KERNEL METHODS WITH MIXED DATA TYPES AND THEIR APPLICATIONS

ARNAU ARQUÉ MARTÍNEZ

Thesis supervisor: LUIS ANTONIO BELANCHE MUÑOZ (Department of Computer Science)

Degree: Master's degree in data science

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

26/06/2023

Abstract

Support Vector Machines (SVMs) represent a category of supervised machine learning algorithms that find extensive application in both classification and regression tasks. In these algorithms, kernel functions are responsible for measuring the similarity between input samples to generate models and perform predictions.

In order for SVMs to tackle data analysis tasks involving mixed data, the implementation of a valid kernel function for this purpose is required. However, in the current literature, we hardly find any kernel function specifically designed to measure similarity between mixed data. In addition, there is a complete lack of significant examples where these kernels have been practically implemented. Another notable characteristic of SVMs is their remarkable efficacy in addressing high-dimensional problems. However, they can become inefficient when dealing with large volumes of data.

In this project, we propose the formulation of a kernel function capable of accurately capturing the similarity between samples of mixed data. We also present an SVM algorithm based on Bagging techniques that enables efficient analysis of large volumes of data. Additionally, we implement both proposals in an updated version of the successful SVM library LIBSVM. Moreover, we evaluate their effectiveness, robustness and efficiency, obtaining promising results.

Agraïments (Acknowledgements)

M'agradaria començar agraint al director d'aquest projecte, Lluís Belanche, el seu guiatge durant el desenvolupament del treball. A més a més, agrair a vosaltres, família; pare, mare i Martina, el vostre suport incondicional durant aquest llarg procés. A tu, Míriam, per ser un dels meus pilars fonamentals. Per acabar, amb especial record i estima,

Sempre a tu, iaia.

Contents

1 Introduction **4**
1.1 Identification of the problem 4
1.2 Objectives and requirements 5
1.3 Tasks definition 6
1.4 Planification 8
1.5 Obstacles and risks 8
1.6 Project structure 9

2 Support Vector Machines and Kernel Functions **10**
2.1 Major Events in Machine Learning and Support Vector Machines 10
2.2 Support Vector Machines formulation 11
2.3 Kernel Functions 13

3 Kernels on mixed data **17**
3.1 Data types and Kernel Functions in modern problems 17
3.2 The Aggregation Kernel 22

4 Support Vector Machines on large problems **24**
4.1 Efficiency improvements in Support Vector Machines 24
4.2 The Bagging Support Vector Machine 25

5 Foundations of the Implementation **29**
5.1 Support Vector Machines implementations 29
5.2 Analysis of LIBSVM 31
5.3 Implementation of the Aggregation Kernel 33
5.4 Implementation of the Bagging-SVM 42
5.5 Other tools 43

6 Experimentation **45**
6.1 Experiment 1: Basic functionality 45
6.2 Experiment 2: Execution time overhead 49
6.3 Experiment 3: Bagging-SVM use case 52
6.4 Experiment 4: Mixed data real-world problem 55

7 Conclusions **61**
7.1 Review of the objectives 61
7.2 Analysis and interpretation of the experiments 63
7.3 Overall analysis and conclusions 64
7.4 Future work 65
7.5 Personal thoughts and assessments 66

List of Figures

1.1 Planification of the project. 8

2.1 Examples of hyperplanes that separate data samples of different classes. 11
2.2 Non-linearly separable data examples. 14

3.1 Segmentation of circular values in the $(0, 1]$ range. 21

4.1 Diagram of SVM model generation using the conventional algorithm. 27
4.2 Diagram of SVM model generation using the Bagging approach. 28

5.1 Graphical representation of the `svm_problem` of LIBSVM^{aggr}. 39

6.1 Example of synthetic 2D data generated from the circular data generator script. . 46
6.2 Graphical representation of the datasets used in Experiment 1. 46
6.3 Average execution time obtained in Experiment 2 using the Linear kernel. 50
6.4 Average execution time obtained in Experiment 2 using the Polynomial kernel. . 50
6.5 Average execution time obtained in Experiment 2 using the Linear kernel. 51
6.6 Execution time and error rates obtained in Experiment 3. 53
6.7 Execution time and error rates obtained in Experiment 3 (range $[1, 50]$). 54
6.8 Distribution of `wingspan`, `length` and `height` features of the Aircraft Dataset. . 56
6.9 Distribution of `serviceCeiling` variable of the Aircraft Dataset. 57
6.10 Distribution of `cannons` and `machineGuns` features of the Aircraft Dataset. 57
6.11 Correlation between numerical features of the Aircraft Dataset. 58
6.12 Value counts of the `country` and `type` features of the Aircraft Dataset. 59

List of Tables

6.1 Parametrizations used for each kernel in Experiment 1. 47
6.2 Experiment 1 Confusion Matrices (Polynomial kernel with $d = 3,\ \gamma = 0.1,\ c_0 = 10$). 47
6.4 Experiment 1 Confusion Matrices (RBF kernel with $\gamma = 0.01$). 47
6.6 Results of Experiment 1 using the Linear Kernel. 48
6.7 Results of Experiment 1 using the Polynomial Kernel. 48
6.8 Results of Experiment 1 using the RBF Kernel. 49
6.9 Percentage of additional time required by LIBSVM^{aggr} compared to LIBSVM. . . 51

List of Listings

5.1 New `svm_node` structure of LIBSVM^{aggr} library. 34
5.2 Derived class for data stored as an integer in LIBSVM^{aggr} library. 35
5.3 `kernel_params` pure virtual structure of LIBSVM^{aggr}. 36
5.4 Data structure to store the parameters of the Polynomial kernel. 36
5.5 Summary of the new header of the `svm_parameter` structure. 37
5.6 Header of the new `feature_set` data structure. 38
5.7 Header of the modified `svm_problem` data structure. 38

6.1 Experiment 4 configuration file using textual features. 59

1 Introduction

This document represents the master's thesis report titled "Kernel Methods with Mixed Data Types and Their Applications." The project was conducted at the Polytechnic University of Catalonia, specifically at the Faculty of Informatics in Barcelona, as part of the Master's program in Data Science.

The first section aims to provide an introduction to the project's topic. We will begin by identifying the problem that motivated us to undertake this research. Subsequently, we will state the objectives, sub-objectives, as well as the functional and non-functional requirements that we aim to address in solving the problem.

Next, we will present the proposed plan to achieve the objectives within the established timeframe, outlining the tasks that will enable us to do so. Following that, we will analyze the potential obstacles and risks that may hinder the project's development. Finally, we will outline the project's structure to ensure coherence in explanations and procedures.

1.1 Identification of the problem

In the field of data science, we are increasingly confronted with diverse problems. The large volumes of data generated by society and the tools we have implemented for their collection and processing have been key factors in this phenomenon [1]. In these modern problems, we encounter a wide variety of data types, ranging from numerical and binary to categorical variables, among many others. In recent years, thanks to advancements in machine learning algorithms, we have begun efficiently working with texts, images and videos.

One of the most widely used algorithms in the field of machine learning is Support Vector Machines (SVM) [2]. SVMs are supervised learning algorithms that solve classification and regression problems. These algorithms reach a solution using, among other elements, kernel functions that measure the similarity between input samples [3].

The first problem to highlight is the almost completely lack of references in the current literature to kernel functions that can handle mixed data, which refers to samples composed of features of different types. According to our knowledge, we have only come across the proposal by Daemen *et al.*, who formulates a kernel function for clinical data. However, we consider that the proposal is rather straightforward and is only designed to handle continuous and binary data [4]–[6]. This absence of kernel functions that adequately capture the degree of similarity between mixed samples poses a significant problem, leaving SVM algorithms in a weakened position when facing modern problems. Not only are there no formulations of kernel functions that can handle such data types, but there are also no available tools that implement them.

It is worth noting that SVMs are exceptional algorithms in several aspects. One of them is their efficiency, unlike, for example, Neural Networks, in handling samples with a large number of features (high dimensionality). However, SVMs encounter performance issues when dealing with large volumes of data. This is because the complexity of their formulation primarily depends on the number of input samples rather than the dimensionality of these samples [7].

In the current literature, we find some algorithm proposals aiming to address this problem [8]–[11]. However, some of these proposals, despite improving efficiency in handling large volumes of data, present significant drawbacks such as increased prediction error or high complexity in implementing the techniques. In some cases, the theoretical aspects of the techniques are formulated without providing any open-source code implementation.

Considering the aforementioned challenges, we believe that there are two significant gaps in the current data science literature regarding SVMs. Firstly, there is a lack of effective formulations and implementations of kernel functions to address problems based on mixed data. Secondly, we consider that there is room for improvement in techniques that facilitate solving problems with large volumes of data using SVMs. Therefore, in this project, our focus is on formulating theoretical solutions to address these issues and providing simple, effective, scalable and open-source implementations to tackle these challenges.

1.2 Objectives and requirements

Having reached this point, we have been able to identify the inherent challenges in the field of Data Science and Machine Learning that have motivated us to develop this project. Based on these challenges, in this section, we will begin by formulating the main objectives of this project. Finally, we will state the functional and non-functional requirements that the proposed solutions must meet.

The main objective of this project is to formulate and provide a functional implementation of a valid kernel function that is capable of analyzing mixed data samples in a way that accurately captures the degree of similarity between them. Consequently, this kernel function can be effectively utilized in Support Vector Machine algorithms, among others.

As a secondary objective, we aim to theoretically define an alternative to the conventional Support Vector Machine algorithm that can efficiently handle problems involving large quantities of data while maintaining a similar error rate. Furthermore, we look forward to providing an effective implementation of this algorithm that validates our hypotheses.

Functional requirements

Once the project objectives have been defined, we believe it is necessary to state the functional requirements that our solutions must satisfy. These requirements are aimed at establishing the constraints that need to be met regarding the theoretical proposals, the training and prediction capabilities of the algorithms and the evaluation of the obtained results.

1. Firstly, it is necessary for the proposed kernel function to represent a **valid kernel** and this fact should be demonstrated based on the definition and properties inherent to kernels.
2. Additionally, the proposed alternative algorithm for utilizing Support Vector Machines (SVMs) in problems involving large quantities of data must be **theoretically valid** and adhere to the fundamental principles of these algorithms.
3. In terms of **training capacity**, the solutions should allow for the processing and analysis of mixed data through the development and adjustment of models based on SVMs.
4. Regarding **prediction capability**, the project proposals must provide valid predictions based on the Machine Learning models generated during the training phase.
5. Lastly, the implemented procedures should facilitate **metrics and tools** that enable the evaluation of model performance in a simple, intuitive and reliable manner.

Non-functional requirements

To conclude, in this section, we will state the non-functional requirements that arise from the previously proposed objectives. These requirements establish constraints not directly related to the functionality of our solutions, such as performance, usability and robustness, among others.

1. The primary non-functional requirement is the **efficiency** of the proposed solutions. Both the algorithm proposal for handling large datasets and the inclusion of new kernel function formulations should demonstrate good performance in terms of execution time and computational resources.
2. Another important requirement is the **usability** of the solutions. The implemented tools should be intuitive, user-friendly and have a clear interface that allows users in the Data Science community to use them without difficulties.
3. In terms of **scalability**, the new algorithms should be capable of working with large volumes of data without sacrificing other fundamental aspects such as efficiency and accuracy in problem solving.
4. Lastly, it will be necessary for the proposed solutions in this project to be easily **manipulable**. This will ensure that users from any field can extend and complement the proposals by, among others, adding new kernel functions and support for new types of data in a simple, unrestricted and effective manner.

1.3 Tasks definition

At this point, we have introduced the project and identified the problem that has motivated its development. Next, we have defined the project's objectives and subobjectives and stated the functional and non-functional requirements. In this section, we will design the set of tasks that will help us achieve the previously established objectives.

Prior to any process or activity associated with a project, it is necessary to carry out a research stage. Therefore, the first group of tasks we will undertake is Research (**R**). The research tasks are oriented towards activities such as state-of-the-art analyses or exhaustive investigations. Through these tasks, we will conduct a thorough investigation of existing SVM libraries and kernel functions developed for mixed variables. This research will provide us with the necessary theoretical basis for the progress of the project and allow us to understand the current limitations and opportunities for improvement in this area. We will divide the research tasks into five distinct subtasks:

- **R1** Study and analyze of the fundamentals of Support Vector Machines algorithms and kernel functions.
- **R2** Examine and assess the state-of-the-art concerning the types of variables present in the current literature and the corresponding implemented kernel functions.
- **R3** Study of kernel functions from the current literature that are oriented towards handling mixed data.
- **R4** Investigate and analyze the current state-of-the-art of Support Vector Machines algorithms designed for handling large-scale datasets as documented in the existing literature.
- **R5** Examine and analyze the state-of-the-art regarding efficient implementations of Support Vector Machines algorithms found in the current literature.

Once the research stage is complete, it will be time to initiate a process of study and design of the techniques that we intend to implement. Therefore, we have decided to define a set of Design (**D**) tasks, which are divided into three subtasks:

D1 Design and formulate kernel functions for different types of variables.

D2 Design and formulate a kernel function that allows us to measure similarity between mixed data.

D3 Design and formulate a Support Vector Machines algorithm capable of efficiently addressing problems involving large volumes of data.

At this point, ideally, we will have gathered the necessary information to design and formulate the techniques that correspond to the previously established objectives. Next, it will be necessary for us to provide a functional implementation of these methodologies. For this reason, we will define a set of Implementation (**I**) tasks, which can be broken down into the following subactivities:

I1 Incorporate support for mixed data into a renowned Support Vector Machines library widely recognized in the current literature.

I2 Implement kernel functions for various types of variables within the library..

I3 Incorporate the previously designed kernel function into the software to handle mixed data vectors.

I4 Integrate the Support Vector Machines-based algorithm capable of efficiently handling large quantities of data.

Subsequently, once we have studied, designed and implemented the methodologies proposed for our objectives, we believe it is necessary to conduct a series of experiments to evaluate their correctness, robustness and efficiency, among other factors. Therefore, we will define the set of Experimentation (**E**) tasks, which include:

E1 Validate the fundamental functionality of the implemented techniques and methodologies.

E2 Apply the mixed data kernel function to solve a real-world problem.

E3 Apply the Support Vector Machines algorithm to handle large quantities of data in a specific use case to validate its effectiveness and efficiency.

Finally, the Documentation tasks will allow us to compile and document the entire process and the results obtained throughout the project. This task is essential for the creation of the final report, which will comprehensively reflect the work carried out, the methodologies used, the results obtained and the conclusions reached.

Through this organization into task groups, we establish a clear and logical structure to address the project's objectives, ensuring adequate progression and efficiency in their achievement. Each group of tasks is closely related to the corresponding objectives, thus ensuring a close association between the tasks performed and the expected objectives to be achieved.

1.4 Planification

Given the magnitude of the project, we have decided to make an estimate of the time it will take us to carry out each of the tasks defined above. In this way, it will be easier for us to organize ourselves and mitigate the possible obstacles that we may encounter during the development of the work.

You can see an approximation of the duration of each group of tasks in Figure 1.1. The estimated time for the Research, Design + Implementation and Experimentation tasks are indicated in yellow, green and blue, respectively. Notice that we have combined the design and implementation tasks as they are closely related and it is challenging to estimate the individual time required for each one.

It is planned that the Documentation of the project will be carried out continuously during its development. Even so, we have planned to reserve the last two weeks of the project entirely for the writing of the final report (gray).

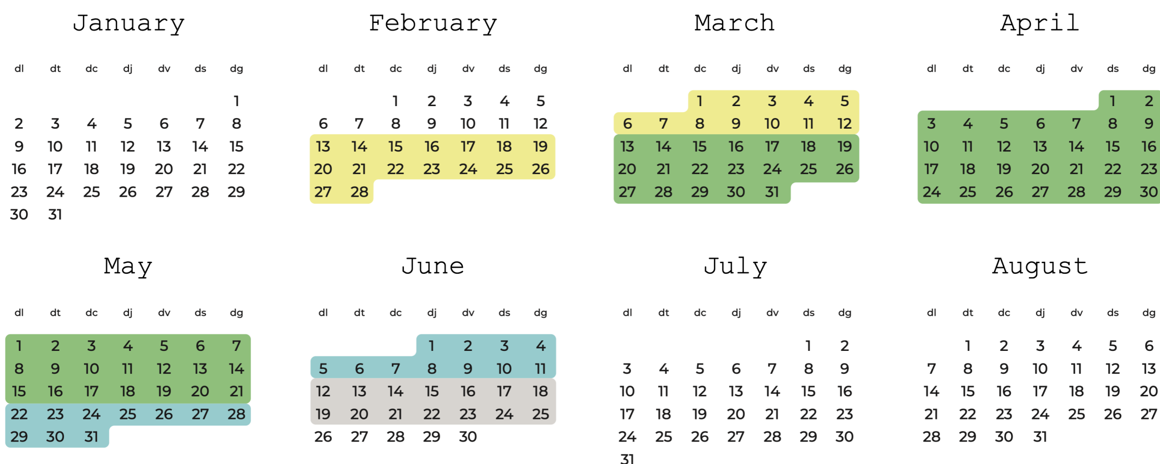


Figure 1.1: Planification of the project.

1.5 Obstacles and risks

During the development of large-scale projects in the field of data science and machine learning, it is inevitable to encounter various obstacles and assume certain risks. As responsible authors, it is vital to be aware of them in order to find effective solutions if necessary.

Firstly, we highlight the potential lack of computational resources. It is possible that the devices we are working with may not be powerful enough to perform the required tasks, resulting in high computation times. This limitation can negatively impact the overall performance of the project, delaying progress and increasing the complexity of the tasks to be performed.

Furthermore, it is important to consider the possibility of unforeseen issues related to the devices used in the project's resolution. Therefore, it would be advisable to have viable alternatives or contingency plans in case of equipment failure or unavailability.

It should be noted that, despite having previous experience in the field of SVMs and kernel functions, the specific challenges of this project may place us in a position of limited expertise in this area. This lack of experience can generate additional difficulties during project execution, requiring greater dedication to acquire the necessary knowledge.

It is also crucial to bear in mind that, when working on data science projects, maintaining an ethical perspective and respecting the privacy of the data used is paramount. As data scientists, it is essential to consider and address gender, race or nationality biases that may exist in the data to prevent resulting algorithms from being discriminatory or perpetuating unfair inequalities.

In the theoretical resolution of data science problems, delays in planning may arise due to the inherent complexity of the project. This complexity may require additional time to properly understand and address specific issues that arise, which can impact the overall development of the project.

Lastly, it is important to emphasize that project deadlines and presentations are fixed and inflexible. This implies the need for rigorous control of planning at all times and the ability to overcome setbacks and adversities that may arise throughout the process.

By considering these potential obstacles and risks, we will be prepared to face the difficulties that may arise during the project's development and will be better equipped to find effective solutions to achieve the proposed objectives.

1.6 Project structure

At this point, we can consider the introduction to the project concluded. In this section, we will discuss the structure followed in this report.

Firstly, we will briefly review the historical background in the field of data science and machine learning that led to the definition and development of Support Vector Machines (SVM) algorithms. Next, we will analyze the main characteristics of SVMs and the role that kernel functions play in them.

Next, we will study the main types of variables found in current literature and the implemented kernel functions for each type. Subsequently, we will analyze, define and formulate a kernel function to handle mixed data.

Following that, we will explore the techniques found in the current literature that are oriented towards handling large quantities of data using Support Vector Machine algorithms. Then, we will propose a new algorithm motivated by the same objective, formulate it and study its complexity.

At this stage, we will study the leading libraries in the field of SVMs found in current literature. Based on this analysis, we will select one of them and expand it to include support for the techniques studied earlier. Once the implemented techniques are available, we will begin an experimentation phase to test their correctness, effectiveness and efficiency on synthetic and real-world problems.

Finally, we will review the objectives set forth in this section, analyze and interpret the results of the experiments and provide an overall conclusion of the project. We will also estimate possible future work and offer a personal assessment of the work accomplished.

2 Support Vector Machines and Kernel Functions

In this chapter, we will briefly review some of the main events in the history of machine learning, Kernel Methods and the early developments of Support Vector Machines (SVM). Subsequently, we will analyze the principles underlying this type of algorithms.

Next, we will study the advantages and drawbacks arising from their application to real-world problems. Once this introduction to SVMs is completed, we will describe the mathematical formulations that underlie them. Finally, we will analyze the role of kernel functions in this procedure.

2.1 Major Events in Machine Learning and Support Vector Machines

In this section, we will explore the development and background of the field of Machine Learning and Support Vector Machines. We will begin by reviewing the key events that have laid the foundation for Support Vector Machines (SVMs).

In 1950, Alan Turing created the Turing Test, establishing the foundations of modern computing and introducing the concept of a computer [12]. Over the following years, computing underwent remarkable advancements, largely driven by the constant increase in computational and storage capabilities of the computer systems of that era.

In the realm of algorithms, Arthur Samuel started working for IBM in 1949. In 1959, he published the first algorithm capable of autonomous learning [13]. This algorithm focused on the computer's ability to win in the game of checkers without any external intervention. Samuel envisioned the potential of teaching computers through games to develop techniques that could be applicable to broader problems. This groundbreaking algorithm, now known as Alpha-Beta Pruning, marked a turning point in the fields of algorithms and Machine Learning. In the same article, Samuel introduced the term "Machine Learning" for the first time in history, defining it as "the field of study that gives computers the ability to learn without being explicitly programmed".

The subsequent years were marked by hope and expectations regarding Machine Learning and Artificial Intelligence. Unfortunately, those expectations did not materialize and a period known as "The AI Winter" ensued [14]. During this time, investment and development in these fields significantly declined.

However, in 1990, there was a significant resurgence of optimism in the field of Machine Learning, thanks to notable contributions in this domain [15]. In fact, in 1992, Boser *et al.* developed a novel Machine Learning algorithm at AT&T laboratories, known as Support Vector Machines (SVMs) [3]. SVMs are a set of supervised learning algorithms that allow solving classification and regression problems by analyzing datasets.

These powerful tools are applicable to both classification and regression problems, among others. In classification, SVMs aim to generate a hyperplane that adequately separates samples from each class. In some cases, this separation is not feasible in the original data space and it becomes necessary to generate the hyperplane in a higher-dimensional space using kernel functions [16]. Regarding regression problems, the solution also involves generating a hyperplane, but in this case, the goal is to achieve the best possible fit of the data in a continuous space.

In the following sections, we will delve into the formulation and analysis of Support Vector Machines, as well as the crucial role played by kernel functions in these algorithms. Please note that from now on, we will use ***bold*** notation to refer to the vectors appearing in the equations.

2.2 Support Vector Machines formulation

At this point, we have provided a brief introduction to the history of Machine Learning and the beginnings of Support Vector Machines. However, in order to work with and manipulate these types of algorithms—which is precisely one of the objectives of this project—we need to understand and break down the mathematical concepts that underpin them.

The main objective of a classifier is to identify the class to which the different instances in the problem belong. As mentioned before, in the realm of SVMs, this identification is achieved by generating a hyperplane that separates the instances of each class. However, in many cases, there are infinitely many different hyperplanes that can correctly separate the data, as depicted in Figure 2.1. In addition to generating this hyperplane, the goal of SVMs is to maximize the distance between the plane and the closest samples. This distance is called the margin, while the samples closest to the hyperplane (which also define the margin) are called Support Vectors (SV).

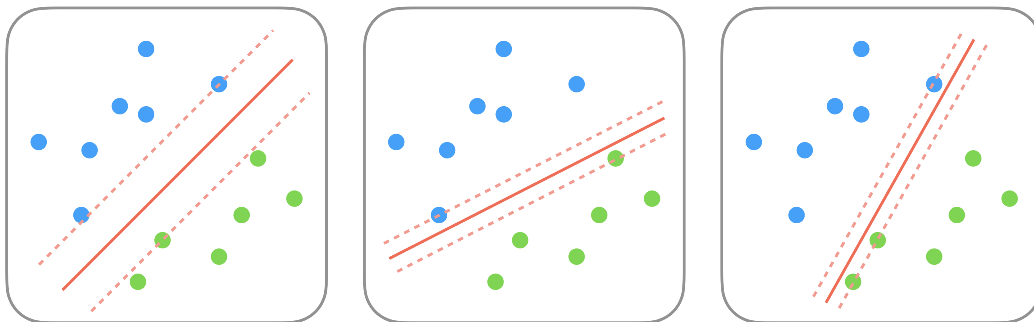


Figure 2.1: Examples of hyperplanes that separate data samples of different classes.

Source: Own elaboration.

Let $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_d)$ and $b = \omega_0$ (bias). We can define a hyperplane as:

$$\pi \ : \ \omega^\intercal x + b = 0 \tag{2.1}$$

In order to ensure a unique solution, it will be required for the vector $\boldsymbol{\omega}$ to be normalized. This way, the constraint stated in equation 2.2 will be satisfied for the vectors $\boldsymbol{x^n} \in \mathbb{R}^d$ that are closest to the plane (support vectors, SVs).

$$|\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b| = 1 \tag{2.2}$$

To maximize the distance between the points close to the plane and the plane itself, we need

to determine how to calculate this distance. Let $\boldsymbol{x} \in \pi$ be a point on the plane and $\hat{\boldsymbol{\omega}} = \boldsymbol{\omega}/\|\boldsymbol{\omega}\|$ be the unit vector. We know that the distance between an SV $\boldsymbol{x^n}$ and the plane π is given by:

$$\text{dist}(\boldsymbol{x^n}, \pi) = |\hat{\boldsymbol{\omega}}(\boldsymbol{x^n} - \boldsymbol{x})| = \frac{1}{\|\boldsymbol{\omega}\|} \cdot |(\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b) - (\boldsymbol{\omega}^\intercal \boldsymbol{x} + b)|$$

Considering the definition of the plane (equation 2.1) and the restrictions imposed on $\boldsymbol{x^n}$ (equation 2.2), we obtain that the distance between $\boldsymbol{x^n}$ and the plane becomes:

$$\text{dist}(\boldsymbol{x^n}, \pi) = \frac{1}{\|\boldsymbol{\omega}\|} \cdot |1 - 0| = \frac{1}{\|\boldsymbol{\omega}\|}$$

This distance is, in essence, the width of the margin separating the support vectors from the plane (see Figure 2.1). Considering that the main objective of SVMs is to maximize this margin, the above equation becomes the objective function (OF) of the following optimization problem (OP):

$$\begin{aligned} \max \quad & \frac{1}{\|\boldsymbol{\omega}\|} \\ \text{subject to} \quad & \min_{n=1\ldots N} |\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b| = 1 \end{aligned} \tag{2.3}$$

Where N is the number of samples in the problem. To simplify the subsequent steps, we will use the following optimization problem (OP), which is equivalent to the statement in equation 2.3:

$$\min \quad \frac{1}{2}\boldsymbol{\omega}^\intercal \boldsymbol{\omega} \tag{2.4}$$

$$\text{subject to} \quad y^n(\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b) \geq 1, \ \text{for } n = 1 \ldots N \tag{2.5}$$

Where $y^n \in \{-1, \ 1\}$ is the class label associated with instance $\boldsymbol{x^n}$. To solve this OP, we will use its Lagrange formulation. We will start by converting constraint 2.5 into its zero form, so it becomes $y^n(\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b) - 1 \geq 0$. Then, we will introduce a non-negative Lagrange multiplier α^n to ensure that the constraint is effectively greater than or equal to zero. Finally, we will add the constraint as part of the objective function. These modifications result in the following equivalent OP:

$$\min_{\boldsymbol{\omega}, \ b} \quad \mathcal{L}(\boldsymbol{\omega}, b, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\omega}^\intercal \boldsymbol{\omega} - \sum_{n=1}^{N} \alpha^n (y^n(\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b) - 1) \tag{2.6}$$

At this point, we could already solve the problem. However, current SVM implementations further simplify the resolution process by using what is known as the *dual form.* This equivalent formulation is based on finding the hyperplane using an OP that only depends on $\boldsymbol{\alpha}$.

To achieve this new form, we need to consider the constraints we will state below. Since we are ultimately looking for maxima and minima, it must hold that both the gradient of the Lagrangian (w.r.t. $\boldsymbol{\omega}$) and the derivative of the Lagrangian (w.r.t. b) are zero:

$$\nabla_{\boldsymbol{\omega}} \mathcal{L} = 0 \iff \boldsymbol{\omega} - \sum_{n=1}^{N} \alpha^n y^n \boldsymbol{x^n} = 0 \iff \boldsymbol{\omega} = \sum_{n=1}^{N} \alpha^n y^n \boldsymbol{x^n} \tag{2.7}$$

$$\frac{\delta \mathcal{L}}{\delta b} = 0 \iff \sum_{n=1}^{N} \alpha^n y^n = 0 \tag{2.8}$$

By substituting these constraints into the Lagrangian in equation 2.6, we obtain a new Lagrangian that depends only on $\boldsymbol{\alpha}$:

$$\mathcal{L}(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha^n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} y^n y^m \alpha^n \alpha^m (\boldsymbol{x^n})^\intercal \boldsymbol{x^m} \tag{2.9}$$

From this new Lagrangian, we can redefine the optimization problem (OP) in equation 2.6 using the constraints we have determined, including the constraint in equation 2.8:

$$\min_{\boldsymbol{\alpha}} \quad \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} y^n y^m \alpha^n \alpha^m (\boldsymbol{x^n})^\intercal \boldsymbol{x^m} - \sum_{n=1}^{N} \alpha^n \tag{2.10}$$

$$\text{subject to} \quad \alpha^n \geq 0, \text{ for } n = 1 \dots N \quad \text{and} \tag{2.11}$$

$$\sum_{n=1}^{N} \alpha^n y^n = 0 \tag{2.12}$$

This optimization problem can be solved using Quadratic Programming (QP). The solution to the problem will be the vector $\boldsymbol{\alpha} = (\alpha^1 \dots \alpha^N)$. However, we should not forget that our ultimate goal is still to obtain the hyperplane $\boldsymbol{\omega}^\intercal \boldsymbol{x} + b = 0$ that separates the data points of different classes. Therefore, to obtain $\boldsymbol{\omega}$ from $\boldsymbol{\alpha}$, we can use the equation 2.7 derived from the Lagrangian and finally, we can isolate b using the constraint 2.2 established at the beginning of the explanation. Note that only those points $\boldsymbol{x^n} \in \text{SV}$ will have an associated $\alpha^n > 0$. For the rest of the points $\boldsymbol{x^m} \notin \text{SV} \implies \alpha^m = 0$. Thus, we can simplify equation 2.7 to:

$$\omega = \sum_{\boldsymbol{x^n} \in \text{SV}} \alpha^n y^n \boldsymbol{x^n}$$

2.3 Kernel Functions

If we look at the title or the objectives of this project, we realize that the concept of kernel functions stands out above everything else. However, despite having defined the mathematical foundations of Support Vector Machines (SVM), this term has not been mentioned. Therefore, the question arises: what does a kernel function have to do with the operation of an SVM?

As we have seen in the examples we presented at the beginning of the previous section (Figure 2.1), the data on which SVMs operate are located in a space $\mathcal{X} = \mathbb{R}^d$ of dimensionality d and the goal is to determine a hyperplane that separates the samples in this space. However, what would happen if the data were not linearly separable in $\mathcal{X}$, as shown in the examples in Figure 2.2? This is where kernel functions come into play.

The general idea of kernels lies in the desire to find a higher-dimensional space $\mathcal{Z}$ in which it is possible to separate the data using a hyperplane and moreover, without having to explicitly visit this new space. But how is it possible to work in a space $\mathcal{Z}$ without the need to explicitly access it?

First, we must ensure that access to this new space $\mathcal{Z}$ is carried out through a transformation that depends exclusively on $\boldsymbol{x^n} \in \mathcal{X}$. In other words, there must exist a function $\Phi : \mathcal{X} \to \mathcal{Z}$ such that $\Phi(\boldsymbol{x^n}) = z^n$. Considering this premise, we see that we have defined the new $\mathcal{Z}$ space, but we are still explicitly visiting it through Φ. It is not difficult to realize that this is precisely what we are trying to avoid. For this reason, an additional restriction will be necessary.

Second, we must ensure that all access to the $\mathcal{Z}$ space is done through a scalar product of the form $(\boldsymbol{z^n})^\intercal \boldsymbol{z^m}$, with $\boldsymbol{z^n}, \boldsymbol{z^m} \in \mathcal{Z}$. As stated in the previous paragraph, by construction of $\boldsymbol{z^n}$ and $\boldsymbol{z^m}$, this scalar product can be expressed as $\Phi(\boldsymbol{x^n})^\intercal \Phi(\boldsymbol{x^m})$. In this way, the product

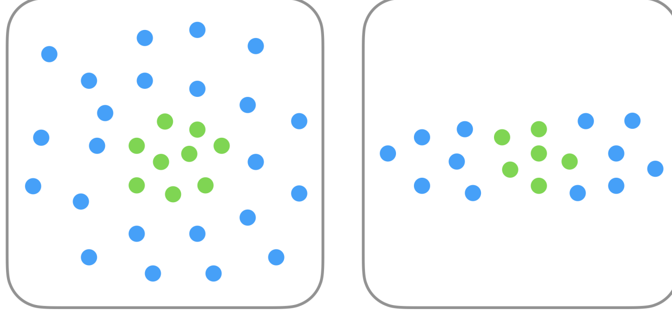


Figure 2.2: Non-linearly separable data examples.

Source: Own elaboration.

becomes an operation that depends exclusively on $\boldsymbol{x^n}, \boldsymbol{x^m} \in \mathcal{X}$. That is, we will be accessing $\mathcal{Z}$ (and thus generating the hyperplane that separates the data in this space) without the need to explicitly visit it.

The expression of the scalar product in the $\mathcal{Z}$ space in terms of $\boldsymbol{x^n}$ and $\boldsymbol{x^m}$ is what we call a *Kernel Function* (KF). These functions are defined as $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, where $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = \Phi(\boldsymbol{x^n})^\intercal \cdot \Phi(\boldsymbol{x^m}) = (\boldsymbol{z^n})^\intercal \boldsymbol{z^m}$.

One of the main properties that arises from the definition of KFs is that it doesn't matter what the initial space $\mathcal{X}$ of the samples is. The only requirement is that the KF, which handles the scalar products in $\mathcal{Z}$, represents a valid kernel. In fact, if we look at the formulation we introduced in the previous section, without KFs, SVMs would only be able to deal with real-valued data (*i.e.* $\mathcal{X} \subseteq \mathbb{R}^d$) that is linearly separable in the input space.

Later on, we will see that it is not necessary to find the expression of Φ to validate a KF. However, before that, let's see how the formulation of SVMs we introduced in the previous section is affected when we introduce the handling of data in the $\mathcal{Z}$ space using kernel functions.

Let's start by revisiting the Lagrangian $\mathcal{L}(\boldsymbol{\alpha})$ (equation 2.10), which represents the objective function (OF) of the optimization problem (OP) that needs to be solved in order to obtain the multipliers α^n:

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{\alpha}) &= \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y^n y^m \alpha^n \alpha^m (\boldsymbol{z^n})^\intercal \boldsymbol{z^m} - \sum_{n=1}^{N} \alpha^n = \\
&= \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y^n y^m \alpha^n \alpha^m k(\boldsymbol{x^n}, \boldsymbol{x^m}) - \sum_{n=1}^{N} \alpha^n
\end{aligned}
$$

In this way, the optimization problem becomes:

$$\begin{aligned}\min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} y^n y^m \alpha^n \alpha^m k(\boldsymbol{x^n}, \boldsymbol{x^m}) - \sum_{n=1}^{N} \alpha^n \\ \text{subject to} \quad & \alpha^n \geq 0, \text{ for } n = 1 \dots N \quad \text{and} \\ & \sum_{n=1}^{N} \alpha^n y^n = 0\end{aligned}$$

As we can see, the constraints of the problem are not affected since they do not depend on the samples in the $\mathcal{X}$ space. Another element of the formulation that will be affected is the computation of $\boldsymbol{\omega}$ (see equation 2.7) based on the values of $\boldsymbol{\alpha}$ obtained in the solution of the previous problem:

$$\omega = \sum_{\boldsymbol{z^m} \in \text{SV}} \alpha^m y^m \boldsymbol{z^m} \tag{2.13}$$

It is evident that we do not have a way to represent $\boldsymbol{\omega}$ as the scalar product of two samples in the $\mathcal{Z}$ space. Therefore, if it were necessary to compute its value, we would have to explicitly visit $\mathcal{Z}$, which is precisely what we are trying to avoid. However, we will see next that it will not be necessary to perform this calculation under any circumstances.

In order to classify a sample $\boldsymbol{x^n} \in \mathcal{X}$, it is only necessary to determine:

$$g(\boldsymbol{x^m}) = \text{sign}(\boldsymbol{\omega}^\intercal \boldsymbol{x^n} + b)$$

Taking into account that we are working in the $\mathcal{Z}$ space, we obtain:

$$\begin{aligned} g(\boldsymbol{z^m}) &= \text{sign}(\boldsymbol{\omega}^\intercal \boldsymbol{z^n} + b) = [\text{considering } 2.13] \\ &= \text{sign}\left(\sum_{\boldsymbol{z^m} \in \text{SV}} \alpha^m y^m (\boldsymbol{z^m})^\intercal \boldsymbol{z^n} + b\right)\end{aligned}$$

Having represented $\boldsymbol{\omega}^\intercal z^n$ as a scalar product in the $\mathcal{Z}$ space, we only need to ensure that b can also be defined in terms of the same terms. Let's see:

$$\begin{aligned} b &= y^n - \boldsymbol{\omega}^\intercal \boldsymbol{z^n} = [\text{considering } 2.13] \\ &= y^n - \sum_{\boldsymbol{z^m} \in \text{SV}} \alpha^m y^m (\boldsymbol{z^m})^\intercal \boldsymbol{z^n}\end{aligned}$$

Therefore, the classification of an instance into a class using kernel functions becomes:

$$g(\boldsymbol{x^m}) = \text{sign}\left(\sum_{\boldsymbol{z^m} \in \text{SV}} \alpha^m y^m k(\boldsymbol{x^m}, \boldsymbol{x^n}) + b\right), \text{ with}$$

$$b = y^n - \sum_{\boldsymbol{z^m} \in \text{SV}} \alpha^m y^m k(\boldsymbol{x^m}, \boldsymbol{x^n})$$

Therefore, it has been demonstrated that we can generate a separating hyperplane in a higher-dimensional space $\mathcal{Z}$ without the need to explicitly visit it. We only need to ensure that the kernel function is valid.

As mentioned earlier, it is not necessary to determine Φ in order to validate and consequently use a kernel function. We can determine the validity of a kernel function using the following methodologies:

1. By construction: We prove that the kernel function $k(\boldsymbol{x^m}, \boldsymbol{x^n})$ can be represented as the dot product of the transformations $\Phi(\boldsymbol{x^m})$ and $\Phi(\boldsymbol{x^n})$.

2. By mathematical properties: Every kernel function $k(\boldsymbol{x^m}, \boldsymbol{x^n})$ will be valid if and only if (1) k is symmetric (*i.e.*, $k(\boldsymbol{x^m}, \boldsymbol{x^n}) = k(\boldsymbol{x^n}, \boldsymbol{x^m})$) and (2) the matrix K (where $K_{ij} = k(\boldsymbol{x^i}, \boldsymbol{x^j})$) is positive semi-definite for all $\boldsymbol{x^1}, \dots, \boldsymbol{x^N}$. This condition is known as *Mercer's condition.* The matrix K is defined such that $K_{ij} = k(\boldsymbol{x^i}, \boldsymbol{x^j})$. This matrix is commonly referred to as the *Kernel Matrix.*

3. By closure properties: We can use the following properties [17] to form new kernel functions from other expressions:
   a) $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = k_1(\boldsymbol{x^n}, \boldsymbol{x^m}) + k_2(\boldsymbol{x^n}, \boldsymbol{x^m})$
   b) $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = a \cdot k_1(\boldsymbol{x^n}, \boldsymbol{x^m})$
   c) $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = k_1(\boldsymbol{x^n}, \boldsymbol{x^m})k_2(\boldsymbol{x^n}, \boldsymbol{x^m})$
   d) $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = f(\boldsymbol{x^n})f(\boldsymbol{x^m})$
   e) $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = k_3(\Phi(\boldsymbol{x^n}), \Phi(\boldsymbol{x^m}))$
   f) $k(\boldsymbol{x^n}, \boldsymbol{x^m}) = (\boldsymbol{x^n})^\intercal \mathrm{B} \boldsymbol{x^m}$

Where $k_1, k_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, $\mathcal{X} \subseteq \mathbb{R}^d$, $a \in \mathbb{R}^+$, $f : \mathcal{X} \to \mathbb{R}$, $\Phi : \mathcal{X} \to \mathbb{R}^M$, $k_3 : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}$ and B is a symmetric positive semi-definite matrix of dimension $d \times d$.

La teoria dels kernels es pot extendre a Hilbert Spaces, que són espais euclidians de dimensionalitat finita o infinita [16].

In addition to enabling us to separate a dataset in a higher-dimensional space, kernel functions can be understood from the perspective of analyzing the similarity between two samples. Often, a scalar product is interpreted as a measure of similarity between two vectors [18]. In our case, the vectors are the data samples in the input space and the scalar product is implicitly carried out through the kernel functions. Hence, the resulting value from these functions also indicates how similar two samples are.

3 Kernels on mixed data

In this chapter, we will study and review some of the most common types of data and variables found in modern problems. We will also explore the existing kernel functions designed to handle these types of variables, if they have been defined. In cases where formulations of kernel functions for specific data types are not found, we will attempt to propose a function that can capture the similarity between samples of the respective variables.

Furthermore, we will analyze the kernel functions tailored to address mixed data found in the current literature. Additionally, we will justify the need to define kernel functions that handle data vectors of different types. Finally, we will state and formulate a new kernel function that will allow us to handle vectors of non-real variables and capture the similarity between this type of data.

3.1 Data types and Kernel Functions in modern problems

Until a few years ago, most problems consisted of simple data types such as continuous numerical, binary or categorical data. However, in recent years, advancements in the field of Machine Learning, its algorithms and the increased computing power of available devices have allowed us to tackle problems with more complex variables. Examples of such variables include circular variables, fuzzy variables, textual variables and even data in the form of images and videos.

In this first section, we will study the different types of data found in modern problems and the kernel functions that enable us to measure similarity between them.

Continuous variables

We will begin by analyzing continuous numerical variables. Continuous numerical variables, also known as real-valued variables, are essential in various fields of study. These types of variables, which take values on the real number line $\mathbb{R}$, represent continuous quantities such as ages, temperatures or income. For example, in a real estate price prediction problem, a real-valued variable could be the surface area of a house. Their utility is widely recognized and numerous studies have been conducted on these variables.

In the current literature, we find multiple kernel functions focused on dealing with data of this type. The most basic of all is the *linear kernel*, whose expression can be found in equation 3.1. Essentially, the linear kernel evaluates the similarity of data points in the original input space using a dot product.

$$k_{linear}(\boldsymbol{x^i}, \boldsymbol{x^j}) = (\boldsymbol{x^i})^\intercal \boldsymbol{x^j} \tag{3.1}$$

Another well-known kernel for continuous data is the *polynomial kernel.* It leverages the expression of the original samples in the form of a polynomial to evaluate their similarity.

As we can see in equation 3.2, this kernel function has three main hyperparameters: $d \in \mathbb{R}^+$ denotes the degree of the polynomial, $c_0 \in \mathbb{R}$ represents its coefficient and $\gamma \in \mathbb{R}$ is a hyperparameter that acts as a multiplier of the dot product of the samples in the input space.

$$k_{poly}(\boldsymbol{x^i}, \boldsymbol{x^j}) = (\gamma(\boldsymbol{x^i})^\intercal \boldsymbol{x^j} + c_0)^d \tag{3.2}$$

It is not difficult to notice that the linear kernel (equation 3.1) is a special case of the polynomial kernel where $d = 1$, $c_0 = 0$ and $\gamma = 1$.

One of the most well-known kernels in the field of numerical data is the *Radial Basis Function* (RBF) kernel. Gaussian kernels are among the most widely used in the literature and have been employed in multiple areas of study [18]. The most common expression of this kernel is shown in equation 3.3.

$$k_{rbf}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \exp\left(-\frac{\left\|\boldsymbol{x^i} - \boldsymbol{x^j}\right\|^2}{2\sigma^2}\right) \tag{3.3}$$

Where the quadratic term $\left\|\boldsymbol{x^i} - \boldsymbol{x^j}\right\|^2$ represents the squared Euclidean distance between the two samples $\boldsymbol{x^i}$ and $\boldsymbol{x^j}$. The parameter $\sigma \in \mathbb{R}^+$ controls the influence that each data point has on the others in the similarity calculation.

An equivalent formulation of the RBF kernel widely used in the literature is shown in equation 3.4. In this case, $\gamma = \frac{1}{2\sigma^2} \in \mathbb{R}^+$.

$$k_{rbf}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \exp(-\gamma \left\|\boldsymbol{x^i} - \boldsymbol{x^j}\right\|^2) \tag{3.4}$$

In addition to the aforementioned kernels, we can also find other alternative kernels such as the sigmoid kernel. This function, as the name implies, takes the form of a sigmoidal function, similar to the logistic regression function. As we can see in equation 3.5, this function has two hyperparameters. Firstly, $\gamma \in \mathbb{R}^+$ regulates, similarly to how it does in, for example, the RBF kernel, the amount of influence each data point has in the similarity calculation. On the other hand, the coefficient $c_0 \in \mathbb{R}$ is a hyperparameter that allows us to control the level of shifting of the sigmoidal function.

$$k_{sigmoid}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \tanh(\gamma(\boldsymbol{x^i})^\intercal \boldsymbol{x^j} + c_0) \tag{3.5}$$

Binary variables

Binary variables are widely known in the field of Machine Learning. From a purely mathematical perspective, a binary variable b can take values $b \in 0, 1$. However, in the literature, we can find binary variables that express their values in the form of two categories, where one category is the negation of the other. Examples of such variables include `blue_eyes` $\in$ {`yes`, `no`}, `state` $\in$ {`open`, `close`} or `exam` $\in$ {`pass`, `fail`}.

In the domain of kernel functions, one of the most well-known for dealing with binary variables is the *Jaccard kernel*. This function is based on the Jaccard index, introduced by the ecologist Paul Jaccard in his article "*The Distribution of Flora in the Alpine Zone*" [19]. Originally, this index determined the similarity between two samples using the following equation:

$$Jac(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.6}$$

Where A and B are the samples. We can observe that, ultimately, the similarity is measured by dividing the size of the intersection of the sets by the size of their union.

From the above expression, we can derive measures of similarity between two vectors of binary variables $x^i, x^j \in \{0,1\}^n$ using the following metrics (with $l = 1 \dots n$):

N_{00} = Number of attributes such that $x^i_l = x^j_l = 0$.
N_{01} = Number of attributes such that $x^i_l = 0$ and $x^j_l = 1$.
N_{10} = Number of attributes such that $x^i_l = 1$ and $x^j_l = 0$.
N_{11} = Number of attributes such that $x^i_l = x^j_l = 1$.

From these metrics, we can obtain the expression of the kernel function:

$$k_{jaccard}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \frac{N_{11}}{N_{01} + N_{10} + N_{11}} \tag{3.7}$$

Note that for those samples in which all matches are of type N_{11}, there will be an indeterminacy of the form 0/0 when applying the kernel function. Therefore, if implemented, the procedure to follow when both samples have all their attributes set to 0 will need to be determined.

Another kernel function based on the binary similarity metrics we just discussed is the *Simple Matching Coefficient kernel* (SMC) [20]. The SMC measures the ratio between the number of matching attributes and the total number of attributes in the samples. We can see the kernel function in question in equation 3.8.

$$k_{smc}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \frac{N_{11} + N_{11}}{N_{00} + N_{01} + N_{10} + N_{11}} \tag{3.8}$$

Unlike the Jaccard kernel (equation 3.7), the similarity measure of the SMC cannot lead to any indeterminacy.

Categorical variables

Another widely used type of variable in the literature is categorical variables. This type of data is primarily characterized by the composition of its domain, which consists of a finite set of values with or without order, known as categories [21]. We can distinguish two types of categorical variables: ordinal and nominal variables [22].

On the one hand, ordinal categorical variables are those that take categorical values in a differentiated order. An example of an ordinal categorical variable would be `coffeeSize` $\in$ {`small`, `medium`, `large`}. On the other hand, nominal categorical variables are those whose domain does not have a pre-established order. An example of a nominal categorical variable would be `city` $\in$ {`Barcelona`, `Girona`, `Lleida`, `Tarragona`}.

During the research process regarding kernel functions associated with this type of variable, we have found that there is not a wide variety available. In the SVM domain, we do not find any available software that allows us to handle problems with mixed data. In fact, usually only kernel functions for continuous numerical variables are implemented. Therefore, in most cases, users tend to convert the values of categorical variables to numerical values using techniques such as OneHotEncoding [23].

One of the most basic functions for measuring the similarity of this type of variable is the *equality kernel*. However, please note that this function only accepts individual categorical values and not vectors of categorical variables. Its operation is simple: if the categories match, the similarity measure becomes 1. Otherwise, the similarity is 0. We can see its definition in equation 3.9.

$$k_{eq}(x_\ell^i, x_\ell^j) = \begin{cases} 1 & \text{if } x_\ell^i = x_\ell^j \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

Nevertheless, Belanche *et al.* propose an interesting family of positive definite kernels for measuring similarity between categorical variables. The kernel is called the *univariate kernel* and is based on determining the similarity between two categories using the proportion with which they appear in the dataset [24]. We can see the definition of the function in equation 3.10.

$$k_{univariate}(x_\ell^i, x_\ell^j) = \begin{cases} h_\alpha(P_Z(x_\ell^i)) & \text{if } x_\ell^i = x_\ell^j \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

Where x_ℓ^i, x_ℓ^j represent the categories associated to the ℓ feature of each sample $\boldsymbol{x^i}, \boldsymbol{x^j}$ and Z is the categorical variable that follows a distribution P_Z (note that $P_Z(x_\ell^i)$ indicates the probability that the variable Z takes the value x_ℓ^i). On the other hand, α is a hyperparameter that parametrizes the inversion function h_α defined in equation 3.11. This function introduces a nonlinear behavior in the calculation of the similarity index between the data.

$$h_\alpha(a) = (1 - (a)^\alpha)^{1/\alpha} \tag{3.11}$$

The univariate kernel provides a more accurate reasoning of the concept of similarity and establishes that not all matches have the same importance compared to the behavior of the equality kernel. Moreover, one of the strengths of the methodology followed by the univariate kernel is that if the probability $P_Z(x^i)$ is very low, the fact that there is a match between two individuals with value x_ℓ^i becomes highly significant.

On the other hand, if the probability of occurrence of a category is very high, it means that the categorical value is very common. Consequently, the fact that two individuals have the same value is not as relevant. The function h_α is responsible for assessing the degree of importance of a match.

Circular variables

One variable that is often under-referenced in the current literature is circular variables. These variables are characterized by having a domain that exhibits a circular ordered structure [25]. In other words, if we choose a value and move from one value to the next in order, we will eventually return to the beginning of the cycle.

An example of a circular variable could be `windDirection` $\in \{0 \dots 359\}$. Another possibility could be `month` $\in \{\texttt{Jan}, \texttt{Feb}, \texttt{Mar}, \dots, \texttt{Nov}, \texttt{Dec}\}$. Note that the domain of a circular variable can be either numerical or categorical, as the only requirement for a variable to be considered of this type is that its structure is cyclical.

We have conducted an extensive research on kernel functions designed to handle this type of variable and unfortunately, we have not found any existing ones. Therefore, we have proposed defining a new one. The formula for the function in question can be seen in Equation 3.12.

$$k_{circular}(x_\ell^i, x_\ell^j) = 1 - \min\{z, 1 - z\} \tag{3.12}$$

Where $z = |x_\ell^i x_\ell^j| \in [0, 1)$ is the absolute difference between the circular values x_ℓ^i and x_ℓ^j. Note that the circular kernel is intended to handle numerical circular variables, specifically values within the range $(0, 1]$ (see Figure 3.1). In other words, it is expected that $x_\ell^i, x_\ell^j \in (0, 1]$. Therefore, before using the kernel, it will be necessary to associate each element of the variable's domain with a value within the established range.

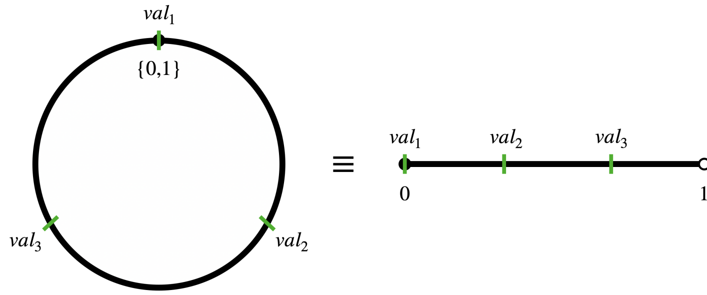


Figure 3.1: Segmentation of circular values in the $(0, 1]$ range.

Source: Own elaboration.

String variables

The last type of variable we will study is string variables. Nowadays, texts have gained significant importance in the field of Machine Learning and Data Analysis. We can understand a *string* as a structured sequence of zero or more characters.

Some examples of string variables include a tweet, a paragraph from a book, the description of symptoms presented by an examined individual in a hospital or a character sequence representing an individual's DNA sequence. In fact, textual variables are widely used in the field of bioinformatics [17].

It is not difficult to realize that being able to analyze the texts that surround us and include them in data analysis problems can improve the problem-solving capabilities of our algorithms.

In the current literature, we find different kernels that deal with textual variables. The simplest one is called the *p-spectrum kernel*. It evaluates the similarity between two textual samples by computing the matches of sub-strings of size p [26]. Its formulation can be seen in Equation 3.13.

$$k_{spectrum}(s,t) = \sum_{i=1}^{|s|-p+1} \sum_{j=1}^{|t|-p+1} k_p(s_{i:i+p}, t_{j:j+p}) \tag{3.13}$$

Where s and t are textual variables. The notation $s_{i:j}$ indicates the substring in s that contains the characters from position i to position j. $|s|$ represents the size (number of characters) of string s. k_p is another kernel function (see equation 3.14) that returns 1 if the substrings $s_{i:i+p}$ and $t_{j:j+p}$ are equal and 0 otherwise.

$$k_p(s,t) = \begin{cases} 1 & \text{if } s = t \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

We can find more efficient formulations of the p-spectrum kernel based on dynamic programming and the use of data structures called *tries* [27]. However, the details of the formulation and implementation are beyond the scope of this project, so we will only mention their existence.

3.2 The Aggregation Kernel

At this point, we have defined the formulation of a Support Vector Machine (SVM) and the role played by Kernel Functions (KF) in these algorithms. Furthermore, we have studied the variables that appear in the main problems of the current literature and the kernels defined for each of them.

We have also observed that while using SVMs, there is nothing preventing us from working with different types of data vectors (real data vectors or binary data vectors, among others) as long as we have a valid kernel capable of capturing the similarity between these vectors.

Despite the existence of kernels designed to handle different types of data (such as RBF for continuous data, SMC for binary data or Univariate for categorical data, among others), we have not seen any cases addressing what happens when the samples to be analyzed consist of mixed data. In fact, this is not a matter that can be overlooked, as the majority of contemporary problems are indeed composed of data of different types. So, can we analyze datasets with mixed data using Support Vector Machines?

As we mentioned, it is indeed possible to solve problems composed of mixed data vectors as long as we find a valid kernel that captures the similarity between them. In the current literature, we have found only one reference to kernels for mixed data in the article "Improved modeling of clinical data with kernel methods". In this article, Daemen *et al.* define an additive kernel (which we will rename as the average kernel) designed to handle medical data consisting of ordinal and binary variables.

The authors define the new kernel based on closure properties 3a (which ensures that the sum of kernels is also a kernel) and 3b (which ensures that a kernel multiplied by a value $a \in \mathbb{R}^+$ is also a kernel). You can find the formulation of the kernel function in equation 3.15.

$$k_{avg}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \frac{1}{L} \sum_{\ell=1}^{L} k_\ell(\boldsymbol{x^i}, \boldsymbol{x^j}) \tag{3.15}$$

In the mentioned function, L is the number of vectors to be processed with different kernels and k_ℓ with $\ell \in [1, L]$ is a valid kernel that captures the similarity between each of these vectors. For example, let's assume our samples consist of a data vector of size d. Features $[1, d/2-1]$ are continuous, while features $[d/2, d]$ are binary. We can evaluate the similarity of the first subset of features using, for instance, $k_1 = k_{rbf}(\boldsymbol{x^i}, \boldsymbol{x^j})$ and the similarity of the second subset using, for instance, $k_2 = k_{smc}(\boldsymbol{x^i}, \boldsymbol{x^j})$. Once we have obtained these similarity scores, we calculate the overall similarity by averaging the scores: $1/2(k_1 + k_2)$. However, besides the definition of the kernel function, the authors do not propose any operational and open implementation to test the results they present.

While it is true that k_{avg} is sufficient for dealing with mixed data vectors, we believe that we can improve the similarity measure by defining a new kernel function based on the aggregation kernel proposed in [6]. We will refer to this new function as the *Aggregation Kernel*. You can find its formulation in equation 3.16.

$$k_{aggr}(\boldsymbol{x^i}, \boldsymbol{x^j}) = \frac{\exp(\gamma \; k_{avg}(\boldsymbol{x^i}, \boldsymbol{x^j})) - 1}{\exp(\gamma) - 1} \tag{3.16}$$

On one hand, the new function introduces a hyperparameter $\gamma \in \mathbb{R}^+$, which will help define the extent to which a sample influences the generation of the separating hyperplane. If the value of γ is high, we will obtain a hyperplane that separates the data by giving significant consideration

to samples near the decision boundary, practically ignoring the more distant samples. On the other hand, if γ takes a very low value, distant data will have a considerably higher weight in the definition of the hyperplane, while nearby data will have little influence. To conclude, notice that $k_{aggr}: \mathcal{X} \times \mathcal{X} \longrightarrow (0, 1]$, where $\mathcal{X}$ is the input space of the data.

Additionally, the application of the exponential function to the resulting value of the average kernel is also incorporated to introduce non-linear behavior in the similarity measure.

The proof that the aggregation kernel represents a valid kernel arises from the closure properties of kernels stated in Section 2.3. Let's see it:

$$
\begin{aligned}
k_{avg} \text{ is kernel} &\Longleftrightarrow [\text{by closure } 3b] \\
\gamma\ k_{avg} \text{ is kernel} &\Longleftrightarrow [\text{by closures } 3b, 3a] \\
\exp(\gamma\ k_{avg}) \text{ is kernel} &\Longleftrightarrow [k \text{ kernel} - 1 \text{ is kernel}] \\
\exp(\gamma\ k_{avg}) - 1 \text{ is kernel} &\Longleftrightarrow [\text{by closure } 3b] \\
\frac{\exp(\gamma\ k_{avg}) - 1}{\exp(\gamma) - 1} \text{ is kernel} &\quad \blacksquare
\end{aligned}
\tag{3.17}
$$

Having defined the aggregation kernel, we have now developed a new kernel function that can serve various purposes. This kernel opens up opportunities to tackle more diverse problem domains, capture the semantics of data more effectively and find applications in fields such as education and research.

The improved kernel has the potential to enhance the performance and flexibility of Support Vector Machines, enabling them to handle mixed data types and provide richer insights. Its implementation can enable the analysis of complex datasets and facilitate advancements in data-driven research and educational applications.

4 Support Vector Machines on large problems

At this stage, we have examined the formulation of Support Vector Machines (SVMs) and Kernel Functions. Additionally, we have studied and analyzed the different types of data encountered in modern data analysis problems, along with the kernel functions designed to handle them. Finally, we have proposed a new kernel function, named the Aggregation Kernel, which is capable of capturing the similarity between vectors composed of mixed data.

One of the main objectives of this project is to propose an alternative to the conventional algorithm of Support Vector Machines (SVMs) in order to improve their efficiency when dealing with large datasets. In this chapter, we will begin by conducting a state-of-the-art review of the various efficiency improvements in SVMs found in the current literature. Subsequently, we will assess an alternative to these existing solutions based on some fundamental principles of Bagging theory.

4.1 Efficiency improvements in Support Vector Machines

While it is true that Support Vector Machines (SVMs) are widely used algorithms in solving modern problems, they also have some drawbacks that make them less attractive when dealing with large amounts of data. In fact, from the algorithm's formulation, it can be inferred that the training complexity of an SVM is at least $\mathrm{O}(N^2)$, where N is the number of training samples.

For this reason, in the current literature, we can find multiple approaches that attempt to mitigate this inefficiency using various techniques. In this section, we will describe and analyze some of them.

One of the first techniques that caught our attention in this field is the one proposed by Awad et al. in 2004. This technique is based on using clustering techniques to reduce the number of candidate samples to become Support Vectors [8]. Specifically, they start by performing clustering analysis to approximate the samples that would become Support Vectors in a standard SVM training. Subsequently, training is carried out using a much smaller training set, resulting in a significant decrease in training time.

Osuna *et al.* propose two methodologies to reduce the runtime complexity of the dual formulation of Support Vector Machines. Firstly, they propose a procedure that employs a regression SVM to approximate the decision boundary. Secondly, they approach the problem from a reformulation of the primal formulation of SVMs that maintains the original primal structure while incorporating the use of kernel functions [9]. This way, the same hyperplane as with the original formulation is obtained, but fewer basis functions are used. By applying both methodologies, an improvement in the training phase execution time is achieved without any noticeable system degradation.

Another approach that also focuses on reducing the training dataset before training the SVM is proposed by Zhan *et al.* The authors define a new methodology that is based on eliminating candidates for Support Vectors (SVs) that may cause the generated hyperplane to have a highly

convoluted shape [10]. Subsequently, the SVM is trained with reduced complexity due to the prior elimination of these problematic SV candidates.

Bottou *et al.* propose an algorithm that is similar to what we have in mind to study and, if applicable, design in subsequent sections. It is called Cross-Training SVM [28]. This reinterpretation of the original algorithm involves partitioning the training data into different subsets. Then, different SVMs are trained using each subset. Subsequently, the decision functions of the trained SVMs are used to selectively discard some of the training examples. This way, the SVM is retrained without including the discarded samples, resulting in a significant reduction in overall training execution time.

Another methodology that also involves partitioning the training dataset is proposed by Graf *et al.* As mentioned earlier, this methodology starts by generating a set of data partitions from the training data. Instead of training the SVM using all the data, an SVM is trained for each partition. Then, the results are successively combined to train new SVMs. The algorithm stops either after a specific number of user-defined iterations or when a global optimum is reached [29]. The authors have named this technique Cascade SVM. One of the strengths of the new algorithm is that it allows for parallelization of the different cascade steps to achieve an even more significant reduction in training time.

Scholkopf *et al.* propose a modification of the original Support Vector Machines algorithm to reduce the long training execution time that can occur when analyzing problems with large amounts of data [11]. The new algorithm starts by introducing an efficient and effective method for selecting a good set of training data. Then, they propose a reduction strategy (referred to as shrinking) for the optimization problem, taking into account that many machine learning problems have few Support Vectors (SVs) compared to the training samples and many SVs with α_i close to the value of the parameter C (*i.e.* there are many samples considered SVs with very little margin). Finally, they propose other computational improvements such as including a cache that stores the most recent results of the sample comparison using the kernel function.

One favorable aspect of the previous article is that it includes the implementation of the techniques described in an open-source library called SVM^{light} [11]. In fact, it is one of the few publications we have found that presents the obtained results along with the complete code of its implementation. It is worth mentioning that these techniques are currently widely used in multiple Support Vector Machine libraries such as LIBSVM [30], among others.

With this last article, we conclude the state of the art regarding efficient implementations of algorithms based on Support Vector Machines. However, other articles addressing the inherent challenges in handling large amounts of data using SVMs can also be found in the current literature. Nevertheless, we believe that we have conducted a comprehensive enough research to consider that the approach we want to study has not yet been proposed by any author. In the next section, we will provide a detailed description of this new methodology.

4.2 The Bagging Support Vector Machine

In this section, we will describe a proposed algorithm based on Support Vector Machines (SVMs) and some principles of Bagging theory. The goal of this new methodology is to attempt to reduce the computational time required to train SVMs when dealing with large volumes of data. Before that, however, we will conduct a brief analysis of Bagging theory and its main principles.

The bagging theory

In 1996, Breiman proposed a technique aimed at improving classification processes through the random generation of multiple subsets of training data, which he named Bootstrap Aggregating or commonly known as *Bagging* [31].

The algorithm in question can be divided into three distinct phases [32]. First, bootstrap sampling is applied to an original dataset D of size $|D| = n$. This technique involves generating $m \in \mathbb{R}^+$ training multisets (*i.e.* they can contain repeated elements) $\{D_1, \ldots, D_m\}$. The original samples are randomly assigned to each multiset with replacement.

Second, a machine learning model M_i is trained for each previously generated multiset D_i, where $i = 1, \ldots, m$. Since each dataset is generated independently, this part of the algorithm can be performed in parallel.

Finally, the aggregation phase is initiated and the procedure depends on the type of task being performed. In classification problems, each testing sample is evaluated using the generated models and each model produces a predicted label. The models act as an expert committee and the accepted class is determined through a majority voting procedure. In regression problems, each model generates a prediction for each testing sample and the accepted value for each individual becomes the average of the predictions.

One of the main advantages of using Bagging in the training of machine learning models is the fact that it reduces the contribution of variance to the prediction error through the bootstrap process. This helps prevent models from being prone to overfitting. However, this methodology may not be as effective for stable algorithms (*i.e.* those algorithms that are not sensitive to data variance).

Bagging-SVM approach

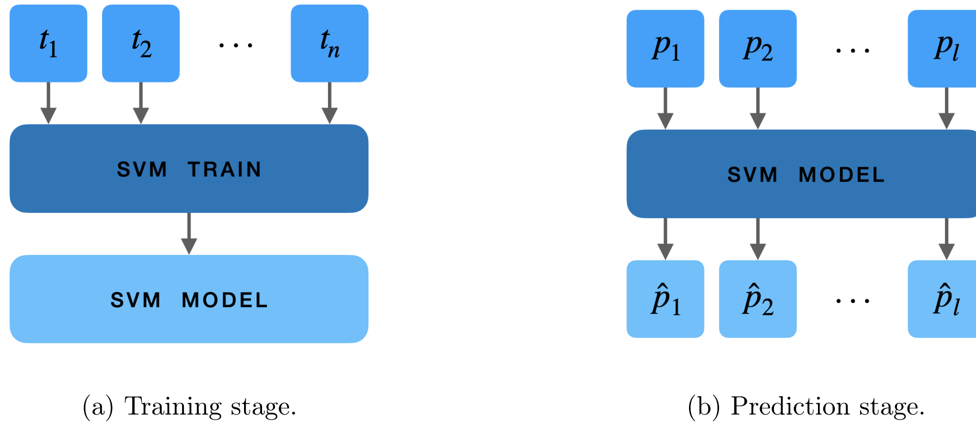
At first glance, Bagging may not seem like a suitable technique to implement in the training process of a Support Vector Machine (SVM). As we mentioned, Bagging helps reduce the contribution of variance to the prediction error, but SVMs are already highly stable algorithms. Furthermore, if we consider the project objectives and, more specifically, this chapter, we intend to study and propose a modification to the SVM algorithm that reduces the training execution time. However, Bagging techniques are primarily focused on improving model accuracy and, in some cases, may even be inefficient.

In this section, we will explore how we can leverage some fundamental principles of Bagging Theory to improve the performance of SVM training on problems with a large amount of data while maintaining the accuracy provided by a conventional model. We have decided to name this algorithm Bagging-SVM.

Let's examine the diagram of the training process of a conventional Support Vector Machine shown in Figure 4.1. It is evident that when dealing with large datasets, the bottleneck in terms of execution time lies in the model generation stage (Figure 4.1a).

Conversely, the prediction phase usually does not represent a significant overhead in execution time (Figure 4.1b). Our algorithm proposes to address the high complexity of the training stage by adopting a procedure similar to Bagging:

1. We generate m subsets of training data randomly.
2. We train m SVM models.
3. We make predictions using majority voting in the case of classification problems or by using an appropriate statistical measure (such as mean or median) in regression problems.



(a) Training stage.

(b) Prediction stage.

Figure 4.1: Diagram of SVM model generation using the conventional algorithm.

Note: $t_1 \ldots t_n$ and $p_1 \ldots p_l$ represent the training and testing samples, respectively. The predicted values by the SVM model for the testing samples are denoted as $\hat{p}_1 \ldots \hat{p}_l$
(Source: Own elaboration).

The main difference lies in how we generate the training subsets (stage 1). Firstly, as mentioned before, SVMs are highly stable algorithms. Therefore, we consider it unnecessary to benefit from the variance reduction provided by conventional bootstrapping techniques.

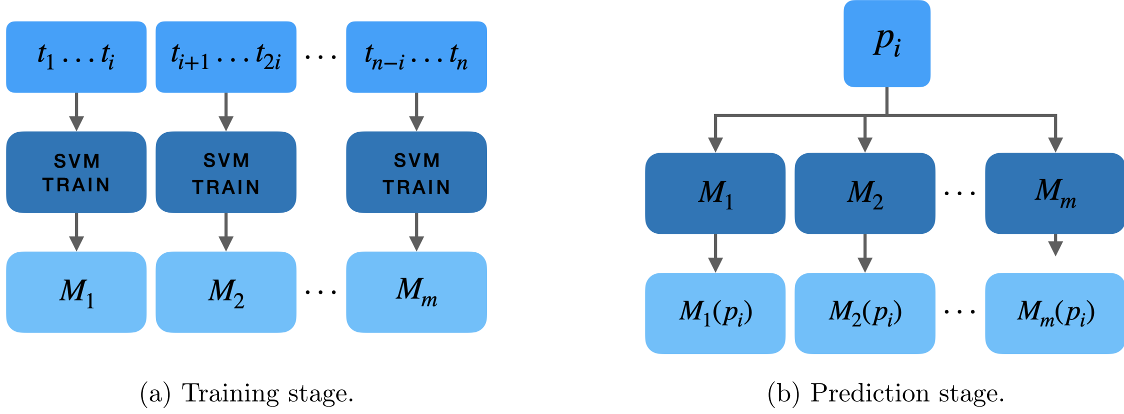
Consequently, the proposed Bagging-SVM will generate m training subsets $\{T_1, \ldots, T_m\}$ randomly and with replacement from the original training sample set $\mathcal{T}$ of size n (*i.e.* $|\mathcal{T}| = n$). Additionally, we will distribute all the training data among the different subsets, ensuring the following conditions hold. You can see a graphical representation in Figure 4.2a and the mathematical representation of the subset constraints in Equations 4.1, 4.2, 4.3. From these equations, it can also be deduced that $\sum_{i=1}^{m} |T_i| = n$.

$$\bigcup_{i=1}^{m} T_i = T \tag{4.1}$$

$$\bigcap_{i=1}^{m} T_i = \emptyset \tag{4.2}$$

$$|T_i| = \begin{cases} \left\lfloor \dfrac{n}{m} \right\rfloor + 1 & \text{if } i \leq \text{mod}(n, m) \\ \left\lfloor \dfrac{n}{m} \right\rfloor & \text{otherwise} \end{cases} \tag{4.3}$$

From this point onwards, stages 2 and 3 are carried out in the same manner as in the original Bagging technique. The different stages can be visually represented in Figure 4.2.



(a) Training stage.

(b) Prediction stage.

Figure 4.2: Diagram of SVM model generation using the Bagging approach.

Note: $t_1 \dots t_n$ and $p_1 \dots p_l$ represent the training and testing samples, respectively. Submodels $M_1 \dots M_m$ are generated for each training subset. Given a testing sample p_i and a model M_j, $M_j(p_i)$ denotes the predicted value of the sample by the corresponding model (Source: Own elaboration).

Complexity of the Bagging-SVM

Once the theoretical foundations of Bagging-SVM have been defined, we will study the complexity of the training stage in this new methodology. The proposed algorithm evenly distributes the n training samples formed by p features into m subsets. Each subset consists of n/m different data points and is used to train a model. Therefore, we obtain a total of m distinct models.

The complexity of training a conventional SVM scales between $\mathrm{O}(pn^2)$ and $\mathrm{O}(pn^3)$ depending on the algorithm implementation and the dataset used [33]. In the Bagging approach each model is trained with n/m input samples. It seems reasonable to assume that the resulting complexity of training a model using Bagging-SVM will be $\mathrm{O}(pn^2/m^2)$. Since we need to train m models, the resulting complexity will be $\mathrm{O}(m(pn^2/m^2)) = \mathrm{O}(pn^2/m)$.

Thus, the training time of Bagging-SVM should be m times faster than the training time of a conventional SVM. If we have m or more execution threads available, we can parallelize the training process, achieving an execution time m^2 times faster than that of training a conventional SVM.

In conclusion, the Bagging-SVM is a novel tool that we have theoretically defined and we are eagerly anticipating its application in subsequent sections. Its potential as a promising approach makes us enthusiastic about its implementation in future work. The Bagging-SVM offers a sophisticated framework that combines the benefits of Bagging with the robustness and stability of Support Vector Machines. We believe that its utilization in real-world applications will provide valuable insights and contribute to advancements in machine learning.

5 Foundations of the Implementation

In previous chapters, we have carried out a brief overview of the history of Support Vector Machines in the field of Data Science. Subsequently, we have studied the different kernels found in the current literature, aimed at handling various types of data and we have stated the formulation of a new kernel function (Aggregation kernel, Section 3.2) that allows evaluating the similarity of mixed data samples.

We have also examined the main efficiency improvements found in the current literature regarding the massive processing of data using Support Vector Machines. Finally, we have proposed a theoretical algorithm for SVM based on some principles of Bagging theory (Bagging-SVM, Section 4.2).

In this section, our goal is to expand an existing library in the literature in order to effectively implement the proposals of the Aggregation kernel and Bagging-SVM. To do so, we will begin by conducting a state-of-the-art review of current implementations of Support Vector Machine algorithms. Subsequently, we will perform an analysis of the selected library. Finally, we will implement both the Aggregation kernel function and the Bagging-SVM algorithm into the library, discussing the main changes and new functionalities introduced.

5.1 Support Vector Machines implementations

In this initial section, we will analyze a set of functional implementations of Support Vector Machine algorithms found in the current literature. To avoid bias in our research, we will examine libraries programmed in different programming languages that offer diverse functionalities.

Kernlab

Kernlab is a library written in the R language, designed to provide basic kernel functionalities to R users [34]. This library allows users to switch between kernels in an existing algorithm and utilize their own kernel functions. Kernlab offers a flexible and comprehensive implementation of SVM, supporting various types of SVM, including binary classification SVM, multiclass SVM and regression SVM.

Additionally, Kernlab implements online kernel algorithms for classification, novelty detection and regression, as well as spectral clustering algorithms, Kernel Principal Component Analysis (PCA) and Kernel Feature Analysis. Despite these functionalities, it is important to note that Kernlab is programmed in R. Despite the critical parts of the library being programmed in C, we believe it would be more suitable to implement the library's functionalities natively in C or C++.

SVMlight

SVM^{light} is a library written in C that was developed in 1998 and has proven to be a valuable tool for solving classification and regression problems using Support Vector Machines (SVM) [35].

One of the key techniques that SVM^{light} implements is the shrinking heuristic, which improves efficiency in handling large volumes of data. Moreover, SVM^{light} employs reasonable descent-based working set selection and kernel evaluation caching, among other strategies, to optimize algorithm performance.

A notable advantage of SVM^{light} is its implementation in C, which enables the generation of efficient and high-performance code. The library provides various predefined kernels such as Linear, Polynomial, RBF and Sigmoid, while also allowing users to define and use custom kernel functions.

However, it is important to highlight that, compared to other libraries implemented in the same language, such as LIBSVM, SVM^{light} may exhibit lower efficiency results [36].

LIBSVM

LIBSVM is a Support Vector Machine (SVM) library written in C/C++ and was initially developed in 2001 [30]. The main objective of this library is to assist users in easily applying SVM to their applications, providing an efficient and robust implementation.

LIBSVM is capable of solving classification and regression problems using various SVM algorithms. In addition to the basic SVM optimization algorithms, it also implements advanced techniques such as multiclass classification and probability estimation.

To improve computational efficiency, LIBSVM employs techniques such as shrinking and caching, which reduce the runtime during training and prediction. These strategies optimize the utilization of computational resources and enhance the overall performance of the algorithm.

A notable feature of LIBSVM is its precise and comprehensive documentation, along with the availability of well-documented source code. Due to its demonstrated efficiency and effectiveness in various problems, this library has received several awards recognizing its excellence [37]. Moreover, LIBSVM provides a wide range of library interfaces implemented in multiple programming languages.

LIBSVM also offers the capability to perform parallel training, leveraging the advantages of multi-core and distributed architectures to accelerate the training process on large datasets.

Shark

Shark is a fast, modular, open-source Machine Learning library written in C++, created in 2008 [38]. This library offers an excellent balance between flexibility and ease of use on one hand and computational efficiency on the other.

Shark includes numerous algorithms from various areas of machine learning and computational intelligence. This encompasses methods for linear and nonlinear optimization, kernel-based learning algorithms, neural networks and various other machine learning techniques.

One of the advantages of Shark is that it provides an effective solution for real-world machine learning problems. Additionally, this library offers a plethora of powerful algorithms that, according to its authors, are not implemented in any other library.

A notable aspect of Shark is its detailed and well-commented documentation through Doxygen. This feature facilitates its usage and understanding, as the code comments provide clear explanations of the functionality and usage of different components within the library.

Final choice and justification

In the literature, besides the previously discussed libraries, we can find other SVM libraries such as mySVM [39] or SVMTorch [40]. However, in our study, we have focused on analyzing the most successful and cited libraries in the current literature.

After a comprehensive analysis, we have decided to implement the techniques proposed in previous sections using the LIBSVM library. This choice is based on several reasons. Firstly, we have prior experience in manipulating LIBSVM thanks to our undergraduate final project [41]. This familiarity and knowledge of the library will facilitate the development of new functionalities.

Secondly, we have exclusively opted for libraries written in C/C++, as we consider it to be the language that allows us to achieve maximum efficiency, which is one of the most important requirements for our project. This choice is based on the ability of C/C++ to generate efficient and fast code, which will be crucial for tackling real-world problems with large volumes of data.

Lastly, another important reason is that LIBSVM offers clean, well-organized and easily manipulable code. This will allow us to implement the new functionalities in a clear and efficient manner.

5.2 Analysis of LIBSVM

Once the choice of the Support Vector Machine library we want to extend has been made and the reasons for it have been justified, we will briefly analyze some of its main features. In this section, however, we will specifically focus on the implementation details, which are ultimately the ones that interest us the most in order to be able to extend it.

LIBSVM is a library that implements various Support Vector Machine algorithms. Its authors, Chang *et al.*, have been awarded different prizes for the efficiency of the software they created in 1998. Over the years, the library has been updated multiple times, adding new features and functionalities. On their website, we can see that there are versions of the library available for a wide range of programming languages [37]. However, as mentioned before, in this project, we will be working exclusively with the C++ version. This version is structured with the following files:

- `svm-train.c` handles the SVM training process.
- `svm-predict.c` implements the SVM prediction procedure.
- `svm-scale.c` is a program focused on data scaling, prior to the SVM training or prediction phase.
- `svm.h` and `svm.cpp` contain the declaration and implementation, respectively, of the data structures and main methods that underlie the training and predictions performed by the SVM.

The library allows for handling continuous data in sparse format. The main data storage structure is the `svm_node`, which contains the value of a feature characteristic of the problem. It consists of the index (int) of the respective feature and its value (double).

To store the different samples, LIBSVM implements the `svm_problem` structure. It is composed of the number of samples comprising the problem, the samples represented by a pointer to an array of `svm_node` pointers and the labels associated with each sample.

To store the SVM parameters, the `svm_parameter` data structure is used, which contains, among others, attributes representing the type of algorithm to be used and its parameters or the type of kernel to employ and its parameters. On the one hand, LIBSVM supports C-SVC, ν-SVC, One-Class, ϵ-SVR and ν-SVR SVM types. On the other hand, the library also implements the Linear, Polynomial, Radial Basis Function (RBF), Sigmoid and Precomputed kernel functions. The Precomputed kernel allows for directly inputting the similarity measures between samples (*i.e.* the Kernel Matrix) instead of providing the different training samples.

During the training process, a model is generated and its information is stored in the `svm_model` structure. This structure contains, among other things, a copy of the SVM parameters (`svm_parameter`), the number of classes present in the data, the number of Support Vectors (SV) generated and the coefficients of the SVs.

In addition to the data structures, the library implements multiple methodologies that enable the training phase and subsequent prediction phase of the algorithm. For the sake of simplicity and due to space limitations in the project, we will not discuss all of them but rather focus on the parts responsible for executing the kernel functions.

The `Kernel` class is primarily responsible for executing the different kernel functions. It contains the C++ code for the previously mentioned functions. LIBSVM utilizes a caching system to store recently computed similarity measures. Hence, it also requires `swap` operations between the different elements of the kernel matrix. Consequently, each kernel function has two associated implementations within the same class.

On the one hand, the library uses a class method for each kernel function. These methods analyze the samples during the training stage and, therefore, allow for the `swap` operations if necessary. These functions also feature procedures that significantly improve efficiency during this stage of the algorithm.

On the other hand, each kernel function also has an associated static method. These functions are designed to be executed during the prediction stage. The main reason is that during this second stage of the algorithm, a high level of optimization is not required. For example, as mentioned earlier, the cache is used during the training stage to avoid a high number of comparisons between individuals in the dataset. In the prediction stage, however, there is no need to keep the kernel matrix in memory and consequently, the caching system becomes unnecessary.

Of course, LIBSVM includes other very important and fundamental methodologies related to SVM algorithms. If you want to explore what they are and the functions they perform, you can find more information in the associated paper for the library.

In summary, LIBSVM is a well-structured, efficient and open-source library that perfectly implements various classification and regression algorithms based on SVMs. Nonetheless, we believe it also has some limitations.

To begin with, as mentioned earlier, it only allows working with real-valued variables. Both its data structures and the main algorithms and methods it implements are limited by this restriction. Consequently, any attempt to implement new kernel functions is also limited to working with continuous data.

In the following sections, we will provide an alternative implementation of LIBSVM that we will call *AggSVM*. This new version will allow for analyzing problems consisting of mixed data.

In order to measure the similarity between such data, we will also implement multiple compatible kernel functions. Primarily, a functional version of the Aggregation Kernel described in Section 3.2 will be included. Additionally, we will incorporate the ability to train and make predictions using the Bagging-SVM algorithm formulated in Section 4.2, which will enable us to tackle problems with large volumes of data efficiently without sacrificing prediction accuracy.

5.3 Implementation of the Aggregation Kernel

At this point, we have been able to analyze and evaluate the main characteristics of LIBSVM. Furthermore, we have identified the critical code points that will be crucial when implementing new features. In this section, we aim to implement the Aggregation Kernel defined in Section 3.2. In addition to the main implementation, the code will require several deep modifications that we will analyze and specify below.

The first modification we have made is a complete migration of the original library to the C++ language. This way, we can take advantage of the latest language features such as regular expressions or lambda functions. The language version we will adopt from now on is C++17.

General structure

In the new version of the library, we have kept the general file structure of the original version, avoiding the need to create new files or remove any existing ones. However, we have restructured the content of the `svm.cc` and `svm.hh` files to improve their organization. We have chosen to move the method and class headers to the `svm.hh` file, while the implementations have been placed in the `svm.cc` file.

Regarding the `svm-scale.cc` file, we have decided not to update the data scaling procedure at this stage of the project. This task will be left for future projects, although it will not be complicated. It will only be necessary to adapt the scaling methodologies to take into account that the data can now have different types.

On the one hand, the `svm-train.cc` file will be responsible for parsing the SVM parameterization, reading the data in the appropriate format and calling the training and model storage methodologies.

On the other hand, the `svm-predict.cc` file will be responsible for reading the models generated during the training phase, parsing the parameters associated with the prediction phase and calling the prediction methodologies.

With this file structure, we have organized the functionalities related to SVM model training and prediction in a clear and separate manner. This will facilitate code maintenance and understanding, as well as provide an organized structure for implementing future improvements and extensions.

Data Structure updates

One of the most important points to address is how we have modified the basic data structures of LIBSVM to allow for handling multiple kernel functions and, consequently, different types of data simultaneously. In this section, we will discuss the modifications made to the existing structures and the new structures required for the proper functioning of the new version of the library.

```
1  struct svm_node {
2      int index;
3      // Converts and sets index and value attributes from string
4      // In case of error, returns an error message reporting it
5      virtual const std::string set_index_value(std::string &index, std::string &
       value) = 0;
6      // Returns the node_type associated with the derived struct
7      virtual node_type get_type() const = 0;
8      // Returns the string representation of the node_type of the derived struct
9      const std::string get_stype() const;
10     // Returns the string representation of the node_type passed as a parameter
11     static const std::string get_stype(const node_type t);
12     // Returns the address (this + i)
13     virtual svm_node* get_ith_node(int i) const = 0;
14     // Prints the content of the node in INDEX:VALUE format
15     virtual void print(FILE *fp) const = 0;
16     // Allocates/deallocates an array for each of the derived data types
17     static void allocate_space(std::vector<svm_node*> &space, std::vector<int>
       &num_nodes);
18     static void free_space(std::vector<svm_node*> &space);
19 };
```

Listing 5.1: New `svm_node` structure of LIBSVM^{aggr} library.

Basic node

After a thorough analysis of the original LIBSVM library, we consider the `svm_node` structure to be the most fundamental structure. A node stores the index of a feature and its value. Recall that LIBSVM allows for inputting data in sparse format, which we want to maintain in our version.

To enable a `svm_node` to store different types of data, we have determined that the best approach is to convert the structure into a pure virtual one. A pure virtual structure in C++ is one that can only be instantiated by its derived structures. In a way, we can understand it as a Parent class that stores common information of the Derived classes.

The new code for the `svm_node` can be seen in Listing 5.1. As we can observe, the node still maintains the integer attribute `index`. This is because regardless of the data type to be stored, it will be necessary to have the feature index information.

In addition to this main attribute, we can see that different pure abstract functions are defined. These functions are the ones that will facilitate creating, storing and displaying the content of the new data types that users want to implement. Note that all the derived classes (representing new data types) will be required to implement these methods.

In this new version of the LIBSVM^{aggr} library, we have decided to implement support for continuous, binary, categorical and string data. Therefore, we have created three derived structures from `svm_node`, namely `real_node` (for continuous data), `int_node` (for binary and categorical data) and `string_node` (for textual data).

Note that allowing the use of a new data type in the library does not necessarily require the implementation of a new derived class from `svm_node`. For example, binary and categorical variables can both be stored using an `int_node` interchangeably (see listing 5.2). However, when introducing a new variable type, it will be necessary to define its kernel and, as we will see later, its associated structure will be responsible for correctly reading, encoding and storing the feature in an `svm_node`.

```
1  struct int_node: svm_node {
2      int value;
3      const std::string set_index_value(std::string &index, std::string &value)
       override;
4      node_type get_type() const override;
5      svm_node* get_ith_node(int i) const override;
6      void print(FILE *fp) const override;
7  };
```

Listing 5.2: Derived class for data stored as an integer in LIBSVM^{aggr} library.

Kernel parameters

Let's briefly interrupt the inspection of the changes implemented in the existing data structures to advance the definition of a new data structure essential for implementing the Aggregation Kernel. So far, choosing which kernel function to use for each problem was a task for the user through the command line using the `-t` flag. However, we will now have a global kernel (Aggregation) and multiple sub-kernels for different features or feature vectors.

In the original library, the `svm_parameter` structure was used to store the parameters of the kernel to be used. However, now the user will be able to use multiple kernel functions on the same dataset. In subsequent sections, we will specify how we will facilitate the establishment of parameters for each kernel. In this section, we will outline the new structure that will be responsible for storing them during runtime.

Similarly to the data types, the user should be able to define multiple kernel functions. That is why we have defined a new data structure called `kernel_params`. It is self-evident that all kernel functions will share certain characteristics. Therefore, this new structure will also be a pure virtual one. In listing 5.3, you can see the simplified header of the new structure. For space limitations, we have only included the most important virtual functions.

As we have seen, every instance of `kernel_params` will store the number of features associated with the kernel, as well as the indices of these features in the input dataset.
If the users whish to include a new kernel function, they will need to implement its corresponding substructure of `kernel_params`. In this initial version of LIBSVM^{aggr}, we have implemented the Linear, Polynomial and RBF kernels for continuous variable vectors. For binary variable vectors, we have implemented the SMC and Jaccard kernels. The Univariate kernel will handle categorical variables, while the Circular kernel will estimate the similarity between circular variables. Lastly, we will use the p-Spectrum kernel to handle string variables. Consequently, all `kernel_params` substructures associated with these kernel functions are already included in LIBSVM^{aggr}.

We can see an example of the parameter storage structure for the Polynomial kernel in Listing 5.4. This kernel consists of three hyperparameters: the degree of the polynomial (`degree`), the scalar product multiplier (`gamma`) and the polynomial coefficient (`coef0`).

We briefly pause to discuss the importance of the virtual structures `svm_node` and `kernel_params` in the new library. With their presence, we greatly facilitate the implementation of new functions and data types. For instance, implementing the `fill_node` method as a pure virtual function in `kernel_params` allows us to achieve significant milestones in terms of code simplicity and execution efficiency.

```
1  struct kernel_params {
2      int *indexes = nullptr; // Feature indexes that the kernel will use
3      int num_features;       // Number of features that the kernel uses
4
5      // Sets all the kernel parameters from a ConfigStorage instance
6      virtual const std::string set_params(ConfigStorage &cfg) = 0;
7      // Returns the kernel_type
8      virtual kernel_type get_type() const = 0;
9      // Returns the kernel's node_type
10     virtual node_type get_dtype() const = 0;
11     // Checks the correctness of the kernel parameters
12     virtual const std::string check_params() const = 0;
13     // Clone (deep copy) the kernel parameters
14     virtual kernel_params* clone() = 0;
15     // Sets the index and value of the svm_node received as a parameter,
16     // taking into account the format of the 'value' supported by
17     // the kernel in question
18     virtual const std::string fill_node(svm_node *space, int i_space, int &
       index, std::string &value) const = 0;
19     // ... Some methods are omitted for simplicity ...
20 };
```

Listing 5.3: `kernel_params` pure virtual structure of LIBSVM^{aggr}.

```
1  struct poly_params: kernel_params {
2      double gamma = -1;
3      double coef0 = -1;
4      int degree = -1;
5      // Constructor
6      poly_params() {}
7      // Setters
8      const std::string set_params(ConfigStorage &cfg) override;
9      const std::string fill_node(svm_node *space, int i_space, int &index, std::
       string &value) const override;
10     // ... Some methods are omitted for simplicity ...
11 };
```

Listing 5.4: Data structure to store the parameters of the Polynomial kernel.

On one hand, we ensure that the instantiation of `svm_node` objects is only required in this function. Since providing the code for the `fill_node` method is mandatory when creating a new `kernel_params` (due to it being a pure virtual function), we ensure that the correct instantiation of the data type is implemented at compile-time. Overall, we consider this feature to be very useful, straightforward, clean and conducive to code expansion.

On the other hand, we are decoupling the data type associated with the kernel from the `svm_node` substructure that will contain the feature value. A clear example of this situation arises with categorical variables: the user can input categories as strings. Nevertheless, the corresponding instance of `univariate_params` will implement the `fill_node` procedure, converting the string into an integer value and storing it in an `int_node`. This "translation" enables more efficient comparisons of categorical values in subsequent computations.

Overall parameters and model

The `svm_parameter` structure also undergoes changes in the new library. As we have mentioned on multiple occasions, unlike LIBSVM, LIBSVM^{aggr} requires the use of multiple kernel functions. At this point, we know that we will store their parameters in the structures derived from

```
1   struct svm_parameter {
2       svm_type svmtype = C_SVC;
3       int bagging = 0;                    /* indicates if we are training a
        bagging-SVM */
4       double gamma = -1.0;                /* for aggregation kernel */
5       int num_kernels = 0;                /* number of kernels that will be used
        */
6       int num_total_features = 0;         /* overall number of features of the
        problem */
7       kernel_params **kparams = nullptr;  /* kernel parameters */
8       // ... Some attributes and methods are omitted for simplicity ...
9   };
```

Listing 5.5: Summary of the new header of the `svm_parameter` structure.

`kernel_params`. A direct consequence of this modification is that it is no longer necessary to store the kernel parameters directly in `svm_parameter`. Instead, it will be necessary to store the set of `kernel_params` defined by the user.

We can observe the most important features of the new header code for `svm_parameter` in listing 5.5. In this code, we can see a new array that will contain the `kernel_params` for all the kernels used in problem resolution. We can also observe that we have retained the attribute `gamma`. In the previous version, this attribute was used as the γ hyperparameter for the Polynomial and RBF kernels. In LIBSVM^{aggr}, the attribute represents the γ hyperparameter for the Aggregation Kernel.

A delicate part of all the implementations we have discussed so far was to ensure that no part of the base Support Vector Machines algorithm was modified. While it is true that, on one hand, the original library does not allow for mixed data handling, on the other hand, its authors have done an exceptional job in terms of algorithm programming.

One of the reasons why the structures have been implemented in this way is precisely what we mentioned in the previous paragraph. Among other benefits, the modification made so far has allowed us to avoid practically modifying the `svm_model` structure, which is responsible for storing the model generated by SVM and is a fundamental part of its training.

Main problem

In the original version of LIBSVM, only the `svm_node` structure (non-virtual) is required to store the data. As we have mentioned in the analysis of the library, a node consists of the index of the feature and its value, which is always of real type. Therefore, a problem only requires an array of nodes (all the data stored continuously) and the total number of data (to determine the size of the previous array).

In the new version of the library, we need to consider that we will be working with multiple kernel functions. Thus, we need to ensure that each kernel function has the least costly access to the data it needs to process. To achieve this goal, we have had to redefine the structure of the `svm_problem`.

Firstly, we have defined the concept of a *feature set* and its structure, `feature_set`. A feature set will contain a set of attributes from an input sample. These features will be associated with a single kernel. Therefore, an individual will consist of different feature sets, as many as the kernels we need to use in the given problem.

```
1  struct feature_set {
2      // Attributes
3      int n = 0;
4      svm_node *data = nullptr;
5      // ... Some methods are omitted for simplicity ...
6  };
```

Listing 5.6: Header of the new `feature_set` data structure.

```
1  struct svm_problem {
2      int l = 0;
3      int num_kernels = 0;
4      double *y = nullptr;
5      feature_set **x = nullptr;
6      // ... Some methods are omitted for simplicity ...
7  };
```

Listing 5.7: Header of the modified `svm_problem` data structure.

Programmatically, a `feature_set` will be composed of an array of `svm_node` and an integer n indicating the size of the array (*i.e.* the number of features contained in the `feature_set`). You can find the code for the `feature_set` header in Listing 5.6.

While in LIBSVM an `svm_problem` contained a pointer to pointers to `svm_node`, now LIBSVM^{aggr} will contain a pointer to pointers to `feature_set`. You can find a graphical representation of the new problem storage methodology in Figure 5.1. We have also included the new code for the `svm_problem` header in listing 5.7.

Kernel Functions

Regarding the `Kernel` class, we have decided to keep the same structure as in the original library. This implies maintaining two functions for each kernel: a class function for the training stage and a static function for the prediction stage. The class function is used during training, while the static function is applied for prediction, as it is not necessary to instantiate the Kernel class to make predictions.

To maximize efficiency in executing the aggregation kernel during the training stage, we have modified the structure of the Kernel class. With the presence of multiple kernel functions, we have indexed pointers to each of these functions in an array. This allows us to avoid repetitive `if-then-else` comparisons and directly access the appropriate kernel function during training, as the `svm_parameter` stores which kernel to use at each moment.

Regarding prediction, we have implemented a mechanism that dynamically determines which kernel function to call in each case. Since the kernel functions are now static, we no longer have the array of pointers to the kernel functions. This may imply additional time in the prediction stage, but it will be thoroughly studied during the experimentation phase to assess its impact.

In this initial version of the LIBSVM^{aggr} library, we have decided to implement the Linear kernel (equation 3.1), Polynomial kernel (equation 3.2), RBF kernel (equation 3.4) for continuous variables. For binary variables, we have implemented the Jaccard kernel (equation 3.7) and the Simple Matching Coefficient kernel (equation 3.8). To evaluate the similarity of categorical variables, we have included the Univariate kernel function (equation 3.10). Finally, to analyze string-type variables, we will use the p-Spectrum kernel (equation 3.13).

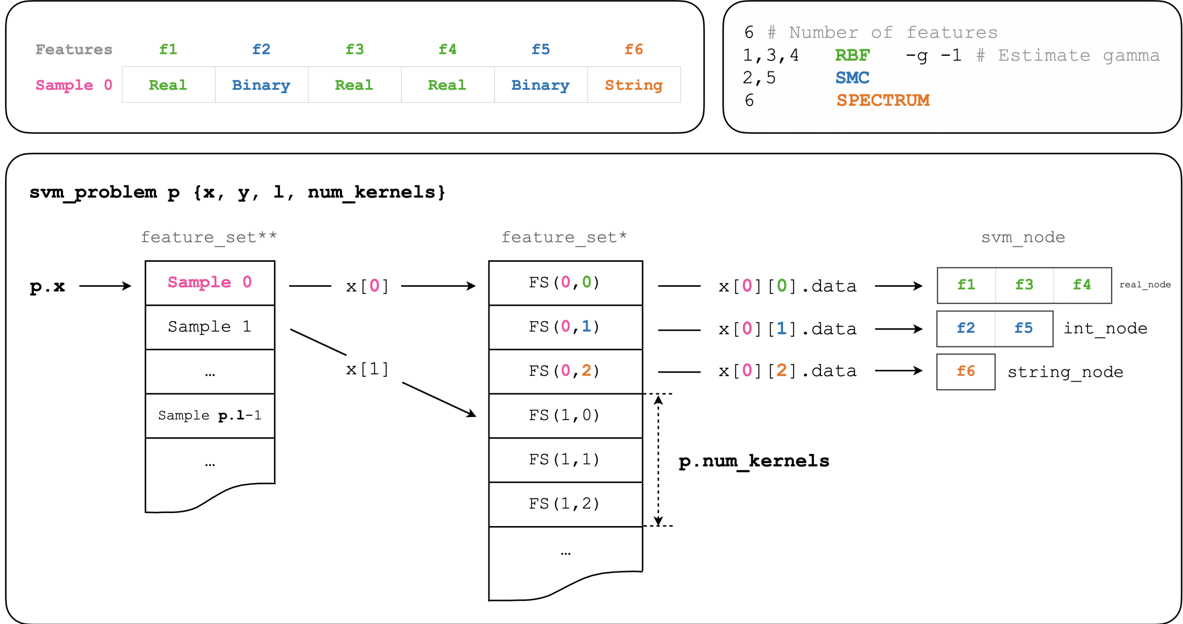


Figure 5.1: Graphical representation of the `svm_problem` of LIBSVM^{aggr}.

In the upper left corner, we can find a sample example. Positioned in the upper right corner is the configuration file. Below, the representation of the storage of the sample in question (Source: Own elaboration).

In addition to implementing the aforementioned features, we have considered a modification of the original kernel function formulation to allow for the execution of problems with a single kernel. In cases where the user wishes to use only one kernel, there is no need to calculate the average similarity value and apply the aggregation kernel since there will be only one similarity. Therefore, in these cases, only the specific kernel function that has been selected will be executed. You can see the alternative formulation in equation 5.2.

$$k_{avg}(\boldsymbol{x^i}, \boldsymbol{x^j}; L) = \frac{1}{L} \sum_{\ell=1}^{L} k_\ell(\boldsymbol{x^i}, \boldsymbol{x^j}) \tag{5.1}$$

$$k_{aggr}(\boldsymbol{x^i}, \boldsymbol{x^j}; L) = \begin{cases} \dfrac{\exp(\gamma \; k_{avg}(\boldsymbol{x^i}, \boldsymbol{x^j}; L)) - 1}{\exp(\gamma) - 1} & \text{if } L > 1 \\ k_1(\boldsymbol{x^i}, \boldsymbol{x^j}) & \text{otherwise} \end{cases} \tag{5.2}$$

Where L is the number of kernels defined for the different data vectors and k_1 represents the single kernel used in the case of a single-kernel problem.

The expansion of the Kernel class has been carried out with a focus on achieving maximum efficiency in both the training and prediction stages. However, we recognize the importance of conducting thorough testing to evaluate and verify the performance of the new code. This testing phase will allow us to ensure the robustness, effectiveness and efficiency of our implementation in different scenarios and using diverse datasets.

The configuration file

In the original version of the library, it was only necessary to parameterize a single kernel, which could be specified through the command line. However, with the introduction of multiple kernels in the new version, this method of parameterization becomes confusing and error-prone.

In order to simplify and improve the kernel parameterization in problem resolution, we have introduced a new element in SVM training stages: the *configuration file*. This file will contain the information associated with all the kernels that are desired to be used.

The configuration file must follow a specific structure. In the first line, a positive integer value indicating the total number of features in the problem is expected. Subsequently, in the following lines, the parameterization of each kernel will need to be specified using the following syntax:

```
<INDICES>   <KTYPE>   <PARAMS>
```

In the above syntax, <INDICES> represents a list of positive integer values separated by commas, indicating the features on which the kernel will be applied. For example, if we want to apply the kernel to attributes 1,3,5,6 and 7, we would specify: 1,3,5,6,7. <KTYPE> indicates the type of kernel we want to use and <PARAMS> represents the parameters associated with that kernel. The file syntax also allows for adding comments using the # character.

This way, through the configuration file, we can specify the parameterization of all the kernels that are part of the problem resolution. This approach avoids common confusion and errors in command line parameterization, improving the clarity and efficiency of the SVM configuration process.

Next, we will proceed to detail the possible parameterizations for each of the kernels implemented in LIBSVM^{aggr}. This information will be crucial for users when configuring the kernels in the configuration file, thus ensuring an optimal and customized utilization of the library.

Linear, SMC, Jaccard and Circular kernels

The Linear, Simple Matching Coefficient (SMC), Jaccard and Circular kernels do not have any specific hyperparameters. Therefore, in the configuration file, we only need to specify the associated indices and the <KPARAM> key as follows:

```
<INDICES>   LINEAR
<INDICES>   SMC
<INDICES>   JACCARD
<INDICES>   CIRCULAR
```

Polynomial kernel

To specify the features associated with a Polynomial kernel, we will use the <KTYPE> POLY. This kernel function has three hyperparameters: degree (positive integer), gamma (positive double) and c0 (double). We can specify these parameters using the flags -d, -c, -c, respectively. Here is an example of parameterization:

```
<INDICES>   POLY   -d 3  -g 1.3  -c 4.7
```

Radial Basis Function kernel

To specify the attributes associated with an RBF kernel, we will use the `<KTYPE> RBF`. This kernel requires a single hyperparameter γ (positive double). To set its value in the configuration file, we can use the flag `-g`. As we will discuss in upcoming sections, LIBSVM^{aggr} provides a methodology to estimate the value of this parameter. To indicate the intention to estimate the value of γ, the flag should be entered with the value `-1`. Let's see an example:

```
<INDICES   RBF   -g -1  # Gamma estimation
<INDICES   RBF   -g 3.3
```

Univariate kernel

We will specify the use of the Univariate kernel using the `<KTYPE> UNIVARIATE`. This function requires a hyperparameter α (positive double). In addition to the hyperparameters, it is necessary to enter all possible categories associated with the attribute, separated by a whitespace. To specify the value of α, we will use the flag `-a`, while the flag `-c` should be used to specify the categories.

Note that this kernel does not allow working with vectors of categorical variables. Therefore, instead of entering a list of indices, only one index will be accepted.

In addition to the mandatory parameters, the Univariate kernel, as we mentioned in its theoretical definition, uses the probabilities of obtaining each category to measure the similarity between samples. LIBSVM^{aggr} provides an automatic method to compute these probabilities using the training data. However, the user can optionally set the probabilities for each category using the flag `-p`. Here are some examples of parameterization:

```
<INDEX>   UNIVARIATE   -a 0.5  -c category1 category2 category3
<INDEX>   UNIVARIATE   -a 1.0  -c category1 category2  -p prob1 prob2
```

Spectrum kernel

The last kernel we have implemented in the library is the p-Spectrum kernel. As the name suggests, it has a hyperparameter `p` (positive integer) that can be defined using the flag `-p`. Similar to the Univariate kernel, the Spectrum kernel does not allow handling vectors of string variables and, therefore, only one index will be accepted. To specify the use of this kernel in the configuration file, it is necessary to use the `<KTYPE> SPECTRUM`, as shown below:

```
<INDEX>   SPECTRUM   -p 3
```

Parameters estimation

During the training phase, estimating the optimal parameters and hyperparameters of SVMs can become a real challenge. For this reason, in addition to having implemented the aggregation kernel and completely restructured the library, we have decided to provide specific procedures to help the user estimate some of these algorithm parameters. Below, we will detail these procedures to facilitate a simpler and more effective use of the library.

Firstly, we have developed a methodology to estimate the value of the parameter C in a C-SVM. This procedure is based on applying the Cross-Validation technique using the training data, with a specific number of partitions (folds) and a set of predefined C values.

The value of C that offers the best results will be selected as the final value for the training process. The user is free to determine the number of partitions and the values of C to use through the options `-v` and `-i` in the library execution, respectively. To indicate that the value of the parameter C is to be estimated, it will be necessary to enter the option `-c -1` through the command line when running the SVM training program.

Secondly, we have provided a methodology to estimate the hyperparameter γ of the aggregation kernel. This method is based on calculating the similarities using the average kernel (see equation 5.1) for 5000 random samples from the training set. The similarities are sorted from lowest to highest and the value of γ is set as the average between the first and third quartile of these sorted similarities. Similar to the case of the parameter C, if the value of the hyperparameter γ is to be estimated, it will be necessary to enter the option `-g -1` on the command line when running the `svm-train` program.

To estimate the γ of RBF kernels, we have also provided a procedure similar to the estimation of the γ of the aggregation kernel. In this case, instead of using the average kernel to calculate the similarities of the selected samples, the squared Euclidean distance is used. As we have mentioned in the description of the configuration file, to indicate that the value of the parameter γ of an RBF subkernel is to be estimated, it will be necessary to enter the option `-g -1` in the corresponding line of the configuration file.

The addition of these parameter estimation procedures to the library represents a notable simplification of use for users as it allows for a more agile and efficient configuration of the algorithms. With these integrated tools, the need for additional calculations or reliance on external resources for optimal parameter adjustment is significantly reduced, saving time and effort during the development and analysis of machine learning predictive models.

Overall, this implies a substantial simplification of the algorithm configuration task and provides professionals with the convenience of using a comprehensive and autonomous solution for parameter optimization, eliminating the need to work with multiple tools or depend on external procedures.

5.4 Implementation of the Bagging-SVM

We have successfully implemented the new Bagging-SVM algorithm (see Section 4.2) in our library, making several modifications for this purpose. As mentioned in previous sections, using Bagging-SVM involves training a set of m models using subsets of the input data, determining their size as stated in equation 4.3.

Implementing this technique in the library has not been a trivial process. We had to reinterpret the way models are generated, trained and stored. It has been a complex and delicate task in terms of implementation. Not only have we incorporated the functionality of Bagging-SVM into LIBSVM^{aggr}, but we have also ensured the coherence and effectiveness of the algorithm to achieve optimal results.

In order to use the Bagging-SVM in our library, the command `-b num_models dirPath` needs to be specified during the execution of `svm-train` program. This command indicates the library that we want to train a Bagging-SVM using `num_models` models and, once they have been generated through multiple training processes, store them in `dirPath` directory. The library will automatically process this information and train the set of models that will constitute the committee of experts for Bagging-SVM.

In the training of Bagging-SVM, we have also provided methodologies to estimate some of the hyperparameters for each generated submodel. Note that now it will be necessary to estimate the parameters for each model, as each one will be trained from a different subset of the input data. Similar to the conventional SVM algorithm, the parameters C of the C-SVM, γ of the aggregation kernel and γ of the RBF subkernels will be estimated. The user can indicate their desire to perform this estimation by setting the parameter value as `-1`.

Regarding predictions, it is necessary to use the command `-b dirPath` in the execution of the `svm-predict` program. This way, the library will identify that predictions need to be made using Bagging-SVM and classify the testing samples using the models as the committee of experts as stated in Section 4.2.

Once we have implemented the training and prediction methodologies of the new algorithm, we conducted a preliminary test to evaluate its behavior and ensure its correctness. The obtained results were positive and promising. It seems reasonable to think that this algorithm has potential and can be useful in many situations involving large amounts of data. In subsequent sections, we will carry out a thorough experimentation with the new software to confirm its proper functioning. During this phase, we will analyze the performance of the new Bagging-SVM in detail to fully validate its effectiveness and usefulness.

5.5 Other tools

In addition to the previously mentioned functionalities, we have made several other significant modifications to our SVM library that we consider important but have not described earlier to simplify the explanation. Below, we will enumerate these modifications and provide a detailed explanation of each.

Firstly, we have implemented a new class called `MessageHandler` in the library. This class allows us to manage all the messages generated at any point in the library. The class stores pointers to the functions responsible for releasing the global parameters of the library when execution ends. This way, in the event of an unforeseen error or simply when the execution is completed, we can ensure that there will be no memory leaks or similar issues. Additionally, the class handles debugging messages and notifications of the status of parameter estimation, training, prediction steps, among others.

Another important modification that has not been previously mentioned is the data reading methodology. In LIBSVM, this task was performed using methods like `strtok` and similar ones. The new version LIBSVM^{aggr} allows for new types of data with different characteristics and formats. This fact has rendered the reading procedures obsolete. Consequently, we have completely rethought the way data is read, utilizing new functionalities of the C++17 language, such as regular expressions.

Another challenge we have faced is the reading of the configuration file. One of our main motivations was to ensure that the library is easily extensible. For this reason, we have invested significant efforts in determining the best way to read and parse the contents of this file. If we had established specific reading methods for the configuration of each kernel, users implementing new kernel functions would have had to face the complexity of implementing complicated and error-prone reading and parsing methodologies.

To overcome this challenge, we have implemented the `ConfigStorage` class. This class is responsible for reading the content of the configuration file line by line and storing the indexes, kernel types and their parameters in an intuitive and accessible manner. The `ConfigStorage` class contains three main variables: `kernel_id`, which is a character string that identifies the kernel (the `<KTYPE>` keyword defined in Section 5.3), `indexes`, which is an array of integers containing the indexes of the features associated with the respective kernel and `params`, which is an `std::map` where the keys represent the different flags (parameters) of the kernel and the values are instances of `std::vectorstd::string` that contain the parameters in string format.

Each derived class of `kernel_param` will receive an instance of `ConfigStorage` to initialize the kernel parameters. This way, if a user needs to create a new kernel function, they will know that the parameters will always have the specified format and can appropriately analyze and validate them without worrying about their reading and subsequent parsing.

With these implementations, LIBSVM^{aggr} has significantly enriched itself, providing essential functionalities for message management, a more flexible and robust data reading system and an extensible and solid configuration system. We consider these improvements reinforce the robustness, utility and manipulability of our library.

6 Experimentation

Upon reaching this point, we have designed and described the kernel function of the Aggregation Kernel (Section 3.2) and completed its implementation in the LIBSVM library (Section 5.3). Additionally, we have also proposed a modification to the conventional Support Vector Machines algorithm based on Bagging techniques to improve the efficiency of these algorithms when dealing with large amounts of data (Section 4.2). Similar to the Aggregation Kernel, we have included the implementation of Bagging-SVM in the LIBSVM library (Section 5.4).

In this chapter, we will propose a series of experiments to evaluate the performance of the new software in different contexts. The primary objective is to determine if the improvements implemented in the LIBSVM software are significant and provide good results.

6.1 Experiment 1: Basic functionality

The first thing we want to test is that the basic functionality of the new software is correct. As we have mentioned in previous sections, although we have established the Aggregation Kernel as the main kernel of the algorithm, when using a single kernel, the original aggregation methodology (see equation 3.16) will not be applied. Instead, this single kernel will be used to evaluate the similarity between different samples (see equation 5.2). This implies that in these situations, we can compare the behavior of LIBSVM^{aggr} with the original version of LIBSVM.

Hypothesis

Our hypothesis is that by parameterizing LIBSVM^{aggr} and LIBSVM in the same way (*i.e.* using the same type of SVM and exactly the same parameterized kernel), the results in terms of accuracy should be the same. However, it is expected that the execution time will increase due to, among other factors, the additional data structure accesses required by the new version.

If our hypothesis is confirmed, it would mean that the new features implemented in LIBSVM^{aggr} do not affect the standard functioning of the SVM algorithm. This would imply that we can use LIBSVM^{aggr} and LIBSVM interchangeably in single-kernel problems with continuous data.

Circular data generator

The data we will use will be synthetically generated and of continuous type. This way, we can use them in both LIBSVM^{aggr} and the original version. To have maximum control over the data generation process, we have implemented a data generation Python script, which can be found at `data/data-generator.py`.

The data generator implements a procedure to create samples distributed into three classes. The samples will be grouped in the form of circumscribed circles. It is possible to generate samples in two or three dimensions. For each dimension, the script allows setting a noise multiplier in the range of $[0, 1]$. The program also enables data division into training and testing sets (by specifying the percentage of data to be included in the training set), displaying a portion of the data in a graph and saving the data in LIBSVM format. An example of data generation can be seen in Figure 6.1.

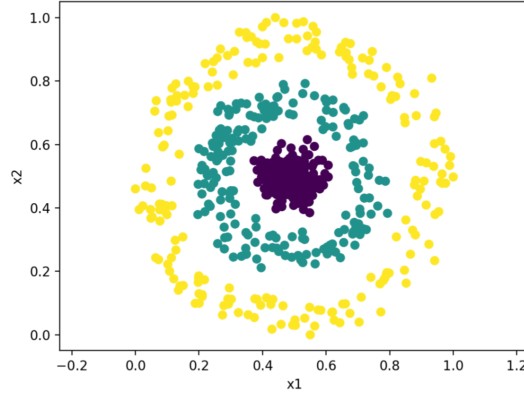


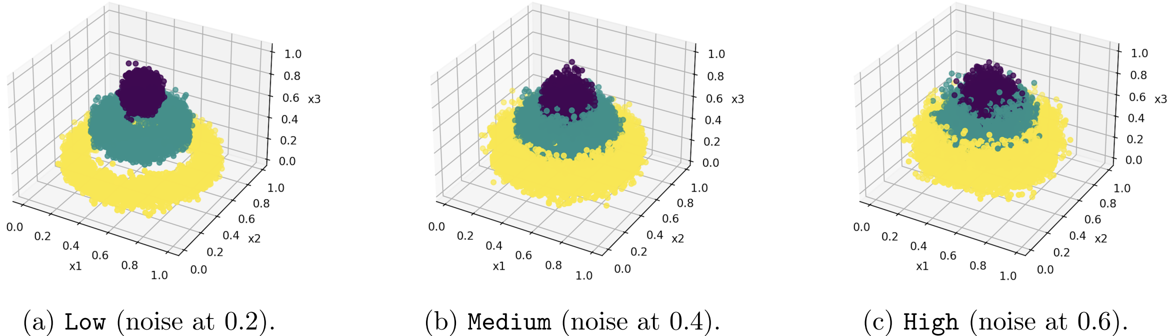
Figure 6.1: Example of synthetic 2D data generated from the circular data generator script.

Note: The samples have been generated in two dimensions with 200 instances per class. The noise multiplier is set to 0.2 in both dimensions (Source: Own elaboration).

Procedure and results

The procedure associated with this first experiment is as follows: we will start by generating three data sets using the circular data generator described earlier. For each data set, we will set a different noise level: `Low` (with a noise multiplier of 0.2 in each dimension), `Medium` (with a noise of 0.4 in each dimension) and `High` (with a noise of 0.6 in each dimension).

All three data sets will consist of a total of 50100 samples (16700 samples per class), of which 40080 will be part of the training subset and 10,020 will be part of the testing subset. The representation of the three data sets can be seen in Figure 6.2 (for simplicity, only 40% of the data has been plotted).



(a) `Low` (noise at 0.2). (b) `Medium` (noise at 0.4). (c) `High` (noise at 0.6).

Figure 6.2: Graphical representation of the datasets used in Experiment 1.

Source: Own elaboration.

For each data set, we will train a C-SVM with a constant parameter value of $C = 10$. We will run LIBSVM^{aggr} and its original version using three different kernels for continuous data: Linear, Polynomial and RBF (Radial Basis Function) kernels. For the Polynomial and RBF kernels, we will use three and four different parameterizations, respectively (note that the Linear kernel has no hyperparameters and therefore cannot be parameterized). A summary of the parameters used in each case can be seen in Table 6.1.

Table 6.1: Parametrizations used for each kernel in Experiment 1.

Kernel	Parameters	Param. 1	Param. 2	Param. 3	Param. 4
Linear	–	–	–	–	–
Polynomial	$(d,\ \gamma,\ c_0)$	$(2,\ 0.1,\ 10)$	$(3,\ 0.1,\ 10)$	$(3,\ 10,\ 100)$	–
RBF	γ	0.01	0.1	10	100

Note: In all cases, the C parameter of the C-SVM has been set to 10.
Source: Own elaboration.

We can observe the results obtained from the aforementioned execution in Tables 6.6 for the Linear Kernel, 6.7 for the Polynomial Kernel and 6.8 for the RBF Kernel. As can be seen, the accuracy percentages coincide for all the executions performed using our implementation LIBSVM^{aggr} and the original software.

One might think that, despite obtaining the same accuracies, the confusion matrices could differ. To rule out this possibility, we generated the confusion matrices for each execution and verified that they match perfectly. Due to space constraints, we will not include all the matrices in this document, but you can see two comparisons in Tables 6.2 and 6.4. In all tables, the rows represent the predicted labels, while the columns represent the true labels of the samples.

Table 6.2: Experiment 1 Confusion Matrices (Polynomial kernel with $d = 3,\ \gamma = 0.1,\ c_0 = 10$).

(a) Using LIBSVM^{aggr}

	Class 1	Class 2	Class 3
Class 1	3199	141	0
Class 2	244	2949	147
Class 3	0	176	3164

(b) Using original LIBSVM

	Class 1	Class 2	Class 3
Class 1	3199	141	0
Class 2	244	2949	147
Class 3	0	176	3164

Table 6.4: Experiment 1 Confusion Matrices (RBF kernel with $\gamma = 0.01$).

(a) Using LIBSVM^{aggr}

	Class 1	Class 2	Class 3
Class 1	2677	649	14
Class 2	671	2023	646
Class 3	25	668	2647

(b) Using original LIBSVM

	Class 1	Class 2	Class 3
Class 1	2677	649	14
Class 2	671	2023	646
Class 3	25	668	2647

Having analyzed the results, we consider our hypothesis proven. It seems reasonable to assume that if we use the same parameterization in LIBSVM^{aggr} and LIBSVM, we obtain exactly the same results. Therefore, we believe that the base implementation of our software version maintains correctness in both the training and prediction phases of the SVMs.

As we had anticipated, the execution time appears to be higher when using LIBSVM^{aggr}. However, we consider that performing a single execution with different parameterizations is not sufficient to estimate the temporal overhead introduced by our version compared to the use of LIBSVM. In the next experiment, we will approximately study the percentage that this overhead represents in relation to the original execution time.

In this early stage of testing the new library, the results obtained are promising and we are eager to continue experimenting. Thus far, everything appears to be functioning perfectly, demonstrating the effectiveness of the implemented features. We look forward to conducting further experiments and validations to gain deeper insights into the performance and capabilities of the software.

Table 6.6: Results of Experiment 1 using the Linear Kernel.

Noise level	Parameters	Accuracy (%)		Execution time (s)	
	–	Original	Aggregation	Original	Aggregation
Low	–	100.000	100.000	0.087	0.728
Medium	–	85.838	85.838	7.338	11.761
High	–	72.725	72.725	15.489	23.031

Note: The $C = 10$ value of the C-SVM is constant.
Source: Own elaboration.

Table 6.7: Results of Experiment 1 using the Polynomial Kernel.

Noise level	Parameters			Accuracy (%)		Execution time (s)	
	d	γ	c_0	Original	Aggregation	Original	Aggregation
Low	2	0.1	10	100.000	100.000	0.067	0.724
	3	0.1	10	100.000	100.000	0.100	0.716
	3	10	100	100.000	100.000	0.050	0.725
Medium	2	0.1	10	91.766	91.766	8.510	12.052
	3	0.1	10	98.393	98.393	6.278	8.370
	3	10	100	98.922	98.932	15.547	16.910
High	2	0.1	10	83.124	83.124	17.026	23.592
	3	0.1	10	92.934	92.934	11.646	15.242
	3	10	100	93.523	93.543	57.083	61.721

Note: d, γ and c_0 are the degree, gamma and coeficient hyperparameters of the polynomial kernel. The $C = 10$ value of the C-SVM is constant.
Source: Own elaboration.

Table 6.8: Results of Experiment 1 using the RBF Kernel.

Noise level	Parameters	Accuracy (%)		Execution time (s)	
	γ	Original	Aggregation	Original	Aggregation
Low	0.01	100.000	100.000	0.872	2.052
	0.1	100.000	100.000	0.220	0.956
	10	100.000	100.000	0.111	0.767
	100	100.000	100.000	0.841	1.621
Medium	0.01	86.267	86.267	14.397	26.710
	0.1	95.898	95.898	8.850	14.124
	10	98.882	98.882	1.317	2.243
	100	98.802	98.802	2.412	3.984
High	0.01	73.323	73.323	25.445	37.824
	0.1	89.651	89.651	17.560	24.352
	10	93.473	93.473	6.649	8.641
	100	93.323	93.323	10.487	13.520

Note: γ is the gamma hyperparameter of the RBF kernel. The $C = 10$ value of the C-SVM is constant.
Source: Own elaboration.

6.2 Experiment 2: Execution time overhead

In the previous experiment, we have observed that the use of the new version LIBSVM^{aggr} introduces an overhead in the execution time. This behavior is expected primarily due to the restructuring of the data structures that we had to implement in order for the library to work with mixed data.

One possible cause of this overhead is the increase in the number of data structure accesses in LIBSVM^{aggr} compared to LIBSVM. For instance, in the original version, we only needed to access the node values (`node.value`). On the other hand, in LIBSVM^{aggr}, we first have to navigate through the different `feature_sets` to reach the specific kernel. Then, we need to access the specific kernel function and perform various `static_casts` (since `svm_node` is now an abstract structure) in order to obtain the correct node type. At first glance, these additional accesses may seem negligible. However, it should be noted that we are dealing with extremely large kernel matrices. Therefore, increasing the number of accesses is critical.

Hypothesis

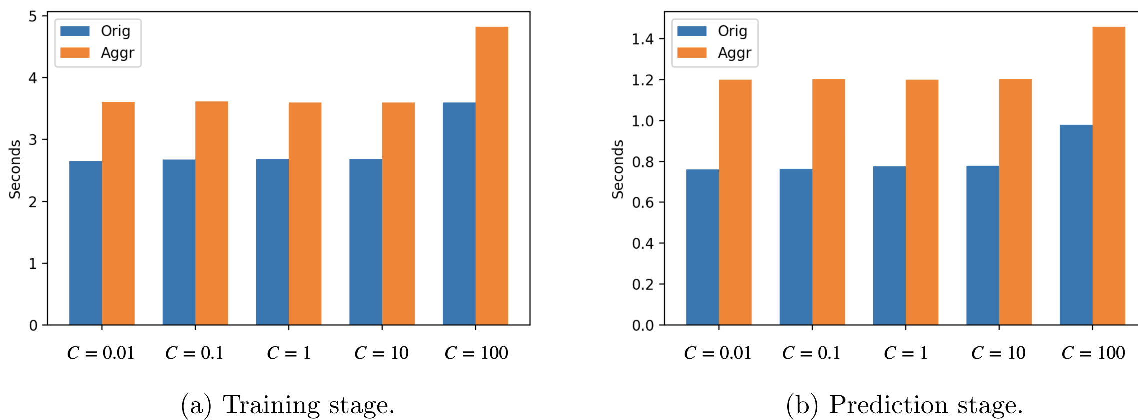
In this experiment, we aim to estimate the overhead introduced in the execution time when comparing the use of LIBSVM^{aggr} with the original version of LIBSVM. Our hypothesis is that the execution time will increase due to, among other factors, the additional accesses required for the library to handle mixed data.

Procedure and results

To conduct this experiment, we will use the `Medium` dataset from the previous experiment (see Figure 6.2b), which is generated with a noise multiplier of 0.4 (see Section 6.1). Once again, we will employ a C-SVM with different parameterizations of the Linear, Polynomial and RBF kernels. For each parameterization, we will perform a total of 50 repetitions.

Based on these results, we will calculate the average training time and prediction time, presenting them in separate graphs. Finally, we will estimate the percentage of extra time required when using LIBSVM^{aggr} compared to the base execution time obtained with LIBSVM.

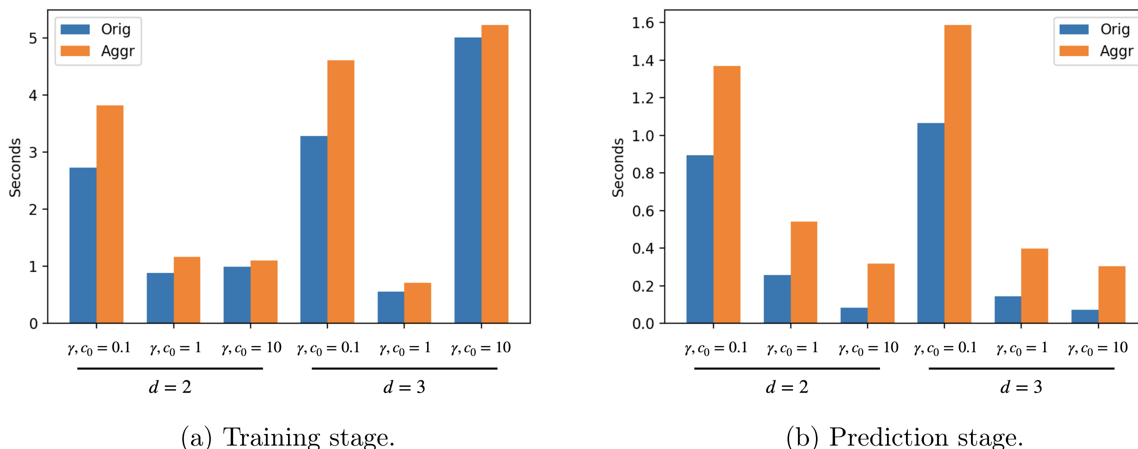
You can observe the graphs representing the execution times for each parameterization of the Linear, Polynomial and RBF kernels in Figures 6.3, 6.4 and 6.5, respectively.



(a) Training stage. (b) Prediction stage.

Figure 6.3: Average execution time obtained in Experiment 2 using the Linear kernel.

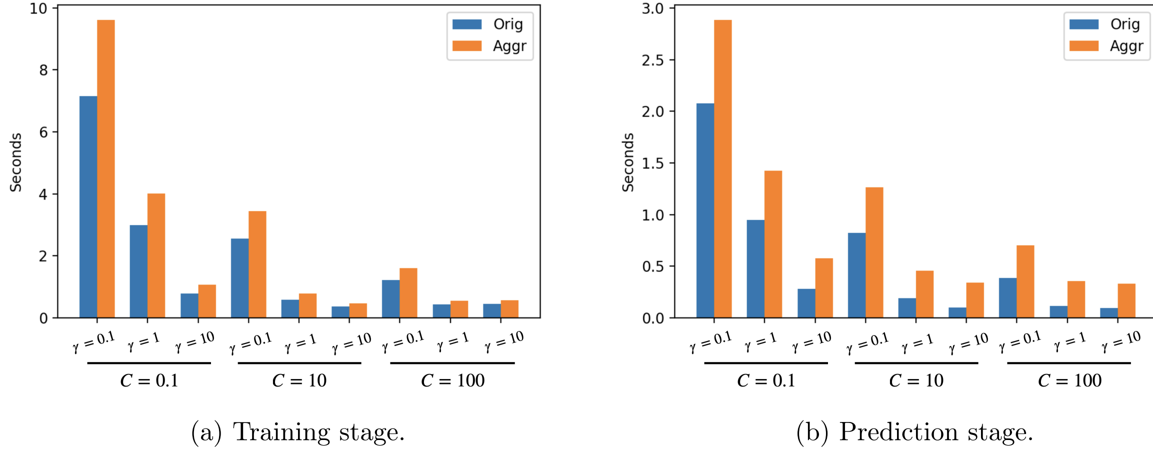
Note: Each bar set corresponds to a different parameterization of the SVM. Each bar represents the average time of executing 50 repetitions (Source: Own elaboration).



(a) Training stage. (b) Prediction stage.

Figure 6.4: Average execution time obtained in Experiment 2 using the Polynomial kernel.

Note: The $C = 10$ parameter of the C-SVM is constant. Each bar set corresponds to a different parameterization of the SVM. Each bar represents the average time of executing 50 repetitions (Source: Own elaboration).



(a) Training stage.

(b) Prediction stage.

Figure 6.5: Average execution time obtained in Experiment 2 using the Linear kernel.

Note: Each bar set corresponds to a different parameterization of the SVM. Each bar represents the average time of executing 50 repetitions (Source: Own elaboration).

In general, we consider our hypothesis to be proven. It seems reasonable to assume that the use of the new software, LIBSVM^{aggr}, introduces an overhead in the execution time, both in the training and prediction stages. We can also observe that, as expected, the prediction time remains lower compared to the training time.

The percentage of extra time incurred by using LIBSVM^{aggr} can be seen in Table 6.9. Interestingly, the percentage of extra time is higher in the prediction phase. We believe this difference is caused by the way the kernel function to analyze each feature vector of the different samples is identified. However, it is important to note that, in general, the prediction time is much lower than the training time. Therefore, this overhead should not have a significant impact on the overall execution time of the SVM.

Table 6.9: Percentage of additional time required by LIBSVM^{aggr} compared to LIBSVM.

Kernel	Train (%)	Predict (%)
Linear	34.71	55.62
Polynomial	23.62	79.56
RBF	33.52	66.10

Source: Own elaboration.

6.3 Experiment 3: Bagging-SVM use case

Once the baseline effectiveness of our LIBSVM^{aggr} implementation has been proven (Sections 6.1 and 6.2), we want to test the performance of the Bagging-SVM algorithm described in Section 4.2. This algorithm, based on Bagging theory, aims to reduce the complexity of training a Support Vector Machine (SVM) when dealing with large amounts of data.

Hypothesis

Our hypothesis is that the training time of the Bagging-SVM, using a significantly large dataset of size n and a total of m models comprising the committee of experts, should be of the order of $O(pn^2/m)$, as estimated in the theoretical study of Bagging-SVM complexity (Section 4.2).

In terms of prediction time, unlike the training time, we believe that it will increase. This assumption seems reasonable since the Bagging-SVM algorithm requires m predictions for each training sample, whereas the conventional SVM algorithm only requires one prediction per sample.

Finally, regarding the error rate, we believe that for relatively low values of m, it should not be significantly affected. We consider that the error rate should remain stable as long as the total number of data points is sufficiently high and, consequently, the number of samples used in the training of each model is large.

Procedure and results

The objective of this experiment is to analyze the evolution of the execution time (in both the training and prediction stages) and the prediction error of the Bagging-SVM algorithm as we increase the number of models m used in the execution. The procedure will involve training the pseudo-model (committee of experts) multiple times using $n \gg 2 \cdot 10^5$ data points and progressively increasing the number of models m.

To ensure that the measurement of execution times is meaningful and robust, we will conduct a total of R repetitions for each value of m. The choice of the value of R is not trivial. It should be noted that training a single SVM with such a large amount of data can result in a long execution time, possibly taking several hours. With many repetitions, the execution could tie up our computer for days. Due to time constraints of the project, we cannot afford such a delay. Therefore, we have determined that a value of $R = 3$ will be sufficient to obtain representative average execution times while keeping the overall execution time manageable.

Another important factor to consider is the choice of values for m to test. Having conducted a preliminary study on the behavior of Bagging-SVM, we have found that variations in m when $m < 100$ have a more significant impact on the training time compared to variations when m is large. Therefore, we have decided to establish three ranges of increasing values for the number of models to be used, which we believe will allow us to clearly observe the evolution of the metrics under study:

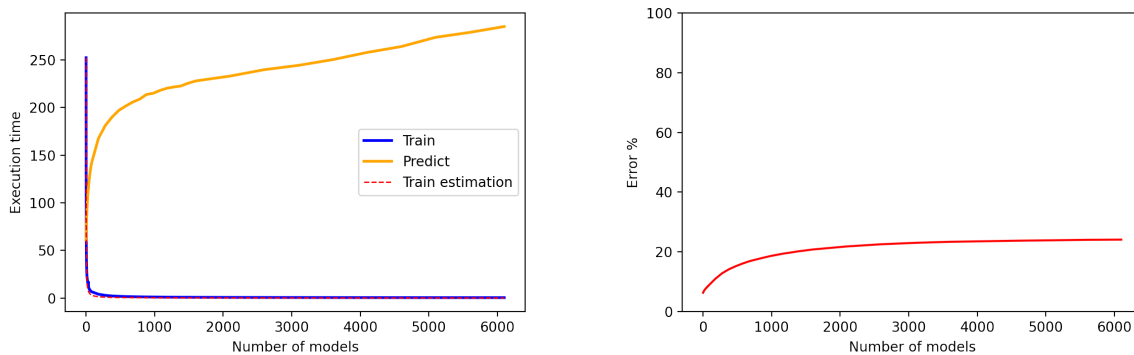
1. From $m = 1$ to $m = 70$ by steps of length=1 (*i.e.* $m = 1,\ 2,\ 3\ \ldots\ 79$).
2. From $m = 80$ to $m = 1480$ by steps of length=100 (*i.e.* $m = 80,\ 180,\ 280\ \ldots\ 1480$).
3. From $m = 1600$ to $m = 6100$ by steps of length=500 (*i.e.* $m = 1600,\ 2100,\ 2600\ \ldots\ 6100$).

As mentioned earlier, for each value of m, we will conduct $R = 3$ repetitions and collect the training and prediction execution times, as well as the obtained error rate.

The last factor to determine is the dataset to be used. As we mentioned, the main requirement is that it contains a significantly large number of data points (on the order of $n \gg \cdot 10^5$). Initially, we considered using a real-world dataset. Although we were able to gather some, either they required extremely meticulous preprocessing or they did not meet the minimum required number of samples. Therefore, we have decided to use our circular data generator again (see Section 6.1).

The advantages of using the data generator are that we can instantiate as many samples as needed and introduce noise to make the problem resolution non-trivial. On the other hand, the data generated by the script are continuous, so we will not be able to fully exploit the implemented Aggregation Kernel. Nonetheless, although we would have liked to test the functionality of the new kernel as well, this limitation is not a significant concern because the sole purpose of this experiment is to evaluate the performance of Bagging-SVM.

The generated data consists of a total of $4.5 \cdot 10^5$ instances ($1.5 \cdot 10^5$ per class), of which $3.6 \cdot 10^5$ form the training set and $9 \cdot 10^4$ form the testing set. The noise multiplier for each dimension of the data is set to 0.4. To train the sub-models (forming the committee of experts) of the Bagging-SVM, we will use a C-SVM with the parameter $C = 100$ and the RBF kernel. The value of the hyperparameter γ of the kernel function will be estimated for each sub-model using the methodology outlined in Section 5.3.



(a) Execution time of train and predict stages as a function of the number of generated models.

(b) Error rate obtained as a function of the number of generated models.

Figure 6.6: Execution time and error rates obtained in Experiment 3.

Source: Own elaboration.

In Figure 6.6, we can observe the results obtained from the experiment. First, let's analyze the graph in Figure 6.6a. Here, we can see the different execution times for training (blue), prediction (orange) and the estimated training time (dashed red) based on the complexity of Bagging-SVM that we had estimated (Section 4.2).

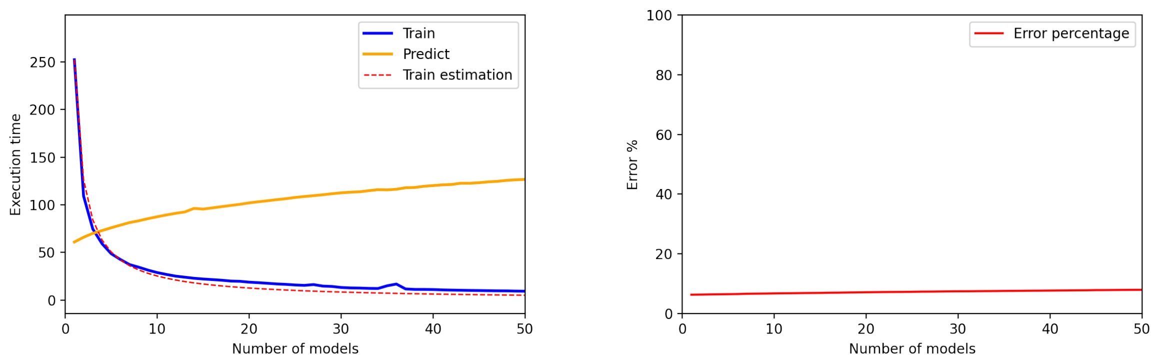
On one hand, one of the most immediate observations when looking at the training cost trend is that once we surpass the barrier of $m = 2$ models, the execution time decreases drastically. Furthermore, it appears to follow the function of Bagging-SVM complexity. In fact, it can be inferred that we don't even need to reach 100 models per execution for the training time to stabilize.

On the other hand, we can see that the cost of predictions increases as we increment the number of models, as we had predicted in our initial hypothesis. Nevertheless, the rate of growth in execution time is more moderate in this case compared to the rate of decrease in training time.

Finally, regarding the error (Figure 6.6b), we also observe an upward trend as we increase the number of models employed. However, the growth of the error is not comparable in any way to the speed of growth and decrease observed in the previous graph. As we mentioned in the hypothesis definition, this moderate increase in error is justified.

It should be noted that for low values of m, each model still has a large number of samples and therefore, the committee of experts can generate considerably reliable predictions. However, as we increase the value of m, each model has very few samples, resulting in significantly biased predictions and consequently, the decisions of the committee of experts tend to have more error.

Although the graphs in Figure 6.6 seem to support our hypotheses, it is evident that the critical values of m in terms of execution time are located in an approximate range of $m \in [1, < 100]$. Therefore, we consider it interesting and enriching to analyze the behavior of the algorithm on a smaller scale. You can find the enlarged graphs for the range $m \in [1, 50]$ in Figure 6.7.



(a) Execution time of train and predict stages as a function of the number of generated models.

(b) Error rate obtained as a function of the number of generated models.

Figure 6.7: Execution time and error rates obtained in Experiment 3 (range $[1, 50]$).

Source: Own elaboration.

Firstly, if we observe Figure 6.7a, we can see how, indeed, when transitioning from using a single model (conventional SVM) to just two models, the training time is reduced by almost half. At this scale, the prediction time increases by just over two seconds. We can appreciate a significant reduction in training time without perceiving major variations in the increase in prediction time until the generation of $m = 10$ models.

Secondly, in Figure 6.7b, we can clearly see how the increase in error in the initial executions is negligible. As mentioned earlier, the improvement in execution time becomes evident in the range $m \in [1, 10]$. Within this range, the error rate does not increase even by 0.5% (0.44%). In fact, the difference in error between executions with $m = 1$ and $m = 50$ does not exceed 2% (1.65%).

In conclusion, the results of the experiment allow us to confidently state that our initial hypothesis has been proven. The Bagging-SVM has demonstrated itself as a viable and promising alternative for tackling problems with large amounts of data using SVMs.

Our study has revealed a significant reduction in training time for the Bagging-SVM compared to the conventional SVM. Furthermore, the increase in prediction time has been moderate, indicating that this technique can be considered for cases where training time is a critical factor.

We are excited to continue testing the performance of the Bagging-SVM in future projects. There are numerous real-world areas and problems where this promising tool could be applied. Further exploration through these experiments will provide a better understanding of its capabilities and limitations and explore its potential across various scientific and technological disciplines.

6.4 Experiment 4: Mixed data real-world problem

This fourth and final experiment is focused exclusively on evaluating the results obtained from applying the Aggregation Kernel to a real-world problem. Additionally, we aim to assess whether the level of expressiveness provided by this new kernel, in terms of measuring similarity between samples, is sufficient to outperform the algorithm and pre-established kernels in LIBSVM.

Hypothesis

The main hypothesis established in this experiment is simple: we believe that using a specific kernel based on the nature of each variable (or vectors of variables) can yield better results than using a single kernel when dealing with mixed data in real-world problems.

As mentioned in previous sections, we believe that the expressiveness and analytical capability offered by evaluating each variable type, taking into account their inherent properties and semantics, can be significant in the domain of problem-solving using Support Vector Machines (SVMs).

In addition to the main hypothesis, we also want to propose a secondary hypothesis based on the following: we believe it is reasonable to think that if we have a vector of variables of the same type, it may be beneficial to divide this vector into different groups of variables that exhibit similar structure or semantics when assigning kernels in the Aggregation Kernel. We do not have theoretical foundations to justify this premise, but it may make sense when analyzing the similarity between different samples.

Aircraft dataset

For conducting this experiment, we will utilize a real-world dataset referred to as the Aircraft Dataset. This dataset has been extracted from [42]. The data was graciously provided by Eduard Morillo, a student enrolled in the Bachelor's Degree program in Data Science and Engineering (FIB, UPC). In return, we furnished a preliminary copy of the LIBSVM^{aggr} software to facilitate its utilization in his project.

The dataset comprises 22 features of diverse typologies (continuous, categorical and textual or string) concerning 345 combat aircraft operational between the years 1928 and 1947. The information was programmatically collected from various specialized sources.

Significantly, this dataset is currently unobtainable in any online repository, rendering it an exclusive resource for our study. Moreover, since no prior works have employed this particular dataset, direct comparisons with existing solutions are not feasible. Nonetheless, the data exhibits high quality and fulfills all the requisite criteria for our research objectives.

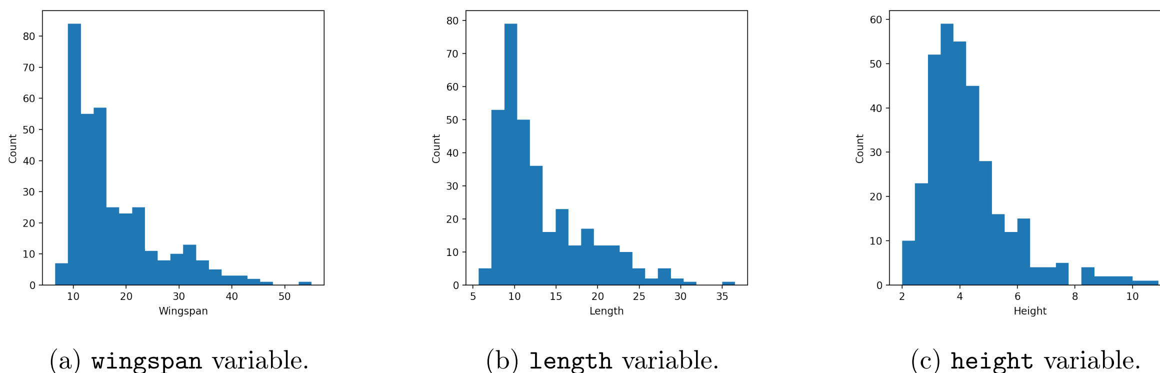
The inherent problem within the dataset revolves around classification. More precisely, we aim to reasonably infer the country of origin for an aircraft based on its distinctive characteristics. Alternatively, an auxiliary objective involves determining the aircraft type.

Prior to addressing the core problem, we will conduct a concise inspection and preprocessing phase on the dataset. Initial scrutiny has revealed the presence of four anomalous samples. Given their negligible representation within the dataset, we have opted to remove them, resulting in a refined dataset comprising 341 samples. Subsequently, we will perform a brief analysis encompassing variable types and their corresponding distributions.

The only textual variable is `plane`. This feature contains the name of each aircraft. Some examples include `Airspeed Oxford Mk.I`, `Yakovlev Yak-9D` and `Blohm und Voss Bv.138 C-1`.

Regarding the numerical variables, there are a total of 18. Some examples include `year` (year of manufacture), `wingspan`, `length`, `height`, `engines` (number of engines) and `cannons` (number of cannons). In general, the numerical features encode technical characteristics of the aircraft, such as crew capacity, payload or bomb load capacity, among others.

Due to space limitations in the project, we will not include all the graphs depicting the distribution of the numerical variable values. Nonetheless, it is worth mentioning a few. Firstly, in Figure 6.8, we can observe the histograms of the `wingspan`, `length` and `height` variables. It is interesting to note that all three variables exhibit relatively similar distributions. The distributions appear to be not far from a normal distribution, although slightly right-skewed.



(a) `wingspan` variable. (b) `length` variable. (c) `height` variable.

Figure 6.8: Distribution of `wingspan`, `length` and `height` features of the Aircraft Dataset.

In contrast, we can observe that the variable `serviceCeiling` (Figure 6.9) exhibits a distribution more closely resembling a standard normal distribution. We have chosen not to modify the original features. However, to correct the skewness of the aforementioned variables, we could compute their logarithms.

Regarding the aircraft armament, in Figure 6.10 we observe that the majority of aircraft tend not to have machine guns or cannons. We could consider generating a binary variable, particularly for `cannons` (Figure 6.10a), indicating whether they have machine guns or cannons. However, we believe that this level of specificity in encoding the armament can be beneficial in solving the problem.

To conclude the analysis of numerical variables, we will conduct a study of the correlation between all the numerical variables. The correlations can be seen in Figure 6.11. We can observe that there are some variables that are significantly correlated. For example, `wingspan`, `length`,

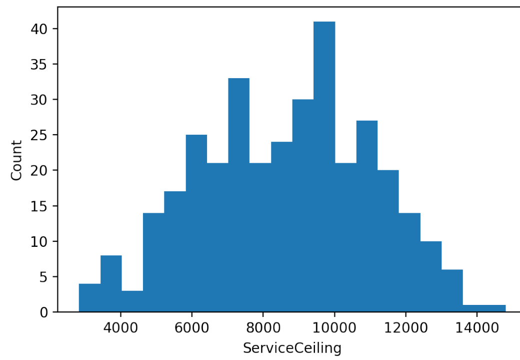
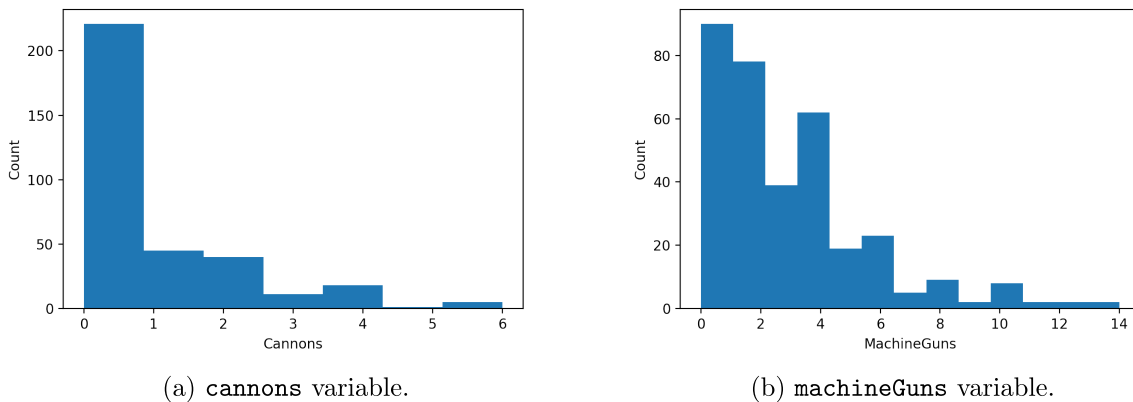


Figure 6.9: Distribution of `serviceCeiling` variable of the Aircraft Dataset.



(a) `cannons` variable.

(b) `machineGuns` variable.

Figure 6.10: Distribution of `cannons` and `machineGuns` features of the Aircraft Dataset.

`height`, `engines` and `takeoffWeight`. This behavior is expected, as larger aircraft will also require more engines and be able to carry more payload during takeoff.

As for the targets of possible classification problems, we have two categorical variables: `country` and `type`. The distribution of counts can be seen in the graph in Figure 6.12. In the graph of countries (Figure 6.12a), it is evident that the majority of aircraft are from the United States (`USA`) and Germany (`D`). France (`F`) has the least representation, although we consider that there are enough samples to classify them correctly.

Regarding the aircraft types, we observe a less proportionate distribution of sample counts compared to the countries (Figure 6.12b). It is clear that the majority of aircraft are of type `Fighter` and `Bomber`. However, we have decided not to redistribute the categories in order to avoid biasing the results.

Finally, we have generated the train and test splits (with 80% and 20% of the data, respectively) and stored them in LIBSVM format. In order to compare the performance of the new software and verify that it indeed provides better results, we have created a dataset compatible with the original LIBSVM library. We have converted the categorical variables to numerical using the One-Hot Encoding technique. However, since we do not have a method to convert textual variables to numerical, we have excluded the `plane` variable from this dataset.

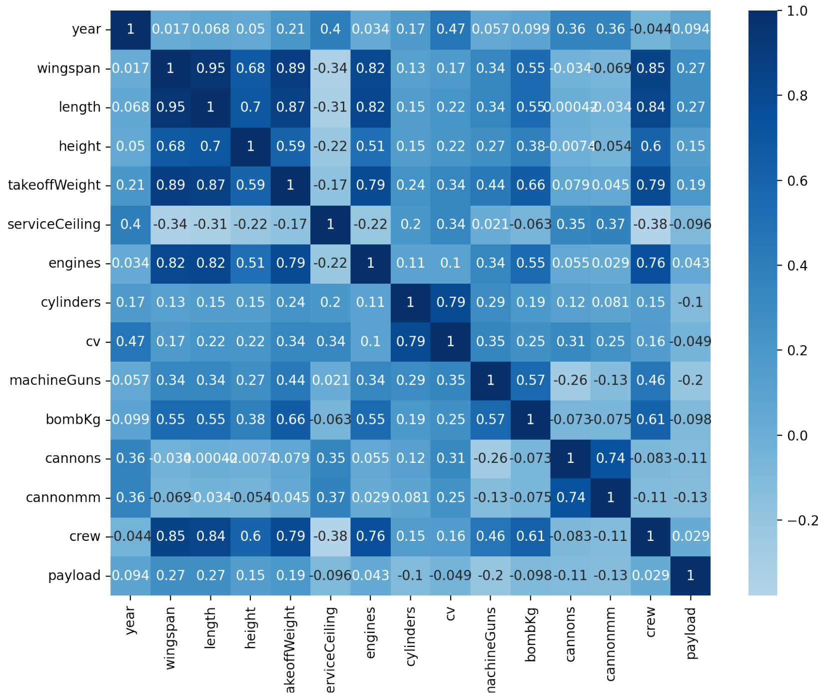


Figure 6.11: Correlation between numerical features of the Aircraft Dataset.

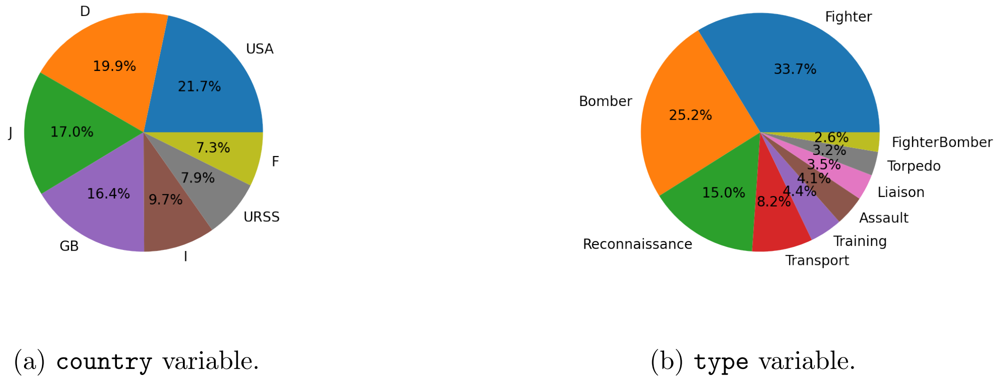
Procedure and results

The first step will be to run the original version of LIBSVM with the compatible data (*i.e.* the dataset containing only numerical variables). For the execution, we have chosen to use an RBF kernel with a C-SVM. Since the software does not provide any tools for parameter estimation, we have performed a 10-fold cross-validation process (using only the training data) with different parameterizations to estimate the most suitable values of C and γ beforehand.

Once the parameters have been estimated, we have executed the SVM with $C = 316.2278$ and $\gamma = 0.1$. The accuracy percentage obtained was 47.83%. These results do not seem particularly promising.

For the second execution, we want to use the Aggregation Kernel for the first time. In this initial test of the new kernel, we aim to group the numerical variables using different RBF kernels. We will distribute the features into different kernels based on their meaning. For example, we believe it would be beneficial to group variables such as `wingspan`, `length` and `height` together in one kernel. For the categorical variables, we will use the Univariate kernel. We will set the hyperparameter α to 1 since we do not have an effective technique to estimate its value. As for the textual variable `plane`, we will use the Spectrum kernel with a window size of $p = 5$ characters. You can see the configuration file used in Listing 6.1.

As we can see from the previous configuration file, the γ parameters of the different RBF kernels will be estimated using the procedure described in previous sections. Similarly, the C parameter of the C-SVM and the γ parameter of the Aggregation kernel will be automatically estimated through the estimation process implemented in LIBSVM^{aggr}.



(a) `country` variable.

(b) `type` variable.

Figure 6.12: Value counts of the `country` and `type` features of the Aircraft Dataset.

```
1  21
2  1 spectrum -p 5      # plane
3  2 univariate -a 1 -c Fighter Bomber Reconnaissance Transport Training Assault
      Liaison Torpedo FighterBomber        # type
4  3,7,8,9,13,14,15,16,19,20,21 rbf -g -1  # others
5  4,5,6 rbf -g -1      # wingspan, length, height
6  17,18 rbf -g -1      # cannon, cannonmm
7  10,11,12 rbf -g -1  # engines, cylinders, cv
```

Listing 6.1: Experiment 4 configuration file using textual features.

The results obtained in this execution are much better than expected, with an accuracy of 92.75%. This outcome is a clear demonstration of how the expressiveness allowed by the Aggregation Kernel translates into a significant improvement in results.

Despite the good results obtained in this second part of the experiment, we believe that the high accuracy percentage may be influenced by an unidentified strong correlation between the `plane` variable (which encodes the aircraft's name) and the `country` target. Recall that the Spectrum kernel evaluates the similarity between samples by computing substring matches of size p. Moreover, it is evident that aircrafts from Japan, for example, will have very specific combinations of characters in their names, which differ from those of Russian or American Aircraft.

Therefore, we have decided to carry out a new execution, this time without using the `plane` variable. The rest of the parameterizations (both the configuration file and the estimation of the C and γ_{rbf} and γ_{aggr} parameters) will remain the same as in the previous experiment. In this new execution, as expected, the accuracy has dropped to 49.28%. While it is true that we have surpassed the results of the execution with the original library, we believe that we can achieve better results with the Aggregation Kernel.

To verify if the reason for the drop in accuracy (besides the removal of the `plane` variable) is the grouping of variables based on their semantics, we have decided to perform one last execution. In this execution, we will associate each numerical variable with a different kernel (note that the Univariate kernel for categorical variables only accepts a single feature, so we do not need to modify anything in this regard compared to the previous execution). Once again, we will use the options to estimate all the parameters C and γ.

We consider the improvement achieved in these results to be quite significant: the accuracy has increased to 54%. Therefore, it seems that grouping variables based on their semantics is not necessarily beneficial. In this case, we can conclude that using a kernel for each variable and estimating their parameters provides more satisfactory results.

We want to conclude this experimental stage by checking if there are differences in predicting the aircraft type instead of the country of origin. Once the experimentation with the `plane` target is completed, it appears that identifying the country of origin of an aircraft based on its characteristics is rather challenging in general.

After reorganizing the datasets, we have retrained LIBSVM (manually estimating the C and γ parameters) and LIBSVM^{aggr} (using the parameter estimation implemented in the software). The accuracy obtained with the original software is 63.77%. In contrast, using the Aggregation Kernel, the accuracy reaches 74.34%.

The Aggregation kernel function appears to improve the results compared to the use of a single kernel function. The obtained results are promising, demonstrating a significant intuition for the effectiveness of the kernel. Although it is true that the dataset used has a limited number of samples, we believe that the results provide meaningful insights into the performance of the kernel. Due to time constraints, we were unable to further experiment with the new kernel, but we are eager to explore its potential in future projects.

7 Conclusions

At this point, we have completed all the stages of research, theoretical formulation and practical resolutions of the project. Therefore, it remains to state the conclusions we have drawn from its development.

In this final section, we will begin by analyzing the achievement of objectives and the fulfillment of requirements. Subsequently, we will analyze and evaluate the results obtained during the experimentation phase, drawing the necessary conclusions. Next, we will present our overall project conclusions. Finally, we will discuss what we believe will be the future work in the project's domain and conclude with a personal assessment of the work at hand.

7.1 Review of the objectives

In this first section, we will analyze the objectives we set at the beginning of the project and evaluate their achievement. Additionally, we will revisit both the functional and non-functional requirements and assess to what extent we have fulfilled them.

Regarding the **main objective** of the project, we have successfully formulated an effective kernel function capable of measuring similarity between vectors of mixed variables, which we have named the Aggregation Kernel (Section 3.2). This kernel function is simple, intuitive and designed to accurately capture the degree of similarity between the analyzed samples. We believe that we have conducted thorough work, validated its effectiveness and ensured that it meets the requirements of the stated objective.

Furthermore, we have not only limited ourselves to the theoretical formulation of the kernel function but also provided its practical implementation in the LIBSVM library. Through this integration, we have enabled support for mixed data, reading parameters for different kernels, training and storing new models, as well as making predictions using mixed samples, among other functionalities. This incorporation of the Aggregation Kernel into the LIBSVM library represents a significant advancement in the application of SVMs for the manipulation and analysis of mixed data, providing users with an effective and comprehensive tool to tackle these types of problems.

As for the **secondary objective**, we have successfully formulated an effective new SVM algorithm called Bagging-SVM based on Bagging techniques (Section 4.2). This formulation aims to address the challenges arising from the treatment of large volumes of data in conventional SVM algorithms, with the goal of reducing the complexity of model generation from samples and maintaining a stable error rate.

On the one hand, we have successfully estimated the complexity of the Bagging-SVM algorithm and the results indicate a significant reduction in training time compared to traditional SVM algorithms (Section 4.2). This ensures the efficiency and scalability of the algorithm when dealing with large volumes of data, which is a critical aspect when tackling real-world problems.

On the other hand, we have successfully implemented the Bagging-SVM algorithm in the LIBSVM library. This functional implementation has allowed us to create and store multiple models in a single training stage, make predictions using multiple models and modify the main SVM algorithm to adapt to the Bagging technique. This has provided us with a powerful and comprehensive tool to address large-scale problems with these algorithms, resulting in a significant improvement in efficiency and the effectiveness of the obtained results.

Requirements review

In this section, we will analyze and evaluate the compliance of the different requirements, both functional and non-functional, that we have established at the beginning of the project (see Section 1.2 and 1.2). We will start by addressing the **functional requirements**.

First and foremost, we have demonstrated the **validity** of the Aggregation Kernel by applying the closure properties of kernels (Section 3.2). Furthermore, we have **validated** the Bagging-SVM algorithm as a viable method within the SVM framework by providing a theoretical formulation and studying its complexity (Section 4.2). During the experimentation phase, we have confirmed that this algorithm is highly efficient in handling large volumes of data, thus satisfying the efficiency requirements.

Both the Aggregation Kernel and the Bagging-SVM algorithm have been **successfully implemented** in a new version of the LIBSVM library called LIBSVM^{aggr}. This facilitates the processing and analysis of mixed data. Moreover, the implementations have proven to be effective in fitting Support Vector Machine models, validating their functionality.

Regarding the **predictive capabilities**, we have validated the proposed hypotheses and demonstrated that the Aggregation Kernel is effective in measuring similarity between samples of mixed data. On the other hand, the Bagging-SVM algorithm has allowed us to reduce complexity in the training stages and maintain a stable error rate in predictions.

All the methodologies implemented in LIBSVM^{aggr} continue to provide the same **metrics** offered by the original LIBSVM library to evaluate the performance of the generated models. In classification problems, we obtain the percentage of accuracy, while in regression problems, we obtain the Mean Squared Error (MSE) and the Squared Correlation Coefficient (R^2). Upon completing the prediction stage, we also have the option to generate a file with the predicted values for each sample.

Regarding the **non-functional requirements**, we consider that the **efficiency** of the proposed software, which includes the functional implementation of the Aggregation Kernel and the Bagging-SVM algorithm, is acceptable. Although LIBSVM^{aggr} introduces a certain overhead in execution time due to issues associated with handling mixed data, we believe that the benefits and new functionalities it provides fully justify this small additional cost.

The **usability** of the proposed solution is excellent. We have maintained the simplicity in parameterization and execution that LIBSVM provided (*i.e.* through the command line). Additionally, we have implemented the configuration file feature, which allows us to easily, intuitively and quickly parameterize the subkernels used in model generation and the application of the Aggregation Kernel.

Finally, the proposed solutions, both in theory and practice, have been designed with **scalability** and **manipulability** in mind. We have carefully defined the theoretical proposals for both the Aggregation Kernel and the Bagging-SVM algorithm to allow for possible modifications, variations and updates proposed by the data science and machine learning community.

Likewise, during the process of implementing these features in LIBSVM^{aggr}, we have taken into consideration the **simplicity** and **ease of code** extension. Some examples include efforts to ensure that supporting new data types and implementing new kernel functions do not become a complex task. This approach allows users to develop their own methodologies in a simple and intuitive manner.

In conclusion, we believe that we have fulfilled the objectives and requirements, both functional and non-functional, that we set at the beginning of the project. Thus, we have been able to provide a set of effective, efficient and functional solutions to the main challenges we have stated.

7.2 Analysis and interpretation of the experiments

Once the objectives and requirements of the project have been analyzed, we will review the results obtained in the experimentation phase and state the conclusions drawn.

In the **first experiment**, we verified that our version of LIBSVM, LIBSVM^{aggr}, is capable of reproducing the same results as the original version, as long as we parameterize the algorithms in the same way. This allows us to conclude that the basic implementation of LIBSVM^{aggr} is correct and, therefore, valid for future explorations and experiments. However, we observed a slight increase in execution time compared to the original version. Therefore, we decided to define a new experiment.

In the **second experiment**, we aimed to determine the exact overhead introduced by LIBSVM^{aggr} in execution time. We assumed that our version requires more execution time since it needs to access data structures more frequently in order to be compatible with mixed data problems. Our hypotheses were confirmed as we found that, on average, LIBSVM^{aggr} requires 31.61% more time during training and 67% more time during prediction compared to the original LIBSVM.

Regarding the obtained percentages, it is important to note that the number of data used during the training phase was much higher than during the prediction stage. Therefore, the overall weighted average of the execution time overhead remains at an acceptable level. Additionally, we identified the code points responsible for this time overhead in prediction, with the perspective of reducing this expense in future updates.

The **third experiment** focused on testing the functionality of the Bagging-SVM algorithm implementation (see Section 4.2) in the LIBSVM^{aggr} library. We expected the training time to be reduced by approximately m units (as estimated in Section 4.2), where m is the number of submodels used in the algorithm. Our hypotheses were successfully verified and we demonstrated that the proposed Bagging-SVM algorithm is much more efficient than the conventional SVM algorithm in training models with large volumes of data. Specifically, Bagging-SVM was able to reduce the training time by half using a total of $m = 2$ models and practically reduce it to one-tenth using $m = 10$ models.

In addition to the aforementioned results, we found that the error rate does not significantly increase when using this Bagging-based approach. Considering the results obtained, we consider Bagging-SVM as a promising alternative for addressing problems with large volumes of data using SVMs and it can have very interesting applications in the current field of data science.

Finally, in the **fourth experiment**, we aimed to test the effectiveness of the Aggregation kernel in a real-world problem. By using the Aggregation kernel, we were able to improve prediction accuracy by a range of 5 to 10% compared to the traditional approach that transforms mixed data into real-number vectors and uses a single RBF kernel.

This leads us to conclude that the inclusion of the Aggregation kernel is a functional tool for addressing real-world problems with mixed data. Furthermore, we believe that this kernel has potential applications in various areas, such as education or data analysis, among others.

7.3 Overall analysis and conclusions

At this point, we have been able to verify the fulfillment of the objectives and requirements we established at the beginning of the project. Next, we have reviewed the results obtained in the experimentation phase and stated our conclusions. In this section, we will recap the achieved milestones and present our final conclusions.

We initiated this project by studying and analyzing the formulation of Support Vector Machines (SVM) and the role that kernel functions play in these algorithms. This knowledge was instrumental in approaching the design and implementation stages of the techniques we will outline below.

Subsequently, we conducted a state-of-the-art review regarding the types of variables encountered in modern problems and the kernel functions found in current literature for each of these variable types. Specifically, we observed that some of the most common variables in contemporary problems are continuous, binary, categorical, circular and textual. For all the aforementioned variable types, we have successfully identified suitable kernel functions, with the exception of circular variables, for which we have introduced a novel kernel function (refer to Section 3.1).

In the realm of the state-of-the-art review, we examined proposals for Support Vector Machine algorithms found in current literature that are aimed at improving the efficiency of these algorithms when dealing with large volumes of data.

Based on the preceding research stage, we observed that current literature lacks a kernel function fully focused on handling mixed data and there is room for improvement in the efficient implementation of SVM algorithms for addressing problems with significant amounts of data.

Subsequently, we formulated and demonstrated the validity of a new kernel function called the Aggregation kernel, which allows for the treatment of mixed data. Additionally, we provided the formulation of a variant of the Support Vector Machine algorithm based on Bagging techniques, which improves the efficiency of these algorithms in handling large volumes of data without increasing the prediction error. We named it Bagging-SVM.

Once the Aggregation kernel and Bagging-SVM were outlined, we provided a new version of the renowned Support Vector Machine library LIBSVM [30], which we named LIBSVM^{aggr}. This version effectively implements both the Aggregation kernel and the Bagging-SVM algorithm.

Furthermore, we included additional functionalities such as support for and treatment of mixed data, the implementation of kernel functions for the aforementioned data types, the possibility to parameterize the kernels through a new configuration file and the implementation of techniques for parameter and hyperparameter estimation for SVM and kernels, among others.

The design and implementation of this new version of LIBSVM have adhered to strict principles of efficiency, effectiveness, usability, scalability and manipulability. We have confirmed the efficiency, effectiveness and scalability of the proposed techniques through four experiments that have yielded promising results. Regarding the usability and manipulability of LIBSVM^{aggr}, we believe that we have designed the code in such a way that both experts and novice users can employ and expand the library in a simple and intuitive manner.

In general terms, we consider that we have designed, studied and provided different techniques and tools in the field of data science, machine learning and, specifically, Support Vector Machines that fill an important gap in the use and study of these algorithms. We firmly believe that our proposals have a promising future with multiple applications ranging from research to pedagogy and best practices.

7.4 Future work

Once the final conclusions of the project have been stated, we believe it is necessary to analyze future work in order to continue improving and developing our proposals. In this section, we will describe the factors that we consider key for the project's continuity.

One aspect we aim to improve is the **efficiency** of the Aggregation kernel during the prediction stage. We believe that restructuring the function calls to the kernel functions can significantly reduce execution time. Due to time limitations, we have not been able to include this modification in the current framework of the project. Nevertheless, we consider it a promising direction for future investigations. This efficiency improvement can contribute to better applicability of LIBSVM^{aggr} in real-world problems.

Another important line of work is to implement **support for new data types** in LIBSVM^{aggr}. This expansion will allow addressing a wider range of problems and enhance the library's flexibility to adapt to different data science scenarios.

It is crucial to test and **evaluate the performance** and functionality of the Aggregation kernel in diverse real-world problems. Despite conducting an extensive experimentation process, we believe it is necessary to put the kernel to the test in real situations with different application domains. This will enable us to gain a deeper understanding of the behavior, strengths and limitations of our proposal.

To contextualize our work within a broader framework of machine learning, it is essential to make **comparisons with other relevant algorithms**. This will enable us to evaluate the relative behavior of the Aggregation kernel and Bagging-SVM compared to other existing techniques. Through this comparison, we can identify the strengths and weaknesses of our proposal and provide a more comprehensive understanding of its effectiveness in different contexts.

Lastly, we want to continue **improving and updating** the new version of the library with the aim of providing the data science community with a powerful, robust and effective tool. This entails ongoing code development, algorithm fine-tuning and feature expansion. It is also important to maintain open communication with the user community and be attentive to their needs and contributions in order to constantly improve the library and make it more useful for data science professionals.

7.5 Personal thoughts and assessments

Once the conclusions have been finalized, I cannot help but share some brief personal reflections and thoughts. From the moment we began planning the project proposal with Lluís Belanche, my supervisor, I anticipated that it would be an enriching experience.

The completion of this work has not been an easy path. It has required a significant number of hours dedicated to research and studying the current scientific literature in the field of data science and machine learning. We have faced challenges and difficulties along the way. The task of extending the LIBSVM library, in particular, has been far from trivial. We have had to design new and innovative techniques, overcoming obstacles that, given the project's time constraints, could have become critical points.

I consider the solutions we have developed to be innovative and the results obtained during the experimentation phase are promising. I am especially moved by the opportunity to share with the scientific community the tools we have had the opportunity to create and develop in this project.

Despite the adversities and challenges we have faced, the development of this project has been an incredibly motivating, stimulating and challenging experience. I have felt deeply satisfied with the opportunity to work on this project and contribute to the field of data science and machine learning with this research.

Bibliography

[1] S. Lohr, "The age of big data," *The New York Times*, Feb. 11, 2012. [Online]. Available: `https://www.nytimes.com/2012/02/12/sunday-review/big-datas-impact-in-the-world.html` (visited on 06/01/2023).

[2] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet]*, vol. 9, pp. 381–386, 2020.

[3] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory*, 1992, pp. 144–152.

[4] A. Daemen, O. Gevaert, and B. De Moor, "Integration of clinical and microarray data with kernel methods," in *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, IEEE, 2007, pp. 5411–5415.

[5] A. Daemen and B. De Moor, "Development of a kernel function for clinical data," in *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, IEEE, 2009, pp. 5913–5917.

[6] A. Daemen, D. Timmerman, T. Van den Bosch, C. Bottomley, E. Kirk, C. Van Holsbeke, L. Valentin, T. Bourne, and B. De Moor, "Improved modeling of clinical data with kernel methods," *Artificial intelligence in medicine*, vol. 54, no. 2, pp. 103–114, 2012.

[7] J. Cervantes, X. Li, W. Yu, and K. Li, "Support vector machine classification for large data sets via minimum enclosing ball clustering," *Neurocomputing*, vol. 71, no. 4-6, pp. 611–619, 2008.

[8] M. Awad, L. Khan, F. Bastani, and I.-L. Yen, "An effective support vector machines (svms) performance using hierarchical clustering," in *16th IEEE international conference on tools with artificial intelligence*, IEEE, 2004, pp. 663–667.

[9] E. Osuna and F. Girosi, "Reducing the run-time complexity of support vector machines," in *International Conference on Pattern Recognition (submitted)*, Citeseer, 1998.

[10] Y. Zhan and D. Shen, "Increasing the efficiency of support vector machine by simplifying the shape of separation hypersurface," in *International Conference on Computational and Information Science*, Springer, 2004, pp. 732–738.

[11] B. Scholkopf, C. J. Burges, A. J. Smola, *et al.*, *Advances in kernel methods: support vector learning*. MIT press, 1999.

[12] A. M. Turing, *Computing machinery and intelligence*. Springer, 2009.

[13] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[14] D. Crevier, *AI: the tumultuous history of the search for artificial intelligence*. Basic Books, Inc., 1993.

[15] A. Aggarwal, *Resurgence of artificial intelligence during 1983-2010*, 2018.

[16] A. J. Smola and B. Schölkopf, *Learning with kernels*. Citeseer, 1998, vol. 4.

[17] J. Shawe-Taylor, N. Cristianini, *et al.*, *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[18] J. D. McCaffrey, *Using dot product as a measure of similarity*, Mar. 2022. [Online]. Available: `https://t.ly/cj8NC`.

[19] P. Jaccard, "The distribution of the flora in the alpine zone. 1," *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[20] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 0123814790.

[21] R. D. Banker and R. C. Morey, "The use of categorical variables in data envelopment analysis," *Management science*, vol. 32, no. 12, pp. 1613–1627, 1986.

[22] F. Blog, *Nominal vs ordinal data: 13 key differences and similarities*, Oct. 2022. [Online]. Available: `https://t.ly/kWKc`.

[23] P. Rodríguez, M. A. Bautista, J. Gonzalez, and S. Escalera, "Beyond one-hot encoding: Lower dimensional target embedding," *Image and Vision Computing*, vol. 75, pp. 21–31, 2018.

[24] L. A. Belanche and M. A. Villegas, "Kernel functions for categorical variables with application to problems in the life sciences," in *Artificial Intelligence Research and Development*, IOS Press, 2013, pp. 171–180.

[25] H. Singh, V. Hnizdo, and E. Demchuk, "Probabilistic model for two dependent circular variables," *Biometrika*, vol. 89, no. 3, pp. 719–723, 2002.

[26] A. Martins, "String kernels and similarity measures for information retrieval," in *Technical Report*, 2006.

[27] S. Sonnenburg, G. Rätsch, and K. Rieck, "Large scale learning with string kernels," *Large Scale Kernel Machines*, pp. 73–103, 2007.

[28] L. Bottou, J. Weston, and G. Bakir, "Breaking svm complexity with cross-training," *Advances in neural information processing systems*, vol. 17, 2004.

[29] H. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," *Advances in neural information processing systems*, vol. 17, 2004.

[30] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, 27:1–27:27, 3 2011, Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[31] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, pp. 123–140, 1996.

[32] I. Corporation, *What is bagging?* 2023. [Online]. Available: `https://www.ibm.com/topics/bagging`.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: `https://scikit-learn.org/stable/modules/svm.html#complexity`.

[34] A. Karatzoglou, K. Hornik, A. Smola, and A. Zeileis, "Kernlab-an s4 package for kernel methods in r," *Journal of statistical software*, vol. 11, no. 9, 2004.

[35] T. Joachims, "Making large-scale svm learning practical," Technical report, Tech. Rep., 1998.

[36] G. Kim, C.-H. Wu, and Y.-S. Jung, "A new ν-svm model for classification,"

[37] C.-C. Chang and C.-J. Lin, *Libsvm – a library for support vector machines (official website)*, 2023. [Online]. Available: `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[38] C. Igel, V. Heidrich-Meisner, and T. Glasmachers, "Shark," *Journal of Machine Learning Research*, vol. 9, pp. 993–996, 2008.

[39] S. Rüping, "Mysvm-manual," 2000. [Online]. Available: `http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/`.

[40] R. Collobert and S. Bengio, "Svmtorch: Support vector machines for large-scale regression problems," *Journal of machine learning research*, vol. 1, no. Feb, pp. 143–160, 2001.

[41] A. Arqué, "Disseny, implementació i estudi de funcions de kernel per a vectors de variables no reals," B.S. thesis, Universitat Politècnica de Catalunya, 2021.

[42] E. Angelucci and P. Matricardi, *Aviones de todo el mundo. La Segunda Guerra Mundial.* Madrid: Espasa Calpe, 1979.