Karlsruhe Institute of Technology

Predicting Hardness and Runtime of SAT Instances

Master's Thesis of

Sajjad Ahmad

at the Department of Informatics
Institute for Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. Peter Sanders
Second reviewer:	Prof. Dr. Carsten Sinz
Advisor:	Dr. Markus Iser
Second advisor:	Dr. Tomáš Balyo

01. May 2022 – 31. October 2022

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . . . .

(Sajjad Ahmad)

Abstract

The Boolean satisfiability problem (SAT) is a classic NP-complete problem, whose instances can take anywhere from a fraction of a second to potentially many years to solve. While there are some contributions in the field of runtime and hardness prediction of SAT instances, these mostly focus on single solvers with few instance families. Moreover, these works neglect the time consumption of the pipeline. This leads to limited generalizability and usability.

In this work, we present a new practical set of features for predicting instance hardness and runtime. We also present ways to minimize the time consumption. In this context, we linked, among other contributions, the number of connected components to the instance family and the instance runtime.

To ensure high generalizability, we trained our models on a dataset with a very large number of instance families and with labels created from the runtimes of many different solvers. The results were highly competitive and scalable. We verified them using the established methods of k-fold cross-validation and hold-out sets.

Zusammenfassung

Das Boolesche Erfüllbarkeitsproblem (SAT) ist ein klassisches NP-vollständiges Problem, dessen Instanzen zur Lösung nur den Bruchteil einer Sekunde, bis hin zu potenziell vielen Jahren benötigen können. Es gibt zwar einige Beiträge auf dem Gebiet der Laufzeit- und Härtevorhersage von SAT Instanzen, diese konzentrieren sich jedoch meist auf einzelne Solver mit wenigen Instanzfamilien. Außerdem vernachlässigen diese Arbeiten den Zeitaufwand der Pipeline. Dies führt zu einer begrenzten Verallgemeinerbarkeit und Anwendbarkeit.

In dieser Arbeit stellen wir einen neuen praxisorientierten Satz von Merkmalen zur Vorhersage der Instanzhärte und Laufzeit vor. Wir stellen zudem Möglichkeiten zur Minimierung des Zeitaufwands vor. In diesem Zusammenhang haben wir unter anderem die Anzahl der verbundenen Komponenten mit der Instanzfamilie und der Instanzlaufzeit in Verbindung gebracht.

Um eine hohe Generalisierbarkeit zu gewährleisten, haben wir unsere Modelle auf einem Datensatz mit einer sehr großen Anzahl von Instanzfamilien und mit Labels trainiert, die aus den Laufzeiten vieler verschiedener Solver erstellt wurden. Die Ergebnisse waren äußerst konkurrenzfähig und skalierbar. Wir haben sie mit den etablierten Methoden der k-fachen Kreuzvalidierung und Hold-out Sets verifiziert.

Contents

Abstract i

Zusammenfassung ii

1 Introduction 1

2 Preliminaries and Related Work 3
2.1 Satisfiability Problem (SAT) . . . 3
2.2 DIMACS CNF Format . . . 3
2.3 Within-Cluster Sum of Squares (WCSS) . . . 4
2.4 K-fold Cross-Validation (CV) . . . 4
2.4.1 Comparison of Traditional Data Split to K-fold Cross-Validation . . . 4
2.5 Variable Incidence Graph (VIG) . . . 6
2.6 Graph Features . . . 6
2.6.1 Centrality . . . 6
2.6.2 Community . . . 7
2.6.3 Small-World Property . . . 7
2.6.4 Clustering . . . 8
2.6.5 Other Graph Features . . . 9
2.7 SAT Features in Related Work . . . 9
2.8 Related Work . . . 10

3 Implementation 12
3.1 Technology Stack . . . 12
3.2 Execution Pipeline . . . 12
3.2.1 Reading CNF Files . . . 12
3.2.2 Feature Extraction . . . 13
3.2.3 Instance Labelling . . . 14
3.2.4 Model . . . 15
3.2.5 Hyperparameter Optimization . . . 16

4 Evaluation 17
4.1 Experimental Setup . . . 17
4.2 Feature Extraction . . . 17
4.3 Family Classification . . . 19
4.4 Runtime Prediction . . . 24
4.5 Categorical Hardness Prediction . . . 28
4.6 Hyperparameter Optimization . . . 31

4.7 Feature Extraction Costs . . . 34
4.8 Pipeline Adjustments . . . 35
4.8.1 Minimal Pipeline Runtime . . . 35
4.8.2 Saving And Loading VIGs . . . 42
4.9 Performance Scaling with Data . . . 45

5 Discussion and Future Works **48**
5.1 Used Features . . . 48
5.2 Created Labels . . . 48
5.3 Model Comparison . . . 49
5.4 Most Important Features . . . 50
5.5 Pipeline Adjustments . . . 51
5.5.1 Minimal Time Overhead . . . 51
5.5.2 Intermediate Pipeline Step . . . 51
5.6 Our Results Compared to Current Research . . . 52
5.7 Model in Practice . . . 53

6 Conclusion **54**

Bibliography **55**

List of Figures

2.1 Visual representation of k-fold CV with hold-out test set. 5
2.2 Visual representation of k-fold CV without hold-out test set. 5
2.3 Graph with colour-coded centrality values. 7
2.4 Graph with three communities. 8

3.1 Our implemented execution pipeline. 13
3.2 Overview of our used labels. 15

4.1 Graph based SAT instance feature set. Booleans are marked with *. 20
4.2 Distribution of family occurrences. 21
4.3 WCSS graph for the Elbow method 29
4.4 Scatter plots of KNN clusterings over PAR2 runtime with varying k. 30
4.5 Mean extraction times of features. 35
4.6 Permutation feature importance on family classification. 37
4.7 Tree based feature importance on family classification. 38
4.8 Dendrogram of hierarchical clustering based on Spearman rank-order. 38
4.9 Heatmap of correlated features. 39
4.10 List of compared graph file formats. 43
4.11 Box plot diagrams to compare the file sizes of different graph formats. 44
4.12 Box plot diagrams to compare the writing times of different graph formats. . . 44

List of Tables

4.1 List of all 18 solvers whose runtimes have been used. 18
4.2 The four most frequent families in the data set used. 21
4.3 Family classification result of different models with varying labels. 22
4.4 Performance comparison of a base Random Forest in family classification. . . 24
4.5 Performance comparison of the most promising models on the PAR2 label. . . 26
4.6 Performance comparison of the most promising models on the log10_PAR2 label. 26
4.7 Runtime regression performance of the AdaBoost and Decision Tree models. . 27
4.8 Base Random Forest performance with, and without, feature scaling. 28
4.9 Performance comparison of the most promising models on the 3-means label. 30
4.10 Hyperparameter optimization results for the MLP. 32
4.11 Hyperparameter optimization results for the SVM. 32
4.12 Hyperparameter optimization results for the RF. 33
4.13 Comparison between base models and optimized hyperparameters. 34
4.14 Cost categories with corresponding features. 36
4.15 Features with their average extraction time and their corresponding cost category. 37
4.16 Hierarchical clustering thresholds and features. 40
4.17 Comparison of model performances when training over a subset of features. . 41
4.18 Classification and regression comparisons for most important features. 41
4.19 Comparison of file size consumption of CNF and GraphML format. 43
4.20 Comparison of the four promising graph formats based on multiple metrics. . 43
4.21 Comparison of scores over small (old) and large (new) data set. 46
4.22 Comparison of the family label for the old and new data set. 46
4.23 Comparison of the log10_PAR2 label for the old and new data set. 47
4.24 Overview of performance results on larger data set. 47

List of Abbreviations

SAT	Boolean Satisfiability Problem
CNF	Conjunctive Normal Form
CV	Cross-Validation
VIG	Variable Incidence Graph
WCSS	Within-Cluster-Sum-of-Squares
P.P.	Percentage Point
PAR2	Penalized Average Runtime With Factor 2
RMSE	Root Mean Squared Error
RF	Random Forest
MLP	Multilayer-Perceptron
SVM	Support Vector Machine
KNN	K-Nearest-Neighbor

1 Introduction

The Boolean satisfiability problem, or SAT problem, is to determine whether a variable assignment exists such that the formula evaluates to true. This is a classic NP-complete problem whose instances can have an exponential runtime and are often considered practically unsolvable [1]. By using assumptions, heuristics and especially by recognizing and utilizing underlying structures, many solvers today can solve numerous instances efficiently and quickly. At the same time, problems of many different areas have been successfully defined and solved as SAT problems, e.g. hardware verification [2], test generation, verification, and optimization [3, 4, 5], and many more [6].

Nowadays, solvers often need only a few seconds to solve an instance. This enables the use of SAT solvers in real-time or time-sensitive applications, such as interactive configurators [7]. But even with the most efficient algorithm and solver, unfavourable instances can lead to long or even exponential runtime, which in turn cause a timeout or errors. As a result, users are frustrated and lose precious time.

However, if the application knew in advance how long it would take to solve a task or how difficult it would be, it could react accordingly and avoid possible timeouts and errors. This would lead to higher user satisfaction, productivity and consequently higher revenues. For these reasons, we address the following questions in this work:

How can we predict the runtime of a SAT instance? How can this be done with as little time overhead as possible? What are sensible labels for instance hardness? And lastly, how can we predict this instance hardness?

Although there have been previous contributions in the area of runtime and hardness prediction of SAT instances, these have mostly focused on single solvers with few instance families [8, 9]. Moreover, the time consumption of the pipeline from preprocessing and feature extraction to inference is neglected in these works. This results in limited generalizability and practicability.

In contrast, we employ data sets of 88 and 130 different instance families, as well as runtimes of 18 and 28 different solvers to ensure high diversity and generalizability. We also investigate the minimal pipeline overhead for a performant predictor.

This work is structured as follows: In Chapter 2, we will first outline the most important concepts necessary to understand this work. In this chapter, we will also place our work in the current research and explain our selection of features based on it. In Chapter 3 we will describe our execution pipeline and evaluate its results in Chapter 4. Finally, we will discuss the results and highlight areas with potential for future work in Chapter 5.

We have made four important contributions in this work: First, we have created runtime and hardness labels for SAT instances that are independent of the solver. Second, we have successfully used machine learning models for performant runtime regression and hardness classification on inhomogeneous and difficult data sets. Furthermore, we have determined the most important features as well as how they can be used to minimize the time consumption of the pipeline for practical use. Finally, we have shown that our model scales exceptionally well with the amount of data used.

2 Preliminaries and Related Work

In this chapter, we will briefly outline the main concepts needed to understand the later chapters of our work. We will also briefly explain the concepts behind the features we have chosen and explain our feature choice based on current research.

2.1 Satisfiability Problem (SAT)

A *literal* is either x or its negation $\overline{x}$ for a boolean variable x. A *clause* is the disjunction ("or", noted as $\vee$) of literals. A propositional formula is in the *conjunctive normal form* (CNF), if it is a conjunction ("and", noted as $\wedge$) of clauses. The *boolean satisfiability problem* (SAT) consists of determining an assignment to the boolean variables such that the CNF formula evaluates to true. An example CNF looks as follows:

$$(\overline{x}_1 \vee x_2 \vee \overline{x}_3) \\ \wedge (\overline{x}_2 \vee \overline{x}_4) \\ \wedge (x_2 \vee x_3 \vee \overline{x}_4)$$

2.2 DIMACS CNF Format

The DIMACS CNF format is the textual representation of a SAT formula in conjunctive normal form. While there are many extensions and variations of the DIMACS CNF format, we only used the basic format. The general structure of any DIMACS CNF text file is as follows:

Lines beginning with the character *c* are comments that are not part of the formula itself and are used only to add additional information and to facilitate the human reader's understanding. Each DIMACS CNF file starts with a header line of the form *p cnf #variables #clauses*, where *#variables* and *#clauses* are replaced by the number of variables and the number of clauses in the formula, respectively. The header line is then followed by the clauses of the formula. Clauses are represented as decimal numbers separated by spaces and new lines. Each decimal number stands for a literal, with negative numbers denoting the negated literal. At the end of each clause belongs a *0* as an end-of-line marker. The example formula $(x \vee y \vee \overline{z}) \wedge (\overline{y} \vee z)$ could be encoded as follows:

p cnf 3 2
1 2 -3 0
-2 3 0

2.3 Within-Cluster Sum of Squares (WCSS)

The Within-Cluster Sum of Squares (WCSS) is used as a metric in clustering algorithms, e.g. K-Nearest-Neighbor (KNN). Nair defined it as "the sum of squares of the distances of each data point in all clusters to their respective centroids" [10, p. 1]:

Definition 2.1 - Within-Cluster Sum of Squares (WCSS): Given cluster centroids C and data points d, $WCSS$ is given by:

$$WCSS = \sum_{C_k}^{C_n} (\sum_{d_i \text{ in } C_i}^{d_m} distance(d_i, C_k)^2) \tag{2.1}$$

2.4 K-fold Cross-Validation (CV)

K-fold cross-validation (CV) is a statistical method used to evaluate machine learning models. The general procedure begins by randomly shuffling the data. If hyperparameter optimization of the model is desired, the data is then split into a training-and-validation set and a hold-out test set. In this case, the ratio of the split is usually 70-80% to 20-30% for the training-and-validation set and test set, respectively. Subsequently, the test set is put aside and not touched until the final model evaluation.

For the hyperparameter optimization, the (training-and-validation) data set is divided into k groups of equal size, called "folds". Each fold is once used as a validation set, while the other k-1 folds are used as the training data set. The evaluation results are retained while the model is discarded. Afterwards, the performance of the model is evaluated based on the k scores. These results are usually summarized using the average and a variance measure, such as standard deviation or standard error. The hyperparameters of the model with the best performance over the validation set are retained, and the model is fitted over the whole training-and-validation set. Finally, the overall performance of the model is evaluated by inference over the unused test set. The resulting score is representative of the model's performance on new, unseen data. Figure 2.1 shows the k-fold cross validation for k = 5, if hyperparameter optimization is desired and therefore a validation set is required.

If no hyperparameter optimization is desired, no validation set is needed, and we do not need to split off a hold-out test set. Instead, the entire data set is divided into k folds. Then, each individual fold is used once as a test set, while the remaining k-1 folds are used as a training set to fit the model. We again get an array of performance scores that can be summarized in the same way. Figure 2.2 shows the k-fold cross-validation for k = 10, if no hyperparameter optimization is desired.

2.4.1 Comparison of Traditional Data Split to K-fold Cross-Validation

A traditional training-test or training-validation-test split does not use all data samples to fit the model. Instead, only those that are included in the training set are used to fit the model. Since most statistical and machine learning models perform worse when trained on fewer samples, this leads to an overestimated error rate for a model fitted to the entire data set. Moreover, the estimated error rate can be highly variable, depending on exactly which

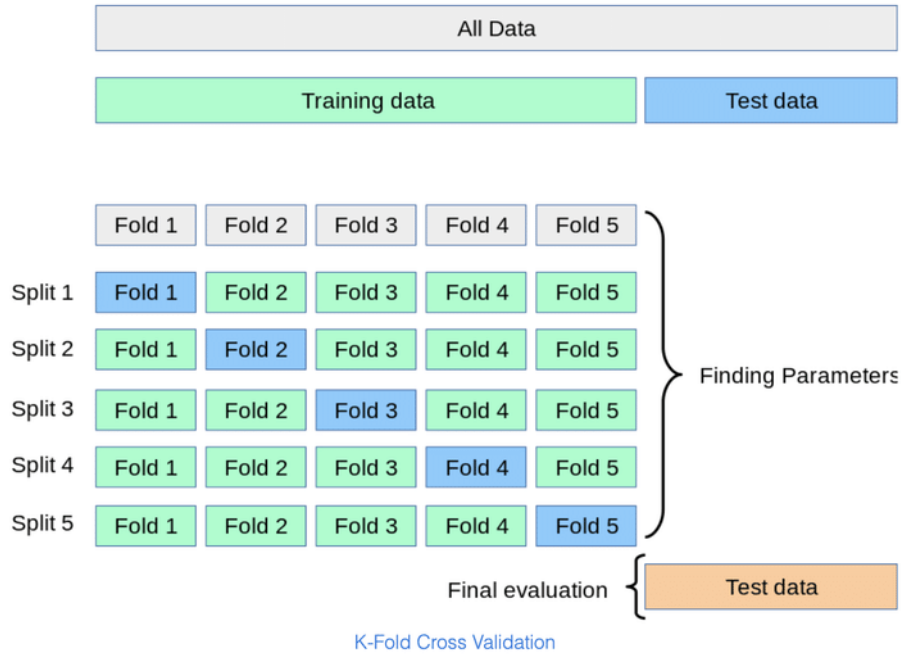


Figure 2.1: A data set (grey) is split into a training data set (green) and a test set (blue). The training data set is then further split into training (green) and validation (blue) folds according to the k-fold cross-validation method for k = 5. Figure taken from [11].

Figure 2.2: A data set (here called “Training set”) is split into training (grey) and test (blue) folds according to the k-fold cross-validation method for k = 10. Afterwards, the performance scores are averaged. Figure taken from [12].

observations are split into which sets. In particular, since the validity of the accuracy depends on the representativeness of the validation or test set for the whole data set. These concerns are greatly exacerbated when dealing with smaller data sets.

In contrast, with cross-validation, each sample in the data set is used once to test or validate the model and $k - 1$ times to train the model. Since all data samples are used to test or validate the inference, there is no issue of representativeness. In addition, as k increases, the difference in size between the training and the whole data set becomes smaller, which reduces the bias of this statistical evaluation method. However, as k increases, so does the computational cost, as more iterations must be performed. In addition, the difference in size between the training and validation or test data sets becomes larger, resulting in higher variance. This problem is slightly counteracted by averaging the score over k iterations. To meet the overall trade-off between variance and bias, k-fold cross-validation is usually performed with k = 10, as this value has been empirically shown to produce error rate estimates that have neither an unreasonably high bias nor a very high variance [13].

In summary, k-fold cross-validation is a powerful statistical method for measuring the performance of a model that addresses the problems of variance, bias, and representativeness of measurements, especially for smaller data sets. This method has consistently demonstrated that it estimates actual scores fairly accurately across a wide range of scenarios.

2.5 Variable Incidence Graph (VIG)

Using a CNF formula, we can create a weighted undirected graph by first creating a graph node for each variable in the formula. Then we connect two nodes, if the corresponding variables appear together in a clause. The weight of each edge is defined by the following equation:

Given a CNF formula ϕ with variables X and clauses C, and an undirected weighted graph $G(V, E)$ with a set of vertices V and edges E, we can define a weight function $w : V \times V \rightarrow \mathfrak{R}^+ \cup \{0\}$ that satisfies $w(x, y) = w(y, x)$. Given an undirected edge $\{x_i, x_j\}$, its weight is defined as follows [14]:

$$w(\{x_i, x_j\}) = \sum_{c \epsilon C \wedge x_i, x_j \epsilon c} \frac{1}{\binom{|c|}{2}} \tag{2.2}$$

2.6 Graph Features

In this section, we will briefly outline the most important concepts behind our features. Our full list of features can be seen in Chapter 4.2 in Figure 4.1. In our work, we first transformed all our CNFs into VIGs and then extracted different graph properties as features. Hence, all our features are graph and network based.

2.6.1 Centrality

Centrality indicates how important a node is in a graph [15]. There are several ways to define central nodes, including degree centrality [15] and betweenness centrality [16]. Saxena et al. have a recent survey on the different centrality measures used in complex networks[17]. Figure

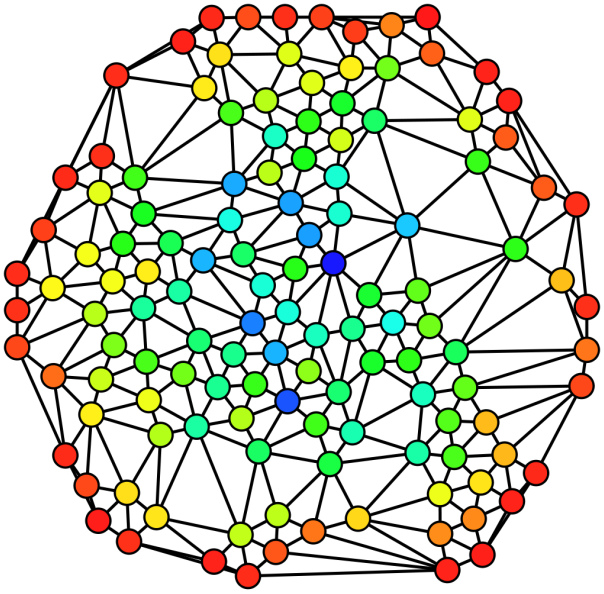


Figure 2.3: Figure from [18]. An undirected graph coloured based on betweenness centrality. Red nodes have the smallest centrality value, while blue have the largest.

2.3 from [18] shows an undirected graph coloured based on the betweenness centrality. Red nodes have the smallest centrality value, while blue ones have the greatest.

2.6.2 Community

A *community* in a graph is a subgraph that has more internal edges than outgoing edges to the rest of the graph. To detect communities, a quality metric Q called modularity is used [19, 20]. A graph with high modularity is easily separable into smaller communities, while a graph with low modularity is not. In other words, high modularity implies dense connections between nodes within a subgraph, with few inter-community edges. Figure 2.4 taken from [21] illustrates this concept.

2.6.3 Small-World Property

To define the small-world property, we first need to define the characteristic path length, the clustering coefficient, and the proximity ratio:

Definition 2.3 - Characteristic Path Length (CPL): For any graph, CPL is given by:

$$CPL = \frac{1}{|V| * (|V| - 1)} \sum_{v \in V} \sum_{v' \in V \setminus \{v\}} spl(v, v') \tag{2.3}$$

where $spl(v, v')$ returns the number of edges in the shortest path between nodes v and v' [22].

Definition 2.4 - Global Clustering Coefficient (CLC): For any graph, CLC is given by:

$$CLC = \frac{\text{number of closed triplets}}{\text{number of all triplets}} \tag{2.4}$$

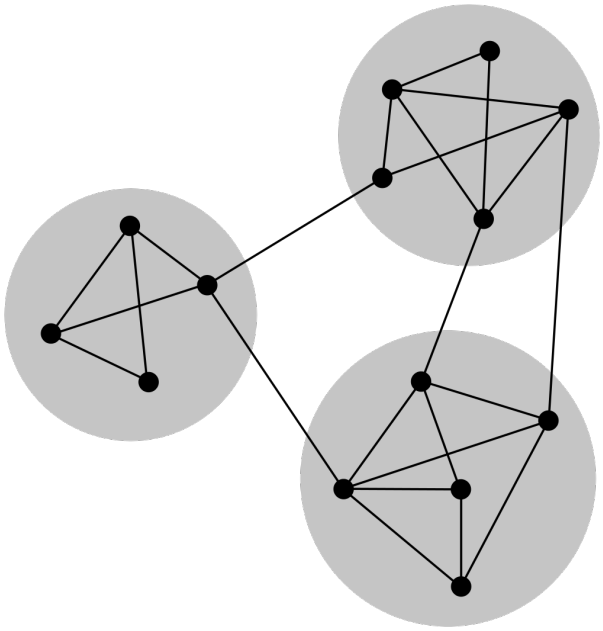


Figure 2.4: Graph with three communities with sparse inter-community edges. Figure from [21].

The proximity ratio is then defined as follows:
Definition 2.5 - Proximity Ratio (Pr) [23, 24]: For any graph, Pr is given by:

$$Pr = \frac{CLC \times CPL_{rand}}{CPL \times CLC_{rand}} \tag{2.5}$$

where CLC is the global clustering coefficient 2.4 and CPL is the characteristic path length 2.3 of a given graph, and CLC_{rand} and CPL_{rand} are the CLC and CPL of a random graph with the same number of nodes and edges, respectively.

Finally, a graph is a small-world, if it has a proximity ratio of $\gg 1$. A practical example of this characteristic is the phenomenon that shortly after meeting a stranger, you often realize that you have a mutual friend.

2.6.4 Clustering

We also included promising clustering measures and properties. One of the most well-known is the *clustering coefficient*, defined in Equation 2.4, which measures the extent to which nodes in a graph tend to cluster together. Our feature measures the clustering coefficient globally. For instance, if all nodes are connected to each other, we have a global clustering coefficient of 1. If the graph has no edges, we have a global clustering coefficient of 0.

In a community, internal edges can be distributed around nodes in different ways, either concentrated around a few highly centralized nodes or evenly distributed across each node. The *hub dominance* metric is used to determine the degree of centralized organization around well-connected nodes. The higher this metric in a community is, the more likely it is to have a hub-like structure [25].

We also included boolean features that check the graph for *proper clustering, one clustering* and *singleton clustering*. Finally, we also added a feature that calculated the *global intra-cluster density* over our graph. The global intra-cluster density is defined as the sum of all existing intra-cluster edges divided by the sum of all possible intra-cluster edges.

2.6.5 Other Graph Features

To include the problem size in our features, we added the number of edges and nodes. A few more promising features that we included were the *number of connected components*, the *degree assortativity coefficient*, as well as the *edge cut value* and *edge cut fraction*.

A connected component of an undirected graph is a subgraph in which each pair of nodes is connected by a path. Moreover, the subgraph must not be part of a larger connected subgraph. In other words, a connected component is the largest subgraph, where for each node, there exists a path to all other nodes.

The degree assortativity coefficient refers to the preference of nodes to connect to similar nodes with respect to node degree over dissimilar nodes [26]. Finally, the edge cut value refers to the total weight of inter-community edges, while the edge cut fraction is the edge cut value divided by the total number of edges.

2.7 SAT Features in Related Work

In this subsection, we will elaborate on our feature choice based on related work.

Our first feature focus was on *centrality*. In graph theory, centrality refers to how important a node is in a graph [15]. In this context, Kambhampati and Liu [27] have shown that the centrality measure can be used to distinguish between random and industrial instances. Their results indicate that in uniform random instances, variables have similar importance, while in industrial instances, some instances are significantly more important than others. Moreover, Katsirelos and Simon [28] linked centrality to the behaviour of CDCL solvers and showed that the centrality measure even distinguishes industrial benchmarks from the same family. For these reasons, we included several centrality variations as features, among them degree centrality [15] and betweenness centrality [16].

Then, we focused on *community* features. A *community* in a graph is a subgraph that has more internal edges than outgoing edges to the rest of the graph, and the qualitative measure *modularity* is used to detect communities in graphs [19, 20]. Intuitively, a graph with high modularity implies dense connections between nodes within a subgraph (a community), with few inter-community edges (compare Figure 2.4). Current research shows that community structures are not, or only weakly, evident in random instances while being clearly evident in industrial and crafted instances [29, 30]. Therefore, modularity can be used to differentiate between instance classes as well. Moreover, multiple works have shown a correlation between the modularity value and instance runtime [31, 8]. For these reasons, we included community measures as well as the modularity value in our features.

After centrality and community measures, we considered the *small-world property*, which is found in many real-world networks [32]. The phenomenon that soon after meeting a stranger, you frequently discover that you share a common friend is a practical example of this property.

Already in 1999, Walsh et al. [23] found many industrial instances to have a high proximity ratio and therefore have the small world property. In 2013, Khambhampati and Liu's research [27] showed that uniform random distributions have a proximity ratio of 1 (which follows by definition), while industrial distributions have a very high proximity ratio. Therefore, the small-world property differentiates between instance classes as well. However, since the time required to determine whether a (large) graph is a small-world graph was too high, we decided to use a large set of different clustering properties and coefficients instead, including the global clustering coefficient and the cluster imbalance value.

Finally, we added some general graph properties, such as the number of connected components or the degree assortativity coefficient, as these looked promising and had not been studied in detail in this context before. For a full list of our features, compare Figure 4.1.

2.8 Related Work

In this section, we will briefly outline the current state of research and place our work in current literature.

The structure of SAT instances has been analysed for many years now. Especially in the last decade, a large portion of research has been dedicated to understanding and measuring the underlying difference in structure of random, crafted, and industrial SAT instances. Most of this work explores the evidence for structural measures in SAT instances and their impact on SAT solver performance. Alyahya et al. have compiled a comprehensive overview on this matter [14].

As current research shows, there is no best solver for all instances and that different solvers perform best on different instance families [33]. For instance, Kilby et al. have shown that conflict-driven clause learning (CDCL) solvers cannot solve random 3-SAT instances with only a few instances, but are able to solve industrial instances with thousands of variables [34]. This in turn suggests that industrial instances must have structural differences compared to random instances.

Since then, many strategies have been employed to study, measure and understand the structural differences, with a recent paper creating a thorough overview of what we know about structural measures in the SAT domain [14]. Understanding the structure, and being able to predict which family an instance belongs to, allows the creation of better and faster solvers. For example, solvers that utilize the underlying structure for a faster solution [34] or so-called portfolio solvers that try to pick the best possible solver for each instance [33].

In order to predict the correct instance family, a majority of studies compute the structural measures for each SAT instance and then calculate and compare the averages and standard deviations. Similar values within a family then suggest structural similarity. A more recent strategy is to train machine learning models using these structural measures as features and SAT families or runtimes as labels [9, 35, 8]. A high performance score on this model indicates that there is indeed a correlation between the labels and features used [31, 14]. Consequently, the structural measures can be considered predictors of instance family and runtime.

While the previously mentioned works show very strong performance, the results have limited generalizability and practicability. On the one hand, the overall time consumption is neglected, which limits the use of their pipeline for time-critical or real-time applications. On

the other hand, a highly curated and homogeneous data set is combined with the runtime of only one or a few performant solvers, leading to poor generalizability for real-world instances with a high degree of variation.

In this work, we used a novel feature set to train a practice-oriented machine learning model for runtime and hardness prediction of SAT instances. To deal with the instance variability of the real world, we used data sets with instances from many different SAT families. Known as one of the hardest instance families in the SAT domain, Cryptography is the largest instance family in our data set. This allows our model to handle difficult instances as well. To ensure that the prediction is solver-independent and has a high degree of generalizability, we averaged the runtime labels across numerous different solvers. We also explored how to minimize the pipeline runtime by omitting the popular pre-processing step and reducing the number of features extracted, while retaining performance as much as possible. This allows our model to strike a balance between performance and time consumption, which is necessary for time-critical and real-time applications. Finally, since many applications do not require accurate runtime prediction, but would rather benefit from a simple categorical label (such as “easy” or “hard”), we also addressed the generation and prediction of applicable hardness labels.

3 Implementation

In this chapter, we will discuss our used pipeline and its most important implementation details. The implemented code can be found at https://github.com/Avrodos/SATRuntimePrediction.

3.1 Technology Stack

In this section, we will elaborate on our used hardware and technology stack. All runtime data was collected on a server-cluster, where each node is equipped with 2 Intel Xeon E5430 CPUs running at 2.66 GHz and 32 GB of RAM. All heavy computing, such as the graph generation or feature extraction, was executed in parallel on a server-cluster, where each node is equipped with 4 Intel Xeon E5-4640 CPUs running at 2.4 GHz with 512 GiB ECC RAM.

All processing steps were implemented as Python scripts, using Python 3.10, if not mentioned otherwise. For feature extraction, we used the NetworKit and NetworkX libraries. They are open-source libraries that provide flexible data structures (e.g. graphs), and a wide range of highly efficient algorithms for analysing networks [36, 37]. For efficient and fast parallel computing, we implemented all data structures and calculations using the Pandas and NumPy libraries [38, 39]. For visualization, e.g. plot generation, we used the established Matplotlib library [40]. Finally, for our models we used the Scikit-learn library, as it is a popular open source machine learning library built on NumPy, SciPy and Matplotlib [41].

3.2 Execution Pipeline

In this section, we will briefly outline the most important steps of our execution pipeline. The execution pipeline is shown in Figure 3.1. In short, we first read CNF files stored in DIMACS CNF format. Then we generated VIGs based on the process explained in Section 2.5. Subsequently, we looped selecting, extracting, post-processing and finally evaluating graph-based features. This loop was repeated until we had a set of usable features. In addition, we had to create suitable labels for our instances. Once we had suitable labels and extracted features, we selected, trained and evaluated different machine learning models. Finally, we compared them and determined the most suitable model.

3.2.1 Reading CNF Files

All our CNF's were saved in the DIMACS CNF file format [42]. The DIMACS CNF file format allows the textual representation of a SAT formula in conjunctive normal form. There are many variations and extensions of the DIMACS CNF file format, however, we only focused on the basic format. The general structure of any DIMACS CNF text file can be seen in Section

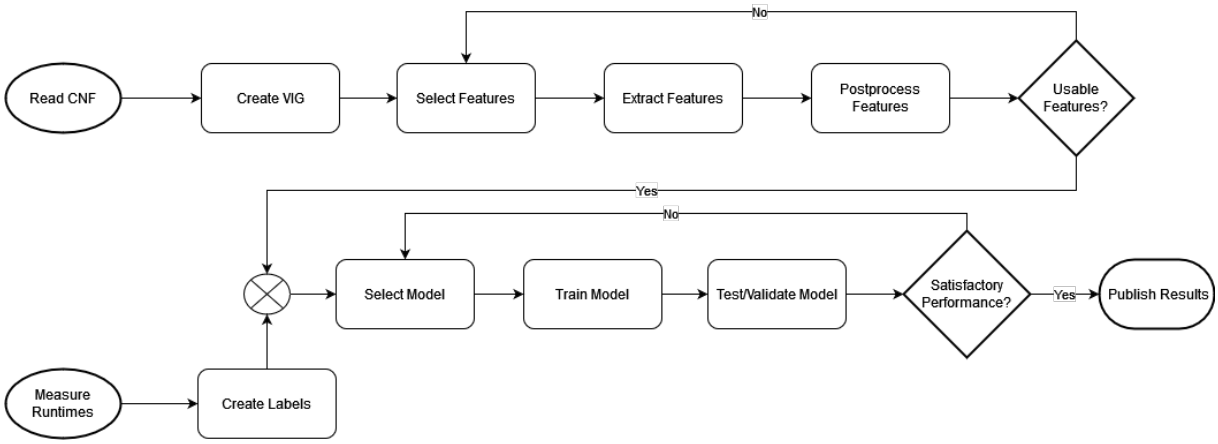


Figure 3.1: Our implemented execution pipeline.

2.2. In short, DIMACS CNF file format makes the reading process of CNF's very intuitive and straight forward. We used a Python script that reads the CNF in the mentioned DIMACS CNF file format and creates a VIG based on the Formula 2.2 in parallel.

3.2.2 Feature Extraction

In this subsection, we will briefly outline the loop in our pipeline that we used to determine our final set of features. After creating the VIGs in memory, we selected a large set of features, based on current research and promising properties (compare Section 2.7). We then extracted the selected features in parallel on a server-cluster. We also added a timeout of 60 minutes to each instance, as we believed this to be a reasonable cut-off for the feature extraction step for most use cases. Simultaneously, we measured the extraction time for all features, which we used to evaluate the average cost of an extracted feature (compare Section 4.7).

It's important to note that some features were extracted as a definite number over the entire graph (e.g., the number of nodes), while other features were extracted as a discrete distribution (e.g., the centrality of each node), which was represented as a NumPy Array. All of these "distribution features" are represented by five metrics measured over this same distribution: minimum, maximum, arithmetic mean, variance, and entropy (see Figure 4.1 in the Evaluation chapter). While this results in an extracted feature being represented by five features in the feature set, the corresponding time measurement only measures the time required to compute the distribution. We consider this sensible since the time required to compute these highly optimized operations, over a NumPy array, a highly optimized data structure, is negligible.

If the extraction of a feature failed, for example, because it consumed too much memory or timed out, it is recorded as "NaN" (not a number). During the feature post-processing, all instances with NaNs are removed. In addition, any necessary scaling and normalization of features (e.g. for Multilayer Perceptrons) is performed in this step as well. Subsequently, all features that took an unreasonable amount of time or had little to no variance were removed, and new features were selected and added to the feature set. This loop continued until we had our novel feature set of 46 features, which we used for the first data set (see Figure 4.1).

After thorough evaluation of the first pipeline iteration, we wanted to test the scalability of our model on a much larger data set (compare Subsection 4.9). We therefore removed the five features belonging to the eigenvector centrality. These features were outliers regarding extraction time, with an average extraction time of over 3000s, compared to the next largest extraction time of under 160s (compare Figure 4.5). And their relative importance was not high enough for us to justify this time overhead. For more details, compare Section 4.7.

3.2.3 Instance Labelling

In this subsection, we will briefly outline the steps we took to calculate our labels. Before we can use our extracted features to train a machine learning model, we first have to identify the ground truth for all instances, i.e., the labels. Since the family classification labels were already part of the instance metadata, no calculations were required for them.

To ensure high generalizability and robustness of our runtime prediction model, we measured the runtime of numerous different solvers for each instance. Based on these measurements, we now needed a number or string that preserved information and was as representative as possible of the expected instance hardness. As we defined a timeout of 5000s for all solvers, we only have a lower bound for the actual solution time, which means that the label must also account for the downward bias.

To address these issues, we used the Penalized Average Runtime with factor 2 (PAR2). PAR2 penalizes all timeout values by multiplying them with two, i.e. doubling them, while keeping the original values of the non-timeout instances. This way, the downward bias is compensated while preserving as much information from the runtimes as possible. For our label, we calculated the PAR2 value for each of the solver runtimes and averaged the result. This results in an excellent solver-independent label for the expected runtime as well as the hardness of an instance. As some models handle logarithmic labels better, we also calculated and tested the logarithm to the base of 10 over the calculated PAR2 label. For the results and evaluation, see Section 4.4.

Not all applications require or desire a specific runtime as a prediction. A categorical estimation of the instance, e.g. as “easy” or “hard”, can often be sufficient. The application can then, for example, omit difficult instances or allocate more resources to solve them. To create the categorical labels, we took the previously computed PAR2 labels and divided them into k clusters using the KNN algorithm. To determine the right number of clusters, we decided to use the Elbow method over the Within-Cluster Sum of Squares (WCSS) defined by Equation 2.1. The goal was to minimize this sum by choosing the right number of clusters. Choosing $k = n$ would always result in an ideal WCSS of 0. However, this would yield no benefit since we would have as many clusters as data points. Therefore, we had to find a tradeoff between WCSS and the number of clusters, which we can find using the Elbow method [43]. To this end, we first created a graph relating the WCSS to the number of clusters used. This graph had a threshold above which the WCSS decreased minimally with the addition of clusters, the so-called “elbow of the graph”— which is the optimal number of clusters according to the Elbow method. We then executed the KNN algorithm for our determined k and labelled the instances according to the result. The ideal k and the resulting clustering can be seen in Section 4.5.

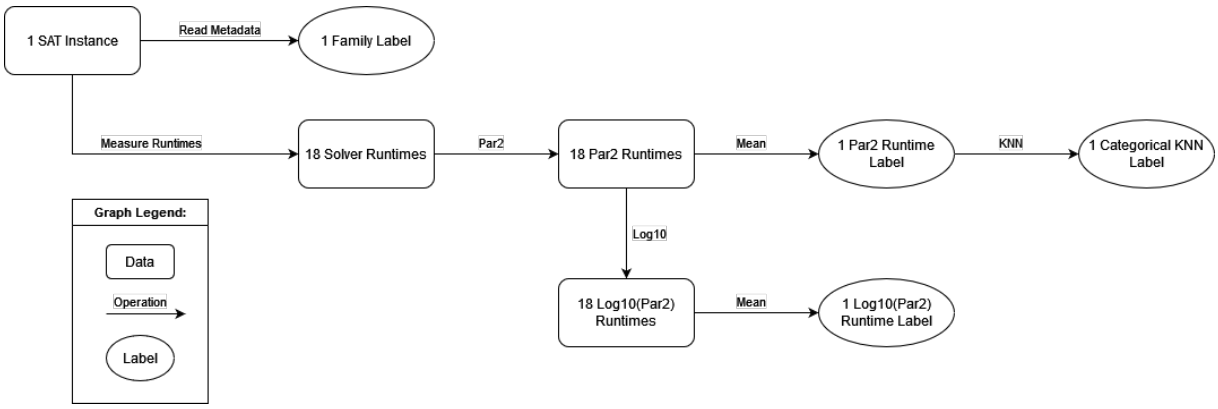


Figure 3.2: Overview of our used labels.

Figure 3.2 gives an overview of the described label creation process and the relationships between the labels. The WCSS graph, our evaluation of the Elbow method, as well as the result of the KNN algorithm can be found in Section 4.5.

3.2.4 Model

In this subsection, we will briefly outline our model selection, training, and evaluation process. After extracting all features and labelling all instances, we began the process of model selection. Our data set consists of instances from many different families, with some families having only a handful of occurrences, and is therefore very diverse and inhomogeneous. Moreover, Cryptography, which is known to be one of the most difficult instance families in SAT space, is the largest instance family in our data set, making the data set correspondingly difficult. Furthermore, relative to the number of occurrences of an instance family, we also have a high feature count. Finally, the problems we plan to address using the machine learning models include but are not limited to a multiclass family classification problem, a runtime regression problem, and a runtime classification problem.

Based on this situation, we were already confident that a Random Forest would be the most suitable machine learning model: Random forests are well suited for multiclass problems, can use a mixture of categorical and numerical features, and can even handle unscaled, highly variable data. Moreover, they can learn complex, highly nonlinear relationships while maintaining a simple structure [44]. They are therefore easier to train and interpret than other machine learning models (e.g., deep neural networks).

However, we still compared the performances of a few of the most popular machine learning models. Since some models require their features to be centred and scaled, we used Scikit-learn's StandardScaler on all our features for all models beforehand. This ensures a fair comparison. For this model evaluation, we selected a model from Scikit-learn's library, trained and tested it and noted the resulting scores. This process was repeated for all promising models. When comparing the models without hyperparameter optimization, we do not need validation sets. We therefore evaluated them using a 10-fold cross-validation over the whole data set, which results in less variance, less bias and less optimistic results than a simple train-test split.

As expected, the Random Forest achieved the best results on all three problems. For the comparison and the detailed results, see Chapter 4.

3.2.5 Hyperparameter Optimization

We also tried to optimize the hyperparameters of the Support Vector Machines, Multilayer-Perceptrons and Random Forests using Random Search [45]: To prevent any data leakage, and a resulting optimistic evaluation of the models, we first split the data set into a train-and-validate set and a hold-out test set in a 75-25 ratio.

For hyperparameter optimization, we first used a Random Search to identify 200 combinations of the most important hyperparameters (according to the corresponding Scikit-learn documentation), with the goal of finding local optima of the hyperparameter settings. The hyperparameters were evaluated with a 3-fold cross-validation on the train-and-validate set. Subsequently, the three best-performing hyperparameter settings were selected and trained on the entire train-and-validate set, and the inference was evaluated on the unseen test set. The best result was used as representative of the corresponding machine learning model. This process was repeated for all three machine learning models.

Overall, the results were unambiguous. Even with optimized hyperparameters, the Support Vector Machines and Multilayer Perceptrons performed worse than the base Random Forest. At the same time, the Random Forest showed only a negligible performance gain from optimized hyperparameters for two of the three labels, but incurred a significantly longer training time. We therefore continued to work with the base Random Forest model, as this not only saves optimization and training time, but also allows for better comparability to related work. The detailed results can be found in Subsection 4.6 of the Evaluation chapter.

4 Evaluation

In this chapter, we will evaluate the results of all our experiments.

4.1 Experimental Setup

In this section, we will present our experimental setup. While there is previous work related to the prediction of instance hardness and runtime, it has mostly focused on homogeneous data sets with a few instance families and the runtimes of one or a few SAT solvers. This results in limited generalizability. Additionally, these works generally struggled with the Cryptography family and neglected the time consumption of their pipelines. This in turn leads to limited practicability.

We will show that it is possible to predict the runtime and hardness of an arbitrary instance regardless of instance family and independent of any particular solver and create a generalizable and practical machine learning model. For this purpose, we used the following experimental setup:

To minimize the time consumption of the pipeline and make it as practical as possible, we omitted all pre-processing of the CNF's and VIG's and kept the pre-processing of the features at the minimum for each experiment. This is in contrast to related works, e.g. [8] and [31].

To abstract over instance families and ensure good generalizability, we gathered a reasonably-sized data set consisting of SAT instances taken from the 2020 and 2021 SAT competitions. We placed great emphasis on a diverse family distribution and high Cryptography fraction. This resulted in a data set consisting of 2373 instances from a total of 88 different families, with Cryptography as the most represented family at 20.24%. We believe this makes the data set sufficiently large, difficult, and diverse to make empirical conclusions.

To abstract over specific solvers, we created our labels using the runtimes of 18 popular solvers, including CaDiCaL, Glucose, Kissat and MiniSat. For the full list of used solvers, see Figure 4.1. For a categorical measure of hardness, we used the KNN algorithm to cluster the runtimes into the optimal number of clusters, which we determined using the Elbow method. We then used the extracted features and the created labels to train and compare multiple popular machine learning models. Since some models assume scaled and centred features, we used Scikit-learn's StandardScaler on our features for all models to assure this in our comparisons, while ensuring a fair comparison. With this setup, a good model performance should translate to fast, practical, and solver independent runtime and hardness predictions.

4.2 Feature Extraction

In this section, we will describe and evaluate our feature extraction process. The features to be extracted should capture the underlying structure of SAT problems and ideally be directly

• CaDiCaL	• Glucose	• Kissat
• CaDiCaL_elimfalse	• Glucose_chanseok	• Kissat_sweep
• CaDiCaL_pripro	• Glucose_syrup	• Lingeling
• CaDiCaL_stability	• Glucose_var_decay099	• LSTech_maple
• Candy	• March_nh	• MiniSat
• Relaxed	• Slime	• YalSAT

Table 4.1: List of all 18 solvers whose runtimes have been used.

correlated to solver runtime. We focussed on promising graph properties, such as community, centrality and clustering features. For many of them, current research suggests a correlation to solver runtime. But we also added features that have not yet been analysed in this context. For a detailed explanation on the decision process of our features, see Section 2.7.

We started off with all our CNF's in the DIMACS CNF file format (compare 2.2). This makes the CNF reading process very intuitive and straight forward. After reading the CNF file, a script builds a VIG based on the Formula 2.2.

Afterwards, we selected a large set of features, which we extracted for the first 100 VIGs and evaluated the average extraction time. We then discarded all features with unreasonably long extraction times of over one hour. These included, for instance, the small-world sigma and omega values and features based on graph diameters. After the removal, we ended up with 19 interesting features. Some features were a concrete value (e.g., the number of nodes in a graph), while others were a distribution (e.g., the degree centrality of each node). To capture the distributions, we used five new features: mean value, minimum value, maximum value, distribution variance, and distribution entropy. This way, we arrived at a total of 46 features. The whole list can be seen in Figure 4.1, which also clearly shows our desired focus:

We have 26 features focussing on different variations of centrality. Katsirelos and Simon [28], as well as Kambhampati and Liu [27] already used eigenvector centrality to discriminate between random and industrial instances. The former also showed correlation to clause learning. Jamali and Mitchell introduced the betweenness centrality and found evidence of it in industrial instances, and that the performance of the Glucose solver can be improved "through preferential bumping of high [betweenness] centrality variables" [46, p. 1]. The page rank, which is a variation of the eigenvector centrality, Katz and degree centrality have, to our knowledge, not been researched regarding differentiating SAT families. Moreover, none of our centralization features have been linked to runtime yet.

We have seven features that focus on communities. In this context, community count represents the number of communities found in the graph. Mean, min, max, coefficient of variance and entropy are based on community sizes, e.g. community mean is the mean size of a community in a graph. And modularity is a quantitative measure for community detection. All these community features are well researched and have already been associated with the differentiation of SAT families, as well as with the runtime of instances [31, 29, 19, 20].

We also have seven independent features focussing on clustering properties. Kambhampati and Liu have shown that the small-world property and the clustering coefficient [27] can be used to distinguish random from industrial instances. As mentioned earlier, we initially tried to employ the small-world property as well, but its computation was too costly on average

to be practical. Instead, we included 6 other clustering-related features, none of which have been previously explored in the SAT domain. By including these, we are attempting to gain new insights. Among them, “Is a proper clustering”, “Is a singleton clustering”, and “Is one clustering”, which, as the feature names imply, are Booleans, i.e., the graph is either a singleton clustering or it is not. They are in contrast to all our other features, which are numeric.

In addition, we have two features to scale the problem: the number of nodes and the number of edges. Finally, we have four general interesting graph properties that seem promising, and have not yet been explored in the context of our problem. We have included them as well, with the goal of gaining new insights.

4.3 Family Classification

In this section, we will evaluate our family classifications results. To determine whether our features would recognize the underlying structural differences between SAT families, which is necessary for efficient solving, we first tried to predict the instance family. Our used data set is about 2400 instances large and made up of 88 different SAT families. Figure 4.2 shows the family distribution in our data set. It is directly evident that 3 families are comparatively common, while most families have only a few occurrences. Some families even have only a single occurrence. This inhomogeneous distribution becomes clear when looking at the four most frequent families in Table 4.2. The table shows that Cryptography, one of the hardest known instance families in the SAT domain, is our most frequent family with 20.24% of the data set. This makes our data set a very difficult one, which in turn ensures that our model can handle difficult instances very well. Bitvector is the second most frequent family with 16.65%. From here on, we see the sharply decreasing fraction of the complete data set: Antibandwith already accounts for only 7.72%, less than half of Bitvector. The same can be observed for Planning, which accounts for 3.75% of our data set, again less than half of Antibandwidth. In Figure 4.2 we also see that apart from the four most frequent families, all families account for less than 70 occurrences or 3% of the total 2373 instances. In short, we have a very difficult, inhomogeneous, and diverse data set. Ideal to train a practical model that has to deal with many different families with varying degrees of difficulty.

For the family classification, we created three different labels that use the family metadata of the individual instances: The label “all_families” which, as the name implies, includes all 88 families. This label was straightforward to create as it is identical to the instance metadata. Because of the inhomogeneous distribution of families mentioned above, we also created the “four_families” label. It has a separate label for each of the three most common families and groups the remaining families together as “other”. Since we evaluated all models with a 10-fold cross-validation, we also created the label “all_families_T=10”, which groups all families with less than 10 occurrences as “other”. The “T” is an abbreviation for threshold. This label tests the highest possible family detail while ensuring that each cross-validation contains at least one representative of the family being tested.

Many models, such as the Support Vector Machine or the Multilayer-Perceptron, assume that the features are centred around 0 and have a variance in the same order [41]. They were therefore unable to perform on our unprocessed features. We therefore scaled all our features for all models using Scikit-learn’s StandardScaler, which ensures that the features fulfil the

Problem Size Features:

1-2. Number of nodes and edges

Centrality Features:

3. Centralization value
4-8. Degree centrality statistics: mean, min, max, variance coefficient and entropy
9-13. Eigenvector centrality statistics: mean, min, max, variance coefficient and entropy
14-18. Page rank statistics: mean, min, max, variance coefficient and entropy
19-23. Katz centrality statistics: mean, min, max, variance coefficient and entropy
24-28. Betweenness centrality statistics: mean, min, max, variance coefficient and entropy

Community Features:

29-34. Community statistics: count, mean, min, max, variance coefficient and entropy
35. Modularity value

Clustering Features:

36. Cluster imbalance value
37. Is a proper clustering*
38. Is a singleton clustering*
39. Is one clustering*
40. Global clustering coefficient value
41. Hub dominance value
42. Global intrapartition density value

Other Graph Features:

43. Number of connected components
44. Degree assortativity coefficient
45. Edge cut value
46. Edge cut fraction

Figure 4.1: Graph based SAT instance feature set. Booleans are marked with *.

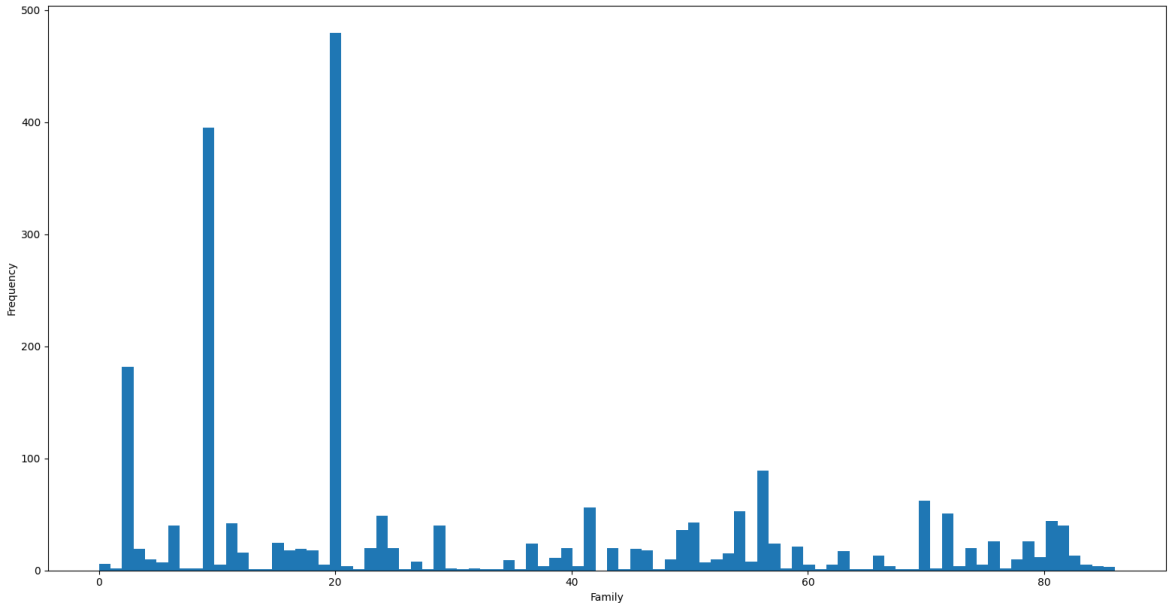


Figure 4.2: Distribution of family occurrences.

Family	Fraction of Data Set
1. Cryptography	0.2024
2. Bitvector	0.1665
3. Antibandwidth	0.0772
4. Planning	0.0375

Table 4.2: The four most frequent families in the data set used. Sorted in descending order according to their fraction.

Model	Performance on all_families label	Performance on all_families_T=10 label	Performance on four_families label
Random Forest	0.9742 ± 0.0052	0.9826 ± 0.0090	0.9945 ± 0.0060
Decision Tree	0.9314 ± 0.0056	0.9416 ± 0.0159	0.9852 ± 0.0051
Multilayer-Perceptron	0.9674 ± 0.0078	0.9716 ± 0.0087	0.9852 ± 0.0058
Support Vector Machine	0.8844 ± 0.0195	0.8882 ± 0.0155	0.9170 ± 0.0232
KNeighbors	0.6096 ± 0.0184	0.7557 ± 0.0148	0.9793 ± 0.0058
AdaBoost	0.3734 ± 0.0098	0.3764 ± 0.0112	0.5969 ± 0.0488
VotingClassifier (RF, MLP, SVM)	0.9682 ± 0.0087	0.9738 ± 0.0112	0.9865 ± 0.0068

Table 4.3: Family classification result of different models with varying labels. Features have been scaled, and all models use their base parameters.

aforementioned assumptions. We used the StandardScaler also on models that don't necessarily require it. This ensures a fair comparison.

For easier and better comparability, between the models and with related works, we evaluated the models using Scikit-learn's base parameters. Only for the KNeighbors classifier, we had to adjust the "k" parameter to reflect the number of categorical labels. Or in other words, the k parameter had to equal the number of families in the current label. For a full list of parameters and their default value, see Scikit-learn's documentation of the individual models [41].

With the data prepared, the labels created, and the features preprocessed, we trained and tested seven models with a 10-fold cross-validation over the whole data set. Table 4.3 shows our results: It is immediately obvious that the performance of the models increases with the decreasing number of families per label. This is to be expected, since the models do not have to differentiate as detailed and the chance of being correct by chance increases strongly: A random guesser would have an average accuracy of only 1/88, i.e. 1.14%, for the all_families label. For the all_families_T=10 label this would be 1/45, i.e. 2.22% and for the four_families label an accuracy of 1/4, i.e. 25%.

Of the compared models, the AdaBoost model consistently performed the worst, with an accuracy of less than 38% for the first two labels and not even 60% for the last label. AdaBoost is a meta-estimator that fits a chosen model on the given training data set. It then fits additional copies of the model on the same data set, adjusting the weights of misclassified instances so that subsequent models focus more on difficult cases of the previous models. This technique is also called Boosting. The result is probably due to the fact that the base parameters define a Decision Tree with a maximum depth of only one as the underlying model. Thus, despite Boosting, i.e. the sequencing of these shallow Decision Trees, the complex underlying structures can probably not be captured.

The next best model is the KNeighbors classifier, which is evidently not suitable for the prediction of a high number of families in our scenario. This is probably because KNeighbors is a clustering based classifier and many families have only a few occurrences. Looking at the distribution of families in Figure 4.2 and the fact that it achieved a performance of almost 98% on the four_families label, this hypothesis is further underlined.

The Support Vector Machine consistently has only acceptable performance, and shows little difference in performance between family labels, despite the vast differences in the number of possible labels. This is probably because SVMs are very good at handling high-dimensional spaces, but are known to underperform when the number of features is higher than the number of samples for a given label [41]. Thus, the accuracy of the SVM is bound to be determined by the predictions of the most frequent families, making the grouping of the least frequent families have little effect.

It was surprising to see that the Decision Tree performed so well despite its simplicity. It achieved over 93% accuracy for all_families, over 94% in predicting all families with at least 10 occurrences, and over 98% for the four_families label. A comparison with the performance of AdaBoost suggests that the unlimited maximum depth must have played a significant role in the achieved performance. It is likely that this helped the classifier to distinguish the less frequent families. However, the low to non-existent randomness in the classifier creation, combined with the unlimited maximum depth and the requirement of only two samples to split an internal node in the base parameters, leads to a very high probability of overfitting.

The Random Forest overcomes these problems by creating an ensemble of Decision Trees. The base Random Forest includes the Bootstrapping mechanism, i.e., each Decision Tree is created from a sample drawn with replacement from the training set. Moreover, only the square root of the number of all features is used in the search for the best split of an internal tree node. So in this case $sqrt(46) = 6.7823$, which is about seven features. These two sources of randomness address the problem of high variance and overfitting that Decision Trees suffer from. Moreover, when averaging the prediction of (by default) 100 Decision Trees, many errors cancel each other out, resulting in lower variance and, in our case, higher performance: The Random Forest outperformed all other models with an accuracy of 97.42% for the all_families label, 98.26% for the all_families_T=10 label, and 99.45% for the four_families label. With the above sources of randomness, the size and diversity of the data set, and the 10-fold cross-validation, overfitting is highly unlikely.

The Multilayer Perceptron (MLP) came quite close to the performance of the Random Forest for all labels. It lost only by 0.68 percentage points (P.P.) for the label all_families, 1.1 P.P. for the label all_families_T=10, and 0.93 P.P. for the label four_families. However, since the MLP is the basis for popular and complex (deep) neural networks, it probably has a lot of potential to improve by adjusting the architecture and hyperparameters. Especially considering that during the training process, the model suggested that the base number of 200 iterations is not enough for convergence and should be increased. But increasing the maximum number of iterations is (usually) accompanied by the adjustment of other hyperparameters and a longer training time. Since the results already show several models with very promising performance, including the MLP model itself, we have omitted hyperparameter optimization at this stage.

The VotingClassifier with the base parameters, combines conceptually different machine learning models by taking the majority vote over the individual predictions. E.g., for the given sample:

$$RF \rightarrow family_1$$
$$MLP \rightarrow family_1$$
$$SVM \rightarrow family_4$$

Label	Performance without feature scaling	Performance with feature scaling
all_families	0.9712 ± 0.0070	0.9742 ± 0.0052
all_families_T=10	0.9831 ± 0.0105	0.9826 ± 0.0090
four_families	0.9945 ± 0.0057	0.9945 ± 0.0060

Table 4.4: Performance comparison of a base Random Forest in family classification, with and without feature scaling.

the resulting prediction of the VotingClassifier would have been "$family_1$". Ties are broken by the order of the used models. E.g., for the given sample:

$$RF \rightarrow family_1$$
$$MLP \rightarrow family_2$$
$$SVM \rightarrow family_3$$

the resulting prediction of the VotingClassifier would have been "$family_1$", as the RF is the first model.

This allows the VotingClassifier to combine a number of (somewhat) similar performing models to compensate for their individual weaknesses. Interestingly, the VotingClassifier managed to outperform all models but the stand-alone RF for each label: It achieved 96.82% for the all_families label, 97.38% for the all_families_T=10 label, and 98.65% for the four_families label. It presumably couldn't beat the RF because the included SVM underperformed compared to MLP and RF, which may have confused the VotingClassifier.

Since Random Forests do not require feature scaling according to their documentation, and we place a high priority on minimizing time consumption, we also evaluated the base Random Forest without scaling the features beforehand. Table 4.4 shows that without scaling, the performance for the all_families label is slightly lower by 0.3 P.P., higher by 0.4 P.P. for the all_families_T=10 label, and identical for the four_families label. All of these values are within the range of variance. Overall, the comparison shows that there is no clear loss of performance when feature scaling is omitted, which is an advantage of Random Forests over many other models.

The conclusion of the family classification is that Random Forest performed best, with VotingClassifier and Multilayer Perceptron in second and third place, respectively. In addition, the Random Forest does not lose any performance when omitting feature scaling, which sets it apart from the other models even more. Overall, the high accuracy of our models indicates that there must be structural differences between instance families and that our features detect and may even correlate with these differences. Therefore, in our next step, we compared the most promising models from the family classification regarding runtime prediction.

4.4 Runtime Prediction

In this section, we evaluate our runtime prediction results. For many applications, it is only worth solving the SAT instance if it can be solved within a certain time limit. This is especially

the case when in contact with users. A predicted runtime would allow the application to easily decide for each use case whether it is worth solving the problem. Other applications could use a continuous measure of instance hardness to dynamically allocate resources to solving the instance. For example, an instance predicted to take 20s could be allocated twice the amount of resources as an instance predicted to take 10s. There are also many other possibilities that lie in-between. So before we can train a machine learning model on our previously extracted features, we first need to find suitable runtime labels.

As with SAT competitions and previous work, all of our solver runtimes had an upper time limit of 5000s, above which the benchmark was counted as a timeout. This is called the Cap-Time k, where k is the threshold (5000s in this case) [9]. The Cap-Time is a right-censored data point, since it only gives the lower bound of the actual time. We therefore treat our runtimes faster than they might have been, which leads to a downward bias in the data, and thus in our prediction. To address this bias, we computed the PAR2 value over our solver runtimes. PAR2 penalizes all timeout values by doubling them, while keeping all other runtimes unchanged. That is, for all timeouts, the 5000s runtimes were doubled to a value of 10000s, while the non-timeouts kept their original value. We believe this is a good indicator of problem hardness, especially if we make it independent of the solver.

To achieve solver independence, we averaged the PAR2 values of our 18 solvers for each SAT instance. The resulting float value was our runtime label "PAR2". We also calculated the $log_{10}(PAR2)$ before calculating the PAR2 mean across all solvers, which resulted in our "log10_PAR2" label. We created this label, as some models perform better on a logarithmic scale. For this label, we had to preprocess our data and remove all entries that had a runtime of 0. This is because the mathematical result of $log_{10}(0)$ is undefined. For a more detailed overview on the label creation process, see Subsection 3.2.3.

After creating these two labels, we used them to compare the most promising models from the family classification step (see Section 4.3). Once again, the features were first scaled and centred using the StandardScaler allowing us to use the same features and data across all models. In addition, we once more only used base parameters. We compared the models based on their R^2-score and RMSE, which were calculated through a 10-fold cross-validation.

R^2, also called coefficient of determination, is a regression function. The best possible value is 1.0. For a constant model that always predicts the average y and ignores the features, the R^2 value would be 0.0. Since regression models can become arbitrarily bad, the score can also be negative. Root-Mean-Square-Error (RMSE) is, as the name implies, the square root of the mean over the squared error. It is the standard deviation of the prediction error. Intuitively, it is a measure that tells us how far off the predictions are. Whether an RMSE is high or low depends solely on the data points. An RMSE of 0.5 for data points ranging from 0 to 10 is low, while it is high for data points ranging only from 0 to 1. In our case, the data points range from 0 to 10000.

Table 4.5 shows our results for the prediction of the PAR2 label. It is evident that the base MLP and SVM were not able to predict the runtimes at all. Both had a negative R^2-score, with the MLP performing worse than the SVM and a very high RMSE of ca. 4077 and 3375, respectively. Such poor performance for these models is probably due to very unsuitable base parameters. Looking at the hyperparameter optimization in Section 4.6, this hypothesis is underlined.

In contrast, the Random Forest performed very well, achieving a promising R^2-score of 0.5345 and an RMSE of about 2296. Both values were far ahead of the models mentioned above.

Model predicting PAR2 label	R^2**-Score**	**RMSE**
Random Forest	0.5345 ± 0.0665	2296.2468 ± 182.7320
Multilayer-Perceptron	-0.4618 ± 0.0826	4076.7335 ± 161.5039
Support Vector Machine	-0.0011 ± 0.0061	3375.1270 ± 94.5291
VotingClassifier (RF, MLP, SVM)	0.2649 ± 0.0304	2892.1883 ± 117.9822
VotingClassifier (RF, RF, RF)	0.5353 ± 0.0654	2294.3703 ± 181.2679

Table 4.5: Performance comparison of the most promising models on the PAR2 label. Features have been scaled and centred before, all models use their base parameters.

Model predicting log10_PAR2 label	R^2**-Score**	**RMSE**
Random Forest	0.5288 ± 0.1212	0.2800 ± 0.0496
Multilayer-Perceptron	-1.7310 ± 5.1564	0.5248 ± 0.4182
Support Vector Machine	0.4087 ± 0.0903	0.3144 ± 0.0373
VotingClassifier (RF, MLP, SVM)	0.2721 ± 0.5652	0.3333 ± 0.1065
VotingClassifier (RF, RF, RF)	0.5257 ± 0.1224	0.2808 ± 0.0492

Table 4.6: Performance comparison of the most promising models on the log10_PAR2 label. Features have been scaled and centred before, all models use their base parameters.

We also tested two VotingClassifiers. The first was trained with an RF, MLP, and SVM as we did in the family classification section. Looking at the performance of the MLP and SVM, the relatively low performance is to be expected. These models were not able to perform well and dragged the overall VotingClassifier down. Because the RF performed so consistently well, we trained a second VotingClassifier on three RFs, each using a different random seed to obtain different models. This VotingClassifier slightly outperformed a single RF. Its R^2-score is higher by 0.0008, and the RMSE also differs only in the decimal range. This difference, however, is small enough to be attributed to variance and does not justify three times the cost of training and inference.

Since many papers evaluate their models on a logarithmic scale, for instance [9], and some models handle logarithmic scales better, we also compared the prediction of the runtime on a logarithmic scale using the log10_PAR2 label. The performance of the MLP on the logarithmic scale actually declined from a R^2-score of -0.4618 to -1.7310. The relative distance to the RMSE scores of the other models also worsened. Surprisingly, the SVM's performance increased sharply, reaching a R^2-score of 0.4087 with an RMSE of 0.3144. Evidently, the SVM is one of the models that perform much better, at least in our evaluation, on a logarithmic scale. The RF is still outperforming the other models by a significant margin, with a R^2 score of 0.5288 and an RMSE of 0.2800. The loss in performance compared to the RF model evaluated on the PAR2 label is so marginal that it is likely attributable to variance. The VotingClassifier results are also comparable to the results on the PAR2 label, with similar performance losses to the RF.

Overall, the MLP performed the worst. With a negative R^2-score, it obviously performed worse than a constant model that only guesses the overall average of the labels. On the one hand, this is probably because the base parameters are potentially very mismatched. MLPs form the basis of modern (deep) neural networks and are highly susceptible to changes in parameters, i.e., they rely on appropriate optimization of hyperparameters for good performance. The

Model	Label	R^2-Score	RMSE
AdaBoost	PAR2	0.2191 ± 0.0236	-2980.4194 ± 91.4264
Decision Tree	PAR2	0.1404 ± 0.1115	-3119.9754 ± 207.1948
AdaBoost	log10_PAR2	0.0538 ± 0.2375	-0.3940 ± 0.0364
Decision Tree	log10_PAR2	0.0490 ± 0.3678	-0.3904 ± 0.0728

Table 4.7: Performance of the AdaBoost and Decision Tree models on the PAR2 and log10_PAR2 labels. Features have been scaled and centred before, all models use their base parameters.

fact that performance increases noticeably with hyperparameter optimization in Section 4.6 underlines this conjecture. On the other hand, it could also mean that an MLP model is simply not well suited for the given problem, since the data set has "only" about 2400 data points and a relatively high number of features compared to the occurrence of each family.

The SVM performed similarly bad as the MLP for the label PAR2, but showed acceptable performance for the label log10_PAR2. Evidently, the SVM can only handle the logarithmic scale for the given problem. However, even on the logarithmic scale, the performance was not comparable to the performance of the RF. While SVMs can be effective in high-dimensional spaces, they tend to overfit when the number of features is larger than the number of samples (as is the case for many of the less frequent families). In addition, non-gaussian SVMs (such as the basic SVMs in the Scikit-learn library) are neither translation invariant nor monotonic transformation invariant [47]. Therefore, careful preparation of data and features is often required. While we scaled and centred our features using the StandardScaler, the mixture of categorical and numerical features with widely varying values could be a cause for problems for the SVM.

For the sake of completeness, the less promising models AdaBoost and Decision Tree were also evaluated. Surprisingly, these models performed slightly better than the MLP models, however, the results still indicate that these models are not suitable for the problem at hand (see Table 4.7). For this reason, we did not include these models any further in the comparisons.

Of all the compared base models, the Random Forest performed by far the best, as expected from the family classification results. It consistently achieved a R^2-score above 0.52 (see Tables 4.5 and 4.6). This means that at least 52% of the variability of the PAR2 runtime is accounted for by our chosen features. This high performance is likely due to several factors: Random Forests can handle a mixture of categorical and numerical features and a highly variable and inhomogeneous data set very well. Moreover, they are also good at dealing with outliers by binning the relevant variables. Finally, they can also handle a relatively high number of features very well.

Similar to the family classification section, we evaluated the Random Forest for the runtime regression problem without scaling as well. Table 4.8 shows that for both labels, the R^2 score and RMSE are slightly higher without prior feature scaling. However, the differences are so minor that they are likely attributable to variance. These results show once again that the Random Forest does not require data or feature scaling. This means that one can skip the feature scaling and centring step when using the Random Forest. This can save a lot of time,

Label	Metric	Without Scaling	With Scaling
PAR2	R^2-score	0.5364 ± 0.0639	0.5345 ± 0.0665
PAR2	RMSE	2301.7807 ± 166.6166	2296.2468 ± 182.7320
log10_PAR2	R^2-score	0.5309 ± 0.1189	0.5288 ± 0.1212
log10_PAR2	RMSE	0.2807 ± 0.0518	0.2800 ± 0.0496

Table 4.8: Comparison of the base Random Forest performance with, and without scaling the features beforehand.

especially for larger data sets, which is another reason to prefer this model over the other evaluated models.

In summary, the base Random Forest is our most promising and performant model for the runtime prediction. Using our features over a data set of 2373 instances with the PAR2 and log10_PAR2 label, it achieved a R^2-score of 0.5364 and 0.5309, respectively. The corresponding RMSE values are at 2296.2468 and 0.2800, respectively. All these values have been achieved without the need of preprocessing the CNF, the data or the features. Only the instances with an average runtime of 0 (i.e. instances for which all 18 solvers finished fast enough for the runtime to be rounded down to 0), had to be removed for the log10_PAR2 labels, as $log_{10}(0)$ is undefined.

4.5 Categorical Hardness Prediction

In this section, we will describe how we created a categorical measure of instance hardness and evaluate how well our model predicts it. Often, a concrete runtime number is not necessary and a simple categorical hardness label (e.g., “easy” or “hard”) may be sufficient. To this end, we have taken the PAR2 runtime labels and grouped them into k clusters using the KNN algorithm. The goal is to create a categorical measure of hardness from the continuous, solver-independent hardness measure we created in the previous section.

The number of clusters chosen for a given data set should not be random. Each cluster is formed by calculating and comparing the distances of the data points within a cluster to its centroid. One way to determine the correct number of clusters is to calculate the Within-Cluster Sum of Squares (WCSS), defined by Equation 2.1. The goal is to minimize this sum by choosing the correct number of clusters. Given n observations in a data set, if we were to choose k = n for the KNN algorithm, we would have an ideal WCSS of 0. However, this has no real added value since we now have as many clusters as data points. Therefore, we need to find a tradeoff between WCSS and the number of clusters, which we can find using the Elbow method [43]: First, we create a graph that relates the WCSS to the number of clusters. In this graph, we can see that when the number of clusters is low, adding a cluster leads to a significant decrease in WCSS. However, above a certain threshold, adding clusters only minimally decreases the WCSS. This threshold is then the optimal k according to the Elbow method.

Figure 4.3 shows that the decrease in the WCSS per added cluster decreases sharply after k = 3. The graph becomes much flatter from this point on, it is the “elbow” of the graph, so to speak. This threshold, and thus the optimal number of clusters and hardness levels in our data set, is 3. This result is further illustrated by visualizing the clusterings and comparing the optimal

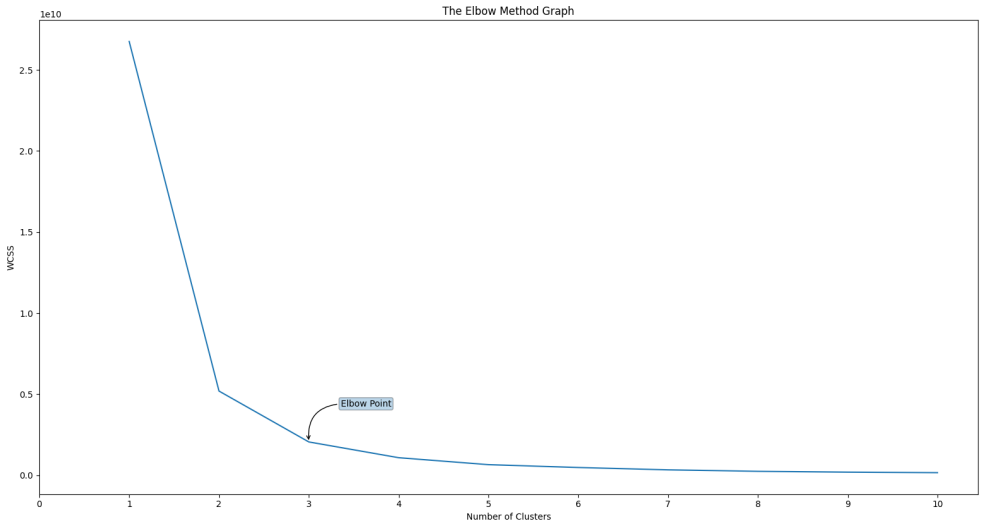
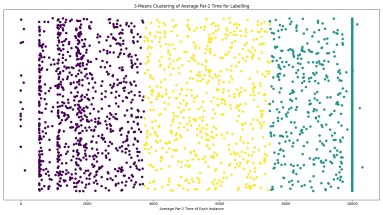


Figure 4.3: Graph assigning number of clusters to resulting WCSS. Using the Elbow method, we can determine the optimal number of clusters in this graph is 3.

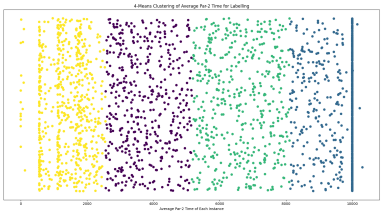
clustering for k = 3 with the clusterings of higher k values. Figure 4.4 visualizes the clusterings for varying k, with Subfigure 4.4a visualizing the clustering for k = 3. We can see that each label has a comparable number of instances and that the algorithm did not simply split the search space (from 0 to 10000) into thirds, but clustered it as effectively as possible. Subfigures 4.4b and 4.4c show that for k = 4 and k = 5, the algorithm simply shifts the thresholds without increasing the distance between clusters or their clarity. If more granularity is needed, one could manually add a “very light” label for the instances near 0 and a “very heavy” label for the timeout labels, which can be seen at the left and right edges of Subfigure 4.4a. This would evidently give a better result than running the KNN algorithm with k = 5.

After creating the hardness labels using the KNN algorithm with k = 3, we named them “easy”, “medium” and “hard” respectively. We called this label “3-means”. Using the new label, we then trained, evaluated, and compared the same five models as in Section 4.4. Once again, the features were scaled and centred beforehand to allow for a fair comparison. The results can be seen in Table 4.9:

Impressively, all five models achieved an accuracy of at least 61%. For comparison, a random estimator would achieve an accuracy of about 33%, depending on the fraction of the most frequent label. Of the compared models, the SVM performed the worst with an accuracy of about 61.16%. The next best model was the MLP, with an accuracy of 66.22%. This was followed by the VotingClassifier, which made the majority decision on an RF, an MLP, and an SVM. These three rankings are unsurprising and likely have the same causes as outlined in the Sections 4.3 and 4.4. What was surprising was the performance of VotingClassifier when voting on three different RFs. This time it performed worse than a stand-alone RF. However, the difference in performance is so small that it could still be attributed to variance. Whether or not this is the case, the VotingClassifier is obviously not worth three times the amount of resources of



(a) KNN clustering for k = 3.



(b) KNN clustering for k = 4.

(c) KNN clustering for k = 5.

Figure 4.4: Scatter plots of KNN clusterings over PAR2 runtime with varying k.

Model	**Performance on 3-means label**
RF	0.7116 ± 0.0225
RF w/o scaling	0.7129 ± 0.0230
MLP	0.6622 ± 0.0258
SVM	0.6116 ± 0.0216
VotingClassifier(RF, MLP, SVM)	0.6768 ± 0.0229
VotingClassifier(RF, RF, RF)	0.7103 ± 0.0220

Table 4.9: Performance comparison of the most promising models on the 3-means label. Features have been scaled and centred before, unless mentioned otherwise. All models use their base parameters.

a single RF. Once again, the base RF is the most practical and best performing model, with an impressive 71.16% accuracy. As was made clear again in the previous two sections, the RF does not require feature scaling or centring. For this reason, we evaluated it without it as well, which yielded an accuracy of 71.29%. Thus, the scaling can be omitted here as well.

For many applications, this classification performance is sufficient to provide noticeable improvements over blindly trying to solve every SAT instance. For example, the application could always solve easy problems and only solve medium problems in certain instances, while always skipping hard problems, which would save a significant amount of time. Alternatively, the application could use more resources and more complex solvers to solve the more difficult instances.

4.6 Hyperparameter Optimization

In this section, we will briefly describe our approach to optimize the models' hyperparameters and evaluate the results. While the base Random Forest model consistently outperformed all other base models, it is still possible that it falls behind when comparing models with optimized hyperparameters. Therefore, we compared the three most promising models from the previous sections with improved hyperparameters: Random Forest, Multilayer Perceptron, and Support Vector Machine. Our work is not about finding the perfect hyperparameters for the models, but testing our features on the given difficult data set, with meaningful labels. In the process, the comparability to other papers should be preserved as much as possible. Therefore, we only aim to provide a general overview of what can be expected with better hyperparameters.

For better comparability, we once again scaled the features for all models. After scaling the features, we had to adjust our training and evaluation method to account for hyperparameter optimization. To avoid an upward biased test result, we split our data set into a train-and-validate set and a hold-out test set at a ratio of 75-25. We then ran 200 iterations of Random Search with a specified distribution and list of hyperparameters. While it is entirely possible that Random Search will not find the optimal parameters down to the decimal digits, the method is ideal for finding generally well-performing parameters given a sufficient number of iterations.

The hyperparameters used in the Random Search were selected based on the Scikit-learn documentation on the most effective hyperparameters for each of the three compared models. In each iteration of Random Search, a randomly selected configuration of parameters is used for a model, which is then evaluated with 3-fold cross-validation. We chose to use 3-fold cross-validation at this stage because it has less variance than splitting the train-and-validate set further into a training and validation set, without increasing the computational cost too much.

We then sorted the results by the average cross-validation score. Then we took the three best hyperparameter configurations of each model, fit each of them on the entire train-and-validate set, and evaluated them on the yet-unseen test set. We performed this process for both runtime prediction and categorical hardness prediction. We omitted the family classification, since the performance there was already very good, with little room for improvement. The adapted training and evaluation process is also the reason the performance of the base models may differ from the previously evaluated ones.

Table 4.10, 4.11, and 4.12 show the results of the base models and of the three best hyperparameter configurations for the MLP, SVM and RF, respectively. Parameters not listed explicitly in the hyperparameter column were left at their default value of the Scikit-learn library. The CV score in the table is the average test score of the 3-fold cross-validation performed to compare the different hyperparameter configurations. The test score is the performance of the model on the yet-unseen test set. We can see that a better CV performance on the training set does not always result in a better performance in the actual test set. This illustrates that a good validation performance doesn't always correlate to a good test performance, even when using cross-validation. Similarly, a high test performance does not necessarily translate to a high performance in the desired application. As always, the results are theoretical and should be verified and adapted for each specific application.

Label	Chosen MLP hyperparameters	CV Score	Test score
PAR2	base parameters	X	-0.7763
PAR2	{solver: "adam", max_iter: 8055, learning_rate: "constant", hidden_layer_sizes: (156,), alpha: 0.0050}	0.2835	0.3556
PAR2	{solver: "adam", max_iter: 4405, learning_rate: "adaptive", hidden_layer_sizes: (156,), alpha: 0.0050}	0.2761	0.3459
PAR2	{solver: "adam", max_iter: 7718, learning_rate: "constant", hidden_layer_sizes: (152,), alpha: 0.0001}	0.2574	0.3637
Log10_PAR2	base parameters	X	0.1646
Log10_PAR2	{solver: "adam", max_iter: 3115, learning_rate: "adaptive", hidden_layer_sizes: (320,), alpha: 0.0500}	0.0569	0.2790
Log10_PAR2	{solver: "adam", max_iter: 1389, learning_rate: "constant", hidden_layer_sizes: (320,), alpha: 0.0100}	-0.0575	0.2223
Log10_PAR2	{solver: "adam", max_iter: 3194, learning_rate: "adaptive", hidden_layer_sizes: (156,), alpha: 0.0500}	-0.0766	0.2229
3-means	base parameters	X	0.6681
3-means	{solver: "lbfgs", max_iter: 238, learning_rate: "constant", hidden_layer_sizes: (710, 710, 710), alpha:0.0050}	0.6715	0.6810
3-means	{solver: "lbfgs", max_iter: 437, learning_rate: "adaptive", hidden_layer_sizes: (81, 81, 81), alpha:0.0500}	0.6682	0.6621
3-means	{solver: "lbfgs", max_iter: 100, learning_rate: "adaptive", hidden_layer_sizes: (116, 116), alpha:0.0001}	0.6650	0.6655

Table 4.10: Hyperparameter optimization results for the MLP. Parameters not listed are left at the default value.

Label	Chosen SVM hyperparameters	CV Score	Test score
PAR2	base parameters	X	-0.7763
PAR2	{C: 114.8874, cache_size: 200, gamma: 0.0384, kernel: "rbf", max_iter: 8214}	0.6594	0.6711
PAR2	{C: 27.9700, cache_size: 500, gamma: 0.0499, kernel: "rbf", max_iter: 8769}	0.6529	0.6703
PAR2	{C: 13.5889, cache_size: 200, gamma: 0.1469, kernel: "rbf", max_iter: 9841}	0.6508	0.6659
Log10_PAR2	base parameters	X	0.1646
Log10_PAR2	{C: 51.1326, cache_size: 200, gamma: 0.2270, kernel: "rbf", max_iter: 3948}	0.4105	0.3750
Log10_PAR2	{C: 31.6023, cache_size: 200, gamma: 0.2418, kernel: "rbf", max_iter: 5099}	0.3725	0.3646
Log10_PAR2	{C: 27.2118, cache_size: 500, gamma: 0.2328, kernel: "rbf", max_iter: 1310}	0.3598	0.3568
3-means	base parameters	X	0.6161
3-means	{C: 114.8874, cache_size: 200, gamma: 0.0384, kernel: "rbf", max_iter: 8214}	0.6594	0.6711
3-means	{C: 27.9700, cache_size: 500, gamma: 0.0499, kernel: "rbf", max_iter: 8769}	0.6529	0.6703
3-means	{C: 13.5889, cache_size: 200, gamma: 0.1469, kernel: "rbf", max_iter: 9841}	0.6508	0.6659

Table 4.11: Hyperparameter optimization results for the SVM. Parameters not listed are left at the default value.

Label	Chosen RF hyperparameters	CV Score	Test score
PAR2	base parameters	X	0.4921
PAR2	{n_estimators: 800, min_samples_split: 2, min_samples_leaf: 1, max_features: "sqrt", max_depth: None}	0.4328	0.4986
PAR2	{n_estimators: 2000, min_samples_split: 5, min_samples_leaf: 1, max_features: "sqrt", max_depth: 70}	0.4298	0.4928
PAR2	{n_estimators: 800, min_samples_split: 5, min_samples_leaf: 2, max_features: "sqrt", max_depth: 110}	0.4267	0.4870
Log10_PAR2	base parameters	X	0.4795
Log10_PAR2	{n_estimators: 800, min_samples_split: 2, min_samples_leaf: 1, max_features: "sqrt", max_depth: None}	0.3947	0.5315
Log10_PAR2	{n_estimators: 2000, min_samples_split: 2, min_samples_leaf: 1, max_features: "sqrt", max_depth: None}	0.3932	0.5329
Log10_PAR2	{n_estimators: 1600, min_samples_split: 2, min_samples_leaf: 1, max_features: "sqrt", max_depth: 100}	0.3916	0.5331
3-means	base parameters	X	0.7223
3-means	{n_estimators: 1600, min_samples_split: 10, min_samples_leaf: 1, max_features: "auto", max_depth: 40}	0.6984	0.7050
3-means	{n_estimators: 1600, min_samples_split: 10, min_samples_leaf: 1, max_features: "sqrt", max_depth: 20}	0.6963	0.7115
3-means	{n_estimators: 1600, min_samples_split: 5, min_samples_leaf: 2, max_features: "auto", max_depth: 100}	0.6919	0.7115

Table 4.12: Hyperparameter optimization results for the RF. Parameters not listed are left at the default value.

In Table 4.13 we summarized our results. We compared the performance of the model with base parameters to the model with the best hyperparameter configuration here. In two cases, the SVM with the log10_PAR2 label and the RF with the 3-means label, the base parameters were actually better than the best configurations found by 200 iterations of Random Search. These were marked with an "*" in the table. In addition, optimizing the hyperparameters resulted in no performance improvement or a performance improvement of less than 2 percentage points in 4 out of 9 cases. This underlines our assessment that the basic parameters in the Scikit-learn library are already very good for many use cases. In Table 4.13 it can also be observed that hyperparameter optimization for MLP and SVM yielded high improvements for the PAR2 label, with a new R^2-score of 0.3637 and 0.2872, respectively. The RF showed little improvement, with a score increase of only 0.0065. Similarly, MLP and RF achieved decent performance gains for the log10_PAR2 label, with score increases of 0.1144 and 0.0536, respectively, while the baseline model actually remained the best performing model for the SVM. The same applies to the RF for the 3-means label. Here, only the MLP and SVM scores increased by 0.0129 and 0.0550, respectively, while the base model remained the best performing model for the RF.

On average, the SVM and the MLP model benefited a lot more from hyperparameter optimization. However, even with hyperparameter optimization, the RF model consistently remained the best performing model. In fact, even with hyperparameters optimized, the MLP and SVM performed worse than the base RF model by a large margin for all labels. This supports our hypothesis that a RF is the best model for the problems at hand. In addition, the RF showed only a negligible performance gain of less than 1.5% with hyperparameter optimization for two of the three labels, but required significantly longer training time. We therefore continued to

Model	Label	Score with base hyperparameters	Score with best hyperparameters
MLP	PAR2	-0.7763	0.3637
SVM	PAR2	-0.0151	0.2872
RF	PAR2	0.4921	0.4986
MLP	log10_PAR2	0.1646	0.2790
SVM	log10_PAR2	0.4520	0.4520*
RF	log10_PAR2	0.4795	0.5331
MLP	3-means	0.6681	0.6810
SVM	3-means	0.6161	0.6711
RF	3-means	0.7223	0.7223*

Table 4.13: Comparison between base models and optimized hyperparameters, as well as between different labels and models. Models where the base parameters were the best parameter choice are marked with *.

work with the RF base model, as this not only saves optimization and training time, but also allows better comparability with other work. We recommend optimizing the hyperparameters of the RF model only for use cases where optimization and training time play a minor role and the absolute best possible performance is required.

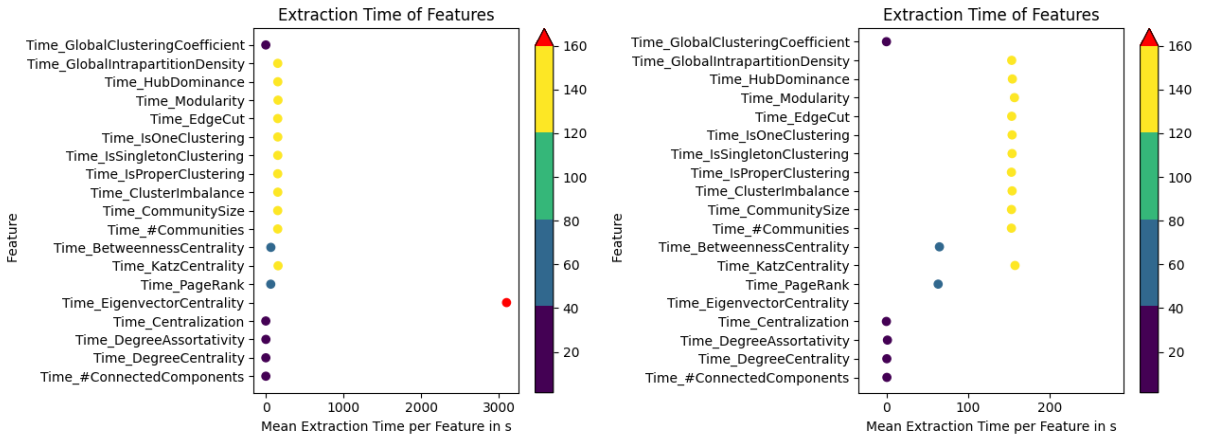
4.7 Feature Extraction Costs

In this section, we will evaluate the extraction costs of our features. Knowing how expensive the extraction of a feature is enables the decision whether it is worth extracting the feature for a given use case. In the later Subsection 4.8.1, we will determine the importance of each feature and combine it with the extraction costs determined here. This in turn will allow us to optimize the feature extraction in our pipeline.

We measured the CPU-time required to extract each feature. This is the time the CPU is busy processing the program instructions and does not include the time the CPU is idle, e.g. due to scheduling. Since the features “#Nodes” and “#Edges” are part of the metadata of the VIG and therefore effectively do not cost computation time, they have been omitted from all subsequent figures and tables on extraction time. In the case of distributions (e.g., the degree centrality of nodes), we combined all 5 corresponding features into one time measurement, since the additional computation time of common distribution operations such as max or variance are negligible when using a highly optimized library such as NumPy.

We then averaged the extraction time of each feature over all 2373 instances. This results in a total count of 19 extraction times. Visualizing these average extraction times per feature revealed four clusters or categories, which we have colour-coded in Figure 4.5. It is immediately apparent that eigenvector centrality is an outlier with its mean extraction time of 3099s. It is the only feature in the “very expensive” cost category. Once it is removed, the remaining three clusters become clearer, as illustrated in Subfigure 4.5b.

In Table 4.14 we created an overview of the four cost categories and which features fall into them. As a counterpart, Table 4.15 shows the average extraction time for each feature and



(a) All extraction times.

(b) Zoomed in.

Figure 4.5: Mean extraction times of features.

which category the feature falls into. A look at the tables shows that the “cheap” category, with a threshold of 1.5 seconds, has a total of five features. We will see in Subsection 4.8.1 that this category can already be sufficient for many use cases. The medium category contains all features that have an average extraction time of $[1.5 - 70)$ seconds, which is only two in total. Finally, the expensive category is the most populated category, with 10 features in the remaining time cost range of $[70 - 160)$ seconds.

4.8 Pipeline Adjustments

In this section, we will evaluate two ways to adjust our pipeline for practical use.

4.8.1 Minimal Pipeline Runtime

The first way to adjust our pipeline is by minimizing the pipeline runtime. This is especially useful for time-sensitive or interactive applications. For this purpose, we first need to know the importance of each feature. If we know which features are important for predicting the model and which are not, we can reduce the number of features extracted and thus reduce the total feature extraction time, which consumes the most time.

To determine the most important features for our model, we first computed the permutation-based and tree-based feature importances. The problem with the computed tree-based feature importance is that it is highly biased and favours high cardinality features over low cardinality features (e.g., binary features), which explains the results in Figure 4.7. Permutation-based feature importance avoids this bias and can be computed for any model, not just tree models. However, according to the permutation feature importance shown in 4.6, permuting a feature reduces the accuracy by at most 0.014, which would mean that none of the features are important. This is in contrast to the high performance of our models evaluated in the previous sections. Some features must be important, which means that the importance results are misleading. The reason is that some of our features are correlated. When features are correlated, permuting

Cost Category	Time Threshold	Feature
Cheap	< 1.5s	GlobalClusteringCoefficient
		Centralization
		DegreeAssortativity
		DegreeCentrality
		#ConnectedComponents
Moderate	< 70s	BetweennessCentrality
		PageRank
Expensive	< 160s	GlobalIntrapartitionDensity
		HubDominance
		Modularity
		EdgeCut
		IsOneClustering
		IsSingletonClustering
		IsProperClustering
		ClusterImbalance
		CommunitySize
		KatzCentrality
Very expensive	>= 160s	EigenvectorCentrality

Table 4.14: Cost categories with their corresponding time thresholds and all features that fall into them.

a feature has little effect on the performance of the model because the same information can be obtained from a correlated feature. This leads to the misleading results in Figure 4.6.

Figure 4.9 shows this feature correlation in the form of a heatmap. The colours range from purple, which represents low to no correlation, to green, which represents medium-high correlation, to yellow, which represents strong correlation. Ideally, the features would only have a correlation to themselves and to the other four distribution capturing features in case of distribution features (e.g., minimum centrality correlates to maximum, medium, etc. of the same centrality feature). This would be visible as yellow blocks along the diagonal. The heat map shows, however, that many features have varying degrees of correlation to many other features. To handle these correlated features, research suggests using hierarchical clustering on the Spearman rank-order correlation [48].

This hierarchical clustering can be seen in Figure 4.8. The closer two features are connected vertically, the stronger or the more direct is their correlation. The most apparent clusters are separated by colours. Using thresholds, you can now choose how fine-grained you want to cluster the features. For example, a threshold at T=2 would result in 2 clusters, while a threshold at T=0 would give all features their own cluster. The higher the threshold, the more important the resulting features. Then, one representative feature is returned for each cluster.

We selected thresholds manually via visual inspection of the generated dendrogram. Since the hardness prediction label 3-means is generated from the runtime labels, the results would be identical to the runtime regression model results and was therefore omitted. Some of the

Feature	Mean extraction time in seconds	Cost Category
#ConnectedComponents	0.6118	cheap
DegreeCentrality	0.5230	cheap
DegreeAssortativity	1.2625	cheap
Centralization	0.004	cheap
EigenvectorCentrality	3099.5754	very expensive
PageRank	63.5156	moderate
KatzCentrality	157.823	expensive
BetweennessCentrality	65.0847	moderate
#Communities	153.4262	expensive
CommunitySize	153.4802	expensive
ClusterImbalance	154.2551	expensive
IsProperClustering	153.4308	expensive
IsSingletonClustering	154.2181	expensive
IsOneClustering	154.2478	expensive
EdgeCut	153.7611	expensive
Modularity	157.1361	expensive
HubDominance	154.5758	expensive
GlobalIntrapartitionDensity	153.7518	expensive
GlobalClusteringCoefficient	0.0772	cheap

Table 4.15: Features with their average extraction time and their corresponding cost category.

Figure 4.6: Permutation feature importance on family classification.

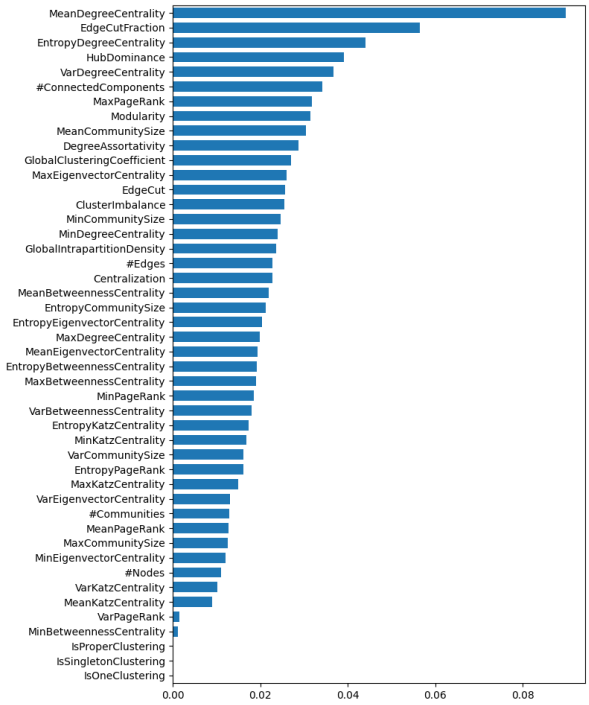


Figure 4.7: Tree based feature importance on family classification.

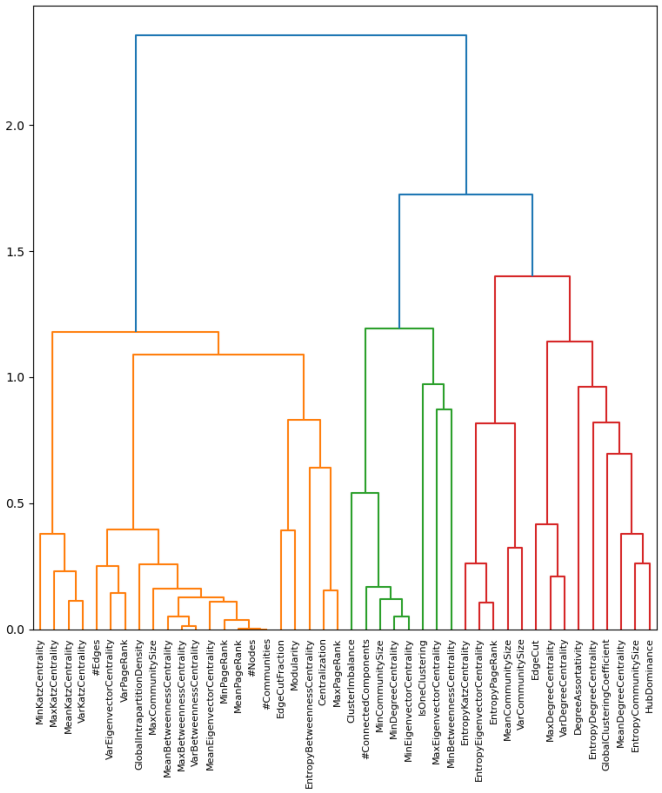


Figure 4.8: Dendrogram of hierarchical clustering based on Spearman rank-order.

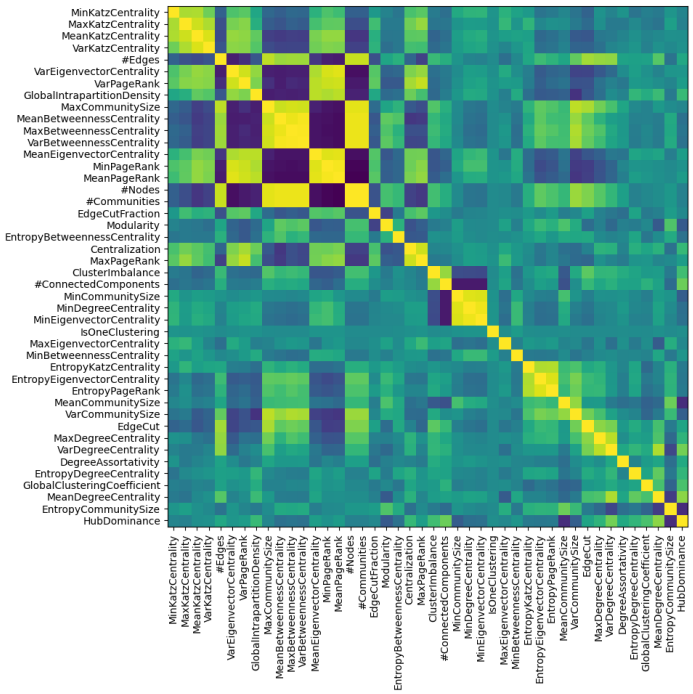


Figure 4.9: Heatmap of correlated features.

thresholds we manually selected, and the corresponding features, can be seen in Table 4.16. We have limited the table to thresholds of T=1.15 and greater to keep it concise.

Interestingly, the list of important features for classification and regression was exactly the same. This supports the hypothesis that solvers use the underlying structure of SAT instances to solve instances efficiently. This in turn suggests that runtimes are correlated with underlying structures. And since we used the average PAR2 runtime of 18 different solvers for the runtime regression, this seems to be the case for efficient solvers in general and not for any particular one. Which, in turn, suggests that our PAR2 label is an appropriate measure of instance hardness.

Since the Random Forest outperformed all practical models in our previous comparisons and generally looked the most promising, we decided to use it for the feature importance comparisons. We trained the Random Forest once for family classification and once for runtime regression and evaluated it using 10-fold cross-validation. For family classification we used the label all_families, and for runtime regression we used the label PAR2. The results can be seen in Table 4.17:

As expected, when the threshold value increases and thus the number of features used decreases, the performance drops as well. However, we can also see that the loss of performance per feature removed increases almost exponentially as the number of features used shrinks. The loss of performance in family classification when using only 8 features instead of all 46 is 0.77 P.P. This is a loss per removed feature of only about 0.0203 P.P. For runtime regression, the loss per removed feature is about 0.0068 P.P.

Now, if we increase the threshold from T=1 to T=1.1 and thus remove one feature, the performance loss is 1.31 P.P. for family classification and 0.27 P.P. for runtime regression, which is over 64 and 39 times larger, respectively. The fact that the loss per removed feature

Threshold	Family classification features	Runtime regression features
T=1.15	#Nodes, #ConnectedComponents, MeanDegreeCentrality, EntropyDegreeCentrality, MaxEigenvectorCentrality, MeanKatzCentrality	#Nodes, #ConnectedComponents, MeanDegreeCentrality, EntropyDegreeCentrality, MaxEigenvectorCentrality, MeanKatzCentrality
T=1.2	#Nodes, #ConnectedComponents, MeanDegreeCentrality, EntropyDegreeCentrality	#Nodes, #ConnectedComponents, MeanDegreeCentrality, EntropyDegreeCentrality
T=1.5	#Nodes, #ConnectedComponents, MeanDegreeCentrality	#Nodes, #ConnectedComponents, MeanDegreeCentrality
T=2	#Nodes, #ConnectedComponents	#Nodes, #ConnectedComponents

Table 4.16: Thresholds and the corresponding feature representatives when using hierarchical clustering.

increases so dramatically underscores the hypothesis that a few features account for most of the prediction. If we now minimize the number of features from the perspective of optimizing feature extraction time, it is reasonable to use a threshold of at least 1.2, since this removes the eigenvector centrality from the used features. Although eigenvector centrality is one of the top six features, it is not worth the additional performance due to its exceptionally high extraction time of 3099s on average in this context.

While increasing the threshold from 1.2 to 1.5 reduces performance by "only" 1.87 P.P. for family classification and 1.65 P.P. for runtime regression, it does not provide any benefit regarding extraction time optimization. Increasing the threshold removes the feature "EntropyDegreeCentrality", which is one of the five features describing degree centrality. However, at T=1.5, the feature "MeanDegreeCentrality" is still included in the list of features used, which means that the distribution of degree centrality must still be calculated. Considering that we are using the highly optimized NumPy library, the cost of computing entropy over the above distribution is negligible. The next option would be to increase the threshold from 1.2 to 2. However, this would result in performance scores of only 0.7513 and 0.2203 for the family classification and runtime regression, respectively. This is no longer a satisfactory performance, and thus not a practical option. The fact that removing the centrality features results in a loss of performance of 17.33 P.P. and 28.48 P.P., respectively, supports prior research findings that centrality has a strong correlation with instance family and runtime.

Overall, these results allowed us to determine the four most important features: "#Nodes", "#ConnectedComponents", "MeanDegreeCentrality" and "EntropyDegreeCentrality". Now that we have determined the four most important features, we looked into combinations of these features to find the time-performance optimum. Table 4.18 shows the four most important features sorted by importance and the performance when a Random Forest is trained and evaluated on a specific set of features with the corresponding mean feature extraction time. From these results we can see, that there are two options to optimize the mean extraction time, depending on the desired performance and speed-up:

By using only the top 4 features instead of all 46, we can already reduce the mean extraction time from 4930.7690 seconds to 1.1348 seconds, while suffering a performance loss of less than 3%. By removing the number of connected components from these four features, we can achieve an even higher speed-up. Only using the top 3 features reduces the extraction time

Threshold	#Features	Family classification accuracy	Runtime regression R^2-score
T=0	46	0.9712± 0.0070	0.5364 ± 0.0639
T=1	8	0.9635 ± 0.0093	0.5338 ± 0.0560
T=1.1	7	0.9504 ± 0.0077	0.5311 ± 0.0095
T=1.15	6	0.9498 ± 0.081	0.5274 ± 0.0390
T=1.2	4	0.9433 ± 0.0118	0.5216 ± 0.0578
T=1.5	3	0.9246 ± 0.0130	0.5051 ± 0.0561
T=2	2	0.7513 ± 0.0109	0.2203 ± 0.0590

Table 4.17: Comparison of model performances when training over a subset of features. The subset of features is given by thresholds and the returned feature representatives when using hierarchical clustering.

Used features	Mean extraction time in seconds	Family classification accuracy	Runtime regression R^2-score
All features	4930.7602	0.9712 ± 0.0070	0.5364 ± 0.0639
Only top 4 features	1.1348	0.9433 ± 0.0118	0.5216 ± 0.0578
Only top 3 features	0.5230	0.9246 ± 0.0130	0.5051 ± 0.0561
Only features 1 & 4	0.6118	0.7845 ± 0.0224	0.2737 ± 0.0532
Only feature 1	0	0.5686 ± 0.0275	0.0376 ± 0.0388
Only top 4 features + family	1.1348	X	0.5413 ± 0.0557
Only top 3 features + family	0.5230	X	0.5361 ± 0.0605
Only features 1 & 4 + family	0.6118	X	0.4374 ± 0.0616
Only feature 1 + family	0	X	0.3965 ± 0.0785
Top 4 features	0	1.#Nodes	1.#Nodes
	0.5230*	2.MeanDegreeCentrality	2.MeanDegreeCentrality
	0.5230*	3.EntropyDegreeCentrality	3.EntropyDegreeCentrality
	0.6118	4.#ConnectedComponents	4.#ConnectedComponents

Table 4.18: Classification and regression result comparisons using the most important features. Features part of the same distribution feature are marked with *.

to 0.5230 seconds, for an additional performance loss of up to 3.17%. This results in a total performance loss of up to 5.84% compared to using all features. Trying to speed-up even further, which is only possible by exclusively using the number of nodes, is not recommended, as the resulting performance can no longer be called usable.

In many use cases, the instance family is already available as metadata. To find the best possible speed-up for these use cases, we also tested combinations of features to which the instance family was added. The results can also be seen in Table 4.18. Since the prediction of a label that is part of the features (in this case the instance family) is trivial, the corresponding column was marked as uninteresting in the comparison using an "X". The results in the table tell us two things: First, the fact that adding instance family as a feature to the top four features resulted in a performance gain of less than 2 P.P. tells us that most of the information gained by knowing the instance family was already covered by the top four features. This further emphasizes the importance of these features. Second, for the analysed use case that the family is already known, the mean extraction time can also be reduced to 0.5230 seconds, but in this case with a performance comparable to the performance when all features are extracted and used. The performance lags only 0.03 P.P. behind, which is in the range of the variance.

Overall, our results show several examples of how a time-critical application can expect a powerful runtime predictor with a very short mean feature extraction time. Moreover, the analysis of our most important features shows that they are well suited to identify the structural differences between SAT families and contain the majority of the needed information for the runtime regression. The fact that removing the degree centrality or the number of connected components leads to a noticeable loss in performance suggests that both play an important role in identifying the underlying structure. While the former underscores previous research, the correlation of the latter with instance family or runtime prediction has not yet been studied in detail in previous research. This should be explored more thoroughly in future work, as it may provide new insights into SAT problems.

4.8.2 Saving And Loading VIGs

The second way to adjust our pipeline is by introducing an intermediate save and load step for the VIGs. We have kept our pipeline as modular and efficient as possible, while making each step parallelizable. However, if smaller pipeline steps are desired, e.g. because simultaneous VIG creation and feature extraction takes too long or sudden crashes are expected, it is recommended to add an intermediate step of VIG saving and loading. To determine in which format the VIGs should best be saved, we analysed and compared several common graph formats supported by the NetworKit library [36]. Figure 4.10 shows the full list of formats we examined. Note that "NetworkitBinaryGraph" is often simply called "NetworkitBinary", which we will also proceed with.

We first sampled 200 SAT instances and created the corresponding VIGs. We then saved these VIGs in each format listed in Figure 4.10 and loaded them back into memory. We found that DOT stored only the edges for all graphs. An example entry was "79 – 78". Moreover, SNAP should have had the format "u v w", where u and v are the nodes of an edge and w is its weight, according to its documentation [36]. However, the actual format of the file was simply "u v", which also resulted in the loss of edge weights. These are probably bugs that need to be fixed on the library side. Either way, we removed SNAP and DOT from the rest of the

Non Binary Formats:

1. SNAP
2. EdgeList
3. METIS
4. DOT

Binary Formats:

6. ThrillBinary
7. NetworkitBinaryGraph, alt: NetworkitBinary

Figure 4.10: List of compared graph file formats.

Instance	CNF size in MB	GraphML size in MB	Growth in %
#1	15.5	96.4	621.9355
#2	79.6	420	527.6382
#3	26.5	759	2864.151

Table 4.19: Comparison of file size consumption of CNF and GraphML format.

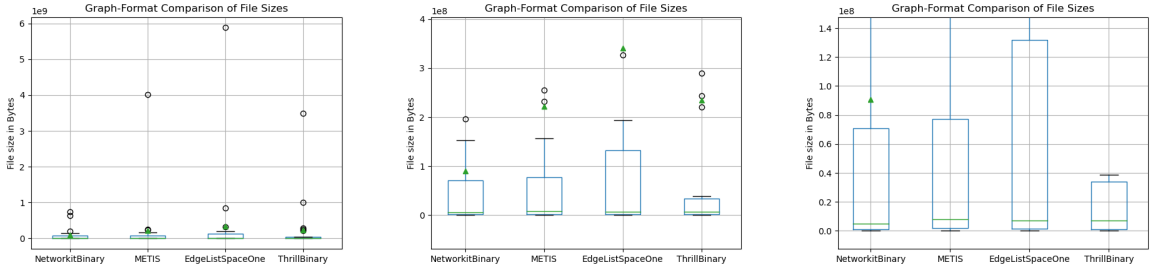
comparison due to the loss of edge weights. GraphML is an XML-based format and seems too large and inefficient for our purposes. After only a few instances, we noticed an exponential space complexity. Table 4.19 shows three example instances and the exponential growth when storing the VIG in GraphML format. The time complexity was similarly inefficient.

This left 4 interesting formats: “EdgeList”, “METIS”, “ThrillBinary” and “NetworkitBinary”. Note that EdgeList is a generic template where the adjacency array of each node is stored on a separate line. We used the concrete format “EdgeListSpaceOne” where spaces are used as separators and the ID of the first node is 0.

Table 4.20 compares the mean and worst case file sizes and time consumption of the four aforementioned graph formats. Here we see that METIS outperforms the non-binary formats in every single metric by a large margin. Similarly, NetworkitBinary outperforms ThrillBinary by a large margin in all four metrics. Additionally, we created box plots of the file size footprint and time usage to compare statistical metrics such as median, standard deviation, and quartiles

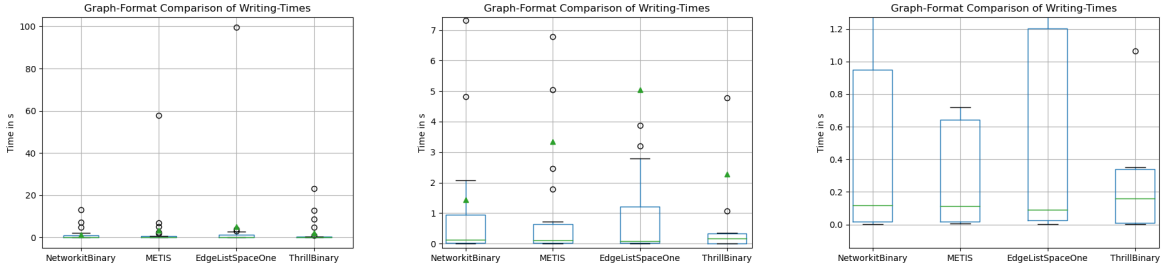
Format	Mean size in MB	Worst case size in MB	Mean time in seconds	Worst case time in seconds
EdgeListSpaceOne	340.3200	5890	5.0400	99.4200
METIS	221.4900	4020	3.3500	57.8470
NetworkitBinary	90.4600	737.7400	1.4300	13.2360
ThrillBinary	234.9100	3480	2.2800	23.1850

Table 4.20: Comparison of the four promising graph formats based on multiple metrics.



(a) No Zoom. (b) Zoomed to compare box plots. (c) Zoomed to compare medians.

Figure 4.11: Box plot diagrams to compare the file sizes of different graph formats.



(a) No Zoom. (b) Zoomed to compare box plots. (c) Zoomed to compare medians.

Figure 4.12: Box plot diagrams to compare the writing times of different graph formats.

to identify outliers. In the file size footprint box plots 4.11, we can see in Subfigure 4.11a that the worst outlier for METIS is at $4e9$, while EdgeListSpaceOne has one at $6e9$. In addition, METIS has a smaller interquartile range and a smaller upper quartile, suggesting lower variance and, on average, smaller file sizes. Looking at the writing-times in Figure 4.12, similar results can be seen. The outlier EdgeListSpaceOne is 40 seconds slower, and the variance of the writing-times is also larger. Since only the medians are more or less the same and METIS outperforms EdgeListSpaceOne as a non-binary graph format in all measured aspects, it thus appears to be the more suitable non-binary graph format. This is further reinforced by METIS' high degree of recognition[49].

Comparing the binary formats NetworkitBinary and ThrillBinary, we can be seen that ThrillBinary's file size outlier of $3.5e9$ is significantly worse than NetworkitBinary's of about $0.8e9$. However, NetworkitBinary has a larger interquartile range and a larger upper quartile, suggesting a higher variance. At the same time, NetworkitBinary appears to have a slightly smaller median. All of these observations can also be made for the writing-times. While the graphs show mixed results, the consistently better measurements in Table 4.20 favour NetworkitBinary. This tips the decision in NetworkitBinary's favour.

In summary, for the graph file formats, we can recommend METIS as a non-binary and NetworKit's own NetworkitBinary format as binary graph format. They are simple formats that have one of the smallest time and space footprint for their corresponding categories, with METIS already being a well-known and widely used format.

4.9 Performance Scaling with Data

In this section, we will evaluate how the performance of our models scales with more data. There is a popular heuristic in the machine learning domain that says: there is no data like more data. We believe that by using a larger data set, we can significantly improve the performance of our models. To this end, we collected a completely new, larger dataset of 5345 instances using the new 2022 SAT competition. We also increased the number of solver runtimes per instance from 18 to 28, while also using new solvers.

We then trained our best performing model, the Random Forest, for all three problems: family classification, runtime regression, and categorical hardness classification. As we established in the previous sections, Random Forests do not require scaled or centred data nor features, so we omitted all preprocessing. Only for the log10_PAR2 label, we had to once again remove all entries with a PAR2 label of 0, since $log_{10}(0)$ is undefined. Additionally, since we found in Section 4.7 that the features capturing eigenvector centrality were an outlier in terms of extraction time, we did not use the five corresponding features for our scale-up test. This left us with a total of 41 features. Our new results therefore represent a lower bound of the possible performance, since eigenvector centrality was among the six most important features (see Table 4.16) and thus contributed positively to the performance of the models. Since the previous hyperparameter optimization did not appear too promising for the Random Forest, we again decided to use the base parameters, which entails better comparability.

Table 4.21 shows a comparison of our overall results, which look very promising: The accuracy for the family classification slightly dropped by 5.68%. Considering the tremendous increase in number of families by 42 families to a total of 130 in the new data set, and therefore variance, the new performance score is still very impressive. This overall increase in difficulty seems to have an even stronger effect when only using the top 4 and top 3 features, where the performance dropped by 11.04% and 12.62%, respectively. While gathering the new data set, we also made sure to keep Cryptography the most common instance family (compare Table 4.22).

Moreover, while the RMSE of the runtime prediction over the PAR2 label marginally stayed the same, the corresponding R^2-score improved by 20.84%. The runtime prediction over the log10_PAR2 label even improved to a R^2-score of 0.7290, which is an improvement of 37.31%. This means, our features now account for more than 72% of the variability of the log10_par-2 time. While our R^2-score is still a little under Li et al.'s R^2-score of 0.8, it was achieved with a harder and more inhomogeneous data set and as a prediction over a solver independent label. When only using the top 4 and top 3 features, without using the family as a feature, the scores increased by 20.65% and 15.82%, respectively. When using only the top 4 and top 3 features with the family metadata included, the score even increases by over 31%. Finally, the accuracy over the 3-means hardness label increased by 18.62% to a new high of 0.8401. This accuracy score suggests a very reliable performance for practical uses.

Interestingly, the RMSE on the log10_PAR2 increased by 167.84% to a total RMSE of 0.7486. The expressiveness of an RMSE depends solely on the comparability of ground truths. For example, if the same model is trained once on a label in grams and once on a label in tons, e.g. 1000000g and 1t, but the data are otherwise completely identical in content, the RMSE would differ by several orders of magnitude.

We therefore compared multiple statistical measures of the log10_PAR2 label over the old and new data set, which can be seen in Table 4.23. Considering that all values are logarithmically

	Old score	New score	Rel. improvement
Accuracy on all_families	0.9712 ± 0.0070	0.9160 ± 0.0136	-5.68%
R^2-score on PAR2	0.5364 ± 0.0639	0.6482 ± 0.0443	20.84%
RMSE on PAR2	2301.7807 ± 166.6166	2293.7586 ± 145.7869	0.35%
R^2-score on log10_PAR2	0.5309 ± 0.1189	0.7290 ± 0.0253	37.31%
RMSE on log10_PAR2	0.2795 ± 0.05	0.7486 ± 0.0394	−167.84%
Accuracy on 3-means	0.7082 ± 0.0208	0.8401 ± 0.0146	18.62%
Acc. on all_families using top 4 features	0.9433 ± 0.0118	0.8392 ± 0.0169	−11.04%
Acc. on all_families using top 3 features	0.9246 ± 0.0130	0.8079 ± 0.0191	−12.62%
R^2-score using top 4 features	0.5216 ± 0.0578	0.6293 ± 0.0345	20.65%
R^2-score using top 3 features	0.5051 ± 0.0561	0.5850 ± 0.0434	15.82%
R^2-score using top 4 features + family	0.5413 ± 0.0557	0.7142 ± 0.0323	31.94%
R^2-score using top 3 features + family	0.5361 ± 0.0605	0.7041 ± 0.0343	31.33%

Table 4.21: Comparison of scores over small (old) and large (new) data set.

scaled (to the base of 10), we can see that the new min is now two orders of magnitude smaller and the standard deviation is larger by one order of magnitude. That changes the range of possible values from around four to five orders of magnitudes to about six to seven. As the values can now be potentially a lot smaller, and therefore, vary a lot more, this could be a possible explanation for the larger RMSE. Since the ground truth seems to differ by quite a bit, we cannot easily compare these two values and cannot necessarily conclude that the new RMSE value is worse than the old value.

We have summarized the new performance results in Table 4.24. Since the RMSE seems to not be comparable between the new and old data set, we did not include it in the overview. Overall, our results show that our features, combined with a base Random Forest model, can be used to predict the instance family. Moreover, using our own PAR2 and log10_PAR2 labels, they can predict the expected runtime of a SAT instance. With the 3-means label, we also created a categorical measure for hardness, which our models can predict successfully. Finally, the overall significant performance gains resulting from the larger data set underline our hypothesis that our model can be further improved with more data. This is especially true given the huge increase in variance and inhomogeneity of the new data set.

Measure	Old family label	New family label
Count	2327	5345
Unique	88	130
Most frequent	Cryptography	Cryptography

Table 4.22: Comparison of the family label for the old and new data set.

Measure	Old log10_PAR2	New log10_PAR2
Count	2327	5345
Mean	3.6151	2.4592
Std	0.4122	1.4420
Min	-0.7899	-2.4362
25%	3.3455	1.4720
50%	3.7366	2.8174
75%	3.9624	3.7305
Max	4.0131	4.0000

Table 4.23: Comparison of the log10_PAR2 label for the old and new data set.

Problem	Label	Performance score
Classification	all_families	0.9160 ± 0.0136
Regression	PAR2	0.6482 ± 0.0443
Regression	log10_PAR2	0.7290 ± 0.0253
Classification	3-means	0.8401 ± 0.0146
Classification using top 4 features	all_families	0.8392 ± 0.0169
Classification using top 3 features	all_families	0.8079 ± 0.0191
Regression using top 4 features	PAR2	0.6293 ± 0.0345
Regression using top 3 features	PAR2	0.5850 ± 0.0434
Regression using top 4 features + family	PAR2	0.7142 ± 0.0323
Regression using top 3 features + family	PAR2	0.7041 ± 0.0343

Table 4.24: All results over larger data set with 5345 instances. Classification problems are listed with their accuracy, regression problems are listed with their R^2-score.

5 Discussion and Future Works

In this chapter, we will discuss our results and point out aspects that can be explored in future work.

5.1 Used Features

In this section, we will briefly discuss our feature choice and how combining them with other features seems promising and should be investigated in future work. The goal of our work was to predict the runtime and hardness of SAT instances. To this end, we first decided to train a classifier for the instance family, with a new set of features. To determine our own feature set, we focused on graph properties such as community, centrality, and clustering properties. We looked at prior work and combined features that were suspected or shown to correlate with instance family and solver runtime with features that had not been analysed yet in the SAT context.

After several iterations of feature selection, extraction, and evaluation, we came up with a total of 46 useful features (see Figure 4.1 in Section 4.2). Among them were five features that modelled the eigenvector centrality of graph nodes. We removed these for the upscaled data set as they were an outlier in terms of extraction time. This resulted in a set of 41 usable features. Although both sets perform very well, we recommend the first feature set only for use cases where feature extraction time is not a concern. Moreover, while the feature sets also perform excellently on their own, combining them with other working feature sets, e.g., the HCS feature set of Li et al., looks promising and should be further explored. In particular, since our feature sets are mainly focused on graph properties of the VIG, combining them with features based on other SAT instance representations would be of great interest.

5.2 Created Labels

In this section, we will briefly discuss our created labels and identify where alternative creation methods should be considered. After extracting features from the data sets, we needed to find fitting labels. For our continuous labels, we computed the average PAR2 value over 18 solver runtimes per instance (compare Section 4.4). This introduced much needed variance into the label, addressing the timeout bias, and accounting for the problem that there is no one best SAT solver for all instances. As a result, our labels should be much more practical than of many related works that use, for example, the runtime of a single modern solver. We believe this to be a good approach for labels that represent the running time and hardness of an instance, independent of a particular solver.

We then used the KNN algorithm as a practical solution for the dynamic categorization of our continuous runtime labels. To determine the optimal number of clusters, we used the

Elbow method on the WCSS graph (compare Figure 4.3). The resulting labels are a good way to categorically estimate the solver-independent hardness of an instance. Moreover, they allow classifiers to be deployed for use cases where specific runtimes are not required nor desired. For example, applications could skip instances classified as "hard" or allocate more resources to solving them. Such a classifier could also be implemented as a preprocessing step for portfolio solvers to determine and execute the best solver for each instance. Creating the categorical labels using the KNN algorithm seemed to be the most intuitive and sensible way to us. However, there are many other options and methods for converting continuous runtime labels into categorical labels that could be explored in future works.

5.3 Model Comparison

In this section, we will briefly discuss our evaluation and comparison of machine learning models. In addition, we will also discuss our model selection and identify points where further research is encouraged. After extracting the features and creating suitable labels, we trained and evaluated several machine learning models, including Support Vector Machines, Multilayer Perceptrons, and Random Forests. We compared the models with base parameters as well as with hyperparameters that we optimized using Random Search. The focus of our work was not to find the best possible configuration for each model, but to identify fundamentally suitable parameters that would tell us what kind of performance improvement we could expect through hyperparameter optimization. We believe that Random Search was the appropriate choice, as the search space was too large for an exhaustive Grid Search.

However, even after optimizing hyperparameters, all other models performed worse than the base model Random Forest. This result seems logical since Random Forests are suited for the relatively high number of categorical and numerical features, the multi-class problems, and the highly inhomogeneous and difficult data set. They also require less preprocessing (e.g., feature or data scaling) and have a simple structure. All of these leave less room for error, making them easier to train and interpret than the compared machine learning models.

At the same time, the Random Forest showed only a marginal performance gain from optimized hyperparameters on two out of three labels, which was accompanied by a significantly longer training time. This is probably because for simple and popular models, such as the Random Forest, the base parameters of the Scikit-learn library are often already very good. We therefore continued to work with the base Random Forest model as it outperformed the other models and omitting the optimization step saves precious time. It also provides better comparability with related work. But especially for much larger data sets with hundreds of thousands of instances, more complex machine learning models such as deep neural networks with more extensive hyperparameter optimization should be explored. Moreover, we only used Random Search to find better hyperparameter configurations. While it is unlikely with 200 iterations combined with 3-fold cross-validation, it is possible that the tested parameters were still far off from fitting parameters. Additional iterations and a Grid Search near the found optima could provide a solution for this potential problem.

We evaluated all our models using a 10-fold cross-validation, as this introduces less bias and variance than a traditional split between training and testing, especially for smaller data sets. Moreover, since all data samples are used once to test the inference, there is no issue regarding

representativeness of the test set. K-fold cross-validation also helps to detect overfitting. Thus, based on our evaluation of the results and due to the inhomogeneous and reasonably large data set, overfitting of our models seems very unlikely. Overall, the performance of our models looks promising:

Our family classifier achieved an accuracy of over 0.97, suggesting that our features do indeed detect and predict underlying structural differences between SAT families. In addition, our runtime regression and hardness classification achieved a score of up to 0.5364 and 0.7082, respectively. The performance and robustness of our runtime models demonstrate that the use of our features enables the prediction of the runtime and hardness of SAT instances, thus successfully answering one of our research questions. At the same time, our labels suggest a possible answer to what sensible hardness labels for a SAT instance could look like.

5.4 Most Important Features

In this section, we will briefly discuss the four most important features of our models and what can be inferred from them. We identified the four most important features in terms of their significance for predicting the family of the instance (classification) or the runtime (regression). This in turn allowed us later to minimize the time consumption of our pipeline. To determine the feature importance, we could not use the regular permutation nor the tree-based methods, as our features were correlated. To deal with the correlated features, we clustered them hierarchically based on the Spearman rank-order and were then able to determine the four most important features: #Nodes, MeanDegreeCentrality, EntropyDegreeCentrality, and #ConnectedComponents.

The number of nodes is likely used by the predictor to scale the problem correctly. MeanDegreeCentrality and EntropyDegreeCentrality are both features that seek to capture the degree centrality of nodes in the VIG. A correlation of this degree centrality to the instance family and instance runtime has already been detected in related work and is further underlined by our results. Finally, the fact that the number of connected components is also one of the top four features for all labels, suggests a correlation with instance family as well as instance runtime and hardness. However, it is important to note that, this correlation has not been studied in detail in previous research yet. Accordingly, thorough analysis of this possible correlation in future work could provide new insights into the structure of SAT problems.

It is noteworthy, that the four most important features that account for most of the prediction are exactly the same for the family classification and the runtime regression. This result suggests that the structural measures that determine the difference between instance families also measure the expected runtime. Since we used the average PAR2 runtime of numerous different solvers for our runtime labels, this observation suggests that our PAR2 label and the resulting categorical 3-means label are an appropriate measure of instance runtime and hardness. These results also lead to the assumption that the structural measures are directly correlated with instance runtime. All of these hypotheses should be explored in greater empirical and theoretical detail in future work.

When training our family classification and runtime regression models with only the top four features, their performance decreased by less than 3% compared to using all features. This indicates that only a few parameters likely play a role in the overall prediction. Furthermore,

adding the instance family as a feature to the four most important features only marginally increases the performance of the model. This suggests that our four most important features alone already capture the instance family information very well.

5.5 Pipeline Adjustments

5.5.1 Minimal Time Overhead

In this subsection, we will briefly discuss how the pipeline can be adjusted to minimize the time overhead for practical purposes. We determined the minimal pipeline time overhead for a performant predictor. By leaving out pre-processing and only extracting the aforementioned top four features, we could reduce the feature extraction time to 1.13 seconds, while keeping more than 97% of the predictors' performance (compared to using all features). If desired, the feature extraction can be reduced even further to only the top three features, which reduces the extraction time to 0.5230 seconds, while still keeping more than 94% of the predictors' performance. A further reduction of features leads to additional performance losses of up to 56.38%. This is no longer an acceptable performance and is thus not a practical option. Furthermore, the fact that removing the centrality features causes such a significant performance loss highlights previous research in that centrality has a strong correlation with instance family and runtime.

When the instance family is available as metadata, the performance of the reduced feature sets can be noticeably improved. If only the three most important features are used and combined with the family metadata, the average extraction time remains at the previously mentioned 0.5230 seconds, but the performance is then comparable to the regression when using all 46 available features. With or without the instance family metadata, this makes the pipeline short enough for time-critical or interactive applications.

Although preprocessing CNFs has the potential to speed up the overall process by simplifying and reducing the formulas [50], we omitted it because the expected time required for preprocessing outweighed the resulting speed-up in our case. It is still worth investigating whether, and by how much, pre-processing can improve the model's performance in general.

5.5.2 Intermediate Pipeline Step

In this subsection, we briefly discuss how the pipeline can be adapted to introduce a graph-saving intermediate step for practical purposes. Overall, we have designed our pipeline to be as modular and efficient as possible, and made each step parallelizable at the same time. However, if pauses or shorter pipeline steps are desired, e.g. because VIG creation and feature extraction consecutively take too long or sudden crashes are expected, we recommend adding an intermediate step to save the VIGs directly after their creation and load them when needed. For the graph file format, we can recommend METIS as a non-binary format and NetworKit's own NetworkitBinary format as a binary format, as they have one of the smallest time and memory footprints for their respective categories. In addition, METIS is already a well-known and widely used format that is easily managed [49, 36].

5.6 Our Results Compared to Current Research

In this section, we will discuss our experimental setup and results and compare them to current research. Although the performance of our model seems lower than that of the Li et al. model, we need to consider the differences in experimental structure: Li et al. used a set of 10869 instances from only five classes and preprocessed all of them with the MiniSAT preprocessor [8]. Moreover, they used and predicted only the runtime of the MapleSAT solver. Therefore, the predicted runtime is only a measure of hardness for this particular solver. Finally, their R^2 score for the model evaluated on the cryptography instances, which are notoriously difficult and hard to predict, was 0.48. Thus, the high R^2 score is mainly due to the predictions of the other 4 categories.

In contrast, we attempted to create a label that is a more general measure of hardness by averaging the PAR2 runtime across numerous solvers. Additionally, Cryptography is the most abundant family (see Table 4.2) in our data set, making it a very difficult data set to predict. Moreover, we used very inhomogeneous data sets with up to 130 different SAT families. In addition, Section 4.9 shows that the performance of our model scales very well with the amount of data. By increasing the size of our data set from about 2400 to 5350 instances, which is still less than half of Li et al.'s data set, we saw performance gains of up to 37.31%. It resulted in new runtime regression and hardness classification performance scores of up to 0.7486 and 0.8401, respectively. Evidently, with a similarly sized data set, we would achieve a more competitive performance on a harder and more inhomogeneous data set while having a solver-independent prediction.

Comparing the performance on the logarithmic scale with the results of Hutter et al. shows that although we used less than a quarter of the data points of Hutter et al. we achieved an RMSE value of 0.2795, which is a significant improvement over their average RMSE value of 0.40. However, we must keep in mind that our problem space, which spans four to five orders of magnitude, is smaller than that of Hutter et al. which spans nearly six orders of magnitude. For our larger data set, whose problem space spans six to seven orders of magnitude, our RMSE is actually higher than that of Hutter et al. Comparing the RMSEs of different papers and drawing a conclusion is therefore very difficult.

In short, while some of our performance scores are lower than those of comparable work, our features, and models provide clear added value. First, we can expect significantly higher performance from our models for a comparable data set size (compare Section 4.9). Future work should compare the performance of our models with related work on a similar data set size. In addition, it should investigate the limit to which performance can be further improved by increasing the size of the data set.

Second, our predictions are much more general and practical: comparable work has mostly focused on single to few solvers with very homogeneous data sets consisting of few instance families. Moreover, these works generally neglected the time consumption of their pipeline. In contrast, we used data sets consisting of up to 130 different instance families and created labels using the runtimes of up to 28 different solvers to ensure high diversity and generalizability. Finally, Cryptography is the largest instance family in our data set and given the aforementioned performance, this underlines that our model can also handle difficult instances very well. For the sake of practicality, future work should compare the performance of models from current research on common difficult and inhomogeneous datasets.

5.7 Model in Practice

In this section, we will briefly discuss potential applications for our results. Ultimately, our work has focused on finding a set of features and a model that together are as practical, efficient, and robust as possible for runtime and hardness prediction of SAT instances. Putting this into practice should be the next logical step in future work. There are a myriad of application areas and uses in which such a predictor could prove useful. These could include integration into real-time applications, such as interactive configurators. These configurators could respond to difficult instances by changing the SAT solver, allocating more resources, or in the worst case, skipping the instance. This way, the timeout, which often occurs only after 3 to 5 minutes of continuous computation, can be prevented, and the user can be given feedback immediately. Such a predictor could also be implemented as a preprocessing step in a portfolio solver. The portfolio solver could then select the best possible SAT solver for the instance based on the predicted family and runtime.

6 Conclusion

In this work, we investigated how to assess and predict the runtime and hardness of SAT instances using machine learning. To this end, we first collected an inhomogeneous set of SAT instances and generated VIGs from each CNF formula. We then determined and extracted a set of graph-based features whose focus was on centrality, clustering, and community features. To solve SAT instances efficiently, modern solvers exploit the underlying structures. To check whether our features recognize and capture these structures, we first trained and compared machine learning models using the SAT instance family as label. The results confirmed that our features capture underlying structures very well.

To have a solver-independent and practical measure of instance hardness, we averaged the PAR2 runtime of many solvers. On the resulting labels, we achieved a R^2-score of up to 0.7290. This means that at least 72.90% of the variability of the runtime is accounted for by our chosen features. In some use cases, concrete runtimes are not of interest. To enable categorical prediction for these, we ran the KNN algorithm on the previously computed PAR2 labels. On the resulting label, we achieved an accuracy of up to 84.01%, suggesting reliable performance in practice.

We also identified the four most important features of our models that accounted for most of the prediction. Among them was the number of connected components, suggesting a correlation with instance family and runtime, which has not been studied thoroughly before in this context. Therefore, close investigation of this result may provide new insights into the structure of SAT problems.

Even though some of our performance results are lower than those of comparable work, our results offer clear added value. First, we can expect significantly higher performance from our models with a comparable data size. To support our hypothesis, we tested the same model on a larger and even more inhomogeneous data set. On this data set, we saw performance gains of up to 37.31%. However, we also found that the RMSE for one of the runtime regressions increased by a noticeable amount. We suspect that the increase in RMSE is due to the two orders of magnitude larger problem space in the larger data set.

Second, our predictions are much more practical and general: In contrast to related work, we used data sets consisting of numerous instance families, and created labels using the runtimes of many different solvers to ensure high generalizability and diversity. Moreover, Cryptography is the largest instance family in our data set, highlighting that our model handles difficult instances as well.

Finally, we also determined how to minimize the time consumption of our pipeline. By omitting preprocessing and extracting only the top three features, the extraction time can be reduced to 0.5230 seconds while retaining more than 94% of the predictor's performance. If the instance family is available as metadata, it can be added to the three features to achieve performance comparable to that achieved using all features.

Bibliography

[1] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[2] Koen Claessen et al. "SAT-solving in practice, with a tutorial example from supervisory control". In: *Discrete Event Dynamic Systems* 19.4 (2009), pp. 495–524.

[3] Tracy Larrabee. "Test pattern generation using Boolean satisfiability". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.1 (1992), pp. 4–15.

[4] JOM Silva and Karem A Sakallah. "Robust search algorithms for test pattern generation". In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE. 1997, pp. 152–161.

[5] Wolfgang Kunz and Dhiraj K Pradhan. "Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.9 (1994), pp. 1143–1158.

[6] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[7] Mikoláš Janota. "SAT solving in interactive configuration". PhD thesis. University College Dublin, 2010.

[8] Chunxiao Li et al. "On the Hierarchical Community Structure of Practical Boolean Formulas". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2021, pp. 359–376.

[9] Frank Hutter et al. "Algorithm runtime prediction: Methods & evaluation". In: *Artificial Intelligence* 206 (2014), pp. 79–111.

[10] Amal Nair. "Beginner's Guide To K-Means Clustering". In: *Analytics India Magazine* (Aug. 2021). URL: https://analyticsindiamag.com/beginners-guide-to-k-means-clustering.

[11] Rimsha Maredia. "Analysis of Google Play Store Data set and predict the popularity of an app on Google Play Store". In: (June 2020).

[12] Johar Ashfaque Aatqb and Amer Iqbal. "Introduction to Support Vector Machines and Kernel Methods". In: (Apr. 2019).

[13] Gareth James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[14] Tasniem Nasser Alyahya, Mohamed El Bachir Menai, and Hassan Mathkour. "On the Structure of the Boolean Satisfiability Problem: A Survey". In: *ACM Computing Surveys (CSUR)* 55.3 (2022), pp. 1–34.

[15] Linton C Freeman. "Centrality in social networks conceptual clarification". In: *Social networks* 1.3 (1978), pp. 215–239.

[16] Linton C Freeman. "A set of measures of centrality based on betweenness". In: *Sociometry* (1977), pp. 35–41.

[17] Akrati Saxena and Sudarshan Iyengar. "Centrality measures in complex networks: A survey". In: *arXiv preprint arXiv:2011.07190* (2020).

[18] Claudio Rocchini. *Hue scale representing node betweenness on a graph*. Published on Wikimedia.org. [Online; accessed 30. Aug. 2022]. Apr. 2007. URL: https://commons.wikimedia.org/wiki/File:Graph_betweenness.svg.

[19] M. E. J. Newman and M. Girvan. "Finding and evaluating community structure in networks". In: *Phys. Rev. E* 69 (2 Feb. 2004), p. 026113. DOI: 10.1103/PhysRevE.69.026113. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[20] M. E. J. Newman. "Fast algorithm for detecting community structure in networks". In: *Phys. Rev. E* 69 (6 June 2004), p. 066133. DOI: 10.1103/PhysRevE.69.066133. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.066133.

[21] $j_h am3$. *A sketch of a small network displaying community structure, with three groups of nodes with dense internal connections and sparser connections between groups*. Published on Wikimedia.org. [Online; accessed 30. Aug. 2022]. Oct. 2011. URL: https://commons.wikimedia.org/wiki/File:Network_Community_Structure.svg.

[22] Falk Schreiber. "Characteristic Path Length". In: *Encyclopedia of Systems Biology*. Ed. by Werner Dubitzky et al. New York, NY: Springer New York, 2013, pp. 395–395. ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_1460. URL: https://doi.org/10.1007/978-1-4419-9863-7_1460.

[23] Toby Walsh et al. "Search in a small world". In: *Ijcai*. Vol. 99. Citeseer. 1999, pp. 1172–1177.

[24] Duncan J Watts and Steven H Strogatz. "Collective dynamics of 'small-world'networks". In: *nature* 393.6684 (1998), pp. 440–442.

[25] Andrea Lancichinetti et al. "Characterizing the community structure of complex networks". In: *PloS one* 5.8 (2010), e11976.

[26] *Node assortativity coefficients and correlation measures — NetworkX Notebooks*. [Online; accessed 1. Sep. 2022]. July 2022. URL: https://networkx.org/nx-guides/content/algorithms/assortativity/correlation.html.

[27] Soumya C Kambhampati and Thomas Liu. "Phase transition and network structure in realistic SAT problems". In: *Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013.

[28] George Katsirelos and Laurent Simon. "Eigenvector centrality in industrial SAT instances". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 348–356.

[29] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. "The community structure of SAT formulas". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2012, pp. 410–423.

[30] Zack Newsham et al. "SATGraf: Visualizing the evolution of SAT formula structure in solvers". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2015, pp. 62–70.

[31] Zack Newsham et al. "Impact of community structure on SAT solver performance". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2014, pp. 252–268.

[32] Xiao Fan Wang and Guanrong Chen. "Complex networks: small-world, scale-free and beyond". In: *IEEE circuits and systems magazine* 3.1 (2003), pp. 6–20.

[33] Lin Xu et al. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of artificial intelligence research* 32 (2008), pp. 565–606.

[34] Ryan Williams, Carla P Gomes, and Bart Selman. "Backdoors to typical case complexity". In: *IJCAI*. Vol. 3. 2003, pp. 1173–1178.

[35] Shai Haim and Toby Walsh. "Online estimation of SAT solving runtime". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2008, pp. 133–138.

[36] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. "NetworKit: An Interactive Tool Suite for High-Performance Network Analysis". In: *CoRR* abs/1403.3005 (2014). arXiv: 1403.3005. URL: http://arxiv.org/abs/1403.3005.

[37] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[38] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[39] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[40] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[41] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[42] *DIMACS CNF - Varisat Manual*. [Online; accessed 23. Sep. 2022]. June 2019. URL: https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html.

[43] Rena Nainggolan et al. "Improved the Performance of the K-Means Cluster Using the Sum of Squared Error (SSE) optimized by using the Elbow Method". In: *Journal of Physics: Conference Series* 1361.1 (Nov. 2019), p. 012015. DOI: 10.1088/1742-6596/1361/1/012015. URL: https://doi.org/10.1088/1742-6596/1361/1/012015.

[44] Leo Breiman. "Bagging predictors". In: *Machine learning* 24.2 (1996), pp. 123–140.

[45] Petro Liashchynskyi and Pavlo Liashchynskyi. "Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS". In: *CoRR* abs/1912.06059 (2019). arXiv: 1912.06059. URL: http://arxiv.org/abs/1912.06059.

[46] Sima Jamali and David Mitchell. "Improving SAT Solver Performance with Structure-based Preferential Bumping." In: *GCAI*. 2017, pp. 175–187.

[47] *Random forest vs SVM*. [Online; accessed 18. Oct. 2022]. Oct. 2022. URL: https://www.thekerneltrip.com/statistics/random-forest-vs-svm.

[48] *Permutation Importance with Multicollinear or Correlated Features*. [Online; accessed 15. Aug. 2022]. Aug. 2022. URL: https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance_multicollinear.html.

[49] George Karypis and Vipin Kumar. "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices". In: (1997).

[50] Niklas Eén and Armin Biere. "Effective preprocessing in SAT through variable and clause elimination". In: *International conference on theory and applications of satisfiability testing*. Springer. 2005, pp. 61–75.