UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione - DEI
Corso di Laurea Magistrale in Ingegneria Informatica

SNNAP: Solver-based Nearest Neighbor for Algorithm Portfolios

Laureando: Marco Collautti

Relatore:
Prof. Matteo Fischetti

Correlatori:
Dr. Yuri Malitsky
Prof. Barry O'Sullivan

Anno Accademico 2012-13

A Mamma e Papà
e ai miei fratelli

Abstract

The success of portfolio algorithms in competitions in the area of combinatorial problem solving, as well as in practice, has motivated interest in the development of new approaches to determine the best solver for the problem at hand. Yet, although there are a number of ways in which this decision can be made, it always relies on a rich set of features to identify and distinguish the structure of the problem instances.

In this thesis, however, it is firstly shown how not all the features in the problem have the same relevancy. Then it is presented how one of the most successful portfolio approaches, ISAC, can be augmented by taking into account the past performance of solvers as part of the feature vector. Testing on a variety of SAT datasets, it is here proved how the new formulation continuously outperforms an unmodified/standard version of ISAC.

This thesis presents a novel strategy that tackles the problem of algorithm portfolios applying machine learning techniques and improves a state-of-the-art portfolio algorithms approaches (ISAC).

Starting from the basic assumptions behind the standard ISAC methodology that the given collection of features can be used to correctly identify the structure of each instance, an important step has been to be able to improve clustering, grouping together instances that prefer to be solved by the same solver. It has been created, and here is presented, a methodology to redefine the feature vector including past performance of the solvers. The final result is named *SNNAP: Solver-based Nearest Neighbor for Algorithm Portfolios* which is here presented.

Acknowledgements

The work for this thesis has been carried out at Cork Constraint Computation Centre (4C) at the department of Computer Science of University College of Cork (Ireland) since January 2013. First of all I will be forever grateful to the Erasmus program that gave me the opportunity to spend 10 amazing months of my life in another european country. I will forever thank all the people that I met in Ireland: you made my adventure something that I will remember forever. I truly believe in our generation: the future will be in our hands; we accepted the challenge of a united European Community and to all of you I truly wish to be the successful managers of the future.

In Cork I had the opportunity to be, for the first time, in a research group and I would like to thank Professor Barry O'Sullivan that, despite his very busy agenda, accepted me to work with him and to be part of his great group. I want to thank my supervisor Professor Matteo Fischetti that although the distance gave me continuous support and encouragements. The biggest thank is addressed to Doctor Yuri Malitsky that, despite the titles, has been the real supervisor for this work. With your infinite patience and guidance you helped me in a topic that was completely obscure to me at the beginning. When you accepted me as a student you truly accepted a challenge and with your directions you showed me what really means doing research and how much you care about it. I truly wish you a brilliant academic career: you deserve it! I will be, moreover, thankful to all the people that welcomed me in 4C: a great place where you can work towards excellent results as much as a place where you can have fun with the colleagues.

I want to thank any person that dedicated me a little piece of his time reading this thesis helping me to correct every mistake I made.

I want to thank every person with which I spent my time in University: it has been a hard path but you made it funnier and easier. Some of you pushed me studying more than what I would have done, some of you gave me the spirit of competition, some of you accepted me to join their groups for some projects even if we didn't know much each other before that and the spirit of collaboration that I found has been incomparable. I don't want to list the names here: you know who you are, and in a list I should choose for an order but in reality you are all equals to me.

On the other hand there is one person that I have to mention: thank you Giovanni Di Liberto! We spent an amazing year together in Cork: we learned together how to live on our own thousands of kilometers from home. Thanks for the chats: we have been able to build our futures and, hopefully, to pursue our dreams; I wish you all the best with your career in academia!

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the computational complexity theory it is supposed (under the assumption that $P \neq NP$) that for many types of problems there is not an algorithm that is able to solve all their instances in polynomial time. In this case the quest of an algorithm that is able to solve the biggest number of instances in the smallest time is an active field of research worldwide. For this kind of hard problems (NP-hard) algorithms able to solve the instances are known but, unfortunately, they may run on an instance for an exponential amount of time in order to find a solution. Problems that fall under the *NP-complete* class are, for example: boolean satisfiability (SAT), knapsack problem, hamiltonian path problem, travelling salesman. NP-complete problems, moreover, have the interesting feature that each one can be transformed into any other NP-complete problem in polynomial time.

In the same field of research are known the combinatorial problems: they consist of finding an optimal object from a finite set of objects and in many such cases exhaustive search is not feasible.

In the last 20 years for solving those kind of problems have been introduced techniques that are called *portfolio algorithms*, an idea that comes from economics. It is well known in two of the biggest communities (constraint programming (CP) and satisfiability (SAT)) that in most of the cases there is not a single solver that is always the best for all the instances. Instead solvers tend to vary greatly among different problems excelling on a particular set of instances while offering subpar performance on everything else.

In many real contexts research groups and companies have released different solvers for the purpose of solving the same problem but often instances of the same problem may exhibit different properties. For this reason solvers have complementing strengths and weaknesses and where one solver might exhibit bad performance, another will excell. It is no longer the case where one solver is able to achieve the best performance over all the others in all cases and *algorithm portfolios* exploit this situation trying to choose the most appropriate one specifically for the problem instances at hand.

This kind of framework relies heavily on how these portfolio techniques make the selection, a decision that can be made in many different ways. These different techniques are closely related to the field of *Algorithm Selection* [1] which is responsible for identifying the most suitable algorithm for solving a problem. One approach can be to train a regression model to predict the performance of each solver, selecting the expected best one [2, 3]. Alternatively, a ranking approach can be used over all solvers [4]. It is also possible to train a forest of trees, where each tree makes a decision for every pair of solvers, deciding which of the two is likely to be better, selecting the one voted upon most frequently [5]. Research has also been conducted on creating a schedule of solvers to call in sequence rather than committing to a single one [6, 7]. An overview of many of these approaches is presented in [8]. Yet regardless of the implementation, portfolio-based approaches have been dominating the competitions in satisfiability (SAT) [6, 2], constraint programming (CP) [7], and quantified Boolean formulae (QBF) [9] and still represent an active research field.

One of the most successful portfolio techniques is referred to as Instance-Specific Algorithm Configuration (ISAC) [10]. Originally the approach was demonstrated to outperform the then reigning regression-based approach, SATzilla, in the SAT domain [2], as well as a number of other existing portfolios [11]. The approach was later embraced by the 3S solver which won 2 gold medals in the 2011 SAT competition [12]. ISAC's applicability has also been demonstrated in other domains such as set covering (SCP) and mixed integer (MIP) problems [10].

This thesis starts from the willingness of further improving the standard ISAC methodology. The original idea has been to improve it by applying feature filtering and studying how different feature filtering approaches can affect the performance of the methodology. ISAC assumes that all features are equally important, but there is no reason why this assumption should be considered always true especially in settings with more than a hundred features. Moreover the hope of feature filtering is also to help the used clustering technique in order to achieve a sort of perfect grouping by removing noisy and disruptive features, which means finding a clustering of the space where instances in the same cluster prefer to be solved by the same solver.

Feature filtering has only been the starting point of the work and has proven to be beneficial to ISAC but yet the original purpose of perfect clustering has been far to be achieved. In particular clustering in most of the cases is used with the euclidean distance metric, but there is no reason to believe that this is always the best to use. For this reason different techniques aimed at learning a new distance metric have been studied. Even if this path didn't lead to an improvement in the performance an insight on the direction to be followed has been gained. In particular has been possible to analyse techniques that let the model take into account the past performance of the solvers on the instances of the training set.

Combining different predictive methods and taking into consideration the recorded performance of the solvers on the known instances, SNNAP has been introduced. Given a new instance SNNAP is able ot predict a subset of solvers that are going to perform consistently better than others. This approach redefines the feature vector to automatically encode the desired objective of having similar instances in the same cluster. However, unlike feature landmarking [13, 14] where the evaluation of multiple simple algorithms provides insight into the success of more complex algorithms, SNNAP tackles a domain where a problem needs to only be evaluated once, but highly efficiently. After this part, using a custom defined distance metric and switching to a nearest neighbors approach, SNNAP is able to group together instances that prefer to be solved by the same solvers with the result of a major improvement over the standard ISAC methodology.

The results presented by this thesis have also been published with a paper [15] in the proceedings of the ECML conference 2013 [16]. The paper has been presented in the 2013 edition of the conference which has been held in Prague in September 2013. Since the beginning of the work for the thesis, moreover, there has been the willingness to contribute to the community releasing the software hoping to promote further development and comparisons with existing and future portfolio methodologies. For this reason and with the precise scope to contribute with an open source software the code for SNNAP (plus a re-implementation of ISAC) has been released under the LGPL licence and is downloadable from [17].

1.1 Thesis Outline

This thesis presents a novel strategy that tackles the problem of algorithm portfolios applying machine learning techniques and improves a state-of-the-art portfolio algorithms approaches (ISAC).

Starting from the basic assumptions behind the standard ISAC methodology that the finite collection of features for each instance can be used to correctly identify its structure, the first step has been to try to understand if all the features are always essential applying different filtering techniques. Adding then the aim to improve clustering, grouping together instances that prefer to be solved by the same solver it has been created a methodology to redefine the feature vector including past performance of the solvers. The final result has been *SNNAP: Solver-based Nearest Neighbor for Algorithm Portfolios* which is here presented.

This dissertation starts in Chapter 2 where is presented an overview of the most important state-of-the-art methodologies in the field of algorithm portfolios. Chapter 3, then, presents an overview of common algorithms and definitions used throughout the thesis as well as a description of the used datasets and the problems they come from. The first trial to improve the standard ISAC methodology is presented in Chapter 4 applying different feature filtering techniques to clustering. Chapter 5 describes some attempts to refine the feature vector in order to include past performances of the solvers in the portfolio. Chapter 6 then is responsible of introducing the novel methodology called SNNAP validating its results on two different problems: *SAT* and *maxSAT*. At the end Chapter 7 presents how the released software has been implemented as well as a short guide to how to use it.

Chapter 2

Related Work

It is becoming increasingly recognized in the constraint programming (CP) and satisfiability (SAT) communities that there is no single best solver for all problems. Instead solvers tend to excel on a particular set of instances while offering subpar performance on everything else. This observation has led to the pursuit of approaches that, given a collection of existing solvers, attempt to select the most appropriate one for the problem instances at hand. The way in which these portfolio solvers make the selection, however, varies greatly. Being that the work presented here is focused on improving the way this choice is made it is worth describing what are current solutions to this problem.

2.1 The Algorithm Selection Problem

Many optimization problems can be solved using several algorithms, therefore with different performances. Considering the set of all possible instances of a problem, it has long been recognized that there is no single algorithm or system that will achieve the best performance in all cases. An important result has been proved by the famous No Free Launch Theorem (NFL) [18] that states that no single algorithm can be expected to perform optimally over all instances. This phenomenon is of great relevance among algorithms for solving NP-Hard problems, because of the high variability from instance to instance of a problem. Starting from the observation that different solvers have complementary strengths and weaknesses and so they show different behaviours on different problem instances, the ideas of running multiple solvers in parallel or to select one solver based on the features of a given instance were introduced. These approaches have been named *algorithm portfolios* [19, 20, 21, 7]. Portfolio research is an active research topic nowadays as shown in [22], with more than 150 papers and counting.

In the last years there have been many different softwares that implement the idea of algorithm portfolios. One of the biggest success stories is that of SATzilla [21], which combines existing Boolean Satisfiability (SAT) solvers and has, for about five years, dominated various categories of the SAT competition [12]. Another similar approach has been applied to the Constraint Programming area (CP) by CP-Hydra [7] where a portfolio of CP solvers won the CSP 2008 Competition.

In the context of algorithm portfolios the ideal solution would be to be able to consult a hypothetical oracle that is able to predict perfectly which is going to be the best algorithm to solve a given problem instance, and then to use it to solve the instance.

The Algorithm Selection Problem, as first described by John R.Rice in 1976 [1] and as presented by [19], has three main aspects that must be tackled:

- The *selection of the set of features of the problem* that might be indicative of the performance of the algorithm;
- The *selection of the set of algorithms* (often referred as *solvers*) that together allow to solve the largest number of instances of the problem with the highest performance;
- The *selection of an efficient mapping mechanism* that permits to select the best algorithm to maximize the performance measure.

The features definition must be unique for all the instances of the same problem. Furthermore, it is of extreme importance that they highlight the differences between distinct instances.

The set of algorithms (often referred as portfolio) can be exploited using several techniques that can be grouped into *instance-oblivious* and *instance-specific* algorithm selection.

2.2 Instance-Oblivious Algorithm Selection

Given a representative set of problem instances (training set), instance-oblivious algorithm selection is the first approach that can be thought to be applied. It attempts to identify the solver or the combination of solvers resulting in the best average performance on all the training data. After the training phase, for each approach of this type the execution follows the same rules independently from the particular instance that is going to be solved.

A trivial solution is to measure the solving time on the training set, and then to use the algorithm that offers the best (eg. arithmetic mean, geometric mean, or median) performance. This approach can be called *Best Single Solver (BSS)* and will be one of the approaches that is going to be used as a comparison of the performance of the presented methodologies. This approach has the drawback of solving many instances with an algorithm that is not the best for them.

A more elaborated solution consists in trying to solve each new instance with a sequence of algorithms, each one with a particular time-limit. The training phase has the aim to identify these sequences of solvers and to assign them the execution time-limit. This approach is called *sequential portfolio.*

Finally, a third example is the *scheduling* technique. The execution follows similar rules to an operative system scheduler. Therefore, the training phase consists in finding an optimal sequence of solvers and a duration for each one of them, with the aim of maximizing the performances.

One of the main drawbacks of instance-oblivious algorithm selection is ignoring the specific instances, solving each new instance in the same way. As already claimed, and will later be demonstrated, there is no single algorithm or system that will achieve the best performance in all the instances of a certain problem [18]. Therefore, in order to obtain a technique that performs better than any of the algorithm in the portfolio, the idea is to select the solver that should perform better on the specific instance.

2.3 Instance-Specific Algorithm Selection

The main idea behind Instance-Specific Algorithm Selection, idealistically, uses an oracle that knows which is the best algorithm for the current instance. Several different techniques (mainly coming from machine learning approaches) have been developed to simulate this oracle, all based on the common assumption that instances prefer different solvers accordingly to the variations and similarities in their structures. Therefore, it is possible to construct a vector of features that aims to represent these structural differences and permits to realize a mapping mechanism between instances and best solving algorithm.

2.3.1 SATzilla

SATzilla [21, 23, 2] is an example of an algorithm portfolios approach applied specifically to the propositional satisfiability problem (SAT). Overall, since its initial introduction in 2007, SATzilla has won medals at the 2007 and 2009 editions [12].

The approach is based on a simple idea: given a new instance, forecast the log runtime of each solver using the ridge regression. A regression-based training technique assumes that the expected performance of a solver can be modeled by some function of the instance features. It is then possible to run the algorithm with the best predicted performance. Therefore, SATzilla uses a well-defined set of features specific for the SAT problem. The approach can be divided in two phases: training process and testing process.

In the first part the features are filtered using forward selection (that tries to select the most important features), then they are expanded using all the quadratic combinations of the reduced feature set. This operation is called quadratic basis function expansion and it is useful to add additional pairwise product features that could describe better a nonlinear function. Finally it is performed again the forward selection on the extended set of features. The training phase finds also the two algorithms that solve the largest number of instances, if each is given a 30 seconds timeout. The identified algorithms will be used as pre-solvers. These pre-solvers will be run for a short amount of time before features are computed, in order to ensure good performance on very easy instances. The importance of applying pre-solving lies in the possibility of decreasing the risk of making a mistake when solving easy instances. Finally a ridge regression model is trained on the training instances not solvable by the pre-solver to predict the log of the runtimes of the algorithms in the portfolio.

The testing process consists of executing the SATzilla strategy on the test istances. In particular, the pre-solvers are executed sequentially. When an instance is unsolved, its features are computed and then SATzilla predicts the expected runtime of each solver. Finally it runs the solver with lowest predicted runtime.

An issue of this approach is that it requires prior knowledge about the relationship between features and performance and that it relies heavily on the quality of the predictions. Furthermore the time predictions are not accurate, but in the case of the SAT competition they were accurate enough to distinguish between good and bad solvers.

2.3.2 CPHydra

CPHydra [7] is an algorithm portfolios approach for Constraint Satisfaction Problems (CSP). The idea is to attempt to schedule solvers to maximize the probability of solving an instance within the allotted time. Given a set of training instances and a set of available solvers, CPHydra collects informations on the performance of every solver on every instance. When a new instance needs to be solved, its features are computed and the k-nearest neighbors are selected from the training set. The problem then is set as a constraint program, that tries to find the sequence and duration in which invoking the solvers so as to yield the highest probability of solving the instance.

CPHydra makes use of *Case-Based Reasoning* (CBR) which is a machine learning methodology that adopts a lazy learning approach and contains no explicit model of the problem domain. In this context past examples are called *cases* and each case is made up of a description of a past example or experience and its respective solution. The full set of past experiences encapsulated in individual cases is called the *case base*.

The most fundamental element of any CBR system is the case base. In CPHydra, cases contain the feature values describing a particular CSP problem instance.

The CBR methodology can then be broken down into four distinct phases:

1. *Retrieval*: To begin a query case is produced. Using its features the case base is searched and the most similar cases to the present problem are retrieved (using a k-nn approach);

2. *Reuse*: CPHydra doesn't simply supply a prediction but also a schedule describing how long each solver should run. The previous phase of retrieval returns the set of solver times for each of the k most similar problem instances found in the case base. This information is then used to generate a solver schedule;

3. *Revision*: During this phase the proposed solution is evaluated. This process involves running each of the solvers found in the previous phase for the proportion of time allocated by the scheduler and determining if the problem is successfully solved by at least one of the solvers within its allocated time slot. If at least one solver solves the problem instance within its time slot then the schedule is deemed a success;

4. *Retention*: Normally once a satisfactory solution has been determined the problem description and solution are added to the case base. However in CPHydra the *solution* attached to each case and the solver times are only indirectly used to produce a solution. In order to create a complete case that could be retained each solver would have to run until it solved the problem instance or timed-out.

The effectiveness of the approach was demonstrated when CPHydra won the CSP Solver Competition in 2008, but also showed the difficulties of the approach since the dynamic scheduling program only used three solvers and a neighborhood of 10 instances.

2.3.3 ISAC

ISAC [10], Instance-Specific Algorithm Configuration, combines a configuration method and unsupervised learning obtaining a high performance algorithm selection method. ISAC can apply a *pure solver portfolio* approach in which the training instances are clustered accordingly to their normalized features. For each cluster, then, ISAC determines the overall best algorithm. At runtime, instead, it determines the closest cluster and tackles the input with the corresponding solver.

Algorithm 1 Istance-Specific Algorithm Configuration

```
1: function ISAC-Train($T, F, A$)
2:     $(\bar{F}) \leftarrow Normalize(F)$
3:     $(k, C, S) \leftarrow Cluster(T, \bar{F})$
4:     for all $i = 1, \ldots, k$ do
5:         $BS_i \leftarrow FindBestSolver(T, S_i, A)$
6:     end for
7:     return $(k, C, BS)$
8: end function

1: function ISAC-Run($x, k, C, BS$)
2:     $j \leftarrow FindClosestCluster(k, x, C)$
3:     return $BS_j(x)$
4: end function
```

Although ISAC can be used also as *parameter configurator* in this thesis when mentioning ISAC only the *pure solver portfolio* capabilities will be considered. For this reason it is described in more details here.

The fundamental principle behind ISAC is that instances with similar features are likely to have commonalities in their structure, and that there exists at least one solver that is best at solving that particular structure. Therefore the approach works as presented in Algorithm 1. In the training phase, ISAC is provided with a list of training instances T, their corresponding feature vectors F, and a collection of solvers A.

First, the computed features are normalized so that every feature ranges in [-1,1]. This normalization helps keep all the features at the same order of magnitude, and thereby avoids the larger values being given more weight than smaller values. Using these normalized values, the instances are clustered. The ultimate goal of clustering is to bring instances together that prefer to be solved by the same solver. Although any clustering approach can be used, in practice *g-means* (which will be described in details in Section 3.4.2) is employed in order to avoid specifying the desired number of clusters. Once the instances are clustered, an additional step has been added to merge all clusters smaller than a predefined threshold into their neighboring clusters. The final result of the clustering is a set of k clusters S, and a set of cluster centers C. To each cluster then is assigned a single solver, which is usually the one that has best average performance on all instances in that cluster. When the procedure is presented with a previously unseen instance x, ISAC assigns it to the nearest cluster and runs the solver designated to that cluster on the instance x.

In practice, this standard version of ISAC has been continuously shown to perform well, commonly outperforming the choice of a single solver on all instances. In many situations ISAC has even outperformed existing state-of-the-art regression-based portfolios [11]. However, the current version of ISAC also accepts the computed clustering on faith, even though it might not be optimal or might not best capture the relationship between problem instances and solvers. It also does not take into account that some of the features might be noisy or misleading. One possible solution to this problem will be evaluated in Chapter 4.

2.3.4 3S

SAT Solver Selector (3S) [6, 24] is an approach developed after the ISAC methodology that has dominated the sequential portfolio solvers at the SAT Competition 2011 where it won gold medals in the CPU-time category on random and crafted instances. 3S combines solver scheduling and clustering. Instead of using a static clustering approach it uses a dynamic clustering technique: the k-nearest neighbors (k-NN). In essence the decision for a new instance is based on prior exprience in the k most similar cases. It means that the first thing to be done is to identify which k training instances are the most similar to the one given at runtime, and then choose the solver that worked the best on these k training instances. Moreover 3S try to learn a schedule of solvers for a pre-solver that maximizes the number of instances solved quickly. To compute the static schedule offline and to select the long-running solver online, 3S combines lowbias nearest neighbor regression with integer programming optimization.

After having applied the k-NN approach 3S computes a schedule that defines the sequence of solvers, along with individual time limits, given a new test instance.

A main issues of 3S is that the solvers working in the scheduler pre-solver (10% of the solving timeout) work independently, without passing information among them. Another problem regards the choice of the long run solver, that can't be corrected if sub-optimal during the solving process. Similar problems are common also to other approaches like ISAC and have been addressed in [25].

2.3.5 SATzilla 2012

The original SATzilla approach has been modified in order to partecipate to the SAT competition of 2012. In practice they propose [26] to train trees to compare each pair of solvers. At the end the approach selects the solver accordingly to a majority vote.

In the training phase SatZilla 2012 finds the best pre-solver. Than a cost sensitive classification model (decision forest) is trained for every pair of solvers in the portfolio, predicting which solver is going to behave better on a given instance. In the following testing phase the pre-solver is let running on the given instance. If it is not successful the features of the instance are computed and then from the classification models previously built for every pair of solvers the expected winners are compared and, at the end, the solver that is expected to be winning more frequently is selected for solving the instance.

Being that a decision forest is trained for every pair of solvers in the portfolio the main issue behind Satizlla 2012 is that it doesn't scale well when the number of solvers increases.

2.3.6 Parallelization

In the last years it has become a standard to have, even for personal uses, machines with multiple cores (4, 8, or more). It has so started the idea to apply algorithm selection in parallel environments.

One idea might be to execute all the solvers in parallel, one for every core of the used machine. The problem now becomes to select p solvers that, as a set, will solve the biggest number of instances. Possible different approaches that try to augment the basic 3S algorithm can be found in [25]. To the problem of scheduling there is the added constraint of a time limit for each processor. The objective of parallelization is still to minimize the number of uncovered instances but now there is the additional constraint of minimizing the total CPU time of the schedule. Moreover an issue might come: the solvers used in the portfolios might be parallel algorithms themselves. Adding to this the fact that some solvers might have different parameterization the problem now is to schedule efficiently a potential infinite set of different solvers.

Starting from the idea behind 3S it is useful to note that the problem of solving the biggest number of instances can no longer be solved by simply choosing the one solver that solves most instances in time; it is needed, now, to decide how to integrate the newly chosen solvers with the ones from the static schedule.

The problem of finding a *parallel solver scheduling* is then set as an Integer Programming (IP) problem that can have more than 1.5 million variables which can't be solved exactly during the execution phase. For this reason the problem is solved heuristically by not considering some variables.

A major issue behind parallelization is that identical processors to which each solver can be assigned introduce symmetries that can hinder optimization since equivalent schedules will be considered multiple times by a sistematic solver.

Another issue is that is not clear how different solvers running at the same time should share information among them, and which type of information is worth being shared.

The application of parallelization to the problem of algorithm selection is interesting and is an active field of research but it goes beyond the scope of this thesis, while it may be a future development for it.

2.4 Summary

This chapter describes the actual research scenario around the problem of algorithm portfolio. It has been described how the problem can be tackled in two different ways: *instance-obliviously* and *instance-specifically.* In particular have been described in greater details the approaches of the second area as the approach here presented (SNNAP) specifically starts from the willingness of analysing and improving ISAC, which belongs to the category of *instance-specific.*

The area of *algorithm selection* is an active field and in this chapter have been presented the most advanced techniques that have been shown empirically to provide significant improvements in the results. Each approach, however, also has a few drawbacks. Instance-oblivious, for example, assumes that all problem instances can be solved optimally by the same solver, a result clearly in constrast with the No Free Lunch (NFL) theorem. Instance-specific, on the other hand, has the drawback of heavily rely only on the set of features to determine the best solver to use for each instance.

Chapter 3

Description of the Scenario

The purpose of this thesis was to validate the performance of SNNAP using the satisfiability (SAT) domain. SAT is one of the most fundamental problems in computer science. This *NP* problem is interesting both for its own sake and because other *NP* problems can be encoded into SAT and solved by the same solvers. Since it is conceptually simple, significant research efforts have been put in developing sophisticated algorithms with highly-optimized implementations. Furthermore there is a competition, the SAT competition [12], that incentives researchers to tackle this problem offering visibility to the best developed solvers. In chapter 6, moreover, will be introduced benchmarks of SNNAP also on a slightly different problem: the Maximum Satisfiability Problem (MaxSAT).

This chapter has been used for describing the scenario in which SNNAP operates: the datasets used will be described as well as the problems from which they come from. Finally, SNNAP makes use of a number of predictive models, the most prevalent of which will be defined.

3.1 The Satisfiability Problem

The satisfiability problem (SAT) has been the first known problem proved to belong to the NP-complete class. That means that there is no known algorithm that is able to efficiently solve all instances of SAT, and it is generally believed (under the famous assumption that $P \neq NP$) that no such algorithm exists.

Instances of SAT are boolean expression written using only *AND*, *OR*, *NOT* operators, literals and parentheses. A *literal* is either a *variable* or the *negation of a variable.* For example x_1 is a positive literal while $\bar{x_1}$ is a negative literal. A *clause*, instead, is the disjunction (OR) of literals (e.g. $\bar{x_1} \vee x_2 \vee \bar{x_3}$) and the conjunction (AND) of clauses is called *expression.*

The question now is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true? A formula of propositional logic is said to be satisfiable if there is an assignment of logical values that makes the formula true. The propositional satisfiability problem (PSAT), which decides whether a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmics, electronic design automation and verification.

The simplest variant of this problem that is NP-complete is represented by the *3CNFSAT*: each clause has exactly three literals and the overall expression is the conjunction (AND) of an arbitrary number of clauses. It can, on the other hand, be proved that if each clause has only 2 literals then the problem is no more NP-complete.

3.1.1 Description of the Dataset

The satisfiability (SAT) domain was selected to test the proposed methodologies due to the large number of diverse instances and solvers that are available. Four datasets are used in this thesis, each of which has been taken from SAT competitions from 2006 to 2012 inclusive [12]. The instances have been divided in four datasets containing 2140, 735, 1098 and 4243 instances divided, respectively, as follows:

RAND: instances have been generated at random;

HAND: instances are hand-crafted or are transformations from other NP-Complete problems;

INDU: instances come from industrial problems;

ALL: instances are the union of the previous datasets.

It has been chosen to rely on the standard set of 115 features that have been embraced by the SAT community [2]. Specifically, for the computatin of the features it has been used the feature code made available by UBC[1]. However, being that has been observed that the features measuring computation time are unreliable, those 9 features have been discarded from the dataset. The employed features are here listed:

Problem Size Features:
1-2 Number of variables and clauses in original formula: denoted v and c, respectively
3-4 Number of variables and clauses after simplification with SATElite: denoted v' and c', respectively
5-6 Reduction of variables and clauses by simplification: (v-v')/v' and (c-c')/c'
7 Ratio of variables to clauses: v'/c'
Variable-Clause Graph Features:
8-12 Variable node degree

[1]http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/

statistics: mean, variation coefficient, min, max and entropy

13-17 Clause node degree statistics: mean, variation coefficient, min, max and entropy

Variable Graph Features:

18-21 Node degree statistics: mean, variation coefficient, min and max

22-26 Clustering Coefficient: mean, variation coefficient, min, max and entropy

Clause Graph Features:

27-31 Node degree statistics: mean, variation coefficient, min, max and entropy

32-36 Clustering Coefficient: mean, variation coefficient, min, max and entropy

Balance Features:

37-41 Ratio of positive to negative literals in each clause: mean, variation coefficient, min, max and entropy

42-46 Ratio of positive to negative occurrences of each variable: mean, variation coefficient, min, max and entropy

47-49 Fraction of unary, binary and ternary clauses

Proximity to Horn Formula:

50 Fraction of Horn clauses

51-55 Number of occurrences in a Horn clause for each variable: mean, variation coefficient, min, max and entropy

Local Search Probing Features, based on 2 seconds of running each of SAPS and GSAT:

56-65 Number of steps to the best local minimum in a run: mean, median, variation coefficient, 10th and 90th percentiles

66-69 Average improvement to best in a run: mean and coefficient of variation of improvement per step to best solution

70-73 Fraction of improvement due to first local minimum: mean and variation coefficient

74-77 Coefficient of variation of the number of unsatisfied clauses in each local minimum: mean and variation coefficient

Clause Learning Features (based on 2 seconds of running Zchaff_rand):

78-86 Number of learned clauses: mean, variation coefficient, min, max, 10%, 25%, 50%, 75% and 90% quantiles

87-95 Length of learned clauses: mean, variation coefficient, min, max, 10%, 25%, 50%, 75% and 90% quantiles

Survey Propagation Features:

96-104 Confidence of survey propagation: For each variable, compute the higher of $P(true)/P(false)$ or $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75% and 90% quantiles

105-113 Unconstrained variables: For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75% and 90% quantiles

Lobjois Branch and Bound Algorithm Selection:

114-115 Branch and Bound Performance Prediction: Mean depth over variables and log number of nodes over variables

Employed features vary greatly in the meaning and they cover aspects as: *local search probing*, based on 2 seconds of running each of SAPS and GSAT (two stochastic local search algorithms for solving SAT); then is taken into consideration the calculation of statistics regarding clause learning gathered in 2-second runs of *Zchaff_rand* [27] where is measured the number of learned clauses. Furthermore are considered features covering information such as the number of variables, the number of clauses, the average number of literals per clause, the proportion of positive to negative literals per clause, the proximity of the instance to a Horn formula[2], the number of clauses with one, two or three variables. A SAT instance can be converted to three different graph representations: the Variable-Clause Graph (VCG), the Variable Graph (VG) and Clause Graph (CG). For example the Variable-Clause Graph is a bipartite graph with a node for each variable, a node for each clause, and an edge between them whenever a variable occurs in a clause. For each of the graph representations are recorded various node degree statistics (for example some features are related to the diameter of the graph associated to the expression where for each node in the variable graph it is possible to compute the longest and shortest path between it and any other node and report the statistics across all nodes). Finally, the runtimes of 29 of the most current SAT solvers have been recorded, many of which have individually shown very good performance in past competitions. Specifically, are used:

- `clasp-2.1.1_jumpy`
- `clasp-2.1.1_trendy`
- `ebminisat`
- `glueminisat`
- `lingeling`
- `lrglshr`
- `picosat`
- `restartsat`
- `circminisat`
- `clasp1`
- `cryptominisat_2011`
- `eagleup`
- `gnoveltyp2`
- `march_rw`
- `mphaseSAT`
- `mphaseSATm`
- `precosat`
- `qutersat`
- `sapperlot`
- `sat4j-2.3.2`
- `sattimep`
- `sparrow`
- `tnm`
- `cryptominisat295`
- `minisatPSM`
- `sattime2011`
- `ccasat`
- `glucose_21`
- `glucose_21_modified`.

Each of the solvers was run on every instance with a 5,000 second timeout. After that instances that could not be solved by any of the solvers within the allotted time limit have been removed. Have further been removed instances that were deemed to be too easy, i.e. those where the best 10 solvers could solve the instance within 15 seconds. This resulted in the final datasets comprising of 1949 Random, 363 Crafted, and 805 Industrial instances, i.e. 3117 instances in total.

Throughout this thesis the results will be evaluated comparing them to three benchmark values:

- *Virtual Best Solver (VBS)*: This is the lower bound of what is achievable with a perfect portfolio that, for every instance, always chooses the solver that results in the best performance;

[2]Where clauses have at most one positive literal.

- *Best Single Solver (BSS)*: This is the desired upper bound, obtained by solving each instance with the solver whose average running time is the lowest on the entire dataset;
- *Instance-Specific Algorithm Configuration (ISAC)*: This is the pure ISAC methodology obtained with the normal set of features and *gmeans* clustering.

The results will always lie between the *VBS* and the *BSS* with the ultimate goal to improve over *ISAC*. Results will be divided according to their dataset.

3.2 The Maximum Satisfiability Problem

The *Maximum Satisfiability Problem* (maxSAT) is similar to the standard SAT problem: it starts from a boolean expression and it tries to determine what is the maximum number of satisfiable clauses. It is considered the optimization problem of SAT. There is not the requirement that all the clauses in the expression are satisfied all at the same time: it aims to find the biggest number of clauses satisfied under the same variables assignment. MaxSAT is a NP-hard problem, so it is not known an efficient algorithm to solve it.

As a result of its NP-hardness large size maxSAT instances (as well as large instances of other hard problems) might not be solved in a reasonable amount of time, and typically solvers make use of approximation algorithms and heuristics.

A maxSAT instance could be identified as belonging to one of four distinct categories:

- *Standard maxSAT* (is the problem of determining the maximum number of clauses, of a given Boolean formula, that can be satisfied by some assignment);
- *Weighted maxSAT* (given a set of weighted clauses returns the maximum weight which can be satisfied by any assignment);
- *Partial maxSAT* (returns the maximum number of clauses which can be satisfied by any assignment of a given subset of clauses while the rest of the clauses must be satisfied);
- *Weighted partial maxSAT* (asks for the maximum weight which can be satisfied by any assignment, given a subset of weighted clauses. The rest of the clauses must be satisfied).

Formally a weighted clause is a pair (C, w), where C is a clause and w is a natural number or infinity, indicating the penalty for falsifying the clause

C. A weighted partial MaxSAT formula is a multiset of weighted clauses $\varphi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$ where the first m clauses are soft and the last *m'* are hard. The set of variables occurring in a formula φ is noted as $var(\varphi)$.

A *(total) truth assignment* for a formula φ is a function $I : var(\varphi) \rightarrow \{0, 1\}$ that can be extended to literals, clauses, SAT formulas. For MaxSAT formulas are defined as $I(\{(C_1, w_1), \ldots, (C_m, w_m)\}) = \sum_{i=1}^{m} w_i(1 - I(C_i))$. The *optimal cost* of a formula is $cost(\varphi) = min\{I(\varphi) \mid I : var(\varphi) \rightarrow \{0, 1\}\}$ and an *optimal assignment* is an assignment I such that $I(\varphi) = cost(\varphi)$. In the *partial maxSAT* problem an instance is composed by hard and soft clauses (to the soft clauses is assigned also a weight) and a variable assignment is asked to satisfy all hard clauses while maximising the sum of weights of the satisied soft clauses.

3.2.1 Description of the dataset

The dataset that will be used in this thesis will deal with *partial MaxSAT problems*. Every *weighted partial MaxSAT* instance has been translated into an *equivalent weighted partial MaxSAT* instance where all the soft clauses are unit. At this point a subset of the SAT features of the resulting formula are used. In particular are considered features describing statistics in the *Variable-Clause Graph*, the ratio between positive and negative literals, the number of clauses with one, two or three literals and the proximity to a Horn formula (including the fraction of Horn clauses and the number of occurences in a Horn clause for each variable). To this set, moreover, have been added as additional features the number of soft clauses for partial MaxSAT instances, and the distribution of the weights for weighted partial MaxSAT instances. The total number of used features is 37 and are here listed:

Problem Size Features:
- 1-2 Number of variables and clauses
- 3 Ratio of soft variables to clauses
- 4-7 Statistics about soft variables: mean, standard deviation, min, max
- 8 Ratio of variables to clauses

Variable-Clause Graph Features:
- 9-13 Variable node degree statistics: mean, standard deviation, min, max and spread
- 14-18 Clause node degree statistics: mean, standard deviation, min, max and spread

Balance Features:
- 19-23 Ratio of positive to negative occurrences of each variable: mean, standard deviation, min, max and spread
- 24-28 Ratio of positive to negative literals in each clause: mean, standard deviation, min, max and spread
- 29-31 Fraction of unary, binary and ternary clauses

Proximity to Horn Formula:
- 32-36 Number of occurrences in a Horn clause for each variable: mean, standard deviation, min, max and spread
- 37 Fraction of Horn clauses

Using a set of solvers that have participated in the last MaxSAT competitions [28], the dataset has been compiled registering the running times of 14 of them:

- `QMaxSat-g2`
- `pwbo2.0`
- `QMaxSat`
- `PM2`
- `ShinMaxSat`
- `Sat4j`
- `WPM1`
- `wbo1.6`
- `WMaxSatz+`
- `WMaxSatz09`
- `akmaxsat`
- `akmaxsat_ls`
- `iut_rr_rv`
- `iut_rr_ls`

Furthermore what is especially engaging about the MaxSAT evaluations, is that there is no single solver that dominates on every category. Instead, there are a number of competing techniques each of which excels on a different benchmark. This is the ideal scenario for constructing a portfolio approach.

3.3 Predictive Models

The main question addressed in this study is: given an instance of a NP-complete problem (SAT for example), how is possible to select the algorithm to use (from the available ones in the portfolio) in order to solve the instance and solve it in the smallest amount of time? It is both interesting solving the biggest number of instances possible, both to solve them in the smaller time possible.

One way in which this problem can be tackled (extensive description of possible solutions has been given in Chapter 2) is using predictive models. A predictive model (a classifier) is the process by which a model is crafted to be able to best predict a given outcome. In order to build such a system the features of the problem are analysed and used. Usually the interesting part is identifying to which of a set of categories a new observation (instance) belongs (the category with which each instance is going to be classified, in this case, is the name of the best solver to solve it). For achieving this result the model is first presented with a training set of data containing observations whose category membership is known. The testing set, instead, is a set of instances whose category membership is to be guessed using the observations on the training set and the model previously built.

It is common practice in the field of machine learning to consider classification as *supervised learning* (learning where a training set of correctly-identified observations is available) or *unsupervised learning*, like clustering, where the examples given to the classifier are unlabeled and the procedure tries to find some sort of hidden structure in the data.

When a new predictive model is created, the main interest relies in having an idea of how well the model will behave on real data and is not trivial to understand how good a model is or why a given model should be preferred over another one. For this reason it is useful to evaluate in some way its performance in order to try to understand how the model will perform on previously unseen datas: how many predictions will be correct? Consequently a random subset of the training set is taken and its instances are used pretending that their category memberships are unknown. Then the real categories are compared to the predicted ones. In order to avoid overfitting it is never suggested to predict the category of an instance that has been used also to train the model: the point is that the model should be able to predict well previously unseen instances. For this reason before the training phase the available dataset is split into two subsets: training set and testing set. However this approach has two main issues:

- In order to train a good model, it is sought to have as many training instances as possible;
- In order to get a good idea of how well the model behaves in practice, it is sought to have a large test set.

Obviously the two requirements contrast each other, so one common solution is to use a Cross-Fold Validation approach: the dataset is split in n parts (usually $n = 10$); the model is then trained using only n - *1* folds, and the remaining fold is used as testing set. The process is repeated n times, each time choosing a different fold as testing set. The n results, then, from the different folds can be averaged to produce a single estimation. The advantage of this method is that all observations are used for both training and testing, and each instance is used for testing exactly once.

In the case of the work done for this thesis, then, each experiment has been repeated 10 times to decrease the bias of the estimates. In the experiment moreover are presented both the average and the PAR-10 performance[3].

3.4 Clustering

Clustering, that is used by ISAC, is the typical example of *unsupervised learning*: while *supervised learning* methods are intended for classification problems, clustering is used for descriptive problems. *Clustering* is the task of grouping set of instances in subsets (clusters) in a way that instances in the same subset present some kind of similarity.

Clustering could be achieved using different techniques, each one with different results. Each approach differs from the others in the notion of

[3]PAR10 score is a penalized average of the runtimes: for each instance that is solved within 5000 seconds (the timeout threshold), the actual runtime in seconds denotes the penalty for that instance. For each instance that is not solved within the time limit, the penalty is set to 50000, which is 10 times the original timeout.

what is a cluster and in the distance metric used. Clustering groups data instances into smaller groups in such a way that similar instances are grouped together, while different instances belong to different groups. The instances are thereby organized into an efficient representation that characterizes the population being sampled. Formally, the clustering structure is represented as a set of subsets $C = C_1, \ldots, C_k$ of S, such that: $S = \cup_{i=1}^{k} C_i$ and $C_i \cap C_j = \emptyset, \forall i \neq j$. Consequently, any instance in S belongs to exactly one and only one subset [29].

Since clustering is the task of grouping similar instances, some sort of measure that can determine whether two objects are similar or dissimilar is required. Commonly the euclidean distance between instances is used, evaluated in the feature space; in addition to this, being that different features might have a different range a pre processing is often required, in order for all the features to have the same importance in determining the distance between instances.

Evaluating the quality of clustering is not a simple task and as such a number of quality indices have been developed. For example a criteria might be the one of measuring the compactness of the clusters looking at the intra-cluster homogeneity, the inter-cluster separability or a combination of these two. Indices of this kind are, for example, *Sum of Squared Error (SSE), Minimum Variance Criteria and Scatter Criteria* [29].

From the many clustering techniques available the developed software gives to the user, when using the ISAC methodology, the chance to choose among:

- X-means (a variant of k-means);
- gmeans;
- hierarchical clustering.

3.4.1 Kmeans

One of the most used clustering method is Lloyd's k-means [30]. The idea behind the proposed methodology (presented in Algorithm 2) is to start from k random points in the feature space. These points represent the centers of the k clusters. The algorithm works alternating two phases: the first step assigns each instance to a cluster according to the shortest distance to one of the k points that were chosen. The second step, instead, updates the k centers, calculating the new ones as the mean of all the instances belonging to that cluster.

k-means is one of the most used clustering techniques for its simplicity and easiness of implementation. Unfortunately the main drawback is that the user has to pre-specify in advance the number of clusters that he is interested to have.

Algorithm 2 k-Means Clustering Algorithm

```
1: function k-MEANS($(X, k)$)
2:     Choose $k$ random points $C_1, \dots, C_K$ from $X$
3:     while Not done do
4:         for all $i = 1, \dots, k$ do
5:             $S_i \leftarrow \{j : \|X_j - C_i\| \leq \|X_j - C_l\| \ \forall l = 1, \dots, k\}$
6:             $C_i \leftarrow \frac{1}{|S_i|} \sum_{j \in S_i} X_j$
7:         end for
8:     end while
9:     return $(C, S)$
10: end function
```

XMeans

X-means has been introduced in [31] with the main purpose of using k-means but without having to specify in advance the number of clusters. This algorithm is the one used in the developed software.

It runs k-means on the input data with an increasing k (up to a chosen upper bound), and then chooses the best value of k based on a Bayesian Information Criterion (BIC). The technique searches the space of cluster locations and the number of clusters in an efficient way trying to optimize the measure given by the BIC. The BIC statistic used by X-means has been formulated to maximize the likelihood for spherically-distributed data. Thus, in general, it overestimates the number of true clusters in non-spherical data.

3.4.2 Gmeans

gmeans has been proposed in [32] as a clustering technique, extension of k-means, with the explicit purpose of automatically selecting the best value for k. The assumption behind the choice of k is that a good cluster is a cluster that exhibits a Gaussian distribution around the cluster center. The Algorithm (presented in Algorithm 3) has been personally implemented in R following the original publication and a Matlab code released by its author. In practice *gmeans* runs k-means with increasing k in a hierarchical fashion until the test accepts the hypothesis that the data assigned to each cluster are more gaussian than before.

The *gmeans* algorithm starts with a small number of k-means centers (1 at the beginning), and then grows the number of centers. At each iteration the algorithm splits into two those centers whose data appear not to come from a Gaussian distribution (this is checked applying the Anderson-Darling-Test [33]). Between each round of splitting, k-means is run on the entire dataset and all the centers to refine the current solution. *gmeans* repeatedly makes decisions based on a statistical test for the data assigned to each center. For this purpose *gmeans* splits the cluster in two by running 2-

Algorithm 3 g-means Clustering Algorithm

```
 1: function g-MEANS($X$)
 2:     $k \leftarrow 1$,$i \leftarrow 1$
 3:     $(C, S) \leftarrow kMeans(X, k)$
 4:     while $i \leq k$ do
 5:         $(\bar{C}, \bar{S}) \leftarrow kMeans(S_i, 2)$
 6:         $v \leftarrow \bar{C}_1 - \bar{C}_2$, $w \leftarrow \sum v_i^2$
 7:         $y_i \leftarrow \sum v_i x_i / w$
 8:         if Anderson-Darling-Test($y$) failed then
 9:             $C_i \leftarrow \bar{C}_1$, $S_i \leftarrow \bar{S}_1$
10:             $k \leftarrow k + 1$
11:             $C_k \leftarrow \bar{C}_2$, $S_k \leftarrow \bar{S}_2$
12:         else
13:             $i \leftarrow i + 1$
14:         end if
15:     end while
16:     return $(C, S, k)$
17: end function
```

means clustering (k-means with $k = 2$). All points in the cluster can be projected onto the line that runs through the centers of the two sub-clusters, obtaining a one dimensional distribution of points. *gmeans* now checks if this distribution is normal using the Anderson-Darling-Test. If the current cluster does not pass the test then is split into the two previously computed clusters, and the process is continued with the next cluster.

In Fig. 3.1 can be seen the differences of clustering by comparing x-means and *gmeans*. To project the data into a visualizable space (3D) PCA has been employed [34]. *gmeans* detects 8 different clusters while *xmeans* 4. Even though a correct answer doesn't exist, it seems that *gmeans* is able to separate better the datas and this is of particular importance in algorithm selection as clusters with too many instances might lead to non optimal choices (too close to *instance-oblivious*).

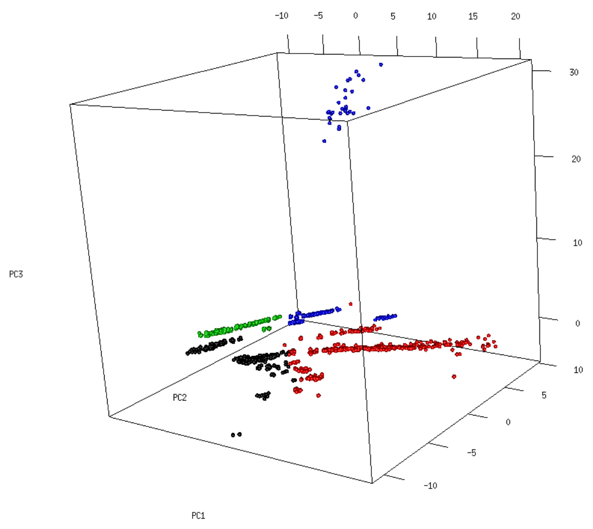
3.4.3 Hierarchical Clustering

The third type of clustering available to the user is *Hierarchical Clustering*. This method constructs the clusters by recursively partitioning the instances in either a top-down or bottom-up fashion [29]. Can be one of the following:

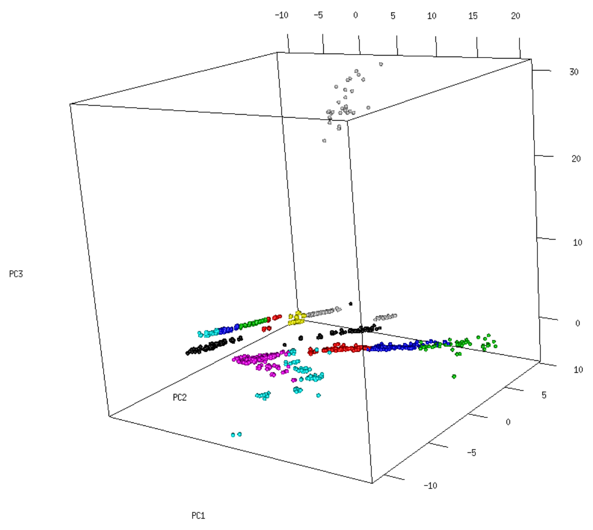
- *Agglomerative*: Each instance, at the beginning, represents its own cluster. Then clusters are merged until the desired cluster structure is reached;

Figure 3.1: Clustering of the RAND dataset accordingly to x-means and *gmeans*. Feature vectors have been projected into 3D using PCA.

(a) Clustering accordingly to x-means



(b) Clustering accordingly to *gmeans*



- *Divisive*: Each instance initially belongs to one cluster. Then the cluster is divided into sub-clusters, which are successively divided into their own sub-clusters. The process continues until the desired cluster structure is reached.

The result of the *hierarchical clustering* is a dendogram[4] representing the nested grouping of the instances. A clustering of the data is obtained by cutting the dendogram at a specified level.

In order to decide which clusters should be combined or split, a measure of dissimilarity between sets of instances is required. Usually this is achieved by use of an appropriate distance metric (generally Euclidean) function d, and a linkage criterion which specifies the dissimilarity of sets as a function of the pairwise distances of instances in the sets. Typical linkage functions between the set of instances in cluster A and cluster B are:

- *Single-link*: $min\{d(a, b) : a \in A, b \in B\}$
- *Complete-link*: $max\{d(a, b) : a \in A, b \in B\}$
- *Average-link*: $\frac{1}{|A||B|} \sum_{a \in A} \sum_{b \in B} d(a, b)$

One main drawback of hierarchical clustering is that it is computationally expensive because it has to calculate the distance between every pair of instance in the dataset.

3.4.4 *K*-nearest neighbors

The predictive model used by SNNAP is called K-nearest-neighbors (k-NN) and can be thought as a type of dynamic clustering. It is an algorithm for classifying instances basing the decision on the k closest training instances in the dataset. The k-NN algorithm is one of the simplest algorithm in all machine learning: an instance is classified by a majority vote of its neighbors, with the instance being assigned to the class most common amongst its k nearest neighbors.

The training instances are vectors in the feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples. In the classification phase, k is a user-defined parameter, and a previously unseen instance is categorized by assigning the label which is most frequent among the k training samples nearest to that point. A commonly used distance metric for continuous variables is Euclidean distance, even though in SNNAP a custom distance metric has been defined.

[4]A tree diagram where the leaves are the instances and the internal node are the clusters to which they belong.

3.5 Chapter Summary

This chapter has the aim to make the reader confident with the most important concepts that are used throughout the thesis. SNNAP is a tool that applies machine learning techniques in order to forecast the solver to be used in solving the presented instances; consequently the used techniques has been here introduced, with an exhaustive overview of clustering as it has been the starting point of the work. Each clustering technique has his own strengths and weaknesses: k-means is the most used one due to his simplicity and its fastness. Unfortunately to the users is asked to set an appropriate number of desired clusters. For addressing this issue two approaches have been proposed: X-means and *gmeans*, with the second shown to provide better performance than the first (this statement will also be proved in *algorithm selection* problems in Chapter 4). Hierarchical clustering, instead, has the main issue of being computationally expensive because it has to calculate the distance between every pair of instance in the dataset. On the other hand hierarchical clustering has the advantage that any valid measure of distance can be used in a straightforward way.

In addition to the description of what *predictive models* are, this chapter presents the settings to which SNNAP is applied: SAT and maxSAT problems. It is given an extensive explanation of what these problems are as well as a description of the employed datasets and their features.

Chapter 4

Feature Filtering

The first objective of this work has been to analyse and implement various feature filtering techniques applied to the original ISAC methodology. The current standard version of ISAC assumes that all features are equally important. But as was shown in [35], this is often not the case, and it is possible to achieve comparable performance with a fraction of the features, usually also resulting in slightly better overall performance. The original work presented in [35] only considered a rudimentary feed-forward selection. In this work, instead, more common and powerful filtering techniques are going to be used. An extensive description of feature filtering methods can be found in [36].

The main purpose of feature filtering is dimensionality reduction: the starting point is a complete dataset (e.g. the SAT dataset has 115 features) and the purpose is to find a way to filter out features that have little chance to be useful in the analysis of data. Generally these kind of filters are based on some kind of performance evaluation metric calculated directly from the data.

In this case the feature filter is a function returning a relevance index $J(S|D)$ that estimates, given the data D, how relevant a given feature subset S is for the task Y (usually classification). Relevance indices may be computed for individual features $X_i, i = 1, \dots, N$, providing indices that establish a ranking order $J(X_{i_1}) \leq J(X_{i_2}) \dots \leq J(X_{i_N})$. At the end the features that have the lowest ranks are filtered out. The point now remains to define what *relevant* means. A feature X is relevant in the process of distinguishing class $Y = y$ from others if and only if for some values $X = x$ for which $P(X = x) > 0$ the conditional probability $P(Y = y|X = x)$ is different from the unconditional probability $P(Y = y)$. In addition to this a good feature should not be redundant, i.e. it should not be correlated with other features already selected.

Now that is clear what means to filter features, and why it is useful, what needs to be specified is the classification problem that is going to be tackled: for this purpose are going to be kept the features that are good predictors of which solver is going to be the best on each instance.

The various feature filtering techniques used are going to return a value between 0 and 1 indicating how relevant the features are. With these values three things can be done:

- Select the k most promising features;
- Select the best x percent features;
- Select a subset of features which are significantly better than other.

4.1 Used techniques

In the developed software four different feature filtering techniques are provided and the FSelector R package [37] has been used:

- Chi Squared;
- Information Gain;
- Gain Ratio;
- Symmetrical Uncertainty.

4.1.1 Correlation-based Filters

Correlation coefficients are perhaps the simplest approach to feature relevance measurements. They can deal also with continuous features that, in this way, don't need a pre-discretization.

Contingency Tables defined for pairs of nominal features X, Y are frequently analysed to determine correlations between variables. They contain the numbers of times $M_{ij} = N(y_i, x_j)$ objects with feature values $Y = y_j, X = x_i$ appear in a database. m training samples may be divided into subsets of M_{ij} samples that belong to class $y_i, i = 1 \dots K$ and have a specific feature value x_j; summing over rows of the M_{ij} matrix marginal distribution $M_{i.}$ of samples over classes is obtained, while summing over columns distribution $M_{.j}$ of samples over distinct feature values x_j is obtained. The strength of association between variables X, Y is usually measured using χ^2 (*chi squared*) statistics:

$$\chi^2 = \sum_{ij} (M_{ij} - m_{ij})^2 / m_{ij} \qquad \text{where } m_{ij} = M_{i.} M_{.j} / m$$

Here m_{ij} represents the expected number of observations assuming X, Y independence.

Actually in the FSelector package is not used the usual Chi Squared index but the Cramer's V coefficient (ϕ_c), based on the chi squared index, where

$$\phi_c = \sqrt{\frac{\chi^2}{N(k-1)}}$$

Here N is the grand total of observations and k is the number of rows or number of columns depending on the smaller between them.

4.1.2 Measures Based on Information Theory

Information theory indices are most frequently used for feature evaluation and in particular are based on the concept of entropy. Information, which is the negative of entropy, is defined as

$$H(Y) = -\sum_{i=1}^{K} P(y_i) \log_2 P(y_i)$$

where $P(y_i) = m_i/m$ is the fraction of samples x from class $y_i, i = 1 \dots K$. Information contained in the joint distribution of classes and features, summed over all classes, gives an estimation of the importance of the feature, where the information contained in the joint distribution is defined as:

$$H(Y,X) = -\sum_{i} \sum_{j=1}^{K} P(y_j, x_i) \log_2 P(y_j, x_i)$$

In case of continuous features, for information theory based indices, it is necessary to discretize them. At this point are easily defined the remaining feature filtering functions used in the developed software:

- *information gain:* $H(Y) + H(X) - H(Y,X)$
- *gain ratio:* $\frac{H(Y)+H(X)-H(Y,X)}{H(X)}$
- *symmetrical uncertainty:* $2 \cdot \frac{H(Y)+H(X)-H(Y,X)}{H(Y)+H(X)}$

It is commonly believed that the *gain ratio* is a stable evaluation and the *symmetrical uncertainty* has a low bias for multivalued features.

4.2 Experimental Results

The first thing tried to understand the potentiality of feature filtering has been to test it applied to the original ISAC methodology (and thus to clustering). The software has been developed in R and in order to apply different feature filtering techniques has been used the *FSelector* package.

The results, presented in Table 4.1 have been organized based on the dataset used and based on the scoring function used. Here BSS is the approach of solving all the instances in the testing set with the solver that have the best performance on the whole training set; VBS, instead, is an oracle that solves each instance in the testing set with its own best solver. The columns, together with the percentage of not solved instances, report both the average running time, both the average PAR10 (a penalized average of the runtimes: for each instance that is solved within the timeout threshold, the actual runtime in seconds denotes the penalty for that instance, instead for each instance that is not solved within the time limit, the penalty is set to 10 times the original timeout). In parenthesis are reported the standard deviations. The averages and the standard deviations are calculated from executing the experiments 10 times.

Restricting the filtering approaches to find the best 15 features, it can be seen in Table 4.1 that the results are highly dependent on the dataset taken into consideration.

For the random dataset there is no major improvement due to using just a subset of the features. Yet it is possible to achieve almost the same result as the original ISAC by just using a subset of the features calculated using the *gain.ratio* function, a sign that not all the features are needed. It can then be observed that the improvements are more pronounced in the hand-crafted and industrial datasets. For them, the functions that give the best results are, respectively, *chi.squared* and *gain.ratio*, but in the latter the result is almost identical to the one given by *chi.squared*.

Instead of fixing the desired number of features that should remain after filtering, it is possible (using an appropriate function in the package *FSelector*) to select a subset of features whose score is significantly better than the score of the others. The results are presented in Table 4.2 and the feature filtering functions used are the one that proved to be the best on each dataset. Using this approach the number of features left is dependent on the dataset: for the RAND dataset the number of left features is 5, for HAND 53, for INDU 33 and for ALL only 3. The results obtained with this different setting are slightly worst than the ones just outlined. Except for the HAND dataset the average running times are generally slower and is possible to claim that this approach selects not enough, or too many (depending on the dataset), features.

In general it doesn't seem that there is some kind of pattern helping to deduce which features are more important than others but relevant features are frequently (depending on the dataset) the one related with survey propagation (SP), Variable-Clause graph (VCG) and Clause Learning (CL).

VCG features are a group of features responsible for describing a SAT instance as a graph. The graph is bipartite with a node for each variable, a node for each clause, and an edge between them whenever a variable occurs

Table 4.1: Results on the SAT benchmark, comparing the Virtual Best Solver (VBS), the Best Single Solver (BSS), the original ISAC approach (ISAC) and ISAC with different feature filtering techniques: "chi.squared", "information.gain", "symmetrical.uncertainty" and "gain.ratio".

RAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	1551 (0)	13154 (0)	25.28 (0)
ISAC	826.1 (6.6)	4584 (40.9)	8.1 (0.2)
chi.squared	1081 (42.23)	7318 (492.7)	14 (1)
information.gain	851.5 (32.33)	5161 (390)	8.7 (0.8)
symmetrical.uncertainty	840.2 (13.15)	4908 (189.5)	8.76 (0.4)
gain.ratio	830.3 (21.3)	4780 (210)	9 (0.4)
VBS	358 (0)	358 (0)	0 (0)
HAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	2080 (0)	15987 (0)	30.3 (0)
ISAC	1743 (34.4)	13994 (290.6)	26.5 (0.9)
chi.squared	1544 (37.8)	11771 (435)	23.5 (0.9)
information.gain	1641 (38.9)	12991 (443)	24.3 (0.9)
symmetrical.uncertainty	1686 (27.3)	13041 (336)	25.7 (0.7)
gain.ratio	1588 (43.7)	12092 (545)	22.4 (1)
VBS	400 (0)	400 (0)	0 (0)
INDU	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	871 (0)	4727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3166 (155.6)	5.2 (0.7)
chi.squared	708.1 (25.3)	3252 (218)	5.8 (0.4)
information.gain	712.6 (7.24)	2578 (120)	4.3 (0.3)
symmetrical.uncertainty	716.4 (16.76)	2737 (150)	4.4 (0.3)
gain.ratio	705.4 (19.9)	2697 (284)	4.1 (0.6)
VBS	319 (0)	319 (0)	0 (0)
ALL	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved*
BSS	2015 (0)	4726 (0)	30.9 (0)
ISAC	1015 (10.3)	6447 (92.4)	11.8 (0.2)
chi.squared	1078 (29.7)	7051 (414)	11.79 (0.8)
information.gain	1157 (18.9)	7950 (208)	15 (0.4)
symmetrical.uncertainty	1195 (28.7)	8067 (341)	15.6 (0.7)
gain.ratio	1111 (17.4)	6678 (225)	13.39 (0.5)
VBS	353 (0)	353 (0)	0 (0)

in a clause. Relevant features are related to statistics on the degree[1] of the variable and clause nodes, giving a clue that information as how many times a node is connected to others are important (it means how many times a variable appears in different clauses). In the random dataset, interesting,

[1]The degree of a vertex in a graph is the number of edges incident to the vertex.

Table 4.2: Results on the SAT benchmark, comparing the Virtual Best Solver (VBS), the Best Single Solver (BSS), the original ISAC approach (ISAC) and ISAC with feature filtering using the "Select the k most promising features" approach.

RAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	1551 (0)	13154 (0)	25.28 (0)
ISAC	826.1 (6.6)	4584 (40.9)	8.1 (0.2)
most promising features	1025 (8.3)	6646.8 (46)	12.5 (0.5)
VBS	358 (0)	358 (0)	0 (0)
HAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	2080 (0)	15987 (0)	30.3 (0)
ISAC	1743 (34.4)	13994 (290.6)	26.5 (0.9)
most promising features	1500 (32.7)	11521.6 (264)	22.8 (0.6)
VBS	400 (0)	400 (0)	0 (0)
INDU	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	871 (0)	4727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3166 (155.6)	5.2 (0.7)
most promising features	794 (5.1)	3248.3 (164.2)	5.4 (0.7)
VBS	319 (0)	319 (0)	0 (0)
ALL	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved*
BSS	2015 (0)	4726 (0)	30.9 (0)
ISAC	1015 (10.3)	6447 (92.4)	11.8 (0.2)
most promising features	1132 (13)	7125 (94.9)	13.3 (0.7)
VBS	353 (0)	353 (0)	0 (0)

are considered relevant features related to clustering in the Clause Graph (where the graph has nodes representing clauses and an edge between two clauses whenever they share a negated literal).

Survey propagation features come from an estimate of variable bias in a SAT formula bases on probabilistic inference [38]. In particular it is estimated the probability that each variable is required to be true or false, or is unconstrained. Those features appear to be important as they are crucial in determining the structure of an instance and its relatively hardness.

The last group of important features is the one related with *clause learning* and they are especially significant in the ALL dataset. Those features are based on 2 seconds of running *Zchaff_rand* [27]. What is measured is the number of learned clauses and the length of the learned clauses after every 1000 search steps, computing statistics over the resulting vectors. These features are highly dependent on the hardness of an instance, as the *survey propagation* features, and this might be a reason why they are selected as important during filtering.

On the other side it might be interesting to look if there are some group of features that are less likely to be selected in feature filtering. General features as the one related to the number of variables and clauses seem not to be important in the description of the instances. The same consideration applies to *balance features*: the one describing the ratio of positive to negative literals per clause, the ratio of positive to negative occurrences of each variable and the fraction of unary, binary and ternary clauses. Another group of features in general not selected with filtering are the one related to the proximity to the Horn formula.

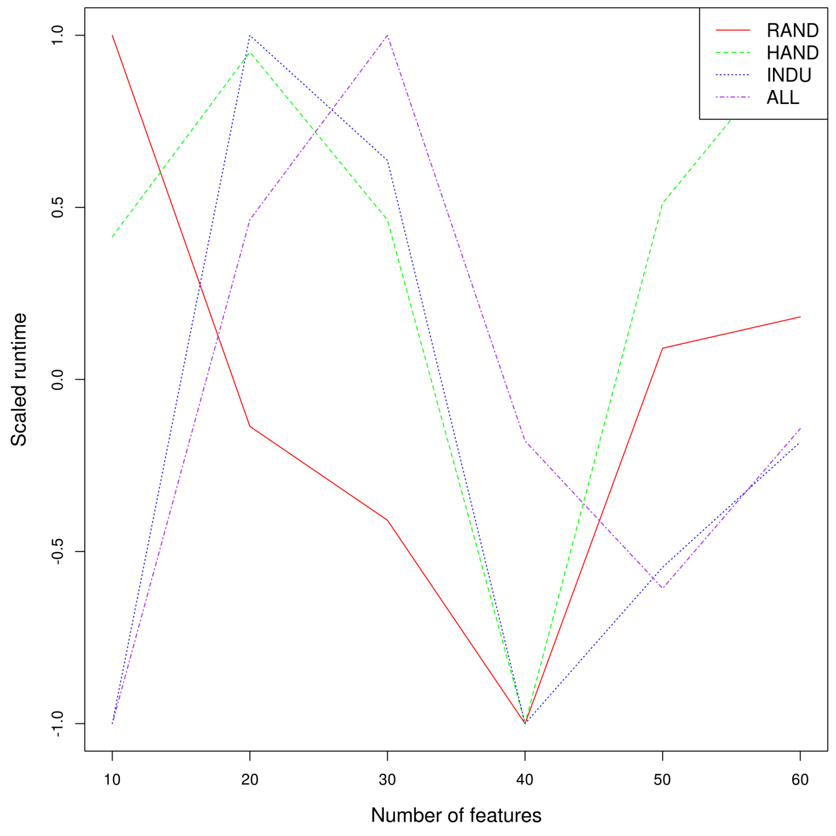
As can be seen in Fig. 4.1 it has been tried to understand how the number of features considered can affect the results of ISAC. Depending on which one of the four datasets considered the results are different but the performances seem to be better under the assumption of using only 10 or 40 features, especially for the HAND and INDU datasets which were the two datasets where feature filtering helped the most to improve. Moreover for all the datasets increasing the number of features over 50-60 causes the performances to decay. This behaviour was an expected behaviour as it proves that increasing the number of features considered, and then using all of them, is not necessary.

In Fig. 4.2 are reported the different runtimes on the four datasets accordingly to the reported feature filtering function and clustering technique used. For the first time is available a numerical comparison between different clustering techniques and *gmeans* results (except for the hand-crafted dataset) to be the best approach to use. Another interesting thing that can be easily noted is that, in general, what affects the most the performance is not the feature filtering function used but the clustering technique. Is useful to keep in mind that in the hand-crafted dataset *gmeans* doesn't provide the best results while in Chapter 6 it will be presented SNNAP that in this dataset, instead, achieves the biggest improvement among all the used datasets.

The advantages in using *gmeans* over the other clustering techniques are many. For example it is not needed to specify the number of desired clusters which is, in the majority of the cases, unknown in advance. Also *Xmeans* gives the same possibility but as seen in our experiment, and as presented in the original paper [32], *gmeans* often shows better results. *Hierarchical clustering*, on the other hand, needs to have specified the number of clusters, resulting in a less attractive choice.

It has to be noted, moreover, that even if in the HAND dataset *gmeans* didn't prove to have the best result, its performances are not much worst than the best performances obtained. In HAND the best result obtained by *gmeans* is only the 8% worst thant the best achieved (*XMeans*): even if it didn't prove to be the best, the gap to be closed is not that big. Among other three datasets INDU has been the one where *gmeans* obtained the smallest gain (6% better than the best obtained by *Xmeans*). On the other

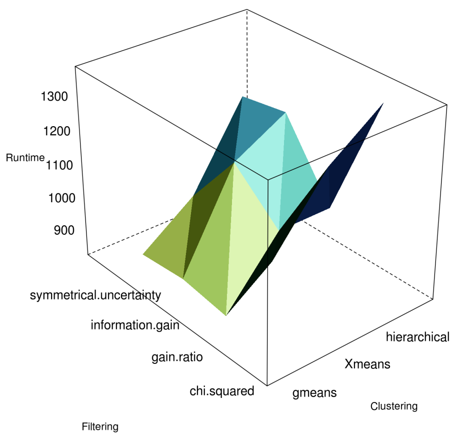
Figure 4.1: The variation of the performances considering different numbers of features.



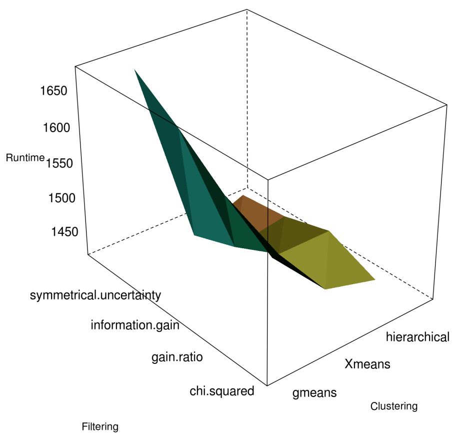
hand in RAND and ALL *gmeans* has been the best being able improve by a factor of 15% and 22%.

Figure 4.2: How clustering techniques and feature filtering functions affect performances.

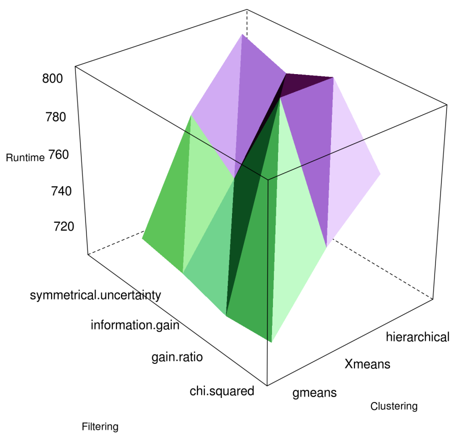
(a) RAND dataset



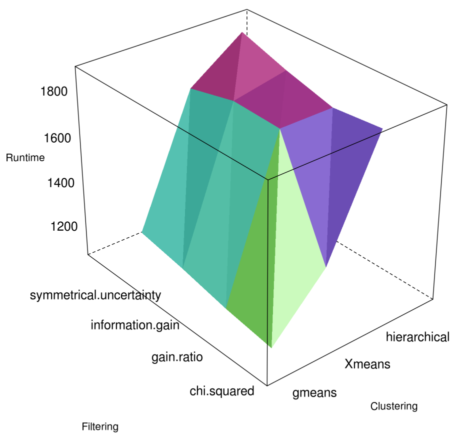
(b) HAND dataset



(c) INDU dataset



(d) ALL dataset



4.3 Chapter Summary

The results outlined in this chapter clearly show that not all features are necessary for clustering, and that it is possible to improve performance through the careful selection of the features or, at least, to obtain similar results. However, it can also be observed that the improvements can sometimes be minor, and are dependent on the filtering approach used and the dataset it is evaluated on. In the RAND and ALL datasets the results after feature filtering were comparable with the original approach: a positive sign being that using a smaller feature space can result in a quicker and easier analysis letting the user concentrate on other sides of the problem. The situation is even better with the other two datasets were the performances have been significantly improved.

These considerations are positive and promising as let the user concentrate on a smaller subset of the problem: dealing with only a few features is more manageable, easier and quicker than having to deal with more than a hundred.

Chapter 5

Refining the Feature Vector

As previously seen feature filtering gave promising results: using just a subset of the original features it has been possible to achieve the same (or slightly better) performances as using the full feature space. The next step has been to try to exploit better the feature space in order to tackle a wickness of how ISAC works: it has been noted that in the same cluster it is possible to find instances that are very similar in the feature space but very different when it comes to compare the performances of the solvers. The aim of clustering applied to this problem should be to be able to group together instances that show similar performances among the solvers. In this way the solver assigned to each cluster whould have optimal results with, idealistically, all the instances in the cluster. The case in which all the instances in the same cluster prefer to be solved by the same solver, and this is true for all the detected clusters, will be called *perfect clustering.*

The first thing tried has been to create a technique that would redefine the distance metric used (euclidean) by ISAC in order to include, in some way, the knowledge of the solver's performance in the calculation. Being that clustering works by constantly evaluating the distances among instances and grouping together the closest ones, the idea, described in this chapter, has been to learn a new distance metric hoping that, when it comes to calculate the distance between two instances, the ones that prefer to be solved by the same set of solvers are put near each other in the space.

5.1 Weighted Distance Metric

While the original version of ISAC employs Euclidean distance for clustering, there is no reason to believe that this is always the best distance function to use for the problem. As an alternative it is possible to think to learn a weighted distance metric, where the weights are tuned to match the desired similarity between two instances. For example, if two instances have the same best solver, then the distance between these two instances should be small. Alternatively, when a solver performs very well on one instance, but poorly on another, it might be desirable for these instances to be separated by a large distance.

One of the first thing that has been tried has been to use *linear regression*. Given a target numeric variable Y and a set of numeric input variables (features) X_i, *linear regression* tries to learn weights w_i such that Y can be approximated as $\sum_i w_i X_i$. Given two instances a and b their distance can be expressed as

$$d(a,b) = \sqrt{\sum_i w_i (X_{i_a} - X_{i_b})^2}$$

When $w_i = 1, \forall i$ then the distance is, by definition, the euclidean distance. In the case presented here, instead, through *linear regression* is interesting to find different weights to see if it is possible to achieve better results. Using different weights w_i means accepting the possibility that not all the features should have the same importance when evaluating the distance between instances: the assumption behind euclidean distance is that all the features contribute with the same importance to determine the distance (and this is the reason why before applying clustering all the features should be normalized so that every feature ranges in the same interval, usually [-1,1]). The purpose of learning a new distance metric, instead, is to understand if it is possible to give to some features more importance than others.

What remains next is to define the target variable Y, which represents the desired distance that is wanted among instances. Many trials have been made, for example if s_a is the solver with the best performance on instance a and s_b is the solver with the best performance on instance b and r is a given function that, given in input an instance and a solver returns the runtime of that solver on the instance, then is possible to define their distance (it is going to be the Y of the linear model) as

$$d(a,b) = \sqrt{(r(a,s_a) - r(b,s_a))^2 + (r(a,s_b) - r(b,s_b))^2}$$

In this way it is going to be measured how much the performance are penalized if for solving instance a the best solver of instance b is used and vice versa, thus grouping together instances that prefer to be solved by the same solvers.

The procedure followed to learn the distance metric is the following (still considering X_i as the numeric input variables):

- Select n instances at random (for example $n = 50$);
- Create a table with $\binom{n}{2}$ rows and a number m of columns equal to the number of features;

- For every combination i, j of the n chosen instances compute the vector of m components calculated as $(X_{i_a} - X_{i_b})^2$. For all i and j store the result in a row of the table;
- For every row of the table attach a new component (called Y) with the desired target value;
- Now is possible to run a linear regression model that will calculate the most appropriate coefficient w_i for predicting the target variable Y.

After this calculation the linear regression model will return the most appropriate weights w_i which can be used in a prediction phase when it is needed to know the distance between two instances. Unfortunately, even if it has been tried to define different target variables Y the results have been very poor.

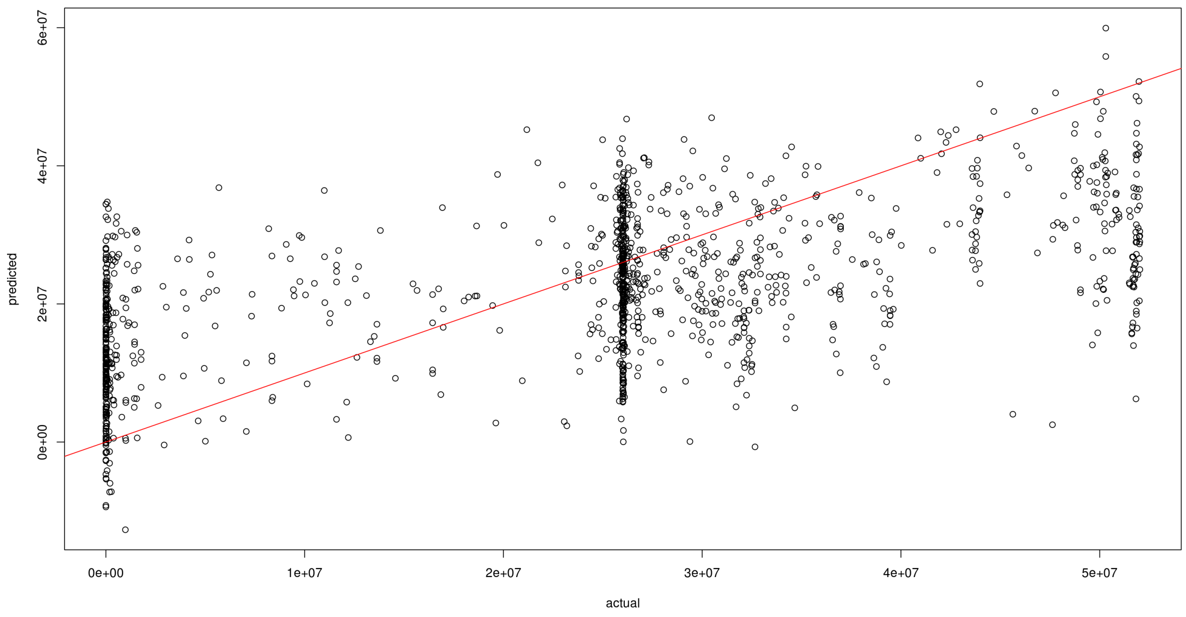
One reason of the unsuccess of this approach is the difficulty by which linear models are able to describe such a function. As can be seen in Fig. 5.1 on the x axes there is the actual distance between instances (the actual value of the target variable Y), while on the y axes there is the predicted distance evaluated applying the weights learned from the linear model (it would be expected to see the points following the red line, as it indicates almost perfection in the predicting phase). It is easy to see that the two measures almost never agree, so if two instances are supposed to be close each other they might be put far away by the just learned distance metric.

In addition to what has already been said there are also some other reasons behind the bad performance: firstly, while it is known that some instances should be closer or farther from each other, the ideal magnitude of the distance cannot be readily determined; secondly, the effectiveness of the distance function depends on nearly perfect accuracy in the prediction phase since any mistake can distort the distance space (it is not known how the distance function should behave, if in addition to this there are errors in the prediction phase the resulting function is far away from the ideal one); thirdly, the exact form of the distance function is not known. It is, for example, acceptable that even though two instances share the same best solver, they should nevertheless be allowed to be in opposite corners of the distance space. It is not necessary that every instance preferring the same solver is placed in the same cluster, but instead is better to avoid contradictory preferences within the same cluster.

5.2 Extending the Feature Space

Due to these complications, other methodologies for refining the feature vector have been proposed. Specifically (going against the principle of feature filtering) it has been proposed to extend the usual feature vector. The

Figure 5.1: Scatter plots of Linear Model on HAND dataset



experiments done in this direction, even if the results haven't been satisfying, will help in the creation and development of SNNAP. The main idea behind this is that for achieving the perfect clustering the performances of the solvers in the portfolio should be, in some way, taken into consideration when calculating the distance between instances.

One possibility for achieving this goal is to take, for each instance, the vector describing the performances of the solvers in the portfolio and to scale these runtimes between -1 and +1. In this setting, for each instance, the best performing solver is assigned a value of -1, while the worst performing is assigned to 1. Everything in between is scaled accordingly. This new vector is attached to the normal feature vector giving a space with a bigger dimensionality. The clustering, then, is done on this new extended feature space, using the Euclidean distance metric. This approach is easily applicable to the training instances: those for which the performances for the solvers are known. Unfortunately in the tesing set those performances are unknown and for solving this issue it has been decided to set those new features to 0 as a mid point.

In Table 5.1 are represented the results. Here BSS is the approach of solving all the instances in the testing set with the solver that have the best performance on the whole training set; VBS, instead, is an oracle that solves each instance in the testing set with its own best solver. The columns, together with the percentage of not solved instances, report both the average running time, both the average PAR10 (a penalized average of the runtimes: for each instance that is solved within the timeout threshold, the actual runtime in seconds denotes the penalty for that instance, instead for each instance that is not solved within the time limit, the penalty is set to 10 times the original timeout). In parenthesis are reported the standard deviations. The averages and the standard deviations are calculated from executing the experiments 10 times.

It can be easily seen that the performance of this approach (called Norm-Times ISAC) have been really poor: never comparable with the running times of the normal ISAC. The main reason was that in extending the feature space too many solvers have been taken into consideration during the computation of the new features, thus putting too many constrains in the distance calculation.

As an alternative, with better results, it has been considered that matching the performance of all solvers is too constraining. Implicitly ISAC assumes that a good cluster is one where the instances all prefer the same solver. For this reason it has been decided to take into account only the performance of the best two solvers per each instance. This was accomplished by extending the normal set of features with a vector of new features (one per each solver), and assigning a value of 1 to the components corresponding to the best two solvers and 0 to all the others. In the testing set, since are not known which are the best two solvers before hand, all the new features are set to the constant value of 0. As can be seen in Table 5.1, depending on which of the four datasets was used different results were achieved (this approach is called bestTwoSolv ISAC): it can be observed a small improvement in the hand-crafted and industrial datasets, while for the other two datasets the results were almost the same as the pure ISAC methodology. This can be taken as a good news as, at least, performances weren't weakened.

5.3 Analysis

The first idea presented in this chapter has been developed with a precise scope: ISAC showed some issues as it comes to evaluating the quality of the discovered clusters. It is highly probable that in the same cluster are put instances which show completely different level of hardness in terms of solving: this doesn't come as a good news as ISAC relies heavily on the

Table 5.1: Results on the SAT benchmark, comparing the Best Single Solver (BSS), the Virtual Best Solver (VBS), the original ISAC approach (ISAC), the ISAC approach with extra features coming from the running times: "NormTimes ISAC" has the normalized running times while "BestTwoSolv ISAC" takes into consideration just the best two solvers per each instance.

RAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	1551 (0)	13154 (0)	25.28 (0)
ISAC	826.1 (6.6)	4584 (40.9)	8.1 (0.2)
NormTimes ISAC	1940 (9.1)	15710 (133.2)	30 (1.1)
BestTwoSolv ISAC	825.6 (5.7)	4561 (87.8)	8.1 (0.2)
VBS	358 (0)	358 (0)	0 (0)
HAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	2080 (0)	15987 (0)	30.3 (0)
ISAC	1743 (34.4)	13994 (290.6)	26.5 (0.9)
NormTimes ISAC	1853 (37)	14842 (310.7)	28.3 (1.3)
BestTwoSolv ISAC	1725 (29.2)	13884 (124.4)	26.5 (0.8)
VBS	400 (0)	400 (0)	0 (0)
INDU	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	871 (0)	4727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3166 (155.6)	5.2 (0.7)
NormTimes ISAC	934.3 (5.4)	5891 (187.8)	10.8 (1.4)
BestTwoSolv ISAC	750.5 (2.4)	2917 (157.3)	4.7 (0.4)
VBS	319 (0)	319 (0)	0 (0)
ALL	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	2015 (0)	4727 (0)	30.9 (0)
ISAC	1015 (10.3)	6447 (92.4)	11.8 (0.2)
NormTimes ISAC	1151 (13.2)	6923 (170.6)	12.5 (0.8)
BestTwoSolv ISAC	1019 (11.5)	6484 (172.3)	11.9 (0.3)
VBS	353 (0)	353(0)	0 (0)

descriptivity of the features in order to find a hidden structure that should be revealed with clustering. In Chapter 4 has been tried to see if filtering non relevant features might help the procedure improving the quality of clustering. This has been the first approach but, obviously, not the only one possible. The main quest has been to answer to the question: is it possible to help in some way clustering?

The idea of learning a new distance function has been promising but it didn't prove to be beneficial: the general assumption behind euclidean distance (all features have the same importance) is the opposite of the idea behind feature filtering (there are some features more important than others). Unfortunately the obtained results didn't prove the hypothesis but, nevertheless, it doesn't have to be considered a dead end. The reason why the approach failed has to be looked in the difficulty of setting an appropriate target variable Y that is able to describe the wanted relationship among instances. It is unknown how two instances should relate themselves when they show similar performances: it is not obvious that they should be close together. And even if the assumptions behind Y were correct it has been found that the hard part is to be able to predict the distance using a linear regression model. The reason why has been chosen to rely on linear regression is that is an easy and quick model. Once the weights w_i have been learnt is just a matter of pluggin them in when evaluating the distance among instances. In case of using a more complex predictive model the results might be more precise but the calculation would be much slower, which is simply not feasible in settings with thousands of instances.

The second tried approach dealed with directly extending the feature space with some information coming from the performance of the solvers. The drawbacks of this technique are two-fold. Firstly, the performance of the solvers is not available prior to solving a previously unseen test instance and for this reason some decision has to be made in order to deal with those missing values in previously unseen instances. Secondly, even in the case in which the new added features are helpful in determining a better clustering, there is usually a large number of original features that might be resisting the desired clustering. At last is not known how to actually extend the features vector: here have been presented two possibilities but, obviously, they are not the only ones.

It would be interesting here to study and develop some techniques that would be able to analyse the current clustering and create some ad-hoc features with the precise scope of modifying grouping of the instances and moving them from cluster to cluster.

Yet even though these extensions to the feature vector did not provide a compelling case to be used instead of the vanilla ISAC approach, they nonetheless supported the assumption that by considering solver performances on the training data it is possible to improve the quality of the overall clustering. This has inspired the Solver-based Nearest Neighbor approach that will be described in the next chapter.

5.4 Chapter Summary

Taking an approach completely on the opposite side respect to the previous chapter this one is related to the exploration of different techniques for extending the feature space instead of reducing it. The ultimate goal is adding new features in a way to help clustering to achieve a perfect grouping: the one in which instances that prefer to be solved by the same set of solvers are grouped together in the same cluster or, at least, where are avoided contradictory preferences within the same cluster.

Even if it has not been possible to significantly improve the performance of the standard ISAC methodology at the end of the chapter has been presented a technique that shows that is possible in some way to help clustering: furthermore this technique, or even better the idea behind it, is the starting point for the development of the here presented SNNAP methodology.

Chapter 6

SNNAP: Solver-based Nearest Neighbor for Algorithm Portfolios

It is important, at this point, to have in mind what is the ultimate goal: the only way to improve the vanilla ISAC methodology is to tweak the system in order to achieve a better clustering that will help to find the optimal solvers for solving the instances. Getting a better clustering means that is highly wanted to have a grouping of the instances such that all the instances in one cluster prefer to be solved by the (idealistically) exact same solver, or at least the loss in performance in solving one instance with the chosen solver is small.

There are two main takeaway messages from extending the feature vector with solver performances as seen before. First, the addition of solvers performance can be helpful, but the inclusion of the original features can be disruptive for finding the desired cluster. Second, it is not necessary to find instances where the relation of every solver is similar to the current instance. It is, in general, enough to just know the best two or three solvers for an instance. Using these two ideas is here introduced SNNAP [15], presented and described in Algorithm 4.

The methodology moves from what has been proposed before because clustering will not be used anymore in favour of a k-NN approach (that can be seen as dynamic clustering). Every instance will be solved using the best solver among its k nearest instances in the training set. Moreover *random forests* will also be used [39]. *Random forests* are machine learning methods that make their decisions based on the results of a set of decision trees constructed at random.

6.1 Algorithm

The starting point is a dataset composed by a list of training instances T. The dataset can be split in two parts, where each row corresponds to one instance: the first part is represented by the features vectors F, while the second part by the running times R of every solver in the portfolio. For example in the RAND (SAT) dataset the input is a table (dataset) with 1949 rows and 144 columns of which the first 115 are features and the remaining 29 are runtimes.

The second thing has been to scale the running times of the solvers on one instance so that the scaled vector will have a mean of 0 and unitary standard deviation. Doing so, for every instance, every solver that behaves better than one standard deviation from the mean will receive a score less than -1, the solvers which behaves worse than one standard deviation from the mean a score greater of 1, and the others will lie in between. This scaling is done for every instance in the training set. This kind of scaling has been crucial in helping the following phase of prediction. As it is obvious at this point the model will not be trained to predict the actual run times, but will be trained to predict just when a solver will perform much better than usual.

The following phase is responsible for training a single model PM for every solver in the portfolio to predict the expected performance on a given instance. This means that based on the features of the instances of the training set m models (in the SAT competition $m = 29$ as the number of available solvers) are trained for the purpose of being able to predict the performance of all the solvers on the testing set. The employed predictive model are *random forests*. These models for being trained need in input the training instances T, their features F and their scaled running times $\bar{R}$. Earlier has been claimed that such models are difficult to train properly since any misclassification can result in the selection of the wrong solver. In fact, this was partly why the original version of ISAC outperformed these types of regression-based portfolios. Clusters, on the other hand, provide better stability of the resulting prediction of which solver to choose. However, here, the purpose is not to use the trained models to predict the single best solver to be used on each instance but the goal is just to know which set of solvers are going to behave better on a particular instance compared to the other solvers in the portfolio. Here using the scaled running times instead of the actual running times makes no difference, the best solver will still have the lowest (scaled) running time.

The second phase is the one related to the prediction part: given a previously unseen instance SNNAP wants to predict which solver will be the best to solve it. The procedure is presented with a previously unseen instance x, the prediction models PM (one per each solver) previously built, the training instances T, their running times R (and the scaled version $\bar{R}$),

Algorithm 4 Solver-based Nearest Neighbor for Algorithm Portfolios

```
1: function SNNAP-Train($(T, F, R)$)
2:     for all instances $i$ in $T$ do
3:         $\bar{R}_i \leftarrow Scaled(R_i)$
4:     end for
5:     for all solver $j$ in the portfolio do
6:         $PM_j \leftarrow PredictionModel(T, F, \bar{R})$
7:     end for
8:     return $PM$
9: end function

1: function SNNAP-Run($(x, PM, T, R, \bar{R}, A, k)$)
2:     $PR \leftarrow Predict(PM, x)$
3:     $dist \leftarrow CalculateDistance(PR, T, \bar{R})$
4:     $neighbors \leftarrow FindClosestInstances(dist, k)$
5:     $j \leftarrow FindBestSolver(neighbors, R)$
6:     return $A_j(x)$
7: end function
```

the portfolio of solvers A and the size of the desired neighborhood k. The procedure first uses the prediction models to infer the performances PR of the solvers on the instance x, using its originally known features. With this step the algorithm will have a vector of dimension m (the number of solvers) and in each bucket there will be a prediction about the performance of the solver on the instance. SNNAP then continues to use these performances to compute a distance (as will be described later) between the new instance and every training instance (the result of this calcualtion is a matrix with one row and $|T|$ columns, each bucket will have the distance between instance x and each instance in the training set), selecting, then, the k nearest among them. The distance calculation takes into account only the scaled running time of the instances of the training set and the predicted performances PR of the different solvers on the instance x. At the end the instance x will be solved using the solver that behaves better (measured as the average running time) on the k neighbors previously chosen.

It is worth highlighting again that the procedure is not trying to predict the actual running times of the solvers on the instances but, after scaling, predicts a ranking among the solvers on a particular instance: which will be the best, which the second best, etc. Moreover, as shown in the next section, is not necessary learning a ranking among all the solvers, but just among a small subset of them, specifically for each instance which will be the best n solvers, with n a small number (typically from 2 to 4).

6.1.1 Choosing the Distance Metric

The k-nearest neighbors approach is usually used in conjunction with the weighted Euclidean distance; unfortunately the Euclidean distance does not take into account the performances of the solvers in a way that is helpful to the purpose of SNNAP. The message that can be taken from the previous chapter is that it is possilbe to intelligently use the performances of the solvers in the grouping phase, so that the performance of the approach can be significantly improved.

The issue behind the euclidean distance is that it takes into account only the features of the instances, but they don't give any clue on the solving times. Instead of the usual Euclidean distance what is needed is a distance metric that takes into account the performances of the solvers. Moreover recalling that the SNNAP procedure has a predicting phase where it tries to predict a ranking among solvers on a particular instance is crucial that the chosen distance metric would allow the possibility of making some mistakes in the prediction phase without too much prejudice on the overall performances. Thus the metric should be trained with the goal that the k-nearest neighbors always prefer to be solved by the same solver while instances that prefer different solvers are separated by a large margin.

Given two instances a, b and the running times of the m algorithms in the portfolio A on both of them $R_{a_1}, \dots R_{a_m}$ and $R_{b_1}, \dots R_{b_m}$, SNNAP identifies which are the best n solvers on each $(A_{a_1}, \dots A_{a_n})$ and $(A_{b_1}, \dots A_{b_n})$ and define the distance between a and b as a Jaccard distance:

$$1 - \frac{|intersection((A_{a_1}, \dots A_{a_n}), (A_{b_1}, \dots A_{b_n}))|}{|union((A_{a_1}, \dots A_{a_n}), (A_{b_1}, \dots A_{b_n}))|}$$

Using this definition two instances that will prefer the exact same n solvers will have a distance of 0, while instances which prefer completely different solvers will have a distance of 1. Moreover, using this kind of distance metric it is no longer important making small mistakes in the prediction phase: even if the ranking between the best n solvers is switched the distance between two instances will remain the same. In the experiments, in general, has been set arbitrarly $n = 3$, as with higher values the performance degrades.

Previous attempts

Before coming up with the idea of the just defined distance metric other possibilities have been tried but, unfortunately, they didn't give the expected results.

One idea that was supposed to give good results was based on the following assumption: if a solver solves an instance in 2 seconds it is not much different than solving it in 9 seconds, as well 300 seconds is more or less

Table 6.1: Example of an erroneous prediction.

	S_1	S_2	S_3	S_4	S_5
a	13	5100	5100	15	87
b	256	5100	5100	200	787

equivalent to 800 seconds. The runtimes can be divided per order of magnitude, stating that solvers are equivalent in case they are able to solve one instance in the same order of magnitude of time.

At this point assume that it is known that the fastest solver on a particular instance solves it in order of 10^2, for this purpose it can be said that is equivalent to solve it with another solver that is able to solve it in the same order of magnitude of time. Given that at this point are known (in the training set) all the solvers that solve a particular instance in the lowest order of magnitude (it is not important whether it is $10^0, 10^1, \ldots$) those solvers can be used to define the distance metric as defined before (as a Jaccard distance).

As before there is also a predicting phase, given that in the testing set is not possible to know the actual running times of the solver on the instances: it is sufficient to be able to predict the order of magnitude in which a solver will solve the instance. This task may seems easy: the categories to be predicted are only five ($10^0, 10^1, 10^2, 10^3$ and "timeout"). Unfortunately the classes are highly unbalanced, with the vast majority of instances in the 10^3 and "timeout" categories. This means that just predicting everything to be in the majority class gives a high accuracy[1] ($\sim 80 - 85\%$). Moreover with this kind of distance metric even the smaller mistakes in the predicting phase cause two instances that should be close each other to fall far away in the space.

Here is just a small example that shows the drawbacks of an erroneous prediction with this approach. Assume that the dataset has 5 solvers S_i and there are two instances: a and b. Table 6.1 gives the running times of the solvers on the two instances (5100 are timeouts).

For instance a the best solvers are S_1 and S_4, that are exactly the same as for instance b, so using the Jaccard distance the distance between a and b is 0, even if the magnitude between the two instances are different (10^1 for a and 10^2 for b). Suppose now that instance b has never been seen before so SNNAP has to predict the performances of the solvers. Unfortunately the predictive models make one small mistake and predict the following magnitude for instance b: ($10^2, 10^1$, "timeout", $10^2, 10^2$), so the magnitude of solver S_2 is uncorrect. Now the distance between a and b is 1, exactly the maximum distance possible. With the distance metric used by SNNAP,

[1]Predictions correctly made.

instead, (and considering only the best three solvers) the distance is 0.5, so a better estimate.

6.2 Numerical Results on SAT

In the carried experiments (for the SAT datasets), using the Jaccard distance metric, for each instance it is sufficient to know which are the n best solvers. In the prediction phase, as already stated, are used random forests which achieved high levels of accuracy: as seen in Table 6.2 the accuracy levels are 91, 89, 91 and 91% (respectively RAND, HAND, INDU and ALL datasets) of the predictions. These percentages have been calculated in the following manner: there are 29 predictions made (one per each solver) per each instance, giving a total of 5626, 1044, and 2320 predictions per category. The accuracy is then defined as the percentage of matches between the predicted best n solvers and the true best n. To compute the accuracy it is useful to think of two matrices (one the real matrix and one the predicted matrix). Both matrices are binary: filled with zeroes and ones. Here ones represent entries corresponding to the top three solvers per each instance, and zeroes are for everything else. The accuracy is then the percentage of cells with the same value in the two matrices.

As seen before is not important to be able to make those predictions with nearly perfection: the achieved accuracies are precise enough for the scope.

Table 6.2: Statistics of the four datasets used: instances generated at random "RAND", hand-crafted instances "HAND", industrial instances "INDU" and the union of them "ALL".

	RAND	*HAND*	*INDU*	*ALL*
Number of instances considered	1949	363	805	3117
Number of predictions	5626	1044	2320	9019
Accuracy in the prediction phase	91%	89%	91%	91%

Having tried different parameters has been choosen arbritrarly to take into consideration only the performance of just the $n = 3$ best solvers in the calculation of the distance metric and a neighborhood size of $k = 60$. Choosing a larger number of solvers may lead to a loss in the results. This is most likely due to scenarios where one instance is solved well by just a limited number of solvers, while all the others time out.

In Table 6.3 are reported the results. Here BSS is the approach of solving all the instances in the testing set with the solver that have the best performance on the whole training set; VBS, instead, is an oracle that solves each instance in the testing set with its own best solver. The columns, together

Table 6.3: Results on the SAT benchmark, comparing the Best Single Solver (BSS), the original ISAC approach (ISAC), the SNNAP approach (SNNAP) (also with feature filtering) and the Virtual Best Solver (VBS)

RAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	1551 (0)	13154 (0)	25.28 (0)
ISAC	826.1 (6.6)	4584 (40.9)	8.1 (0.2)
SNNAP	791.4 (15.7)	4119 (207)	7.3 (0.2)
SNNAP + Filtering	723 (9.27)	3138 (76.9)	5.28 (0.1)
VBS	358 (0)	358 (0)	0 (0)
HAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	2080 (0)	15987 (0)	30.3 (0)
ISAC	1743 (34.4)	13994 (290.6)	26.5 (0.9)
SNNAP	1063 (33.86)	6741 (405.5)	12.4 (0.4)
SNNAP + Filtering	995.5 (18.23)	6036 (449)	10.5 (0.4)
VBS	400 (0)	400 (0)	0 (0)
INDU	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	871 (0)	4727 (0)	8.4 (0)
ISAC	763.4 (4.7)	3166 (155.6)	5.2 (0.7)
SNNAP	577.6 (21.5)	1776 (220.8)	2.6 (0.4)
SNNAP + Filtering	540 (15.52)	1630 (149)	2.4 (0.4)
VBS	319 (0)	319 (0)	0 (0)
ALL	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	2015 (0)	4727 (0)	30.9 (0)
ISAC	1015 (10.3)	6447 (92.4)	11.8 (0.2)
SNNAP	744.2 (14)	3428 (141.2)	5.8 (0.2)
SNNAP + Filtering	692.9 (7.2)	2741 (211.9)	4.5 (0.1)
VBS	353 (0)	353 (0)	0 (0)

with the percentage of not solved instances, report both the average running time, both the average PAR10 (a penalized average of the runtimes: for each instance that is solved within the timeout threshold, the actual runtime in seconds denotes the penalty for that instance, instead for each instance that is not solved within the time limit, the penalty is set to 10 times the original timeout). In parenthesis are reported the standard deviations. The averages and the standard deviations are calculated from executing the experiments 10 times.

In this case the best improvement, as compared to the standard ISAC, is achieved in the Hand-crafted dataset. Not only are the performances improved by a factor of 40%, but also the percentage of unsolved instances is halved; this also has a great impact on the PAR10 evaluation. It is interesting to note that the hand-crafted dataset is the one that proves to be most difficult, in terms of solving time, while being the setting in which is achieved the most improvement.

A significant improvement is also achieved, although lower than that with the Hand-crafted dataset, on the Industrial and ALL ($\sim$ 25% and 27%) datasets. Here the number of unsolved instances was also halved. In the random dataset has been accomplished the lowest improvement but, yet, it was possible to overtake significantly the standard ISAC approach.

Interesting, as seen in chapter 4, the Hand-crafted dataset was the only one were *gmeans* didn't prove to be the best clustering technique to use. Being that, on the opposite side, SNNAP produced the best improvement in this dataset, is possible to conclude that ISAC, with *gmeans*, is not able to exploit at its best the feature space of those instances.

Being that feature filtering achieved promising results with ISAC it has been tried to apply feature filtering also to SNNAP and the results are shown in Table 6.3. Feature filtering is again proving beneficial, significantly improving the results for all our datasets and giving, again, a clue that not all 115 features are essential. Results in the table have been reported only for the more successful ranking function (*gain.ratio* for the Random dataset, *chi.squared* for Hand-crafted and Industrial and the overAll dataset).

These consistent results for SNNAP are encouraging. In particular, it is clear that the dramatic decrease in the number of unsolved instances is highly important, as they are a key in lowering the average and the PAR10 scores. This result can also be observed in Table 6.4, where are presented the percentage of instances solved/not solved by each approach. In particular the most significant result is achieved, again, in the HAND dataset where the number of instances not solved by ISAC, but solved by SNNAP is 17.4% of the overall instances, while the number of instances not solved by SNNAP but solved by ISAC is only 3.3%. This difference is also considerable in the other three datasets. Deliberately, it has been chosen to show this matrix only for the version of ISAC and SNNAP without feature filtering as it offers an unbiased comparison between the two approaches, as has been shown that ISAC does not improve significantly after feature filtering.

Another useful statistic is represented by the number of times that an approach is able to select the best solver for a given instance. In the random dataset SNNAP is able to select the best solver for 39% of the instances, as compared with 35% for ISAC. For the Hand-crafted dataset those values are 25% and 17%, respectively, for the Industrial 29% and 21%, respectively, and for the ALL 32% and 26%, respectively. These values suggest that ISAC is already behaving well on the RAND dataset and, for this reason, the improvement achieved with this new approach is smaller in that case, while the improvement is more significant in the other three datasets. These results also show that there is still room for further improvement.

Continuing in the analysis of the results it is useful to analyse the frequencies with which the solvers are chosen by the three strategies for the SAT dataset: the Virtual Best Solver (VBS), the Best Single Solver (BSS),

Table 6.4: Matrix for comparing instances solved and not solved using SNNAP and ISAC for the four datasets: RAND, HAND, INDU and ALL. Values are in percentages.

RAND		
SNNAP \ISAC	*Solved*	*Not Solved*
Solved	89.4	3.3
Not Solved	2.5	4.8

HAND		
SNNAP \ISAC	*Solved*	*Not Solved*
Solved	70.2	17.4
Not Solved	3.3	9.1

INDU		
SNNAP \ISAC	*Solved*	*Not Solved*
Solved	93.9	3.5
Not Solved	0.9	1.7

ALL		
SNNAP \ISAC	*Solved*	*Not Solved*
Solved	85.8	8.4
Not Solved	2.4	3.4

ISAC and SNNAP. Table 6.5 presents the frequency with which each of the 29 solvers in the portfolio were selected by each strategy, highlighting the best single solver (BSS) for each category. In this table is clear that ISAC tends to favor selecting the Best Single Solver, instead of choosing the Virtual Best Solver (VBS). This is particularly visible in the Hand-crafted dataset: the BSS (15) is chosen only in 8% of the instances by the VBS, while it is the more frequently chosen solver by ISAC. Note also that in the ALL dataset the VBS approach never chooses the BSS, while this solver is one of the top three chosen by ISAC. On the other hand, the more often a solver is chosen by the VBS, the more often it is chosen by SNNAP. This big discrepancy between the VBS and BSS in this dataset is one of the reason for the poorer performance of ISAC and one of the reasons for the improvement observed when using SNNAP.

6.2.1 Tuning the parameters for SNNAP

Until now has been decided to arbitrarly decide to consider only 3 solvers in the evaluation of the Jaccard distance metric and to use a k-nn approach with a fixed $k = 60$ independently on the dataset used.

The developed software, instead, gives to the user the possibility to find automatically which is the best setting for those two parameters. Trivially are tried all the combinations of the two parameters inside a fixed range (e.g.: from 2 to 10 solvers and with k between 20 and 120).

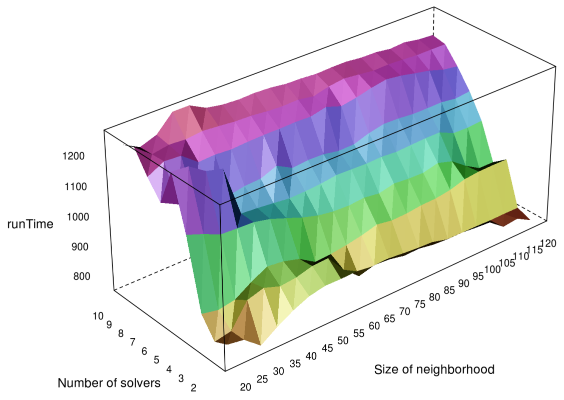
Interestingly, as shown in Fig. 6.1 the size of the neighborhood doesn't affect significantly the performance for SNNAP: with a fixed number of solvers the performances remain comparable with different values of k. The behaviour is specially visible in the two datasets with the biggest number

Table 6.5: Frequencies of solver selections for VBS, ISAC and SNNAP. Results are expressed as percentages and entries with value < 0.5 have been reported as '-'. In bold the top three solvers for each approach are identified. Reported are also the Best Single Solver (BSS) for each dataset. Solvers are: 1: clasp-2.1.1_jumpy, 2: clasp-2.1.1_trendy, 3: ebminisat, 4: glueminisat, 5: lingeling, 6: lrglshr, 7: picosat, 8: restartsat, 9: circminisat, 10: clasp1, 11: cryptominisat_2011, 12: eagleup, 13: gnoveltyp2, 14: march_rw, 15: mphaseSAT, 16: mphaseSATm, 17: precosat, 18: qutersat, 19: sapperlot, 20: sat4j-2.3.2, 21: sattimep, 22: sparrow, 23: tnm, 24: cryptominisat295, 25: minisatPSM, 26: sattime2011, 27: ccasat, 28: glucose_21, 29: glucose_21_modified.

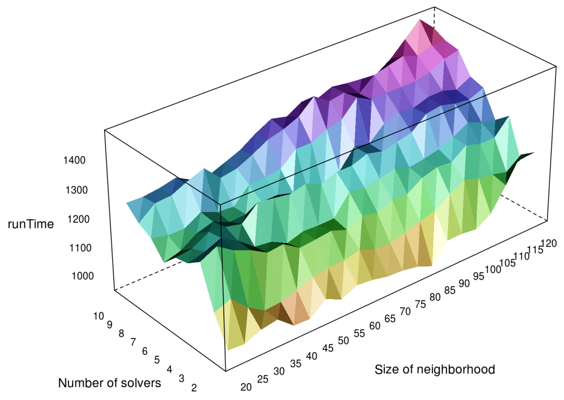
RAND	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100	-	-
VBS	-	-	-	-	-	-	-	-	-	-	-	17	6	**19**	2	-	-	-	-	-	3	**18**	5	-	-	4	**24**	-	-
ISAC	-	6	1	-	-	-	-	-	-	-	-	8	-	**20**	-	**12**	-	-	-	-	-	-	-	-	-	-	**52**	-	-
SNNAP	-	5	-	-	-	-	-	-	-	-	-	3	1	**27**	2	2	-	-	-	-	0	**12**	4	-	-	2	**41**	-	-
HAND	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100	-	-	-	-	-	-	-	-	-	-	-	-	-	-
VBS	-	3	1	2	2	1	**7**	-	1	**13**	4	-	5	7	8	1	3	2	-	1	**7**	**10**	5	3	1	6	3	2	2
ISAC	-	**22**	2	1	1	1	2	-	-	1	-	2	-	1	**40**	-	-	-	-	-	-	**26**	-	-	-	-	-	-	-
SNNAP	-	10	1	1	1	-	-	-	-	**32**	-	-	-	1	**25**	-	-	-	-	-	2	4	-	4	-	**19**	-	-	-
INDU	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	-	-	-	-	100	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
VBS	-	5	1	3	**15**	**8**	3	2	5	4	4	-	-	-	3	1	1	9	-	-	-	-	-	4	5	-	-	**20**	3
ISAC	-	1	-	1	**31**	5	-	-	6	-	**17**	-	-	-	-	-	-	5	-	-	-	-	-	-	2	-	-	**32**	-
SNNAP	-	1	-	1	**38**	-	-	-	-	-	2	-	-	-	4	1	-	**19**	-	-	-	-	-	4	-	-	-	**29**	-
ALL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
BSS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100	-	-	-	-	-	-	-	-	-	-	-	-	-
VBS	-	2	-	1	4	2	2	1	1	3	2	11	4	**13**	3	-	1	3	-	-	3	**13**	4	2	1	3	**15**	5	1
ISAC	-	6	1	1	**14**	-	-	-	2	-	-	2	-	12	5	**12**	-	-	-	-	-	4	-	4	-	2	**31**	2	-
SNNAP	-	5	-	1	10	-	-	-	-	3	-	2	1	**18**	2	5	-	3	-	-	-	**11**	2	3	-	2	**24**	4	3

Figure 6.1: How varying the size of the neighborhood and the number of solvers considered affects performances.

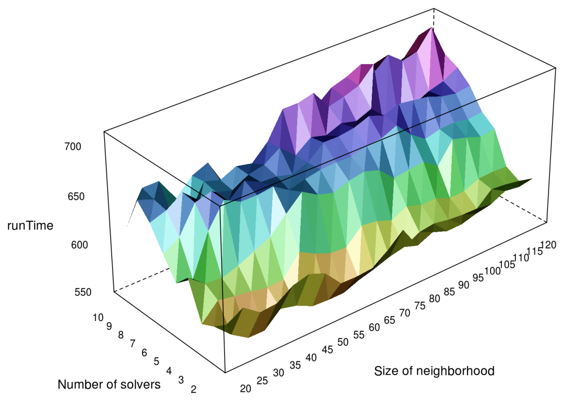
(a) RAND dataset



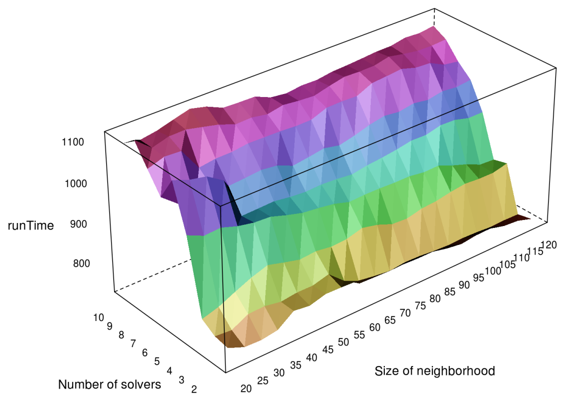
(b) HAND dataset



(c) INDU dataset



(d) ALL dataset



of instances (RAND and ALL), while for the other two datasets the plots show that it is better choosing a size of the neighborhood less than 70-90 instances.

The performances, on the other hand, are much more affected by the variation in the number of solvers used in the calculation of the distance metric. This behaviour could be explained analysing the dataset: most of the instances are solved by a small number of solvers while the others are in time-out. As a silly example consider the two instances in Tab. 6.6: if considering only 3 solvers their distance is 0. If instead the number of solvers to be considered is increased to 4 then their distance become 3/7(for instance a all the solvers are then considered as S_1, S_2, S_6, S_7 have equal performances). On the opposite side than before, here, the datasets affected the most by the variations in the number of solvers are the RAND and ALL.

Table 6.6: Erroneous evaluation taking into account too many solvers.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
a	5100	5100	103	657	1430	5100	5100
b	5100	5100	1	37	22	766	5100

6.3 Numerical Results on MaxSAT

In this section are going to be presented the results on a different dataset: the instances come from the problem of Maximum Satisfiability (MaxSAT) which has been presented in chapter 3.

Initially the dataset is composed by 2679 instances, 38 features each and the recorded running times of 14 solvers; the instances that remain after that the ones deemed too easy and the ones not solved by any solver have been filtered out are 2162.

The results of ISAC and SNNAP on this dataset are presented in Tab. 6.7. Here BSS is the approach of solving all the instances in the testing set with the solver that have the best performance on the whole training set; VBS, instead, is an oracle that solves each instance in the testing set with its own best solver. The columns, together with the percentage of not solved instances, report both the average running time, both the average PAR10 (a penalized average of the runtimes: for each instance that is solved within the timeout threshold, the actual runtime in seconds denotes the penalty for that instance, instead for each instance that is not solved within the time limit, the penalty is set to 10 times the original timeout). In parenthesis are reported the standard deviations. The averages and the standard deviations are calculated from executing the experiments 10 times.

The feature filtering function used has been *chi.squared*, the size of the neighborhood for SNNAP has been set to 25 and the number of solvers considered is 2.

Compared to previous results the first thing to note is that the number of instances non solved by the approaches is very small (between 1 and 5 instances); this is caused by the really good performances of the solvers in the portfolio. Moreover, it is worth to be reminded, instances deemed too hard (not solved by any solver) or too easy have been deleted from the dataset.

Also for MaxSAT the gain obtained by using SNNAP compared to ISAC is significant but, on the opposite hand, feature filtering doesn't prove to be significantly beneficial (in ISAC is even worst). The reason has to be looked in the fact that the instances in the dataset have already few features (38 in MaxSAT compared to more than a hundred in SAT).

Table 6.7: Results on the MaxSAT benchmark, comparing the Best Single Solver (BSS), the original ISAC approach (ISAC) (also with feature filtering), the SNNAP approach (SNNAP) (also with feature filtering) and the Virtual Best Solver (VBS)

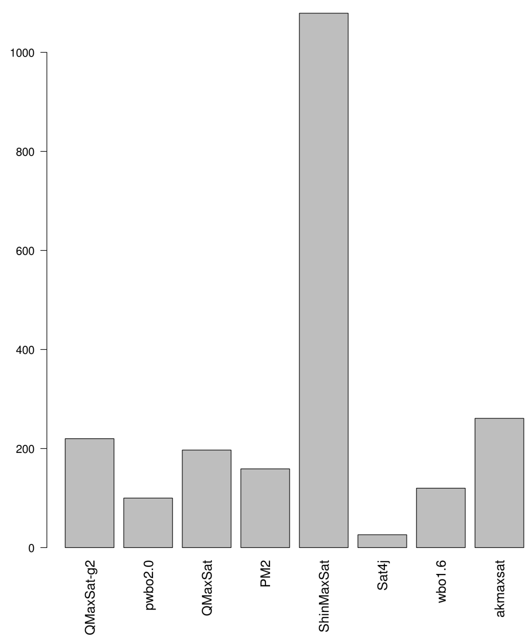
RAND	*Runtime - avg (std)*	*Par 10 - avg (std)*	*% not solved - avg (std)*
BSS	891.9 (0)	891.9 (0)	0 (0)
ISAC	276 (4.9)	278 (6.8)	0.1 (0.1)
ISAC + Filtering	333.2 (18.5)	333.2 (18.5)	0.1 (0.1)
SNNAP	148.1 (9.7)	159.4 (22)	0.1 (0.2)
SNNAP + Filtering	155.1 (11)	181.1 (21)	0.2 (0.2)
VBS	87.54 (0)	87.54 (0)	0 (0)

As done previously with SAT is it useful to compare the used solvers by ISAC and SNNAP. The analysis is presented in Fig. 6.2. The Best Single Solver for this dataset is *ShinMaxSat*. It is clear as even in this case ISAC tends to select the BSS too often (is the most preferred solver by the approach), while SNNAP is able to differentiate itself and the most selected solver is *PM2*. *PM2*, moreover, is the most selected solver by the VBS (26.2% of the instances), while *ShinMaxSat* is selected in the 25.1% of the cases. ISAC is able to select the VBS for the 37.6% of the instances while SNNAP succeeds in the 45.4% of the cases. These higher percentage compared to SAT are also one of the reason of the low number of unsolved instances.

As presented in Fig. 6.3 it is clear that also for MaxSAT the most influencing parameter in SNNAP is the number of solvers considered, with a clear predominance of low values. Moreover for the size of the neighborhood is to be preferred to keep the number of instances lower than 50-60.

Figure 6.2: Used solvers by ISAC and SNNAP on instances of MaxSAT.

(a) ISAC



(b) SNNAP

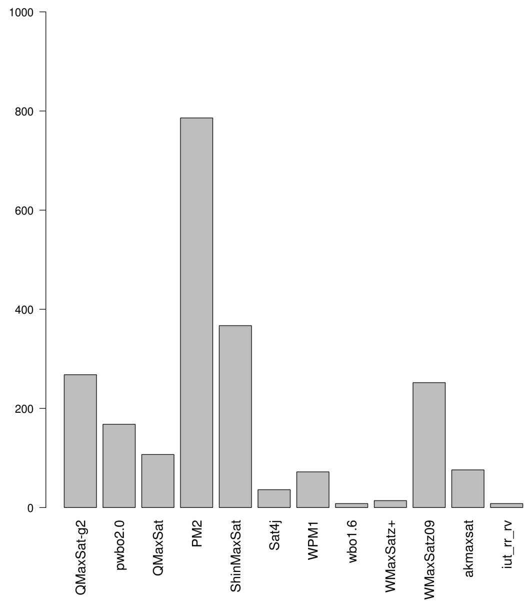
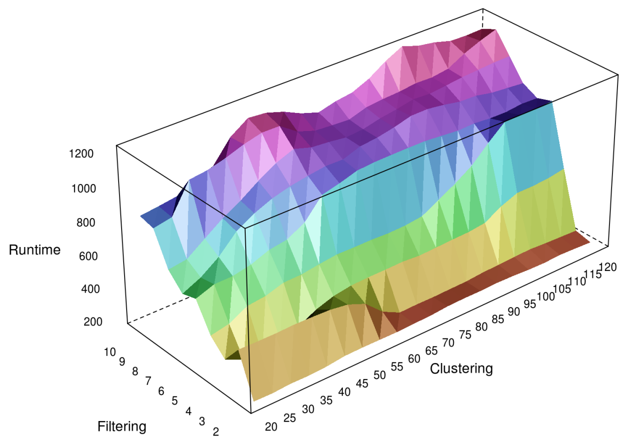


Figure 6.3: How the performance of SNNAP are affected varying the parameters in MaxSAT.



6.4 Chapter Summary

In this chapter has been presented SNNAP: Solver-based Nearest Neighbor for Algorithm Portfolios a novel methodology that tries to improve the state of the art in the field of Algorithm Portfolios. The presented approach starts from the willingness of improving the basic assumptions behind the standard ISAC technique which relies entirely on the provided features. SNNAP, instead, tries to include past performances of solvers for helping the decision of the solver to be used on a particular instance. The basic assumption (based on the observation of ISAC performances) is that the standard feature vector is not enough descriptive in the choice of the solver to be used: instances that prefer to be solved by the same solver need to be close together in the space so that a clustering approach could group them.

Combining different machine learning techniques has been shown how is possible to predict which solvers are going to behave well on a particular instance and basing the decision on the k nearest instances is possible to chose one to solve it.

The beneficial effects of SNNAP have been proved on different datasets coming from different NP-hard problems: four datasets from SAT and one from MaxSAT, showing how has been possible to outperform the standard ISAC methodology.

Chapter 7

Implementation of the software

The paper describing SNNAP has been presented at ECML 2013 [16] and accepted for publication in the conference's proceedings. The impact of scientific research, however, is many time limited by the irreproducibility of the results. For promoting further developments in the area of algorithm portfolios and for encouraging the use of SNNAP from other research groups a goal of the thesis has always been to release the code through an open souce licence (LGPL).

It has been released the developed code as well as the datasets used for the experiment. The aim of releasing the code, moreover, gave further incentives focussing on maximum clarity of the documentation, on the programming style (including comments) and on the easiness of the installation and execution of SNNAP.

The software is downloadable from [17].

7.1 Overview

The software has been entirely developed in R [1] and has been succesfully tested with the versions 3.0.0 and 2.15.1.

The code is implemented for giving to users the possibility to try two algorithm portfolios approaches: ISAC and SNNAP, with the possibility to apply feature filtering. In particular the implementations of the two approaches are the one described in [24, 15]. Even though in this thesis many more techniques have been described only the most successful have been released.

The released software allows the users to apply SNNAP to their datasets, and in turn compare the results with ISAC, the performance of the best

[1]http://www.r-project.org/

single solver, and the virtual best performance that can be achieved by a portfolio approach. The code is made available under the LGPL[2] license in hopes of helping promote further development and comparisons with existing and future portfolio methodologies.

The provided code is an R script that performs different algorithm portfolios techniques:

- SNNAP;
- SNNAP with feature filtering;
- ISAC;
- ISAC with feature filtering;
- Best Single Solver (BSS);
- Virtual Best Solver (VBS).

The software can be downloaded as a *zip* file and it is structured in folders:

- *SNNAP/*: Main Directory;
  - *src/*: The folder with the actual code for SNNAP, ISAC and *gmeans*;
  - *data/*: The folder containing the dataset;
  - *results/*: The folder where the results will be saved;
  - *optimalParam/*: The folder where the optimal parameters for SNNAP will be saved.

The main file is called SNNAP.R, is an R script, and is placed in the *src* folder. At the beginning it contains a line which defines a vector called *"approaches"* which is responsible for the declaration of which approaches should be executed among the available ones.

The script gives the user the possibility of changing amongst different clustering techniques (for ISAC), as well as the feature filtering ranking function.

The scoring functions available are (*FSelector* [37] package used):

- *chi.squared*;
- *information.gain*;
- *gain.ratio*;

[2]http://opensource.org/licenses/LGPL-3.0

- *symmetrical.uncertainty.*

while the clustering approaches among which users can chose are:

- *gmeans*;
- *XMeans* [40];
- *hclusterpar* [41].

The source code for gmeans is provided as it has been implemented in R and is placed in the *src* folder.

7.2 Installation

The software makes use of different external R libraries that should be installed in the computer (as well as a recent version of R). In particular these libraries are:

- *nortest*: for executing a normality test in gmeans;
- *foreach*: for letting the parallel execution of each fold of the script;
- *doMC*: for exploiting the multicore capabilities of the machine;
- *lattice*: used for plotting a 3D surface with the optimal parameters for SNNAP;
- *FSelector*: used for feature selection;
- *RWeka*: for the XMeans clustering technique;
- *amap*: for the hierarchical clustering technique;
- *randomForest*: used in SNNAP for predicting the performances of solvers.

Each library should be installed following the instructions at `http://cran.r-project.org/doc/manuals/r-release/R-admin.html#Installing-packages` . For example, if connected to internet is sufficient to type the following command in an R console:

```
install.packages("[Name of the package]")
```

e.g.:

```
install.packages("amap")
```

To make the installation of the additional packages quicker and easier to the user in the main directory of SNNAP there is a file called *config.R* that can be executed typing in a terminal (the current/working directory should be the main directory of SNNAP):

```
R CMD BATCH --vanilla config.R
```

or its content may be copied and pasted into an R console. The script will install the required packages.

7.3 Running the Software

The software has been developed giving to the user the maximum customisation in terms of the approaches that should be run and with which setting of the parameters. Some of the paramaeters are set at the beginning of the source of code of SNNAP, others are thought as parameters to be given in input to the software while executing it. In particular it has been used the package *optparse* [42] to manage those arguments.

To execute the software the user should open a terminal inside the directory *src*; a typical command to execute the script is:

```
Rscript SNNAP.R [options] file.features file.times [optional file
with test instances]
```

e.g.:

```
Rscript SNNAP.R [options] ../data/SAT/HAND.features ../data/SAT/HAND.times
```

For having an overview of the arguments is sufficient to type:

```
Rscript SNNAP.R --help
```

which will return the following list of arguments:

- -r RANKING, –ranking=RANKING : The ranking function to use for applying feature filtering [default chi.squared];
- -c CLUSTERING, –clustering=CLUSTERING : The clustering technique to use [default gmeans];
- -t TIMEOUT, –timeout=TIMEOUT : The timeout used when the dataset has been built [default 5000];
- -n NAME, –name=NAME : The suffix name for output files (output-[name].txt, log-[name].txt, important_features-[name].txt) [default *empty*];
- -s SEED, –seed=SEED : The seed to use [default 12345];

- -k KNN, –knn=KNN : The size of the neighborhood for the k-nn [default 60];
- -d DISTANCE, –distance=DISTANCE : The number of solvers to consider in the evaluation of the distance metric [default 3];
- -g, –giveTesting : If TRUE the script will expect as last parameter the name of a file containing instances to be considered as testing set (no k-fold cross validation);
- -h, –help : Show this help message and exit.

Some arguments are mandatory (like the path to the files with features and times) and others not. A description of every argument is presented here:

- *[Ranking]* Is the ranking function to be used for feature filtering. Should be one of:
  - chi.squared;
  - information.gain;
  - gain.ratio;
  - symmetrical.uncertainty.

  For feature selection, each instance is assigned a class based on the solver that performs best on it. In practice, it is recomended using *chi.squared*;
- *[Clustering]* Is the clustering technique to be used with ISAC. Should be one of:
  - gmeans;
  - XMeans;
  - hclusterpar.

  In practice, it is highly recommended to use *gmeans*;
- *[Timeout]* Is the value of the timeout used when the dataset has been built. This is used when computing the number of instances that timeout, as well as the PAR10 score. Here PAR10 is a penalized average, where each timeout is recorded as having taken 10 times the timeout;
- *[Name]* Is a string that will be part of the files that are created for printing the results of the script. In the folder results/SAT will be created three different files:

1. *output-[Name].txt*;
2. *log-[Name].txt*;
3. *important_features-[Name].txt*.

The three files have, respectively, the output with the results of the script, the log, and the name of the important features as selected by the feature filtering technique applied to SNNAP;

- *[Seed]* The seed to be used;
- *[KNN]* The size of the neighborhood for the SNNAP approach;
- *[Distance]* The number of solvers to be considered in the evaluation of the distance metric by SNNAP;
- *[GiveTesting]* If set to TRUE the script will expect as last parameter the name of a file containing instances to be considered as testing set (no k-fold cross validation);
- *[File.Features]* The path to the file containing the feature values for each instance. Each line of the feature file states the unique name of the instance followed by a list of features that were computed for this instance. This file should be comma delimited and each instance should have the same number of numeric features. For an example, look at the included files in the data folder.

<instance name 1>,	<feature 1>,	<feature 2>,	…	<feature n>
<instance name 2>,	<feature 1>,	<feature 2>,	…	<feature n>
…	…	…	…	…

  The first line of this file should be the header, where the first column is represented as "instance", and the rest are used to define the appropriate features name;

- *[File.Times]* The path to the file containing the running times of the solvers on each instance. Each line of the performance file states the unique name of the instance followed by a list of performances for each solver in the portfolio. This file should be comma delimited and each instance should have the same number of solver evaluations which are in the same order. For an example, look at the included files in the data folder.

<instance name 1>,	<solver 1>,	<solver 2>,	…	<solver n>
<instance name 2>,	<solver 1>,	<solver 2>,	…	<solver n>
…	…	…	…	…

The first line of this file should be the header, where the first column is represented as "instance", and the rest are used to define the appropriate solver;

- *[Optional file with test instances]* this file will be considered only in case the parameter *giveTesting* has been set to TRUE. It lets to specify which instances in the dataset should be considered as unique testing set. All other instances will be considered as training set. The file should be composed of as many rows as the number of instances we want to consider as testing set. Each row should only contain one name of one instance that should be considered in the testing set. For example:

  <instance name 1>
  <instance name 2>
  <instance name 3>
  . . .

7.4 Notes

The parameters regarding the size of the neighborhood and the number of solvers to be considered in SNNAP are heuristically set based on the users experience. However, if (at line 96) the parameter *findOptimalParam* of the file *SNNAP.R* is manually set to TRUE then SNNAP will perform an automatic evaluation of the optimal parameters for the size of the neighborhood and the number of solvers to be considered. For the size of the neighborhood the value considered are those between *minNN* and *maxNN* by steps of 5, while for the number of solvers to use between *minSolv* and *maxSolv* (these four variables can be set to custom values in the file *SNNAP.R*).

The results of this evaluation is a matrix that will be saved in the folder *optimalParam* as a *.rData* file; moreover in the same folder there will also be a 3D surface plot with the results (like the plots in Fig. 6.1). For every combination of size of neighborhood and number of solvers will be saved the average running time of the SNNAP approach.

By default the script performs a n-fold cross validation with $n = 10$, and each fold is spawned as a different thread on the machine. The dataset is randomly split in n parts and every time 1 of these parts is choosen as the testing set while the remaining $n - 1$ are considered as training set. This framework is repeated n times, in a way that each instance is considered exactly once in the testing set. The results are then reported as the overall average runtimes among all the instances solved by the solver chosen by the specific portfolio technique.

7.5 Sample Output

The output of the software on the provided data should look like:

```
[1] "../data/SAT/HAND.features"
[1] "../data/SAT/HAND.times"
[1] "gmeans"
[1] "gain.ratio"
[1] 5000
Seed:  12345
Number of instances:  363
Number of features:  115
Number of solvers:  29
Number of test instances per fold:  36
Number of solvers to compare:  10

                         PAR       AVG  NS  CL NF
virtualBestSolver   401.1489  401.1489   0 0.0 115
FFSNNAP            6735.8460 1036.0395  46 0.0 56.8
SNNAP              6786.9096 1077.1029  46 0.0 115
FFISAC            12239.3633 1601.8979  86 2.7 56.8
ISAC              14722.7830 1858.8718 104 3.6 115
bestSingleSolver  15934.0166 2080.5732 112 0.0 115
```

The first lines give information about some of the parameters used as: the path to the input files, the clustering technique used, the feature filtering ranking function, the time-out value, the seed used and a few properties of the dataset.

The table with results automatically orders the algorithms based on the reported PAR10 score, starting with the lower bound on the achievable performance (*virtualBestSolver*). Moreover are displayed the following statistics[3]:

- *PAR*: is the penalized average of the runtimes: for each instance that is solved within the timeout threshold the actual runtime in seconds denotes the penalty for that instance. For each instance that is not solved within the time limit the penalty is set to 10 times the original timeout;
- *AVG*: the standard average of runtimes;

[3]Note that the number of clusters and the number of features are fractional because the reported results are averaged over the n folds that were run for cross-validation. Moreover for feature filtering in this example was used the approach to select a subset of features which are significantly better than other (so not a fixed number)

- *NS*: the number of not solved instances;
- *CL*: the average number of clusters (only for ISAC);
- *NF*: the number of features used by the approach (so for approaches with feature filtering this number is less than the number of total features available).

Chapter 8

Conclusion

This thesis presents a novel methodology called SNNAP (Solver-based Nearest Neighbor for Algorithm Portfolios) that proposes a new technique to tackle the problem of *algorithm selection*. It is accepted the theory stating that for a given problem it doesn't exist a single solver that is good for all the instances and so techniques to best select the solver to solve a particular instance have been studied. The possibility to develop such a successful approach starts from an in-depth analysis of one of the state-of-the-art software: ISAC (Instance-Specific Algorithm Configuration). ISAC is a successful approach for tuning a wide range of solvers for SAT, MIP, set covering, and others. This approach assumes that the features describing an instance are enough to group instances so that all instances in the cluster prefer the same solver. Yet there is no fundamental reason why this hypothesis should hold.

The basic assumptions behind ISAC have been here analysed and strengthened: a major result has been to be able to show that actually not all the features are always necessary for tackling the problem of algorithm portfolio. Some of them are more important and using just a subset it has been proved that is possible to achieve similar (or slightly better) performances. The same idea of having a ranking among features has been studied approaching the same problem from a different perspective: modifying the distance metric used in conjunction with clustering. Even though it didn't give good results it gave interesting ideas in the possibility of extending the used feature space to help guide the clustering process.

Using predictive models it is shown how is possible to predict with a high accuracy which solvers are going to be the best on each instance. Using this information and creating a custom distance metric Solver-based Nearest Neighbors (SNNAP) has been crafted: using k-nearest neighbors SNNAP is able to assign to previously unseen instances the solver to use and it is able to significantly outperfom an unmodified version of ISAC.

The benefit of the SNNAP approach over ISAC is that the cluster formulated by the k-NN comprises of instances that are more similar to the new instance, something that ISAC assumes but has no way of enforcing. Additionally, a very important point, the new approach is not as sensitive to incorrect decisions by the predictive model. For example, it does not matter if the ranking of the top n solvers is incorrect, any permutation is acceptable. Furthermore, even if one of the solvers is incorrectly included in the top n, the k-NN generates a large enough training set to find a reasonable solver that is likely to work well in general.

This synergy between prediction and clustering that enforces the desired qualities of the clusters is the reason why SNNAP consistently and significantly outperforms the traditional ISAC methodology. Consequently, this thesis presents a solver portfolio for combinatorial problem solving that out-performs the state-of-the-art in the area.

8.1 Future Work

With this work it has been given an initial insight on how SNNAP can improve ISAC and from here many other possibilities can be developed. SNNAP has been proved to be successful only on two datasets and a major work could be done on top of it in order to improve it even more: for sure the releasing of the software under the LGPL licence has been done looking into this direction.

One of the first problem that might be addressed will be: is it possible, and how, to combine SNNAP and tuning? ISAC before addressing the problem of algorithm portfolios has been developed with the explicit intent to be able to automatically tune a software specifically on top of the instances that have to be solved. One possibility has been not only to assign to each cluster one solver, but also a parameterization of it that could lead to the best performances available. SNNAP unfortunately doesn't rely on a static definition of clustering, but uses k-nn: the extension of tuning to this setting is not obvious but, nevertheless, less interesting.

The second thing regards how feature filtering is applied. It has already been stated that feature filtering has to be set as a classification problem: in SNNAP it has been chosen to label each instance with the name of the best solver to solve it. It would be important to understand if this is best way to tackle the problem and see if other ideas might apply better to it. It might be the case in which an instance is labeled with the name of a solver even if the second best solver is slower just by few seconds. Obviously in this setting each one of the label would be correct, so it should be investigated how to behave in such a situation.

Another idea may relate to the problem of *feature learning and generation*: it is generally known how the clusters (in ISAC, for example) should look like: the instances in the same cluster should have similar performances with the same solvers. It might be useful to create a framework that is able, analysing the current clusters, to create some ad-hoc features that are able to modify the space in a way that instances that need to be close each other but are far in the original feature space are grouped together. The idea is similar to the one presented in Chapter 5 when the feature space has been extended including the solvers performances. This time, instead of directly using the performances, the features should be learned automatically basing the creation on the current clusterisation.

In this thesis, moreover, the benchmarks are compared only with the standard version of ISAC; future work should be able to compare them to a newest version of ISAC as well as with SATzilla 2012 and 3S.

In summary, this thesis lays out the groundwork for a novel methodology for addressing the problem of algorithm portfolios: SNNAP. Hopefully further research will expand it and improve it even more.

Bibliography

[1] John R. Rice. “The Algorithm Selection Problem”. In: *Advances in Computers* 15 (1976), pp. 65–118.

[2] Lin Xu et al. “SATzilla: Portfolio-based Algorithm Selection for SAT”. In: *CoRR* (2011).

[3] Bryan Silverthorn and Risto Miikkulainen. “Latent Class Models for Algorithm Portfolio Methods”. In: *AAAI* (2010).

[4] Barry Hurley and Barry O’Sullivan. “Adaptation in a CBR-Based Solver Portfolio for the Satisfiability Problem”. In: *ICCBR* (2012).

[5] Lin Xu et al. *SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models.* SAT Competition. 2012.

[6] Serdar Kadioglu et al. “Algorithm Selection and Scheduling”. In: *CP* (2011), pp. 454–469.

[7] Eoin O’Mahony et al. “Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving”. In: *AICS* (2008).

[8] Lars Kotthoff, Ian Gent, and Ian P. Miguel. “An Evaluation of Machine Learning in Algorithm Selection for Search Problems”. In: *AI Communications* (2012).

[9] Luca Pulina and Armando Tacchella. “A self-adaptive multi-engine solver for quantified Boolean formulas”. In: *Constraints* 14.1 (2009), pp. 80–116.

[10] Serdar Kadioglu et al. “ISAC –Instance-Specific Algorithm Configuration”. In: *ECAI* (2010), pp. 751–756.

[11] Yuri Malitsky and Meinolf Sellmann. “Instance-Specific Algorithm Configuration as a Method for Non-Model-Based Portfolio Generation”. In: *CPAIOR* (2012), pp. 244–259.

[12] *SAT Competitions.* `http://www.satcompetition.org/`.

[13] Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. “Meta-Learning by Landmarking Various Learning Algorithms”. In: *ICML* (2000).

[14] Hilan Bensusan and Christophe Giraud-Carrier. "Casa Batlo is in Passeig de Gracia or landmarking the expertise space". In: *ECML* (2000).

[15] Marco Collautti et al. "SNNAP: Solver-Based Nearest Neighbor for Algorithm Portfolios". In: *ECMLPKDD* (2013), pp. 435–450.

[16] *European Conference on Machine Learning and Principles and practice of Knowledge Discovery in Databases 2013.* `http://www.ecmlpkdd2013.org/`.

[17] *Download SNNAP.* `http://www.dei.unipd.it/~collautt/downloads.html`, `http://4c.ucc.ie/~ymalitsky/downloads.html`.

[18] David H. Wolpert and William G. Macready. "No free lunch theorems for optimization". In: *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION* 1.1 (1997), pp. 67–82.

[19] Carla P. Gomes and Bart Selman. "Algorithm portfolios". In: *Artif. Intell.* 126.1-2 (2001), pp. 43–62.

[20] Kevin Leyton-Brown et al. "Boosting as a Metaphor for Algorithm Design". In: *CP* (2003), pp. 899–903.

[21] Lin Xu et al. "The Design and Analysis of an Algorithm Portfolio for SAT". In: *CP* (2007), pp. 712–727.

[22] *Algorithm Selection literature summary.* `http://4c.ucc.ie/~larsko/assurvey/`.

[23] Lin Xu et al. "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *J. Artif. Intell. Res. (JAIR)* 32 (2008), pp. 565–606.

[24] Yuri Malitsky et al. "Non-Model-Based Algorithm Portfolios for SAT". In: *SAT* (2011), pp. 369–370.

[25] Yuri Malitsky et al. "Parallel SAT Solver Selection and Scheduling". In: *CP* (2012), pp. 512–526.

[26] Lin Xu et al. "Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors". In: *SAT* (2012), pp. 228–241.

[27] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. "Zchaff2004: An Efficient SAT Solver". In: *SAT (Selected Papers* (2004), pp. 360–375.

[28] *MaxSAT Competition.* `http://maxsat.ia.udl.cat/introduction/`.

[29] Lior Rokach and Oded Maimon. "Clustering Methods". In: *The Data Mining and Knowledge Discovery Handbook* (2005), pp. 321–352.

[30] Stuart P. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–136.

[31] Dan Pelleg and Andrew W. Moore. "X-means: Extending K-means with Efficient Estimation of the Number of Clusters". In: *ICML* (2000), pp. 727–734.

[32] Greg Hamerly and Charles Elkan. “Learning the K in K-Means”. In: *NIPS* (2003).

[33] T. W. Anderson and D. A. Darling. “Asymptotic Theory of Certain Goodness of Fit Criteria Based on Stochastic Processes”. In: *Annals of Mathematical Statistics* 23 (1952), pp. 193–212.

[34] J Shlens. “A tutorial on principal component analysis. Version 3.01 Systems Neurobiology Laboratory”. In: *Salk Institute for Biological Studies on line: http://www. snl. salk. edu/˜ shlens* (2009).

[35] Christian Kroer and Yuri Malitsky. “Feature Filtering for Instance-Specific Algorithm Configuration”. In: *ICTAI* (2011), pp. 849–855.

[36] Wlodzislaw Duch and Google Duch. “Filter Methods”. In: *Feature extraction, foundations and applications* (2004), pp. 89–118.

[37] *FSelector R package*. `http://cran.r-project.org/web/packages/FSelector/index.html`.

[38] Eric I. Hsu et al. “Probabilistically Estimating Backbones and Variable Bias: Experimental Overview”. In: *CP* (2008), pp. 613–617.

[39] Leo Breiman. “Random Forests”. In: *Machine Learning* (2001), pp. 5–32.

[40] *RWeka R package*. `http://cran.r-project.org/web/packages/RWeka/index.html`.

[41] *Amap R package*. `http://cran.r-project.org/web/packages/amap/index.html`.

[42] *OptParse R package*. `http://cran.r-project.org/web/packages/optparse/index.html`.