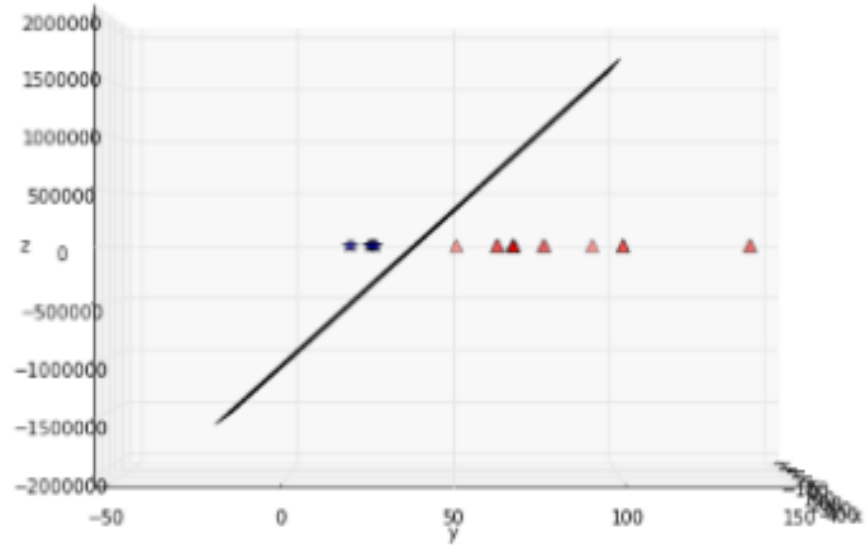

Figure 7.4: Data Seperable by plane in 3D (Side View)

Here's a simple set of instructions we can follow to categorize this dataset:

- Convert each vector of two-dimensional into a three-dimensional one using the code outlined earlier.
- Train the SVMs using this new 3D dataset
- Whenever we have a fresh example to predict, modify it using the method described in **transform** before giving it to the **predict** method.

Remember, you have flexibility in choosing the number of dimensions to convert the data to — it could be any number, like five, ten, or even a hundred dimensions.

7.1.2 Which Transformation to apply?

Deciding which transformations to make to your data depends heavily on the specific dataset you're working with. A significant aspect of achieving success in the realm of machine learning is the ability to transform your data in a way that optimizes the performance of your chosen machine learning algorithm. However, there isn't a foolproof formula for this; it often requires learning from experience through trial and error. It's wise to ensure you examine any established guidelines or conventions for data transformation provided in the documentation before applying any algorithm. To delve deeper into preparing your data, you might want to explore the section on dataset transformation available on the scikit-learn website.

7.2 What is a Kernel?

In the previous part, we learned about a simple technique to apply to datasets that can't be easily separated. However, a major downside of this approach is the need to modify each individual instance. If we're dealing with an extremely large number of examples, possibly in the millions or billions, and the modification process is intricate, it could end up being an incredibly time-consuming task. This is where kernels step in to offer a solution.

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:

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

In the below code example, 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 8100.

Listing 7.2: Example - dot product of transformed vectors

```
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**!

Consider the code:

Listing 7.3: Example - dot product of transformed vectors

```
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.

Listing 7.4: Dot product using kernel

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

# We do not transform the data.
print(polynomial_kernel(x1, x2)) # 8100
```

When you consider it, this is quite remarkable.

The vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ are part of the two-dimensional real space $\mathbb{R}^2$. The kernel function calculates their dot product as though they've been changed into vectors belonging to a three-dimensional real space $\mathbb{R}^2$, and it accomplishes this without actually carrying out the transformation or explicitly calculating their dot product!

To put it succinctly: **a kernel is a function that yields the outcome of a dot product carried out in a different space**. To state it more precisely:

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

7.3 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: $\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x})$, we can rewrite the soft margin Wolfe Dual as:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathcal{K}\left(\mathbf{x}_i \cdot \mathbf{x}_j\right) \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \forall i = 1 \ldots m \\ \text{s.t.} \quad & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

That's it. We have made a single change to the dual problem — we call it the **kernel trick**. Applying the kernel trick simply means replacing the dot product of two examples with 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 to classify non-separable data.

We also need to change the hypothesis function to use the kernel function:

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

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*.

7.4 Kernel Types

7.4.1 Linear Kernel

This is the simplest kernel. For two vectors $\mathbf{x}$ and $\mathbf{x}'$ it is simply defined by:

$$\mathcal{K}\left(\mathbf{x}_j \cdot \mathbf{x}_i\right) = \mathbf{x} \cdot \mathbf{x}'$$

The liner kernel works well for text classification problems.

7.4.2 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:

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

The parameters c represent a constant term, and d, represents the degree of the kernel. The Python implementation is given below:

Listing 7.5: Polynomial Kernel definition

```
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 the below code, we see that it returns the same result as the kernel from the earlier example when we use degree 2.

Listing 7.6: Example - Polynomial Kernel, degree = 2

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

# We do not transform the data.
print(polynomial_kernel(x1, x2, degree=2)) # 8100
```

The result of training an SVM with this kernel is:

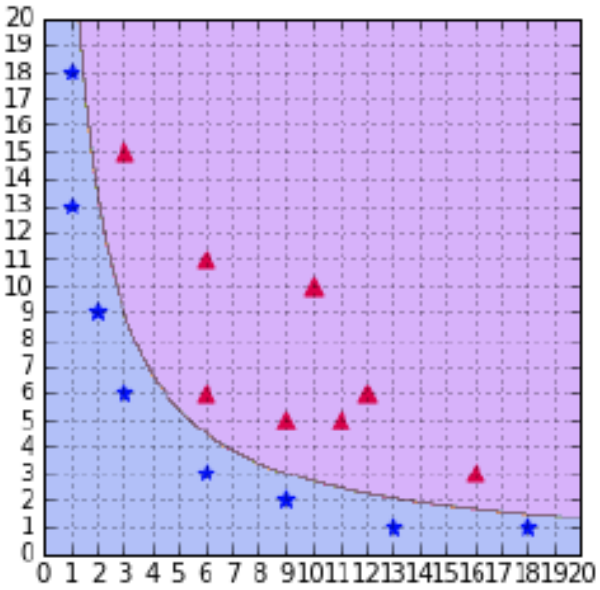

Figure 7.5: SVM using a polynomial kernel to separate the data (degree=2)

A polynomial kernel with a degree of 1 and without a constant term is essentially the same as a linear kernel (left figure). If you raise the degree of the polynomial kernel, the boundary that separates data points will become more intricate. This complexity might start to be swayed by individual data points, as depicted in the right figure. However, utilizing a high-degree polynomial comes with a risk. While it might improve results on your test data, it can lead to **overfitting**, where the model fits the training data too closely and struggles to generalize to new, unseen data.

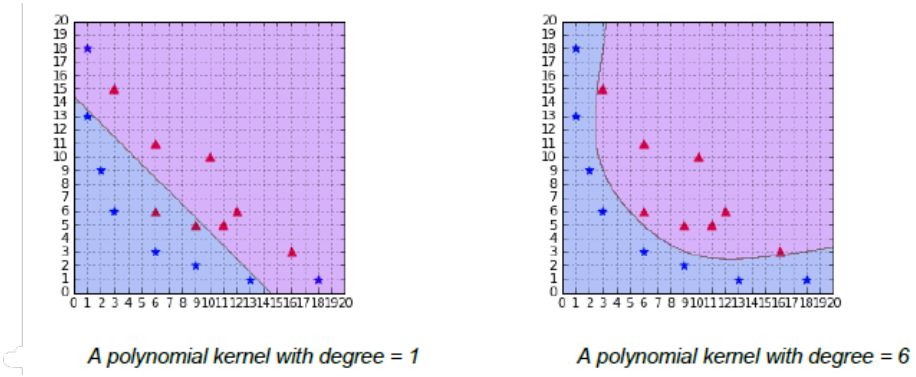

Figure 7.6: Polynomial Kernel with Linear and Overfitting example

7.4.3 The RBF Kernel

Sometimes polynomial kernels are not sophisticated enough to work. When you have a difficult dataset like the one depicted below, the limitations will become evident quickly:

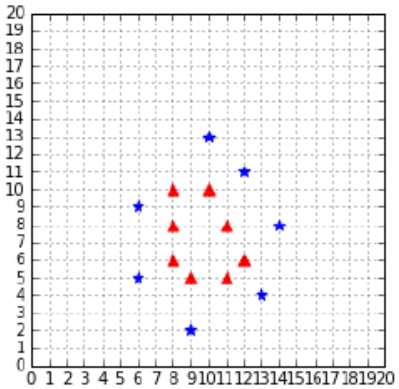

Figure 7.7: Limitations of Polynomial Kernel

As we can see in the next figure, the decision boundary of a polynomial kernel is very bad at classifying the data.

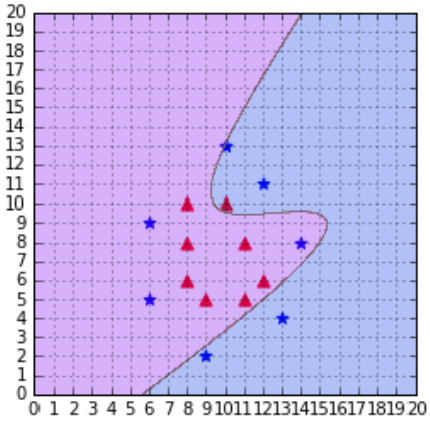

Figure 7.8: Limitations of Polynomial Kernel (degree=3, C=100)

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:

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

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 $\mathbb{R}^\infty$.

Proof of this assertion can be found here.

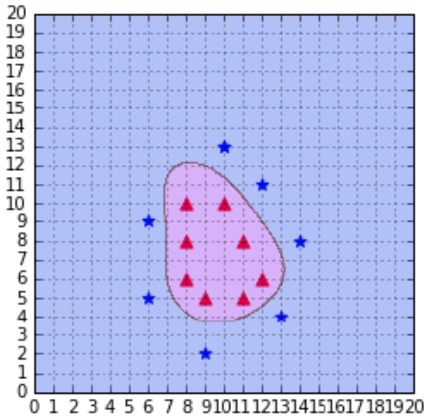

Figure 7.9: RBF Kernel (gamma = 0.1) in action

The below figure shows the impact of parameter γ. When γ is too small, the model behaves like a linear model (left), and when it's too large, the model is influenced by each data point (right).

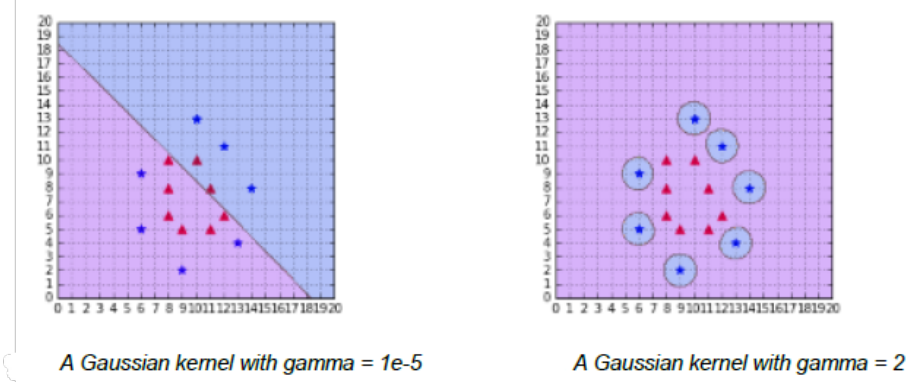

Figure 7.10: RBF Kernel varying with gamma

For more information about gamma, you can read this scikit-learn documentation page.

7.5 Which Kernel to use?

It's a good idea to start with an RBF kernel since it often performs effectively. Yet, if you've got the time, experimenting with different kernel types is worth-

while. **Kernels essentially gauge how alike two vectors are**, and your knowledge of the specific problem can greatly influence this choice. Crafting your own kernel is an option too, but it demands a solid grasp of the mathematical principles underpinning kernels. For further insights, you can refer to the work by *Cristianini & Shawe-Taylor (2000)*.

7.6 Summary

The kernel trick plays a crucial role in enhancing the capabilities of Support Vector Machines. It enables us to utilize SVMs for a wide array of problems. In this chapter, we explored the limitations of the linear kernel and how a polynomial kernel can effectively classify data that isn't easily separable. We eventually delved into one of the most frequently employed and potent kernels: the RBF kernel. It's important to keep in mind that there exist numerous kernels, and it's worth exploring kernels specifically designed to address the types of problems you're dealing with. The choice of the appropriate kernel in conjunction with the correct dataset significantly influences your success or failure when working with SVMs.

Chapter 8

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:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathcal{K}\left(\mathbf{x}_i \cdot \mathbf{x}_j\right) - \sum_{i=1}^{m}\alpha_i \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \forall i = 1 \ldots m \\ \text{s.t.} \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

It is a kernelized version of the soft-margin formulation we saw in the previous chapter. The objective function we are trying to minimize can be written in Python as:

Listing 8.1: Kernel Objective function

```
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.

8.1 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 $\alpha = (\alpha_1, \alpha_2, \dots \alpha_m)$ 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 $\alpha = (\alpha_1, \alpha_2, \dots \alpha_m)$, 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.

In short, SMO solves a sequence of several simpler optimization problems.

8.2 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. 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.

8.3 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.

8.4 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.

8.4.1 The analytical solution

At the beginning of the algorithm, we start with a vector $\alpha = (\alpha_1, \alpha_2, \ldots \alpha_m)$ in which $\alpha_i = 0, \forall i = 1 \ldots m$. 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 $0 \leq \alpha_i \leq C, \forall i = 1 \ldots m$ 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 the below figure (which displays an example where $C = 5$).

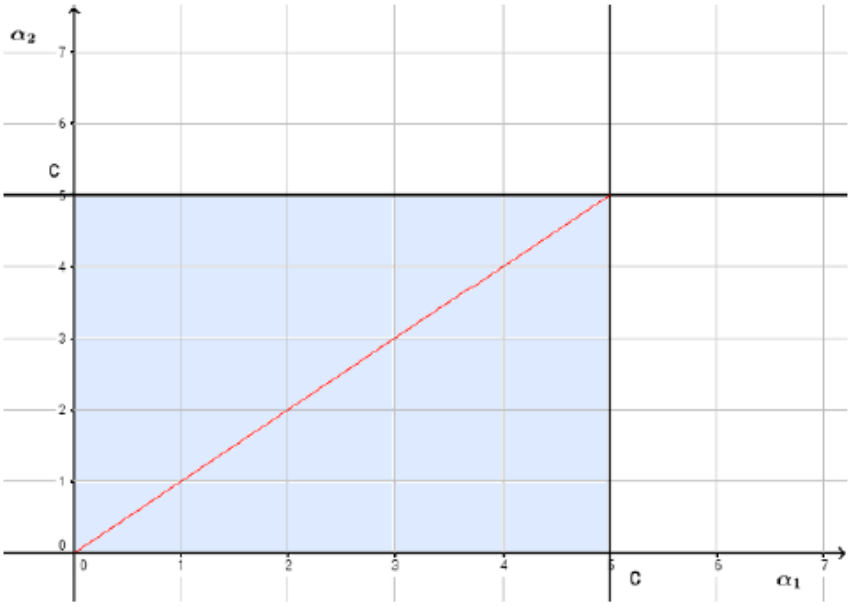

Figure 8.1: Boxed Constraints for SMO

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 ($y_1 \neq y_2$).

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

$$\alpha_1 y_1 + \alpha_2 y_2 = constant = {\alpha_1}^{old} y_1 + {\alpha_2}^{old} y_2$$

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*.

The formula to compute ${\alpha_2}^{new}$ is:

$${\alpha_2}^{new} = \alpha_2 + \frac{y_2 \left(E_1 - E_2\right)}{\mathcal{K}\left(\mathbf{x_1}, \mathbf{x_1}\right) + \mathcal{K}\left(\mathbf{x_2}, \mathbf{x_2}\right) - 2\mathcal{K}\left(\mathbf{x_1}, \mathbf{x_2}\right)}$$

Here, $E_i = f(\mathbf{x_i}) - y_i$ is the difference between the output of the hypothesis function and the example label. $\mathcal{K}$ is the kernel function. We also compute bounds, which apply to ${\alpha_2}^{new}$; it cannot be smaller than the lower bound, or larger than the upper bound, or constraints will be violated. So ${\alpha_2}^{new}$ is clipped if this is the case.

Once we have this new value, we use it to compute the ${\alpha_1}^{new}$ using this formula:

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

8.4.2 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. 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.

Listing 8.2: First heuristic - part 1

```
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. Note that the algorithm finishes when progress is no longer made.

Listing 8.3: First heuristic - part 2

```
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
```

8.4.3 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 ${\alpha_2}^{new}$? We use the following formula:

$$ {\alpha_2}^{new} = \alpha_2 + \frac{y_2 \left(E_1 - E_2\right)}{\mathcal{K}\left(\mathbf{x_1}, \mathbf{x_1}\right) + \mathcal{K}\left(\mathbf{x_2}, \mathbf{x_2}\right) - 2\mathcal{K}\left(\mathbf{x_1}, \mathbf{x_2}\right)} $$

Remember that in this case that we have already chosen the value α_1. Our goal is to pick the α_2 whose ${\alpha_2}^{new}$ will have the biggest change. This formula can be rewritten as follows:

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

where:

$$step = \frac{y_2 \left(E_1 - E_2\right)}{\mathcal{K}\left(\mathbf{x_1}, \mathbf{x_1}\right) + \mathcal{K}\left(\mathbf{x_2}, \mathbf{x_2}\right) - 2\mathcal{K}\left(\mathbf{x_1}, \mathbf{x_2}\right)}$$

So, to pick the best α_2 amongst several α_i, we need to compute the value of the 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 $\mathcal{K}$ three times for each step, and this is costly. Instead of doing that, Platt came up with the following approximation:

$$step \approx |E_1 - E_2|$$

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. The next snippet makes it clear.

Listing 8.4: Second heuristic

```
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

    #continued on next page
```

Listing 8.5: Second heuristic (contd.)

```
    # ...
    # 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
```

8.5 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 are 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 9

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 contains data from three different producers. Several approaches 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 as follows.

Listing 9.1: Four Class Classification Dataset

```
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])
```

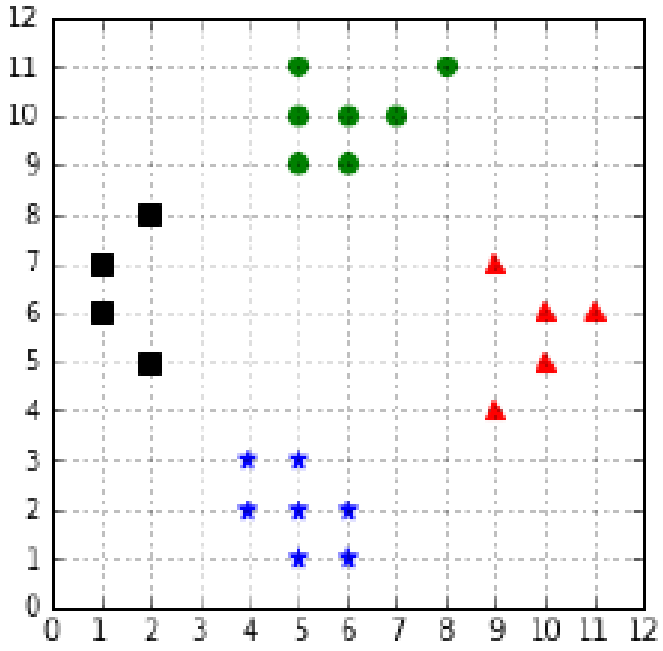

Figure 9.1: Four Class Classification

9.1 Solving multiple binary problems

9.1.1 One-against-all

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

Listing 9.2: One-against-all

```
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 class 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. As a result, we obtain one decision boundary per classifier.

Listing 9.3: One-against-all (contd.)

```
# 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)
```

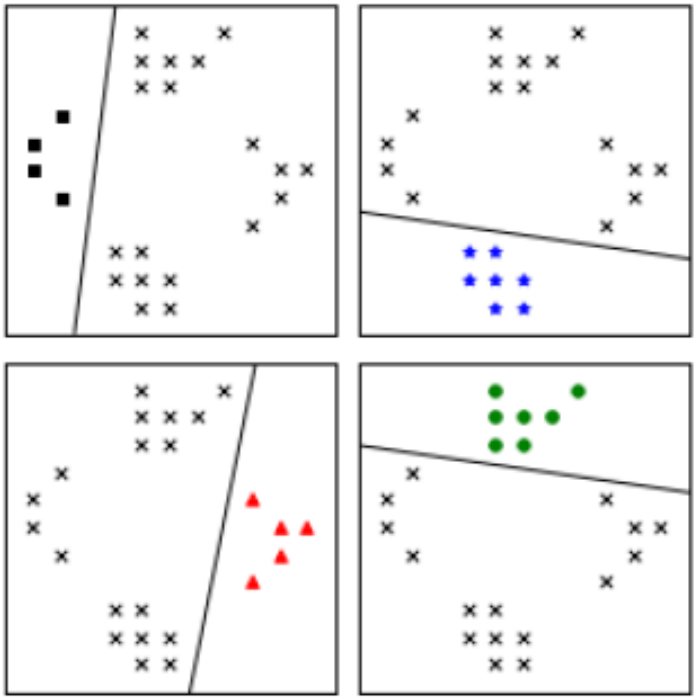

Figure 9.2: One-against-all approach with one classifier per class

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). The next 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 classes 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.

Listing 9.4: One-against-all (contd.)

```
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)
```

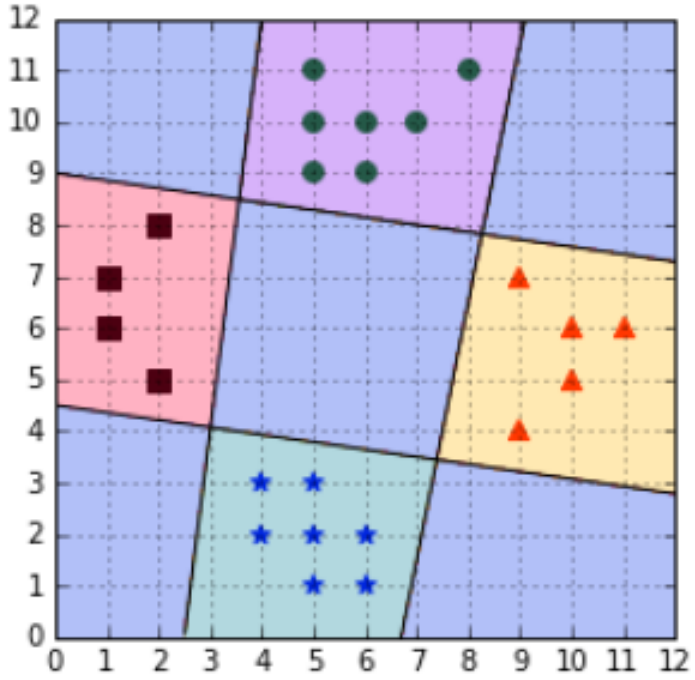

Figure 9.3: One-against-all and associated ambiguities

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 the next code sample. 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.

Listing 9.5: One-against-all (contd.)

```
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 below. 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 the results of the others, its class will be assigned incorrectly to some examples.

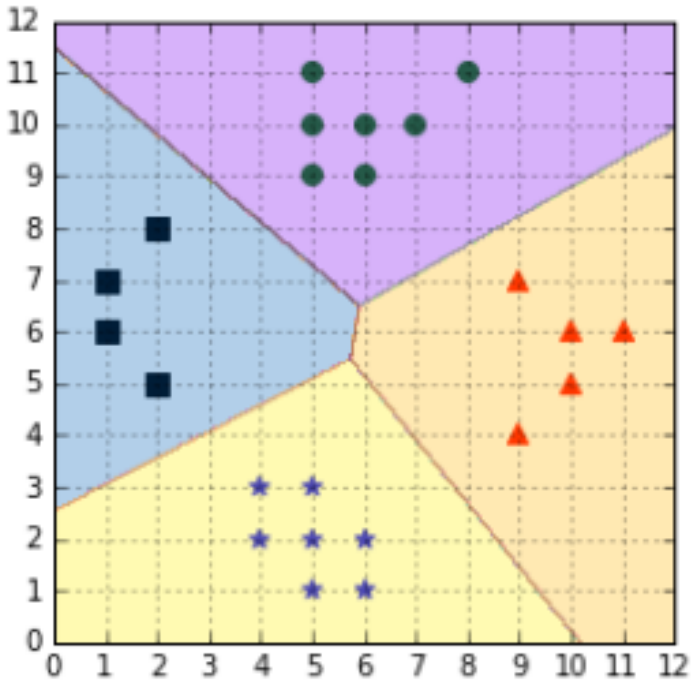

Figure 9.4: One-against-all with Heuristics

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.

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 below.

Listing 9.6: One-against-all with sklearn/LinearSVC

```
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]
```

9.1.2 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 $\frac{K \times (K-1)}{2}$ classifiers for K classes. Each classifier is trained on a subset of the data and produces its own decision boundary.

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.

Listing 9.7: One-against-one

```
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]
```

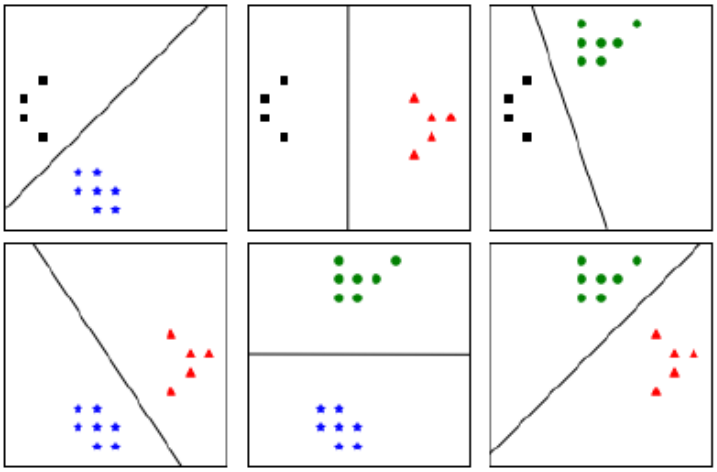

Figure 9.5: One-against-one construct with one classifier for each pair of classes

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.*

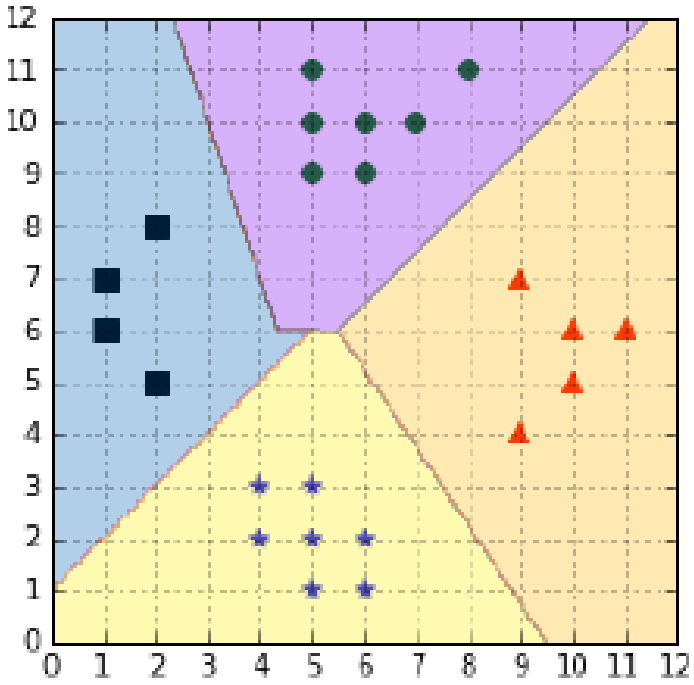

Figure 9.6: Predictions are made using a voting scheme

The above figure shows us that the decision regions generated by the one-against-one strategy are different from the ones generated by one-against-all. It is interesting to note (next figure) 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.

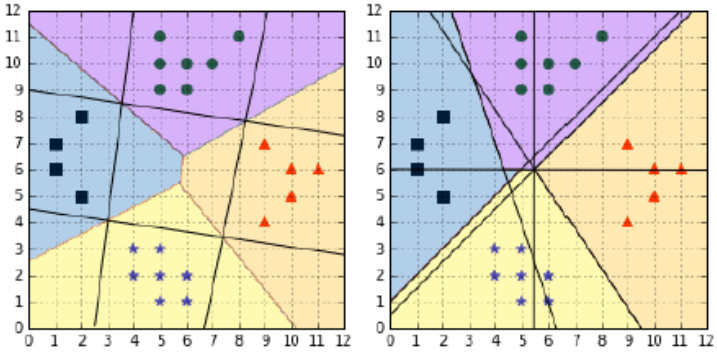

Figure 9.7: Comparison of one-against-all (left) and one-against-one (right)

The one-against-one approach is the default approach for multi-class classification used in **sklearn**.

Listing 9.8: One-against-one using sklearn

```
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.*

9.1.3 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.* 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 predicts 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 the below figure. Each node of the graph is a classifier for a pair of classes.

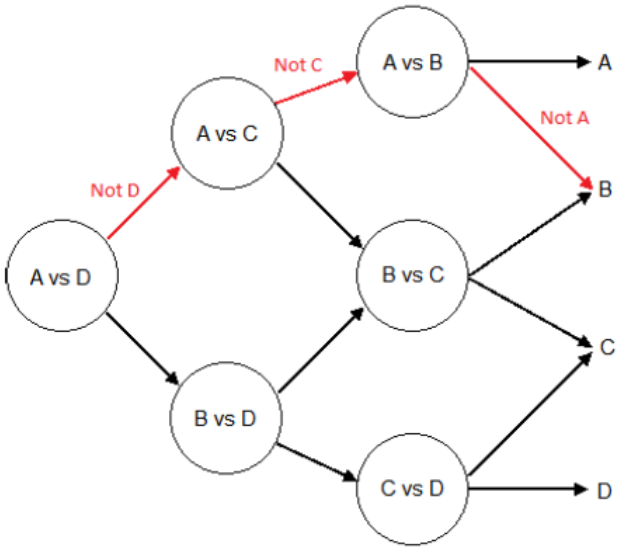

Figure 9.8: Prediction using DAG

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. In the code below, 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.

Listing 9.9: One-against-one using sklearn

```
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)
```

9.2 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.

9.2.1 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.

9.2.2 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. We can further see that the C&S predictions are different from the one-against-all and the one-against-one methods.

Listing 9.10: Crammer and Singer Algorithm

```
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]
```

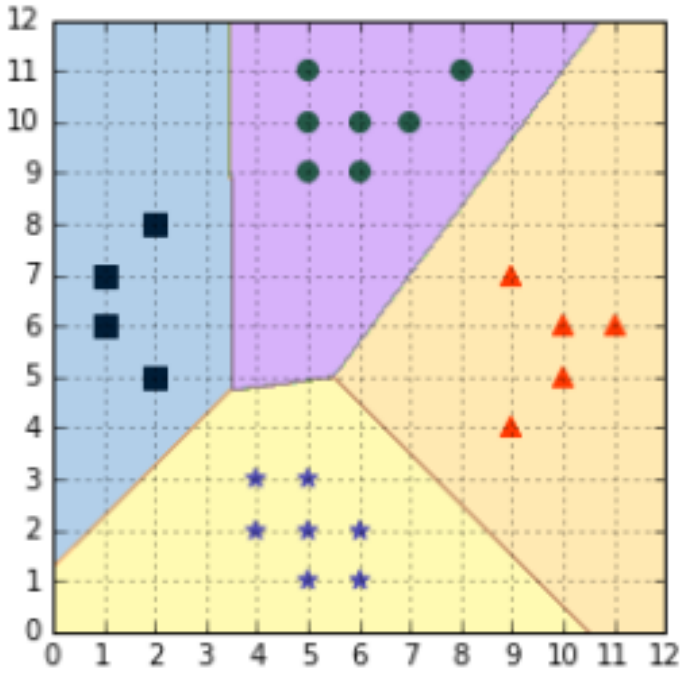

Figure 9.9: Prediction using Crammer and Singer

9.3 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.

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

Figure 9.10: Overview of multi-class SVM methods

9.4 Summary

Thanks to many improvements over the years, there are now several methods for doing multiclass 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*.

Chapter 10

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 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 they work. It is the subject of a field called computational learning theory (SVMs are coming from statistical learning theory).

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 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.

Chapter 11

Appendix A: Datasets

The following code is used to load the simple linearly separable dataset used in most chapters of this book. This is also available on the GitHub page.

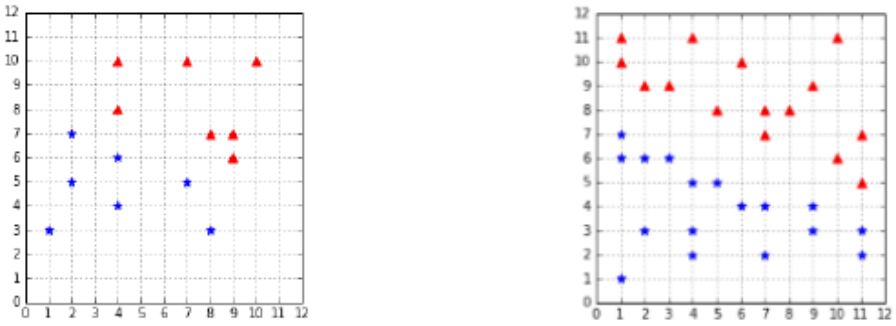

Figure 11.1: Training and Test sets

The method **get_training_examples** returns the data shown in the left, while the method **get_test_examples** returns the data shown in the right figure.

Listing 11.1: Getting Training and Test sets

```
from mySVM.datasets import *
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
```

The below code shows a typical use case:

Listing 11.2: Getting Training and Test sets

```
from mySVM.datasets import get_dataset, linearly_separable as ls
import numpy as np

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

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
```

Chapter 12

Appendix B: The SMO Algorithm

Listing 12.1: The SMO Algorithm

```
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 it stays at the end of the
            ↪  line.
            # Equation 17
            if a2_new < L:
                a2_new = L
```

```
75              elif a2_new > H:
76                  a2_new = H
77          else:
78              # Under unusual circumstances, eta will not be positive.
79              # Equation 19
80              f1 = y1 * (E1 + self.b) - a1 * k11 - s * self.a2 * k12
81              f2 = self.y2 * (self.E2 + self.b) - s * a1 * k12 \
82                      - self.a2 * k22
83              L1 = a1 + s(self.a2 - L)
84              H1 = a1 + s * (self.a2 - H)
85
86              Lobj = L1 * f1 + L * f2 + 0.5 * (L1 ** 2) * k11 \
87                          + 0.5 * (L ** 2) * k22 + s * L * L1 * k12
88              Hobj = H1 * f1 + H * f2 + 0.5 * (H1 ** 2) * k11 \
89                          + 0.5 * (H ** 2) * k22 + s * H * H1 * k12
90
91              if Lobj < Hobj - self.eps:
92                  a2_new = L
93              elif Lobj > Hobj + self.eps:
94                  a2_new = H
95              else:
96                  a2_new = self.a2
97
98              # If alpha2 did not change enough the algorithm
99              # returns without updating the multipliers.
100             if abs(a2_new - self.a2) < self.eps * (a2_new + self.a2 \
101                 + self.eps):
102                 return False
103
104             # Equation 18
105             a1_new = a1 + s * (self.a2 - a2_new)
106
107             new_b = self.compute_b(E1, a1, a1_new, a2_new, k11, k12,
                ↪  k22, y1)
108
109             delta_b = new_b - self.b
110
111             self.b = new_b
112
113             # Equation 22
114             if self.use_linear_optim:
115                 self.w = self.w + y1*(a1_new - a1)*X1 \
116                     + self.y2*(a2_new - self.a2) * self.X2
117
118             # Update the error cache using the new Lagrange
                ↪  multipliers.
119             delta1 = y1 * (a1_new - a1)
120             delta2 = self.y2 * (a2_new - self.a2)
121
122             # Update the error cache.
```

```
123             for i in range(self.m):
124                 if 0 < self.alphas[i] < self.C:
125                     self.errors[i] += delta1 * self.kernel(X1,
                        ↪  self.X[i]) + \
126                         delta2 * self.kernel(self.X2,self.X[i]) \
127                         - delta_b
128             self.errors[i1] = 0
129             self.errors[i2] = 0
130             self.alphas[i1] = a1_new
131             self.alphas[i2] = a2_new
132 
133             return True
134 
135     def compute_b(self, E1, a1, a1_new, a2_new, k11, k12, k22, y1):
136         # Equation 20
137         b1 = E1 + y1 * (a1_new - a1) * k11 + \
138                     self.y2 * (a2_new - self.a2) * k12 + self.b
139 
140         # Equation 21
141         b2 = self.E2 + y1 * (a1_new - a1) * k12 + \
142                     self.y2 * (a2_new - self.a2) * k22 + self.b
143 
144         if (0 < a1_new) and (self.C > a1_new):
145             new_b = b1
146         elif (0 < a2_new) and (self.C > a2_new):
147             new_b = b2
148         else:
149             new_b = (b1 + b2) / 2.0
150 
151         return new_b
152 
153     def get_error(self, i1):
154         if 0 < self.alphas[i1] < self.C:
155             return self.errors[i1]
156         else:
157             return self.output(i1) - self.y[i1]
158 
159     def second_heuristic(self, non_bound_indices):
160         i1 = -1
161 
162         if len(non_bound_indices) > 1:
163             max = 0
164             for j in non_bound_indices:
165                 E1 = self.errors[j] - self.y[j]
166 
167                 step = abs(E1 - self.E2) # approximation
168                 if step > max:
169                     max = step
170                     i1 = j
171 
```

```
172         return i1
173
174     def examine_example(self, i2):
175         self.y2 = self.y[i2]
176         self.a2 = self.alphas[i2]
177         self.X2 = self.X[i2]
178         self.E2 = self.get_error(i2)
179
180         r2 = self.E2 * self.y2
181
182         if not((r2 < -self.tol and self.a2 < self.C) or
183             (r2 > self.tol and self.a2 > 0)):
184             # The KKT conditions are met, SMO looks at another
                ↪  example.
185             return 0
186
187         # Second heuristic A: choose the Lagrange multiplier which
188         # maximizes the absolute error.
189         non_bound_idx = list(self.get_non_bound_indexes())
190         i1 = self.second_heuristic(non_bound_idx)
191
192         if i1 >= 0 and self.take_step(i1, i2):
193             return 1
194
195         # Second heuristic B: Look for examples making positive
196         # progress by looping over all non-zero and non-C alpha,
197         # starting at a random point.
198         if len(non_bound_idx) > 0:
199             rand_i = randrange(len(non_bound_idx))
200                 for i1 in non_bound_idx[rand_i:] +
                    ↪  non_bound_idx[:rand_i]:
201                     if self.take_step(i1, i2):
202                         return 1
203
204         # Second heuristic C: Look for examples making positive
            ↪  progress
205         # by looping over all possible examples, starting at a random
206         # point.
207         rand_i = randrange(self.m)
208         all_indices = list(range(self.m))
209
210         for i1 in all_indices[rand_i:] + all_indices[:rand_i]:
211             if self.take_step(i1, i2):
212                 return 1
213
214         # Extremely degenerate circumstances, SMO skips the first
            ↪  example.
215         return 0
216
217     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
```

The following code demonstrates how to instantiate a SmoAlgorithm object, run the algorithm, and print the result.

Listing 12.2: Using The SMO Algorithm

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

```

```
6  def linear_kernel(x1, x2):
7      return np.dot(x1, x2)
8
9  def compute_w(multipliers, X, y):
10     return np.sum(multipliers[i] * y[i] * X[i] for i in
       ↪  range(len(y)))
11
12 if __name__ == '__main__':
13     seed(42) # to have reproducible results
14
15     X_data, y_data =
       ↪  get_dataset(linearly_separable.get_training_examples)
16
17     smo = SmoAlgorithm(X_data, y_data, C=10, tol=0.001,
18                 kernel=linear_kernel, use_linear_optim=True)
19
20     smo.main_routine()
21     w = compute_w(smo.alphas, X_data, y_data)
```

Bibliography

[Abu2012] Abu-Mostafa, Y. S. (2012). *Learning From Data. A MLBook*.

[Biernat2016] Biernat, E., & Lutz, M. (2016). *Data science: fondamentaux et études de cas. Eyrolles*.

[Bishop2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning. Springer*.

[Boyd2004] Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization. Cambridge University Press*.

[Burges1988] Burges, C. J. (1988). *A Tutorial on Support Vector Machines for Pattern. Data Mining and Knowledge Discovery, 121-167*.

[Crammer2001] Crammer, K., & Singer, Y. (2001). *On the Algorithmic Implementation of Multiclass Kernelbased Vector Machines. Journal of Machine Learning Research 2*.

[Cristianini2000] Cristianini, N., & Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines. Cambridge University Press*.

[Dogan2011] Dogan, U., Glasmachers, T., & Igel, C. (2011). *Fast Training of Multi-Class Support Vector Machines*.

[Ghaoui2015] El Ghaoui, L. (2015). *Optimization Models and Applications*. Retrieved from http://livebooklabs.com/keeppies/c5a5868ce26b8125

[Gershwin2010] Gershwin, S. B. (2010). *KKT Examples*. Retrieved from MIT Mechanical Engineering Course: http://ocw.mit.edu/courses/mechanical-engineering/2-854-introduction-to-manufacturingsystems-fall-2010/lecture-notes/MIT2_854F10_kkt_ex.pdf

[Gretton2016] Gretton, A. (2016, 03 05). *Lecture 9: Support Vector Machines*. Retrieved from http://www.gatsby.ucl.ac.uk/ gretton/coursefiles/Slides5A.pdf

[Han2012] Han, X., & Berg, A. C. (2012). *DCMSVM: Distributed Parallel Training For Single-MachineMulticlass Classifiers.*

[Hsu2002] Hsu, C.-W., & Lin, C.-J. (2002). *A Comparison of Methods for Multiclass Support Vector Machines. IEEE transactions on neural networks.*

[Hsu2016] 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] Ng, A. (n.d.). *CS229 Lecture notes - Part V Support Vector Machines.* Retrieved from http://cs229.stanford.edu/notes/cs229-notes3.pdf

[Osuna1997] Osuna, E., Freund, R., & Girosi, F. (1997). *An Improved Training Algorithm for Support Vector.* Proceedings of IEEE NNSP'97.

[Platt1998] Platt, J. C. (1998). *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines.* Microsoft Research.

[Platt2000] Platt, J. C., Cristianini, N., & Shawe-Taylor, J. (2000). *Large margin DAGs for multiclass classification.* MIT Press.

[Rojas1996] Rojas, R. (1996). *Neural Networks: A Systematic Introduction.* Springer.

[Russell2010] Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach.* Pearson.

[Tyson2004] Tyson Smith, B. (2004). *Lagrange Multipliers Tutorial in the Context of Support Vector Machines.* Newfoundland.

[Vapnik1982] Vapnik, V. (1982). *Estimation of Dependences Based on Empirical Data.* Springer.

[Vapnik1998] Vapnik, V. N. (1998). *Statistical Learning Theory.* Wiley.

[Weston1999] Weston, J., & Watkins, C. (1999). *Support Vector Machines for Multi-Class Pattern Recognition* Proceedings of the Seventh European Symposium on Artificial Neural Networks.