UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

Draft

Analysis of the SVM-RFE algorithm for feature selection

Author:
Robert PLANAS

Director:
Luis A. BELANCHE

Bachelor Degree in Informatics Engineering
Specialization: Computing

June 15, 2021

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'informàtica de Barcelona

–

Bachelor Degree in Informatics Engineering

Analysis of the SVM-RFE algorithm for feature selection

by Robert PLANAS

English: In machine learning, feature selection algorithms such as SVM-RFE are used to find a subset of statistically relevant features. In this project, we propose multiple extensions of this algorithm, to try to improve its performance or computational cost. The extensions we've tried, include non-lineal kernels, internal sampling and dynamic step.

Spanish: En aprendizaje automatico, algoritmos como SVM-RFE son usados para encontrar subconjuntos de caracteristicas estadisticamente relevantes. En este proyecto proponemos multiples extensiones para este algoritmo, con el objectivo de encontrar mejoras en su acierto o su coste computacional. Las extensiones que hemos provado inlcuyen kernels no lineales, muesteo interno y paso dinamico.

Catalan: En apranentatge automatic, algoritmes com el SVM-RFE son utilitzats per trobar sunconjunts de carecteristiques estatisticament rellevants. En aquest projecte proponem multiples extensions per aquest algoritme amb l'objectiu de trobar millores en el seu encert o el seu cost computacional. Les extensions que hem probat inclouen kernels no lineals, mostreig intern i pas dinamic.

List of Symbols

n	Number of observations	
m	Number of dimensions	
X	Dataset	$\{\vec{x_1}, \vec{x_2}, \ldots, \vec{x_n}\}$
$\vec{x_i}$	Example, observation, ...	$\{x_1, x_2, \ldots, x_d\}$
Y	Dataset labels	$\{y_1, y_2, \ldots, y_n\}$
$\vec{w}$, $\mathbf{w}$	Weight vector	$\{w_1, w_2, \ldots, w_d\}$
b	Constant term, bias	E.g. $\vec{w} \cdot \vec{x} + b = 0$
$\vec{u} \cdot \vec{v}$	Dot/Scalar product (vector)	$u_1v_1 + u_2v_2 + \cdots + u_nv_n$
$\mathbf{u}^{\mathrm{T}}\mathbf{v}$	Dot/Scalar product (matrix)	$u_1v_1 + u_2v_2 + \cdots + u_nv_n$
α_i, γ_i	Lagrange multipliers	
ξ_i	Slack variables	
C	Regularization parameter	
$\langle \mathbf{u}, \mathbf{v} \rangle$	Inner product	E.g. $\vec{u} \cdot \vec{v}$
$\lvert x \rvert$	Absolute value	$\sqrt{x^2}$
$\lvert\lvert \vec{u} \rvert\rvert$	Euclidean length	$\sqrt{\vec{u} \cdot \vec{u}}$
$d(\vec{u}, \vec{v})$	Euclidean distance	$\lvert\lvert \vec{u} - \vec{v} \rvert\rvert$
$\phi(\vec{x})$	Feature map	$\phi : D \to H$
$k(\mathbf{x_i}, \mathbf{x_j})$	Kernel function	$\langle \phi(\mathbf{x_i}), \phi(\mathbf{x_j}) \rangle$
γ, σ	RBF Kernel Parameter	
$\mathbf{H}$	Hessian Matrix of dual SVM	$\mathbf{H_{i,j}} = y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$
$\vec{s}$	Surviving features	$[1, 2, \ldots, d]$
t	Constant Step	

Contents

Chapter 1

Introduction

This bachelor thesis of the Computer Engineering Degree, specialization in Computing, has been done in the Facultat d'Informàtica de Barcelona of the Universitat Politècnica de Catalunya and directed by Luis Antonio Belanche Muñoz, doctorate in Computer Science.

1.1 Context

In statistics, machine-learning, data-mining, and other related disciplines, it is often the case that there is redundant or irrelevant data in a dataset[1]. Indeed, before we can start working with the data, some form of data analysis and cleaning is required. Data cleaning may include removing duplicated rows or rows with missing values, removing observations that are clearly out-layers, removing irrelevant variables (e.g. name, surname, email address), etc.

With the new era of Big Data, datasets have increased in size, both in number of observations and in dimensions. Applying classical data-mining and machine-learning algorithms to this high-dimensional data rises multiple issues collectively known as "the curse of dimensionality". One such issue is the elevated, usually intractable, cost and memory requirements derived from the non-linear (on number of dimensions) complexities of the algorithms. Another issue has to do with data in a high-dimensional space becoming sparse and negatively affecting the performance of algorithms designed to work in a low-dimensional space. And finally, a third issue is that with a high number of dimensions the algorithms tend to overfit, that is, they don't generalize enough and end up producing models that perform worse with real data that their predicted performance with the training data. (Li et al., 2017)

Simple manual data cleaning is not enough to achieve satisfactory amounts of dimensionality reduction. In this case we can use automatic techniques. We can classify such techniques in two categories, feature extraction, and feature selection.

1.1.1 Feature Extraction

Feature extraction techniques transform the original high-dimensional space into a new low-dimensional space by extracting or deriving information from the original features. The premise is to compress the data in order to pack the same information at the expense of model explainability[2]. Continuing with our data compression analogy, virtually all feature extraction techniques perform lossy compression, that

[1] A table with rows / records / observations and columns / variables / features / dimensions / predictors / attributes.

[2] The ability to explain why certain predictions are made. Also, interpretability.

is, some data is lost which makes the process irreversible. Notice that, if the process was reversible then feature extraction would not decrement explainability.

Some well known feature extraction algorithms include Principal Component Analysis (PCA) and auto-encoders, the first being a linear transformation over the feature-space and the second a neuronal network. PCA may be extended with a kernel method in order to make non-linear transformations. Similarly, we will also use non-linear kernels to extend SVM-RFE.

1.1.2 Feature Selection

In contrast, feature selection only selects a subset of the existing features, ideally the most relevant or useful. This may imply a greater loss of information compared to feature extraction, but it doesn't reduce explainability. Some problems require feature selection explicitly. In domains such as genetic analysis and text mining, feature selection is not necessarily used to build predictors. For example in micro-array analysis feature selection is used to identify genes (i.e. features) that discriminate between healthy and disease patients.

Feature selection methods may be classified by how they are constructed in three categories:

- **Filter:** A *feature ranking criteria* is used to sort the features in order of relevance, then select the k-most relevant.
- **Wrapper:** They use a learning machine (treated as a black box) to train and test the dataset with different subsets of variables. They rank the subsets based on the performance (score) of the model. A wide range of learning machines and search strategies can be used. Within the greedy strategies we find *forward selection* and *backward elimination*.
- **Embedded:** Like wrapper methods but more efficient. They use information from the trained model itself to make feature selection. Because they don't use the score, they can also skip testing the model.

In both wrapper and embedded methods greedy strategies can be used. **SVM-RFE** is a feature selection algorithm, of the embedded class, that uses Support Vector Machines (SVM) and a greedy strategy called Recursive Feature Elimination (RFE). This algorithm is an instance of backward elimination. It starts with a set of all features and eliminates the less relevant each iteration. Within each iteration SVM-RFE behaves like a filter method and uses a feature ranking criteria to decide which feature to eliminate. SVM-RFE takes advantage of the fact that, for linear SVM, the ranking criteria is to take the variable with the smallest weight in each iteration, where the wights are the coefficients of the hyperplane resulting from the SVM training. (Guyon and Elisseeff, 2003)

1.2 State of the art

The SVM-RFE algorithm was first proposed in a paper on the topic of cancer classification (Guyon et al., 2002). This paper uses the SVM-RFE algorithm to identify highly discriminant genes, encoded as features, that have plausible relevance to cancer diagnosis. Since then, SVM-RFE has remained a popular technique for gene selection and the original paper cited more than four thousand times.

The paper already proposes some natural extensions to the algorithm, such as eliminating multiple features each iteration based on the ranking and a *step* constant. Further research has been on improving and extending different parts of the algorithm. These include the use of a Gaussian kernel (Xue et al., 2018), using multiple SVM in the same iteration (Wang et al., 2011) or simply trying to find a better ranking criterion (Mundra and Rajapakse, 2007).

1.3 Objective

The main objective of this project is to research extensions of the SVM-RFE algorithm and try to optimize it. Optimizations may be in the form of improved performance or a reduction in time utilization. We've classified the possible extensions as follows.

Main extensions

These focus on improving the performance of the selection.

- **Non-linear kernel:** Use a more general ranking criterion and apply it to handle SVM with arbitrary kernels.
- **Multi-class criteria:** Find a ranking criterion that can handle multiple weight vectors in a useful way.

Secondary extensions

Focus on reducing the computational cost.

- **Internal sampling:** Use a different subset of the observations on each iteration.
- **Dynamic step:** Instead of using a constant value in each iteration, calculate it dynamically.
- **Stop condition:** Determine the amount of features that are relevant (SVM-RFE only provides a ranking) in an effective and inexpensive manner.

Combination of various extensions

- **Combo:** Mix *internal sampling*, *dynamic step* and *non-linear Kernels* into a single implementation.

1.3.1 Objective break down

To accomplish this objective, the project has been subdivided in two parts, each having specific tasks for each extension:

Theoretical Part

- Do research in SVM-RFE and in the extensions that will be tackled in the project.
- For each extension:
  - Design the algorithm and write its formalization in pseudocode.

  - Define the expected advantages or disadvantages of this extension over the base SVM-RFE.
  - Compute the time complexity.

Practical Part

- Program the base SVM-RFE algorithm and the extensions.
- For each extension:
  - Analyze its behavior for artificial data sets.
  - Analyze its behavior for real-world data sets.
- Compare the results obtained with the ones expected.
- Combine multiple extensions to further improve the algorithm.
- Draw conclusions about all the results obtained in the project.

1.3.2 Stakeholders

This project is intended to be of use for many involved parties. The most directly involved group, is the tutor and the researcher. Luis Antonio Belanche Muñoz is the tutor of this project. Robert Planas Jimenez would be the researcher. Feature selection algorithms is one of the areas of research of the tutor, and he has wanted to explore extensions to the SVM-RFE algorithm. He will lead and guide the researcher for the correct development of the project. The researcher is responsible for planning, developing and documenting the project, as well as experimenting, analyzing and drawing conclusions.

The other group of interested parties would be stakeholders that do not interact with the project directly but still benefit from it. In the first place we have researchers on the fields of bioinformatics and data mining, that use machine learning methods (specifically, SVM-RFE) for micro-array analysis, text analysis, or other of its popular applications. Indirectly, companies that make use of any findings will also benefit. Finally, the general population may also benefit from better diagnostics and more effective drugs.

1.3.3 Potential obstacles and risks

Some obstacles and risks identified that could potentially prevent the correct execution of the project are:

- **Deadline of the project:** There is a deadline for the delivery of the project. This being a research project however, is considerably hard to estimate how much time tasks will take, or even decide whether a task has been finished or not.
- **Bugs on some libraries:** This is considered of low risk, but is still a possibility that errors on the software package used extend to code, making it work incorrectly.
- **Insufficient computational power:** Machine learning algorithms, in general, can be very resource intensive. It could be the case that our hardware can not handle some datasets.

- **Hardware related issues:** A hard drive failure could occur that would end in lost data, or a failure in a router could disconnect us from the internet.
- **Health related issues:** In addition to health issues that can occur at any time without prior notice, we're in the middle of a pandemic.

1.4 Methodology

1.4.1 Framework

The methodology that I will use for the project is a combination of waterfall and Kanban methodologies. Waterfall will be used to define the general phases of the project, and Kanban for tracking the individual tasks. In waterfall tasks can not start until the previous task has been completed, and thus following strict deadlines is important. Phases will be managed like this, one phase will not start until the previous phase ends. Each phase will be composed of multiple tasks, which will then be managed by the Kanban methodology.

Kanban is much more flexible than waterfall. Its principal objective is to manage tasks in a general way, by assigning different statuses to them. Kanban stands out by its simplicity, and will continue to endorse that simplicity by managing the visual representation of the cards in a simple, plain, text formatting. Each card will be in a row, with the first column defining its name and the other its status. The statuses we've considered are:

- **To do:** A basic idea of the task is present.
- **Definition:** The task is in the theoretical part.
- **Implementation:** The task is in the practical part.
- **Completed:** The task is finished.

An uppercase letter `"X"` will mark the current state of any given card. If more granular information is required, other marks may be used instead. For example, to indicate that the task is paused a `"P"` would be used. To indicate the progress within some stage a percentage would be used. For easy monitoring, this table will be kept in the `readme.md` file of our `GitHub` repository.

1.4.2 Validation

We will use a GitHub repository as a tool for version control, which will allow us to share code easily and recover from data lose. The repository will contain both the code for the experiments, each in one subfolder, and code for the documentation. In order to verify the implemented code, it will be tested with multiple data sets. In the practical part, hyper-parameters will be selected using some model validation technique, such as the cross-validation. Each experiment will be done at least 6 times and the average of the results will be the final result.

Face-to-face meetings with the tutor of the project will be scheduled once every two weeks. In these meetings the current project status will be discussed, and the tasks to do during the following two weeks will be defined. In case of unexpected problems, extraordinary meetings can be arranged.

Chapter 2

Project Planning

This thesis is worth 15 ECTS credits, each of which with an estimate cost of 25 to 30 hours. Therefore, the total time allocated for this project, as indicated by the faculty, is of 375 to 450 hours. This time is to be distributed in 100 days, from 03/08/21 to 06/15/21, with an estimated work of 4 to 5 hours a day. The date of the oral defense is planned for the first week of July, this sets the deadline to be on the 06/18/21.

An extra 3 ECTS credits are to be used for project managment, this is roughly 80 hours, which makes the total time estimate for the whole project (Theisis + Project Managment) to be at best 450 hours and at worse 540 hours. In order to make a proper plannification, we have defined the estimated cost to be at 500 hours.

2.1 Task definition

In this section it is presented all the tasks that will be carried out along the project. For each, a description, duration and a list of dependencies with other tasks are given.

Project management is a mandatory group of such tasks, albeit not very useful, considering that: One, this project is done by a single individual, with assistance from a project director; And two, this is a research project, which makes planning of specific tasks difficult, since it is the results from the research that drive the next steps to be done.

- **Context and scope:** We have to indicate the general objective(s) of the project, contextualize it and justify the reason for selecting this subject area.
- **Project planning:** This will help us not lose focus while we're working on the project.
- **Budget and sustainability:** For this specific project, this is irrelevant. The budget required is negligible and already defrayed; and the impact, beyond trivial matters, is likely zero and otherwise unknown.
- **Final project definition:** Review the work done in the project management tasks.
- **Meetings:** Online meetings are scheduled once every two weeks with the tutor of the project. Discussion of the status and next tasks to do will be done.

This project is research focused. Therefore, before starting the practical tasks research on the various topics needs to be done. This will involve collecting and analyzing previous studies that tried new methods and extensions to the SVM-RFE algorithm. We will also have to document ourselves in the SVM and feature selection

areas, as well as the algorithms and the statistical theory used in the studies. Some basic understanding of bioinformatics may also be required considering the use case of most if these studies.

- **Research** previous work on the literature on extensions of the algorithm and create a short **report** with the findings.
- Write the algorithm formalization in pseudocode.
- Define the expected advantages or disadvantages of this method over the base SVM-RFE.
- Compute the time complexity.

Once the initial research is done the algorithm must be codified and tested. This group is composed of the following tasks:

- **Program the base SVM-RFE algorithm.** For this we will use a library for the SVM. For the RFE part, since we need to be able to extend the algorithm, we will program it from scratch.
- **Program the extensions of the SVM-RFE algorithm** based on the research done and the pseudocode.
- **Test the new extensions with artificial data.** This requires creating models and testing their performance with artificially generated data sets.
- **Obtain data sets with real data.** We will use publically available data-sets, such a *Madelon* or *Digits*. We can then compare our results with those of other studies.
- **Test the new extensions with real data.** This requires creating models and testing their performance with real data sets.
- **Analyze the results** obtained in the experiments and draw conclusions. A report will be made for reference during the final documentation phase.

Related to testing and analyzing the results, if we want to draw fair conclusions, comparing our results with the state of the art is important. However, the start of the art is bast and fluid, and making a fair comparison with all available research papers is a daunting task. Instead, we will use a common ground, in this case the NIPS 2003 feature selection challenge. The results of this challenge, as well as a general analysis of the algorithms used, will serve as a reference point. Thankfully an analysis of these results has already been done (Guyon et al., 2004). This simplifies our problem, now we only need to compare our algorithms performance with those already presented in the challenge.

Once finished, the final documenting phase will begin. Firstly, we will collect all the information obtained in the experimental and analysis part, which will be available in the form of the reports that we've done along the tasks. Afterwards, we can start writing the documentation of the project. This will include a fairly extensive review on concepts related to SVM, statistics, feature selection and RFE. Finally, we will have to prepare for the oral defense of the project.

ID	Description	Hours	Dependencies
T1	Project Managment	80	
T1.1	Context and Scope	20	T2.1
T1.2	Project planning	10	
T1.3	Budget and sustainability	10	T1.2
T1.4	Final project definition	20	T1.1, T1.2, T1.3
T1.5	Meetings	20	
T2	Theoretical Part	160	
T2.1	Research	90	
T2.2	Formalize	20	T2.1
T2.3	Analyze	50	T2.2
T3	Practical Part	160	
T3.1	Program the base SVM-RFE algorithm	10	T2.1
T3.2	Program the extensions	50	T2.1, T3.1
T3.3	Test with artificial data	20	T3.2
T3.4	Test with real data	30	T3.2
T3.5	Analyze the results	50	T3.3, T3.4
T4	Documentation	100	
T4.1	Writte the documentation	80	T2, T3
T4.2	Prepare the thesis defense	20	T4.1

TABLE 2.1: Summary and time estimates of the tasks.

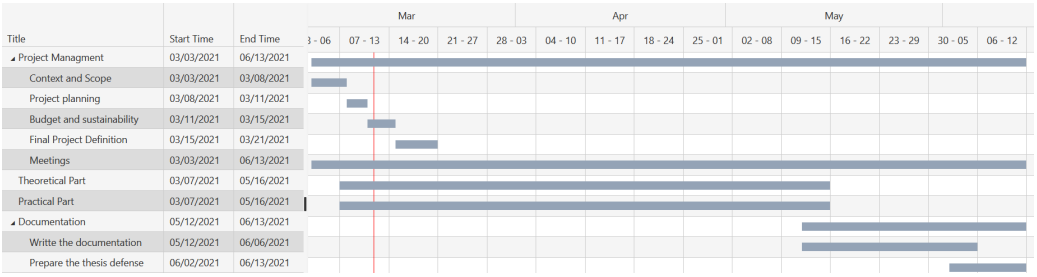


FIGURE 2.1: A summary of the tasks represented with a gantt chart. Notice that all the theoretical and practical tasks are done in parallel.

2.2 Resources

Our project needs resources to carry out its correct development. These resources have been divided in 4 different groups: human, hardware, software and material resources.

2.2.1 Human resources

There are three human resources that are directly involved in this project.

- **The researcher:** He is responsible for the development of the project, that is, he will have to plan, analyze, program, experiment and document the project.
- **The director/tutor:** He is responsible for leading and guiding the researcher for the correct development of the project.

- **The GEP tutor:** He is in charge of reviewing the project management tasks done in the initial stage of the project.

2.2.2 Hardware Resources

The most essential resource needed is a computer connected to the internet. In this project a personal computer will be used. Its specializations are 16 GB of RAM and a CPU *AMD Ryzen 7 4800HS*, with a base speed of 2.9 GHz and 8 cores. Hardware required for a connection to the internet (a router, an access point, etc) is also taken into account.

2.2.3 Software Resources

For project management tasks `Google Calendar` will be used. A number of other Google products such as `Google Mail`, `Google Drive` or `Google Meet` will also be used as tools required for communication with the director or storing of information.

The documentation will be written in LaTeX, a document preparation system often used in academia. It has the advantage to integrate well with version control systems. A LaTeX template named "`Masters/Doctoral Thesis`" will be used to facilitate the typesetting and styling of the document. This template was made by Vel and Johannes Böttcher and licensed under `LPPL 1.3`, based on previous work from Steve R. Gun and Sunil Patel and used with minor modifications. The original is available at `http://www.latextemplates.com/`. The document will be written with the `Microsft Visual Studio Code` editor, using the `LaTeX Workshop` extension, and it will be compiled with the `MiKTex` distribuition installed on a `Linux` machine running virtualized within a `WSL` container on top of the actual operating system, a `Microsoft Windows 10`. For browsing the internet the latest available version of the `Mozilla Firefox` web browser will be used, and references to papers will be kept with the `Zotero` reference manager.

For the practical part, the programming language of choice will be `Python3`. This is currently one of the programming languages with better popularity in machine learning and related applications. Various libraries and software packages will be used for different tasks. For parsing datasets the `pandas` library will be used. For data visualization the `mathplotlib` library will be used. Also, a general tool-set designed for machine learning, the library `sklearn`, will be used. Finally, code will be documented in-place during its development with the `Jupyter Notebook` software.

Other libraries and software packages could also be used if the need arises.

2.2.4 Material Resources

In a research focused project such as this, access to scientific journals, books and similar research material is needed. Some articles related to this research are freely available on the internet, but some require a subscription or single time payment. Fortunately, the UPC, the university this project is being developed at, has a subscription agreement with most of these journals and provides free access to their memebers, including the students.

2.3 Risk Management

The potential risks and obstacles have already been introduced in section 1.3.2. In this section we will focus on a contingency plan to mitigate the risks.

- **Deadline of the project:** The flexibility of the Kanban methodology should help us modify our schedule and working hours if required. If it becomes apparent that the deadline will not be meet, that is, more than 50 hours a week of work are required to finish the project in time, an extension of the deadline can be requested.
- **Bugs on some libraries:** Alternative libraries can be used. If no alternative is found, because most of the used libraries are open source, a *bugfix* could be implemented.
- **Insufficient computational power:** This can be mitigated by using a small sample of the data-set. Working with fewer data, although faster, can induce a small performance reduction and make the results not comparable with each other. Therefore, this will solution is not ideal.
- **Hardware related issues:** To avoid data loss, all code and documentation will be routinely uploaded to the cloud in a `GitHub` repository and `Google Drive` account.
- **Health related issues:** Not much can be done if a health related problem occurs, but various preventive actions, such as a good diet, exercise and resting habits can be promoted.

Chapter 3

Budget and Sustainability

In this section a budget estimation will be made. It will include personnel costs per task, generic costs and other costs. Moreover, some questions regarding the sustainability aspect of the project will be answered.

3.1 Budget

3.1.1 Costs by role and activity

In this section we will add the personnel costs to the tasks defined in section 2.1. For each task a cost will be calculated based on the hourly pay wage per role and the time spend each. Four roles have been defined with their corresponding hourly wage, these are:

- The **project manager** (T1) who is responsible for leading the project direction, planning and correct development.
- The **researcher** (T2), who must perform research in the topic and related ideas, experiment, analyze the results and draw conclusions from them.
- The **programmer** (T3), who must set up the developer environment, code the algorithms in the specified programming language and test them for correctness.
- The **technical writer** (T4), who is responsible for writing this project documentation. This includes the reports for each task that requires it and the final thesis.

These roles have a clear mapping to the task groups defined in table 2.1. All roles will be played by the researcher (see section 2.2.1) except the project manager role, which will be played by the researcher, the director and the GEP tutor. For a detailed description of each task cost see table 3.1.

Notice that, although we aligned the project roles with the task groups so that we could simplify our calculations, it is not required to do so. A more complicated scheme where multiple tasks are assigned to a given role is also possible. In such cases we would also have to think about weather the work distribution is uniform or not, and assign percentages if it isn't.

Role	Year (€)	Year +SS (€)	Hour +SS (€)	Task	Task Cost (€)
Project Manager	39 000	50 700	28.7	T1	2 296
Researcher	32 000	41 600	23.5	T2	3 760
Programmer	26 000	33 800	19.1	T3	3 065
Technical writer	22 000	28 600	16.2	T4	1 260
Total					**10 381**

TABLE 3.1: Annual estimated salary for the different project roles (*Salarios, ingresos, cohesión social* 2017). The amount of working hours in a year is defined to be 1764. *+SS indicates "Social Security included"*.

3.1.2 Generic costs

Amortization

In this section the amortization costs for the resources purchased in a single payment are calculated. Notice that since all the software used is free and open source, it doesn't contribute to the cost, thus it is not displayed here. In fact, the only resource that is valid for an amortization analysis is the computer specified in section 2.2.2.

The equation we use to compute the amortization cost for each resource is the following:

$$\text{Amortization (€)} = \text{Cost (€)} \times \frac{1}{4 \text{ years}} \times \frac{1}{100 \text{ days}} \times \frac{1}{5 \text{ hours}} \times \text{Hours Used} \quad (3.1)$$

If we apply the amortization equation to the computer, which we purchased for €999.95, and assuming 500 hours of usage, its estimated amortized cost is €249.98.

Electric cost

In this section we only calculate a rough estimate. Calculating accurately the cost of electricity involves many variables, and it's outside the scope of this thesis. The average cost of electricity in Spain, in therms of kWh, is €0.12. We only count the cost when the hardware is turned on. The following table (3.2) shows the individual and total cost per item.

Item	Power (W)	Time used (h)	Consumption (kWh)	Cost (€)
Computer	180	500	90	10.8
Router	10	500	5	0.6
Total				**11.4**

TABLE 3.2: Electric cost estimate.

Internet cost

The internet cost in my current location is €29.00 per month, this is roughly about €0.95 per day (variance is introduced because a month length is not constant). The

amount of working hours per day is assumed to be 5, thus the total cost the whole project is:

$$€0.95\ /\text{day} \times 100\ \text{days} \times 5/24\ \text{hours} = €19.79$$

Work space

This project will be developed at my parents home at Figueres, with a rent of €400. Since the amount of people living there is 2, the actual cost is €200.

Total generic costs

Table 3.3 summarizes the total generic costs of this project.

Group	Cost (€)
Amortization	249.98
Electricity	11.4
Internet	19.79
Rent	200
Total	**481.17**

TABLE 3.3: Total generic cost estimate.

3.1.3 Other costs

Contingencies

Unexpected problems that were not foreseen may appear during the development of the project, which would take part of our budget. For this reason it as always a good idea to prepare a contingency budget calculated from the budgets we've calculated up to now. We will apply a 10% contingency margin, that is €108.62.

Incidental cost

Unexpected problems that were foreseen and can be mitigated are described in section 2.3. Some of these mitigations may incur some extra cost. To be able to handle that cost in this section we will calculate a budget based on the predicable problems, their chances of actually occurring, and the expected cost associated for solving them.

- **Deadline of the project:** If an extension of the deadline is requested, that would require extra hours expend by the researcher. Assuming the extra time required to be 50 hours that would give us a cost of €1,175.
- **Bugs on some libraries:** Implementing a *bugfix* we assume would cost around 20 hours, a task done by the programmer. The estimated cost is €382. The risk is small.
- **Insufficient computational power:** If using smaller datasets where not an option, we would abandon the project, since purchasing a new, more powerful, computer or renting a supercomputer is too expensive. The risk is very small.

- **Hardware related issues:** New hardware would be purchased, if possible only the faulty component would be replaced. This would have an estimated cost of about €100. The risk is small.

The following table summarized the expected cost due to foreseen incidents.

Risk	**Expected (€)**	**Risk (%)**	**Cost (€)**
Deadline of the project	1 175	30	352.5
Bugs on some libraries	382	10	38.2
Insufficient computational power		5	
Hardware related issues	100	10	10
Total			**400.7**

TABLE 3.4: Total incidental cost estimate.

3.1.4 Total cost

The total cost for the project is summarized in table 3.5. The cost has been computed as the sum of the justified costs explained in the other sections of the budget plan.

Section	**Cost (€)**
Costs by role and activity	10 381.00
Generic costs	481.17
Other costs	509.32
Total	**11 371.49**

TABLE 3.5: Total cost estimate.

3.2 Sustainability

This project does not have a major environmental impact, since it's a research project. Still, some impact, a very small amount, is expected from the electricity consumption and hardware used. This can hardly be reduced, as is required for the development of the project. If the project succeeds at finding improvements to the SVM-RFE algorithm, it will reduce computing power requirements for future researches that use it compared to the state-of-the-art solutions. Because this algorithm is used in medical research, a substantial improvement could create a chain reaction that benefits the health of the population in general. This, however, is a very optimistic expectation.

3.2.1 Environmental dimension

Have you estimated the environmental impact of undertaking the project?

This project will not have a major environmental impact. Still, some impact, a very small amount, is present in the form of electricity consumption and the method used by the provider to generate such electricity. Also, the hardware used will eventually contribute to the technological waste problem, not to mention the environmental cost for their production.

Have you considered how to minimize the impact, for example by reusing resources?

The impact is directly derived from the project requirements and can thus not be reduced easily. A low-cost computer doesn't necessarily imply a lower impact on the environment or lower electricity consumption. Recycling is of course always an option, but the decision to do so will be taken way after the project has been completed.

How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution environmentally improve existing solutions?

This project will use the same resources as those used in state-of-the-art alternatives. Therefore, this solution does not environmentally improve existing solutions. If the project succeeds at finding an improvement to the SVM-RFE algorithm however, it could imply a reduction on computational power required to solve some problems, and thus also a reduction in power consumption.

3.2.2 Economic dimension

Have you estimated the cost of undertaking the project (human and material resources)?

Yes, see section 3.1.

How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution economically improve existing solutions?

If improvements to the SVM-RFE are found in this project, this will make any research that makes use of it less expensive and produce better results.

3.2.3 Social dimension

What do you think undertaking the project has contributed to you personally?

This is the last step required to finish my degree in computer science. Finalizing this degree will allow me to enter the work force and become self-sufficient. Beyond that, it has introduced me to the fields of bioinformatics and data mining, as well as shown the importance of feature analysis and how it can be not just a cleaning prerequisite for a more important problem, but the main dish.

How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution socially improve (quality of life) existing. Is there a real need for the project?

Improvements to the SVM-RFE algorithm may or may not be found. Even if found it is still unknown if they will be substantial enough. Moreover, an analysis of the variants of the algorithm will be useful for those who want to use SVM-RFE in the most optimal way.

Chapter 4

Background

In this section we're going to review the concepts required to understand the SVM-RFE algorithm. Section 1.1 already introduced some concepts around the SVM-RFE algorithm, placed it in context, and enumerated some of its applications. In this section we'll focus on the inner workings of the algorithm and how these parts add together.

4.1 Machine learning

Machine learning is a subfield in the broader discipline that is artificial intelligence, with a particular take in statistics. These algorithms, also called learning machines or just machines, use data to learn patterns and make predictions. The data is collected in a separated unrelated process, and structured in the form of a *dataset* (a collection of data). Once the dataset is further cleaned and prepared, the learning machine finally consumes it in a process called *training*. After this process the machine produces a *model*, which is a function that can be used to make predictions on new data.

Two canonical problems in machine learning are regression and classification problems.

4.1.1 The dataset

A dataset is simply a collection of data. In the context of machine learning this dataset will be used to make predictions, take decisions, or find patterns. For a machine learning algorithm to be able to consume a dataset the first step is to represent it in tabular form.

Most datasets come already in tabular form. Some of the most notable exceptions are datasets involving images. In this case computer vision methods are often used to extract numeric data representing characteristics of the image. Sometimes a more direct transformation can also be made, for example making each feature be the intensity level of a single pixel in the image. This of course produces a high number of features, most of which are redundant or irrelevant (e.g. features representing pixels in the background).

In a dataset represented as a table, columns describe different *features*, *properties* or *attributes* of some group of objects and rows represent *instances* or *examples* of that group. For example, if objects were vehicles then features could include the brand, power, weight, maximum speed, and other such characteristics of various vehicles, each of which would be in a row. Different names are used in different contexts. One of the most typical naming conventions comes from the statistics domain which refers to columns as *variables* and rows as *observations*. Sometimes different nomenclature is used to differentiate features in different stages of the cleaning and preparation process, but in this thesis we use them all instinctively.

Src	Dst	NAT-Src	NAT-Dst	Action	Sent (B)	Rcvd. (B)	Packets	Elapsed (sec)
57222	53	54587	53	allow	94	83	2	30
56258	3389	56258	3389	allow	1600	3168	19	17
6881	50321	43265	50321	allow	118	120	2	1199
43537	2323	0	0	deny	60	0	1	0
50002	443	45848	443	allow	6778	18580	31	16

TABLE 4.1: Example dataset extracted from the Internet Firewall Data (Ertam, 2019). Only five observations and main variables shown.

4.1.2 Classification

For the classification problem we want to predict in witch group or *class* to assign some new observation. Table 4.1 provides an example of what could be a good classification problem. Imagine we are building a firewall and want it to predict if some new packet in the network should be allowed to pass. We could train a classification learning machine with the dataset represented in the table (the full version) and produce a model capable of doing such predictions. Although it is trivial to identify a pattern in the example, it may not be so for real world examples. Automatic *pattern recognition* is key in solving many real world problems, and it is one of the features of these algorithms.

Mathematically, the set of observations is defined as $X = \{\vec{x_1}, \vec{x_2}, \ldots, \vec{x_n}\}$ and the set of the corresponding labels or target classes is $y = \{y_1, y_2, \ldots, y_n\}$ with every element of this set being some class $y_i = C_k$ of a discrete set of classes with size K. They are called *domain set* and *label set* respectively. For our example, the classes would be $C = \{\text{allow}, \text{deny}\}$. Thus, we can describe the goal of a classification problem as assigning some class C_k to an input vector $\vec{x}$.

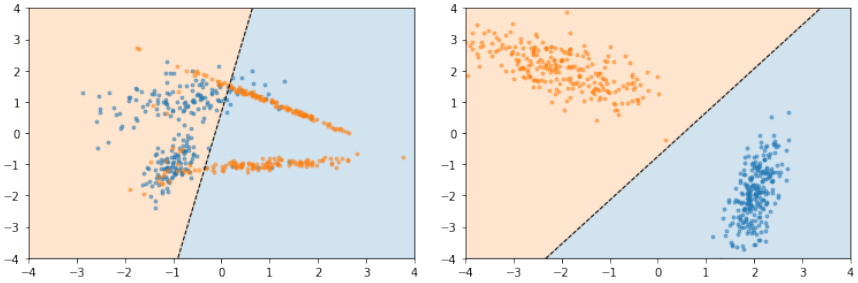
Although we've used strings to represent classes, part of the preparation process of the dataset involves turning all values into numerical scalars, so our classes would actually be some natural numbers. Also notice that every $\vec{x_i}$ is a vector containing a single numerical value for each feature excluding the label.

We make a distinction between two-class problems (or binary) and multi-class problems. Two-class problems typically use $C = \{0, 1\}$ as classes and thus can be modeled with a boolean function or, if a probability is desired, a function that returns values between 0 and 1. This simplifies the model substantially. In fact some learning machines, such as the SVM, can only work with two-class problems, and use different methods to extend to the multi-class version.

4.1.3 Visualization

A visualization of the problem in some euclidean space is useful to understand how most classification algorithms work, and in particular SVM. Typically, classification models divide the input vector space[1] into *decision regions*. The boundaries of such regions are called *decision boundaries* or *decision surfaces*. If the decision boundary is in the form of some hyperplane of dimension $(m - 1)$, with m being the dimension of the space, then we say that it's a linear model. A dataset whose classes can be completely separated by such linear decision boundary is said to be *linearly separable*.

[1] The euclidean space of minimum dimension containing all possible input vectors $\vec{x}$.



Not linearly separable Linearly separable

FIGURE 4.1: Decision regions and 1-D hyperplane boundary of some linear model for two datasets. Points are observations, colors indicate the class.

Given a boundary we can determine in which region some new point $\vec{x}$ falls using a *discriminant function*. One of the most trivial cases is when the boundary is lineal. It is convenient now to do a small refresh on the equation that describes a hyperplane.

Equation of a line (slope-intercept form)

$$y = mx + t \tag{4.1}$$

Where m is the *slope* or *gradient*, x is the independent variable of the function $y = f(x)$ and t is the y-intercept value, the point of the function where the line crosses the y-axis, i.e. $t = f(0)$. This description has the advantage that can be directly represented as a function $f(x) = mx + t$.

Equation of a line (standard form)

$$ax + by = c \tag{4.2}$$

This is an equivalent form. This one however is not representable by a function $f : \mathbb{R} \to \mathbb{R}$, instead it is often represented as a set $L = \{(x, y) \mid ax + by = c\}$. This allows representing vertical lines and also features the useful property that (a, b) is the normal vector[2] of the line.

Equation of a line (general form)

$$ax + by - c = 0 \tag{4.3}$$

A simple linear transformation of the standard form produces the general form (*Wikipedia - Line (geometry), 2021*). This conserves the normal vector and also has the advantage of being representable by an implicit function[3]. Notice that if represented in 3-D it would produce a plane that is perpendicular to the xy plane, since its normal would always have the form $(a, b, 0)$.

[2] A vector that is perpendicular to the surface.

[3] A function of multiple variables $f : \mathbb{R}^n \to \mathbb{R}$ such that we only consider solutions where $f(X) = 0$. For example a circle can be defined with an implicit function $f(x, y) = x^2 + y^2 - r$, but if we were to plot it in 3-D we would instead see an inverted cone.

Equation of a 3-D Plane

$$ax + by + cz + d = 0 \tag{4.4}$$

We can extend the general form of a line with one more dimension, it only requires adding the new term cz for the new dimension. We've also changed the sign of the constant term d for compactness. This is a conceptual move, not an algebraic one.

Equation of a Hyperplane

$$w_1 x_1 + \cdots + w_n x_n + w_0 = 0 \tag{4.5}$$

This is a generalization of the equation of a 3-D plane for n dimensions. It also happens to be the definition of a *linear equation*. The variables $w_1, \ldots, w_n$ are called *coefficients*, *parameters* or *weights*, and the variable w_0 is the constant term. It is important to avoid confusing x_i with $\vec{x_i}$, the first one is the coordinates of a point in some dimension, and the other is an observation of a dataset (a point). In particular, it may be the case that $\vec{x_i} = (x_1, x_2, \ldots, x_i, \ldots, x_n)$.

We may want to compact this expression more by using vectors. So, another form for representing the equation of a hyperplane is:

$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 = 0 \tag{4.6}$$

Where $\mathbf{w}$ is called the *weight vector* and w_0 the *bias*. We can turn this equation into a discriminant function for two classes by simply considering what happens with points that are not in the hyperplane. By definition such points meet one of the two inequations:

$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 > 0$$
$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 < 0$$

A point will be a solution of one of these inequations depending on whether it is in the subspace, i.e. discriminant region, facing the direction of the normal or the opposite.

A linear binary classification learning machine is thus an algorithm that given some dataset finds appropriate values for the parameters $\mathbf{w}$ and w_0 of the hyperplane in order to produce a discriminant function $y : \mathbb{R}^n \to \mathbb{R}$ such as:

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 \tag{4.7}$$

4.1.4 Performance

Usually models produced by a learning machine do not always make correct predictions, instead we consider them good enough if they can classify new data correctly most of the time. In order to quantize how good of a predictor some model is we can use various performance metrics.

The most typical metric for classification problems is *accuracy*. This is the ratio of correct predictions versus the total number of predictions made. The inverse to the accuracy ($1.0 - \mathit{accuracy}$) is defined as the *error*. Note that the classification accuracy for a binary problem will always be some percentage above 50%. This is because

a classifier performing consistently worse, can be turned into a good classifier by simply flipping the output of the discriminant function. Thus, the worst possible classifier is that of a coin toss, i.e. a uniform random distribution, with an expected accuracy of exactly 50%.

In some problems a distinction is made between misclassifications depending on which class is misclassified. An example of this is how misclassifying a patient with cancer with a healthy diagnosis is worse than misclassifying a healthy patient with a cancer diagnosis, the consequence of one is potentially death due to lack of treatment while the other is simply more investigation. For these problems a quadratic amount of cases appears. For the binary classification the standard nomenclature is to name the classes *positive* and *negative* and then prefix them with *true* or *false* depending on whether the prediction was correct or not. In this way a matrix called *confusion matrix* is created, with the diagonal containing all the correct predictions.

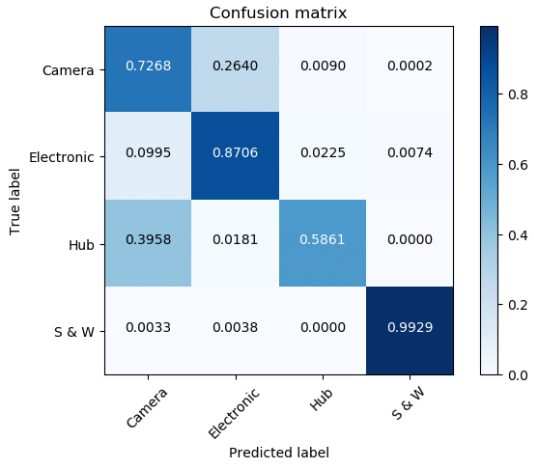


FIGURE 4.2: Confusion matrix extracted from the paper "Automatic Device Classification from Network Traffic Streams of Internet of Things" (Bai et al., 2018).

From a confusion matrix various other metrics can be extracted, such as *precision*, *specificity* or *recall*, but it is unlikely that we make use of them in this project. Often models internally use a *loss* or *cost* function that they try to minimize in order to optimize the parameters, the inverse of which is called *utility function*. Even if the model doesn't internally use one such function, one can be constructed form performance metrics, which is then referred as the *score*.

It is known that learning requires both *generalization* and *memorization*. If a model memorizes the dataset, thus has high accuracy on data already in the dataset, but doesn't generalize well, thus has low accuracy when predictions are done on new data, we say there is *overfitting*. Notice that for this to be detected we must test the dataset with new data. In order to do so a dataset is often split in *train* and *test* subsets, and although accuracies form both subsets may be reported, only the accuracy of the test subset may be taken as good.

One of the possible causes of overfitting is the *Bias-Variance* trade-off. It's an effect in which if you select a very simple model then the algorithm fails to generalize (bias) but selecting a very complex algorithm increases memorization and leads to high variability in the presence of an unseen observation (variance). The best model is thus one with a middle-ground complexity.

Selecting the complexity of an algorithm can be done with the use of some extra parameters, not fitted by the training phase of the leaning machine, that we call *hyper-parameters*. The existence of this extra parameters depends on the models, some may not have any. Because these parameters modify the model, searching for the best hyper-parameters is called *model selection*. Model selection is often done in a brute force manner, by simply trying different values of the hyper-parameters and selecting the ones that give the best result. Although this coarse strategy is usually enough, more complex search strategies such as successive halving or genetic algorithms (Claesen and De Moor, 2015) can also be used. In the case of one single hyper-parameter we can draw a plot and visualize how the score evolves. Because model selection may be understood as some kind high level training, specially when complex search strategies are used, a third division on the dataset may be made, named the *validation* set.

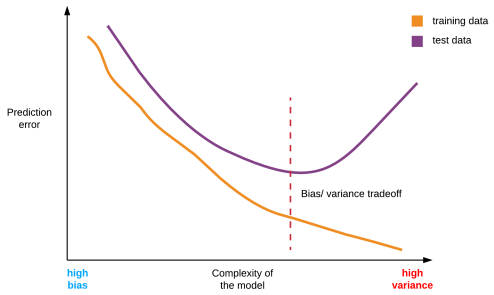


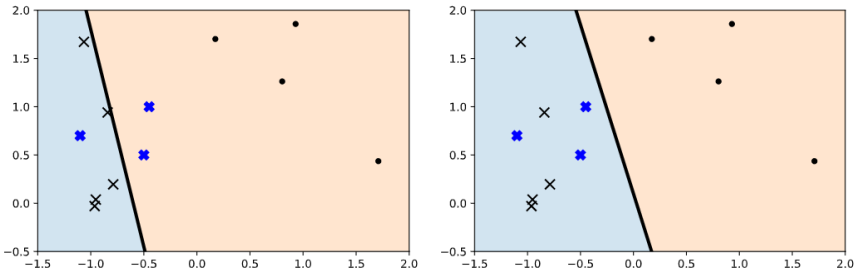
FIGURE 4.3: Bias-Variance trade-off. Source (Bisong, 2021).

Machine learning algorithms work better the more observations they are trained with, however it is not often the case that we have an unlimited amount of them. In analyzing the performance of a leaning machine, or doing model selection, it may be useful to repeat the experiments multiple times with different data of the same distribution, i.e. from the same dataset. This provides second order statistics, such as the mean or the covariance, on the resulting scores, which may prove very useful to get a better idea of what is going on. This however implies further slicing the dataset in as many slices as experiments one wants to do. After so much slicing, the remaining training subsets would often contain very few observations, thus reducing the performance of the models and nullifying the advantages of having a probability distribution on the scores.

A method called *cross-validation* can be used to make multiple experiments without increasing the amount of data. The method first makes K partitions called *folds*. For each fold, one experiment is made such that the fold is used as the validation set and the remaining folds become the training set. This allows a portion $(K-1)/K$ of the observations to be used for training in each run and allows assessing performance using the whole dataset. In the particular case where $K = N$, useful for when the amount of observations is really low, we name this method *leave-one-out* cross-validation. It is also possible to shuffle the data, which allows even more re-utilization. Although this method is computationally expensive (since it requires running the whole experiment K times), it has the advantage of being trivial to parallelize, and thus it can run at increased speeds in computers with a multi-core CPU architecture.

4.2 Support Vector Machines

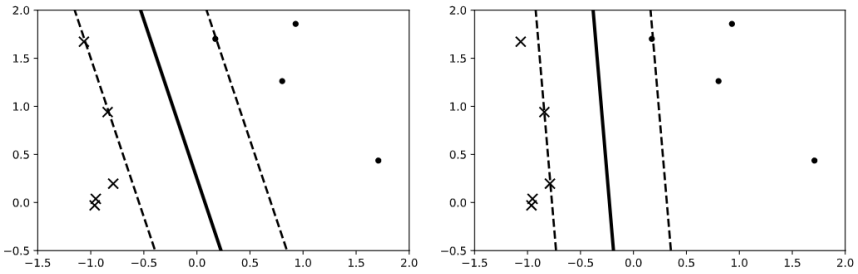
As it has been discussed in section 4.1.3, the decision boundary of a binary classification problem may be represented with a hyperplane, and it is the weights of this hyperplane what the learning machine tries to fit such that the hyperplane correctly separates the two classes. There may be infinitely many hyperplanes that correctly separate the examples of two classes, but some of them are better suited than others. For the purpose of generalization, we want decision regions that can accommodate new data outside the convex hulls[4] of each group of examples. That is, we want some kind of separation between the hyperplane and the observations.



Bad boundary Good boundary

FIGURE 4.4: Showing how one decision boundary may be better than another. `X` filled in blue represents new data.

There exist various learning machines that accomplish this separation, e.g. the *Fisher's linear discriminant*. Another such machine is the Support Vector Machine (SVM), which is based on the idea of finding the biggest margin between the extreme points of each class and setting the decision boundary in the center of such margin. For this reason SVM machines are said to be *Maximum Margin Classifiers*.



Maximum margin Quite good margin

FIGURE 4.5: Two quite good decision boundaries with a margin, a maximum margin classifier would choose the left one.

The *margin* is defined as the perpendicular distance (i.e. in the direction of the normal) between the decision boundary and the closest of the examples. There is only one decision boundary that maximizes the margin. Only a limited amount of examples, defined by the number of dimensions, will reach the limits of that margin (and thus be the closest), these are called the *support vectors*.

[4] In geometry, the *convex hull* or *convex envelope* of a shape is the smallest convex set that contains it.

4.2.1 Discriminant function

We've already seen how to we can build a discriminant function from the equation of the hyperplane. Here we will explore this in a bit more depth and take a different approx.

Given a vector of unknown length $\vec{w}$ that is perpendicular to the hyperplane (thus the normal) such that its origin is at the origin of the coordinate system. And given another vector $\vec{x}$ also with origin at the origin of the coordinate system. Note that, by how we are defining them, they are actually points, but we may want to think of them as vectors for now. We are interested in knowing in which decision region the point $\vec{x}$ is placed.

To do so, we make the dot product between $\vec{w}$ and $\vec{x}$, which, in particular, will give us a scalar corresponding to the length of the projection of $\vec{x}$ over $\vec{w}$. Note that for points within the hyperplane, that length will be some constant value c. Thus, we can know if point $\vec{x}$ is in one region or another by comparing it with c. This can be expressed with the inequality $\vec{w} \cdot \vec{x} \geq c$. By making a variable $b = -c$ and expressing the dot product as a product of matrices, we can conveniently rearrange this equation as the decision rule:

$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + b \geq 0 \tag{4.8}$$

This is analogous to the decision rule found in section 4.1.3, in fact if we make an equality instead of an inequality we discover the equation of the hyperplane, from there we could do the process in reverse until we find the general equation of a line.

4.2.2 Margin formalization

Notice from equation 4.8 that the constant value of b depends directly on the length $||\vec{w}||$, and because this length is not limited (there are infinite vectors that are normal to the hyperplane), an infinite amount of combinations exists.

Therefore, we may want to restrict the decision rule to a single combination. Also, it is convenient that this restriction takes such a form that allows defining the margin. First we assign to our classes the numerical values $C = \{-1, 1\}$. Then we impose the condition that points in a decision region outside the margin must have values greater than ± 1, thus for points $\mathbf{x}_{(+)}$ in class "1" and $\mathbf{x}_{(-)}$ in class "-1" we get:

$$\mathbf{w}^{\mathrm{T}}\mathbf{x}_{(+)} + b \geq 1$$
$$\mathbf{w}^{\mathrm{T}}\mathbf{x}_{(-)} + b \leq -1$$

This pair of equations can be simplified to a single equation by considering the vector $\vec{y}$ that we defined in section 4.1.2. Because of the numerical values we've assigned to the classes, it will contain elements such that every $y_i \in \{-1, 1\}$. Notice that if we multiply both sides by y_i (and given that we know what the value of y_i will be in each case), we can produce the single equation:

$$y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) \geq 1 \tag{4.9}$$

From this equation we may find the values of the hyperplanes that border the margin (named *gutter*), that is, the points where the last equation is exactly 1.

Equation of the gutters

$$y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) - 1 = 0 \tag{4.10}$$

Although now we know the equation for the gutters (analogy for the margin being a street), what we're really interested about is in the distance between them. We can calculate that distance as the difference vector form any two points in the gutters projected using the unit vector of $\vec{w}$:

$$(\vec{x}_{(+)} - \vec{x}_{(-)}) \cdot \frac{\vec{w}}{||\vec{w}||}$$

Notice that form equation 4.10 we can derive $\vec{w} \cdot \vec{x}_{(+)} = 1 - b$ and $\vec{w} \cdot \vec{x}_{(-)} = b - 1$, applying the distributive property we obtain:

$$\frac{(1-b) - (b-1)}{||\vec{w}||} \implies \frac{2}{||\vec{w}||}$$

Since we've defined the margin to be the distance from the decision boundary to a gutter (i.e. where support vectors are), and since the decision boundary is at the same distance to both gutters, then we can see that the length of the margin will be half the distance between the gutters.

Length of the margin

$$\frac{1}{||\vec{w}||} = \frac{1}{\sqrt{\mathbf{w}^{\mathrm{T}}\mathbf{w}}} \tag{4.11}$$

4.2.3 Optimization problem

The problem of finding the greatest margin can be formulated as an optimization problem. Specifically we want to maximize $1/||\vec{w}||$ subject to the constraints defined in equation 4.9. This is equivalent to minimizing $||\vec{w}||$ subject to the same constraints. For mathematical convenience (which we will see later), we can also divide by 2 and square the objective function without it affecting the optimal solution.

Primal form

$$\arg\min_{\vec{w},b} \frac{1}{2}||\vec{w}||^2 \quad \text{s.t.} \quad \forall i : y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) - 1 \geq 0 \tag{4.12}$$

This is a constrained quadratic optimization problem. There are various known algorithms that can solve constrained optimization problems, such as the *simplex* algorithm. However, in our case it may be appropriate to use the *Lagrangian* method. This method doesn't directly return a solution, instead it transforms the problem into another version, from which a direct solution may be more easily computed.

Lagrangian

Imagine we had an easier optimization problem. E.g. a maximization problem such that its optimization function $f : \mathbb{R}^{m-1} \to \mathbb{R}$ is a function $f(\vec{x})$ of m dimensions,

and it has a single equality constrain $g(\vec{x}) = 0$ of $D-1$ dimensions. Notice how the constraint is "projected" on top of $f(\vec{x})$. Then the Lagrangian method consists on optimizing a new function (equation 4.13), via introducing a new parameter λ called the *Lagrangian multiplier*.

$$L(\vec{x}, \lambda) = f(x) - \lambda g(x) \tag{4.13}$$

Notice how there is only a subset of the images of $f(\vec{x})$ for which f and g intersect. Where the intersection is defined as $\{\vec{x} \in \mathbb{R}^{D-1} \mid \exists \vec{c} : f(\vec{x}) = c \wedge g(\vec{x}) = 0\}$. Many values of c contribute to the intersection, however we are only interested in the maximum or minimum values, since for these $\vec{x}$ becomes a *stationary point*.

An interesting property of these functions is that in the stationary points the direction of the normal of both functions must be the same. Note that if the normal of f was not orthogonal to the surface of g, we could increase c by moving a short distance along the constraint surface. Also remember that the normal is the same as the gradient of that function, i.e. ∇f. Therefore, there must exist a parameter $\lambda \neq 0$ (or $\lambda > 0$ for a minimization problem) such that:

$$\nabla f - \lambda \nabla g = 0 \tag{4.14}$$

From this we see that the constrained stationary condition is obtained by setting $\nabla_x L = 0$. Notice that the gradient is defined as the partial derivatives of each coordinate in the vector, that is:

$$\nabla_x L = \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \cdots, \frac{\partial L}{\partial x_n} \right)$$

At first glance it may look overwhelming, but given that L is the same for all x_i and that we can reuse the arithmetic operators for operations between vectors, doing the gradient is very similar than doing the partial derivate over a single scalar.

Going back to the SVM formalization, we can define the Lagrangian of the primal form (equation 4.12) as:

$$L(\vec{w}, b, \vec{\alpha}) = \frac{1}{2}||\vec{w}||^2 - \sum \alpha_i \left[y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) - 1 \right] \tag{4.15}$$

Now we want to take the partial derivative with respect to $\vec{x}$ and equal to 0. Note that $||\vec{w}|| = \sqrt{\mathbf{w}^{\mathrm{T}}\mathbf{w}}$, if we elevate this expression to the power of two we can eliminate the square root. The derivative for one component is $\frac{\partial[(1/2)\mathbf{w}^{\mathrm{T}}\mathbf{w}]}{\partial w_i} = w_i$, combining them (derivative of the vector) we get $\vec{w}$. Therefore:

$$\frac{\partial L}{\partial \vec{w}} = \vec{w} - \sum \alpha_i y_i \vec{x}_i = 0 \implies \vec{w} = \sum \alpha_i y_i \vec{x}_i \tag{4.16}$$

Also, by doing the gradient with respect to b and comparing with 0 we get:

$$\frac{\partial L}{\partial b} = -\sum \alpha_i y_i = 0 \implies \sum \alpha_i y_i = 0 \tag{4.17}$$

If we plug these expressions back in equation 4.15 we can construct the *dual form*. Note that $||\vec{w}||^2 = \mathbf{w}^{\mathrm{T}}\mathbf{w}$.

$$
\begin{aligned}
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \sum \alpha_i \left[y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) - 1\right] \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \sum \alpha_i \left[y_i \mathbf{w}^{\mathrm{T}}\mathbf{x_i} + y_i b - 1\right] \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \sum \left[\alpha_i y_i \mathbf{w}^{\mathrm{T}}\mathbf{x_i} + \alpha_i y_i b - \alpha_i\right] \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \sum \alpha_i y_i \mathbf{w}^{\mathrm{T}}\mathbf{x_i} - \sum \alpha_i y_i b + \sum \alpha_i \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \sum \alpha_i y_i \mathbf{w}^{\mathrm{T}}\mathbf{x_i} - b\sum \alpha_i y_i + \sum \alpha_i \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \sum \alpha_i y_i \mathbf{w}^{\mathrm{T}}\mathbf{x_i} + \sum \alpha_i \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \vec{w} \cdot \sum \alpha_i y_i \mathbf{x_i} + \sum \alpha_i \\
L &= \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) - \left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right) + \sum \alpha_i \\
L &= \sum \alpha_i - \frac{1}{2}\left(\sum \alpha_i y_i \vec{x_i}\right) \cdot \left(\sum \alpha_j y_j \vec{x_j}\right)
\end{aligned}
$$

Dual form

$$
\arg\max_{\vec{\alpha}} \sum \alpha_i - \frac{1}{2}\sum\sum \alpha_i \alpha_j y_i y_j \vec{x_i} \cdot \vec{x_j} \quad \text{s.t.} \quad \forall \alpha_i : \alpha_i \geq 0 \wedge \sum \alpha_i y_i = 0 \tag{4.18}
$$

This variation of the optimization problem has the immediate advantage of optimizing over $\vec{\alpha}$ instead of $\vec{w}$. In cases where the number of dimensions exceeds the number of examples, solving this formulation is more efficient. Another historical reason for which the dual form has been preferred is because it allows using kernels (Chapelle, 2007), something we will see later in section 4.3.

Both this form and the primal form will require the use of a quadratic optimization solver. The general complexity of such algorithms is $O(n^3)$. However, various techniques can be used that accomplish to reduce this complexity for both the primal and the dual forms.

For this dual formulation, instead of directly finding the values of $\vec{w}$ we find the values of $\vec{\alpha}$. If we want to later calculate the values of the weight vector we can use the *representer theorem* (equation 4.16). For the constant parameter b we can use the margin boundary equation 4.10, which with some algebra can be expressed as $b = y_i - \mathbf{w}^{\mathrm{T}}\mathbf{x_i}$.

4.2.4 Regularization

Until now, we've seen how it is desirable to maximize the margin, but we've purposely ignored the common situation where examples are not linearly separable. It could simply be that classes follow a distribution that is not separable using a hyperplane, in which case we would use kernels. But it could also be the case that although classes are close to be linearly separable there is some overlapping in their distributions.

We can solve this problem by redefining our margin as a *soft margin*, such that we allow some amount of observations to be incorrectly classified, thus falling within the margin or in the wrong side of the decision boundary, but with a penalty that

increases with the distance to the correct side of the margin. To do so we introduce *slack variables*, one for each example, such that $\xi_i = 0$ if the example i is correctly classified, $0 < \xi_i < 1$ if within the margin, and $\xi_i \geq 1$ if it's in the wrong side of the margin.

Now we can reformulate our *primal* with this new variables, penalizing the sum of misclassifications and also introducing a *regularization parameter* C that allows to specify the desired trade-off strength between correct classification and slack.

Primal form with regularization

$$\arg\min_{\vec{w},b} \frac{1}{2}||\vec{w}||^2 + C\sum_{i=1}^{N}\xi_i \quad \text{s.t.} \quad \forall i : y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) \geq 1 - \xi_i \quad \wedge \quad \xi_i \geq 0 \tag{4.19}$$

An equivalent way to perform regularization is by using an empirical risk minimization approx. Given our decision rule (equation 4.8), we need to find a *loss function* that works well for classification problems, such as the *hinge loss*, defined as:

$$l(t) = \max\{0, 1 - t\} \quad \text{where} \quad t = y_i f(x_i) = y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) \tag{4.20}$$

This can also be written:

$$l(t) = \begin{cases} 0 & \text{if} \quad t \geq 1 \\ 1 - t & \text{if} \quad t < 1 \end{cases}$$

For a given training set we seek to minimize the total loss, while regularizing the objective with l_2-regularization (i.e. $||\vec{w}||^2$). This gives the unconstrained regularization problem equivalent to equation 4.19:

$$\arg\min_{\vec{w},b} \frac{1}{2}||\vec{w}||^2 + C\sum_{i=1}^{N}\max\{0, 1 - y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b)\} \tag{4.21}$$

Using the same Lagrangian process shown for the hard margin version, we can obtain the dual form of the soft margin version as follows:

$$L(\vec{w}, b, \vec{\xi}, \vec{\alpha}, \vec{\gamma}) = \frac{1}{2}||\vec{w}||^2 + C\sum\xi_i - \sum\alpha_i\left[y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x_i} + b) - 1 + \xi_i\right] - \sum\gamma_i\xi_i$$

Dual form with regularization

$$\arg\max_{\vec{\alpha}} \sum\alpha_i - \frac{1}{2}\sum\sum\alpha_i\alpha_j y_i y_j \vec{x}_i \cdot \vec{x}_j \quad \text{s.t.} \quad \forall\alpha_i : 0 \leq \alpha_i \leq C \wedge \sum\alpha_i y_i = 0 \tag{4.22}$$

This is notably similar to the hard margin version, where only one constrain has changed.

4.3 Kernel Methods

The use of kernel functions enables learning machines such as SVM to have non-linear decision boundaries, thus allowing them to make correct predictions even if the dataset is not linearly separable (but separable nonetheless), see Figure 4.6.

4.3.1 Feature map

A feature map is a function $\phi : D \to H$ that transforms, or maps, or projects, vectors in some space D (typically $\mathbb{R}^m$) to another space H. In the context of machine learning, D is often the *input space* (the space we've been working with until now) and H the *feature space*. When no feature map is used, there is no distinction between the two.

For the sake of using kernels, we restrict feature maps to those whose range H is a *Hilbert space*. A Hilbert space is a *metric space* that defines an inner product and also is *complete* with respect to the distance function induced by that inner product (*Wikipedia - Hilbert space*, 2021).

To verify that a space is a real metric space we need to see that, for any vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ and scalars a, b:

1. The inner product is symmetric:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{x}, \mathbf{y} \rangle$$

2. The inner product is lineal:

$$\langle a\mathbf{x} + b\mathbf{y}, \mathbf{z} \rangle = a\langle \mathbf{x}, \mathbf{z} \rangle + b\langle \mathbf{y}, \mathbf{z} \rangle$$

3. The inner product of the same element is positive definite:

$$\langle \mathbf{x}, \mathbf{x} \rangle \geq 0$$

   Where $\langle \mathbf{x}, \mathbf{x} \rangle = 0$ only if $\mathbf{x}$ is neutral, i.e. $\vec{x} = (0, 0, \ldots, 0)$.
   These three properties are enough to define a *product space*, to make it also a metric space we need to add the next two.

4. The triangle inequality holds:

$$d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$$

5. The more general Cauchy–Schwarz inequality, from which the triangle inequality can actually be derived.

$$|\langle \mathbf{x}, \mathbf{y} \rangle| \leq ||\mathbf{x}||\,||\mathbf{y}||$$

It is not hard to extend these properties to complex spaces, although outside the scope of this project. Finally, to see that this space is complete, we need to check that every Cauchy sequence in this space is convergent, i.e. has a limit also in the space. In other words, given a metric, such as the euclidean distance, there will always be points $\mathbf{x}$ and $\mathbf{y}$ such that for any $r > 0$ we have that $d(\mathbf{x}, \mathbf{y}) < r$. Euclidean spaces $\mathbb{R}^n$ as well as the complex space $\mathbb{C}$, and others, are examples of Hilbert spaces.

4.3.2 Kernel functions

A *similarity function* is a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that uses some similarity measure (e.g. euclidean distance) to determine if two objects (e.g. points, vectors) are similar and returns a real number specifying how much. Kernel functions are a class of similarity functions for which a feature map is implicitly defined, such that:

$$k(\mathbf{x_i}, \mathbf{x_j}) = \langle \phi(\mathbf{x_i}), \phi(\mathbf{x_j}) \rangle \tag{4.23}$$

We can compute explicitly a kernel function if we have the feature mapping defined, for instance, given:

$$\phi : \mathbb{R}^2 \to \mathbb{R}^4 \qquad \phi(x_1, x_2) = (x_1^2, x_2^2, x_1 x_2, x_1 x_2)$$

Remember that the inner product in an euclidean space $\mathbb{R}^n$ is defined as the dot product. Then the kernel function, for two points $\mathbf{a}, \mathbf{b} \in \mathbb{R}^2$ is:

$$\begin{aligned} k(\mathbf{a}, \mathbf{b}) &= \langle \phi(\mathbf{a}), \phi(\mathbf{b}) \rangle = \langle \phi(a_1, a_2), \phi(b_1, b_2) \rangle \\ &= a_1^2 b_1^2 + a_2^2 b_2^2 + a_1 a_2 b_1 b_2 + a_1 a_2 b_1 b_2 \\ &= (a_1 b_1)^2 + (a_2 b_2)^2 + 2(a_1 b_1)(a_2 b_2) \\ &= (a_1 b_1 + a_2 b_2)^2 = \langle \mathbf{a}, \mathbf{b} \rangle^2 \end{aligned}$$

Notice how a kernel function may actually simplify the computation and reduce the required amount of operations compared to actually transforming the points and then applying the inner product. In this case, for instance, we see that although the feature mapping is doubling the amount of dimensions, the kernel function that uses it can be simplified to an inner product in the domain. Thus, we can also define a kernel function without defining the feature map explicitly, e.g. $k(\mathbf{a}, \mathbf{b}) = \langle \mathbf{a}, \mathbf{b} \rangle^2$. Also note that there may be multiple feature maps that result in the same kernel.

One powerful alternative technique to construct kernels implicitly is to build them out of simpler kernels, like building blocks. This can be done using a set of known properties. Given valid kernels $k_1(\mathbf{a}, \mathbf{b})$ and $k_2(\mathbf{a}, \mathbf{b})$, with $\mathbf{a}, \mathbf{b} \in \mathcal{X}$, the following kernels are also valid:

$$k(\mathbf{a}, \mathbf{b}) = c k_1(\mathbf{a}, \mathbf{b}) \tag{4.24}$$
$$k(\mathbf{a}, \mathbf{b}) = f(\mathbf{a}) k_1(\mathbf{a}, \mathbf{b}) f(\mathbf{b}) \tag{4.25}$$
$$k(\mathbf{a}, \mathbf{b}) = q(k_1(\mathbf{a}, \mathbf{b})) \tag{4.26}$$
$$k(\mathbf{a}, \mathbf{b}) = \exp(k_1(\mathbf{a}, \mathbf{b})) \tag{4.27}$$
$$k(\mathbf{a}, \mathbf{b}) = k_1(\mathbf{a}, \mathbf{b}) + k_2(\mathbf{a}, \mathbf{b}) \tag{4.28}$$
$$k(\mathbf{a}, \mathbf{b}) = k_1(\mathbf{a}, \mathbf{b}) k_2(\mathbf{a}, \mathbf{b}) \tag{4.29}$$
$$k(\mathbf{a}, \mathbf{b}) = k_r(\phi(\mathbf{a}), \phi(\mathbf{b})) \tag{4.30}$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial function with nonnegative coefficients, and k_r only applies to euclidean spaces $\mathbb{R}^n$ (Bishop, 2006).

Using this knowledge we can construct some of the most popular kernels:

Linear kernel

$$k(\mathbf{a}, \mathbf{b}) = \langle \mathbf{a}, \mathbf{b} \rangle \tag{4.31}$$

Polynomial kernel

$$k(\mathbf{a}, \mathbf{b}) = (\langle \mathbf{a}, \mathbf{b} \rangle + c)^d \tag{4.32}$$

For some $d \in \mathbb{N}$ and $c \geq 0$, when $c = 0$ is said to be homogeneous.

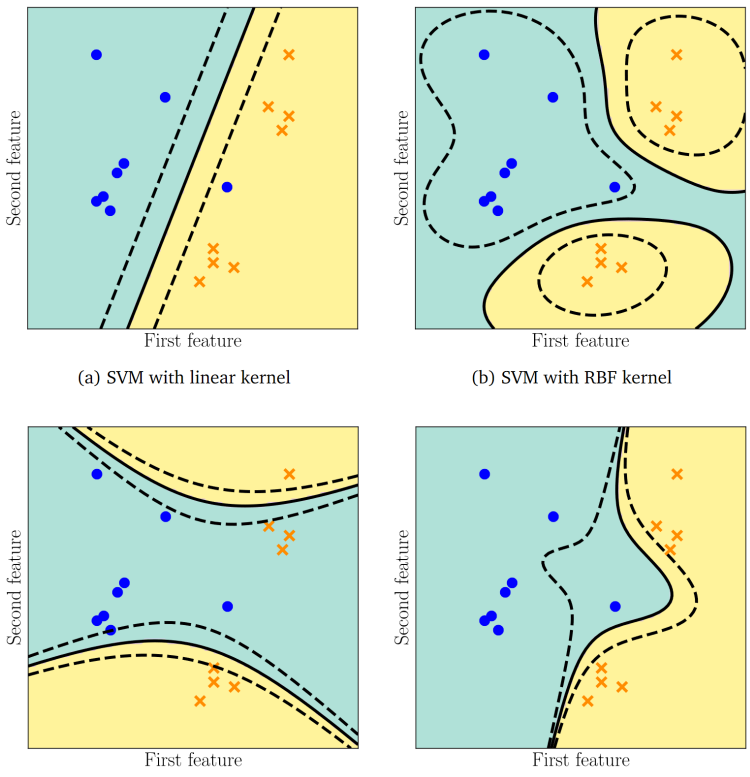
RBF (Radial Basis Function) / Gaussian / Squared Exponential Kernel

$$k(\mathbf{a}, \mathbf{b}) = \exp(-\gamma ||\mathbf{a} - \mathbf{b}||^2) \tag{4.33}$$

Where γ is a parameter that sets the "spread" of the kernel and $\exp(x) = e^x$. Recall that a Gaussian distribution has a bell-shaped curve, that is, closer points have more similarity (and thus a greater value) than more separated points. By setting $\gamma = \frac{1}{2\sigma^2}$ we can write it in the equivalent Gaussian form:

$$k(\mathbf{a}, \mathbf{b}) = \exp\left(-\frac{||\mathbf{a} - \mathbf{b}||^2}{2\sigma^2}\right) \tag{4.34}$$

This kernel has multiple interesting properties. It can be expressed as an infinite sum of polynomial kernels, this implies that the projection, i.e. the feature space, is a space with infinite dimension (Bernstein, 2017). In contrast with other kernels, where using the implicit kernel function only provides an improvement in computational cost, in this case not needing to calculate the feature map explicitly allows turning a problem that is not computable into one that can be computed trivially.



(a) SVM with linear kernel

(b) SVM with RBF kernel

(c) SVM with polynomial (degree 2) kernel

(d) SVM with polynomial (degree 3) kernel

FIGURE 4.6: Input space decision boundary of SVM with different kernels. Source (Deisenroth, Faisal, and Cheng Soon Ong, 2020).

4.3.3 The kernel trick

Recalling the SVM formulation with regularization, equation 4.19, what we want to do now is to use a feature map such that the SVM operates on the feature space instead of the input space. If we modify every instance $\mathbf{x}$ with $\phi(\mathbf{x})$, replace the dot product with the inner product, and produce the dual using the Lagrangian method over this new formulation, we will obtain:

$$\arg\max_{\vec{\alpha}} \sum \alpha_i - \frac{1}{2}\sum\sum \alpha_i\alpha_j y_i y_j \langle \phi(\vec{x_i}), \phi(\vec{x_j})\rangle \quad \text{s.t.} \quad \forall \alpha_i : 0 \leq \alpha_i \leq C \wedge \sum \alpha_i y_i = 0$$

Notice that with this new formulation we can replace the inner product with a kernel function and profit from all the advantages that kernels provide over having to explicitly compute the inner product in the feature space. This is called the "kernel trick".

SVM Dual form (kernel version)

$$\arg\max_{\vec{\alpha}} \sum \alpha_i - \frac{1}{2}\sum\sum \alpha_i\alpha_j y_i y_j k(\mathbf{x_i}, \mathbf{x_j}) \quad \text{s.t.} \quad \forall \alpha_i : 0 \leq \alpha_i \leq C \wedge \sum \alpha_i y_i = 0 \quad (4.35)$$

With this new version we also have the possibility to precompute all values of the kernel. We do this by defining a matrix $K_{i,j} = k(\mathbf{x_i}, \mathbf{x_j})$ called *Gram matrix* or sometimes *kernel matrix*. Because of the properties of kernels, a kernel matrix will always be symmetric and positive semi-definite, i.e. $\forall \mathbf{z} \in \mathbb{R}^n : \mathbf{z}^T K \mathbf{z} \geq 0$.

In the context of image processing, a convolution between a kernel matrix (also called *convolution matrix*) and an image is used to do all kinds of filtering, including blurring, sharpening, embossing, edge detection, and more. This specific kernel matrix is generated using all discrete points representing pixel positions (the center of cells in a grid) with an origin in the center of the grid.

Sometimes the dual SVM formulation will be expressed in terms of the kernel matrix directly. Another matrix formulation can be done by using the *Hessian*[5] matrix, which for SVM is defined as $\mathbf{H_{i,j}} = y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$. This allows expressing equation 4.35 as a cost function (to be minimized) with the formula $(1/2)\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}\boldsymbol{\alpha} - \alpha^{\mathrm{T}}\mathbf{1}$, where $\mathbf{1}$ (in bold) is a vector of 1.

Another advantage of using kernels is that, since they don't restrict the input space to real numbers, you can now use SVM to classify between all kind of objects, e.g. sets, sequences, strings, graphs and distributions.

4.4 SVM-RFE

Continuing our discussion on section 1.1.2, SVM-RFE is an embedded feature selection algorithm. In this section we shall explore in more detail how and why this algorithm works.

[5] A Hessian matrix is a square matrix of second-order partial derivatives of a function. In our case, the function used is the dual formulation of SVM. However, we only use this matrix here for convenience, we don't use its mathematical properties. (Sontag, 2013)

4.4.1 Ranking criteria

One basic idea of feature selection is to use a feature ranking and then select a number r of the best ranking features as the final selection. For classification problems, one possible way to produce such a ranking is to evaluate how well a single feature contributes to the separation of the classes, these are called *correlation filter methods* and the evaluation function *correlation coefficients*, e.g:

$$w_i = \frac{\mu_i(+) - \mu_i(-)}{\sigma_i(+) + \sigma_i(-)}$$

where μ_i and σ_i are the mean and standard deviations for the observation in the $(+)$ and $(-)$ class respectively. Large positive values of w_i indicate strong correlation with class $(+)$ and large negative values with class $(-)$. The absolute or the square of w_i can be used as the *ranking criteria*, i.e. the scores of each feature such that when sorted produce a feature ranking. Other similar correlation based ranking criterion include *Fisher*, *Pearson*, *Spearman* or *Kendall* coefficients.

Note that a feature that is correlated to some class is by itself a class predictor (albeit an imperfect one). This class predictors may be combined, either directly or with some scheme, e.g. *weighted voting*, to produce a linear discriminant classifier (Guyon et al., 2002).

$$D(\vec{x}) = \vec{w} \cdot (\vec{x} - \mu)$$

The opposite is also true, given a wight vector, e.g. one produced by an SVM, the individual coordinates of the vector can be interpreted as individual correlation coefficients. The inputs that are weighted by the largest value influence most the classification decision. Therefore, if the classifier performs well, those inputs with the largest weights correspond to the most informative features.

An alternative geometric interpretation, which leads to the same result, can also be made. By definition, the wight vector produced by a linear SVM corresponds to the normal vector of the decision boundary (a hyperplane). Setting one of the coordinates of such vector $\vec{w}_i$ to 0 will change its direction and thus produce a rotation on the hyperplane. This rotation will modify the decision regions, thus, points located in the space where the region changes (the intersection space) will be classified incorrectly (assuming no regularization). If we want to reduce the classification error a good idea is to reduce the space in the intersection, which is equivalent to reducing the amount of rotation on the hyperplane or the change in direction of the normal vector. Coordinates with a value close to 0, when the value is set to 0, will produce small changes in direction, whilst coordinates where the difference $|0 - w_i|$ is bigger will produce bigger changes. I.e, coordinates with the largest absolute weights correspond to the most informative features (when they are removed more error is produced) and coordinates with the lowest absolute weight are the less informative. As such, we can make a ranking criterion by simply taking the absolute value $|w_i|$, or the square $(w_i)^2$, of the coordinates of the wight vector produced by an SVM.

One problem of this interpretation is that it only works when the decision boundary is lineal. However, it gives us a hint on how to make a generalization for non-linear decision boundaries: We can use the change in objective function (or cost function) when a feature is removed as a ranking criterion. Let J be the SVM cost function $J = (1/2)\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}\boldsymbol{\alpha} - \alpha^{\mathrm{T}}\mathbf{1}$. First, we train the machine and compute the $\boldsymbol{\alpha}$ parameters. Then, to compute the change in cost function $DJ(i)$ caused by removing

the feature i, we leave the $\boldsymbol{\alpha}$ unchanged (for performance reasons) and recompute the matrix $\mathbf{H}$, yielding $\mathbf{H}(-i)$ where the notation $(-i)$ means that the component i has been removed. The resulting ranking coefficient is:

$$DJ(i) = [(1/2)\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}\boldsymbol{\alpha} - \alpha^{\mathrm{T}}\mathbf{1}] - [(1/2)\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}(-i)\boldsymbol{\alpha} - \alpha^{\mathrm{T}}\mathbf{1}]$$

General ranking coefficient for SVM

$$DJ(i) = (1/2)(\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}\boldsymbol{\alpha} - \boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}(-i)\boldsymbol{\alpha}) \tag{4.36}$$

Note that when the kernel is lineal then $\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}\boldsymbol{\alpha} = ||\mathbf{w}||^2$, therefore:

$$\begin{aligned} DJ(i) &= (1/2)(||\mathbf{w}||^2 - ||\mathbf{w}(-i)||^2) \\ &= (1/2)[(w_1^2 + w_2^2 + \cdots + w_D^2) - (w_1^2 + \cdots + w_{i-1}^2 + 0 + w_{i+1}^2 + \cdots + w_D^2)] \\ &= (1/2)(w_i^2) \quad \sim \quad (w_i)^2 \end{aligned}$$

Computationally, the non-lineal version is more expensive, however we may be able to optimize it by only recomputing support vectors (since $\boldsymbol{\alpha}$ is unchanged), and by caching partial results on the other components of these vectors.

There are other ranking coefficients that use the information form the model trained by some learning machine, e.g. *Relevancy and Redundancy Criteria* (Mundra and Rajapakse, 2007), or the *Span Estimate Gradient* (Rakotomamonjy, 2003).

Particularly, one method that also returns a ranking coefficient $(w_i)^2$ for the lineal case is based on the OBD algorithm (Guyon et al., 2002). It approximates the difference with a Taylor series expansion[6] to second order. At the optimum of J the first order term can be neglected, yielding:

$$DJ(i) = (1/2)\frac{\partial^2 J}{\partial w_i^2}(0 - w_i)^2 = \frac{\partial \mathbf{w}}{\partial w_i}(w_i)^2 = (w_i)^2$$

4.4.2 Recursive Feature Elimination

The criteria $DJ(i)$ estimates the effect of removing one feature at a time on the objective function. It doesn't perform particularly well when several features are removed. This problem can be overcome by using an iterative procedure, called RFE (Recursive Feature Elimination), where we eliminate one feature and retrain the learning machine each iteration.

This is an instance of backward feature elimination. For performance reasons it may be more efficient to remove multiple features at a time. We introduce a step parameter t for that. In this case the method produces a feature subset ranking, a nested set of the form $F_1 \subset F_2 \subset \cdots \subset F$. This feature subset ranking may better be represented as a list of vectors, such that when flattened produces the final feature ranking.

Since the ranking criteria only considers the removal of one feature, while RFE fixes this problem by applying the ranking criteria at each feature subset size, we conclude that RFE will generate better feature rankings the smaller the step.

[6] Any real or complex differentiable function can be expressed as a Taylor series, which is a polynomial of the form $f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots$, where a is some point in the domain. The function can thus be approximated by restricting the polynomial to some order n.

Note that with this method, top ranked features are not necessarily the most relevant when taken individually, it is when taken together that features of a subset F_m are relevant in some sense.

In the following figure we present a pseudocode implementation of the SVM-RFE algorithm, which is the particular case of RFE where SVM are used. For clarity purposes this implementation is restricted to linear SVM and the associated ranking criteria, it doesn't calculate performance nor does it store individual feature subsets.

Algorithm 1: SVM-RFE

Input: t `// t = step`
Output: $\vec{r}$
Data: $X_0, \vec{y}$

1 $\vec{s} = [1, 2, \ldots, d]$ `// subset of surviving features`
2 $\vec{r} = []$ `// feature ranking list`
3 **while** $|\vec{s}| > 0$ **do**
`/* Restrict training examples to good feature indices */`
4 $X = X_0(:, \vec{s})$
`/* Train the classifier */`
5 $\vec{\alpha} =$ `SVM-train`(X, y)
`/* Compute the weight vector of dimension length` $|\vec{s}|$ `*/`
6 $\vec{w} = \sum_k \vec{\alpha}_k \vec{y}_k \vec{x}_k$
`/* Compute the ranking criteria */`
7 $\vec{c} = [(w_i)^2 \text{ for all } i]$
`/* Find the` t `features with the smallest ranking criterion */`
8 $\vec{f} =$ `argsort`$(\vec{c})(:t)$
`/* Iterate over the feature subset */`
9 **for** $f_i \in \vec{f}$ **do**
`/* Update the feature ranking list */`
10 $\vec{r} = [\vec{s}(f_i), ...\vec{r}]$
`/* Eliminate the feature selected */`
11 $\vec{s} = [...\vec{s}(1 : f_i - 1), ...\vec{s}(f_i + 1 : |\vec{s}|)]$
12 **end**
13 **end**

Note that, although we're using vectors to store the feature ranking, by using another structure (like a dictionary in `Python`) we could store the specific feature subsets, this is actually done in our implementation.

We assume that the complexity of solving the quadratic problem of an SVM using LIBSVM is $O(dn^2)$ (Abdiansah and Wardoyo, 2015). In practice, it depends on the implementation[7] and optimizations used therein. This complexity may also change whether kernels are used or not and whether the primal or the dual form is used.

Given that the complexity of solving the dual quadratic problem of an SVM is $O(dn^2)$, this procedure requires $O(d^2n^2)$ time to finish. Note that the whole loop of line 9 can be done in constant time with the appropriate data structures.

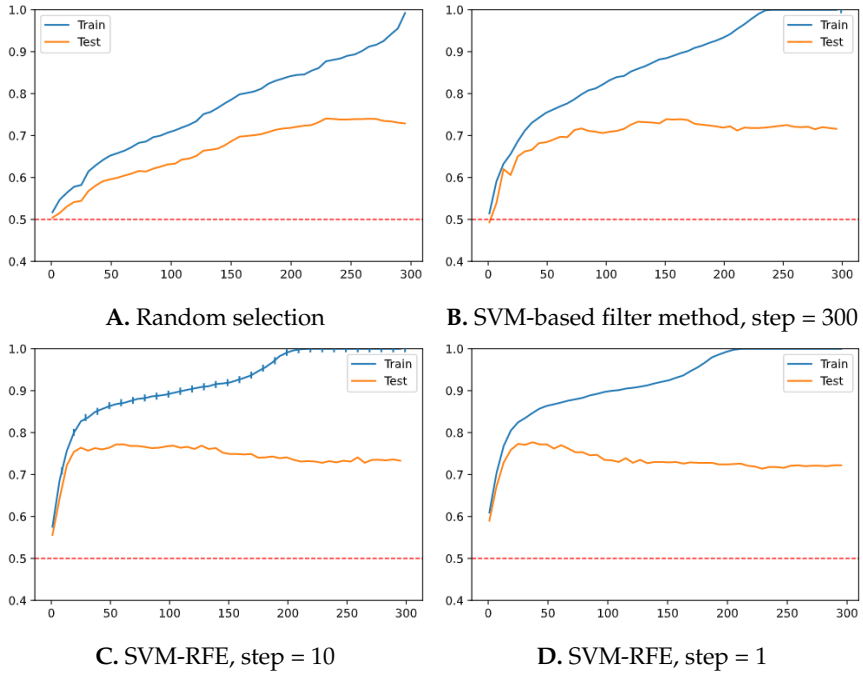
[7] Some of these algorithms include SMO (Sequential Minimal Optimization) and SGD (Stochastic Gradient Descent).

4.4.3 Assessing performance

Since learning machines will produce different levels of performance depending on the dataset, a baseline feature selection method is required in order to make proper comparisons. We can use a *random feature selection* as the baseline method. This method is a good baseline because we expect that no method should perform worse than a purely uninformed one. It simply consists in making a random feature ranking and testing the accuracy of SVM with multiple feature subsets, specifically we make each feature subset by removing t features each time based on the ranking order. For numerical stability we may also use cross-validation and take the mean of the results.

Although with this method the feature ranking is made at no cost, evaluating the accuracy at each step (which we later call the "validation phase") is just as computationally expensive as SVM-RFE itself. This is because one SVM training has to be performed each iteration. It is expected that this method will perform well when all features are used, and decrease linearly in accuracy as features are removed.

When using SVM-RFE we actually have two different methods for evaluating the performance of the feature ranking. One method is to evaluate the accuracy at each iteration of the algorithm, by first passing a validation set to it. This, however, requires to use the same hyper-parameters, including step, for training and validation. A better solution may be to decouple both phases, a ranking (SVM-RFE) phase and a validation phase. That is, we use SVM-RFE (or any other algorithm) purely for generating the ranking. After that, we fit and test at various feature subset sizes. This allows producing performance metrics with the same step regardless of the method used to create the ranking, including filter methods or a random ranking.



A. Random selection

B. SVM-based filter method, step = 300

C. SVM-RFE, step = 10

D. SVM-RFE, step = 1

FIGURE 4.7: Mean accuracy using 20-fold CV of an SVM classifier with different selection methods.

Figure 4.7 has been made with an artificial dataset of 300 features, only 50 of which informative. Note that we provide both train and test accuracy, only the

second is a valid metric of the expected accuracy on new samples, however, plotting the train accuracy is also useful to determine the overfitting as well as provide clues on the behavior of the selection algorithm.

As expected, random selection (plot **A**) performs significatively worse than any other method. We also verify, by comparing plots **B** and **C**, our previous assumption that the more iterations (smaller step) the better the accuracy. This relationship, however, doesn't seem to be lineal, as the same effect can not be appreciated when comparing plots **C** and **D**.

When evaluating the performance of a feature selection algorithm two variables must be considered, one is accuracy performance and the other is amount of features. A selection with a great accuracy but a lot of features may not be as good as a section with a decent accuracy and only a handful of features. That is, we want to maximize accuracy while minimizing amount of features. Thus, to get an idea of what the actual performance, we will use plots whenever possible.

Formally this is known as a multi-objective optimization problem. In this kind of problems a solution is not necessary unique, instead it is a set of *Pareto optimal*[8] solutions, i.e. solutions such that no other solution in the set of feasible solutions is better. Of course this also means that all Pareto optimal solutions are just as good. To score the solutions a *scalarization function* is used, such that the multi-objective problem becomes a single objective optimization problem. This function establishes a trade-off between the different objectives and must be chosen based on expert criteria, i.e. an expert decides what is more important for the specific problem at hand.

If the criteria is known beforehand, it can be used to guide the optimization problem. In our case, for example, it could be used to guide model selection, see (Section 4.1.4). This is known as an a priori method. If all feasible solutions (or a representative subset) are first computed, this is known as a posteriori method. One typical scalarization function is the weighted sum, also called *linear scalarization*.

Linear Scalarization

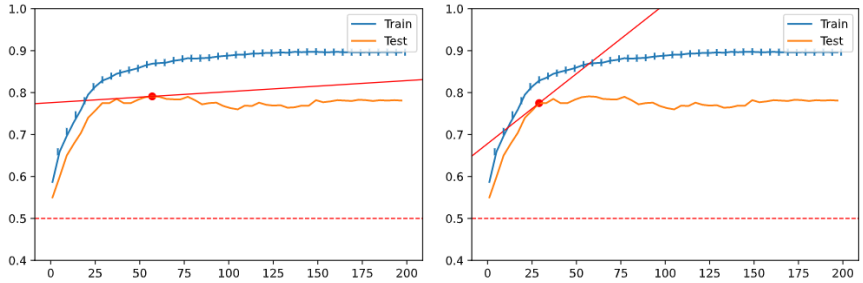
$$\arg\min_{\mathbf{x}\in\mathbf{X}} \sum_{i=1}^{k} w_i f_i(\mathbf{x}) \tag{4.37}$$

where each w_i is a weight and each $f_i(\mathbf{x})$ is a cost function (Vasumathi and S, 2019). In our case we would only have two cost functions, the error at each feature subset $1 - \texttt{Acc}(F_i)$, and the percentage of features selected $|F_i|/|F|$. Thus, the scalarization function will be of the form:

$$\arg\min_{F_i\in F} w_1(1 - \texttt{Acc}(F_i)) + w_2(|F_i|/|F|)$$

Calculating all feature subset costs allows us to find and plot the optimal feature subset size, as shown in figure 4.8.

[8] A concept more commonly used in game theory, decision theory or economics.



$w = (0.95, 0.05)$ $w = (0.6, 0.4)$

FIGURE 4.8: Different optimums caused by changing the trade-off for the linear scalarization function, projected as a red line.

Chapter 5

Experiments

This chapter is organized in sections corresponding to each of the extensions we've outlined in section 1.3. For each extension we provide a general description of the idea, the rationale behind it, a pseudocode, a brief complexity analysis, and the experimental results. The first section of this chapter does not correspond to any extension and is instead an introductory block.

5.1 Introduction

5.1.1 General Framework

The general workflow for testing the extensions is summarized in the following diagram.

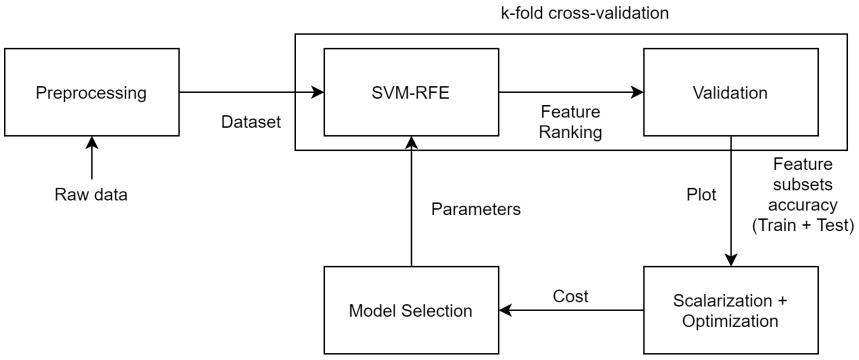


FIGURE 5.1: Experimental testing process, divided in phases.

`Sklearn`, being a general machine learning library, has its own implementation of the base `RFE` algorithm. Given that it is open source, we were able to base our own implementation on it. After pruning the code to its bare minimum (no dependencies), we began to add our own extensions. We organized each extension in its own folder, isolated from the rest, and copied the base implementation there.

For the actual experiments we've used `Jupyter Notebooks` (integrated in `Visual Studio Code`). Each experiment may use different notebooks depending on what is being tested exactly. Some code that is common to all notebooks, such a code for plotting, has been placed at the beginning (for convenience). The implementation of SVM-RFE and associated usability methods has been moved to separate `Python` files. This is both a convenient software pattern and a requirement of the parallelization library of our operating system.

We use *k-fold cross-validation* to perform the experiments multiple times with different folds, as explained in section 4.1.4. For the same experiment, the same amount of folds is used, but different amounts may be used in different experiments.

We've parallelized this procedure with the standard `multiprocessing.pool` library. Given that our CPU has 8 cores, for the most computationally expensive experiments we're using 6 or 7-fold cross-validation in order to be able to finish the experiment in a single round (with one or two spare cores to be able to keep working in the machine). The execution time is always calculated as the mean of the elapsed times of every SVM-RFE execution (single core).

For plotting we've used the `Mathplotlib` library. We have made a custom plot function that includes all relevant information for SVM-RFE. Having a standard plot design allows for easy comparison as shown in Figure 4.8. This plot displays the accuracy (vertical axis) of each feature subset (by size, horizontal axis) based on some feature ranking. We display both the train accuracy and the test accuracy calculated in the validation phase. We also show where the optimal is based on the linear scalarization method. Furthermore, we also show what the minimum accuracy is, based on the amount of classes, with a doted red horizontal line. And finally, we show with an overlay scatter plot of small vertical blue lines, what the actual feature subsets sizes are and what accuracy they had during the execution of SVM-RFE (training accuracy). We can also deduce the step used and the amount of iterations from this information. It is important not to confuse this plot with plots describing an iterative optimization process or model selection.

5.1.2 The data

Sklearn Generator

The general machine learning library `Sklearn` provides tools to generate artificial datasets for various problems. We're using the `make_classification` generator, which creates normally-distributed clusters of points placed at the vertices of an hypercube. Multiple clusters can corresponding to a single class. This specific generator also introduces interdependence between features and various types of noise.

We've selected this generator because it allows to specify the amount of informative (i.e. non-redundant, useful) features the dataset should have, it can generate multi-class datasets, and by changing the amount of clusters per class it can control the separability of the data.

We don't use a single dataset of this kind in our experiments. Instead, we use datasets generated with the parameters that we think can better help evaluate the expected properties and correctness of the extensions.

MADELON

This is one of the five datasets proposed for the NIPS (Neural Information Processing Systems) 2003 challenge in feature selection. The dataset remains publicly available in the UCI (University of California, Irvine) machine learning repository. The results of this challenge can be found at the workshop web page (Guyon et al., 2004).

The winner of the challenge got an accuracy of 92.89% with 8 selected features. However, this is using test data that is not publicly available. Using only the available data we've found articles describing the use of this dataset where the maximum accuracy reached is 88%.

This dataset is constructed similarly to the `sklearn make_classification` generator. It uses clusters of points placed at the vertices of some hypercube, however, instead of a single hypercube it uses five of them. Also, this method labels each in hypercube in one of two classes randomly. Five features correspond to the which hypercube a point is in, with an extra 15 being redundant features extracted from

Name	Observations	Features	Informative	Classes
`make_classification`	-	-	-	-
MADELON	2000	500	5	2
Digits	1797	64	??	10

TABLE 5.1: Summary of dataset properties.

linear combinations of the first 5. The total amount of features per data point is 500, 480 of them being noise (also called probes).

Of this dataset, 2000 observations are publicly available and where used in this project, 600 are part of a separate validation set not used in this project, and 1800 are part of a test set not used in this project and also not publicly available. All sets have the same amount of positive and negative samples.

Digits

For the multi-class classification extension we've used the digits dataset (Optical Recognition of Handwritten Digits Data Set). This is also a dataset available in the UCI machine learning repository and is also provided ready to use in `Sklearn`. This dataset contains 5620 instances of 64 features corresponding to 10 possible handwritten digits. Each feature has been extracted from a 32×32 bitmap by counting the pixels of 4×4 non-overlapping blocks. There are 64 informative features.

For this dataset common accuracy results are around 98%, but we've found some cases where it can reach 100%.

5.2 Dynamic Step

This extension is based on the constant step variant of SVM-RFE (Algorithm 1), however, instead of using some constant number t as the step in each iteration, we calculate that number dynamically. The most straightforward way to do this is by using a percentage.

5.2.1 Description and reasoning

The percentage is a hyper-parameter. It is used within every iteration to eliminate a number of the least ranked features. A constant step has already been used in practice, but it is expected that this method will be significantly faster without effecting the accuracy performance, or even improving it.

Other similar modifications are also found in the literature, including using the square root of the remaining features `SQRT-RFE`, an entropy based function `E-RFE`, or `RFE-Annealing` which sets the step at $|\vec{s}|\frac{1}{i+1}$, thus, changing the percentage each iteration (Ding and Wilkins, 2006).

That is, the amount of features eliminated "r" at j-th iteration is the summation of the amount of features eliminated each iteration, i.e. np^i:

$$r = \sum_{i=1}^{j} (np^i) \tag{5.1}$$

It may be more interesting to see complexity as in the amount of iterations required to eliminate r features. Assuming r and p to be constants, it can be derived from equation 5.1 as follows:

$$
\begin{aligned}
r = n\sum_{i=1}^{j} p^i = n\sum_{i=0}^{j-1}(p^i) - n + np^j \\
\frac{r}{n} = \frac{1-p^j}{1-p} - 1 + p^j \\
\frac{r}{n} = \frac{(1-p^j) - (1-p) + (1-p)p^j}{1-p} \\
\frac{(1-p)r}{n} = (1-p^j) - (1-p) + (p^j - p^{j+1}) \\
\frac{(1-p)r}{n} = 1 - p^j - 1 + p + p^j - p^{j+1} \\
\frac{(1-p)r}{n} = p - p^{j+1} \\
-\frac{(1-p)r}{n} + p = p^{j+1} \\
\log_p\left(-\frac{(1-p)r - np}{n}\right) = \log_p(p^{j+1}) \\
\log_p\left(-\frac{(1-p)r - np}{n}\right) = j+1 \\
\log_{1/p}\left(-\frac{n}{(1-p)r} + 1/p\right) = j+1 \implies O(\log_{1/p} n)
\end{aligned}
$$

We assume that, the bigger the step each iteration, the worse the performance of the ranking. This is consistent with what we've seen in figure 4.7. However, since we're eliminating the worst variables first, eliminating more of them at once shouldn't affect performance because it is likely they would've been eliminated in the next iterations anyway. The fewer the iterations remaining, the riskier it becomes to eliminate multiple variables at once, and thus a smaller step is beneficial.

Our expectations for this extension are:

- **Improvement in time complexity:** Given that SVM complexity $O(mn^2)$ is linearly dependent on the amount of dimensions d, we know that each iteration is faster than the one before it. By increasing the step in the first iterations we should drastically reduce the time it takes to complete, even if then we decide to reduce the step in later, less time-consuming, iterations.
- **Improvement in accuracy performance:** Using dynamic step, compared to a constant step $t > 1$, can also improve the accuracy. This is because the last iterations may be performed with a step $t \geq i \geq 1$.

Note that by adjusting the percentage you can decide for which of these two objectives you want to optimize, it may also be possible to find a middle case in which both the accuracy and the time are improved.

5.2.2 Pseudocode formalization

Algorithm 2: SVM-RFE with DynamicStep

Input: p // p = percentage, $0 \leq p \leq 1$
Output: $\vec{r}$
Data: $X_0, \vec{y}$

$\vec{s} = [1, 2, \ldots, m]$ // subset of surviving features
$\vec{r} = []$ // feature ranking list
while $|\vec{s}| > 0$ **do**
  /* Restrict training examples to good feature indices */
  $X = X_0(:, \vec{s})$
  /* Train the classifier */
  $\vec{\alpha} = \texttt{SVM-train}(X, y)$
  /* Compute the weight vector of dimension length $|\vec{s}|$ */
  $\vec{w} = \sum_k \vec{\alpha}_k \vec{y}_k \vec{x}_k$
  /* Compute the ranking criteria */
  $\vec{c} = [(w_i)^2 \text{ for all } i]$
  /* Compute t based on the percentage */
  $t = p|\vec{s}|$
  /* Find the t features with the smallest ranking criterion */
  $\vec{f} = \texttt{argsort}(\vec{c})(:t)$
  /* Iterate over the feature subset */
  for $f_i \in \vec{f}$ **do**
    /* Update the feature ranking list */
    $\vec{r} = [\vec{s}(f_i), ...\vec{r}]$
    /* Eliminate the feature selected */
    $\vec{s} = [...\vec{s}(1 : f_i - 1), ...\vec{s}(f_i + 1 : |\vec{s}|)]$
  end
end

Note that since we do a non-lineal skip focused on the iterations that take more time we can achieve a reduced complexity of $O(\log(d)dn^2)$. This can be illustrated in Figure 5.2, where we show the relationship between the amount of remaining features and iterations for the different step methods.

The relative time cost of each method can be estimated as the area under the curve. Note that in this plot the amount of iterations is fixed, however in practice it may make more sense that dynamic step methods perform much more late iterations than constant step.

5.2.3 Results

Analysis with artificially generated data

We generate a 2 class dataset with the following code. The scalarization trade-off used is of 80% accuracy, 20% feature subset size. All results are mean values extracted from a 7-fold cross-validation procedure.

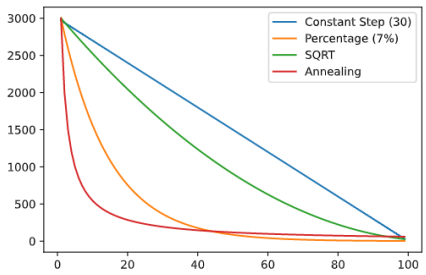
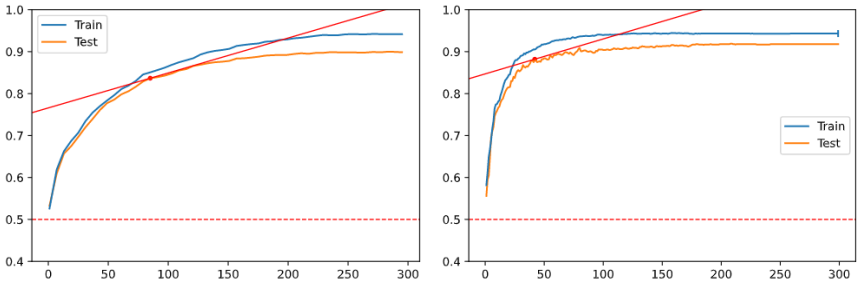


FIGURE 5.2: Amount of features remaining (vertical axis) at each iteration (horizontal axis).

```
X, y = make_classification(
    n_samples = 1000, n_clusters_per_class=3, n_features = 300,
    n_informative = 100, n_redundant=100, n_repeated=20,
    flip_y=0.05, random_state=2, class_sep=2
)
```

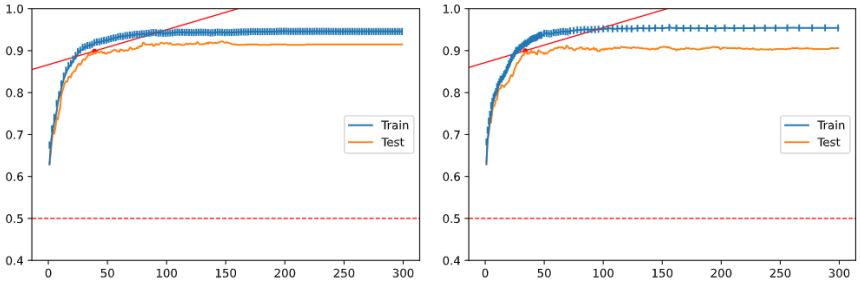
We start by comparing how the dataset performs under a random feature selection or a simple filter method. For the validation phase a linear SVM has been used, with the regularization parameter found by grid search and being $C = 0.00001$.



Random AT (85, 0.83, 0.187) Filter AT (42, 0.88, 0.122)

FIGURE 5.3: Accuracy of feature rankings produced either by a random method or an SVM-based filter method. AT (*feat.*, *acc.*, *cost*)

Based on these results we expect that SVM-RFE must perform better than the filter method. Or objective now is to find if using a dynamic step can perform similarly or better than SVM-RFE in less amount of time. From a grid search model selection procedure we'be selected the best feature rankings, shown in Figure 5.4.



Const Step $t = 2, C = 0.00001$ Dynamic Step $p = 0.05, C = 0.0001$

FIGURE 5.4: Accuracy of the best feature rankings produced by SVM-RFE with constant or dynamic step.

In the following tables we detail the results of the model selection. The cell corresponding to the best model of each algorithm (by cost) and associated time have been highlighted.

Step	**2**			**10**			**50**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.000001	39	88.40%	0.119	41	89.40%	0.112	38	88.70%	0.116
0.000010	39	89.90%	0.107	57	90.70%	0.112	54	89.80%	0.118
0.000100	55	91.00%	0.109	61	91.30%	0.110	59	90.10%	0.118
0.001000	81	89.40%	0.139	77	87.90%	0.148	81	87.10%	0.157
0.010000	104	88.70%	0.160	103	89.10%	0.156	116	87.90%	0.174

TABLE 5.2: Grid search of SVM-RFE with constant step.

C/Step	**2**	**10**	**50**
0.000001	0:03.691	0:00.643	0:00.124
0.000010	0:05.242	0:01.050	0:00.190
0.000100	0:07.001	0:01.440	0:00.279
0.001000	0:08.039	0:01.522	0:00.334
0.010000	0:11.834	0:02.424	0:00.577

TABLE 5.3: Execution time (min:sec.msec) of SVM-RFE with constant step.

Percentage	**0.04**			**0.12**			**0.20**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.000001	26	88.20%	0.112	22	86.60%	0.122	37	88.10%	0.120
0.000010	43	89.80%	0.110	36	89.80%	0.106	60	89.30%	0.126
0.000100	34	90.00%	0.103	59	91.00%	0.111	42	90.40%	0.105
0.001000	56	87.30%	0.139	63	87.20%	0.144	91	89.00%	0.149
0.010000	87	88.40%	0.151	104	87.40%	0.170	107	87.90%	0.168

TABLE 5.4: Grid search of SVM-RFE with dynamic step.

C/Percentage	**0.04**	**0.12**	**0.20**
0.000001	0:01.048	0:00.324	0:00.202
0.000010	0:01.455	0:00.443	0:00.283
0.000100	0:01.947	0:00.612	0:00.380
0.001000	0:02.505	0:00.709	0:00.404
0.010000	0:03.487	0:01.079	0:00.639

TABLE 5.5: Execution time (min:sec.msec) of SVM-RFE with dynamic step.

Note that the three best models for dynamic step are all better than the best model using constant step. This results however have to be taken with a grain of salt, given that variance is present and the difference is only marginal.

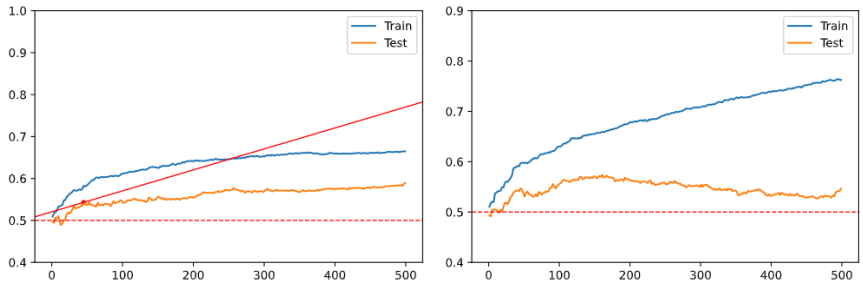
A more clear-cut improvement can be seen on the computational cost, which had a speedup of x2.7 even when the model used by dynamic step had a greater value of C. (Equation 5.2.3).

$$\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{5.242}{1.947} = 2.692 \tag{5.2}$$

```
NOTE: The speedup is better the highier the amount of features.
```

Analysis with Madelon

Like before, we start comparing how the dataset performs under a random ranking (Figure 5.5).



$C = 10^{-5}$ $C = 10^{3}$

FIGURE 5.5: Accuracy of an SVM classifier with a random feature selection with the Madelon dataset.

It seems to perform poorly, even when all features are used. We thought this could be caused by the regularization parameter, so we tried various values of C on a logarithmic scale (Figure 5.6). At first glance, however, it doesn't look like a regularization problem.

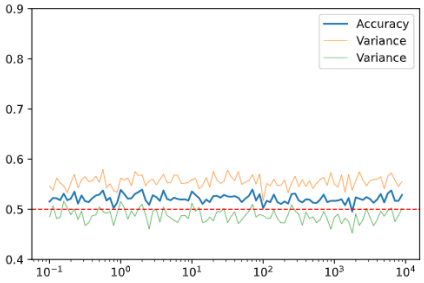
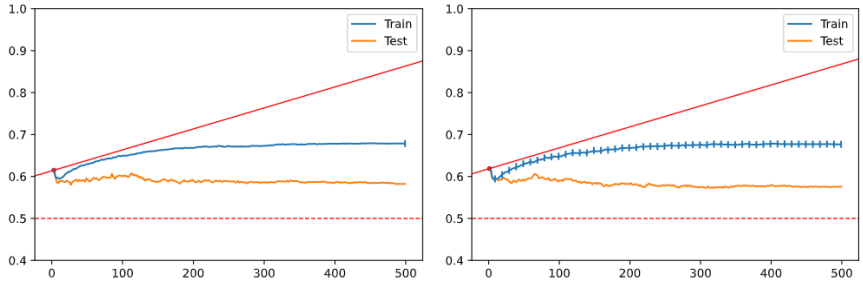


FIGURE 5.6: Hyper-parameter search of C. Displaying test accuracy of a linear SVM classifier using all features.

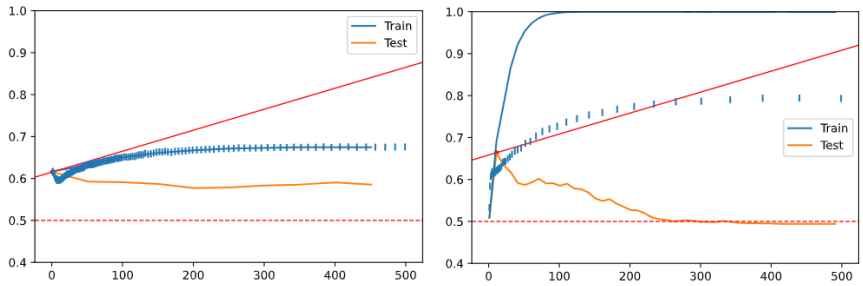
Next've we've tried to see what happens if we apply normal SVM-RFE directly, with $C = 10^{-6}$, see Figure 5.7. Unfortunately, not only is the accuracy much lower than expected (Section 5.1.2), but also no improvement can be appreciated when we decrease the step. This indicates that dynamic step will not work, as it relies on this property.

Since the problem is not regularization, then it may be that the data is not linearly separable. Although we can not implement a non-linear kernel for the SVM-RFE procedure yet (it is another extension), we may be able to improve the situation by using a non-lineal kernel on the validation phase (Figure 5.8).



Filter Step = 10

FIGURE 5.7: Accuracy of the ranking produced by SVM-RFE.



Lineal, $C = 10^{-6}$ Polynomial, degree 6, $C = 0.6$

FIGURE 5.8: Accuracy of the ranking produced by a dynamic step SVM-RFE with lineal and polynomial kernels.

As expected, dynamic step doesn't improve the accuracy in this scenario. We do see an improvement when using a non-lineal kernel on the validation phase, but again produces the same result as the filter method.

5.2.4 Conclusions

This is a summary of the pros and cons that we've found for this extension:

- Using some form of dynamic step may improve the accuracy in certain scenarios, specially in situations where there is a benefit from having a small step.
- Using a percentage doesn't introduce a new hyper-parameter since it replaces the existing step constant.
- Dynamic step is generally much faster than a constant step for the same levels of accuracy, specially when many features are used and the step is small.
- With independence of the accuracy improvement, dynamic step can still be used to speed up considerably the validation phase.
- The percentage and final point are better adjusted when the shape of the data is known.

In general, we recommend using dynamic step whenever possible, after the initial exploration of the data and if possible after the execution with the associated filter method.

5.3 Sampling

In this experiment we want to see if using a different subset of observations each iteration can reduce computational cost without negatively affecting performance.

5.3.1 Description and reasoning

Sampling is the process of selecting a representative part of a population. In machine learning sampling is done when creating a dataset, where each observation is a *sample* of some unknown distribution.

Sampling may also be done, a second time, when preparing the dataset. Classification algorithms often require that all classes have a similar number of observations, when this condition is not meet it is said that the dataset has *sampling bias*. To avoid sampling bias, the dataset is resampled with another method called *stratified random sampling*. This method consists on making a new dataset with only a subset of the observations, while imposing the restriction that all classes must have the same number of observations.

Because SVM computational cost is directly related to the amount of examples (i.e. samples), using only a subset of observations on each iteration can reduce this cost. In doing that, it is expected that performance will decrease, but we hope that in choosing a different subset each iteration, the performance will increase to levels similar to those we would see while using all data.

The reasoning for this last statement is that, although the algorithm only uses a subset of the data each iteration, by the time the algorithm finishes all data should've been used. Also, it is known that sampling improves generalization, thus we hope that this approx will reduce the gap between test and train error compared to the version with all data.

5.3.2 Pseudocode formalization

Algorithm 3: SVM-RFE with Random Sampling

Input: t, k // t = step, k = number of samples, $0 < k \leq |X_0|$
Output: $\vec{r}$
Data: $X_0, \vec{y}$

$\vec{s} = [1, 2, \ldots, m]$ // subset of surviving features
$\vec{r} = []$ // feature ranked list
while $|\vec{s}| > 0$ **do**
  /* Determine subset of examples by random sampling */
  $idx =$ `random_sample`$(|X_0|, k)$
  /* Restrict subset of examples to good feature indices */
  $X = X_0(idx, \vec{s})$
  /* Train the classifier */
  $\vec{\alpha} =$ `SVM-train`(X, y)
  /* Compute the weight vector of dimension length $|\vec{s}|$ */
  $\vec{w} = \sum_k \vec{\alpha}_k \vec{y}_k \vec{x}_k$
  /* Compute the ranking criteria */
  $\vec{c} = [(w_i)^2$ for all $i]$
  /* Find the t features with the smallest ranking criterion */
  $\vec{f} =$ `argsort`$(\vec{c})(:t)$
  /* Iterate over the feature subset */
  for $f_i \in \vec{f}$ **do**
    /* Update the feature ranking list */
    $\vec{r} = [\vec{s}(f_i), ...\vec{r}]$
    /* Eliminate the feature selected */
    $\vec{s} = [...\vec{s}(1 : f_i - 1), ...\vec{s}(f_i + 1 : |\vec{s}|)]$
  end
end

5.3.3 Results

Analysis with artificially generated data

We generate a 2 class dataset with the following code. The scalarization trade-off used is 80% accuracy, 20% feature subset size. Are results are mean values extracted form a 20-fold cross-validation procedure. For this dataset we've used $C = 10$ across all experiments.

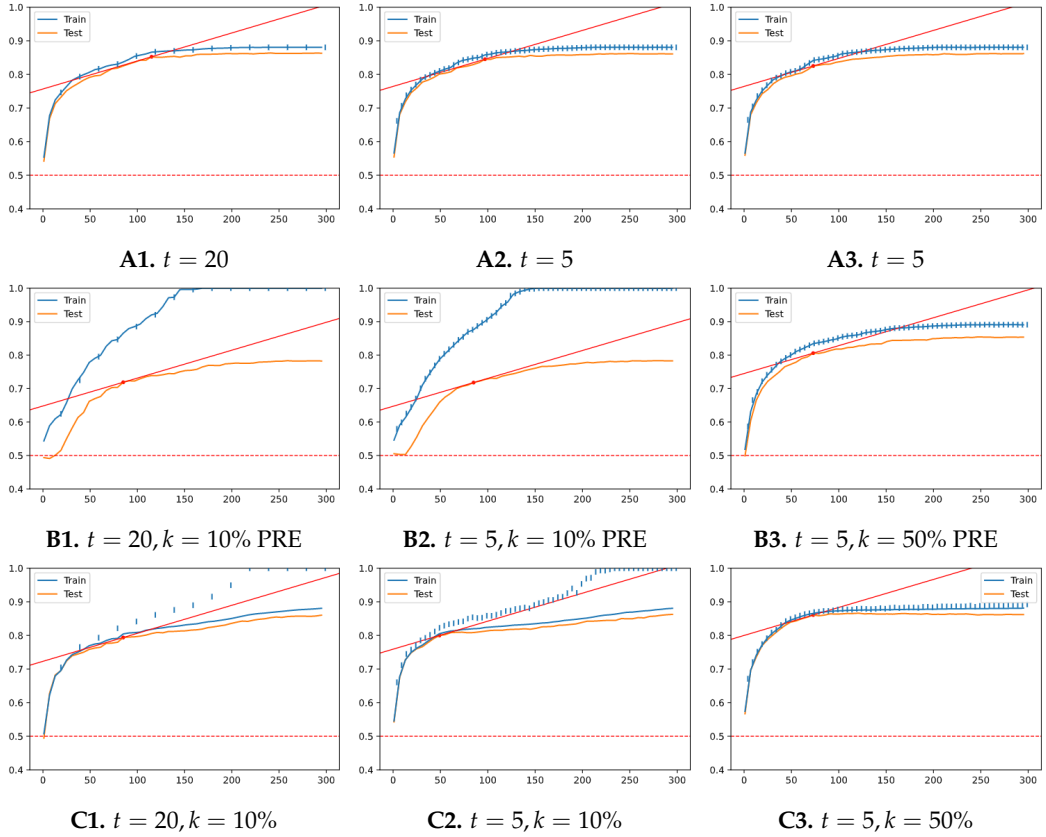
```
X, y = make_classification(
    n_samples = 8000, n_clusters_per_class = 2, n_features = 300,
    n_informative = 100, n_redundant=2, n_repeated=0,
    flip_y= 0.01, random_state=8, class_sep = 1.0
)
```

To test this extension we've created 3 algorithms:

- **A:** No sampling is done, all examples are used as usual.

- **B:** Sampling is done once, as a preprocessing step, not within SVM-RFE.
- **C:** Sampling is done within SVM-RFE, as explained in Section 5.3.1.

Figure 5.9 illustrates the results for 3 different parameter sets (3×3 grid).



A1. $t = 20$ **A2.** $t = 5$ **A3.** $t = 5$

B1. $t = 20, k = 10\%$ PRE **B2.** $t = 5, k = 10\%$ PRE **B3.** $t = 5, k = 50\%$ PRE

C1. $t = 20, k = 10\%$ **C2.** $t = 5, k = 10\%$ **C3.** $t = 5, k = 50\%$

FIGURE 5.9: Grid comparing algorithms on each row **A-B-C** and the hyper-parameter configuration for each **1** (step = 20), **2** (step = 5, sample = 10%) and **3** (step = 5, sample = 50%).

	1			**2**			**3**		
	Feat.	Acc.	Score	Feat.	Acc.	Score	Feat.	Acc.	Score
A	115	85.25%	0.194	97	84.51%	0.189	73	82.51%	0.189
B	85	71.88%	0.281	85	71.79%	0.282	73	80.58%	0.204
C	85	79.38%	0.221	49	79.99%	0.193	73	86.08%	0.160

TABLE 5.6: Perfomrance.

Note how algorithm **C** performs significatively better than **B** for all parameter sets, although it takes a bit more time. This extra time consumption can be attributed to the resampling procedure being performed each iteration of SVM-RFE versus only once. This will only be noticeable when the amount of samples is big. However, algorithm **C** is always faster than algorithm **A** and even performs better in one parameter set.

	1			**2**			**3**		
	Iter.	Time (s)	Var. (s^2)	Iter.	Time (s)	Var. (s^2)	Iter.	Time (s)	Var. (s^2)
A	15	07.82	0.26	60	30.35	7.68	60	30.87	2.21
B	15	01.46	0.16	60	05.29	1.86	60	16.26	4.70
C	15	03.18	2.14	60	13.43	5.51	60	19.64	1.75

TABLE 5.7: Times.

Analysis with artificially generated data 2

We've recycled the dataset generated for the dynamic step extension (Section 5.2.3) and repeated the experiment using the same parameters, including the 7-fold cross-validation and fixing a constant step of 10. The following tables summarize the result.

Percentage	**1.0**			**0.5**			**0.2**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.000010	37	89.50%	0.109	55	88.70%	0.127	43	84.10%	0.156
0.000100	37	89.50%	0.109	37	87.50%	0.125	43	82.20%	0.171
0.001000	85	88.10%	0.152	37	85.20%	0.143	37	79.11%	0.192
0.010000	103	85.30%	0.186	67	85.10%	0.164	13	75.99%	0.201
0.100000	85	84.00%	0.185	67	84.90%	0.165	19	78.00%	0.189

TABLE 5.8: Grid search of SVM-RFE with preprocessed sampling.

C/Percentage	**1.0**	**0.5**	**0.2**
0.000010	0:00.972	0:00.308	0:00.119
0.000100	0:01.249	0:00.422	0:00.166
0.001000	0:01.799	0:00.497	0:00.481
0.010000	0:02.604	0:02.106	0:02.492
0.100000	0:09.914	0:02.279	0:00.207

TABLE 5.9: Execution time (min:sec.msec) of SVM-RFE with preprocessed sampling.

Percentage	**0.5**			**0.2**			**0.1**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.000010	31	85.20%	0.139	37	86.20%	0.135	67	88.01%	0.141
0.000100	55	90.10%	0.116	43	89.70%	0.111	73	90.00%	0.129
0.001000	37	90.30%	0.102	49	89.90%	0.113	49	88.60%	0.124
0.010000	49	90.20%	0.111	49	89.00%	0.121	49	87.20%	0.135
0.100000	37	88.30%	0.118	55	86.70%	0.143	43	85.90%	0.141

TABLE 5.10: Grid search of SVM-RFE with sampling.

C/Percentage	0.5	0.2	0.1
0.000010	0:00.303	0:00.130	0:00.101
0.000100	0:00.315	0:00.129	0:00.097
0.001000	0:00.331	0:00.145	0:00.098
0.010000	0:00.376	0:00.166	0:00.114
0.100000	0:00.499	0:00.262	0:00.170

TABLE 5.11: Execution time of SVM-RFE with sampling.

We see similar results in this experiment, reaching both better performance and time even compared to the version without sampling (percentage of 100%). We note that the best regularization parameter C is kept the same when using preprocessed sampling, yet it is shifted when using sampling within RFE.

Analysis with Madelon

We've tested this extension with the Madelon dataset, but we've obtained similar results as in the previous extension. That is, there is no improvement in accuracy compared to the filter version. This makes using any kind of sampling non-practical given that if only one iteration is done then sampling outside or inside SVM-RFE are equivalent. Furthermore, outside sampling produces worse results.

5.3.4 Conclusions

This is a summary of the pros and cons we've found for this extension:

- It is always faster than using no sampling, but slower than using sampling as a preprocessing step.
- It generally performs much better than using a preprocessed sampling, and may even improve on the performance of the version with no sampling.
- Its performance depends on the amount of iterations, improving the more there are and being equivalent to preprocessed sampling for the filter case.

In general, we recommend using SVM-RFE with sampling whenever possible, specially to improve the speed of the algorithm.

5.4 Stop Condition

This modification intends to find an approximation to the optimal feature subset size within the SVM-RFE phase. This information allows using dynamic step more effectively. Using this extension we don't require a validation phase, which makes it very computationally efficient.

5.4.1 Description and reasoning

Relationship with *Dynamic Step*

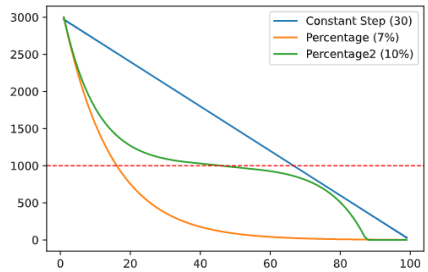
We had some previous success with the *dynamic step* extension. One problem with that extension, however, is that it functions better when the number of informative features is known. By default, *dynamic step* reduces the step size the fewer features

are left remaining. This generally improves performance since bigger steps (i.e. fewer iterations) are used in the more computationally expensive parts of the algorithm and smaller steps are used on a less expensive region.

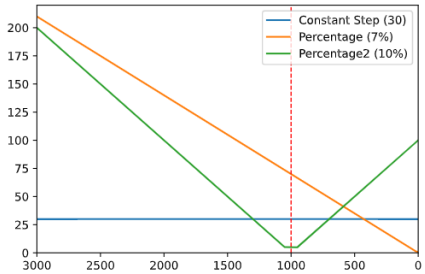
Because smaller steps also imply an improvement in accuracy, we would want to concentrate them on the most critical region of our feature subset space, that is, around the optimal feature subset size. We can easily modify our implementation of *Dynamic Step* so that it is centered at this point by applying our percentage on the distance to that point instead of the amount of remaining features. This time, since this point may be in a region that is more computationally expensive, we may also want to specify a minimum step parameter. Equation 5.3 combines these ideas. Where p is the percentage, min is the minimum step, c is the optimal feature subset size and i is the amount of remaining features.

$$t = \max(p|c - i|, min) \tag{5.3}$$

Figure 5.10 adds the new method to the dynamic step formulations described in the *Dynamic Step* extension (Figure 5.2). Note that for the left plot the slope of the curve corresponds with the step used in that iteration. The area under the curve is an approximation of the computational cost. For the left plot, the areas below the constant step line correspond to regions where, due to a smaller step, an increase in accuracy for that feature subset is possible. This, however, will only be useful in practice if that region is within the optimal feature subset size (red doted line).



Features remaining (vertical axis) at each iteration (horizontal axis).



Step (vertical axis) for each feature subset size (horizontal axis).

FIGURE 5.10: The new method "Percentage2" has a much smaller slope/step at the optimal feature subset size (1000).

Estimating the optimal feature subset size

There are multiple ways to estimate this point. Of course, running a whole cycle of SVM-RFE and validation (refer to figure 5.1) and calculating the optimal feature subset size is always an option. However, doing this previously to the SVM-RFE phase for every hyper-parameter set would be very expensive and completely defeat the purpose (since it would be more expensive than using constant step). Another option is to use the amount of informative features as an approximation (this is what we do in the *Combo* experiments), however this information is not always available.

If we're running a model selection process, it may be good enough to simply run a whole cycle once (with constant step), extract the optimal feature subset size for that hyper-parameter configuration and use it as an estimate for the rest. This, however, requires to make an initial run while guessing the hyper-parmaters, and it needs to perform reasonably well.

We may have better luck by doing an initial grid search under simplified datasets (such as with sampling or a big constant step on SVM-RFE). This, however, may still take a considerable amount of time (due to validation requiring a small step) and the results can still not extrapolate well to the full version (since an extreme reduction in sample size may induce shifts on the accuracy distribution of feature subsets).

Approximating accuracy by feature importance

The idea is to approximate the optimal feature subset size based on information we already have within SVM-RFE. Specifically, we want to use the feature importance calculated with the ranking criteria as an estimator of accuracy. This has the advantage that the whole validation phase can be skipped while hopefully still retaining a good approximation. It is also important that the same scalarization trade-off can be used here as well.

We assume that the importance of a feature is directly correlated with the error that is generated when the feature is removed. The total error generated for some feature subset can then be calculated as the commutative sum of the importances for all removed features up to that point. We then scale this result to be between 0.5 and 1.0.

Instead of scaling between two fix values we can get a better approximation by making use of RFE iterations. Each iteration we perform a small testing round (only with training data) to get a true training accuracy. We can then scale our approximation within the bounds of these values. Note that we could also linearly interpolate between this known values and get similar results, without a validation phase. This would however require small values for the step. The beauty of this method is that it still works even when the step used is large (or even equal to the amount of features).

5.4.2 Pseudocode formalization

Algorithm 4: SVM-RFE with Stop Condition

Input: t, t_0 // t = step, t_0 = threshold, $0 \leq t_0$
Output: $\vec{r}, \vec{q}$
Data: $X_0, \vec{y}$

$\vec{s} = [1, 2, \ldots, m]$ // subset of surviving features
$\vec{r} = []$ // feature ranking list
$\vec{q} = []$ // accuracy estimators
while $|\vec{s}| > 0$ **do**
  /* Restrict training examples to good feature indices */
  $X = X_0(:, \vec{s})$
  /* Train the classifier */
  $\vec{\alpha} = \texttt{SVM-train}(X, y)$
  /* Compute the weight vector of dimension length $|\vec{s}|$ */
  $\vec{w} = \sum_k \vec{\alpha}_k \vec{y}_k \vec{x}_k$
  /* Compute the ranking criteria */
  $\vec{c} = [(w_i)^2 \text{ for all } i]$
  /* Find the t features with the smallest ranking criterion */
  $\vec{f} = \texttt{argsort}(\vec{c})(:t)$
  /* Iterate over the feature subset */
  for $f_i \in \vec{f}$ **do**
    /* Append importance of the feature to be eliminated */
    $\vec{q} = [\vec{c}(f_i), ...\vec{q}]$
    /* Update the feature ranking list */
    $\vec{r} = [\vec{s}(f_i), ...\vec{r}]$
    /* Eliminate the feature selected */
    $\vec{s} = [...\vec{s}(1 : f_i - 1), ...\vec{s}(f_i + 1 : |\vec{s}|)]$
  end
end

Algorithm 5: Stop Point Extraction

Input: $\vec{q}, min, max, w_1, w_2$ // $\vec{q}$ = importance estimators
// w_0, w_1 = scalarization coef.
Output: $\arg\min \vec{c}$

$\vec{c} = []$ // list of cost of every feature subset
$\vec{s}$ s.t. $s_i = \sum_{i=1}^{k} q_k$
for $e_i \in \vec{s}$ **do**
  /* Estimate the accuracy from the cumulative importance */
  $\texttt{Acc} = (1 - e/\max(\vec{s}))/(max - min) + min$
  /* Scalarization */
  $c_i = w_1(1 - \texttt{Acc}) + w_2(i/|\vec{s}|)$
end

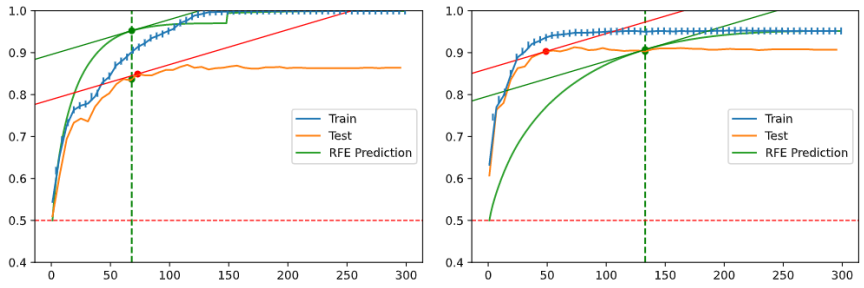
5.4.3 Results

Analysis with artificially generated data

We reuse the 2 class dataset from the *Dynamic Step* extension. The scalarization trade-off used is 80% accuracy, 20% feature subset size. All results are mean values extracted form a 7-fold cross-validation procedure.

```
X, y = make_classification(
    n_samples = 1000, n_clusters_per_class = 3, n_features = 300,
    n_informative = 100, n_redundant=100, n_repeated=20,
    flip_y= 0.05, random_state=2, class_sep = 2.0
)
```

Plot 5.11 illustrates our algorithm. The green line is our (train) accuracy estimate. We estimate the optimal feature subset size (green dot) and compare it to the true optimal (red dot) found by doing a full SVM-RFE and validation cycle using a much smaller step.



Step = 150. Good. Step = 300. Bad.

FIGURE 5.11: Accuracy estimation (green) for feature ranking generated with SVM-RFE with a step of 10.

Step		**300**			**150**			**60**	
C	Opt.	Est.	Diff.	Opt.	Est.	Diff.	Opt.	Est.	Diff.
0.00001	43	104	61	31	105	74	43	58	15
0.00010	49	133	84	37	108	71	55	50	5
0.00100	97	123	26	73	93	20	67	80	13
0.01000	73	66	7	97	80	17	103	119	16
0.10000	73	60	13	79	71	8	73	119	46

TABLE 5.12: Grid search of SVM-RFE with multi. The cost includes standard error.

Step	**300**			**150**			**60**		
C	Opt.	Est.	Diff.	Opt.	Est.	Diff.	Opt.	Est.	Diff.
0.00001	14.3%	34.7%	20.3%	10.3%	35.0%	24.7%	14.3%	19.3%	5.0%
0.00010	16.3%	44.3%	28.0%	12.3%	36.0%	23.7%	18.3%	16.7%	1.7%
0.00100	32.3%	41.0%	8.7%	24.3%	31.0%	6.7%	22.3%	26.7%	4.3%
0.01000	24.3%	22.0%	2.3%	32.3%	26.7%	5.7%	34.3%	39.7%	5.3%
0.10000	24.3%	20.0%	4.3%	26.3%	23.7%	2.7%	24.3%	39.7%	15.3%

TABLE 5.13: Grid search of SVM-RFE with multi. The cost includes standard error.

	Estimate			**SVM-RFE (Step = 10)**		
***C*/Step.**	**1.0**	**0.5**	**0.2**	**1.0**	**0.5**	**0.2**
0.00001	0:00.095	0:00.098	0:00.211	0:02.276	0:02.070	0:02.222
0.00010	0:00.112	0:00.143	0:00.275	0:02.907	0:02.897	0:02.870
0.00100	0:00.118	0:00.173	0:00.350	0:03.069	0:03.156	0:03.329
0.01000	0:00.213	0:00.300	0:00.520	0:04.428	0:05.253	0:05.015
0.10000	0:01.068	0:01.173	0:02.380	0:16.667	0:20.553	0:21.317

TABLE 5.14: Execution time of SVM-RFE with non-linear kernels.

5.4.4 Conclusions

This is a summary of the pros and cons we've found for this extension:

- It often works. Even when we only do one or two iterations the approximation is often quite close (<10%) the true optimal feature subset size.
- It doesn't always work. Even when we do use five iterations we may still get quite bad (>10%) approximations for certain parameter configurations.
- It's fast. One run takes roughly only 1/20 of the total time a normal SVM-RFE run would take, and completely skips validation.

In general, we recommend using it when the amount of informative features is not known. We can integrate it with the model selection to automatically find what the optimal feature subset size is and use the extensions that require such information.

5.5 Multi-Class

In this section we extend SVM-RFE to the multi-class classification problem.

5.5.1 Description and reasoning

When it comes to extending SVM to handle a multi-class problem two common methods exists, OvR (One-vs-Rest) and OvO (One-vs-One). In both cases the idea is to divide the problem in a set of binary classification problems. Then we use a joint decision function that operates on the results of each pair of problems. Because we're

not really making any predictions during the SVM-RFE phase, we can not use this joint decision function. Instead, we must find a way to merge the feature rankings obtained from each problem, i.e. find a joint ranking criteria.

We know that the ranking criteria is an estimator of the importance of some feature for a given binary decision problem. It can be the case that a feature is very useful to distinguish between two classes but useless for the rest. In this case a joint ranking criteria formed by taking the mean, the median or the sum will result in poor selections. A better idea would be to take the maximum. However, it may also be desirable to estimate the joint importance a feature has. For that we may want to consider its individual importance in more than one problem. That is, a feature that is important in more than one pair is more relevant than another that is only important in one such pair, even when the second importance value is greater. A way to perform such ranking would be, for instance, the sum of the squares.

Note that `sklearn` only supports OvO, and is therefore the option we will use.

5.5.2 Pseudocode formalization

Algorithm 6: SVM-RFE for multi-class classification problems

Input: t, k // t = step, k = degree
Output: $\vec{r}$
Data: $\chi_0, \vec{\psi}$
$\vec{s} = [1, 2, \ldots, m]$ // subset of surviving features
$\vec{r} = []$ // feature ranking list
while $|\vec{s}| > 0$ **do**
  /* Restrict training examples to good feature indices */
  $\chi = \chi_0(:, \vec{s})$
  /* Compute the joint ranking criteria */
  $C = [[], ...]$
  for $X \subseteq \chi$, $\vec{y} \subseteq \vec{\psi}$ *with* X *and* $\vec{y}$ *being an instance of OvO* **do**
    /* Train the classifier */
    $\vec{\alpha} =$ `SVM-train`$(X, \vec{y})$
    /* Compute the weight vector of dimension length $|\vec{s}|$ */
    $\vec{w} = \sum_k \vec{\alpha_k} \vec{y_k} X_k$
    /* Append feature importance list */
    $C = [...C, [(w_i)^2 \text{ for all } i]]$
  end
  /* Join feature importances */
  $\vec{c} = \sum_j C^k_{i,j}$ for all i
  /* Find the t features with the smallest ranking criterion */
  $\vec{f} =$ `argsort`$(\vec{c})(:t)$
  /* Iterate over the feature subset */
  for $f_i \in \vec{f}$ **do**
    /* Update the feature ranking list */
    $\vec{r} = [\vec{s}(f_i), ...\vec{r}]$
    /* Eliminate the feature selected */
    $\vec{s} = [...\vec{s}(1 : f_i - 1), ...\vec{s}(f_i + 1 : |\vec{s}|)]$
  end
end

Note that although now the SVM taring is done within an extra loop, each instance is only a sample of the original. The total amount of samples sum to n and thus the complexity remains roughly the same.

5.5.3 Results

Artificially generated data

We generate a 10 class dataset with the following code. The scalarization trade-off used is 80% accuracy, 20% feature subset size. All results are mean values extracted form a 7-fold cross-validation procedure.

```
X, y = make_classification(
```

```
    n_classes=10,
    n_samples = 4000, n_clusters_per_class = 1, n_features = 100,
    n_informative = 20, random_state=8, flip_y= 0.01
)
```

As usual when it comes to a new dataset, we start by plotting the accuracy of a random ranking. (Figure 5.12).

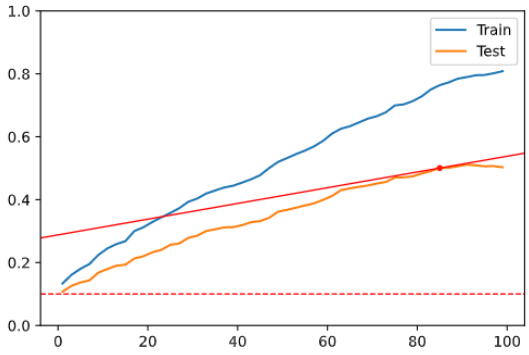


FIGURE 5.12: Random

We have designed four configurations that we will use for model selection. Three of them fix the parameter k in line 11 of algorithm 6. The fourth one replaces the whole line with the calculation of the coefficient of variance, which can be defined as $c_v = \frac{\sigma}{\mu}$. Note that for the case where $k = 1.0$ our implementation matches that of `sklearn`, and for values of K approaching infinity this formulation is equivalent to calculating the maximum.

Figure 5.13 illustrates the performances of the four experiments. Table 5.15 holds the specific values for comparison. Note how the best parameter for k is not always 1.0 but shifts to this value as C decreases.

For this experiment we've calculated the standard error of the cost. In this thesis the cost is usually calculated over the mean of the accuracies and only one value is obtained. In order to provide a standard error, this time we've calculated the cost independently and performed the mean of the costs. Note that due to scalarization both results are not equivalent.

k		**0.3**			**1.0**			**3.0**	
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.05	21	62.50%	0.324 ± 0.009	19	62.40%	0.316 ± 0.009	19	62.60%	0.322 ± 0.010
0.10	21	61.81%	0.329 ± 0.011	21	62.48%	0.321 ± 0.012	23	62.50%	0.320 ± 0.012
0.20	21	62.49%	0.328 ± 0.009	23	61.89%	0.327 ± 0.013	23	60.02%	0.347 ± 0.010
0.50	23	59.48%	0.336 ± 0.011	25	58.92%	0.345 ± 0.012	29	53.59%	0.390 ± 0.008
1.00	23	59.22%	0.345 ± 0.013	35	60.39%	0.367 ± 0.009	33	53.38%	0.395 ± 0.011
2.00	29	59.40%	0.342 ± 0.013	37	58.81%	0.381 ± 0.012	43	56.48%	0.402 ± 0.012

TABLE 5.15: Grid search of SVM-RFE with multi. The cost includes standard error.

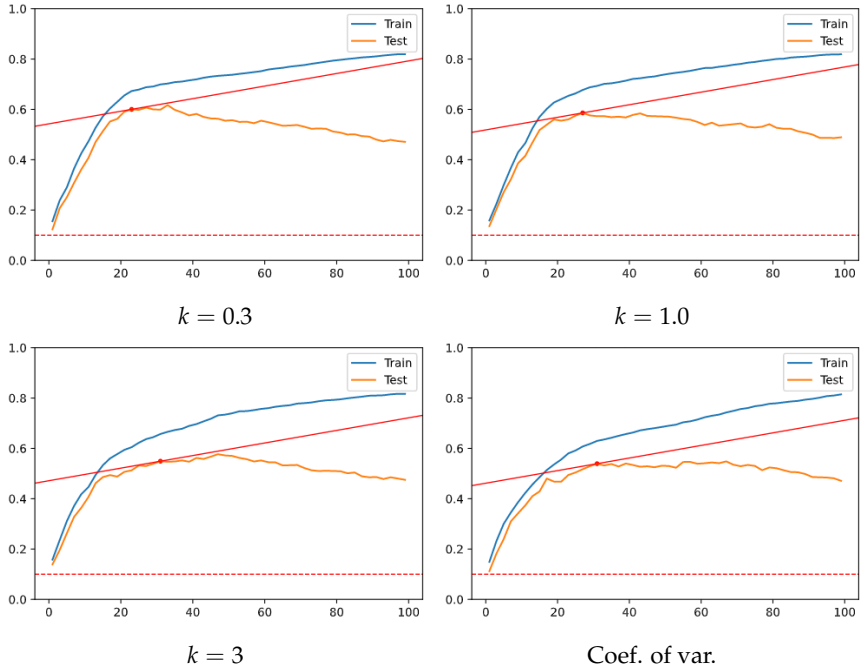


FIGURE 5.13: SVM-RFE ranking performance with C=0.5 for the multi-class problem.

Digits dataset

We start with the accuracy of a random ranking (Figure 5.14) to use as a baseline. Then proceed to plot the same four configurations as in the last experiment (Figure 5.15).

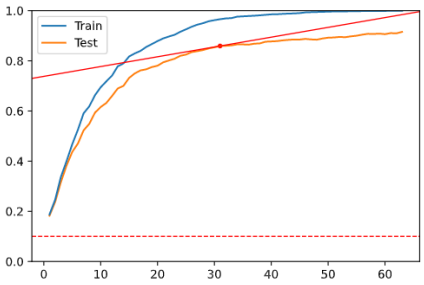


FIGURE 5.14: Random

Table 5.16 summarizes the results. We observe that there is not enough difference between cases $k = 0.3$ and $k = 1.0$ to determine if one is better than the other with certainty.

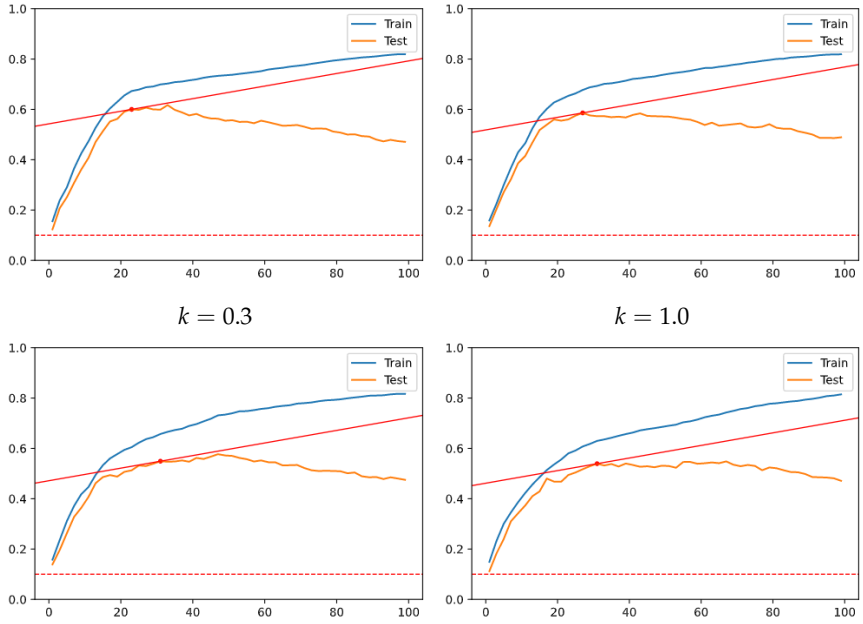


FIGURE 5.15: SVM-RFE ranking performance with C=0.5 for the digits problem.

k	**0.3**			**1.0**			**3.0**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.02	17	93.17%	0.097 ± 0.005	17	93.30%	0.096 ± 0.004	19	93.71%	0.095 ± 0.004
0.05	15	91.18%	0.101 ± 0.005	19	93.24%	0.103 ± 0.004	19	93.29%	0.102 ± 0.006
0.10	17	92.18%	0.102 ± 0.005	19	94.29%	0.094 ± 0.003	19	93.65%	0.099 ± 0.003
0.20	19	93.82%	0.100 ± 0.004	17	92.87%	0.098 ± 0.004	19	93.00%	0.100 ± 0.003
0.50	19	94.29%	0.098 ± 0.004	17	92.41%	0.099 ± 0.005	19	92.83%	0.103 ± 0.005
1.00	17	92.88%	0.099 ± 0.003	21	94.48%	0.098 ± 0.003	19	92.88%	0.105 ± 0.004
2.00	17	92.59%	0.102 ± 0.003	19	93.29%	0.103 ± 0.003	19	93.47%	0.101 ± 0.004

TABLE 5.16: Grid search of SVM-RFE with multi. The cost includes standard error.

5.5.4 Conclusions

This is a summary of the results for this extension:

- Multi-class can be easily implemented in SVM-RFE by using a *sum*.
- Using the coefficient of variance produces consistently worse results than the *sum*.
- Using a degree $k \neq 1.0$ may reduce the error when the value of C is big, but the results are inconclusive.
- Using a *max* results in worse performance.

In general, we recommend using a simple *sum*. Try $k \neq 1.0$ may be a good option when $C \geq 0.5$, but it is not recommended as a first try.

5.6 Non-linear Kernels

This extension intends to apply the required modifications in the calculation of the ranking criteria so that non-linear kernels can be used in the SVM.

5.6.1 Description and reasoning

When a problem is not linearly separable, we know that a hard-margin SVM will not be able to correctly place a decision boundary. In this case a soft-margin SVM may be used, but it only works to some extent and if the underlying distribution is near linearly separable. If it is not the case, much better results can be achieved by using non-linear kernels.

To use this method within SVM-RFE we must first be able to compute the ranking coefficient from a non-linear kernel. In contrast with the linear kernel case, where the ranking coefficient can be simplified to $(w_i)^2$, for non-linear kernels, however, since it is a more general case, no simplification can be performed. Instead, we use the general ranking coefficient for SVM (Equation 4.4.1), which we restate here:

$$DJ(i) = (1/2)(\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}\boldsymbol{\alpha} - \boldsymbol{\alpha}^{\mathrm{T}}\mathbf{H}(-i)\boldsymbol{\alpha})$$

Note that the *hessian* matrix $\mathbf{H_{i,j}} = y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$ needs be computed each iteration (since the dimension of $\mathbf{x_i}$ and $\mathbf{x_j}$ will change), and also for each feature removed in each iteration. This is slow. Specifically, since we do have to do an extra loop within RFE for each feature, and require at least $O(mn^2)$ for calculating the Hessian, we know that the complexity will be of $O(m^3n^2)$. However, various optimizations exist, as discussed in the last parts of Section 4.4.1. The optimizations we are going to implement are:

Updating the hessian matrix only for the support vectors

We know that an observation i that is not a support vector will cause $\alpha_i = 0$. These values are then multiplied by the hessian matrix, but because they are zero they make no contribution to the overall importance calculation. Since we know that they will make no contribution anyway we can also skip recomputing the rows and columns on the Hessian matrix that we know will get multiplied by non-support vectors.

Using a sparse Hessian matrix that only includes support vectors would also work and have the additional benefit of taking less memory to store. However, due to limitations in the `precomputed` mode of `sklearn` SVC implementation, we can not do that.

This optimization does not change the overall complexity but may still cause a significative reduction on time in cases where C is small and few support vectors are chosen.

Caching previous results

Depending on the kernel function we may be able to precompute some parts of it if all we want to do is update it for the case where one feature is removed.

Specifically for polynomial kernels we are going to use the following property, which allows us to precompute $\langle \mathbf{a}, \mathbf{b} \rangle$:

$$\langle \mathbf{a}(-i), \mathbf{b}(-i) \rangle = \langle \mathbf{a}, \mathbf{b} \rangle - \mathbf{a}_i \mathbf{b}_i$$

And for Gaussian kernels we are going to use the following property, for which we instead need to precompute $\langle \mathbf{a} - \mathbf{b}, \mathbf{a} - \mathbf{b} \rangle$:

$$\begin{aligned} ||\mathbf{a}(-i) - \mathbf{b}(-i)||^2 &= \langle \mathbf{a}(-i) - \mathbf{b}(-i), \mathbf{a}(-i) - \mathbf{b}(-i) \rangle \\ &= \langle \mathbf{a} - \mathbf{b}, \mathbf{a} - \mathbf{b} \rangle - (\mathbf{a}_i - \mathbf{b}_j)^2 \end{aligned}$$

This optimization allows us to update the Hessian matrix where only one feature is removed in $O(n^2)$, therefore the overall complexity of SVM-RFE becomes $O(m^2 n^2)$.

5.6.2 Pseudocode formalization

Algorithm 7: SVM-RFE with general Kernel

Input: t, k `// t = step, k = kernel function`
Output: $\vec{r}$
Data: $X_0, \vec{y}$

$\vec{s} = [1, 2, \ldots, m]$ `// subset of surviving features`
$\vec{r} = []$ `// feature ranking list`
while $|\vec{s}| > 0$ **do**
  `/* Restrict training examples to good feature indices */`
  $X = X_0(:, \vec{s})$
  `/* Precompute hessian matrix */`
  $\mathbf{H_{i,j}} = y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$ for all $\mathbf{x_i}, \mathbf{x_j} \in X$
  `/* Train the classifier */`
  $\vec{\alpha} =$ `SVM-train`(X, y, k)
  `/* Compute the ranking criteria */`
  $\vec{c} = [c_1, c_2, \ldots, c_{|\vec{s}|}]$
  for $c_l \in \vec{c}$ **do**
    `/* Compute new hessian with the feature` l `removed */`
    $\mathbf{H_{i,j}}(-l) = y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$ for all $\mathbf{x_i}, \mathbf{x_j} \in X(-l)$
    `/* Calculate ranking coefficient */`
    $c_l = (1/2)(\boldsymbol{\alpha}^{\mathrm{T}} \mathbf{H} \boldsymbol{\alpha} - \boldsymbol{\alpha}^{\mathrm{T}} \mathbf{H}(-l) \boldsymbol{\alpha})$
  end
  `/* Find the` t `features with the smallest ranking criterion */`
  $\vec{f} =$ `argsort`$(\vec{c})(:t)$
  `/* Iterate over the feature subset */`
  for $f_i \in \vec{f}$ **do**
    `/* Update the feature ranking list */`
    $\vec{r} = [\vec{s}(f_i), ...\vec{r}]$
    `/* Eliminate the feature selected */`
    $\vec{s} = [...\vec{s}(1 : f_i - 1), ...\vec{s}(f_i + 1 : |\vec{s}|)]$
  end
end

5.6.3 Results

Notes on the implementation

Using non-linear kernels requires a change in the underlying implementation we've been using for SVM. Until now, we've been relying on the `LinearSVC` class provided by `Sklearn`, this implementation is in turn based on the `LIBLINEAR` implementation for SVM written by the *National Taiwan University*. The authors state that this solver is much faster than the more general version, thus the reason we've been using it, but it can only handle linear kernels. The general version, `LIBSVM`, created by the same team, is the alternative we're going to use instead. Because now we need to switch, it is interesting to see what will be the increase in computational cost, shown in the table 5.17.

For this first test we've generated artificial datasets with 100 features each, 20 which are informative. Because we want to test the differences with the linear kernel we've used 6 clusters per class, this makes the problem more difficult for lineal separators.

The following table makes a time comparison using different amount of samples. These are the elapsed times that take completing a validation round using a random ranking. This is similar to RFE without the ranking criteria computations. We've not used SVM-RFE here because we want to demonstrate how the change on the underlying implementation, and not the change in our algorithm, is already a cause for slow down.

Obs.	`LIBLINEAR`	`LIBSVM` *Linear*	`LIBSVM` *Precomputed*
500	02.67s	08.17s — x0.32	12.04s — x0.22
1000	02.94s	17.30s — x0.17	23.84s — x0.12
2000	04.28s	43.96s — x0.10	50.36s — x0.08

TABLE 5.17: Cost in time and speedup of a random selection under different implementations and sample sizes.

Note that the sample size increases the cost more than linearly, this may suggest that methods such as dynamic sampling can perform even better than with the `LIBLINEAR` implementation.

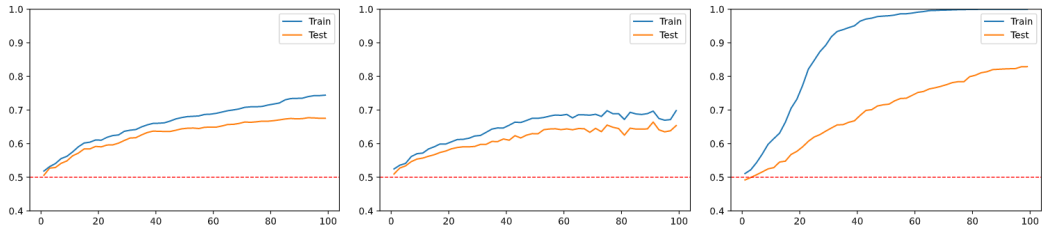
We also note that passing a precomputed kernel matrix (*Gramm* matrix) instead of letting `sklearn` compute it internally, also increases the time. This is likely an implementation detail, and it is not significant enough to require further investigation.

We now plot some examples to compare differences in accuracy (Figure 5.16). Note that the precomputed kernel, which is a degree-3 polynomial kernel, performs much better at the start. Thus, we can expect it to also perform significantly better with SVM-RFE.

We can appreciate some differences between the two linear kernels, but they're still clearly following the same curve, which indicates that both implementations are equivalent.

Analysis with artificially generated data

We generate a 2 class dataset with the following code. The scalarization trade-off used is 80% accuracy, 20% feature subset size. All results are mean values extracted



`LIBLINEAR` `LIBSVM` *Linear* `LIBSVM` *Precomputed*

FIGURE 5.16: Comparison of random selection with 2000 observations.

form a 7-fold cross-validation procedure. We've used a constant step of 10 for SVM-RFE and X for validation.

```
X, y = make_classification(
    n_samples = 500, n_clusters_per_class = 10, n_features = 300,
    n_informative = 100, n_redundant = 100, n_repeated = 0,
    random_state=2, flip_y= 0.01, class_sep = 3
)
```

As usual, we start by plotting the accuracy performance for a random ranking. Note that for this plot the validation is still a SVM with a linear kernel.

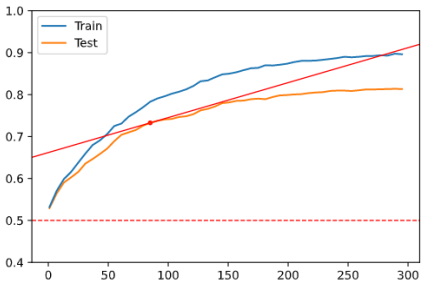
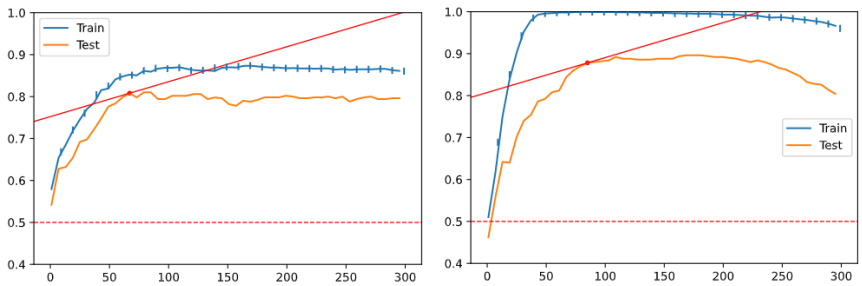


FIGURE 5.17: Random

We observe experimentally in figure 5.18 that, for this dataset, using a non-lineal kernel improves the accuracy of SVM-RFE. Table 5.18 summarizes the results from the model selection phase.



Linear. $C = 0.001$ Poly-2. $C = 10^{-6}$

FIGURE 5.18: Accuracy of SVM-RFE feature rankings with the best found regularization coefficient C for linear and polynomial kernels.

Non-lineal kernels also involve more hyper-parameters, thus increasing the importance of model selection. The choice of kernel is in itself a hyper-parameter. If a polynomial kernel is chosen then the *degree* and *coefficient* are also hyper-parameters, if a Gaussian kernel is chosen then *gamma*.

deg.	**1**			**2**			**3**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
1e-10	1	42.40%	0.461	1	46.60%	0.428	55	66.73%	0.303
1e-09	1	43.41%	0.453	1	48.80%	0.410	127	79.99%	0.245
1e-08	13	54.15%	0.375	1	55.40%	0.357	115	84.39%	0.202
1e-07	1	47.20%	0.423	67	82.21%	0.187	97	79.20%	0.231
1e-06	1	47.41%	0.421	85	87.81%	0.154	127	83.00%	0.221
1e-05	7	61.59%	0.312	79	86.82%	0.158	145	84.98%	0.217
1e-04	43	76.22%	0.219	79	86.19%	0.163	151	85.19%	0.219
1e-03	67	80.81%	0.198	73	85.18%	0.167	109	81.98%	0.217
1e-02	55	79.58%	0.200	85	87.60%	0.156	175	87.59%	0.216
1e-01	85	82.39%	0.198	73	86.18%	0.159	109	83.39%	0.206

TABLE 5.18: Grid search of SVM-RFE with polynomial kernel.

	Optimized			**Not optimized**		
C/**Deg.**	**1**	**2**	**3**	**1**	**2**	**3**
1e-10	0:19.509	0:21.968	0:22.364	0:47.609	0:48.374	1:16.220
1e-09	0:21.554	0:22.573	0:22.636	0:51.182	0:48.316	1:13.988
1e-08	0:21.617	0:22.506	0:22.221	0:49.338	0:53.185	1:15.945
1e-07	0:21.705	0:22.073	0:21.914	0:49.022	0:52.526	1:08.661
1e-06	0:21.588	0:20.976	0:21.411	0:51.579	0:54.387	1:07.181
1e-05	0:21.426	0:18.521	0:21.937	0:53.271	0:53.834	1:06.723
1e-04	0:21.513	0:19.455	0:21.622	0:49.130	0:50.594	1:07.257
1e-03	0:16.416	0:19.160	0:22.190	0:50.915	0:50.858	1:06.532
1e-02	0:13.730	0:19.116	0:21.048	0:48.972	0:48.041	1:06.149
1e-01	0:14.009	0:18.330	0:21.039	0:52.361	0:47.305	1:06.022

TABLE 5.19: Execution time of SVM-RFE with non-linear kernels.

Table 5.19 allows us to get an idea of what is the reduction in time we can get with the optimizations described in section 5.6.1. With this information we calculate a speed-up of roughly x3. For the implementation of the optimizations we've used the `numba` library. This library seamlessly compiles Python functions and allows our optimized Python implementation to compete in speed with the low-level, but not optimized, implementation used by `sklearn`. We suspect that this automatic Python compilation is not as good as a low-level implementation, therefore it is expected that a low-level implementation of SVM-RFE would be able to produce even more time reduction.

Analysis with MADELON

The scalarization trade-off used is 80% accuracy, 20% feature subset size. All results are mean values extracted form a 7-fold cross-validation procedure. We're using a

constant step of 20 for SVM-RFE and 2 for the validation phase. We've normalized the data as a preprocessing step.

Figure 5.19 compares the accuracy performance when a non-linear kernel is used within SVM-RFE versus only in the validation phase. For a comparison with a linear SVM validation phase go back to Figure 5.5.

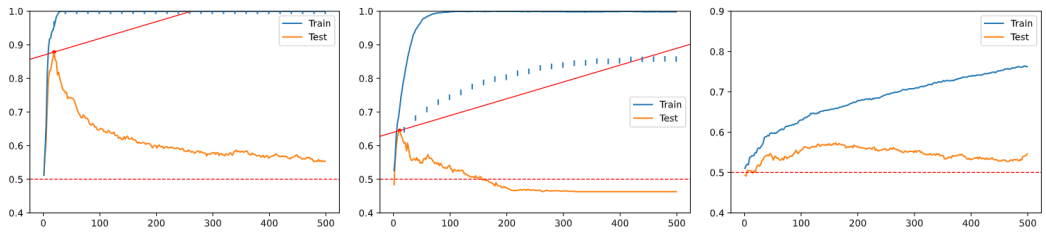


FIGURE 5.19: Accuracy of SVM-RFE feature rankings with the best found regularization coefficient C for linear and poly-7 kernels. Non-linear validation phases consist of SVM with a Poly-7 kernel.

Tables 5.20 and 5.21 describe the model selection phase for polynomial kernels.

5.6.4 Conclusions

This is a summary of the results for this extension:

- For datasets that are fundamentally not linearly separable, using non-linear kernels produces a huge improvement in accuracy.
- Using non-linear kernels is slow, it makes the ranking criteria calculation require an extra loop and become the bottleneck.
- Optimizations improve the cost of the ranking criteria calculation.
- Calculations could be potentially made even faster by using a compiled implementation.

In general, we recommend using non-linear kernels whenever there is suspicion of the data not being linearly separable. If used, we always recommend implementing the optimizations described in this section.

5.7 Combo

This extension is a combination of the *Non-linear Kernel*, the *Dynamic Step* and the *Sampling* extensions. Because it is straightforward to combine them, we do not provide here the pseudocode.

5.7.1 Description and reasoning

The three extensions have been selected on the hypothesis that they do not interfere with each other, given that they optimize different parts of the algorithm. We've not included the *Multi-class* extension here because we wanted to test the combination specifically on the Madelon dataset. The *Stop Condition* extension is also not include because it optimizes the validation phase, rather than SVM-RFE itself.

One possible incompatibility of the *Sampling* combined with *Dynamic Step* is that the extra time required to resample is increased on the later iterations due to the shrinking of the step. To avoid this, and other similar inconvenient we've slightly modified the *Dynamic Step* implementation with an extra parameter *dstop*. These parameters design a feature subset size that will be the target of dynamic step (instead of 0), and after which point no resampling will be done (the whole dataset will be used).

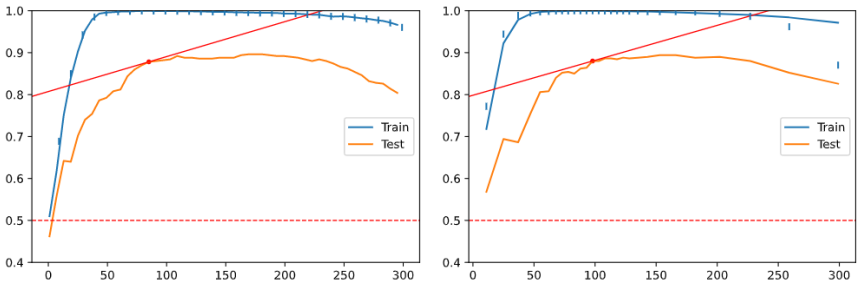
5.7.2 Results

Analysis with artificially generated data

We reuse the artificial dataset generated in section 5.6.3, as a remainder, this is a 2 class dataset made with the following code. The scalarization trade-off used is 80% accuracy, 20% feature subset size. All results are mean values extracted form a 7-fold cross-validation procedure.

```
X, y = make_classification(
    n_samples = 500, n_clusters_per_class = 10, n_features = 300,
    n_informative = 100, n_redundant = 100, n_repeated = 0,
    random_state=2, flip_y= 0.01, class_sep = 3
)
```

For this experiment we've used a dynamic reflective step with a percentage of 20%, a minimum step of 5 and centered at 100 features. We've also used internal sampling of 70%. Figure 5.20 compares two rankings with the same parameter set, the left one corresponds to a result from the *Non-linear kernel* extension, the right one from this extension.



Poly-2. $C = 10^{-6}$ Poly-2. $C = 10^{-6}$

FIGURE 5.20: Performance comparison of SVM-RFE feature rankings. Both show similar accuracy, but the one in the right is twice as fast.

Tables 5.22 and 5.23 summarize the results from this experiment.

deg.	**1**			**2**			**3**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
1e-10	11	45.01%	0.447	11	46.81%	0.433	47	70.63%	0.266
1e-09	11	44.40%	0.452	11	47.00%	0.431	108	82.20%	0.214
1e-08	11	45.97%	0.440	37	52.16%	0.407	113	81.79%	0.221
1e-07	11	45.40%	0.444	83	82.41%	0.196	108	82.80%	0.210
1e-06	11	49.17%	0.414	98	88.01%	0.161	113	82.02%	0.219
1e-05	11	66.20%	0.278	88	86.79%	0.164	108	82.20%	0.214
1e-04	37	75.61%	0.220	88	86.01%	0.171	108	82.82%	0.209
1e-03	55	77.58%	0.216	83	86.59%	0.163	108	81.81%	0.218
1e-02	78	82.20%	0.194	83	88.60%	0.147	123	82.79%	0.220
1e-01	62	77.20%	0.224	88	88.41%	0.151	143	83.81%	0.225

TABLE 5.22: Grid search of SVM-RFE with polynomial kernel.

C/**Deg.**	**1**	**2**	**3**
1e-10	0:08.003	0:08.314	0:08.781
1e-09	0:07.922	0:08.231	0:08.670
1e-08	0:07.928	0:08.341	0:08.646
1e-07	0:08.075	0:08.506	0:08.523
1e-06	0:08.035	0:07.961	0:08.453
1e-05	0:08.198	0:07.834	0:08.525
1e-04	0:07.956	0:07.739	0:08.571
1e-03	0:06.849	0:07.851	0:08.461
1e-02	0:06.134	0:08.189	0:08.510
1e-01	0:05.541	0:08.097	0:08.582

TABLE 5.23: Execution time of SVM-RFE with non-linear kernels.

By comparing this results to Table 5.19 we calculate an approximate speed-up of x2 compared to the optimized non-linear kernel version.

We have also tried using a Gaussian kernel (Table 5.24). Although using it we can obtain grater values of accuracy, the amount of features it selects also increase significatively. Overall the RBF kernel doesn't perform as good as the polynomial for this dataset. Note that RBF kernel results for the *Non-lineal kernel* extension are the same and have been omitted. The time table has also been omitted.

γ	**0.00036**			**0.00046**			**0.00056**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
1e+00	161	88.41%	0.200	132	85.39%	0.205	102	79.82%	0.229
1e+01	157	91.39%	0.174	134	88.60%	0.181	104	83.01%	0.205
1e+02	145	89.81%	0.178	134	89.01%	0.177	110	86.99%	0.177
1e+03	157	91.59%	0.172	132	87.80%	0.186	102	85.79%	0.182
1e+04	149	89.22%	0.186	134	87.39%	0.190	106	81.59%	0.218
1e+05	161	90.01%	0.187	132	87.80%	0.186	106	82.20%	0.213

TABLE 5.24: Grid search of SVM-RFE with RBF kernel.

Analysis with MADELON

For this experiment we've used a dynamic reflective step with a percentage of 8%, a minimum step of 3 and centered at 20 features. We've also used internal sampling of 20%. Figure 5.21 corresponds to the feature ranking accuracy with the best parameters found in the model selection phase.

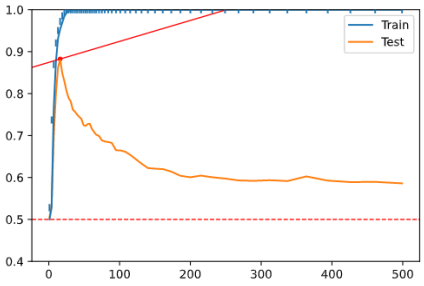
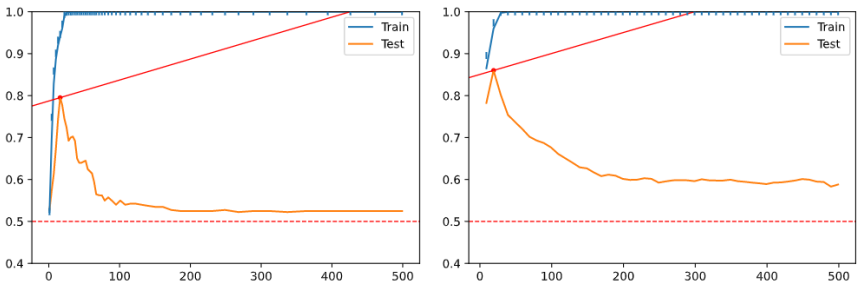


FIGURE 5.21: Poly-7, $C = 0.6$

Tables 5.25 and 5.26 summarize the results. We note that although we were not able to improve the accuracy compared with the version only with non-polynomial kernels, we've achieved a speed-up of x6.

The following figure (Figure 5.22) illustrates what happens if we remove either *Dynamic Step* or *Sampling* from our *Combo* extension. Removing internal sampling significatively reduces the accuracy. Using a constant step instead of dynamic also performs worse, either by increasing the elapsed time when a small step is used, or decreasing the accuracy when the value of the step is large. Although the accuracy reduced by using a large step (e.g. 50) is small (3% drop), it may produce an important speed up of the SVM-RFE phase. However, the validation phase still requires using dynamic step or a very small constant step to avoid the accuracy performance dropping considerably.



Preprocessed sampling of 20% Constant step of 10

FIGURE 5.22: Performance comparison of SVM-RFE (Combo) feature rankings with some extension removed. Poly-7, $C = 0.6$

We have also tried using a Gaussian kernel (Table 5.27). The results are not as good as using a polynomial, but it produces smaller feature subsets. The timetable has been omitted.

5.7.3 Conclusions

This is a summary of the results for this extension:

- As expected, this combo produces a substantial speed-up.

- Accuracy is not negatively effected by the new method.
- Finding the correct parameters for *Dynamic Step* and *Sampling* requires knowledge on the data.

Although this method requires extra parameters, these have well-defined ranges and are not very sensitive to changes. This makes it possible to find adequate values with a simple initial exploratory analysis without requiring more complex model selection algorithms.

In general, we recommend using this combo approach whenever the data is not linearly separable and the approximate amount of informative features is known. A previous exploration of the data set may be required.

deg.	**1**			**2**			**3**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	123	56.83%	0.349 ± 0.013	21	69.67%	0.242 ± 0.010	21	74.26%	0.211 ± 0.014
0.2	67	55.75%	0.344 ± 0.014	21	69.58%	0.241 ± 0.011	21	76.50%	0.192 ± 0.008
0.3	49	54.66%	0.334 ± 0.013	21	70.58%	0.238 ± 0.020	21	77.41%	0.183 ± 0.010
0.4	31	54.67%	0.343 ± 0.015	19	71.17%	0.231 ± 0.007	19	77.67%	0.175 ± 0.013
0.5	11	54.24%	0.338 ± 0.018	19	70.67%	0.231 ± 0.010	21	77.91%	0.183 ± 0.017
0.6	57	55.51%	0.346 ± 0.012	21	71.75%	0.225 ± 0.011	21	78.42%	0.170 ± 0.007
0.7	59	53.42%	0.355 ± 0.013	21	72.08%	0.225 ± 0.009	21	78.42%	0.174 ± 0.007
0.8	27	56.25%	0.338 ± 0.011	17	70.50%	0.226 ± 0.011	21	77.59%	0.177 ± 0.010
0.9	25	53.76%	0.356 ± 0.006	19	71.33%	0.223 ± 0.010	19	78.25%	0.178 ± 0.005
deg.	**4**			**5**			**6**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	21	74.25%	0.212 ± 0.013	19	74.34%	0.212 ± 0.013	19	71.00%	0.235 ± 0.011
0.2	21	80.91%	0.159 ± 0.006	19	81.16%	0.154 ± 0.006	19	79.75%	0.170 ± 0.014
0.3	19	82.75%	0.141 ± 0.010	19	84.66%	0.128 ± 0.011	21	83.08%	0.137 ± 0.014
0.4	19	83.67%	0.133 ± 0.004	21	86.42%	0.113 ± 0.007	19	85.41%	0.121 ± 0.007
0.5	21	84.17%	0.131 ± 0.010	21	86.25%	0.116 ± 0.011	19	86.49%	0.112 ± 0.008
0.6	19	84.08%	0.135 ± 0.012	19	85.67%	0.117 ± 0.009	19	86.76%	0.105 ± 0.009
0.7	19	84.50%	0.129 ± 0.004	19	86.67%	0.111 ± 0.009	19	87.00%	0.110 ± 0.010
0.8	19	84.33%	0.130 ± 0.007	17	85.50%	0.113 ± 0.005	19	86.75%	0.107 ± 0.006
0.9	19	84.84%	0.127 ± 0.008	19	85.76%	0.117 ± 0.010	19	86.58%	0.109 ± 0.003
deg.	**7**			**8**			**9**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	19	69.17%	0.240 ± 0.011	17	66.84%	0.258 ± 0.010	15	64.90%	0.261 ± 0.025
0.2	15	70.58%	0.218 ± 0.009	7	60.93%	0.295 ± 0.026	7	55.66%	0.332 ± 0.011
0.3	11	72.00%	0.205 ± 0.022	7	65.09%	0.265 ± 0.031	7	55.76%	0.321 ± 0.018
0.4	19	86.09%	0.116 ± 0.005	9	69.19%	0.232 ± 0.032	5	66.08%	0.262 ± 0.018
0.5	19	88.41%	0.100 ± 0.004	19	86.16%	0.113 ± 0.011	7	66.68%	0.240 ± 0.014
0.6	19	87.58%	0.106 ± 0.005	19	87.34%	0.104 ± 0.006	17	82.08%	0.129 ± 0.013
0.7	19	87.00%	0.104 ± 0.007	19	87.26%	0.107 ± 0.007	19	87.08%	0.107 ± 0.010
0.8	19	86.92%	0.106 ± 0.004	19	86.17%	0.109 ± 0.008	19	85.92%	0.113 ± 0.004
0.9	19	86.16%	0.111 ± 0.008	19	85.75%	0.110 ± 0.011	19	85.75%	0.117 ± 0.006

TABLE 5.20: Grid search of SVM-RFE with polynomial kernel.

C/deg.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0.1	2:06.42	2:05.38	2:06.95	2:12.24	2:13.10	2:14.76	2:11.08	2:17.51	2:17.33
0.2	2:07.59	2:04.58	2:06.57	2:12.31	2:12.88	2:12.93	2:12.17	2:17.10	2:24.43
0.3	1:55.10	2:04.76	2:07.29	2:11.48	2:12.93	2:12.88	2:11.95	2:16.58	2:17.76
0.4	1:55.69	2:01.94	2:07.51	2:13.60	2:12.26	2:13.43	2:10.84	2:16.53	2:17.77
0.5	1:52.74	2:01.71	2:07.26	2:11.87	2:13.73	2:13.98	2:11.35	2:17.51	2:17.31
0.6	1:52.05	2:01.22	2:06.24	2:11.54	2:12.87	2:13.46	2:10.76	2:17.58	2:16.56
0.7	1:51.70	1:59.91	2:05.55	2:11.61	2:12.28	2:13.21	2:11.43	2:16.34	2:16.19
0.8	1:50.38	1:57.42	2:05.32	2:12.67	2:12.39	2:15.10	2:09.23	2:17.35	2:18.27
0.9	1:48.50	1:58.06	2:03.74	2:11.86	2:11.66	2:12.94	2:11.02	2:17.79	2:16.37

TABLE 5.21: Execution time (min:sec) of SVM-RFE with Polynomial Kernel optimized.

deg.	**1**			**2**			**3**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	22	55.45%	0.335 ± 0.004	19	69.65%	0.245 ± 0.009	19	75.40%	0.200 ± 0.011
0.2	61	55.60%	0.345 ± 0.008	19	71.05%	0.235 ± 0.007	19	78.50%	0.177 ± 0.008
0.3	31	58.95%	0.323 ± 0.011	19	71.05%	0.228 ± 0.009	19	78.15%	0.178 ± 0.005
0.4	13	58.25%	0.319 ± 0.010	19	71.50%	0.232 ± 0.010	22	78.20%	0.177 ± 0.008
0.5	49	56.15%	0.342 ± 0.010	19	71.75%	0.222 ± 0.004	19	78.65%	0.176 ± 0.006
0.6	28	57.25%	0.334 ± 0.011	19	71.65%	0.227 ± 0.009	19	78.65%	0.174 ± 0.007
0.7	49	56.55%	0.336 ± 0.013	19	71.40%	0.228 ± 0.005	19	78.85%	0.174 ± 0.008
0.8	34	56.05%	0.346 ± 0.011	16	71.40%	0.229 ± 0.006	19	79.10%	0.172 ± 0.005
0.9	28	56.90%	0.342 ± 0.011	22	70.90%	0.234 ± 0.008	19	79.60%	0.168 ± 0.011
deg.	**4**			**5**			**6**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	19	77.24%	0.183 ± 0.014	16	73.60%	0.198 ± 0.021	16	77.54%	0.185 ± 0.016
0.2	19	83.90%	0.135 ± 0.005	19	84.15%	0.129 ± 0.009	16	79.55%	0.146 ± 0.010
0.3	19	84.25%	0.130 ± 0.005	19	85.55%	0.116 ± 0.008	16	81.00%	0.142 ± 0.017
0.4	19	83.75%	0.135 ± 0.005	19	86.85%	0.113 ± 0.006	19	86.50%	0.113 ± 0.008
0.5	19	84.45%	0.130 ± 0.004	19	86.75%	0.113 ± 0.007	16	85.35%	0.116 ± 0.006
0.6	19	84.15%	0.132 ± 0.005	19	86.90%	0.108 ± 0.008	19	87.80%	0.103 ± 0.004
0.7	19	84.65%	0.130 ± 0.005	19	86.65%	0.111 ± 0.005	19	86.45%	0.110 ± 0.005
0.8	19	84.35%	0.132 ± 0.010	19	86.85%	0.112 ± 0.007	19	87.20%	0.104 ± 0.006
0.9	22	85.10%	0.123 ± 0.005	19	86.65%	0.113 ± 0.006	19	86.30%	0.109 ± 0.004
deg.	**7**			**8**			**9**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	16	72.45%	0.189 ± 0.012	16	73.26%	0.190 ± 0.020	13	71.15%	0.223 ± 0.024
0.2	13	77.80%	0.174 ± 0.027	10	71.00%	0.206 ± 0.024	10	65.40%	0.258 ± 0.027
0.3	7	69.00%	0.219 ± 0.026	7	60.44%	0.280 ± 0.023	10	58.35%	0.318 ± 0.010
0.4	10	77.60%	0.176 ± 0.023	7	62.50%	0.276 ± 0.029	7	62.25%	0.284 ± 0.022
0.5	13	85.65%	0.117 ± 0.012	10	69.14%	0.212 ± 0.029	7	65.61%	0.258 ± 0.031
0.6	16	88.25%	0.098 ± 0.004	10	72.20%	0.195 ± 0.025	10	70.94%	0.214 ± 0.030
0.7	19	87.35%	0.105 ± 0.009	13	85.50%	0.116 ± 0.008	10	70.95%	0.215 ± 0.037
0.8	19	87.90%	0.104 ± 0.005	16	87.05%	0.102 ± 0.005	10	81.16%	0.135 ± 0.010
0.9	19	87.45%	0.106 ± 0.006	19	87.05%	0.105 ± 0.004	16	84.75%	0.115 ± 0.009

TABLE 5.25: Grid search of SVM-RFE (combo) with polynomial kernel.

C/deg.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0.1	0:20.24	0:20.80	0:20.75	0:24.10	0:21.58	0:21.40	0:21.68	0:21.64	0:20.16
0.2	0:20.66	0:20.78	0:20.65	0:24.01	0:21.97	0:21.31	0:21.44	0:22.25	0:20.15
0.3	0:21.37	0:20.61	0:20.76	0:22.29	0:21.52	0:21.93	0:21.98	0:22.16	0:21.79
0.4	0:19.71	0:21.18	0:20.76	0:21.30	0:22.64	0:21.73	0:21.62	0:21.83	0:22.00
0.5	0:19.89	0:20.43	0:21.56	0:21.47	0:21.46	0:21.54	0:21.61	0:21.87	0:22.03
0.6	0:19.09	0:20.13	0:20.42	0:21.08	0:21.39	0:21.83	0:21.82	0:22.03	0:22.24
0.7	0:19.19	0:20.02	0:20.83	0:21.64	0:21.06	0:21.61	0:21.79	0:22.34	0:22.05
0.8	0:19.49	0:19.76	0:20.97	0:21.09	0:21.26	0:21.62	0:21.33	0:21.73	0:21.92
0.9	0:19.49	0:19.73	0:20.39	0:21.22	0:21.49	0:22.12	0:21.39	0:22.16	0:23.06

TABLE 5.26: Execution time (min:sec) of SVM-RFE (combo) with Polynomial Kernel optimized.

γ	**0.010**			**0.025**			**0.050**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	13	61.70%	0.308 ± 0.010	19	69.05%	0.251 ± 0.009	16	71.60%	0.225 ± 0.009
0.2	22	63.50%	0.295 ± 0.011	22	73.05%	0.220 ± 0.009	16	76.15%	0.191 ± 0.009
0.3	22	65.75%	0.271 ± 0.009	22	74.50%	0.209 ± 0.006	16	77.10%	0.187 ± 0.012
0.4	28	68.05%	0.264 ± 0.007	22	76.40%	0.195 ± 0.003	16	76.20%	0.186 ± 0.010
0.5	22	68.95%	0.251 ± 0.007	22	76.70%	0.190 ± 0.006	16	80.75%	0.157 ± 0.011
0.6	25	69.60%	0.243 ± 0.008	19	77.50%	0.186 ± 0.007	16	79.70%	0.165 ± 0.014
0.7	28	71.05%	0.239 ± 0.008	22	78.35%	0.176 ± 0.007	16	80.90%	0.153 ± 0.009
0.8	19	69.30%	0.244 ± 0.005	22	78.45%	0.177 ± 0.010	16	79.35%	0.169 ± 0.014
0.9	22	70.90%	0.235 ± 0.006	22	78.75%	0.176 ± 0.008	16	79.70%	0.164 ± 0.014
1.0	25	71.70%	0.228 ± 0.008	19	79.60%	0.168 ± 0.010	16	81.40%	0.153 ± 0.010
γ	**0.1**			**0.2**			**0.3**		
C	Feat.	Acc.	Cost	Feat.	Acc.	Cost	Feat.	Acc.	Cost
0.1	16	76.10%	0.196 ± 0.012	7	67.75%	0.260 ± 0.012	4	57.95%	0.333 ± 0.018
0.2	13	78.30%	0.167 ± 0.009	7	68.00%	0.248 ± 0.020	4	58.15%	0.324 ± 0.023
0.3	13	79.90%	0.159 ± 0.012	7	67.10%	0.250 ± 0.022	4	60.45%	0.310 ± 0.020
0.4	13	78.95%	0.166 ± 0.010	7	75.30%	0.199 ± 0.011	10	56.54%	0.337 ± 0.025
0.5	13	82.60%	0.143 ± 0.005	7	72.75%	0.211 ± 0.024	4	60.55%	0.310 ± 0.015
0.6	13	82.95%	0.137 ± 0.012	7	73.65%	0.202 ± 0.018	4	64.50%	0.276 ± 0.020
0.7	13	82.15%	0.141 ± 0.009	7	72.75%	0.209 ± 0.027	4	56.30%	0.330 ± 0.017
0.8	13	82.40%	0.139 ± 0.007	7	77.30%	0.182 ± 0.016	4	62.30%	0.299 ± 0.017
0.9	13	82.75%	0.139 ± 0.010	7	79.25%	0.158 ± 0.011	7	55.05%	0.343 ± 0.018
1.0	13	84.50%	0.124 ± 0.011	7	79.80%	0.164 ± 0.012	4	58.40%	0.314 ± 0.029

TABLE 5.27: Grid search of SVM-RFE (combo) with RBF kernel.

Chapter 6

Conclusions

This chapter is a general review on the problems, solutions, and achievements related to the development of this project.

6.1 Problems encountered during development

6.1.1 Changes to the planning

Although the writing of the theory section of this thesis was planned for the final phase, it has been switched to the first phase together with research. This has was needed because, in order to have a clear understanding of the ongoing research, it was useful to write it down. Having a written version of the research helps clarify ideas and have an immediate source of truth for the problems encountered while designing the experiments.

This was specially important for the theory on kernels of SVM and the theory of ranking criteria of SVM-RFE. Without a clear understanding of it, the non-linear kernel extension (a cornerstone experiment) would not been possible to implement.

Some delays also occurred due to unforeseen circumstances. These forced some meetups to be pushed to a later date and the development to stop partially or totally. The specific circumstances where:

- 2 week delay caused by medical problems.
- 1 week delay caused by ransomware infection on a managed (family business) server.

6.1.2 Cancelled extensions

Avoid CV

Our definition consisted in using other classifier algorithms, such as LDA (Linear Discriminant Analysis) or Logistic Regression. These however fall too much outside the scope of this project. If they were to be introduced, the theory for both would also need to be written. The theory about the specific characteristics that may make them more suitable, or not, for the RFE procedure would need to be researched as well.

Not only a lot of work would have to be put on researching classifiers that do not really have a direct connection with the SVM-RFE algorithm itself, but this work would also be incompatible with non-linear kernels (due not direct application of the general ranking criteria formula), thus creating two disjoint research.

Historic of weight

This was dropped soon after realizing how the ranking criteria of SVM-RFE works. It is stated in the theory that weights of previous iterations will necessarily be worse estimators of feature importance compared to the current iteration. Therefore, doing a mean, or a similar average, will inevitably lead to worse performance.

Two alternatives have been proposed but also dropped:

1. Utilize the variance accumulated in all previous iterations in some sensible manner.

2. Instead of using the weight, use the alpha values of the last iteration to initialize the SVM optimization problem. This is expected to reduce the computational cost required to reach a solution.

The first alternative lacks any kind of theoretical background, and is likely to not perform any better. Is thus, simply not attractive enough of an experiment to pursue (high chance of failing to produce any improvement at all).

The second alternative is more interesting but has one major flaw. In our current framework we're not implementing the quadratic solver program required to solve SVM, instead we're using the C++ implementation LIBSVM/LIBLINEAR. We're not using C++, we're using Python. This means that in order to use this implementation we're limited to a Python API that performs the binding. This API does not expose the alpha values on initialization, only as a final result. In order to make these aviable we would need to make changes to the C++ implementation, recompile, bind, and then modify the Python API. This is way too much outside the scope of this project and will take much more time than that allocated for correcting bugs in the initial plan.

Stop condition

Initially this extension was planed as a standalone method to reduce the cost in time of the RFE algorithm. Although the experiment itself was a success, that is, the implementation worked as expected, the resulting gain in performance was minimal due to the fact that the stop point (optional feature subset size) is often in the last iterations, and these are precisely the ones that are less computationally expensive.

We reformulated the experiment as method whose objective was merely to find an approximation to of the optional feature subset size, and then use that information to improve on the dynamic step extension.

6.1.3 Non-linear Kernel

This has clearly become the bulk of the project as it required the most effort for both the research and the implementation. Various problems where found during the implementation phase:

No API for retrieving the Kernel Matrix

`Sklearn` does not provide an API that allows retrieving the kernel matrix. Usually it computes the kernel internally based on its name and parameters, and hides it from the user. Fortunately, `sklearn` does provide a precomputed mode that allows computing the kernel matrix yourself and pass it to the solver. This mode is poorly documented, and it was necessary to read to source code to find how to handle it.

Using this mode also slightly increased the computational cost of SVM-RFE, even though `sklearn` own functions where being used to generate the kernel matrix.

Slow optimizations

Experimentally we found that computing the ranking criteria took about 90% of the time on each run of SVM-RFE with the naive implementation. This is grounds for applying the known (but not formalized) optimizations mentioned on the original research paper. This consists on caching results from previous computations and restricting the computation to support vectors.

To apply these optimizations we had to drop the `sklearn` kernel function implementation and made our own. This proved immediately problematic because we could not use `numpy` for most computations and had to rely on explicitly declaring a double loop. `Numpy` speeds up computations by relying on a low level compiled implementation of its functions. By not using it, our equivalent implementation in pure Python performed much slower (in the order of 70 times slower).

We mitigated this problem by using a library called `numba`. This library compiles selected Python functions so that we can get similar speeds to what we would get with an implementation made in a compiled language. Still, our implementation in `numba` was about 3 times slower than with `sklearn`.

Applying the optimizations we managed to get a speed increase by a factor of 3. Although the code is slower, the complexity is one degree faster, from $O(dn^2)$ for the `sklearn` version to $O(n^2)$ for ours.

6.2 Closing thoughts

We believe to have found some useful techniques to improve the computational cost of SVM-RFE, as shown in the *Combo* extension. Each of these techniques, however, has their own limits. It remains to be seen how useful these techniques would be in practice as an off-the-shelf method.

Other extensions, such as *Multi-class*, didn't have much success compared with already well known solutions, and it is doubtful that any application for them may be found.

We've implemented SVM-RFE for non-linear kernels, and some optimizations that where not widely known. This may prove useful to future developers. We've also shown, although already known, that non-lineal kernels may drastically improve the performance of SVM-RFE.

Bibliography

Abdiansah, Abdiansah and Retantyo Wardoyo (Oct. 2015). "Time Complexity Analysis of Support Vector Machines (SVM) in LibSVM". en. In: *IJCA* 128.3, pp. 28–34. ISSN: 09758887. URL: http://www.ijcaonline.org/research/volume128/number3/abdiansah-2015-ijca-906480.pdf (visited on 06/14/2021).

Bai, Lei et al. (Oct. 2018). "Automatic Device Classification from Network Traffic Streams of Internet of Things". In: pp. 1–9.

Bernstein, Matthew N. (Mar. 2017). *The Radial Basis Function Kernel*. URL: http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf (visited on 04/27/2021).

Bishop, Christopher (2006). *Pattern Recognition and Machine Learning*. Microsoft Research Ltd. ISBN: 0-387-31073-8.

Bisong, Ekaba (2021). *Machine Learning: An Overview*. URL: https://ekababisong.org//ieee-ompi-workshop/ml-overview/ (visited on 04/17/2021).

Chapelle, Olivier (May 2007). "Training a Support Vector Machine in the Primal". In: *Neural Computation* 19.5, pp. 1155–1178. ISSN: 0899-7667. URL: https://doi.org/10.1162/neco.2007.19.5.1155 (visited on 04/19/2021).

Claesen, Marc and Bart De Moor (Apr. 2015). "Hyperparameter Search in Machine Learning". In: *arXiv:1502.02127 [cs, stat]*. arXiv: 1502.02127. URL: http://arxiv.org/abs/1502.02127 (visited on 04/18/2021).

Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong (2020). *Mathematics For Machine Learning*. Cambridge University Press.

Ding, Yuanyuan and Dawn Wilkins (Sept. 2006). "Improving the Performance of SVM-RFE to Select Genes in Microarray Data". en. In: *BMC Bioinformatics* 7.2, S12. ISSN: 1471-2105. URL: https://doi.org/10.1186/1471-2105-7-S2-S12 (visited on 03/28/2021).

Ertam, Fatih (Feb. 2019). *Internet Firewall Data Data Set*. URL: https://archive.ics.uci.edu/ml/datasets/Internet+Firewall+Data (visited on 03/25/2021).

Guyon, Isabelle and André Elisseeff (2003). "An Introduction to Variable and Feature Selection". In: *Journal of Machine Learning Research* 3.Mar, pp. 1157–1182. ISSN: 1533-7928.

Guyon, Isabelle et al. (Jan. 2002). "Gene Selection for Cancer Classification using Support Vector Machines". en. In: *Machine Learning* 46.1, pp. 389–422. ISSN: 1573-0565.

Guyon, Isabelle et al. (Jan. 2004). "Result Analysis of the NIPS 2003 Feature Selection Challenge". In: vol. 17.

Li, Jundong et al. (Dec. 2017). "Feature Selection: A Data Perspective". In: *ACM Comput. Surv.* 50.6, 94:1–94:45. ISSN: 0360-0300. URL: https://doi.org/10.1145/3136625 (visited on 03/01/2021).

Mundra, Piyushkumar A. and Jagath C. Rajapakse (2007). "SVM-RFE with Relevancy and Redundancy Criteria for Gene Selection". en. In: *Pattern Recognition in Bioinformatics*. Ed. by Jagath C. Rajapakse, Bertil Schmidt, and Gwenn Volkert. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 242–252. ISBN: 978-3-540-75286-8.

Rakotomamonjy, Alain (Mar. 2003). "Variable selection using svm based criteria". In: *J. Mach. Learn. Res.* 3.null, pp. 1357–1370. ISSN: 1532-4435.

Salarios, ingresos, cohesión social (2017). Tech. rep. Encuestas de Estructura Salarial. INE. URL: https://www.ine.es/jaxiT3/Tabla.htm?t=10911.

Sontag, David (Sept. 2013). "Kernel methods & optimization". en. In: p. 5. URL: https://people.csail.mit.edu/dsontag/courses/ml14/slides/lecture3_notes.pdf.

Vasumathi, D. and Thangavelu S (Mar. 2019). "Scalarizing functions in solving multi-objective problem-an evolutionary approach". en. In: *Indonesian Journal of Electrical Engineering and Computer Science* 13.3. Number: 3, pp. 974–981. ISSN: 2502-4760. URL: http://ijeecs.iaescore.com/index.php/IJEECS/article/view/14164 (visited on 04/30/2021).

Wang, Jingjing et al. (Nov. 2011). "Classification of lip color based on multiple SVM-RFE". In: pp. 769–772.

Wikipedia - Hilbert space, (Apr. 2021). en. Page Version ID: 1017590047. URL: https://en.wikipedia.org/w/index.php?title=Hilbert_space&oldid=1017590047 (visited on 04/26/2021).

Wikipedia - Line (geometry), (Apr. 2021). en. Page Version ID: 1010000668. URL: https://en.wikipedia.org/w/index.php?title=Line_(geometry)&oldid=1010000668 (visited on 04/17/2021).

Xue, Yangtao et al. (Oct. 2018). "Nonlinear feature selection using Gaussian kernel SVM-RFE for fault diagnosis". en. In: *Appl Intell* 48.10, pp. 3306–3331. ISSN: 1573-7497. URL: https://doi.org/10.1007/s10489-018-1140-3 (visited on 03/18/2021).