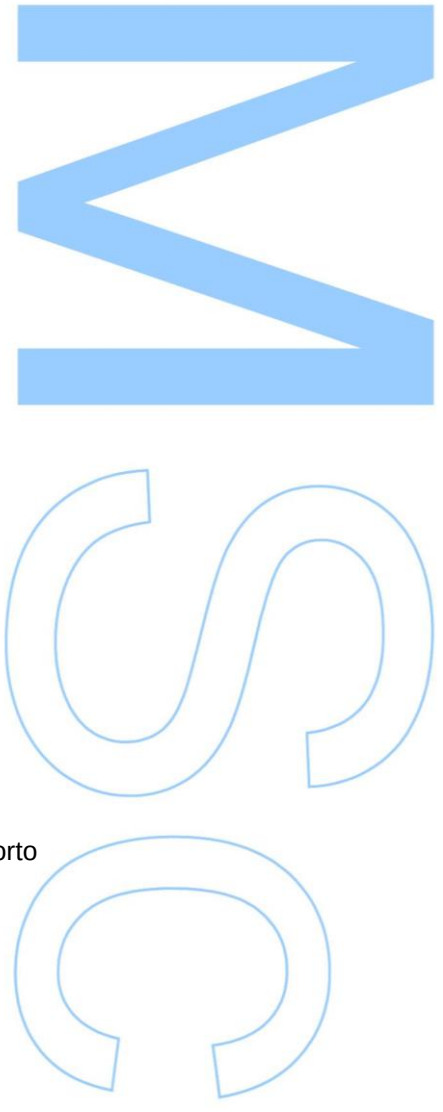
A Parallel Computing Hybrid Approach for Feature Selection

Jorge Miguel Barros da Silva
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2015

Orientador
Ana Aguiar, Professora Auxiliar, Faculdade de Engenharia da Universidade do Porto

Coorientador
Fernando Silva, Professor Catedrático, Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

To my grandfather,
who I will always remember.

Acknowledgements

Many people helped me not only during this thesis but throughout my academic career, I would like to take this opportunity to thank them all. Foremost, I would like to express my sincere gratitude to my supervisors Dr. Ana Aguiar and Dr. Fernando Silva for their continuous advice and support. They have provided me the necessary tools and have always allowed me complete freedom throughout my dissertation. This work would have been impossible without their guidance, knowledge and patience.

I would like to thank Dr. Isabelle Guyon and Dr. Lukas Romaszko for kindly reopening the NIPS challenge for my tests. Their contribution allowed me to improve my dissertation and I am deeply grateful for that. I would like to thank all IT members for conceding me the opportunity to be part of such amazing group and for helping me improve as a professional.

Special thanks must go to my family for providing me unconditional support and encouragement during my time in graduate school.

My best friends Nuno, Rui, Tânia, Diogo, António, Joana thank you for the caring, emotional support, laughs, and entertainment you guys have provided. There is no way I can fully express how much you all mean to me.

Last but certainly not the least, thank you Patrícia, for all your love and for always being there for me.

Resumo

O principal objetivo da seleção de caraterísticas é escolher o menor subconjunto possível de características que tenha o menor erro de generalização em relação ao conjunto total. Devido a conseguir reduzir com eficiência o espaço das características, esta técnica é conhecida por melhorar o desempenho da classificação, reduzir o tempo predição e diminuir os custos de aquisição de dados. Isto torna a seleção de características um passo de pré-processamento fundamental para as tarefas de classificação.

Esta tese apresenta um novo algoritmo híbrido de seleção de caraterísticas. A novidade deste trabalho é um novo wrapper, que consiste numa estratégia não uniforme, dividida em duas fases. A primeira recolhe o maior número de potenciais boas soluções quanto possível, em seguida, estas são exploradas até à melhor pontuação que podem alcançar. Além disso, para reduzir o seu custo computacional, a estrutura do wrapper favorece a utilização de técnicas de paralelização.

Este trabalho explora duas maneiras de paralelizar o algoritmo proposto. A primeira demonstra como é possível fazê-lo utilizando maquinas em locais diferentes. Por isso utilizada o paradigma de memória distribuída. A segunda aproveita as vantagens de múltiplos processadores na mesma maquina, utilizando o ambiente de memória partilhada. O desempenho das duas estratégias de paralelização foi avaliado utilizando até 33 processadores. Os resultados demonstram mostram speedups quase lineares pare as duas, com a estratégia de memória partilhada a ser melhor que a outra.

Para avaliar a qualidade da seleção de características do algoritmo, foram utilizados os cinco datasets do concurso da Neural Information Processing Systems. Os nossos resultados foram comparados com as submissões anteriores utilizando três métricas: exatidão, tamanho da solução e uma métrica que combina as duas. Para a primeira métrica, o algoritmo apresentado teve um rank médio de 60% para todos os datasets, enquanto que a segunda teve sempre no top 15%. Os resultados da métrica combinada tiveram sempre na metade superior sendo que para um dataset obtivemos o rank de 22 em 879 submissões.

Todo o código desenvolvido neste trabalho está disponível no github.

Abstract

The ultimate goal of feature selection is to select the smallest subset of features that yields minimum generalisation error from an original set of features. By effectively reducing the feature space, this technique is known to improve classification performance, reduce prediction time, and decrease the cost of data acquisition. This makes feature selection an essential pre-processing step for any classification task.

This thesis presents a new hybrid feature selection algorithm. The main novelty of this work is the wrapper, which consists in a non-uniform strategy divided in two phases. The first phase gathers as many potential good solutions as possible, then the second one explores them up to the best score they can reach. Furthermore, to mitigate its heavy computational cost, the wrapper maintains a structure that favours the use of parallelization techniques.

This work explores two parallel strategies to execute the proposed algorithm in parallel. The first one exposes how it is possible to solve the problem using machines in different physical places. Therefore, it uses the distributed memory paradigm. The second one takes advantage of multiple processing units present in the same machine, using the shared memory parallel paradigm. The parallel performance of both strategies is evaluated using up to 33 cores. The results show near linear speedups for both strategies, with the shared memory strategy outperforming the distributed one.

To assess the quality of the feature selection algorithm, we used the five datasets from the Neural Information Processing Systems challenge. Our results are compared to previous submissions using three different metrics: accuracy, size of solution, and a metric that combines both. For the first parameter, our algorithm ranks on average among the top 60% for all the datasets while, for the second one it is among the top 15%. For the combined metrics, our results rank among the top half and for one particular dataset we were able to obtain a rank of 22 out of 879 submissions.

The code developed during the work has been made available in github.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data has become an important asset in today's society. In fact, it has been asserted as a new economic class as gold or currency [51]. Despite its value, raw data, especially in cases where there are large amounts of it, is useless. The value of data is related to the efficiency of machine learning (ML) algorithms in extracting knowledge from it.

Machine learning is an area in computer science that explores the construction of algorithms that are capable of finding patterns making predictions on data. These algorithms do not follow explicit programmed instructions, instead they create models about the data which allows them to make data-driven predictions or decisions [31]. Due to its potential this area has been thoroughly studied and several algorithms already exist and have been successfully applied on different problems. Topics such as speech and image recognition, medical diagnosis, and robotics have benefited from the improvements in this area. Furthermore, machine learning has been asserted as the key for innovation, competition, and productivity [8].

According to their characteristics, problems are divided into different categories on machine learning. This thesis will focus on classification problems, which consists in using an algorithm to predict the classification label of an example of data. To achieve that, the algorithm has to be trained with examples of data and their corresponding labels. In figure 1.1 we can see an example of these problems where the algorithm is fed with characteristics of images, also known as features, and then it is told which ones are cats or dogs, which are the labels of the problem. This is known as the training part and it produces a classifier. Later, new examples of characteristics of images, not used during the train part, are shown to it, and one expects the classifier to be able to classify them. The goal of classification algorithms is to produce a classifier that is

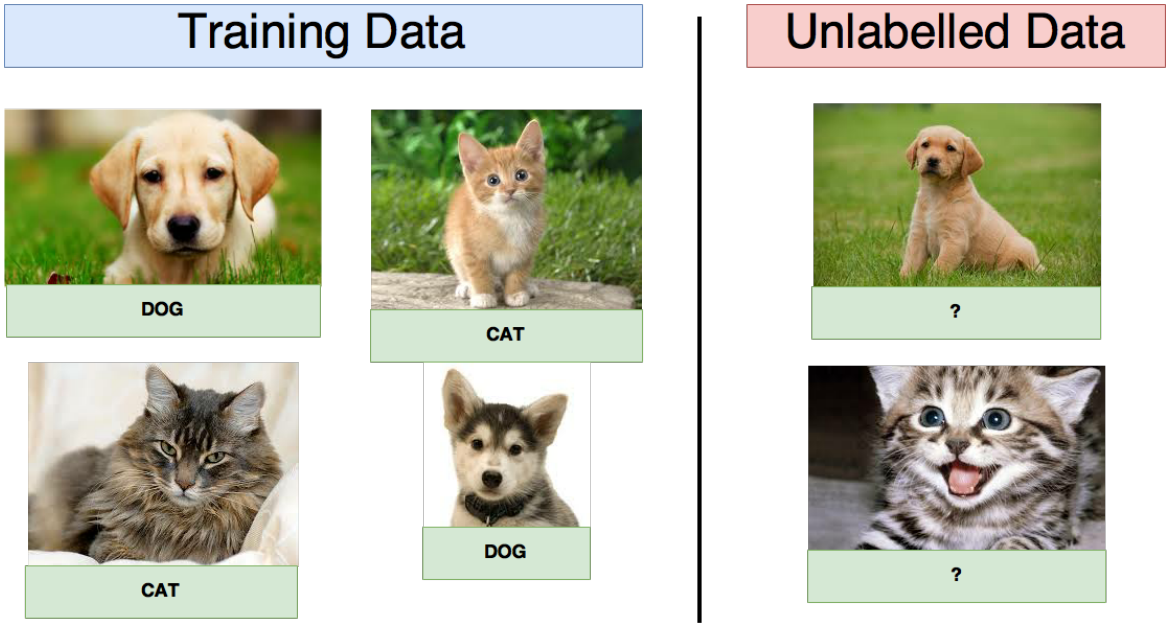


Figure 1.1: Example of a classification problem on Machine Learning.

able to accurately predict new data.

Although classification problems have been widely studied, preparing a classifier for such tasks is not easy. It is common for machine learning users to face difficulties such as: how much data is needed, what features should be added, and does the dataset have outliers and/or noisy data [22]. Usually, researchers gather as much information as possible about a problem and turn that into a processed dataset for classification purposes. This methodology often leads to datasets with a large number of features. In these cases, problems such as curse of dimensionality and overfitting are known to deteriorate the performance of the learning algorithm [13, 18, 43]. Therefore, it is critical in any classification problem to reduce the number of features to a smaller subset before training the classifier.

Feature selection (FS) is the process of selecting a subset of the original features so that the feature space is reduced according to a certain evaluation criteria [32]. The goal is to find a smaller subset that yields the minimum generalisation error. This technique can efficiently reduce the dimensionality (number of features) of the problem. Feature selection algorithms are divided into three categories: filter, wrapper, and embedded [32]. The first ones assesses the quality of features by looking at the properties of data. Wrappers use the learning algorithm to evaluate subsets of features. Embedded encapsulate feature selection with classifier construction. In addition to these categories, there are hybrid methods which intend to use a filter approach as a

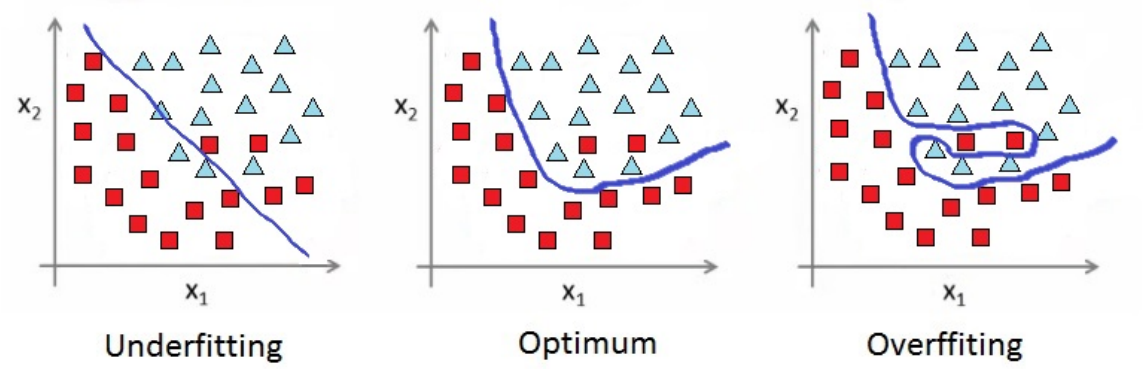


Figure 1.2: Model fitting on training data.

pre-processing step for a wrapper algorithm.

1.1 Motivation

A key requirement to successfully build a classifier is to have data filling the space or at least the part of it where the model is validated [13]. The amount of required data grows exponentially with the number of features (dimensions). Moreover, it is common on classification problems that the chunk of available data is not too large. Therefore, using a high number increases the volume of the space and data becomes sparse through it. This introduces the concept of the Hughes effect [45], which is the name of the curse of dimensionality problem in machine learning. It states that the predictive power of a machine learning algorithm decreases as the dimensionality increases with a fixed number of training examples.

In addition to the Hughes effect, there is other problem that is directly related to using an excessively number of features to produce a classifier. The complexity of a classification model increases with the dimensionality of the dataset. A complex model is more fitted to the training data, which means that it starts *"memorizing"* data rather than learn from it. Thus, loosing its capability to generalise and drastically failing to make predictions for new data. This problem is very common and known as overfitting [13, 43]. Figure 1.2 illustrates the three cases for model fitting. On the first one, the model is not fit enough to the data. By contrast, on the last it is overfitted. The image in the middle, represents a model that despite not being able to perfectly predict the training data, will be the best one to predict new examples, which is the goal of the classification problems.

High dimension datasets poses a real threat to classification problems. Therefore, feature selection is commonly used as a pre-processing step in these tasks. By lowering the dimensions of datasets, feature selection not only increases the performance of the learning algorithm and the understanding of the classification process, but also reduces computational time of prediction and costs of data acquisition [55]. However, feature selection is a very challenging task. In order to select a subset of features, feature selection algorithms require searching through the feature space, testing subsets of features, and evaluating them to find the final solution. The search space consists of all possible subsets, which for a dataset with n features, produces a search space of size 2^n. This makes finding an optimal subset of features intractable in high dimensional datasets. Moreover, many problems of this kind are asserted as NP-Hard [38]. Several algorithms exist in literature that tackle this problem. Commonly, they have to compromise the goodness of their solutions in order to provide results in a practicable time.

1.2 Objectives

This thesis presents Mutual Information Two-phased Wrapper Search (MITWS), a new feature selection algorithm based on an hybrid approach. The idea is to first use a filter methodology to reduce the number of features and then apply a wrapper search in order to find the final subset. According to the results reported in literature, wrapper methods tend to find better solutions [30, 17]. However, they are not frequently used in high dimensional datasets due to their computational cost. By using a filter as a pre-processing step before the wrapper search, the feature selection algorithm proposed in this thesis shows that it is possible to use this later technique on datasets with a large number of features.

MITWS will combine an already existing filter approach with a novel wrapper search strategy developed during this work. This new technique, encapsulates a new heuristic that is divided into two phases and intends to propose a different strategy to search for solutions, while maintaining a structure that would favour the use of several processing units. Furthermore, to improve computational performance the novel wrapper will be implemented on both shared and distributed memory parallel environment, using different strategies on each one.

1.3 Thesis Structure

The structure of this thesis is the following. Chapter 2 deals with literature review of feature selection. It starts with a generalization of the problem, explaining each type of algorithm while providing information about the most used ones. Chapter 3 thoroughly explains the MITWS algorithm while chapter 4 discusses the novel wrapper search. Chapter 5 presents two strategies to achieve parallelization of the MITWS algorithm. Chapter 6 address the performance of the implemented work with respect to parallel performance and quality of feature selection. Finally, chapter 7 addresses conclusions and future work.

Chapter 2

Literature Review

The fact that classifiers have low performance for high dimensionality datasets, turns feature selection into an indispensable component on the process of creating them. Given a dataset with several features, the idea of FS is to select those that are relevant, while removing irrelevant and redundant ones (see figure 2.1 for an example). Vergara et al. [58] explain three possible levels of relevance: strongly, weakly, and irrelevant. The first one, refers to features that provide unique information about the class, meaning that they cannot be replaced without loss of information. Weakly relevant features grant class knowledge, still they are not unique in the dataset, and may be replaced by others. Irrelevant features do not contribute with any info, therefore they can be discarded.

In addition to improving classification performance, there are other advantages of feature selection: facilitating data visualization and comprehension, reducing the costs of dataset acquisition and storage, reduce training and classification times [22]. As a consequence, FS has been widely studied. However finding the best features for a classification task is still challenging. The volume of the search space makes it infeasible to perform an exhaustive search in most cases. Therefore, numerous different

Heigth (m)	Heigth (cm)	Weigth (kg)	Heart Rate (bpm)	Hair Color	Label
1.80	180	80	70	Black	Man
1.65	165	68	65	Brown	Woman
1.55	155	62	63	Blond	Woman
1.90	190	91	72	Brown	Man

Feature Selection →

Heigth (m)	Weigth (kg)	Heart Rate (bpm)	Label
1.80	80	70	Man
1.65	68	65	Woman
1.55	62	63	Woman
1.90	91	72	Man

Figure 2.1: Performing feature selection on a dataset.

strategies that use heuristics to decrease the computational cost of the problem have been proposed. This led to the existence of several algorithms in the literature. In this chapter, we will discuss the most important ones.

2.1 Classification Problems Workflow

Let us start by analysing the required workflow to solve a classification problem in order to explain the process and to understand where feature selection fits in. The first step for a classification task is to identify the problem. At this stage, the labels for the problem should be defined. The next is to construct a dataset for the classification task. It is crucial that data is related to the problem that is being modelled, otherwise it will not be possible to achieve good classification results. As a simple example, it is not possible to anticipate the weather using heart rate information. In addition to that, only the most informative features about the problem should be used. In cases where there is not a complete knowledge about that, brute-force is an alternative. In this scenario, a large number of variables are measured and inserted into the dataset, expecting that best features can be isolated in the future. As Kotsiantis et al. [31] stated:

"A dataset collected by brute-force is not directly suitable for induction because of the noise and possible missing feature values."

It is possible to deal with this matter, which leads us to the next step in the classification workflow: data pre-processing. At this stage, key issues such as missing values and outlier detection should be handled. There are several statistic analysis approaches to deal with these problems [1, 26]. Additionally, this is the stage of the problem where the number of features of the problem can be reduced using a feature selection algorithm.

Selecting the classification algorithm is the step that follows. A wide range of classification algorithms exist, and despite their differences it is not easy to foresee which one is the best for a given problem. Thus, it is a common approach to test and compare several of them, and in the end, keep the one that provides the best results [31].

Classifiers evaluation is most regularly based on the prediction accuracy. A typical technique is to divide labelled data in two thirds to train the model, and use the remaining to test the accuracy. However, this procedure often leads to bad generalisation outside of the evaluation dataset. Therefore, to reduce the generalisation error,

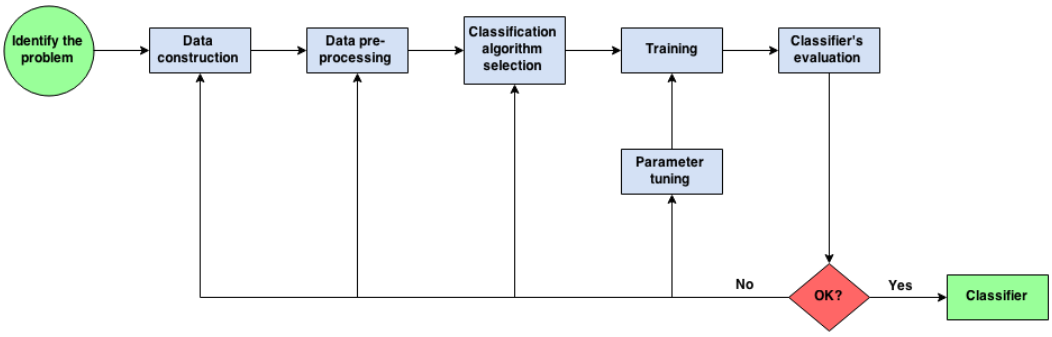


Figure 2.2: Workflow for classification problems.

more complex techniques, like cross-validation [29], can be used.

As long as the overall procedure is not able to produce a satisfactory classifier, the process should return to previous stages. There are many causes that may negatively affect the performance of a classifier [31]:

1. Relevant features are not being correctly identified.
2. Dataset does not have enough examples.
3. The number of features is too large.
4. The selected pre-processing technique is not good enough.
5. The elected classifier is not suited for the problem or needs parameter tuning.

Thus, it is not clear to each stage the workflow should return. The ultimate goal of every classification task is to perfectly predict unseen data. However, this is extremely unlikely to occur, and these tasks tend to last very long. Usually, they become an ongoing work where several new attempts are made in order to improve the predicting ability of the classifier. Figure 2.2 illustrates the mentioned workflow.

2.2 General Procedure for Feature Selection

Feature selection algorithms operate by combining a search strategy to find combination of features, with an evaluation method to score them. In the end of the process, the highest score subset(s) are considered the solution(s). Despite the existence of several algorithms, they all follow a general procedure that consists of four steps: subset generation, subset evaluation, stopping criteria, and result validation [42, 38, 32].

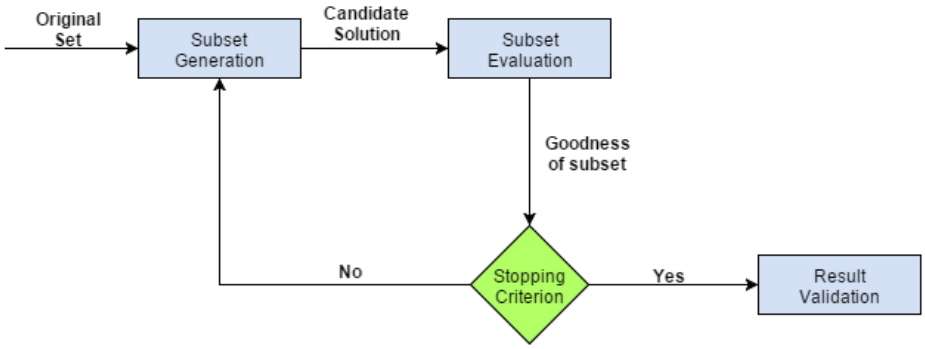


Figure 2.3: The four steps of feature selection.

The first one determines which subsets will be tested on the process, the next one represents the function that assigns a score to subsets, consequently allowing ranking them. The stopping criteria regulate the intensiveness of the search. Finally, the results validation is the part where the quality of the solution is assessed. Figure 2.3 illustrates these steps, which we will discuss in more detail in the following sections.

2.2.1 Subset Generation

Subset generation represents the process of the heuristic search, where each state in the space specifies a candidate subset for evaluation. Two key issues are addressed at this step: successor generation and search organization.

Successor Generation

Successor generation designates the method for expanding a subset into several new ones. According to authors of [38], there are four basic operators to address this:

- **Forward:** New subsets are produced by adding multiple features one at the time to the subset to which successors are being generated.
- **Backward:** New subsets are produced by removing multiple features one at the time from the subset from which successors are being generated.
- **Compound:** This operator applies k forward steps, followed by l backward ones. By doing so, new iterations between different features are explored [32].
- **Random:** Subsets are randomly selected.

Search Organization

Search strategy designates the walk-path through the states, defining, along the way, from which subsets successors should be generated. This part defines the computational cost of the feature selection algorithm, as well as its ability to find solutions. Therefore, it is plausible to assume this is the most important part of the procedure. Authors of [38] categorized searches into: complete, sequential, and random. Due to the large amount of algorithms that use genetic searches, in this thesis this is added as a category.

- **Complete Search:** This type of search guarantees finding the optimal solution. The exhaustive approach is an example of a complete search. However, to being characterized as complete, it is not required to be exhaustive. Instead, some heuristic functions may be used to cut the search space, without compromising optimal solutions. Branch and Bound and Beam Search [53, 21] are some examples of other complete searches.
- **Sequential Search:** Define a group of subsets to test in a certain level, and select the best to generate the successors to the next one. From within levels, the number of features on the solutions either increases or decreases one at the time. These searches are easy to implement and usually provide results very fast. However, the quality of the solutions is often poor. Some examples of these searches are stated in [24].
- **Random Search:** In this approach, the idea is to randomly guide the search. Las Vegas and Las Vegas Incremental algorithms [42] are both examples that fit this category.
- **Genetic Search:** These are a different type of searches that already incorporate the successor generation. Their idea is to mimic the process of natural selection which consists of three operators: selection, mutation and crossover. Initially several candidate solutions are spawned. Then, the quality of each one is evaluated and the best ones are selected. On the next step, new potential solutions are generated combining the elected ones from the previous stage. Genetic operators such as crossover and mutation are used at this point. The process repeats until the end. There are several searches of this type in literature [63].

Due to the importance of this part, it is relevant that the presented searches are further analysed. Sequential ones are the best in terms of computational cost. On a dataset

with n features, most of these methods require testing a maximum of $\sum_{i=0}^{n} n–i$ potential solutions. However, they do not produce good results. Their inability to find quality subsets is related to the fact that the addition or removal of a feature to the solution is permanent. During the search, specially in the early stages, there are not any proofs that the elected features should be part of the optimal solution [17]. Therefore, since the beginning the quality is being jeopardized.

The complexity of random searches is totally dependent of the defined amount of tests. Additionally, it is hard to predict the quality of solutions. They rely on the fact that randomness can help escaping from local optima. Usually, these searches are associated to cases where there is an individual ranking of features. Therefore, the randomness of the process can be controlled by it, improving the likelihood of the best features being selected [11].

The computational cost of genetic searches depends on the size of the initial population and on the times the process is repeated. As those numbers increase, so does the probability of finding better solutions. However, these searches tend to converge to local optima [35].

Complete searches find the global optima. However, they have the highest computational cost and tend not to be used in high dimensional datasets [38]. During the process, they rely in a cutting state heuristic. Its intensiveness decreases as more states are cut-off in the early stages. Additionally, these searches, with the exception of the exhaustive one, require using a subset evaluation function that prevents the heuristic from cutting subsets that lead to the global optima. These specific evaluators impact the definition of the optimal solutions conditioning the characteristics of the final solution. The monotonicity property for the Branch and Bound algorithm [53] can be used as an example of these requirements. Table 2.1 summarizes the advantages and disadvantages of the mentioned searches.

Table 2.1: Pros and cons of feature selection searches.

Search	**Pros**	**Cons**
Sequential	Low computational cost	Low quality solutions
Random	Manageable computational cost	Rely on randomness
Genetic	Average computational cost Good quality solutions	Tend to converge to local optima
Complete	Guarantee finding global optima	Very high computational cost Require specific subset evaluation functions

2.2.2 Subset Evaluation

This part defines the process of obtaining a score for the subsets tested. The evaluation criterion delineates the quality of a subset and it affects the definition of the optimal solution. More precisely, the global optima to a certain evaluation criterion, may not even be a local optima on a different one. There are two groups that categorize evaluation functions: independent and dependent [38].

Independent Criteria

Independent criteria evaluates the quality of a subset of features considering the characteristics of the data. Most of the times these metrics are used to assess quality of an individual feature. Additionally, they are associated to filter approaches, which will be reviewed in section 2.3.1. Based on the metrics used, these criteria are divided across several categories [42]:

- **Distance or divergence measures:** the capability of features to differentiate the conditional probability between classes is assessed. Jeffrey's divergence and Kaga's divergence are some examples of these metrics [32].
- **Information or uncertainty measures:** the information that a feature adds to classes is determined. This concept is called information gain, and as an example we have Shannon entropy and all its variants [58].
- **Probability of error measures:** the ability of a feature to minimize the probability of a classification error is estimated. Bayesian probability is the most known example of this technique [57].
- **Dependency measures:** assess the capability of features in predicting the labels. Correlation coefficients can be seen as an example [22].
- **Interclass distance measures:** the distance in the data space is used to determine which are the best features to separate different classes. Euclidian distance is one example of such technique [32].
- **Consistency measures:** they are based on the principle that features with same values should belong to the same class. The violation of this rule results into a penalty for the feature, allowing a score to be obtained. Some of these approaches are stated in [42].

Dependent Criteria

Dependent criteria use the learning algorithm to estimate the quality of features. Every evaluated subset is used to train the model, then the performance of the later is used to assign a score to the subset. In classifications tasks the most common way to quantify the learning algorithm performance is to use the accuracy of predictions. Evaluation functions that use a dependent criteria are restrictedly associated to wrapper models which will be thoroughly discussed in section 2.3.2.

2.2.3 Stopping Criteria

Stopping criteria defines the conditions to end the search. Some used criteria are [32]:

- The search is complete.
- Some specified bound is reached. The bound may be a defined minimum or maximum cardinality of the subset of features or a maximum number of iterations.
- Finding a sufficiently good subset.
- The successors of the current state do not improve the evaluation criteria.

2.2.4 Result Validation

After the whole process is completed, a final subset of features is obtained. In order to check whether it is good enough for the problem, it is important to validate it. A straightforward method for validation is to directly measure the solution using the prior knowledge about the dataset. If there is information about the relevant features, it is possible to validate a solution by comparing the selected features against the ones that are known to be relevant. In these cases, information about which features are irrelevant or redundant is also important, mainly because their presence in the solution indicates its quality.

The most common scenario is the lack of knowledge about the dataset. Therefore, different techniques must be used. For example, it is possible to use the performance of the final solution on the model, and compare it to the one obtained with the whole set of features or any other subsets. Additionally, it is normal to use different model algorithms and compare the final solution on them [9].

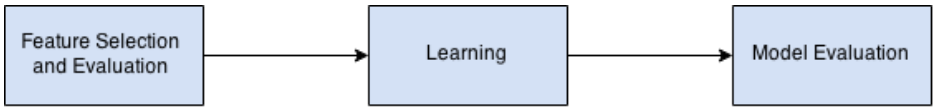


Figure 2.4: Overview of filter approach.

2.3 Categorization of Feature Selection Algorithms

As stated before, there are several feature selection algorithms, which are categorized into three groups: filter, wrapper, and embedded [32].

2.3.1 Filter Algorithms

Filter algorithms use an independent metric to evaluate single features or subsets of features in order to identify which ones are more relevant to the problem. They assume complete independence between the data and the learning algorithm. As result of that, the same strategy can be combined with distinct learning algorithms.

These type of algorithms have low computational cost. However, related to the process of finding solutions, they are known to have worse performance than other types of feature selection algorithms [17].

An overview of filter algorithms is illustrated in figure 2.4. The first state represents the part where the metric is used to obtain a solution. After that, the result is used to train the learning algorithm. In the end, the goodness of the final solution is evaluated at the model evaluation stage.

Section 2.2.2 identified the five types of metrics that can be used in filters. Nevertheless, filter algorithms can still be divided into univariate or multivariate, depending on the way they search the features [50].

Univariate refers to methods that rank individual features by assigning a score to each one. Then, the rank can be used to select the solution, or to guide a search towards it [15, 6]. These methods are very fast to compute, but fail to remove redundant features that negatively impact learning performance. There are several examples in the literature [6, 16, 22, 60].

On the other hand, feature selection algorithms that evaluate subsets of features are named multivariate. They are known to be extremely efficient in removing redundant features, however they are prone to overfit the model. As a result, it is a common

approach to use a univariate method to filter the most irrelevant features, before using an multivariate method [22]. There exist several algorithms with these characteristics [6, 42, 7].

Figure 2.5 illustrates the differences between the two categories.

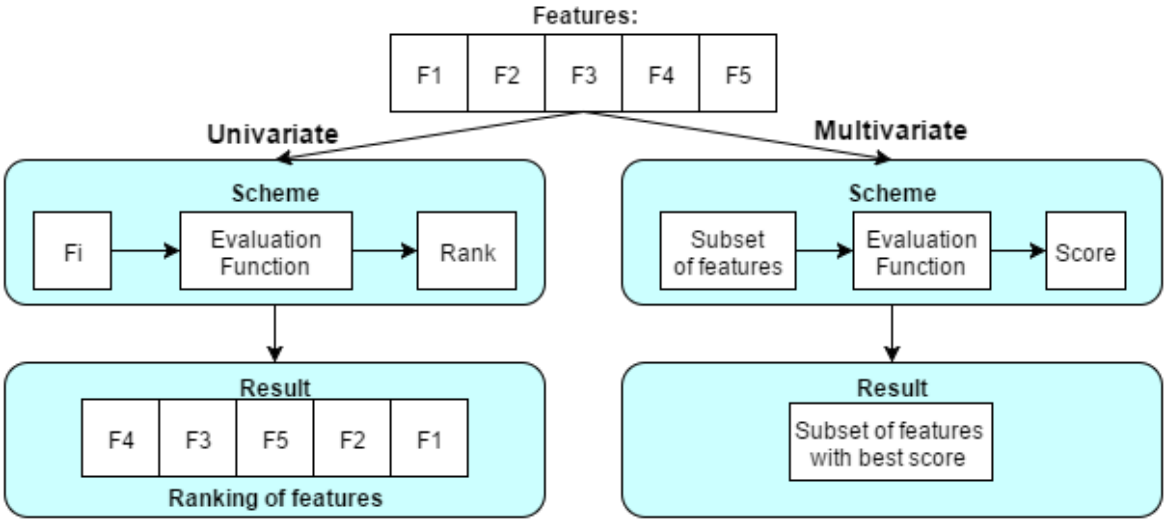


Figure 2.5: Differences in univariate and multivariate methods.

2.3.2 Wrappers

The idea of wrappers is to use the learning algorithm as a *"black-box"* to guide the search towards the solution. Therefore, dependent criteria evaluation functions are used on these models. Figure 2.6 illustrates the overview of a wrapper approach.

Since the learning algorithm is used in the process of selecting features, wrappers usually find better solutions [30, 17]. However, these solutions are strictly related to the selected algorithm. Therefore, it is not advisable to use a wrapper to obtain a subset of features with the intention of using it in several learning algorithms.

In terms of computational cost, wrapper algorithms are expensive, since they require training and testing the learning algorithm at least one time for each potential solution. Most of the times, cross-validation techniques are used [17], which even aggravates the problem. As a result, these methods are not frequently used on datasets with large amount of features.

Combining different search strategies with distinct classification algorithms results in creating new wrapper methods. Therefore, there are several algorithms in the

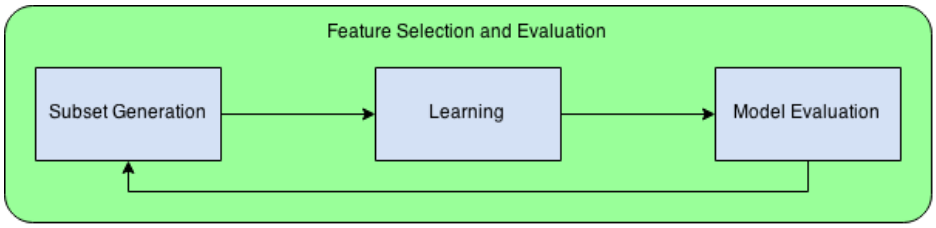


Figure 2.6: Overview of wrapper approach.

literature. Some examples are: the hill climbing and best-first search used on Kohavi's work [30], scatter search [39], sequential backward or forward search [6], and genetic searches [27].

2.3.3 Embedded

Embedded methods are inspired in wrappers and filters, trying to use the best qualities of both. They encapsulate feature nomination with classifier construction. By doing that, the feature selection part interacts with the learning algorithm which results in better solutions, as is the case of wrappers. However, they do not require training the model multiple times, which makes them less computational expensive. Furthermore, these methods are very specific to a learning algorithm. Meaning that an embedded method can only be used with the exact learning algorithm it was built to work with.

According to Tang et al. [55] there are three types of embedded algorithms:

1. **Pruning:** these methods first train the classifier with all the data, and then try to remove features while maintaining the classifier performance. The RFE-SVM algorithm introduced at [23] is an example.
2. **Build-in:** these approaches have a mechanism that selects the features as it constructs the classifier. Decision trees are the best known example of this type [14].
3. **Regularization models:** these strategies use objective functions to try and minimize fitting errors while eliminating features that are not needed for the learning algorithm. There are several examples of regulation models: L1-norm, LASSO, and Concave Minimization [34, 55].

Figure 2.7 illustrates the overview of embedded algorithms. In the same way as wrappers, there is no separation between learning and feature selection. However,

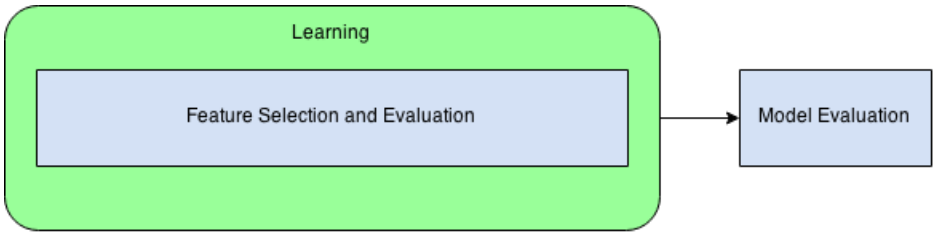


Figure 2.7: Overview of embedded approach.

the model is being created while features are being selected and it is not ready until the end of the process. Therefore, the model evaluation step has to be done at the end of the whole process.

2.4 Comparing Feature Selection Algorithms

In this chapter many feature selection algorithms were presented, but still many were left out. Performing a comparative study of all of them is a very difficult task. Mainly, because it is very challenging to find out the effectiveness of a FS method, without knowing in advance which features are relevant on the dataset. This is the most common case. Moreover, distinct datasets present different challenges which makes it hard to generalize the quality of algorithms.

The usual way to compare FS algorithms is to use different strategies on the same problem, and assert the quality of each final solution. Parameters such as number of features, accuracy of the model, and time can be used.

Belanche et al. [6], tested several fundamental algorithms to assess their performance in a controlled experimental scenario. Quoted from their conclusion:

"Our results illustrate the pitfall in relying in a single algorithm and sample data set, very specially when there is poor knowledge available about the structure of the solution or the sample data size is limited."

The mentioned study, reinforces the idea shared by many researchers that there is no clear indication which is the *"best algorithm"* for feature selection [41]. Many comparative studies of existing feature selection methods have been done in the literature. For example, the work at [6] applied seven filters, two embedded, and two wrapper methods to 11 synthetic datasets, and compared their performances. Another example is the work of Duch et al. [15], that compared five entropy-based

Table 2.2: A taxonomy of feature selection techniques [50].

Model search	Advantages	Disadvantages	Examples
Filter	Univariate		
	Fast	Ignores feature dependencies	Information Gain
	Scalable	Ignores interaction with the learning algorithm	Pearson Correlation
	Independent of classifier		
	Multivariate		
	Models feature dependencies	Slower than univariate techniques	LVF
	Independent of classifier	Less scalable than univariate techniques	FOCUS
	Better computational complexity than wrappers	Ignores interaction with the learning algorithm	mRMR
Wrapper	Deterministic		
	Simple	Risk of overfitting	Sequential searches
	Interacts with the classifier	More prone to getting stuck on local optima	
	Less computationally intensive than randomized	Classifier dependent selection	
	Randomized		
	Less prone to local optima	Computationally intensive	Random searches
	Interacts with the classifier	Classifier dependent selection	Genetic searches
	Models feature dependencies	Higher risk to overfitting	
Embedded	Interacts with the classifier	Classifier dependent selection	RFE-SVM
	Better computational complexity than wrapper	Changing classifier means changing algorithm	Decisions trees
	Models feature dependencies		LASSO

filter methods. Similarly to the first presented study, here authors also concluded that there is not one best method for different datasets. Related to high dimension datasets, work at [14], analysed different FS methods combined with various classifiers. As a conclusion, authors showed the importance of using FS methods instead of all the available features. However, once again they did not conclude that there is a FS method that performs better than all the others. Hao et al. [24] compared the performance of sequential search and genetic algorithms, reaching the conclusion that no algorithm consistently outperforms the others.

Several more studies that compare feature selection algorithms with similar outcomes could be added to the list. Because of that, instead of trying to reason which algorithm performs better, table 2.2 presents the advantages and disadvantages of each type of feature selection.

2.5 Hybrid Approach

The three main categories for feature selection algorithms were discussed. However there is another methodology whose importance has been growing. This relatively new approach is called hybrid and its goal is to combine filter and wrapper methods for

performance improvement. As previously discussed, filter methods are computational fast but often fail to produce good solutions. On the other hand, wrapper methods achieve good results but they are very time consuming, and impracticable to use on high dimensional datasets. The idea behind an hybrid approach is to first use a filter method to reduce the feature space and then adopt a wrapper mechanism to select the final solution. By doing that, a high dimensional dataset will be transformed into a lower dimensional one, therefore using a wrapper approach becomes practical.

There are several filter methods that can cut the feature space. For example, the IFSFFS algorithm [60] uses F-score filter metric to rank individual features, then the rank is used to more efficiently guide the wrapper search. On other example, the QBB algorithm [42] applies the LFV filter algorithm in order to generate subsets, which are utilized as starting points for a branch and bound search. One last example is presented in [61], where authors refer to a mutual information metric to remove features before proceeding to a more computational expensive wrapper search.

2.6 Parallel Feature Selection

Selecting the ideal set of features is far from an easy task. It usually requires many attempts until the desired result is attained. A conventional methodology is to change parameters on the algorithms or test different ones to compare results. Moreover, depending on the size of the dataset and on the algorithm chosen, a feature selection process can take a large amount of time.

Parallel computing emerged as a potential solution to tackle this problem. Carrying multiple computations simultaneously to solve a problem relies on the principle that large problems can, in fact, be divided into smaller ones [12]. Taking a closer look into the general procedure for feature selection, it consists in generating and evaluating huge amounts of subsets until a stop condition is reached, and the best subset is provided as final solution. This problem could be easily divided into smaller tasks, where each one is defined as the overall process on each subset, turning feature selection into an ideal candidate for parallel computing techniques. This triggered researchers to exploit parallelism within feature selection algorithms in order to improve their execution times. For example, Azmandian et al. [4] used graphic processing units to accelerate their feature selection algorithm. Lopez et al. [39] also resorted to parallelism to speed up a scatter search in order to obtain better performance in terms of execution time and quality of solution.

Chapter 3

A Hybrid Feature Selection Approach

From the previous chapter it is possible to conclude that filter feature selection methods are fast to compute, but often fail to produce good results. Otherwise, wrappers are able to find the best solutions, however the amount of computations required by them are huge, thus making it impracticable to use them on high dimensional datasets. Hybrid approaches emerge as a technique to help in this issue. By using a filter algorithm to decrease the search space of the problem as a first step, they enable wrapper approaches to be used despite the number of features on the dataset.

Inspired by these facts, an hybrid method is proposed in this thesis, combining a filter and wrapper method, aiming to preserve the advantages of both while mitigating their drawbacks. This chapter presents the Mutual Information Two-phased Wrapper Search (MITWS) which is a new hybrid approach and one of the contributions of this work. The innovative algorithm combines an already existing filter approach with a novel wrapper approach developed in the context of this work.

The MITWS algorithm is divided into two parts: filter and wrapper, as illustrated by figure 3.1. The first one, uses an univariate method to rank features individually. Based on a threshold and on the estimated ranks, features are selected to the next phase. The goal of this part is to use a less costly computational method to reduce the search space. Therefore, removed features are considerer irrelevant and are not used any further in the next stages of the algorithm. Individual ranks are obtained using the Mutual Information (MI) [58] as metric to evaluate features. The wrapper part searches the feature space by using a novel meta-heuristic in order to find the final

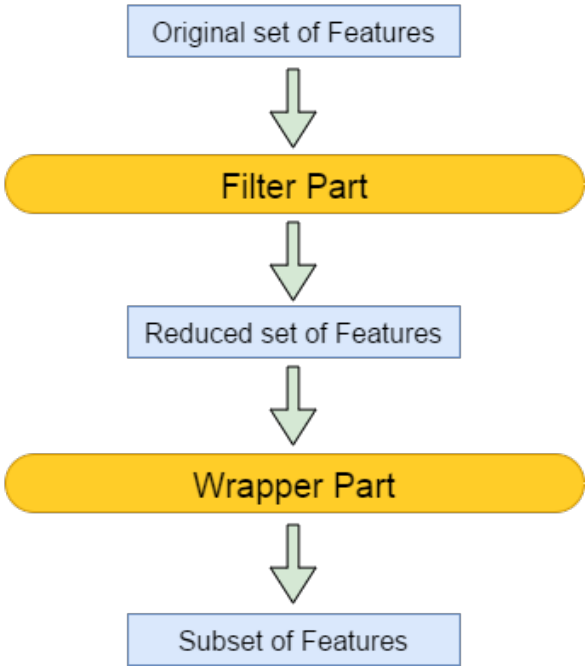


Figure 3.1: Schematic of the MITWS algorithm.

subset of features. As previously mentioned, wrappers require a learning algorithm to evaluate the goodness of subsets. By comparing Support Vector Machines (SVM) with several learning algorithms in the context of classification problems, Vinodhini and Chandrasekaran [59] show that SVMs in general outperform other classifiers. Thus, making SVMs a more desirable choice for the proposed method.

Two additional algorithms were added to MITWS: Uncertain Class Attribute Interdependency Maximization (UCAIM) [19] and Grid Search [37]. The first one is used to discretize data, which is a mandatory procedure to calculate MI in cases where variables have continuous values. The Grid Search is a very popular method used to estimate the parameters of learning algorithms.

The workflow of the MITWS algorithm is illustrated in figure 3.2. The first step is to prepare data, which will later be discretized with the UCAIM algorithm. Then, the feature space is reduced by eliminating features that are not able to pass the MI filter. The next stage, if necessary, is to estimate the SVM parameters using the Grid Search. Finally, the algorithm executes the wrapper search which is responsible to find the subset of features that is presented as final solution. All the previous steps are explained in more detail in the following sections, with the exception of the wrapper search. Since this last part introduces a novel strategy which makes it the most significant contribution of this thesis, a full chapter is dedicated to it (chapter 4).

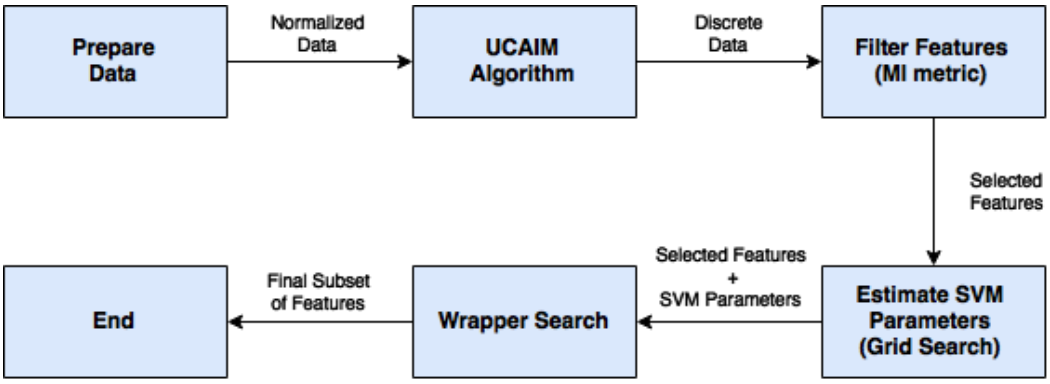


Figure 3.2: Workflow of the MITWS algorithm

3.1 Data Preparation

This is the stage where data is read from files and pre-processed. In most cases, this process includes techniques to find outliers that may jeopardize the performance of the learning algorithm. Although there are several techniques to detect and remove them, this process usually requires some knowledge about the dataset. This procedure is rather specific to the data and thus it is not included as part of the present hybrid approach. Instead, MITWS assumes that the dataset is already clean and ready for learning purposes. In any case, this stage implements normalization of the feature values to a scale from 0 to 1. This is a recommended procedure in order to improve the performance of learning algorithms [22].

3.2 UCAIM Algorithm

In order to discretize data, UCAIM, an evolution of the original CAIM algorithm [33] was added to MITWS. Both methods have the goal to delineate intervals on data in such a way that the interdependence between features values and class labels is maximum. Despite the fact that both algorithms perform well, the evolutionary approach adds the offset component, which takes into account cases where data is unbalanced (number of examples in the data is not equally distributed by the class labels). The UCAIM algorithm has been shown to outperform the original one in these cases. Moreover, it performs as well as the CAIM on datasets where data is balanced [19].

The UCAIM algorithm starts by setting the initial discretization scheme, D, as a set

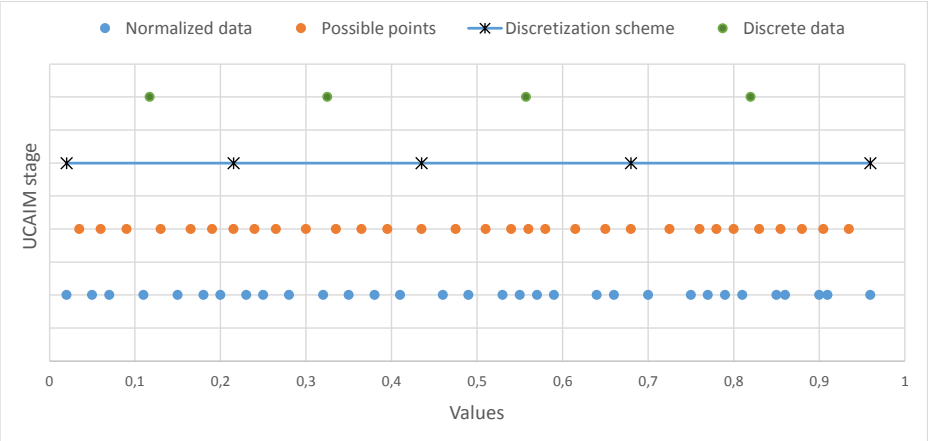


Figure 3.3: Steps of the UCAIM algorithm

of two elements: the maximum and minimum values. Then, it proceeds to define a set of possible points. These are all the midpoints between each adjacent pair in the sorted and non-duplicate set of values. After that, UCAIM iteratively tries to add possible points to D. At each round, all possible points are added, one at the time, to D. Then, formula 3.1, which tries to maximize the interdependence between classes, is used to evaluate the quality of D with the recently added point. At the end of the round, the point with the best score is definitely appended to D. The process stops when no point could improve the score that D has at the start of the round, and there are at least as many intervals as classes. By the end of the UCAIM algorithm, a discretization scheme D is obtained. Later, for each feature value, the interval on D where it belongs is discovered, and the value is converted to the midpoint of that interval. Thus, achieving the desired discrete data.

Algorithm 1 represents the steps required to find D for a given feature F_i and its possible values V_i on a classification problem with S label classes. Additionally, figure 3.3 illustrates an example of all the estimated values during different stages of the process.

$$\text{UCAIM}(F_i, D) = \frac{\sum_{r=1}^{n} \frac{max_r^2 \times Offset_r}{M_{+r}}}{n} \tag{3.1}$$

where n is the number of intervals, r iterates through all intervals, max_r is the maximum value inside an interval, M_{+r} is the total number of values on the interval, and offset:

$$\text{Offset}_r = \frac{\sum_{i=1}^{S}(max_r - q_{ir})}{S-1} \tag{3.2}$$

where S represents the classes labels, q_{ir} are the number of values in interval r that belong to class i, and max_r is the maximum number of values in interval r across all classes. Basically, Offset_r represents the average difference of the number of points in all classes to the number of points in the class that has the most points in that interval.

Algorithm 1 UCAIM Algorithm

```
1: procedure UCAIM(V_i, S)
2:     values ← RemoveDuplicates(V_i)
3:     min, max ← FindLimits(values)
4:     B ← GeneratePossiblePoints(values)
5:     K = 1, D ← {min, max},
6:     BestS ← 0, BestP ← {}
7:     while K ≤ S or GlobalUCAIM < BestS do
8:         GlobalUCAIM ← BestS
9:         D ← D ∪ BestP
10:        for P ∈ B do
11:            auxD ← D ∪ P
12:            auxS ← GetUCAIMScore(auxD)
13:            BestS, BestP ← UpdateBest(P, auxS)
14:        K = K + 1
```

3.3 Filter Part

In contrast to some feature selection algorithms, the aim of the filter part is not to select a final subset of features. Instead, MITWS uses it as a pre-processing step to eliminate features and make it practicable for a more intensive search on the wrapper part. Therefore, the presented filter should have the following characteristics:

1. **Evaluate single features.** Several filter approaches evaluate subsets of features (multivariate methods section 2.3.1). However, to keep a low computational cost on the presented approach, features are evaluated individually to avoid searching for feature subsets.
2. **Not very restrictive.** The percentage of removed features should not be very large. Although as less features pass the filter the faster the wrapper ends, it is difficult to accurately assess the quality of a feature just by using

a metric [32]. Moreover, it has been shown that features considered irrelevant when individually evaluated, are in fact important when inserted into a specific set of features [22]. Hence, to avoid compromising the goodness of the final solution, it is important to avoid removing a large number of features at this stage.

There are several algorithms in the literature that fulfil the first requirement. These methods are called univariate (section 2.3.1). As previously mentioned, the two most commonly used univariate metrics are Mutual Information (MI) and Pearson Correlation Coefficients (PCC). Both metrics measure the dependence between two variables. Nonetheless, there is a key difference between them. MI measures the general dependence between the variables while PCC measures linear dependence. Li et al. [36] tested this property and concluded that this makes MI a better metric. Moreover, MI has been widely used on feature selection [58, 22, 15]. For these reasons, MI was selected as metric to evaluate individual features.

In order to calculate MI, a method to estimate the joint probability must be used. To better understand the calculation of MI, suppose that for a feature f, there are i possible values and j possible labels. Hence, $P_{i,j}$ represents the joint probability of the ith possible value of f belonging to the label j. Then, to calculate MI the following formula is used:

$$\mathrm{IG}(f) = \sum_{i} \sum_{j} P_{i,j} log \frac{P_{i,j}}{(P_i \times P_j)} \tag{3.3}$$

Calculating the MI score for every feature does not remove features by itself, so after obtaining the scores, a strategy was defined to create a threshold. The idea is to remove features whose MI score is below the threshold. Since MI scores diverge a lot when changing datasets, a fixed threshold could not be defined. Instead, users are able to manipulate a percentage P which is then used on the formula 3.4 to calculate the threshold. Basically, the threshold is defined as a percentage of the maximum MI score obtained.

$$\text{threshold} = MI_{max} - (MI_{max} \times P) \tag{3.4}$$

Although MITWS does not present any restrictions to the definition of the value of P, it is recommended to use a value that does not define a very restrictive threshold.

3.4 Grid Search

SVM was the selected learning algorithm to evaluate subsets of features. These algorithms have some parameters that must be tuned in order to provide better results [37]. However, it is common for researchers not to know which parameters should be used because they vary depending on the task. On our proposed method, users are allowed to define the parameters, but if they do not specify them,the algorithm estimates the best to be used.

MITWS tests several parameters and selects the ones that provide the best results using a Grid Search. This method tests parameters in two ranges: first, in a larger one and then, after choosing one value from it, in a smaller scale range. For example, supposing that for a parameter i, the large range is represented by the values $L_i = \{..., 2^7, 2^9, 2^{11}, 2^{13}, ...\}$ and imagining that the selected value from the L_i is 2^9, the algorithm proceeds to search the final parameter value in the following range $S_i = \{..., 2^{8.5}, 2^{8.75}, 2^9, 2^{9.25}, 2^{9.5}, ...\}$. There are cases where it is required to estimate more than one parameter. In these scenarios, all the combinations of possible values and parameters are tested.

To test a possible value for a parameter, a classifier is modelled. Then, the performance of the later is adopted as metric to rank the best values. Typically, Grid Search is applied on the final classifier in order to tune it. However, on the presented approach, several classifiers will be handled during the process of selecting the subset of features that will be used on the final one. Moreover, performing a Grid Search for every considered classifier, is not an option. Doing that would increase the wrapper computational cost so much that even with a filter to decrease dataset dimensionality, and parallel computing techniques, it would take huge amounts of time to find a solution for an average number of features. Therefore, the solution is to estimate the parameters before starting the wrapper search.

Our proposed algorithm, performs a Grid Search on n randomly generated subsets before starting the wrapper and after the filter part. The best set of parameters for each subset count as a vote, and in the end, the parameters set with most votes is selected. In cases where the highest number of votes is the same in more than one set, the process generates n new random subsets and the test is reproduced for the tied values. This process is repeated until there are no more ties. Thus, obtaining a final value for the parameters.

Algorithm 2 demonstrates the process of estimating parameters for the SVM algorithm

on MITWS.

Algorithm 2 Grid Search

```
1: procedure GRID SEARCH(n, dataset, possibleValues )
2:     subsets ←generateRandomSubsets(n)
3:     large = True
4:     while True do
5:         votes ← ∅
6:         votes ←getVotes(dataset, subsets, possibleValues)
7:         if tiedVotes(votes) then
8:             possibleValues ←filterTiedVotes(possibleValues, votes)
9:         else
10:            if large then
11:                large = False
12:                possibleValues ←getSmallerRangeValues(votes)
13:            else
14:                break
15:        parameters ←getTopVote(votes)
16:
17:
18: procedure getVotes(dataset, subsets, possibleValues)
19:     votes ← ∅
20:     for subset ∈ subsets do
21:         bestScore = 0
22:         bestValue = 0
23:         for value ∈ possibleValues do
24:             score ←getSVMScore(dataset, subset, value)
25:             if score > bestScore then
26:                 bestScore = score
27:                 bestValue = value
28:         votes ← votes ∪ bestValue
29:     return votes
```

Chapter 4

Wrapper Search

The proposed feature selection approach was designed to adopt existing algorithms for most of the tasks that must be performed. However, the wrapper search is a new meta-heuristic which, together with the strategies for its parallel execution, makes it the main contribution of this work. It is the most complex part of MITWS and the functions used at this stage define its computational cost and ability to find good solutions. For the sake of understanding, the explanation is further divided into three sections: search strategy, subset evaluation, and successor generation.

4.1 Search Strategy

Despite the fact that several search strategies exist and have been successfully used on wrapper approaches, a novel meta-heuristic which is divided in two phases is introduced in this section. It aims to create a different strategy to search for solutions, while maintaining a structure that can be easily executed with multiple processing units. The proposed search organizes subsets as nodes on a tree, whose first level is composed by n starting subsets, each with a single feature not removed by the filter. Afterwards, subsets are tested and, if they are good candidates, they are further expanded into several more subsets. From now on, the words subsets and nodes are used interchangeably.

The innovative idea of the proposed search is to explore broadly different regions of the search space, looking for the areas of higher classification accuracy, and then focus on searching the local maxima in each region. Thus, the search strategy does not have

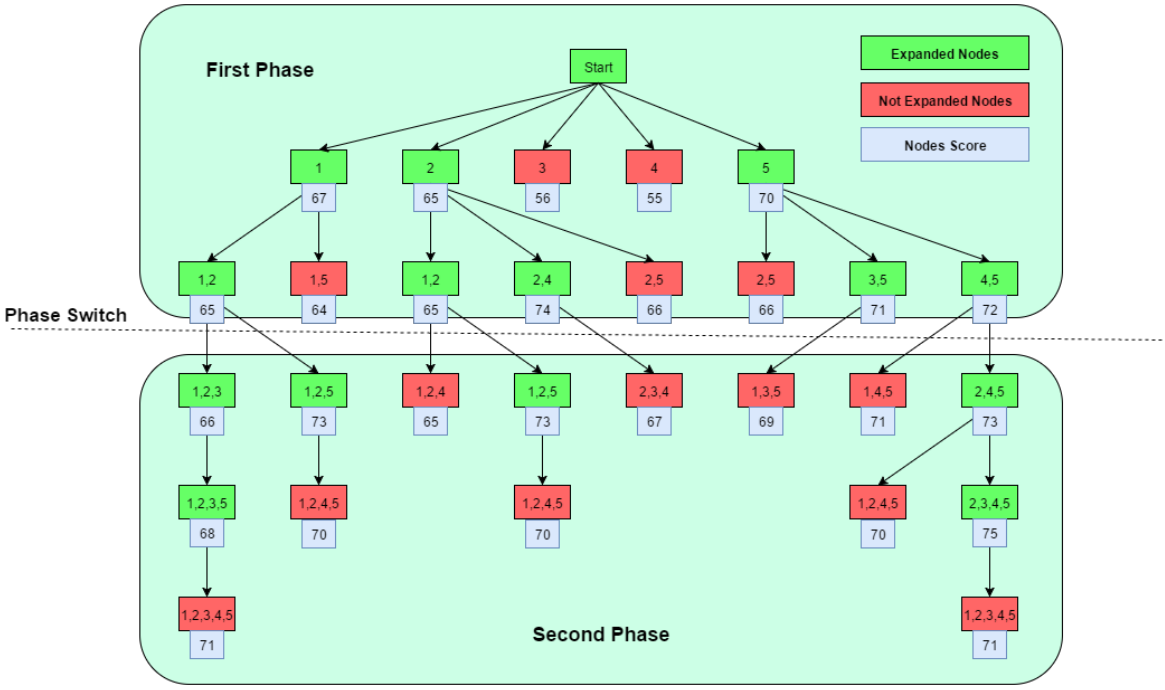


Figure 4.1: Example of the proposed wrapper search.

an uniform behaviour, but is divided into two phases. First, as many good solutions as possible are gathered. Then, they are improved up to the best score they can reach. The transition between phases takes place when subsets reach a certain number of features. Figure 4.1 illustrates an example of the implemented wrapper search.

In the first phase, nodes are explored using a breadth first strategy. The decision to expand a node is based on the distance from its score to the global best. In this step, a threshold is defined and nodes whose score differs from the global best by less than a threshold amount, have their successors generated. During the second stage, nodes are explored using a depth first strategy. In addition to that, they are expanded while they still improve the score of the subsets from which they originate. The search stops when there are no more nodes left to explore.

During the second phase in order to avoid excessive work, a mechanism probabilistically cuts subsets based on how distant their score is from the global best, according to the following table:

% Distance to global	Cut probability
$d < 0.5$	0%
$0.5 \leq d < 1.0$	25%
$1.0 \leq d < 1.5$	50%
$d \geq 1.5$	75%

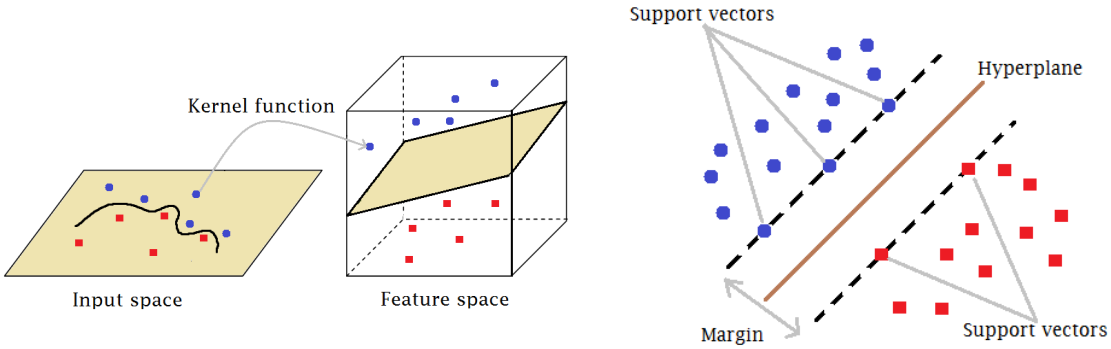


Figure 4.2: How Support Vector Machines work.

The mechanism executes every t seconds, where t is a value which can be user defined. Additionally, the first stage threshold that decides if nodes are expanded and the size at which stages switch, can also be configured. All these parameters have a great impact on the amount of nodes explored in the search. Thus, it is possible to control how restrictive the search is by changing them. Insights about their impact will be given in chapter 6.

4.2 Subset Evaluation

The idea of SVMs is to map vectors of features into higher dimension spaces. Then, finding hyper planes that separate classes, grouping the most points from the same class as possible. Hyper planes are defined using support vectors, which are subgroups of points from each class. Additionally, SVMs have several kernels to choose from. Each one defines a different way to map data into higher dimensions, consequently impacting the position of hyper planes. In order to select an adequate kernel, size and type of data should be taken into account [37]. The number of configurable parameters on SVMs, depends on the adopted kernel. Figure 4.2 illustrates the two crucial components of a SVM. The left one represents the mapping of the data into higher dimensions, and the other, is the hyperplane that divides classes.

The objective of using an SVM component in our strategy is to evaluate the goodness of a subset of features. In order to improve generalisation outside the training dataset, a cross-validation strategy is used at this stage [37]. This technique consists in defining a k number folds and dividing data examples into k groups. Then, for each tested subset, the classifier is trained with $k-1$ groups and the accuracy tested with the remaining. This process is repeated k times for each subset. In the end the algorithm

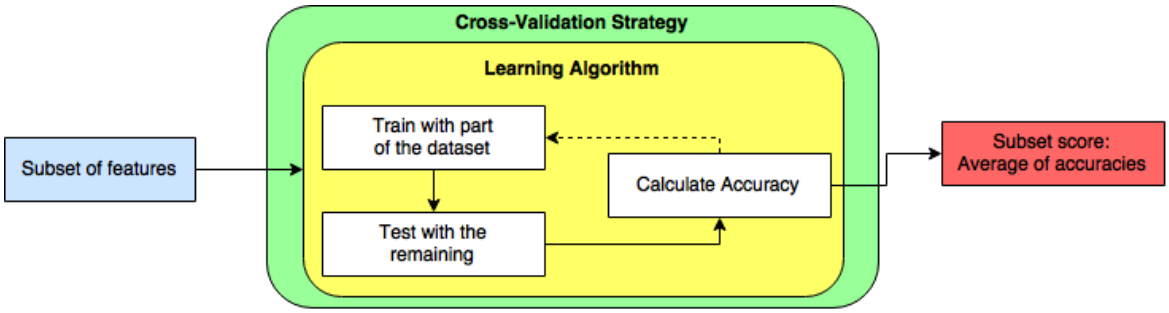


Figure 4.3: Subset evaluation using SVM.

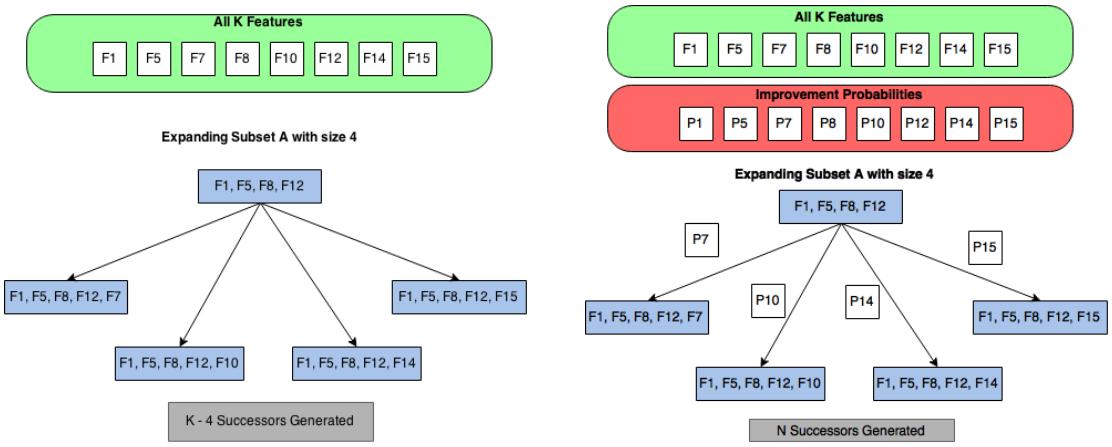
gets a score for the subset. This score is the average of accuracy obtained for each fold. This process is illustrated in figure 4.3.

4.3 Successor Generator

The idea is to expand a subset S_j into several new ones, where each of them is represented by S_j added with a single new feature that was not part of it. In order to decide how many successors of a subset are generated, two different options are provided.

The first strategy is to expand a subset S_j into as many successors as the number of features that are not part of it. For example, if S_j has n features and the total number of features on the wrapper is K, then expanding S_j will result in $K - n$ new subsets. One example of this type of expansion is given by figure 4.4a.

An alternative approach, each feature can be added with a specific probability, according to a likelihood of improving the evaluation score, estimated in a pre-processing step described below. By doing so, our approach increases the likelihood of features with high improvement score being added to new subsets. Figure 4.4b illustrates this process. There, each new subset has a P_i probability of being generated. The likelihood of a certain feature contributing to an improvement in the evaluation criteria is estimated in a pre-search phase. The procedure starts by generating r random subsets, and evaluates each one of them using the subset evaluation procedure. Then, the improve capability of F_i is assessed by either adding or removing it from each subset and checking if the score of the subset improved. In the end, the number of times I_i, that a feature improved a subset is obtained and used to calculate the likelihood of improvement using equation 4.1. These values are then used when the



(a) Successor generation without probabilities (b) Successor generation with probabilities

Figure 4.4: The two types of successor generation on the wrapper search

wrapper search decides to expand a node.

$$P_i = \frac{I_i}{r} \tag{4.1}$$

Since the second approach does not expand a subset into all possible successors, it explores a smaller number of nodes. Thus, making the whole wrapper strategy less computational costly. In chapter 6, the impact of both approaches on the final solution and number of subsets tested will be discussed.

4.4 Work Repetition

From the previous section, it was clear that our successor generator function may spawn subsets that have already been tested before. Moreover, from figure 4.1 it is possible to note that the search strategy does not handle the problem. Since the function that evaluates subsets is deterministic, testing a subset more than once is a waste of computations. Additionally, repeated work is a serious issue to the execution time of the wrapper search.

To solve this duplicated work problem, we added an efficient data structure, in this case an hash table. Our strategy consists in producing a unique identifier for every generated subset during the search, and use it as the hash key to save the associated subsets in the hash table. Then, every time the successor generator procedure is

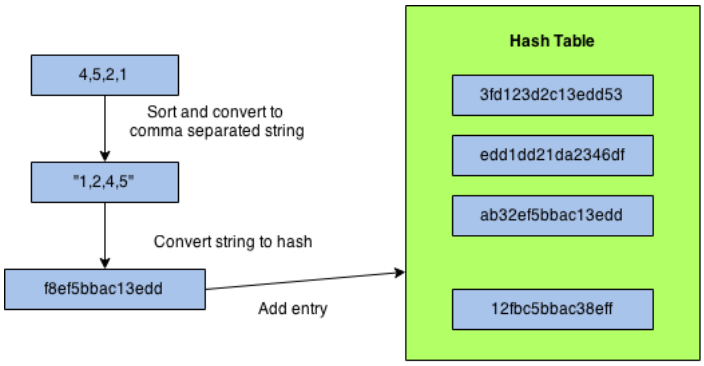


Figure 4.5: Mapping a subset into the hash table

executed, a very efficient table look up checks which of the new spawned subsets have already been generated before. Those that have been generated before are discarded while the new ones have its unique identifier added to the structure.

The hash value is obtained by first sorting the features in a subset, then by adding a comma separator for each pair, and then by converted the whole subset into a string. Finally, any possible hash function can be used to create the entry. The mentioned process is illustrated by figure 4.5.

4.5 Overview of the Wrapper Search

Let us start by explaining the need for a new wrapper strategy and then, introduce the novelty in our strategy. As addressed during the literature review in chapter 2, there is no best method to solve all feature selection problems. This issue is directly related to the fact that a search that guarantees finding the optima solution in a practicable amount of time does not exist. Therefore, proposing new strategies is useful to tackle the problem.

The novelty of the wrapper is related to the developed search. Taking a look at the other components, using a SVM to evaluate subsets of features is a standard procedure that has been used in other cases [62, 40]. Related to the successor generation functions, one strategy consists in applying a forward operator introduced in section 2.2.1. The other one uses a similar methodology of feature selection algorithms that ranks features, and then use the rank in order to increase the likelihood of better features being selected. This is the case of the Las Vegas Incremental algorithm [42]. On the other hand, the presented search combines characteristics from different searches in

order to, to the best of my knowledge, create a completely innovative approach. For example, the way subsets are organized during the search reassembles the strategy used in a Branch and Bound algorithm [53]. The sequential forward wrappers [24] were the inspiration to define the initial starting points. The idea of combining a breadth search with a depth search came from the work of Zhou et al. [64]. Nevertheless, the cited work used the methodology to find the treewidth of graphs and such approach was never applied to solve the problem of feature selection. Combining all the mentioned components and adapting some of them, resulted in our wrapper proposal, which is aimed to explore the feature space in a unusual and efficient way.

To conclude the discussion on the new wrapper approach, we present how the three mentioned components interact. The main idea is to have all the features that pass the filter defined as starting points of the search. Then, the wrapper will iterate through all of them, testing each with the evaluation function and expanding the ones that have promising scores. During the expansion of a subset, the hash table is used to remove the generated subsets that have already been tested, and to store the ones that did not. At the end of the process, all truly new subsets are added to the worklist. The whole process is demonstrated by the algorithm 3.

Algorithm 3 Search Strategy

```
1: procedure SEARCH(size, W, data, hashTable, probs, timer )
2:     lastStage ← False
3:     while W ≠ empty do
4:         s ← REMOVELAST(W)
5:         if SIZE(s) ≥ size then
6:             lastStage ← True
7:         score ← SVMCLASSIFICATION(s, data)
8:         if WORTHEXPAND(score, s, lastStage) then
9:             newN, hashTable ← GENERATESUCCESSORS(s, hashTable, probs)
10:            UPDATEGLOBALSCORE(score)
11:            if lastStage then
12:                W ← W ∪ newN
13:                W ← CHECKCUTMECHANISM(timer, W)
14:            else
15:                W ← newN ∪ W
16:
17: procedure GENERATESUCCESSORS(subset, hashTable, probs)
18:     newSubsets ← GENERATESUCESSORSSUBSET(subset, probs)
19:     newWork = ∅
20:     for newSubset ∈ newSubsets do
21:         if newSubset ∉ hashTable then
22:             newWork ← newWork ∪ newSubset
23:             hashTable ← hashTable∪ GETHASH(newSubset)
24:     return newWork, hashTable
```

Chapter 5

Parallelized Computing Approach

Parallel computing relies on the principle that large problems can often be divided into smaller ones, which are then solved concurrently [12]. There are three aspects to consider in order to check if a sequential algorithm is a good candidate for parallelization [11]:

1. **Easy Partitioning.** Refers to the difficulty in dividing the problem into several tasks. Algorithms based on a main loop usually are easy to decompose in several tasks. On the other hand, purely sequential code is not.
2. **Independent Partitioning.** Indicates the dependency between the partitioned task execution. As more dependent they are, more communication needs to be exchanged between processing units. Thus, increasing the challenge of achieving good parallel performance.
3. **Easy Load Balancing.** Specifies how easy it is to equally divide the amount of work between the processing units. In cases where the balancing is not properly achieved, processes stay idle for a significant amount of time, negatively impacting the parallel performance gain.

in this chapter the parallelization of the MITWS algorithm is thoroughly discussed.

5.1 MITWS Parallelization

The computational cost of the MITWS algorithm is highly related to the cost of the wrapper. Although the intensiveness of this part can be manipulated by the user

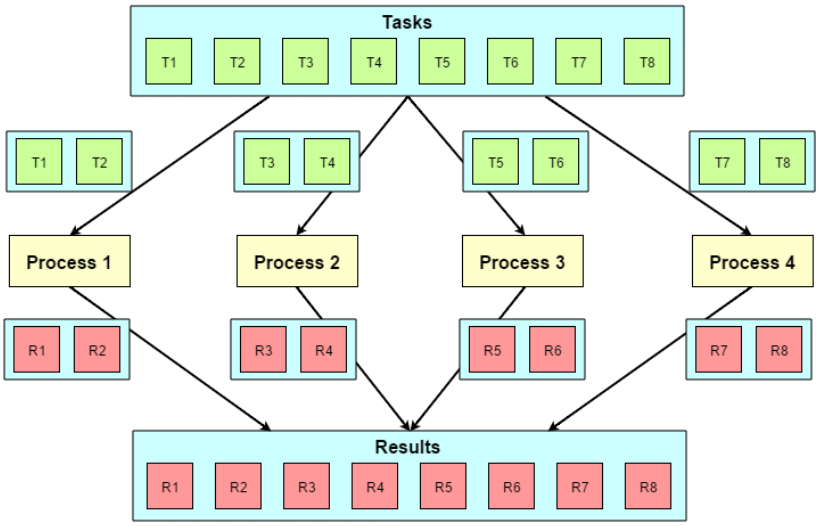


Figure 5.1: Parallel scheme for UCAIM, Filter, and Grid Search.

defined parameters, in most cases a massive number of subsets will still be explored. Therefore, finding a way to realise the wrapper search in parallel, will help in reducing the computation time.

Regardless of the impact of the UCAIM, Filter, and Grid Search parts on the overall execution of MITWS, the presented algorithm also runs them in parallel, for performance purposes. Table 5.1 introduces how parallelization of each part is achieved with respect to the three parallel aspects previous mentioned.

Table 5.1: Parallelization of the first three parts of MITWS

Procedure	**Task Partitioning**	**Task Dependency**	**Load Balancing**
UCAIM	Estimate the discretization scheme for an individual feature	Features discretization schemes are independent of each other	Features equally divided among all processing units
Filter (MI)	Calculate MI score for a single feature	MI score of a feature does not affect any other	Each processing unit receives the same number of features
Grid Search	Evaluate the best parameters for an individual random subset	The best parameters for a subset are independent from others	Random Subsets are equally distributed among processing units

For each part, the problem was easily divided into smaller tasks which are independent from each other. As a result, processes only needed to exchange messages to receive work and send results. In addition, since the time required to complete a task is nearly the same for all of them, work balancing was achieved by equally dividing tasks among the processing units. The parallel scheme for each part is represented in figure 5.1.

Due to the nature of the problems, the parallelization of these three parts was achieved

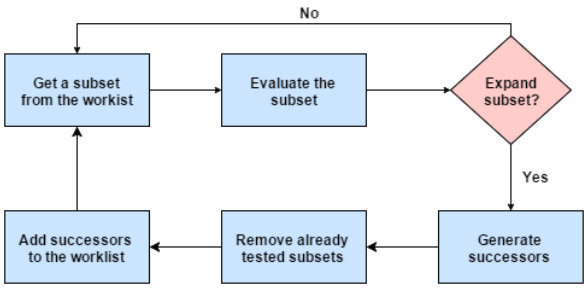


Figure 5.2: Workflow of a single wrapper task

with no major difficulty. On the other hand, the wrapper presented some challenges which will be discussed in the next section.

5.2 Wrapper Parallelization

In the context of our wrapper strategy, a task can be defined in four main steps: get a subset from a worklist, evaluate it and decide to either expand it or not. In the expanding procedure it is necessary to generate new subsets and remove those that have been already tested. The last step is adding the remaining subsets to the worklist. Figure 5.2 illustrates this workflow.

By contrast to the other parts of MITWS, the wrapper tasks are not totally independent. The process of getting a subset and evaluating it, can be executed without affecting or depending on other tasks. However, the decision to expand, during the first phase of the wrapper, is based on the global best score, which results from other tasks. Moreover, the procedure to remove subsets that have already been tested has to take into account subsets explored across multiple processing units in order to avoid repeated work. Therefore, the parallel wrapper requires information to be exchanged between processes.

The strategy to avoid repetition of work in the wrapper search relies on keeping a constantly updated global hash table with all the subsets tested. Keeping such structure always up-to-date in a parallel environment may become computationally costly in terms of performance mainly because every generated subset must be added as soon as possible to the hash table. Therefore, communication between processes is often required. Additionally, if the repeated work problem is not successfully removed, then the parallel performance of the algorithm is drastically reduced thus, making it the main challenge of the proposed wrapper parallelization. In order to

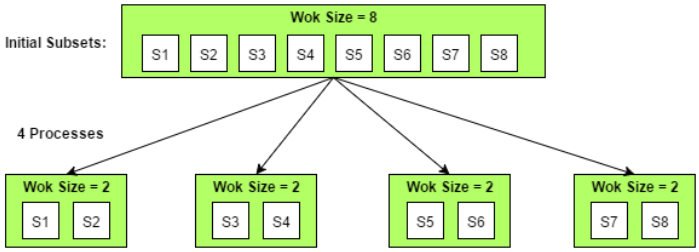


Figure 5.3: Work initialization on both strategies

tackle this problem two strategies were implemented and will be discussed in the following sections. The idea is to demonstrate two possible ways to obtain good parallel performance on the presented wrapper on two distinct environments: distributed and shared memory.

On both strategies, a local worklist is defined on every process. At the beginning of the search, the individual features that reach the wrapper are transformed into subsets and equally distributed by processes' worklist. This initialization is represented in figure 5.3. During the search, processes concurrently test subsets from their own local worklist using the previous definition of a task. Every time a process expands a subset, the new work generated, is added to its worklist. The part that removes subsets that have already been tested is different for each strategy, therefore it will be discussed during their introduction.

5.3 Master-Slave Strategy

The first implemented strategy is based on the Master-Slave (MS) paradigm for parallel programming [5]. This paradigm commonly involves two sets of processors: a unique master and several slaves. The former is responsible for pre and post processing tasks. On the other hand, the latter are in charge of the actual execution of work. Contextualizing with the wrapper parallelization, the idea of the strategy is to have slaves testing subsets, while the master keeps track of the global information such as the hash table and the best score. Additionally, the latter will be in charge of helping slaves remove subsets that have been already tested from their worklists, and give them information about the best score during the search.

In the original MS approach, the slaves communicate restrictedly with the master, and usually only two messages are required during the whole process. In the first one, the master sends the work to the slaves, and the last one is where the slaves send

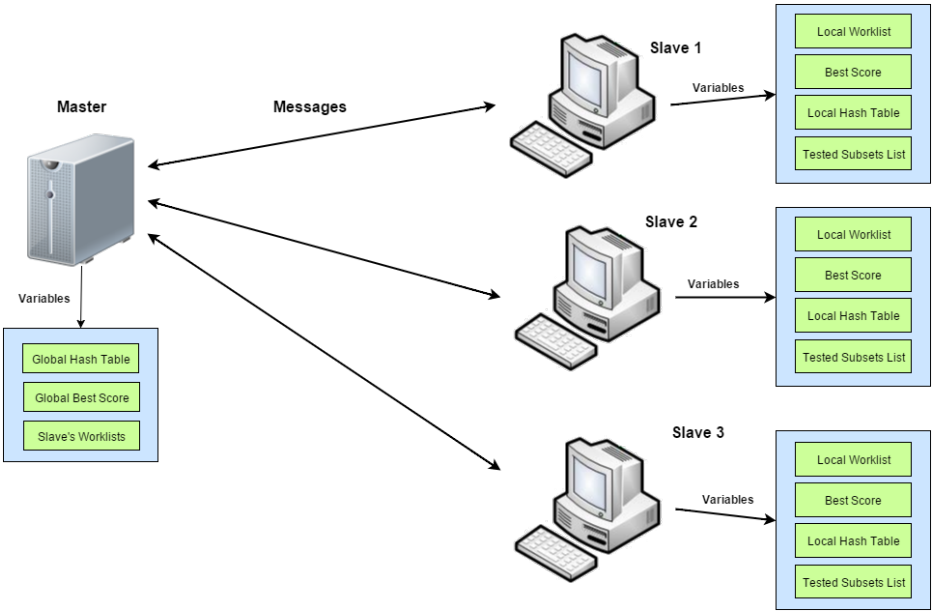


Figure 5.4: Parallel Scheme for Master-Slave Strategy

the results back. This strategy would not fit the proposed wrapper because of the requirements of having a constantly updated hash table and best global score. On the presented parallel strategy, slaves are still restricted to communicate with the master, however they do it often. The remaining of the section will thoroughly discuss the parallel strategy.

5.3.1 Strategy Setup

For the sake of understanding lets start by defining the information available on both type of processes and then, address their workflows. It is important to highlight that the information, as well as the behaviour of the processing units addressed in this chapter, is related to the parallel part, and in order to keep the discussion as simple as possible, variables required for the standard wrapper search introduced in chapter 4 are left out.

Each slave has:

- **Worklist:** stores the subsets that a slave will test.
- **Local hash table:** keeps track of the subsets already tested on the slave.
- **Best score** stores the best score known for the slave.

- **Tested subsets:** list that collects the tested subsets between each communication with the master.

The master has:

- **Global hash table:** gathers the hash values of every subset tested during the search.
- **Global best score:** records the best global score.
- **Slave's worklists** stores the current worklist of each slave.

The overall scheme is illustrated by figure 5.4.

5.3.2 Slave Workflow

The goal of the slaves is to process every subsets explored during the search. To achieve this, they are initialized with different sets of subsets which are stored in their *Worklist*, and proceed to iteratively test each element of the list. Every time a slave generates new work, it is added to its own *Worklist*. Additionally, every slave keeps track of all the subsets they generate (in the *Local hash table*), and test (in the *Tested subsets*).

As far as communication is concerned, each slave exchanges messages often with the master. From the slave's point of view, the messages represent requests sent to the master and the respectively answer. The two types of request are the following:

1. The slave requests the master to remove subsets that have already been tested by other slaves from its *Worklist*.
2. The slave runs out of work and requests the master for more.

The slave's workflow is represented by a loop where each iteration consists of all the necessary steps to test a subset. This loop, as well as the necessary steps, will be discussed below.

Main Loop

The slave's main loop iterates until a break condition is reached. At each iteration, it starts by checking if the *Worklist* still has subsets. If it does, then the slaves proceed

to execute the previous definition of a parallel task for a single subset (section 5.2). In the case it does not, the slave communicates with the master to request work. If the master sends back an empty *Worklist* then the slave ends its execution.

The last part of the iteration is to check how much time has passed since the last time the slave communicated with the master. In case the elapsed time is greater than *CommRate*, then the slave will check if the master is available to remove the duplicated work from its *Worklist*. If the master removes the repeated work, the elapsed time and the list with all the tested subsets since the last communication (*Tested subsets*), are restarted. The *CommRate* variable is user defined and allows to control the rate at which the slave tries to communicate with the master. In chapter 6, this issue will be discussed in more detail. Algorithm 4 demonstrates the behaviour of the loop executed by each slave.

Algorithm 4 Slave's Main Loop

```
1: procedure ProcessWork(worklist, commRate)
2:     localHT, testedList ← ∅
3:     bestScore = 0.0
4:     elapsedTime = 0
5:     lastStage = False
6:     while True do
7:         if isEmpty(worklist) then
8:             worklist, bestScore ← RequestWork(bestScore, testedList)
9:             if isEmpty(worklist) then
10:                return
11:            testedList ← ∅
12:        subset ← RemoveLast(worklist)
13:        testedList ← subset ∪ testedList
14:        worklist, localHT ← ProcessTask(subset, localHT)
15:        elapsedTime ← getElapsedTime( )
16:        if elapsedTime > commRate then
17:            com, worklist, bestScore ← RemoveDuplicates(worklist, bestScore, testedList)
18:            if com then
19:                elapsedTime = 0
20:                testedList ← ∅
```

In order to avoid pseudo code redundancy, the *ProcessTask* function, refers to the steps of testing a subset represented by the lines 5 to 15 from the code in algorithm 3. However, instead of using a global hash table, which the slaves do not have access to, it uses the *Local hash table*.

Communication

There are two operations that require communication with the master. From the slave's point of view, these communications do not require any computations. Ba-

sically, they consist in sending a request with some information and waiting for the return of variables. Independently of the request, the slave always sends the same parameters: *Worklist*, *bestScore*, and *testList*. The usefulness of all these parameters will be addressed while discussing the master's workflow. Moreover, the received message always consists of two variables. The first one is the *Worklist* and the other is the global best score known by the master. The later is required to provide slaves with the awareness of the global best score.

The first request is the *RequestWork*, this consists in a procedure that sends the empty *Worklist* and expects to receive back some work. If the slave does not receive any, this is interpreted as the signal to terminate the search. Additionally, the slave has to stay idle until a response is returned.

The RemoveDuplicates function is a bit more complex. Calling this procedure does not imply communication between slave and master. The first thing the slave does is to check if the master is available using the *gotMessage* function. If so, the slave sends the request and it receives a *Worklist* without repeated work. Conversely, if the master is not available, the slave will not waste time waiting for it, and immediately returns to process another iteration of the loop. As a quick note, it is important to notice that if the communication happens, the *testList* and the *elapsedTime* are restarted. Algorithm 5 demonstrates the *RemoveDuplicates* procedure.

Algorithm 5 Remove Duplicates Procedure

```
1: procedure RemoveDuplicates(worklist, bestScore, testList)
2:     com = False
3:     if gotMessage then
4:         worklist, bestScore ←sendInfo(worklist, bestScore, testList)
5:         testList ← ∅
6:         com = True
7:     return com, worklist, bestScore, testList
```

5.3.3 Master Workflow

The goal of the master is to control the slaves and all the information that requires global awareness. Its first task is to start every slave. This is achieved by sending the required parameters along with a different *Worklist* for each slave. Then, the master proceeds to answer slave's requests, communicating with them on a round-robin fashion [28].

As previously mentioned, there are two types of requests that trigger different master's actions. The *RemoveDuplicates* procedure requires examining all the subsets

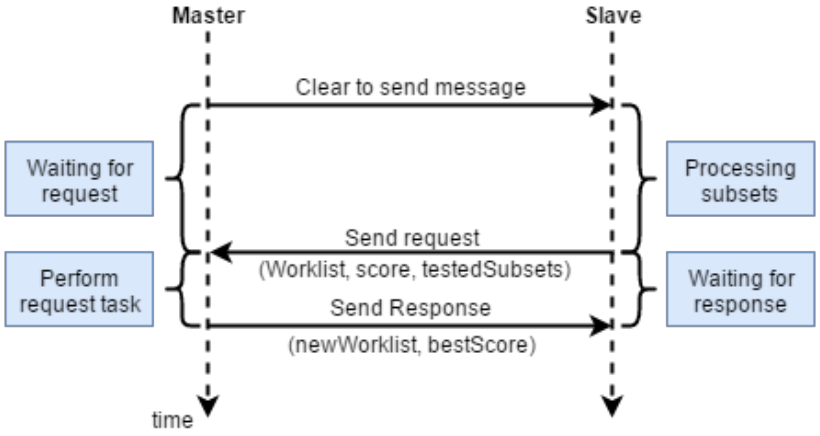


Figure 5.5: Master-Slave communication flow

of a *Worklist*, removing those that have already been processed or that are already scheduled to be tested on a different slave. In the *RequestWork* case, the master needs to get part of a *Worklist* from another slave, and send it back to the one which the request came from. The work of the master will be thoroughly discussed below.

Main Loop

After initiating all the slaves, the master proceeds to the main loop, where it iterates until the wrapper search ends. Inside this loop, there is an inner cycle that iterates through all slaves. At each iteration, the master communicates with a different slave. This process begins with the sending of a message to the selected slave, which means that the master is waiting to communicate with it. Then, the master will wait until it gets a request with three parameters: *Worklist*, *Best Score*, and *Tested Subsets*. Depending on the content of the *Worklist*, the master performs a different task. However, the communication will always end with the master sending a *Worklist* and the *Global Best Score* to the slave. To make this part clear, the flow of messages is illustrated by figure 5.5.

Independently of the task performed, there are some steps that the master has to perform. The first one is to update the *Global Best Score* with the received *Best Score*, in case the later is greater than the former. This step guarantees that the best score discovered during the search is always updated. The master also has to update the *Global Hash Table* with the *Tested Subsets* received at each communication. This is crucial to keep track of all the tested subsets while the search is executing. Additionally, after responding to the request, the *Slave's Worklist* has to be updated

with the new *Worklist* sent to the slave. Despite not being always up-to-date, due to the round robin polling of slaves by the master, the *Slave's Worklist* is important to remove work repetition. The subsets stored in the *Global Hash Table* are subsets already tested by slaves. Subsets that are already scheduled to be processed in the slave are left out of this structure. Therefore, in order to efficiently remove the duplicated work from a slave it is necessary to check if the subsets in its *Worklist* are already scheduled to be processed in another slave's *Worklist*.

The search ends when all slaves run out of work at the same time. This is determined by the inability of the master in finding work to share when it is requested. Algorithm 6 illustrates the main loop of the master.

Algorithm 6 Master's Main Loop

```
1: procedure ManageSlaves( )
2:     bestScore = 0.0
3:     hashTable ← ∅
4:     search = True
5:     while search do
6:         for slave ∈ slaves do
7:             SendMessage(slave)
8:             worklist, score, testedSubsets ← ReceiveMessage
9:             if score > bestScore then
10:                bestScore = score
11:            hashTable ←addToHashTable(testedSubsets)
12:            if empty(worklist) then
13:                newWorklist ←ShareWork(slave, slavesWorklist, hashTable, bestScore)
14:                if empty(newWorklist) then
15:                    search = False
16:                    Break
17:            else
18:                newWorklist ←RemoveDuplicates(worklist, slavesWorklist, hashTable, slave)
19:            SendWork(newWorklist, bestScore, slave)
20:            slavesWorklist_slave ← newWorklist
```

This concludes the master's activity during the main loop, the remaining of the section will address the different tasks on the manager.

Remove Duplicated Work

Starting by the *RemoveDuplicates* function which is the simpler of the two. Given a *Worklist* of a slave, this function removes all subsets that are already either on the *GlobalHashTable* or in the *Worklist* of any other slave. Algorithm 7 illustrates this procedure.

Algorithm 7 Remove Duplicates Procedure

```
1: procedure RemoveDuplicates(worklist, hashTable, slavesWorklist)
2:     newWL ← ∅
3:     for subset ∈ worklist do
4:         hashValue ←getHash(subset)
5:         if hashValue ∈ hashTable then
6:             Continue
7:         add = True
8:         for list ∈ slavesWorklist do
9:             if notSlavesList(slave, list) then
10:                if hashValue ∈ list then
11:                    add = False
12:                    Break
13:        if add then
14:            newWL ← newWL ∪ subset
15:    return newWL
```

Share Work

The *ShareWork* function is called when the *Worklist* received from a slave is empty. When this happens, the master proceeds to communicate with the slave which he sees as having the most work. The concept of "thinking" is important, that is because the master estimates the amount of work on the slaves, using the *Slaves worklists*. However, this list is only up-to-date after the communication with the respective slave.

If the master communicates with another slave that also has no work, it adds that slave to a list of slaves requesting work and proceeds to enquire the next one. Finally, when a slave that has work is found, its *Worklist* is equally divided among itself and all the other slaves that are in the request work list. Before dividing the *Worklist*, its duplicated subsets are eliminated using the previous *RemoveDuplicates* function.

At the end of the procedure, all slaves that communicated with the master receive a new *Worklist* and the best score. Additionally, the master also has to update their current work in the *Slave's worklist*. In the case there are no slaves with work to share, the master sends an empty list to all as the signal to end all computations. Algorithm 8 illustrates this procedure.

In order to decrease the length of the presented pseudo-code, we intentionally did not represent some less important parts. For every received message (line 7) during the procedure, the *bestScore* needs to be updated if the new value is higher, and the *testedSubsets* must be added to the *hashTable*.

This ends the discussion of the master-slave strategy, one of the two options to achieve

Algorithm 8 Share Work Procedure

```
1: procedure ShareWork(slave, slavesWorklist, hashTable, bestScore)
2:     shareWork ← ∅
3:     while shareWork == ∅ do
4:         shareSlave ←GetSlaveWithMostWork(slavesWorklist)
5:         SendMessage(shareSlave)
6:         requestList ∪ shareSlave
7:         worklist, score, testedSubsets ← ReceiveMessage
8:         if empty(worklist) then
9:             slavesWorklist_shareSlave ← ∅
10:            if length(requestList) == length(slavesWorkLlist)−1 then
11:                SendEmptyList(requestList)
12:                return ∅
13:        else
14:            newWorklist ←RemoveDuplicates(worklist, hashTable, SlavesWorklist)
15:            dividedWork ←DivideWork(newWorklist, requestList)
16:            for rs ∈ requestList do
17:                SendWork(dividedWork_rs, bestScore, rs)
18:                SlavesWorklist_rs ← divideWork_rs
19:            return dividedWork_slave
```

parallelism on MITWS conferred in this thesis. Proceeding to the analysis of the other strategy.

5.4 Shared Memory Strategy

The idea of the second presented strategy is to take advantage of a shared memory environment for interprocess communication. In such conditions, processes have simultaneous access to the same memory zones, thus providing a less costly way to exchange information between processes, as well as the advantage of avoiding the use of redundant copies of information [54].

Programs that correctly take advantage of these characteristics are usually very efficient and scalable. However, programming them can be quite troublesome. The shared memory paradigm introduces the challenge of keeping memory coherent. If two processes try to operate on the same memory zone at the same time, the memory will most likely become incoherent making its values unpredictable. This is exemplified by a simple program: considering a shared variable called *count* and one program which uses 10 threads that concurrently want to add 1 to the shared variable. If *count* starts with the value 0, then in the end of the program the expected value would be 10. However, since there is no mechanism that controls the access to the shared variable, the value of *count* is unpredictable. This happens due to the concurrent

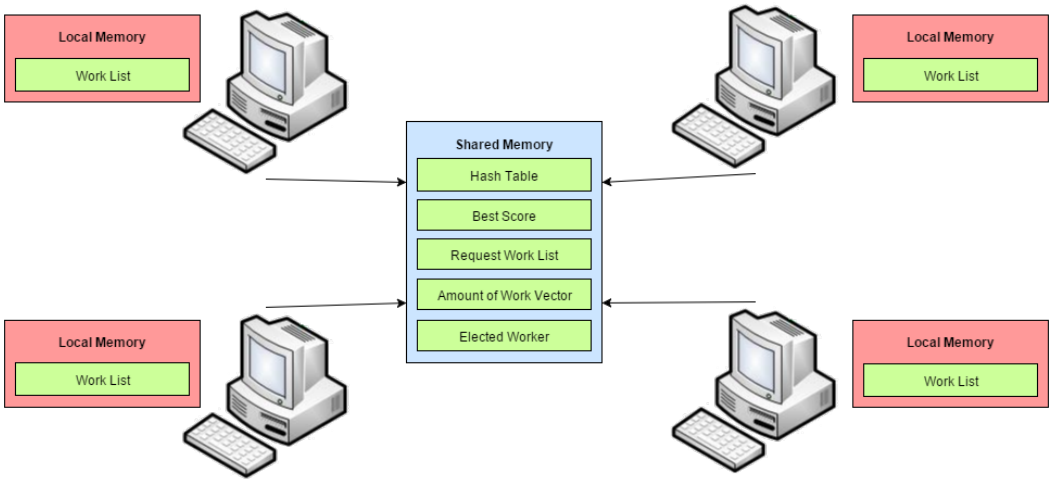


Figure 5.6: Parallel scheme for the shared memory strategy

writing operations on the shared variable [54].

In the case of the parallel wrapper, memory incoherence can lead to repeated work or even loss of the best score. Therefore, this is an issue that must be addressed. One way to guarantee memory coherence is to use mutual exclusion mechanisms. One solution is to use locks, which prevent more than one process to operate on the same memory zone concurrently [54]. Despite the fact that this sounds like a good solution, it is not the ideal one. These mechanisms introduce an overhead to the program which could drastically decrease the performance gains obtainable through parallelization. For the sake of understanding, before going into more details about these issues, the strategy setup will be explained.

5.4.1 Strategy Setup

Since this strategy does not use a master-slave paradigm, the processes that compute work are called workers instead of slaves. In the same way as the first implemented strategy, each worker has its own *Worklist* which is initialized with some subsets. This is the only variable that is local to them. All the remaining information is located on the shared memory. Therefore, the variables that can be accessed by all workers are:

- **Hash table:** used to record the hash values of the already generated subsets.
- **Best score:** stores the best score during the wrapper search.
- **Request Worklist:** keeps track of the workers needing work.

- **Amount of work vector:** each slot represents the size of a worker *Worklist*.
- **Elected worker:** every time some worker requests work, this variable records the worker selected to send it.

The parallel scheme is illustrated by figure 5.6.

5.4.2 Memory Coherence

The hash table located in the shared memory is key to avoid the problem of repeated work. Having every worker constantly updating the hash table and search in it for repetitions is not a good idea. In the same way as the previous *"count"* example, the workers would start writing in the same memory spaces. Thus, the hash table would become incoherent, probably leading to the loss of entries which would result in testing repeated subsets.

As previously discussed, mutual exclusion mechanism could be used to guarantee the coherence of the memory. To understand why this is not a viable solution to the parallel wrapper let us go through some facts. A mutual exclusion solution can be attained by the use of a lock. While a worker is in possession of the lock and is operating on the shared memory, all the others do not have access to it. The problem, is the fact that while the other workers are waiting for the memory zone to become free, they are wasting computational time. Moreover, the rate at which the workers require access to the variables is also relevant. Considering write operations in the *Hash Table*, every time a worker generates a new subset, it has to add its hash value to the memory. This happens very often and as a result, workers would frequently require access to the memory zone. Therefore, they would waste too much time waiting for the access, which consequently would drastically decrease the parallel performance. As a matter of fact, the rate is dependent on several aspects such as: number of features, size of the dataset, and number of workers. However, the access rate to the shared memory will be high in most cases. Due to these reasons, mutual exclusion mechanisms were not used to control access to the *Hash Table*, and a different approach was implemented.

Instead of using a unique *Hash table*, n hash tables are created, where n is the number of workers in the wrapper search execution. Each hash table is associated with a different worker, which is the only one allowed to perform write operations in it. However, any worker is granted access to read from any hash table at any given time. Therefore, workers are capable of storing in their hash table the hash of subsets

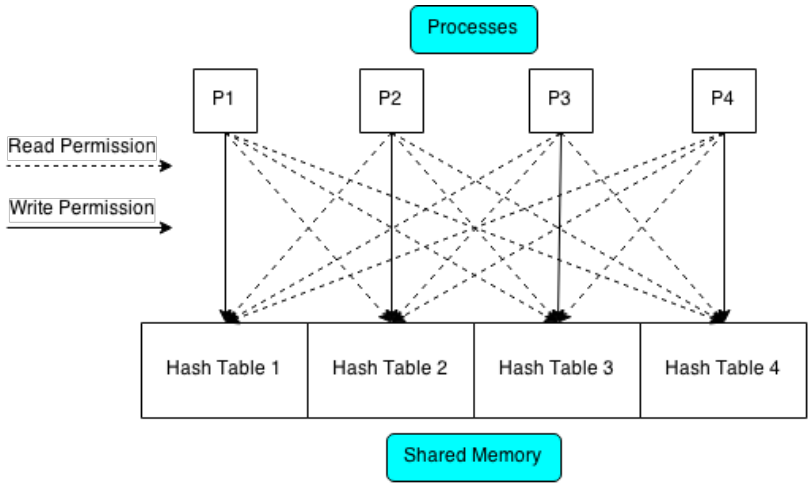


Figure 5.7: Hash Table Partitioning

they tested and checking all hash tables for repetitions without having to wait for access. Due to the fact that there are no concurrent write operations in any particular hash table, memory coherence is guaranteed. To understand this, it is important to highlight that reading does not change the memory, therefore, workers have free access to read all the hash tables.

The strategy is not bulletproof with respect to always guaranteeing that the hash table is up-to-date. There is a scenario where worker i could incorrectly declare subset s as new work. Consider that s was already generated on worker j, and j is on the process of writing the new subset to its hash table. While j is operating on the memory, i checked the hash table of j and did not find s there because j was still updating the memory. Then, i would declare s as a new subset and in fact, it is not. Nevertheless, due to the size of the search space of our wrapper, this is not a problem. The likelihood of this situation happening is quite low, and if in fact happens, the impact on the parallel performance is barely noticeable. Processing a low amount of repeated subsets is not an issue on such large scale problems. Conversely, avoiding wasting computational time in waiting for access to variables is a huge advantage. Thus, the present strategy is more beneficial than using mutual exclusion

The mentioned approach only refers to the *Hash table* zone, regarding to other variables on shared memory. The *amount of work vector* behaves with the same way as the *Hash table*. Each worker has its own slot, where only the owner can modify its value. The other variables: *Best score*, *Request work list*, and *Elected worker* use mutual exclusion. This mechanism was introduced due to two main reasons:

1. The access rate to these variables is much lower, as a result, wasting computational time is not a problem.
2. All variables have an high impact on the search and having some kind of incoherence on them could imply severe consequences.

By contrast to the introduced approach on the *Hash Table*, the mutual exclusion always guarantees the workers to read the most updated value of the variables.

5.4.3 Workers Workflow

After describing the problems with using shared memory and explaining the setup for the strategy, it is time to address the workflow of the workers. On this second strategy, every process has the same function. Their workflow consists in processing subsets from their worklists, and verifying when they are required to send work to any other worker, or even request it themselves. All these steps will be discussed in more detail in the remaining of the section. Once again, only aspects related to the parallel strategy are illustrated on its presentation.

Main Loop

In the same way as the first strategy, every worker has a loop which iterates through the subsets in their *Worklists*. At the beginning of each iteration, a worker has to verify two conditions. The first one is to check if it has work left. If not, the worker adds itself to the *Request work list*, determines the worker with most work, and selects it, using the *Elect work* variable, to send part of its *Worklist*. If the worker receives an empty list from the request, then it ends the search. The other test consists in verifying whether the worker has been selected to send part of its work to any other worker. If it did, then it calls a procedure that will divide its *Worklist* among all the workers that do not have any subsets left.

Afterwards, the iteration proceeds to test a subset that is removed from the *Worklist*. Each subset is processed using the previous definition of a parallel task (section 5.2). Finally, the last step, consists in updating the worker's amount of work in the global variable *Amount of work vector*. Algorithm 9 illustrates the behaviour of the main loop of a worker.

Once again, to avoid pseudo-code redundancy, the execution of a parallel task for

Algorithm 9 Worker's Main Loop

```
1: procedure PROCESS WORK(worklist)
2:     while True do
3:         if ISEMPTY(worklist) then
4:             worklist ← REQUESTWORK(ownWorker)
5:             if ISEMPTY(worklist) then
6:                 BREAK
7:         if NEEDTOSENDWORK( ) then
8:             worklist ←SENDWORK(worklist, ownWorker)
9:         subset ← REMOVELAST(worklist)
10:        PROCESSTASK(subset)
11:        amountWork_ownWorker ←LENGTH(worklist)
```

a subset was replaced by a function call, *ProcessTask*. However, there are several differences that have to be pointed out.

To start with, the search uses the global *Best Score* variable to check if subsets are worth expanding. Reading the variables does not cause memory coherence problems, however to update the global *Best score* every time a new highest score is found, the mutual exclusion mechanism has to be used. This process is established with algorithm 10.

Algorithm 10 Update Best Score

```
1: procedure UPDATESCORE(newScore)
2:     if newScore > bestScore then
3:         GETLOCKSCORE( )
4:         if newScore > bestScore then
5:             bestScore = newScore
6:         RELEASELOCKSCORE( )
```

It is worthy to point out that double checking if the new score is higher than the previous best is not a mistake. This was used to guarantee that while the worker was waiting for the lock, the *Best Score* did not change to a value higher than the *newScore*.

During the process of generating successors, each worker is in charge of controlling their repeated work. Due to the fact that on the presented strategy the hash table is divided into several ones, a new procedure was added to the *ProcessTask* part. Basically, this function checks if every new generated subset is already present in any hash table and those that are not, are added in the hash table of the worker that generated them. This is demonstrated by algorithm 11.

Besides the main loop, the *Request Work* and *Send Work* are the other two functions that will be explained in more detail.

Algorithm 11 Check For Repetition

```
1: procedure GenerateSuccessors(subset, ownWorker)
2:     newSubsets ← GenerateSuccessorsSubset(subset)
3:     newWork ← ∅
4:     for newSubset ∈ newSubsets do
5:         add = True
6:         for worker ∈ workers do
7:             if newSubset ∈ hashTable_worker then
8:                 add = False
9:                 Break
10:        if add then
11:            newWork ← newWork ∪ newSubset
12:            hashTable_ownWorker ← hashTable_ownWorker ∪ getHash(newSubset)
```

Request Work

Every time a worker runs out of work, it has to request it from other workers. There are several steps in order to do that. The first one is to add itself to the *Request work list*. Then, it has to select the worker with most work, using the *Amount of work vector*, and change the value of the *Elected worker* to the selected one. When there are no workers left with subsets to explore, empty lists are sent to every process in order to end the search.

The whole procedure is protected with a mutual exclusion mechanism, to guarantee the coherence of the worker elected and the *Request work list*. Algorithm 12 illustrates this part of the parallelization.

Algorithm 12 Request Work

```
1: procedure RequestWork(ownWorker)
2:     getLockShare( )
3:     requestWork ← requestWork ∪ ownWorker
4:     electedWorker ← getWorkerMostWork( )
5:     if workerToSend == −1 then
6:         for worker ∈ workers do
7:             if worker ≠ ownWorker then
8:                 sendWork(worker, ∅)
9:         return ∅
10:    releaseLockShare( )
11:    worklist ←ReceiveWork( )
12:    return worklist
```

Share Work

In the case that a process is elected to send work, it equally divides its *Worklist* among all the processes in the *Request work list* and itself. The first step to do so, is

to obtain the same lock used to add workers to that list. This prevents new workers to add themselves to the list while the elected worker is already sharing work. Then, the *Worklist* is finally split into several parts, and each one is sent to a different worker.

There is a special scenario, where the selected worker does not have enough work to share. In that case, it will postpone the work sending task to the next iteration, in order to verify if it is able to generate more work while processing another subset. This is a recursive case until the worker either gets enough work to share, or runs out of it. If the later happens, the worker will call the *RequestWork* function, which will nominate other worker to send work.

Work sharing is a vital feature not only on this strategy but on the master-slave approach as well. It would be impossible to achieve good parallel performance without it. Algorithm 13 illustrates the *ShareWork* function.

Algorithm 13 Share Work

```
1: procedure SendWork(worklist, ownWorker)
2:     if notEnoughWork thenworklist
3:         return worklist
4:     getLockShare( )
5:     dividedWorklist ← DivideWork(worklist, requestWork)
6:     for worker ∈ requestWork do
7:         SendWork(worker, dividedWorklist_worker)
8:     requestWotk ← ∅
9:     releaseLockShare( )
10:    return dividedWorklist_ownWorker
```

Chapter 6

Performance Tests

In previous chapters, we introduced a new hybrid feature selection algorithm, named MITWS, which uses a novel meta heuristic. Additionally, we proposed two parallel strategies using different memory paradigms to take possible advantage of multiple processing elements in order to improve execution time. We now aim to assess the performance of our strategies. We evaluate the performance with respect to two main factors: performance of parallel execution and quality of feature selection. Thus, accordingly, this chapter is divided into two big sections each presenting the test results for each main performance factor. A last section is also included to provide details on where the produced work is available and how it can be used.

6.1 Parallel Performance

In this section the parallel performance will be addressed. In this test, the speedup of each implemented strategy will be focused. Speedup is a metric for relative performance enhancement when executing a task. This notion was established by Amdahl's law [3]. The term can be utilized to show the effect of any performance improvement. In this work, we will use it to measure the parallel gain of our proposed parallel strategies when the number of processing units increases. Speedup is calculated using the following formula:

$$Speedup = \frac{T_{old}}{T_{new}} \tag{6.1}$$

where T_{old} refers to the execution time of the non improved implementation and T_{new}

to the time on the improved version. In our tests, T_{old} represents the execution time of the strategy when a single processing unit is used and the T_{new} exhibits the execution time when more processing units are utilized. It is important to highlight that for the speedup calculations a pure sequential version of the strategy was not implemented. The parallel strategies require new structures to be defined, some conditions to be checked and some extra procedures in order to properly execute. Therefore, using them with a single processing unit will have an overhead when compared to a pure sequential version. In our case, we estimate that this overhead is barely noticeable. The number of defined structures is related to the number of processing units, the extra procedures basically divide data and share work across multiple processing units, and the conditions are not that computationally expensive. As a result, the overhead when using a single processing unit is low and we did not have the necessity to implement a sequential version of the strategy in order to estimate the speedups. However, it is worth to point out that the T_{old} that will be shown, would have been slightly lower if a pure sequential version was adopted.

The higher the speedup is, the better parallel gains are obtained. The maximum speedup achievable on an application depends on its structure. This limit is given by the Amdal's law [3]. Despite some recent theories which point to a different way to calculate the maximum speedup achievable [25], in this work the Amdahl's theory will still be used as reference. Basically, the mentioned law states that the maximum speedup of a program is limited by the sequential parts of it. These are chunks of the code that cannot be executed in parallel. By itself the concept is quite easy to understand, if an application which takes 10 minutes to execute, has 2 minutes of its time in a part of the code that cannot be parallelized. Then, no matter how many processing units are used, the execution time will not be lower than 2 minutes. Therefore, the maximum speedup is limited.

On the presented strategy, there are no parts of the algorithm that cannot be executed in parallel. Therefore, the process of evaluating the performance of the strategies will be much easier and the speedups will be calculated using the previous formula.

6.1.1 Testbed Description

Addressing the technologies used in the development and testing of both strategies. The programming language used on both strategies was Python2.7 [49]. For machine learning functions such as the SVM classifier, cross-validation techniques, etc, the

scikit-learn [52, 46] was adopted.

To achieve parallelism, we defined multiple processes using the unix fork. This is not the best approach for the shared memory strategy because, by default, processes do not have access to the same memory zone. However, python GIL [47] prevents multiple threads from executing python bytecodes at once[1]. This means, that it is not possible to use concurrent execution of threads, which would be the natural choice for such memory paradigm. Instead, in order to simulate shared memory between processes we used the multiprocessing manager [48]. As a note, simulating shared memory causes an overhead on the computations that may negatively impact the parallel performance of the algorithm. Therefore, it is expected that using our shared memory strategy with threads will improve the results we are reporting later in this section.

Related to the communication between processes, the master-slave strategy uses pipes for interprocess communication, while for obvious reasons the shared memory approach relied on processes sharing the same memory space to exchange information.

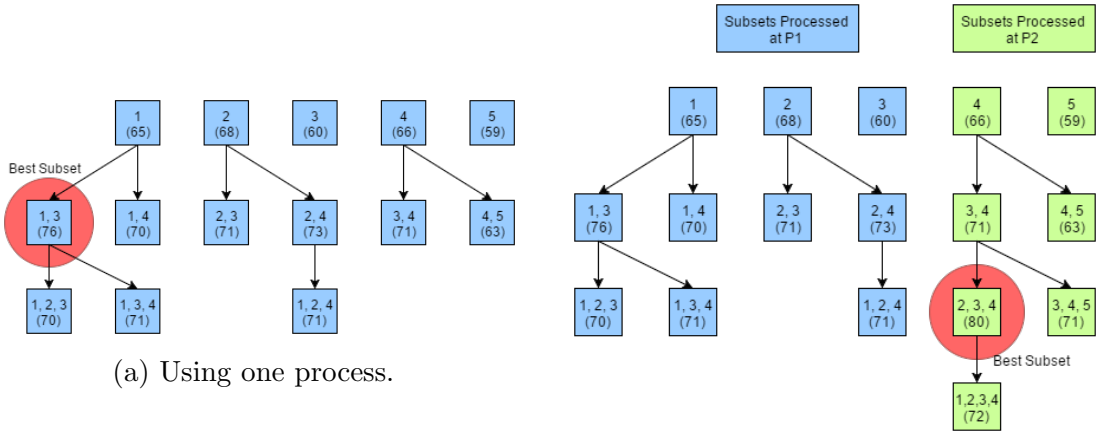
Hardware Details

The tests were obtained using one node of a cluster of CRACS/INESC TEC research unit, located at the Department of Computer Science within the Faculty of Sciences of the University of Porto. Each node has four Quad AMD Opteron 6376 processors, totalizing 64 cores with 32 cores available for floating point operations. Additionally, each processor operates at 2.3 GHz and has a 16 MB L2 and a 16 MB L3 cache. The total amount of RAM available on the system is 256 GB.

6.1.2 Parallel Test

The common approach to assess the speedup of an application is to run it several times using a different number of processing units. This is what will be presented on this chapter. However due to the nature of MITWS, it is not possible to select a dataset, execute it, and expect to get accurate speedup values. The reason for that is the stochastic part of the algorithm. At each execution, the probabilities vector that controls the generation of successors would be different. Additionally, the cut mechanism that operates during the second stage of the wrapper would remove a

[1]This issue was discovered at a late stage of the work which prevented migrating the whole implementation to a different programming language.



(a) Using one process.

(b) Using two processes.

Figure 6.1: Wrapper search example.

distinct number of subsets. On top of that, as more processing units are being used on the MITWS algorithm, the more exhaustive the search is going to be, thus increasing the likelihood of finding a better solution. This phenomenon can be explained with an example, considering the first stage of the wrapper search. Here, the decision to expand a subset after obtaining its score relies on how distant it is from the best score so far. Therefore, it is plausible to assume that the sooner a high score subset is found, the less subsets will be expanded, resulting in a lower number of tests.

Examining figure 6.1a, where the top part of a square represents the features of a subset, and the bottom one its score. Additionally, the edges coming from a square represent the successors of a subset, and the overall image illustrates the execution of the wrapper search using a single processing unit. As it is possible to note, there were a total of 14 tested subsets and the best score found was 76.

If the number of processing units is changed to two, the search tested a total amount of 17 subsets and the best score found was 80. This can be observed in figure 6.1b.

The reason why there were a different number of explored subsets which resulted in different scores is easy to understand. While using a single process, the subset $[1, 3]$ results in a score much higher than the others subsets with two features. This culminated into not expanding the subset $[3, 4]$. When using two processing units, the work is divided among them. Therefore, when the second process decided to expand the subset $[3, 4]$, it was not aware of the highest score of $[1, 3]$ that were being tested at the same time on the other processing unit. This resulted in expanding some more subsets which produced a better score.

The search space on the given example is small, and only uses two processing units, however it proves the point that the number of processing units changes the exhaustiveness of the MITWS algorithm. It is impossible to find a formula that relates the increase number of tested subsets with the number of processing units. Additionally, if a formula is found for a certain dataset, it is extremely likely that it will not work for a different one. The lack of *"rules"* about subset scores is what makes feature selection such a difficult problem.

To address the speedups on the presented parallel approach, the time it takes to process a certain number of subsets using different numbers of processing units will be used. The idea is to define a test in which, independent of the number of processing units, the final number of tested subsets is always the same. Basically this means that in the end, the presented results will address the capability of the parallel algorithm in exploring more nodes in less time. However, if a speedup of 10 is achieved for 16 processing units, that does not translate into: "if a solution takes X seconds to be found using a single processor, then 16 processes will make it 10 times faster to find the solution", due to the previous mentioned reasons. Instead, it means that using 16 processes, the algorithm will be able to test subsets 10 times faster.

Test Conditions

The defined test consists in verifying the worst case scenario when 20 features reach the wrapper search. This scenario is characterized by the following properties:

- Every tested subset is expanded independently of the score.
- When expanded, each subset is combined with all possible features. This represents the successor generation without using the probabilities vector.
- The cutting probability for all subsets during the second stage of the wrapper search is 0%.

More precisely, this results into an exhaustive search with a total amount of $2^{20} - 1 = 1048575$ subsets tested. Due to the fact that every subset is expanded, and it tries to generate all the possible combinations with itself, the number of attempts at producing and testing repeated subsets is very high. This is a useful feature to test the capability of the presented solutions in avoiding duplicated work.

6.1.3 Master-Slave approach

Before addressing the speedups test, it is necessary to go back to the *CommRate* variable introduced in section 5.3.2. This variable controls the amount of time it has to pass until the slave tries to communicate with the master. On the presented strategy, it is important to control this because, while the master is removing repeated work from the slave's *Worklist*, the last one is not producing any work. Thus, this component of the strategy, although being crucial, can be described as wasted time. Therefore, if a slave communicates very frequently with the master, it will waste a large amount of time. On the other hand, if it does not, it will increase the likelihood of processing repeated work.

Basically, the *CommRate* represents a trade-off between the capability of not performing duplicated work and the time wasted checking for repetitions. Moreover, it is not easy to find the best value for this variable because it changes depending on the problem's dataset. That is the reason it's defined as a user variable which can be manipulated on the algorithm.

In order to give some insight about this trade-off, 17 processing units were used to execute several times the previous defined test conditions. On each test, the value of the *CommRate* was modified. The idea is to show how this variable impacts the general performance of the algorithm. Figure 6.2 illustrates the results comparing the number of repeated tests performed with the execution time on each *CommRate* value.

The outcomes reinforce the previous idea that as the frequency of communications increases, the less repeated subsets are tested. However, that did not always translate into better execution times. According to the results, using *CommRate* around the 50 seconds is the best solution. The amount of repeated work on these values, is compensated by the not so frequent time wasted removing the repetitions. The results also provide useful information for future uses of the parallel strategy. As it is possible to note, the execution times are much worse for low values of *CommRate*. Therefore, this variable should defined using values between the interval 40 to 100.

6.1.3.1 Master-Slave Parallel Tests

Table 6.1 presents the results of the speedups tests for the master-slave strategy. During the tests the *CommRate* was set to at 50 seconds, the best value obtained from the previous test.

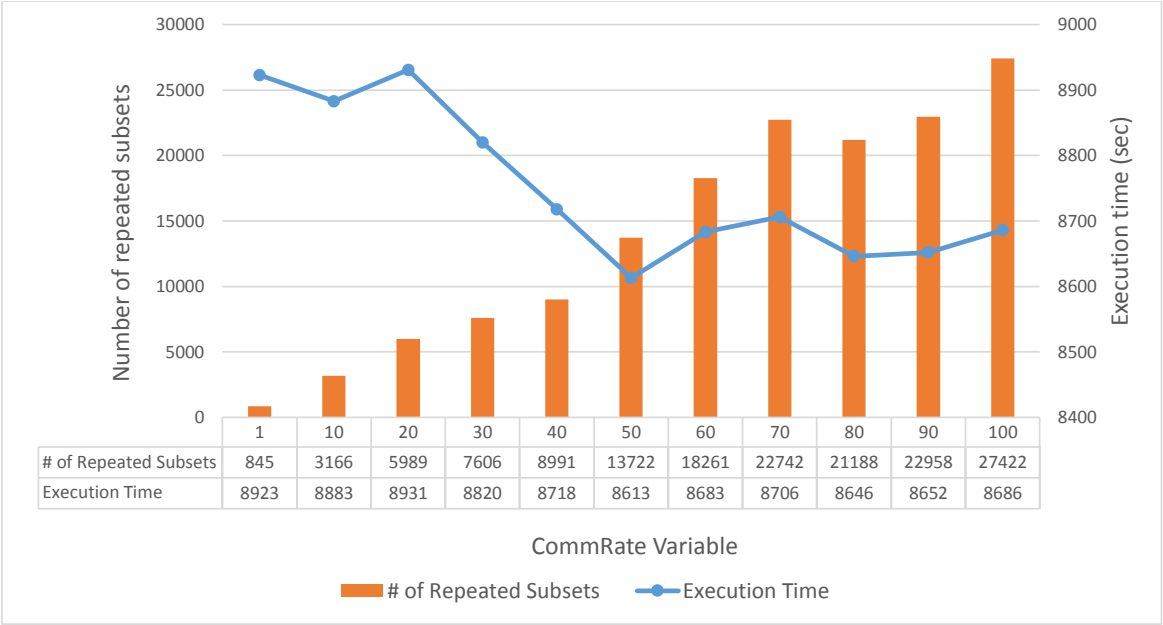


Figure 6.2: Results of testing the CommRate variable

Table 6.1: Results of the speedup test on the Master-Slave approach.

# of processes	Execution time (sec)	# tested subsets	% of repeated subsets	Speedup
1	136513	1048575	0.0000	1.00
2	136513	1048575	0.0000	1.00
3	77196	1049653	0.1028	1.77
5	35118	1052073	0.3336	3.89
9	17166	1057822	0.8819	7.95
13	11505	1062299	1.3088	11.87
17	8657	1062574	1.3350	15.77
21	6885	1066574	1.7165	19.83
25	5799	1079628	2.9614	23.54
29	5235	1090226	3.9722	26.08
33	4624	1116719	6.4987	29.52

In addition to the table, figure 6.3 compares the obtained speedup against the ideal one.

6.1.4 Shared Memory approach

The results for the speedups tests using the shared memory strategy are presented in table 6.2.

Figure 6.4 illustrates the evolution of the speedup with an increasing number of processing units, also comparing it with the ideal value.

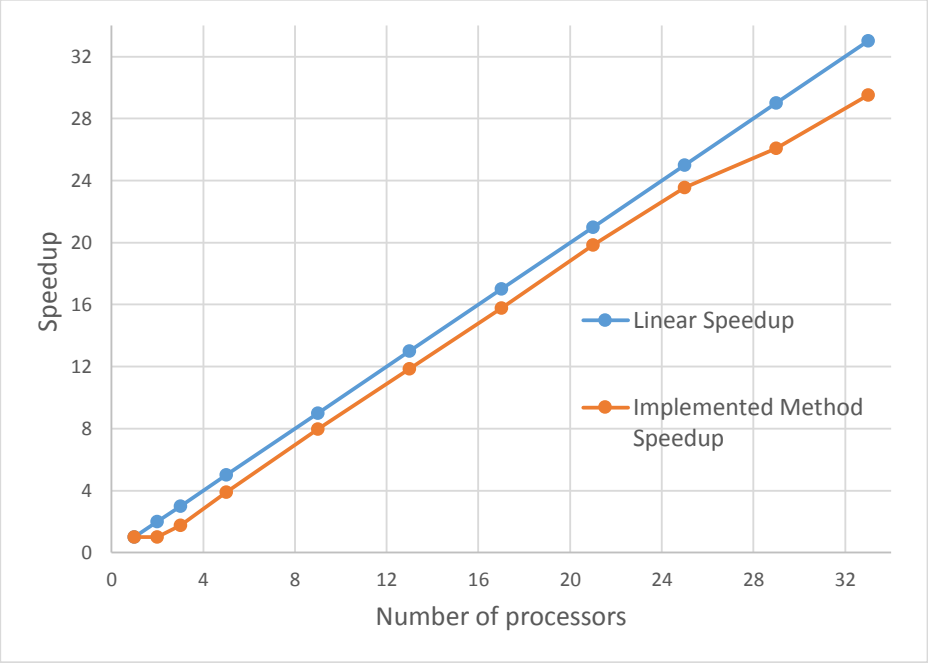


Figure 6.3: Speedup of the implemented master slave strategy.

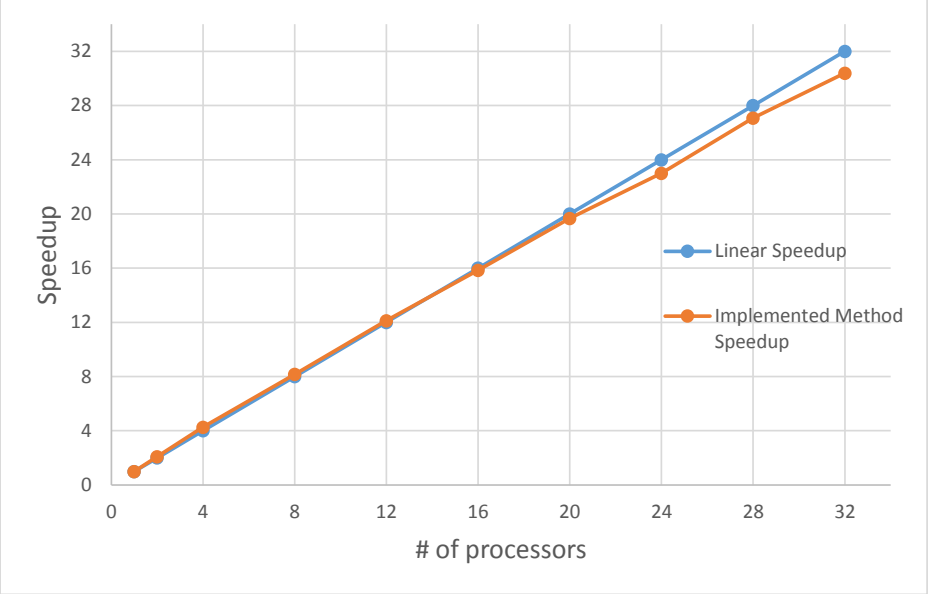


Figure 6.4: Speedup of the implemented shared memory strategy.

Table 6.2: Results of the speedup test on the Shared memory strategy.

# of processes	Execution time (sec)	# tested subsets	% of repeated subsets	Speedup
1	133984	1048575	0.0000	1.00
2	64148	1048575	0.0000	2.09
4	31427	1048575	0.0000	4.26
8	16406	1048575	0.0000	8.17
12	11055	1048576	0.0001	12.12
16	8457	1048575	0.0000	15.84
20	6812	1048575	0.0000	19.67
24	5825	1048578	0.0003	23.00
28	4948	1048581	0.0006	27.08
32	4409	1048579	0.0004	30.39

6.1.5 Comparing the Two Approaches

Examining the presented speedups results, it is possible to conclude that both methodologies scale well when several processing units are used. However, the shared memory strategy outperforms the distributed one. This section will start by doing an individual analysis of both strategies, and then will address their discrepancy related to performance.

Starting with the master-slave strategy, when a small number of processing units is used, the gain on parallel performance is not relevant. This is explained by the fact that this strategy requires the master, which basically means that one process will not be doing any work, and its only propose is to control the others. As the number of processing units increases, the use of the master is mitigated and the parallel gain starts to increase. Moreover, the number of repeated work performed during the tests grows with the number of processing units. Although these values are not alarming, they still have some impact on the overall execution time.

The shared memory strategy achieves speedups near the linear values. In fact, in some cases, the algorithm was able to obtain super-linear speedups [2]. Additionally, with respect to avoiding repeated work, the algorithm proved to be extremely efficient. In the worst case, 0.0006% of the subsets were tested more than once. This is an irrelevant number in more than one million tested subsets.

Differences in performance

It is quite easy to explain the differences related to the performance of both strategies. In fact, part of it was already explained in section 5.4, where the advantages of using shared memory are stated. As mentioned, shared memory provides a cheap way of

interprocess communication, something that is not possible on distributed memory. Moreover, it is plausible to consider that the hash table which stores all the subsets, is the center piece of the wrapper. On shared memory, every worker has access to it every time they want and this variable is constantly updated. On the other hand, the slaves on the first strategy do not have access to the global hash table, only the master does. Therefore, they have to rely on the master to remove a big part of their repeated work. Since this has a cost, and the master cannot do it to all the slaves at all times, some repeated work eventually happens, which negatively impacts the performance. Finally, the main reason for the discrepancy is related to the usage of a master that counts as a process but does not dispatch any work.

In summary, the shared memory is the perfect environment for the proposed wrapper. However, the distributed memory strategy has a huge advantage. It is a lot more difficult to have computer systems that are allowed to share memory with each other, than to have access to multiple machines that are physically distant from each other. Therefore, the master-slave strategy, despite performing a little worse, is probably easier to implement on a large number of machines, and that is the reason why it was added as part of this work.

6.2 Feature Selection Results

In this section the performance of the MITWS algorithm with respect to feature selection will be addressed. In this test, MITWS will be applied to find subsets of features in some public datasets, and then its results will be compared to several other algorithms that solved the same problem. The idea is to understand where MITWS ranks among other algorithms. In the test, the quality of the solution will be evaluated regarding: accuracy of generalisation, number of features of the solution, and a combination of both.

In 2003, a feature selection challenge was presented at the Neural Information Processing Systems (NIPS) conference. It aimed to find algorithms that significantly outperform methods using all features [44]. In order to achieve it, five binary classification problems were defined. Each one was represented by a dataset with different characteristics, which were divided into three sets: train, validation, and test.

For the first two sets, the participants had access to data and its labels. However, for the test set, data was unlabelled. The challenge consisted in using any machine

learning, statistical analysis, feature selection, and/or any other technique on the train and validation set, to produce a classifier that provides the best accuracy on predicting the labels of the test set. The predicted labels were submitted to a web page which provided feedback about the accuracy of the participant. Additionally, every submission was recorded, in order to rank the participants.

The challenge closed right after the conference, however its results are still available [44] and there is still a web page that allows for new test submissions [10]. To assess the quality of MITWS, the algorithm was applied to the five datasets in order to produce a subset of features that will be used to construct a classifier. During this process, no other technique besides those that are part of MITWS, was adopted. For comparison, the performance of the resulting classifier in each dataset, was confronted to past submissions of the NIPS challenge.

Typically, feature selection algorithms provide only one subset of features as solution. By contrast, on MITWS the user is allowed to define a value S that represents the number of solutions he wants to receive. Meaning that the algorithm will record the top S solutions found during the search, and in the end it will print them sorted by their score. This characteristic of MITWS, is useful because it is common that there are several solutions with almost the same score. What differentiates the quality of these solutions, is their ability to predict unseen data. This is something that cannot be tested during feature selection. Therefore, by providing the user with the option of selecting the number of solutions, we are allowing them testing the performance of all solutions in unseen data, without rerunning MITWS. For example, in the test presented in this section, we chose to select 20 solutions for each dataset in order to use the one that is able to achieve the best accuracy for the unseen data of the validation set. The process used to obtain a classifier for the NIPS dataset will be explained next.

For each dataset, the steps in order to produce the classifier were the following:

1. Apply MITWS to the train set of the dataset, with the propose of finding 20 subsets of features.
2. Each subset was adopted to create a classifier and predict the labels of the validation set.
3. Select as final classifier, the one that was able to obtain the best accuracy for the validation set.
4. Use it to predict the labels of the test set, and submit the results to the

website [10].

The remaining of this chapter will introduce the datasets adopted, the results of MITWS, and conclude with a detailed analysis about the performance of the algorithm.

6.2.1 NIPS Datasets

The NIPS challenge datasets are available in several web pages [44, 10, 56]. Here, a brief description about the classification problem of each one is provided:

- **Arcene** is a task to distinguish cancer versus normal patterns from mass-spectrometric data.
- **Dexter** is a text classification problem in a bag-of-words representation. The idea is to filter text about *"corporate acquisitions"*.
- **Dorothea** is a drug discovery dataset. Chemical compounds, represented by structural molecular features, must be classified as active (binding to thrombin) or inactive.
- **Gisette** is a handwritten digit recognition problem. The problem is to separate the highly confusable digits "4" and "9".
- **Madelon** is an artificial dataset created for the challenge.

For the challenge purposes, several probes were added to the problem in each dataset. In the context of the challenge, probes are features that are irrelevant for the classification, and intend to increase the difficulty of the task.

Datasets are available in three different data formats, their characterization is the following:

- **Sparse Binary:** it is not required that every data example has values for all the features. Moreover, the ones that have are represented either by a *"0"* or *"1"*.
- **Sparse Integer:** the same as sparse binary, but instead of having a binary value, feature values are represented by an integer.
- **Dense:** every feature has a value which is an integer.

Finally, for each dataset, table 6.3 addresses the number of examples given in every individual set, the number of features, and the format of the data:

Table 6.3: Characteristics of the NIPS challenge datasets.

Dataset	Data Format	# Train	# Validation	# Test	# Features
Arcene	Dense	100	100	700	10000
Dexter	Sparse Integer	300	300	2000	20000
Dorothea	Sparse Binary	800	350	800	100000
Gisette	Dense	6000	1000	6500	5000
Madelon	Dense	2000	600	1800	500

6.2.2 NIPS Results

It would be pointless to use both strategies to obtain the results for the feature challenge. Therefore, the presented results were obtained using the shared-memory one. The choice relied on the fact that this strategy has better performance in parallel environment and therefore will obtain a solution faster.

Regarding user defined parameters during the tests, the following values were used:

Parameter	Value
Search threshold during first stage	0.5%
Cutting mechanism during final phase	900 seconds
Successors generation	probability estimation
Probability estimation tests	25
Grid search tests	25
SVM kernel	RBF
Number of processing units	62

These were the fixed values for all tests, however some parameters such as cross-validation technique and percentage of the filter had to be adapted according to each dataset. Table 6.4 presents the used parameters and the best solution obtained for each problem.

The best subset of each dataset was used in the training part of the classifier. Then, the labels for validation and test set were predicted. The accuracy value for each are presented in table 6.5. The final two columns, represent the rank of the accuracy obtained on the test set, and number of features in the final solution when compared to all the previous challenge submissions.

For the sake of understanding, the rank columns represent the MITWS position followed by all the participants. Additionally, not every past submission had the

Table 6.4: Parameters and solutions of MITWS on NIPS datasets.

Dataset	Ft	Nfpf	Cross-validation	Sfs	Final score	Time (sec)
Arcene	0.50	232	Leave-one-out	14	99.00	2156
Dexter	0.94	364	Leave-one-out	72	98.50	32539
Dorothea	0.90	450	5 folds	30	97.63	32962
Gisette	0.97	121	5 folds	85	97.32	53560
Madelon	0.97	255	10 folds	14	87.10	33117

Ft = Filter threshold, Nfpf = Number of features post-filter, Sfs = Size final subset.

Table 6.5: Results of MITWS on the NIPS challenge.

Dataset	Train	Validation	Test	Accuracy Rank	# Features Rank
Arcene	99.00	82.00	74.56	892/1503	108/1455
Dexter	98.33	83.67	81.65	819/1007	132/936
Dorothea	95.63	94.29	77.18	475/812	70/768
Gisette	98.97	96.80	96.67	465/932	138/879
Madelon	93.35	87.50	88.67	344/1059	248/1001

amount of features of the solution available. Therefore, the number of counted tests on the challenge is different for the two presented ranks.

Related to accuracy, the results in average rank among the top 60% for all the datasets. In terms of number of selected features the outcomes are much better since it puts the MITWS algorithm among the top 15% of all the submissions. Lets look with more detail at these results.

The outcome of the accuracy came at no surprise, since the goal was to develop a classifier which accurately predicted the test set. Despite being part of the machine learning workflow, feature selection is only part of the process and usually several more techniques such as outlier detection, noisy data removal, and generation of synthetic data are required [31]. Moreover, knowledge about the specific problem at hand can improve the generalisation result by targeting feature choice, or through the use of another metric for calculating the feature subset score in the search [13].

On the presented experiments, the focus was to test the ability of MITWS to find a good solution according to the score function, which was the accuracy of the learning algorithm on the training data. Although cross-validation strategies were used to improve generalization, they were not enough, and in general the classifier presented a much lower accuracy, when predicting unseen data (validation and test sets). Nevertheless, the results confirm that the proposed hybrid approach, without any further analysis of the dataset nor additional techniques, was able to produce quite acceptable results.

With regards to the number of features selected, the MITWS was able to select less features than most algorithms. Although, this metric was not used to rank the algorithms during the original challenge, it is very important in the context of feature selection. Sometimes, it is preferred to produce a smaller size subset than a bigger one with better performance. Decreasing the cost of producing the dataset and improving classification time are examples of reasons to justify that choice.

In fact, it would be important to correlate, for every submission, the number of features selected and the accuracy. With this intent, figure 6.5 allows a graphical visualisation of the percentage of features reduced and accuracy for every submission in every dataset of the NIPS challenge. In each image, the center of the red circle demonstrates the location of the MITWS produced solution. As a note, to ease understanding of these images, only submissions that are able to obtain accuracies and percentage of feature reduction higher than 50% are considered. For an illustration that represents all the submissions see figs. C.1 to C.5 appended to this work.

In addition to the figure, to rank the correlation between feature reduction and accuracy among submissions, a score metric was defined using the following formula:

$$TestScore = (\%\_of\_feature\_reduction * 50\%) + (accuracy * 50\%) \tag{6.2}$$

On this score, the same importance is attributed to the size of the features subset and accuracy of the classifier. Converting all the past submissions, as well as the MITWS results, to the new score, originated a new rank presented in table 6.6.

Table 6.6: MITWS rank using the combined metric.

Dataset	**Accuracy**	**% Feature reduction**	**New score**	**Combined rank**
Arcene	74.56	99.86	87.21	410/1455
Dexter	81.65	99.64	90.65	579/936
Dorothea	77.18	99.97	88.58	391/768
Gisette	96.67	98.58	98.12	22/879
Madelon	88.67	97.20	92.94	323/1001

For the combined metric, the MITWS ranks always on the top half of the scores. Additionally, the score of the *Gisette* dataset is in the top 25 which is a big achievement.

To summarize the analysis on the performance of MITWS related to feature selection, in all cases the algorithm was able to produce a final subset of features that was around 99% smaller than the original set of features, and capable of creating a classifier with high accuracy. Although there are not guarantees that the optimal subset was found

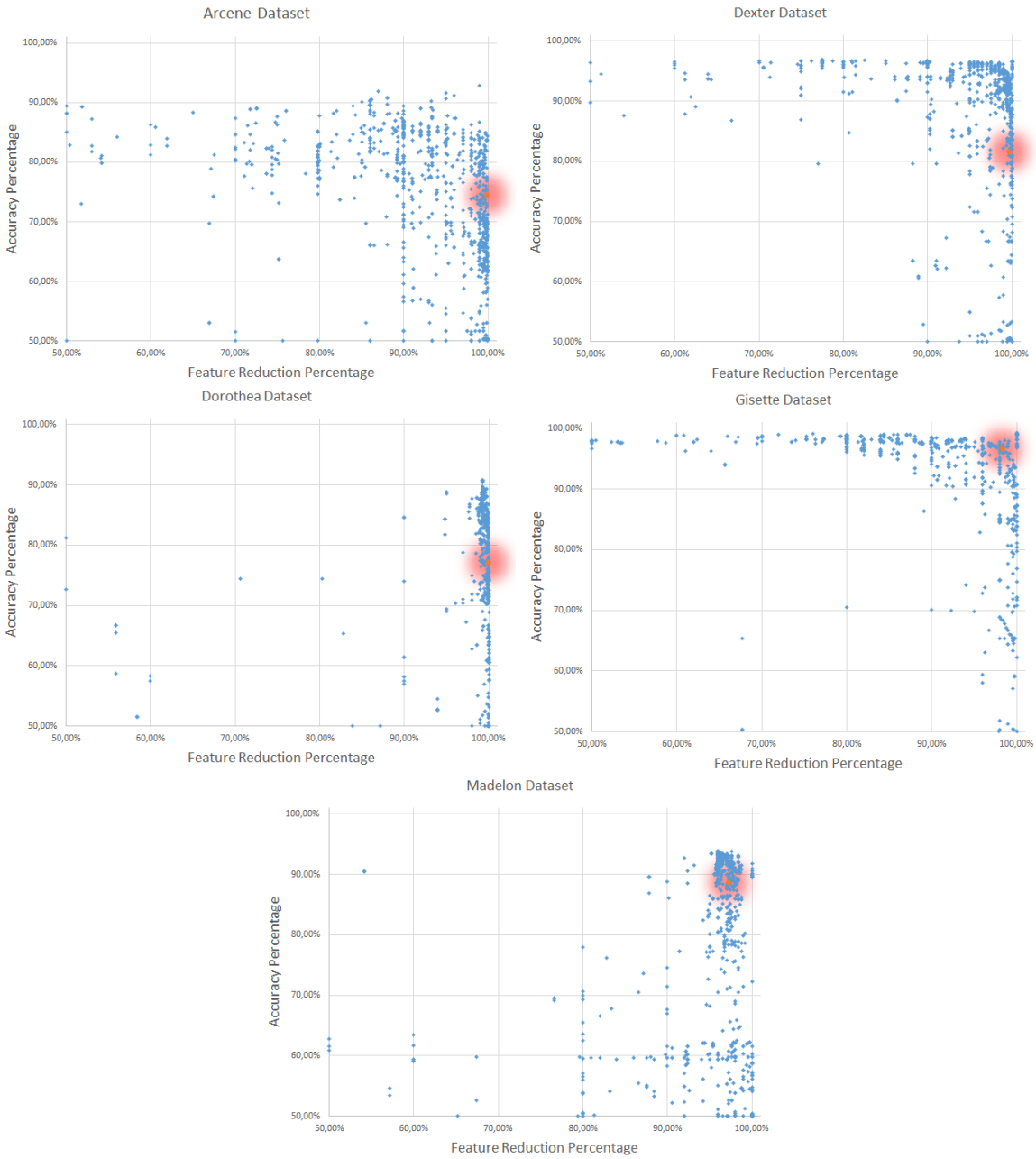


Figure 6.5: Correlating feature reduction and accuracy for several submissions in the NIPS challenge.

(it would require to test the whole search space), the wrapper search was able to obtain a final subset near the perfect score in 4 out of 5 datasets. Moreover, the results were obtained in a practicable amount of time, considering the size of the datasets and the cross-validation methods used.

6.2.3 Testing MITWS Parameters

During this work several user parameters on the MITWS algorithm were mentioned, this part of the thesis will review all of them and assess their impact on the algorithm's performance. Since the *CommRate*, which is the only user defined parameter that differentiates the two parallel strategies, has already been explained, the discussion will be related to the presented feature selection algorithm and not its parallel strategies. Nevertheless, it is important to refer that the presented results along this section were obtained using the shared memory strategy.

Most of the parameters are used to control the exhaustiveness of the wrapper search. As it was mentioned, as more exhaustive the search is, the better are the chances of finding better solutions. However, the execution time is highly influenced. Moreover, it is important to keep in mind that waiting a lot more time for a solution that is only a little bit better, may not be worth it. This obviously depends on the goals of the feature selection. Analysing each user defined parameter.

The number of tests to estimate the grid search and the improvement probabilities are the only two parameters that do not have a direct impact on the wrapper search. Both are used in a pre-search stage and the idea is that, as more tests are made, the chances of estimating better values increases.

The filter percentage defines how many features are able to reach the wrapper part of the algorithm. Advice about this feature was already given in section 3.3 and its value will always depend on the dataset. It affects the exhaustiveness of the search because as more features are present in the wrapper part, the bigger the search space is.

The cut mechanism on the second phase of the wrapper consists of a timer that regulates how often subsets are randomly removed from the search. The idea is to decrease the amount of work at the final stage of the presented wrapper. Therefore, the more often it removes subsets, the less will be explored.

The search threshold that controls the subset expansion during the first phase of the

wrapper is responsible for the amount of explored subsets. During this stage of the search, the goal is to gather subsets that are close to the global best solution. The concept of close is defined by this parameter. Hence, if the percentage increases, so does the number of subsets tested.

The two remaining parameters are the size at which the wrapper switches phases and the option to control how successors are generated. For the last part, there are two options, either use the standard that explores a subset with all the possibilities, or estimate the improvement probability vector and use it to generate successors. Related to the number of tested subsets, the first strategy will result in more explored subsets, since it expands every one in all its possibilities.

The size that determines the change on phase is a more complicated problem. Recapitulating the goal of the two-phases, the idea is to gather as many subsets close to the best score as possible during the first stage, and then explore them until the best score they can reach. However, analysing how this translates into the number of tested subsets it is not easy. Therefore, some tests were made to address this issue.

The *Madelon* dataset from the NIPS challenge was used for the test. The idea was to verify the score of the final solution, along with the number of tested subsets, as the size to switch phases increase. Additionally, the impact of both successor generation functions was assessed by repeating each test using both procedures. Figure 6.6 illustrates the results.

From the outcomes its possible to conclude that the size has a huge impact in the score of the final solution and the number of tested subsets. Furthermore, small sizes should be avoided, mainly because they process more subsets and tend to find worse solutions. Regarding to the successor generation, the improvement probability always tests a fewer number of subsets as expected. However, the quality of its solutions is equivalent to the standard one. This reinforces the idea that a more exhaustive search does not guarantee finding better solutions.

To summarize, table 6.7 demonstrates the impact of the parameters mentioned during this section on the exhaustiveness of the wrapper search.

Table 6.7: Impact of the parameters on the wrapper search.

	Parameter changes	Wrapper exhaustiveness
Filter percentage		
Improvement probability		
Search threshold		
Switch search size		
Cutting mechanism		

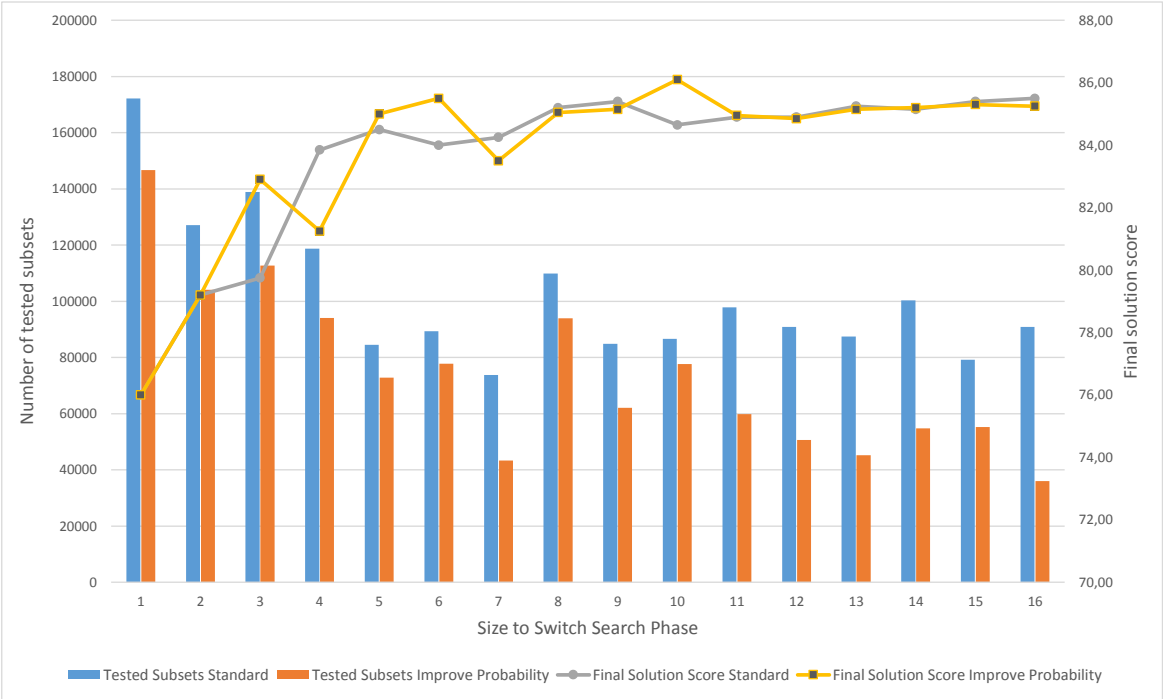


Figure 6.6: Impact of both successor strategies and the size to switch search phase on the number of tested subsets and final solution score.

6.3 MITWS Availability

The presented work during this thesis aims to further develop a scientific area that has been widely studied in recent years. For that reason, and with the goal of helping machine learning researchers with yet another tool, the produced work is provided under open source licence and available at [20].

Nevertheless, not everyone who works with machine learning has a background in programming, nor has the knowledge to dig through *Python* code to adapt the algorithm to their needs. Therefore, with the goal of making this work fully available to any researcher, a system that allows the manipulation of every parameter on MITWS algorithm, using a configuration file in the XML format, was implemented. The remaining of this chapter will explain how the system can be used.

The following example illustrates the attributes and their expected type of values, in the configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<settings>
        <setting name="train_file"> filename or filename1, filename2</setting>
```

```
        <setting name="valid_file"> filename or filename1, filename2</setting>
        <setting name="test_file"> filename </setting>
        <setting name="dataset_type"> name </setting>
        <setting name="dataset_name"> name </setting>
        <setting name="number_of_features"> number </setting>
        <setting name="number_of_processes"> number </setting>
        <setting name="number_of_solutions"> number </setting>
        <setting name="grid_tests"> number </setting>
        <setting name="probability_estimation_tests"> number </setting>
        <setting name="cross_validation_strategy"> number </setting>
        <setting name="svm_kernel"> name </setting>
        <setting name="kernel_parameters"> numbers </setting>
        <setting name="estimate_improvement_probabilities"> no or yes </setting>
        <setting name="percentage_filter"> number (0-1) </setting>
        <setting name="size_to_switch_search_stage"> number </setting>
        <setting name="threshold_search"> number (0-100)</setting>
        <setting name="search_cutting_timer"> number </setting>
</settings>
```

The relation between attributes name, and their function in MITWS should be clear. Nevertheless, for clarification, they will be discussed following the order of the XML file.

The configuration file was adapted to process the tests for the NIPS challenge. For that reason there are three file attributes, where each one represents a different set. Data can be inserted into two ways: using only one file where the last column is the label or introducing data and labels into different files. The exception is the test set, where there are no labels available, therefore only one is required. Although three files are presented, it is possible to use only the train one, for the case where the user expects the final subsets of features and do not want to further test them.

The *dataset_type* is the format of the data. Currently all formats of the NIPS challenge are acceptable. In addition to that, the CSV format was appended. The *dataset_name* is just a way to refer to the dataset and the number_of_features is the total amount of features in it.

The *number_of_processes* indicates how many processing units should be used. The *number_of_solutions* stipulate the number of solutions that should be given in the end of the execution. The test attributes define the amount of tests on the probability estimation and grid search. The *cross_validation_strategy* specify the number of folds to use during the process of evaluating a subset. The *svm_kernel* defines the kernel of the SVM (currently available: RBF and linear), while the *kernel_parameters* define its parameters. If this last attribute is not a number, the MITWS will use the grid search to estimate it. The

estimate_improvement_probabilities selects the strategy to generate successors, setting it to "yes" enables the improvement probabilities.

The last four parameters control the exhaustiveness of the search, and the attributes name is the same as defined on the previous chapter where their impact was assessed.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This work aimed to propose a new feature selection algorithm and a strategy to execute it with several processing units. During this thesis the new algorithm, MITWS, which combines several pre-existing techniques with a novel wrapper search was presented and discussed. Moreover, two strategies to execute MITWS in parallel were presented, one for distributed memory and another for shared memory. Furthermore, all the produced work was thoroughly analysed through different perspectives.

Assessing the results of the presented thesis, related to parallel performance and in terms of scalability, both strategies conferred good results. The distributed memory methodology, proved to be a reliable way of exploit parallelism using machines in distinct physical locations. On the other hand, the shared memory approach produced a method that fully takes advantage of the characteristics of this paradigm. It is essential to highlight that the parallel gain with this strategy was able to reach super-linear speedup values. Comparing the two approaches, the later has better parallel performance as detailed in section 6.1.5.

As far as the performance of feature selection goes, MITWS was tested against several different solutions of the well-known NIPS challenge. Three metrics were considered during comparison: accuracy, size of the final solution, and a combination of both. Examining the first one, which was the one adopted on the challenge, MITWS were able to rank around the 60% for all datasets. Due to the reasons stated in 6.2.2, the results were not expected to be any better. Nevertheless, the MITWS methodology proved to be a reliable way of finding good solutions, however it can be improved by adding new techniques or testing different components.

It is important to state that the novel wrapper search introduced in this thesis was able to find solutions near the perfect score in a reasonably amount of time. These are promising indicators of its capability.

Finally, in order to make this an ongoing work, the presented work was provided under a open-source licence. Additionally, a simple interface for those who do not have programming background was added, to reach a higher number of users.

7.2 Future Work

As the results of the NIPS challenge state, there are some aspects that can be improved in MITWS. The core of the algorithm is the novel wrapper search which presented good results. However, components like the subset evaluation, strategy to filter features, classification algorithms, or even new techniques such as outlier detection, should be tested and if proven to perform well, added to the algorithm. One way of continuing to enhance the presented algorithm, is to test the wrapper search with several different methods in order to determine which should be used. Additionally, distinct components for different stages of the algorithm could be defined as parameters in a configuration file.

With regards to the parallel strategy, both proposed strategies performed well in the tests made in this work. However, there are different highly parallel systems that would require creating new strategies. These are the case of FPGAs and GPUs. Implementing new parallel methodologies that take advantage of these structures would be a good future step for this work.

Appendix A

Acronyms

IT-Porto	Instituto de Telecomunicações do Porto
ML	Machine Learning
FS	Feature Selection
NP-Hard	Non-deterministic Polynomial-time Hard
GPU	Graphics Processing Unit
IG	Information Gain
MI	Mutual Information
PCC	Pearson Correlation Coefficients
MITWS	Mutual Information Two-phased Wrapper Search
SVM	Support Vector Machines
UCAIM	Uncertain Class Attribute Interdependency Maximization
MS	Master-Slave
NIPS	Neural Information Processing Systems
XML	eXtensible Markup Language
CSV	Comma-Separated Values
FPGA	Field Programmable Gate Array

Appendix B

Produced Papers

During this thesis, a paper was submitted to the IEEE 18th International Conference on Computational Science and Engineering. The work presented the MITWS algorithm with the shared memory strategy for parallel execution. The paper was accepted and will be presented at Porto, Portugal in 22th of October, and published in the conference proceedings.

The submission was attached to this work.

A Parallel Computing Hybrid Approach for Feature Selection

Jorge Silva
Instituto de Telecomunicações & DCC,
Faculdade de Ciencias,
University of Porto, Portugal
Email: up201007483@alunos.dcc.fc.up.pt

Ana Aguiar
Instituto de Telecomunicações
Faculdade de Engenharia,
University of Porto, Portugal
Email: aaguiar@fe.up.pt

Fernando Silva
CRACS/INESCTEC,
Faculdade de Ciencias,
University of Porto, Portugal
Email: fds@dcc.fc.up.pt

***Abstract*—The ultimate goal of feature selection is to select the smallest subset of features that yields minimum generalization error from an original set of features. This effectively reduces the feature space, and thus the complexity of classifiers. Though several algorithms have been proposed, no single one outperforms all the other in all scenarios, and the problem is still an actively researched field. This paper proposes a new hybrid parallel approach to perform feature selection. The idea is to use a filter metric to reduce feature space, and then use an innovative wrapper method to search extensively for the best solution. The proposed strategy is implemented on a shared memory parallel environment to speedup the process. We evaluated its parallel performance using up to 32 cores and our results show 30 times gain in speed. To test the performance of feature selection we used five datasets from the well known NIPS challenge and were able to obtain an average score of 95.90% for all solutions.**

I. Introduction

In 2011, a report by McKinsey Global Institute asserted that machine learning is the key for innovation, competition, and productivity [1]. For several years machine learning has been widely studied, and new techniques and algorithms are constantly emerging. However, preparing a classifier for a classification task is not easy and researchers are commonly faced with difficulties such as: how much data is needed, what features should be added, and does the dataset has outliers and noisy data [2]. Usually, researchers gather as much information as possible about a problem and turn that information into a processed dataset for machine learning purposes. This methodology often leads to datasets with a large number of features, which in most cases means poor performance from the learning algorithm. The problem is commonly known as the curse of dimensionality [3]. Moreover, as more features are used the higher is the risk of overfitting, which means adapting a learning algorithm so much to the training data, that it starts *"memorizing"* examples instead of learning from them. Thus, drastically decreasing prediction accuracy for unseen data [3].

Feature selection is the process of selecting a subset of the original features so that the feature space is reduced according to a certain evaluation criteria [4]. The goal is to find the smallest subset possible that yields the minimum generalization error. There are several advantages of using feature selection: improving classification performance, reducing the time it takes to classify unseen data, and achieving a better understanding of the process that generates data [5]. Since feature selection is able to effectively reduce the dimension of the data, it is a commonly used technique to tackle the previous mentioned classification problems.

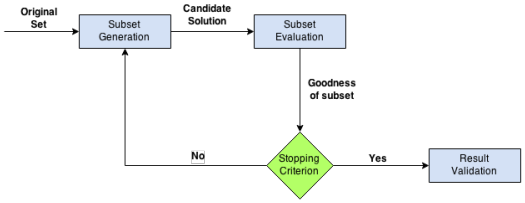


Fig. 1: The four key steps of feature selection.

Feature selection methods require a full search of the feature space, testing subsets of features, and evaluating them to find the final solution. The search space consists of the combination of all possible subsets, which for a dataset with n features produces a feature space of size 2^n. This makes an exhaustive search impracticable in most cases. For problems with a large number of features, finding an optimal subset of features is usually intractable and many problems of this kind are asserted as NP-hard [6]. Several algorithms exist in the literature that tackle this problem. Despite their differences they all follow the same general approach, which consists in four key steps: subset generation, evaluation of subset, stopping criterion, and result evaluation [4]. The first step defines how successors of a subset are generated and how the search is guided. The second step represents a function that is used to measure the quality of a subset. The conditions that make the search stop are defined in the stopping criterion. Finally, results validation is the process where the final solution from the feature selection algorithm is evaluated for its quality. Figure 1 illustrates an overview of the general process.

Depending on the size of the dataset and on the approach, feature selection algorithms can take significant time to reach the stopping conditions. Because of that, parallel computing emerges as an option to tackle this problem. Taking a closer look at the general procedure, the problem can be reformulated as a set of multiples tasks. On this scenario, a task could be defined as the process of generating a new subset, evaluating it, and checking if the stopping criterion is reached. Understandably, the processing of a task is completely independent of processing any other. This makes the feature selection problem an ideal candidate for parallelization.

Commonly, feature selection algorithms need to compromise the goodness of their solutions in order to provide results in a practicable time. Moreover, wrapper strategies are known for producing the best results [6], however they are not usually used in high dimensional datasets because of their computational cost. This work proposes a new hybrid feature selection algorithm that uses a filter procedure to reduce the feature space and then uses a wrapper search implemented on a shared memory parallel environment to find the final solution. With this approach, we aim to achieve better solutions by using a more computationally expensive approach that explores more of the search space, combined with parallelism to speedup execution.

The remainder of the paper is structured as follows: next we present a brief review of the current state of art of feature selection algorithms. In section III we thoroughly explain each component of our approach and how they act together. Section IV details the proposed novel heuristics applied in our wrapper search component. Section V details the implementation of our strategy on a shared memory parallel architecture. Section VI assesses the parallel performance of the implemented algorithm. Section VII empirically evaluates and discusses the results attained with our strategy on several public datasets. Finally, the last section discusses future work and present conclusions on our work.

II. State of Art

Feature selection has been widely studied and as result a large number of algorithms have been proposed. These algorithms can be categorized into three groups: filter, wrapper, and embedded [4]. Filter algorithms use a classifier independent metric to evaluate either individual features or subsets. The idea is to identify which features are more relevant to the learning task. These methods assume complete independence between data and the learning algorithm. As a result, the final solution could be applied to several learning algorithms without the need to run the filter algorithm more than once. Usually the metric is fast to compute, therefore filter methods have low-computational cost. However, in most cases they fail to produce the optimal subset of features and usually perform worst than other types of feature selection algorithms. Examples of filter algorithms in literature are found in [7], [2], [6], [5].

Wrapper algorithms find the final solution using a learning algorithm as part of the evaluation criteria. The main idea of these methods is to use the learning algorithm as a *"black-box"* to guide the search for the optimal solution. The learning is applied to every candidate solution and the goodness of the subset is given according to the performance of the learning algorithm. Due to the learning algorithm being directly used on the process of selecting features, these methods tend to find better solutions. Nonetheless, the final solution only applies for the selected learning algorithm, since using a different one will most likely result on a different final solution. These methods have higher computational cost as they require training and classifying data for each candidate solution. Moreover, cross-validation techniques are commonly used, which further increases the computational cost of the algorithm [8]. Combining different search strategies with different classification algorithms results in a new wrapper method, and several examples can be found in the literature [9], [7], [10].

Embedded methods are inspired by wrapper and filter algorithms and try to use the best qualities from both types. These algorithms encapsulate feature selection with classifier construction. By doing that, the feature selection part interacts with the learning algorithm. However, it does not require training the classifier and thus they are usually faster than wrapper methods. Since these methods do not separate feature selection from learning, they are very specific to a learning algorithm. Meaning that an embedded method can only be applied to a specific learning algorithm. Tang et. al. [11] categorizes embedded methods into three groups and provides examples of algorithms for each type.

A. Hybrid Methods

Approaches that combine two categories of feature selection algorithms are gaining importance in the community. They are called hybrid methods and combine filter and wrapper methods in order to further improve the feature selection process. The idea behind these methods is to use a filter method to cut the search space into a smaller space, and then use a wrapper method to select the final solution. As examples of hybrid algorithms, we have the IFSFF algorithm [12] which uses a filter method to rank features in order to guide more efficiently the wrapper search. Another example is the Quick Branch and Bound algorithm which uses a filter approach to define subsets as starting points for a wrapper algorithm [7]. More hybrid algorithms are described in [6].

B. Parallel Feature Selection

Selecting the ideal set of features is far from an easy task. It usually requires many attempts until the desired result is attained. A conventional methodology is to change parameters on the algorithms or test different algorithms to compare results. Moreover, depending on the size of the dataset and on the algorithm chosen, a feature selection process can take a large amount of time. This triggered researchers to exploit parallelism within feature selection algorithms in order to improve their executions times. For example, Azmandian et. al. [13] used GPUs to accelerate their feature selection algorithm. Li et. al. [14] also resorted to parallelism to speed up a genetic search in the context of feature selection.

III. Overview of the Proposed Algorithm

In this section, we introduce our proposed hybrid method. It starts with a filter approach that ranks features individually. Based on a threshold and on the calculated rank, features are selected to the next phase. The goal of the filter is to use a less costly computational method to reduce the search space. Therefore, removed features are considered irrelevant and are not used any further in the next stages of the algorithm. We use Mutual Information (MI) [5] as the metric to individually rank features. The wrapper phase searches the feature space by using a novel meta-heuristic in order to find the final subset of features. Wrapper methods use a learning algorithm to evaluate the goodness of a subset. In our approach we use Support Vector Machines (SVM) [15] as our learning algorithm.

The filter and wrapper components represent the main functions of the algorithm and are responsible for selection

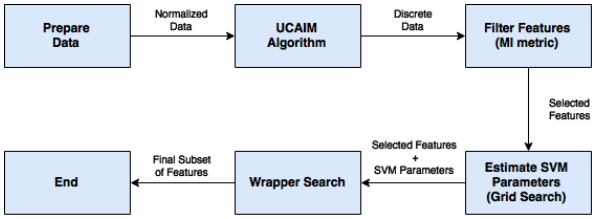


Fig. 2: Workflow of the proposed method.

features. Because, the wrapper search is the most significant contribution of this paper, we dedicate a full section to it (Section IV). Both components require some pre-processing steps that are executed by two additional algorithms: Uncertain Class Attribute Interdependency Maximization (UCAIM) [16] and Grid Search [17]. The first one is used to discretize data, which is a recommended step when using SVMs, and it is a mandatory procedure to calculate MI in cases where variables have continuous values. The Grid Search is a very popular procedure used to estimate the parameters of learning algorithms.

The workflow of our proposed method is illustrated in figure 2. We start by preparing data, then discretize it with UCAIM algorithm. Then, the feature space is reduced by eliminating features that are not able to pass the MI filter. The next step is to estimate the SVM parameters using the grid search. Finally, the algorithm runs the wrapper search which is responsible to find the subset of features that is presented as final solution. All algorithm steps are explained in more detail in the following sections.

A. Prepare Data

This is the stage where data is read from files and pre-processed. In most cases, pre-processing data includes techniques to find outliers that may jeopardize the performance of the learning algorithm. Although there are several techniques to detect and remove outliers, this process usually requires some knowledge about the dataset. This procedure is rather specific to the dataset and thus we do not include it as part of our method. Instead, we assume that the dataset is already clean and ready for the algorithm. In any case, this stage implements normalization of the feature values to a scale from 0 to 1. This is a recommended procedure in order to improve the performance of learning algorithms [2].

B. UCAIM Algorithm

In order to discretize data, we selected the UCAIM algorithm, which is an evolution of the original CAIM algorithm. Both methods have the goal to delineate intervals on data in such a way that the interdependence between features values and class labels is maximum. Despite the fact that both algorithms perform well, the evolutionary approach adds the offset component, which takes into account cases where data is unbalanced. The UCAIM algorithm has been shown to outperform the original one [16].

The UCAIM algorithm starts by setting the initial discretization scheme, D, as a set of two elements: the maximum and minimum values. Then, it proceeds to define a set of possible points. These are all the midpoints between each adjacent pair in the sorted and non-duplicate set of values. After that, UCAIM iteratively tries to add possible points to D. At each round, all possible points are added, one at the time, to D. Then, formula 1, which tries to maximize the interdependence between classes, is used to evaluate the quality of D with the recently added point. At the end of the round, the point with the best score is definitely appended to D. The process stops, when no point could improve the score that D has at the start of the round. By the end of the UCAIM algorithm, we get a discretization scheme D. Later, for each feature value, we discover the interval on D where it belongs, and convert the value to the midpoint of that interval. Thus, achieving the desired discrete data.

Algorithm 1 illustrates the steps needed to find D for a given feature F_i and its possible values V_i on a classification problem with S label classes.

$$UCAIM(F_i, D) = \frac{\sum_{r=1}^{n} \frac{max_r^2 \times Offset_r}{M_{+r}}}{n} \quad (1)$$

Where n is the number of intervals, r iterates through all intervals, max_r is the maximum value inside an interval, M_{+r} is the total number of values on the interval and offset:

$$Offset_r = \frac{\sum_{i=1}^{S} (max_r - q_{ir})}{S - 1} \quad (2)$$

where S represents the classes labels, q_{ir} are the number of values in interval r that belong to class i, and max_r is the maximum number of values in interval r across all classes. Basically, $Offset_r$ is the average difference of the number of points in all classes to the number of points in the class that has the most points in that interval.

Algorithm 1 UCAIM Algorithm

```
1:  procedure UCAIM(V_i, S)
2:      values ← REMOVEDUPLICATES(V_i)
3:      min, max ← FINDLIMITS(values)
4:      B ← GENERATEPOSSIBLEPOINTS(values)
5:      K ← 1, D ← {min, max},
6:      BestS ← 0, BestP ← {}
7:      while K ≤ S or GlobalUCAIM < BestS do
8:          GlobalUCAIM ← BestS
9:          D ← D ∪ BestP
10:         for P ∈ B do
11:             auxD ← D ∪ P
12:             auxS ← GETUCAIMSCORE(auxD)
13:             BestS, BestP ← UPDATEBEST(P, auxS)
14:         K ← K + 1
```

C. Filter Metric

In contrast to some feature selection algorithms, we do not intend to use a filter approach to find a final subset of features. Instead, our method uses it as a pre-processing step to eliminate features and make it practicable for a more intensive

search on the wrapper part. Therefore, our filter should have the following characteristics:

1) **Evaluate single features.** Several filter approaches evaluate subsets of features. However, to keep a low computational cost, we avoid searching for feature subsets and evaluate features only individually.
2) **Not very restrictive.** The percentage of removed features should not be very large. Although as less features pass the filter the faster the wrapper ends, it is difficult to accurately assess the quality of a feature just by using a filter metric. It has been shown that features considered irrelevant when individually evaluated, are in fact important when inserted into a specific set of features [2]. Hence, to avoid compromising the performance of the final solution, it is important to avoid removing a large number of features at this stage.

There are several algorithms in the literature that fulfil the first requirement of our list, these methods are called univariate [12]. Two of the most commonly used metrics of this type are Mutual Information (MI) and Pearson Correlation Coefficients (PCC) [2]. Both metrics measure the dependence between two variables. Nonetheless, there is a key difference between them. MI measures the general dependence between the variables while PCC measures linear dependence. Li et al. [18] tested this property and concluded that this makes MI a better metric. Based on that work and on the amount of other works that use MI [5], we decided to use it as the selected metric to our filter approach.

Calculating the MI score for every feature does not remove features by itself, so in the next step we define a strategy that filters features taking into account the required second characteristic. The idea is to define a threshold and to remove features for which the MI score is bellow that threshold. Since MI scores diverge a lot when changing datasets, it is not possible to use a fixed threshold. Instead, we define the threshold as a percentage of the maximum MI score, and leave out features with MI below the threshold.

D. Grid Search

As previously mentioned, we use the SVM as our learning algorithm. As we will show in the next sections, SVMs have some parameters that must be tuned in order to provide better results. However, it is not uncommon for researchers to not knowing which parameters to use. On our method, we let users define the parameters; yet, if they do not specify them, the algorithm estimates the best parameter set to use. We test several parameters and select the ones that provide the best results using a grid search. This method tests parameters in two ranges. First, a large scale range and then, after choosing one value for the larger range, a smaller scale range is used. For example, suppose that for a parameter i the first possible values are $L_i = \{..., 2^7, 2^9, 2^{11}, 2^{13}, ...\}$. Now imagine that the selected value from the L_i is 2^9, hence the algorithm proceeds to search the final parameter in the following range $S_i = \{..., 2^{8.5}, 2^{8.75}, 2^9, 2^{9.25}, 2^{9.5}, ...\}$.

Typically grid search is applied to tune the classifier using the set of features. However, because we use the classifier to select a subset of features, we need to choose the parameters before knowing the feature set to be used. In addition, it is impracticable to perform a grid search for every subset being evaluated during the wrapper search. Thus, we perform a grid search on n randomly generated feature subsets after the filter and before starting the wrapper search. The best set of parameters for each feature subset counts as a vote, and the parameter set with most votes is selected. In cases where the highest number of votes is the same for more than one set of parameters, we generate n new random subsets and the test is repeated for the tied set of parameters. This process is repeated until there are no more ties.

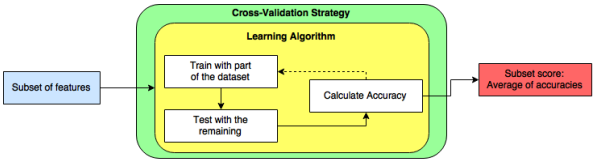


Fig. 3: Subset Evaluation

IV. Wrapper Search

Our proposed feature selection algorithm was designed to make use of existing algorithms for most of the tasks that must be performed. However, the wrapper search is a new meta-heuristic which, together with the strategy for its parallel execution, makes it a main contribution of this work. This is the most complex part of our algorithm and the functions used at this stage define its computational cost and its ability to find good solutions. For the sake of understanding, we further divided its explanation into three sections: learning algorithm, search strategy, successor generation. The following section complements the description with the parallel strategy.

A. Learning Algorithm

We use SVM as the learning algorithm to our method based on the work [15] in which the authors compare several learning algorithms in the context of classification problems. They concluded that SVM in general outperforms other classifiers. SVMs can have different kernels, and each one defines a distinct way to map data into higher dimensions. In order to select an adequate kernel, size and type of data should be taken into account [17]. The number of parameters to be estimated also depends on the kernel selection.

The function of the SVM in our approach is to evaluate the goodness of a subset of features. For each tested subset, we train the classifier with a part of the data and test it with the remaining. This process is commonly known as cross-validation [17] and the number of times it is performed for a subset depends on the defined number of folds. In the end of the whole process the algorithm gets a score for the subset. This score is the average of the accuracy obtained for each fold. This process is illustrated by figure 3.

B. Search strategy

Although several search strategies exist and have been used on wrapper approaches, we decided to implement a new meta-heuristic that explores the search space more thoroughly. The idea was to create a different strategy to search for solutions,

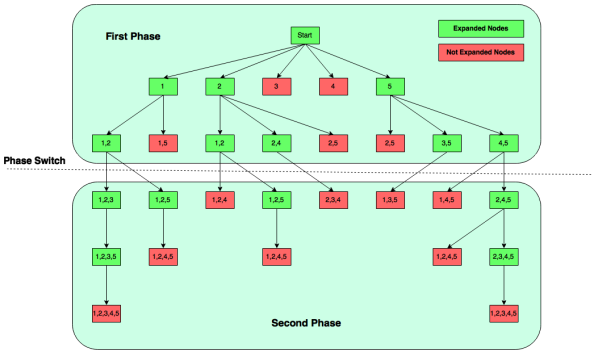


Fig. 4: Example of the implemented wrapper search

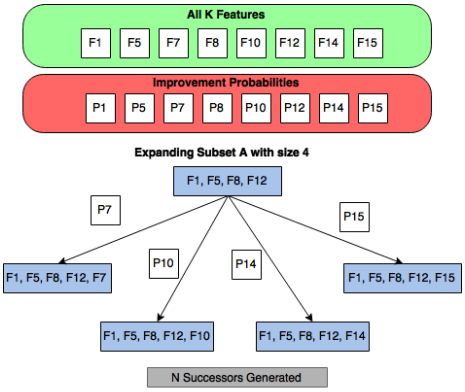


Fig. 5: Successor Generation using improvement probabilities

while maintaining a structure that could be easily run with multiple processors. The proposed search organizes subsets as nodes on a tree whose first level is composed by n starting subsets, each with a single feature not removed by the filter. From now on, we use subsets and nodes interchangeably.

The innovative idea of the proposed search is to explore broadly different regions of the search space, looking for the areas of higher classification accuracy, and then focus on searching the local maxima in each region. Thus, the search strategy doesn't have an uniform behaviour, but is divided into two stages. First, we gather as many good solutions as possible. Then, we improve them up to the best score they can reach on the second stage. The transition between stages takes place when subsets reach a certain number of features. Figure 4 illustrates an example of the implemented wrapper search.

In the first stage, the decision to expand a node or not is based on the distance from the obtained score to the global best. In this step, a threshold is defined and nodes whose difference of score to the global best is lower than the threshold are expanded further. During the second stage, nodes are searched using a depth first strategy and they are expanded while they still improve the score of their parent. The search stops when there are no more nodes left to explore. In this stage, a mechanism cuts subsets with a certain probability to avoid excessive work. The probability of each subset being cut is defined based on how distant its score is from the global best, according to the following table:

% Distance to global	Cut probability
$d < 0.5$	0%
$0.5 \leq d < 1.0$	25%
$1.0 \leq d < 1.5$	50%
$d \geq 1.5$	75%

The mechanism executes every t seconds, where t is a value which can be user defined.

The defined threshold in the first stage and the size at which stages switch have a great impact on the amount of nodes explored in the search. Thus, it is possible to define how restrictive the search is by manipulating these parameters.

C. Successor Generator

Finally, we explain how to generate successors of a subset. The idea is to add to a subset S_j several features that are not yet part of it. Each feature can be added with a specific probability, according to a likelihood of improving the evaluation score, estimated in a pre-processing step described below. If k features are selected to be added, then k new successors of S_j are generated. Each one represents S_j combined with a new feature. By doing so, our method increases the likelihood of features with high improvement score being added to new subsets. Figure 5 illustrates this process.

The likelihood of a certain feature contributing to an improvement in the evaluation criteria is estimated in a pre-search phase. The procedure starts by generating n random subsets, and evaluates each one of them using the subset evaluation procedure. Then, for every F_i we test how the score of a subset improves when F_i is either added or removed. In the end we count how many times F_i improved subsets to calculate the likelihood of improvement. These values are then used when the wrapper search decides to expand a node.

It is clear that our successor generator function may generate repeated subsets. Since the function that evaluates subsets is deterministic, testing a subset more than once is a waste of computations. But the search strategy does not handle the problem. Thus, we added a mechanism to avoid work repetition to the process of generating successors. The hash value of every tested subset is added to an hash table. Then, every time a new subset is generated, a look up in an hash table checks whether it has been tested before, if the answer is positive, then the successor is discarded.

D. Overview of the wrapper search

In the previous sections, we discussed the several components of the proposed wrapper search. It is also important to understand how they act together. Algorithm 2 details the overall wrapper search strategy.

V. Parallelized Method

The UCAIM, Filter and Grid Search parts of our proposed approach are not computationally costly, however, the wrapper part is. Our search strategy is very intensive in the number of

Algorithm 2 Search Strategy

```
1:  procedure SEARCH(size, W, data, hashTable, probs )
2:      lastStage ← False
3:      while W ≠ empty do
4:          s ← REMOVELAST(W)
5:          if SIZE(s) ≥ size then
6:              lastStage ← True
7:          score ← SVMCLASSIFICATION(s, data)
8:          if WORTHEXPAND(score, s, lastStage) then
9:              newN ← EXPAND(s, hashTable, probs)
10:             UPDATEGLOBAL(score)
11:             if lastStage then
12:                 W ← W ∪ newN
13:             else
14:                 W ← newN ∪ W
```

explored nodes. Thus, finding a way to realise the search in parallel will obviously help in reducing the computation time. Although the first three parts are not very expensive, we have also parallelized them. On UCAIM and Filter parts, our method achieves parallelism by dividing features by processors. By contrast, on the Grid Search the random generated subsets are divided by the processors. Since in all parts, problems were divided into smaller tasks and each one of them is independent, this presented no major difficulty.

On the other hand, the wrapper presents a challenge that can compromise the performance of the algorithm. The wrapper requires some information such as the hash table and best score to be global accessible and constantly updated. Keeping such structures always updated when executing with multiple processes may become computationally costly in terms of performance.

A. Parallel Scheme

We can visualise the problem of our wrapper search as a set of tasks. Each task is the process of getting a subset from a work list, evaluate it, decide to either expand it or not, and in the case of expanding, generate new subsets, remove those that have been already been tested and in the end, add all the remaining ones to the work list.

The idea of our parallel wrapper is to define local work lists on every process and use them to store subsets that still have to be processed. Then, iteratively, having each process computing a task for a subset. The processes would get the subset from their local work list and add new work there as well. Additionally, in order to remove new generated subsets already tested, a global hash table that is accessible by all process is used. There, the hash value of every tested subset is stored.

At the beginning of the search, the individual subsets of features are equally distributed among all processes and each one starts the computations as previous described. During the search, some processes will run out of work while there are others with a lot of work left. In order to provide a good work balance, a process P_i that hasn't any work, is able to request it from other process P_j. When P_j sees the request, it will send part of its work back to P_i.

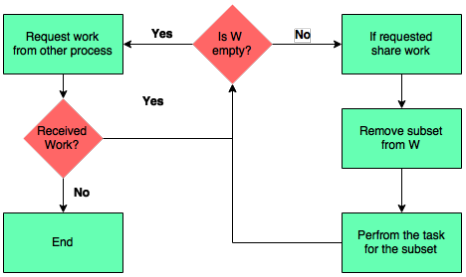


Fig. 6: Workflow of a single process

The overall workflow for a single process is illustrated by figure 6, where W represents the local work list.

B. Global information

To achieve the desired parallelism, we decided to use a shared memory environment where processes have access to the same memory addresses simultaneously. This paradigm provides a cheap way of communication between processes and avoids redundant copies of informations across multiple processes.

Global information is required to keep an hash table updated as well as to store the best score found during the search. To access and update this information we had to use different strategies. The best score is a single variable defined in shared memory. The access to it is controlled by a mutual exclusion mechanism to avoid race conditions.

The hash table required some additional work, using a unique hash table and have every process constantly updating it and searching on it for repetitions is not an option. Mainly because processes would start writing in the same memory spaces. Thus, the hash table would become incoherent, probably leading to the loss of entries which would result in lots of repeated work. On the other hand, using a mutual exclusion mechanism is not viable because waiting for the access to the memory would drastically decrease the parallel performance.

In order to make this work we divide the hash table into p partitions, each one is assigned to a process P_i. Only process P_i is allowed to write in the partition assigned to it. However, any process is free to read from any partition at any given time. By doing that, each process stores the hash of its tested subsets in its own partition. Then, when it has to check if a new generated subset is new, it can read from every partition without having to wait for permission. Mutual exclusion was avoided and memory coherence is guaranteed because for every single memory address, only one process is allowed to write on it.

VI. Performance of the Parallelization

In this section we assess the performance of the implemented parallel search with the increase in the number processes used in the computation. It is common to run the same program several times using a different number of processing units and then get the execution times of each one of them. However, in our case, changing the number of processes also changes the amount of visited nodes. For this reason and to

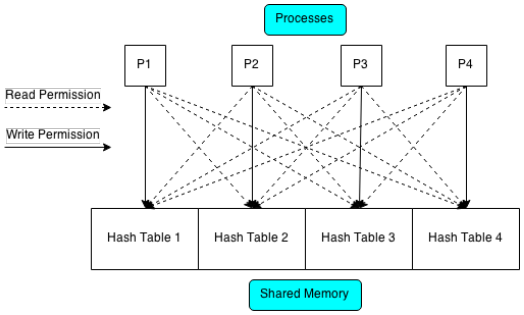


Fig. 7: Hash scheme in shared memory.

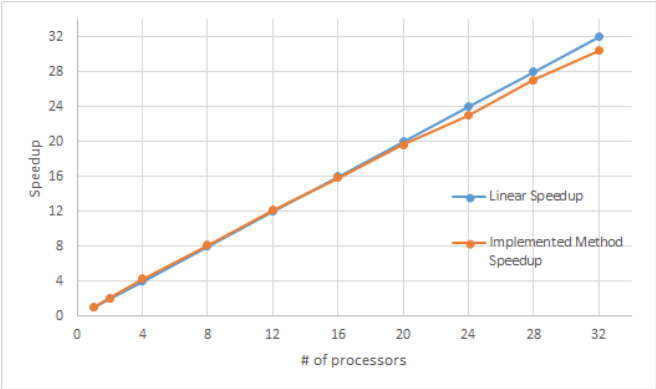


Fig. 8: Comparing Speedup against linear

perform all tests with the same conditions, we decided to test the worst case scenario when 20 features reach the wrapper search. The worst case scenario for our search is defined by the following properties:

1) Every subset is expanded
2) When expanded, each subset is combined with all possible features
3) The cutting probability for all subsets is 0%.

More accurately, this results in an exhaustive search with a very large number of attempts resulting from repeated work. Also, this means that regardless of the number of processors, all tests would search the same feature space which is a total of $2^{20} - 1 = 1048575$ subsets. Table I illustrates the results of our test and figure 8 compares the obtained speedup against the linear.

TABLE I: 20 Feature Exhaustive Search Data

# Processors	Execution Time (sec)	Speedup	Number of tests
1	133984	1.00	1048575
2	64148	2.09	1048575
4	31427	4.26	1048575
8	16406	8.17	1048575
12	11055	12.12	1048576
16	8457	15.84	1048575
20	6812	19.67	1048575
24	5825	23.00	1048578
28	4948	27.08	1048581
32	4409	30.39	1048579

The results show an almost linear speedup in performance of the parallel search when 32 cores are used. In some cases, the implemented strategy was able to achieve speedups greater than the ideal values. Moreover, it was able to efficiently avoid repeated work by having every process keeping track of their tested subsets in its partitioned hash table and checking all the partitioned hash to avoid repeating them. In the worst case, only 6 subsets out of the 1048575 total subsets were tested more than once.

VII. Feature Selection Results

In 2003, the NIPS[19] feature selection challenge was created and its aim was to find which algorithm would preform better in terms of classification. The contest consists of five public datasets, each divided into three sets: train, validation, and test. For the first two, it is possible to access both data and labels, while on the last one only data is available. The idea of the challenge is to use feature selection and machine learning techniques on the train and validation sets in order to construct a classifier that is able to accurately predict the labels of the test set. At the end of the process, one can submit the generated predictions on the website and it provides feedback about the results. Nowadays the challenge is still open and it allows researchers to benchmark their systems. Thus, we decided to use it in order to test the performance of our approach. The following table presents the characteristics of the five datasets[19]:

Dataset	Type	Features	Train Examples	Validation Examples	Test Examples
Arcene	Dense	10000	100	100	700
Dexter	Dense	5000	6000	1000	6500
Gisette	Sparse integer	20000	300	300	2000
Dorothea	Sparse binary	100000	800	350	800
Madelon	Dense	500	2000	600	1800

Regarding user defined parameters, we used 0.5 for the search threshold, 8 for the size at which the search changes, and 900 seconds for the cutting mechanism. In addition to that, we executed each test using 62 cores and RBF as the SVM's kernel. These were the fixed values for all tests, however some parameters such as cross-validation technique and percentage of the filter had to be adapted to each dataset. The following table presents the used parameters and the results obtained:

Dataset	Filter Threshold	Features Post-Filter	Cross-Validation	Size Final Subset	Final Score	Time (sec)
Arcene	0.50	232	Leave-one-out	14	99.00	2156
Dexter	0.94	364	Leave-one-out	72	98.50	32539
Gisette	0.97	121	5 folds	85	97.32	53560
Dorothea	0.90	450	5 folds	30	97.63	32962
Madelon	0.97	255	10 folds	14	87.10	33117

After obtaining a final subset for each dataset, we used it to train a classifier only using data from the train set. Then, we predicted the labels for every one of the three sets: train, validation, and test. Later, results were submitted to the NIPS website and the accuracy of our predictions as well as the rank among all the submissions are illustrated on the following table:

The results we obtained ranked among the top 60% for each dataset. This outcome came at no surprise considering

Dataset	Accuracy Train	Accuracy Validation	Accuracy Test	Rank
Arcene	99.00	82.00	74.56	892/1503
Dexter	98.33	83.67	81.65	819/1007
Gisette	98.97	96.80	96.67	465/932
Dorothea	95.63	94.29	77.18	475/812
Madelon	93.35	87.50	88.67	344/1059

that the goal of the challenge set used is directed to evaluate the overall performance of the classification system. Despite being part of the machine learning workflow, feature selection is not the whole process and usually several more techniques such as outlier detection, noisy examples removal and generation of synthetic data are required [20]. Moreover, knowledge about the specific problem at hand can improve the generalisation result by targeting feature choice, or through the use of another metric for calculating the feature subset score in the search. In our experiments, we focused on testing the ability of our search algorithm to find what was defined as a good subset according to the score, which was the accuracy of the learning algorithm on the training sets. Although cross-validation strategies were used to improve generalization, they were not enough, and in general the classifier was not able to predict well on unseen data. Nevertheless, we are quite happy with the results on the NIPS challenge, because we could confirm that our hybrid approach, without any further analysis of the dataset nor additional techniques, was able by itself to produce quite acceptable results.

In terms of feature selection, our approach, in most cases, was able to produce a final subset of features that was much smaller than the original set of features and that had very high accuracy score. Although we cannot guarantee that the optimal subset was found, it would require to test the whole search space, our algorithm was able to achieve results near the perfect score in 4 out of the 5 tests. Moreover, the results were obtained in a practicable amount of time, considering the size of the datasets and the cross-validations methods used.

VIII. Conclusion

In this paper, we propose a new hybrid feature selection approach that resorts to a novel parallel search strategy to speedup execution. It starts by reducing the feature space using a filter and then uses a wrapper method to find the final solution. Because of their high-computation cost, wrapper methods are not commonly used on high dimensional datasets. In our experimental evaluation, we tested high dimensional datasets, thus showing how it is possible to take advantage of parallelism to thoroughly search larger spaces in a practicable amount of time. Our initial results show an almost linear speedup up to 32 cores while being able to find solutions with near perfect score.

We are aware that the accuracy of the classifier can be improved by employing better generalization methods, but that was not the goal of the task at hand. We intend to experiment on more problems and with other filter methods, namely PCC [2] and Relief [4], as well as with other classifiers Decision Trees and kNN[20], methods that fit well with the approach we proposed.

Acknowledgment

We would like to thank Isabelle Guyon and Lukasz Romaszko who kindly reopening the NIPS 2003 challenge.

This work was partially supported by national funds through project VOCE (PTDC/EEA-ELC/121018/2010), and in the scope of R&D Unit UID/EEA/50008/2013, funded by FCT/MEC through national funds and when applicable co-funded by FEDER/PT2020 partnership agreement.

References

[1] J. Manyika and et. al., "Big data: The next frontier for innovation, competition, and productivity," *McKinsey Global Institute*, no. May, p. 156, 2011.

[2] I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *J. Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[3] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[4] V. Kumar and S. Minz, "Feature Selection: A literature Review," *Smart CR*, vol. 4, no. 3, pp. 211–229, 2014.

[5] J. R. Vergara and P. A. Estévez, "A review of feature selection methods based on mutual information," *Neural Computing and Applications*, vol. 24, no. 1, pp. 175–186, 2013.

[6] H. Liu and L. Yu, "Toward Integrating Feature Selection Algorithms for Classification and Clustering," *IEEE Trans. Knowledge and Data Engineering*, vol. 17, no. 4, pp. 491–502, 2005.

[7] L. Molina, L. Belanche, and A. Nebot, "Feature selection algorithms: A survey and experimental evaluation," in *ICDM'2002*, 2002, pp. 306–313.

[8] K. Dunne, P. Cunningham, and F. Azuaje, "Solutions to Instability Problems with Sequential Wrapper-based Approaches to Feature Selection," *J. Machine Learning Research*, pp. 1–22, 2002.

[9] R. Kohavi and G. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.

[10] Y. Saeys, I. Inza, and P. Larrañaga, "A review of feature selection techniques in bioinformatics," *Bioinformatics*, vol. 23, no. 19, pp. 2507–2517, 2007.

[11] J. Tang, S. Alelyani, and H. Liu, "Feature Selection for Classification: A Review," in *Data Classification: Algorithms and Applications*, C. Aggarwal, Ed. CRC Press, 2014.

[12] J. Xie and W. Xie, "A Novel Hybrid Feature Selection Method Based on IFSFFS and SVM for the Diagnosis of Erythemato-Squamous Diseases," *JMLR Workshop on Applications of Pattern Analysis*, vol. 11, pp. 142–151, 2010.

[13] F. Azmandian, A. Yilmazer, J. G. Dy, J. A. Aslam, and D. R. Kaeli, "GPU-Accelerated Feature Selection for Outlier Detection Using the Local Kernel Density Ratio," in *ICDM'2012*, 2012, pp. 51–60.

[14] R. Li, J. Lu, Y. Zhang, and T. Zhao, "Dynamic Adaboost learning with feature selection based on parallel genetic algorithm for image annotation," *Knowl.-Based Syst.*, vol. 23, no. 3, pp. 195–201, 2010.

[15] V. G. C., "Performance Evaluation of Machine Learning Classifiers in Sentiment Mining," *Int. Journal of Computer Trends and Technology (IJCTT)*, vol. 4, no. 6, pp. 1783–1786, 2013.

[16] J. Ge, Y. Xia, and Y. Tu, "A Discretization Algorithm for Uncertain Data," in *DEXA'2010, LNCS 6262*, 2010, pp. 485–499.

[17] C. Hsu, C. Chang, and C. Lin, "A Practical Guide to Support Vector Classification," *BJU international*, vol. 101, no. 1, pp. 1396–400, 2008.

[18] W. Li, "Mutual information functions versus correlation functions," *Journal of Statistical Physics*, vol. 60, no. 5-6, pp. 823–837, 1990.

[19] "NIPS 2003 feature selection challenge," http://www.nipsfsc.ecs.soton.ac.uk/results/?ds=overall, accessed: 2015-04-23.

[20] S. B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques," *Informatica (Slovenia)*, vol. 31, no. 3, pp. 249–268, 2007.

Appendix C

Appended Images

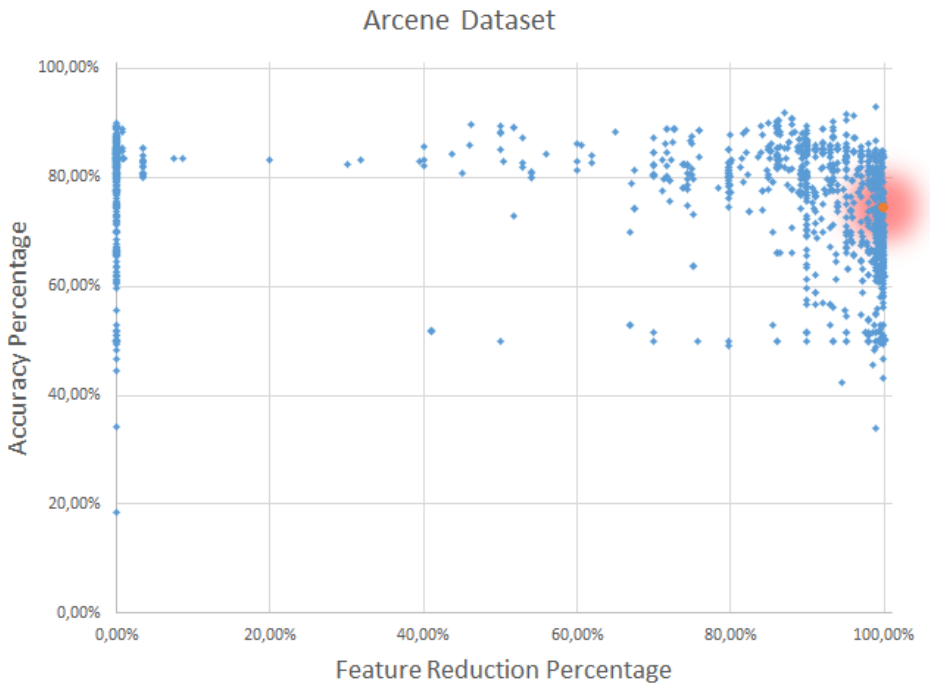


Figure C.1: Correlating feature reduction with accuracy for every submission in the Arcene dataset.

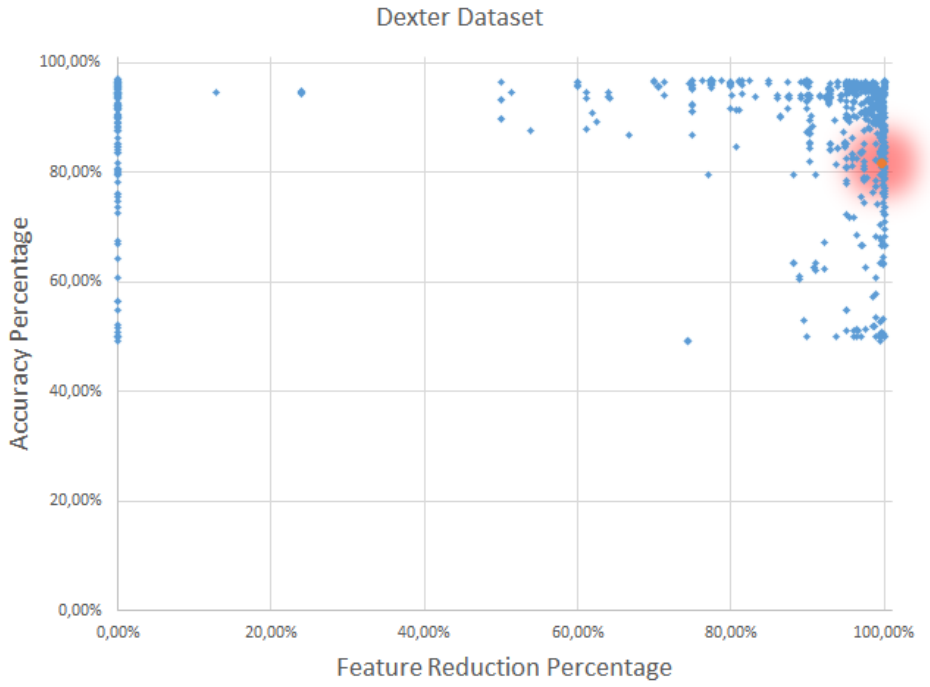


Figure C.2: Correlating feature reduction with accuracy for every submission in the Dexter dataset.

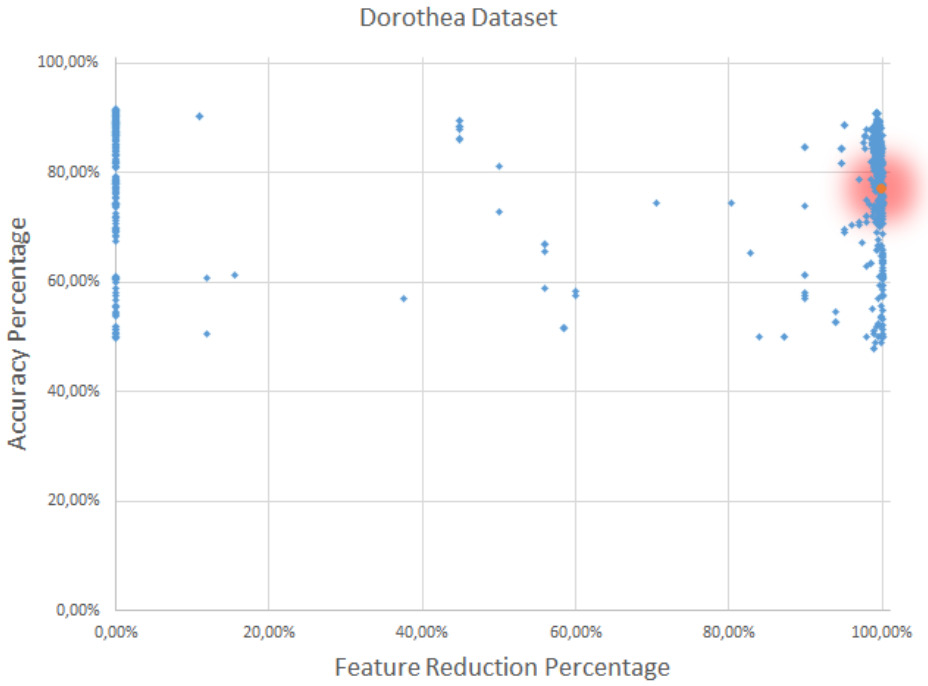


Figure C.3: Correlating feature reduction with accuracy for every submission in the Dorothea dataset.

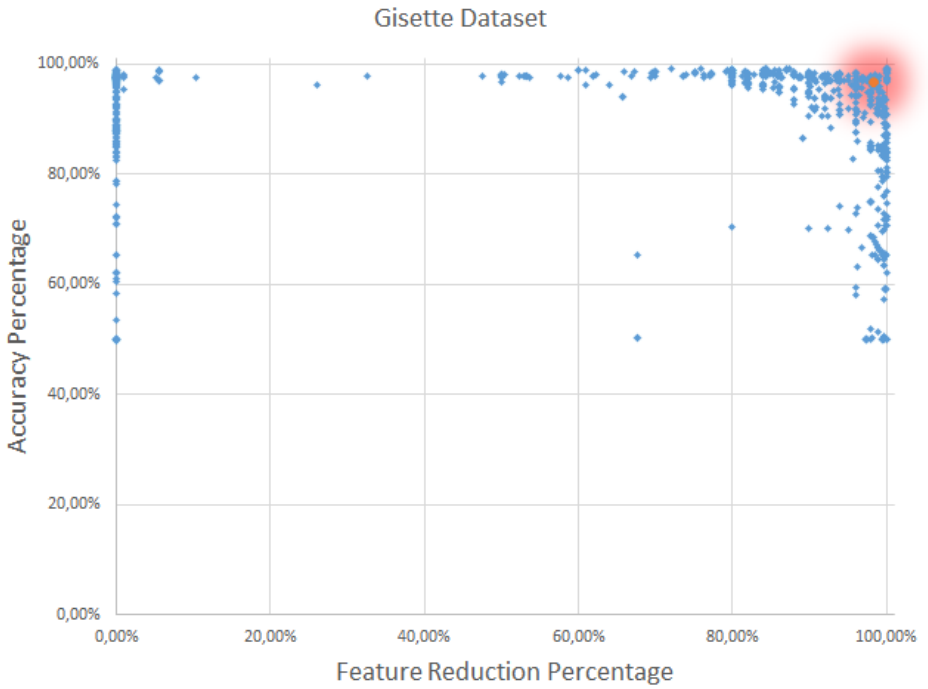


Figure C.4: Correlating feature reduction with accuracy for every submission in the Gisette dataset.

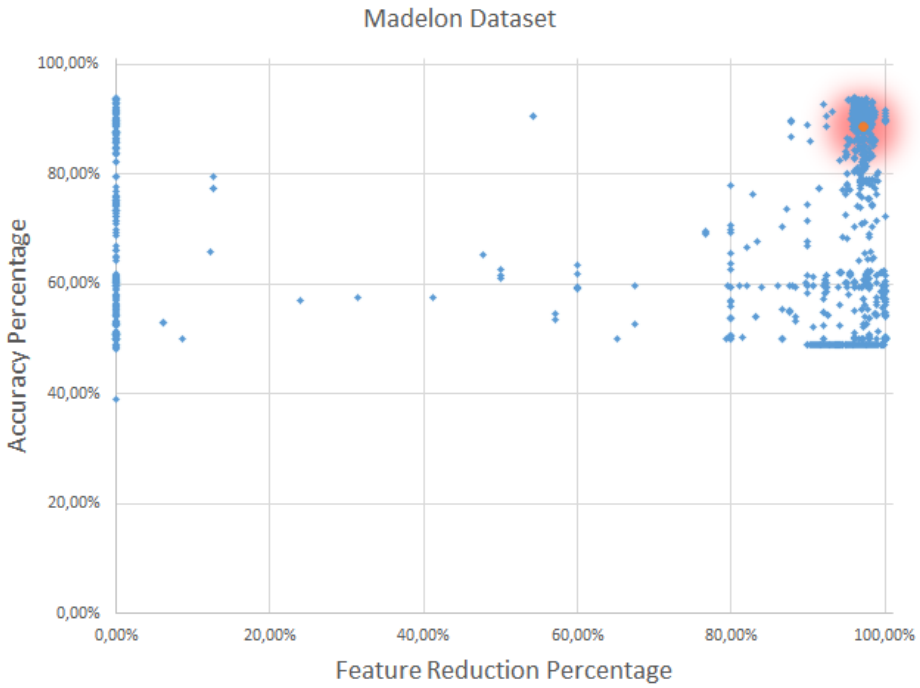


Figure C.5: Correlating feature reduction with accuracy for every submission in the Madelon dataset.

References

[1] Charu C Aggarwal and Philip S Yu. Outlier detection for high dimensional data. In *ACM Sigmod Record*, volume 30, pages 37–46. ACM, 2001.

[2] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[4] Fatemeh Azmandian, Ayse Yilmazer, Jennifer G Dy, Javed Aslam, David R Kaeli, et al. Gpu-accelerated feature selection for outlier detection using the local kernel density ratio. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 51–60. IEEE, 2012.

[5] Olivier Beaumont, Arnaud Legrand, and Yves Robert. The master-slave paradigm with heterogeneous processors. *Parallel and Distributed Systems, IEEE Transactions on*, 14(9):897–908, 2003.

[6] Llus A Belanche and Félix Fernando González. Review and evaluation of feature selection algorithms in synthetic problems. *arXiv preprint arXiv:1101.2320*, 2011.

[7] Waad Bouaguel and Ghazi Bel Mufti. An improvement direction for filter selection techniques using information theory measures and quadratic optimization. *arXiv preprint arXiv:1208.3689*, 2012.

[8] Brad Brown, Michael Chui, and James Manyika. Are you ready for the era of 'big data'. *McKinsey Quarterly*, 4:24–35, 2011.

[9] Yi-Wei Chen and Chih-Jen Lin. Combining svms with various feature selection strategies. In *Feature extraction*, pages 315–324. Springer, 2006.

[10] CodaLab Competitions feature selection challenge. `https://www.codalab.org/competitions/3931?secret_key=d6c218a3-3b83-4eed-8e39-5b895c5a5e35#results`. Accessed: 2015-05-14.

[11] Jerffeson Teixeira de Souza, Stan Matwin, and Nathalie Japkowicz. Parallelizing feature selection. *Algorithmica*, 45(3):433–456, 2006.

[12] Peter J Denning and Walter F Tichy. Highly parallel computation. *Science*, 250(4985):1217–22, 1990.

[13] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[14] Ciro Donalek, S George Djorgovski, Ashish Mahabal, Matthew J Graham, Alan J Drake, Thomas J Fuchs, Michael J Turmon, A Arun Kumar, N Sajeeth Philip, Michael Ting-Chang Yang, et al. Feature selection strategies for classifying high dimensional astronomical data sets. In *Big Data, 2013 IEEE International Conference on*, pages 35–41. IEEE, 2013.

[15] Wlodzislaw Duch, Tadeusz Wieczorek, Jacek Biesiada, and Marcin Blachnik. Comparison of feature ranking methods based on information entropy. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 2, pages 1415–1419. IEEE, 2004.

[16] Włodzisław Duch, Tomasz Winiarski, Jacek Biesiada, and Adam Kachel. Feature ranking, selection and discretization. In *Proceedings of Int. Conf. on Artificial Neural Networks (ICANN)*, pages 251–254, 2003.

[17] Kevin Dunne, Padraig Cunningham, and Francisco Azuaje. Solutions to instability problems with sequential wrapper-based approaches to feature selection. *Journal of Machine Learning Research*, 2002.

[18] Jerome H Friedman. On bias, variance, 0/1—loss, and the curse-of-dimensionality. *Data mining and knowledge discovery*, 1(1):55–77, 1997.

[19] Jiaqi Ge, Yuni Xia, and Yicheng Tu. A discretization algorithm for uncertain data. In *Database and Expert Systems Applications*, pages 485–499. Springer, 2010.

[20] GIT mitws repository. `https://github.com/JSilva90/MITWS`. Accessed: 2015-09-14.

[21] Puneet Gupta, David Doermann, and Daniel DeMenthon. Beam search for feature selection in automatic svm defect classification. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 212–215. IEEE, 2002.

[22] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[23] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[24] Hongwei Hao, Cheng-Lin Liu, and Hiroshi Sako. Comparison of genetic algorithm and sequential search methods for classifier subset selection. In *null*, page 765. IEEE, 2003.

[25] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[26] Victoria J Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[27] Thanyaluk Jirapech-Umpai and Stuart Aitken. Feature selection and classification for microarray data analysis: Evolutionary methods for identifying predictive genes. *BMC bioinformatics*, 6(1):148, 2005.

[28] Leonard Kleinrock. Analysis of a time-shared processor. *Naval research logistics quarterly*, 11(1):59–73, 1964.

[29] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[30] Ron Kohavi and George H John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1):273–324, 1997.

[31] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques, 2007.

[32] Vipin Kumar and Sonajharia Minz. Feature selection: A literature review. *Smart CR*, 4(3):211–229, 2014.

[33] Lukasz Kurgan, Krzysztof J Cios, et al. Caim discretization algorithm. *Knowledge and Data Engineering, IEEE Transactions on*, 16(2):145–153, 2004.

[34] Thomas Navin Lal, Olivier Chapelle, Jason Weston, and André Elisseeff. Embedded methods. In *Feature extraction*, pages 137–165. Springer, 2006.

[35] Riccardo Leardi, R Boggia, and M Terrile. Genetic algorithms as a strategy for feature selection. *Journal of chemometrics*, 6(5):267–281, 1992.

[36] Wentian Li. Mutual information functions versus correlation functions. *Journal of statistical physics*, 60(5-6):823–837, 1990.

[37] Chih Jen Lin, Chih-Wei Hsu, and Chih-Chung Chang. A practical guide to support vector classification. *National Taiwan U., www. csie. ntu. edu. tw/cjlin/papers/guide/guide. pdf*, 2003.

[38] Huan Liu and Lei Yu. Toward integrating feature selection algorithms for classification and clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 17(4):491–502, 2005.

[39] Félix Garcıa López, Miguel Garcıa Torres, Belén Melián Batista, José A Moreno Pérez, and J Marcos Moreno-Vega. Solving feature subset selection problem by a parallel scatter search. *European Journal of Operational Research*, 169(2):477–489, 2006.

[40] Sebastián Maldonado and Richard Weber. A wrapper method for feature selection using support vector machines. *Information Sciences*, 179(13):2208–2217, 2009.

[41] Mvurya Mgala and Audrey Mbogho. Selecting relevant features for classifier optimization. In *Advanced Machine Learning Technologies and Applications*, pages 211–222. Springer, 2014.

[42] Luis Carlos Molina, Lluís Belanche, and Àngela Nebot. Feature selection algorithms: A survey and experimental evaluation. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 306–313. IEEE, 2002.

[43] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[44] Neural Information Processing Systems Conference feature selection challenge. `http://web.archive.org/web/20130512034606/http://www.nipsfsc.ecs.soton.ac.uk/datasets`. Accessed: 2015-05-14.

[45] Thomas Oommen, Debasmita Misra, Navin KC Twarakavi, Anupma Prakash, Bhaskar Sahoo, and Sukumar Bandopadhyay. An objective analysis of support vector machine based classification for remote sensing. *Mathematical geosciences*, 40(4):409–424, 2008.

[46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[47] Python GIL global interpreter lock. `https://wiki.python.org/moin/GlobalInterpreterLock`. Accessed: 2015-03-19.

[48] Python Manager python documentation. `https://docs.python.org/2/library/multiprocessing.html#managers`. Accessed: 2015-03-19.

[49] Python python 2.7 release. `https://www.python.org/download/releases/2.7/`. Accessed: 2015-06-15.

[50] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[51] K Schwab, A Marcus, JO Oyola, W Hoffman, and M Luzi. Personal data: The emergence of a new asset class. In *An Initiative of the World Economic Forum*, 2011.

[52] Scikit-learn machine learning in python. `http://scikit-learn.org/stable/`. Accessed: 2015-06-17.

[53] Petr Somol, Pavel Pudil, and Josef Kittler. Fast branch & bound algorithms for optimal feature selection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(7):900–912, 2004.

[54] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[55] Jiliang Tang, Salem Alelyani, and Huan Liu. Feature selection for classification: A review. *Data Classification: Algorithms and Applications*, page 37, 2014.

[56] UCI machine learning repository. `https://archive.ics.uci.edu/ml/index.html`. Accessed: 2015-05-14.

[57] Aki Vehtari and Jouko Lampinen. Bayesian input variable selection using posterior probabilities and expected utilities. *Report B31*, 2002.

[58] Jorge R Vergara and Pablo A Estévez. A review of feature selection methods based on mutual information. *Neural Computing and Applications*, 24(1):175–186, 2014.

[59] G Vinodhini and RM Chandrasekaran. Performance evaluation of machine learning classifiers in sentiment mining. *International Journal of Computer Trends and Technology*, 4, 2013.

[60] Juanying Xie, Weixin Xie, Chunxia Wang, and Xinbo Gao. A novel hybrid feature selection method based on ifsffs and svm for the diagnosis of erythemato-squamous diseases. In *WAPA*, pages 142–151, 2010.

[61] Eric P Xing, Michael I Jordan, Richard M Karp, et al. Feature selection for high-dimensional genomic microarray data. In *ICML*, volume 1, pages 601–608. Citeseer, 2001.

[62] Enzhe Yu and Sungzoon Cho. Ga-svm wrapper approach for feature subset selection in keystroke dynamics identity verification. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 3, pages 2253–2257. IEEE, 2003.

[63] Silvia Casado Yusta. Different metaheuristic strategies to solve the feature selection problem. *Pattern Recognition Letters*, 30(5):525–534, 2009.

[64] Rong Zhou and Eric A Hansen. Combining breadth-first and depth-first strategies in searching for treewidth. In *IJCAI*, volume 9, pages 640–645, 2009.