Universiteit Leiden
Opleiding Informatica & Economie

Fully Automatic Machine Learning: Hyper-parameter Optimization and Model Selection with Scikit-learn

Rachelle Blok

02/09/15

Siegfried Nijssen
Erik Schultes

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Fully Automatic Machine Learning: Hyper-parameter Optimization and Model Selection with Scikit-learn

Rachelle Blok

Abstract

Finding the machine learning algorithm that works the best on a given data set is not an easy task: not only are there many algorithms in the literature to choose from, each algorithm also has parameters that need to be set. For this study, we participated in the first round of a machine learning challenge. The optimization problem of selecting the best algorithm and setting the optimal hyper-parameters is investigated. We show that this problem can be addressed by creating a Python program that automatically selects the best algorithm. Different preprocessing methods and algorithms are evaluated. The hyper-parameters are optimized using a random search method. The performance of our model is tested on 20 data sets coming from different domains. Each data set spans a range of difficulties, including different tasks, missing values and different types of attributes.

Contents

Chapter 1

Introduction

Machine learning studies algorithms that can learn from data and make predictions on data. By providing example inputs to an algorithm, this algorithm can learn from these examples and build a model. This model can then be used to make predictions on data that was not seen before. The advantage of this model is that it is not explicitly programmed. The model adapts automatically when new data is added.

The purpose of machine learning is to transform raw data into meaningful information by using algorithms. Suppose we have a data set that contains information about patients in a hospital. This data set is split into two types of attributes: descriptive attributes and one or more target attributes. The descriptive attributes are used to predict the target attribute(s). For example descriptive attributes can be gender and age. Based on these attributes, we want to predict if a patient will survive. This attribute *survive* is called the target attribute. In order to predict the target attribute *survive*, a model should be trained on a data set where the target attribute is known. Historical data can be used for this purpose. This is called *supervised machine learning* because we train on a data set where the outcome is known. Many different algorithms can be used to train on the data set, for example linear models and support vector machines. Subsequently, the target attribute *survive* should be predicted as well as possible on a data set that was not seen before. The target attribute can take on two values: 1 for survive and 0 for die. For each patient this value is predicted. This is called *classification*. We classify the patients into two groups: patients that will survive and patients that will die. The performance of each algorithm is measured by a scoring metric. For example, the performance is measured by the amount of patients where the predicted value and the real target value are compared and the predicted value was correct. For each algorithm the performance can vary. The goal is often to select the algorithm that performs best, in other words the algorithm that predicts the target attribute as well as possible. This optimization problem is called *model selection*.

Machine learning has become an increasingly important research field as it can play a role in many real-world applications. A few domains in which machine learning is applied are robotics, speech recognition and computer vision. For example, for robotics machine learning is used to participate in a competition which involves robot driving in a desert. The robot is able to detect distant objects. For speech recognition, machine learning is used to train a system to recognize speech.

The process of collecting a data set to predicting a target attribute is called the data modeling chain. This process involves the following steps:

1. formalizing a question into a modeling approach: what do we want to know and how are we going to achieve this
2. selecting appropriate data: which data are we going to use
3. designing a model: which algorithm are we going to use
4. fitting the model to data (training)
5. making predictions on new data (testing)
6. interpreting the results

It takes a lot of effort for data scientists to design an appropriate model for model selection. A logical thought is to write a program that automatically selects the best algorithm based on characteristics of the data set. Automating the full modeling chain is difficult. However, model fitting can and should be completely automated [2]. Automating the design of a model and making predictions can be automated as well. For this study, such automatic machine learning will be investigated.

Chapter 2

Research question

Many tools are available for machine learning purposes, but it still is a complex task to choose the right learning algorithm for a particular data set. This problem is called *model selection*. Model selection is a hard problem, as the search space over algorithms and model parameters is large. Every algorithm has parameters that can be set to different values. These parameters are called *hyper-parameters*. For example, for the K-Nearest Neighbours algorithm a certain amount of neighbours needs to be chosen in order to run the algorithm. The problem of identifying a good value for hyper-parameters is called *hyper-parameter optimization*. As there is a growing need for statistical data analysis by non-experts, model selection and hyper-parameter optimization should be simplified and automated as much as possible. A lot of research was conducted about model selection and hyper-parameter optimization separately. Surprisingly, the combination of the two optimization problems has not yet been investigated thoroughly. The hyper-parameter optimization method will be discussed in chapter 5.3. The suggested model selection strategy will be discussed in chapter 6.1.

For this study model selection and hyper-parameter optimization will be combined in order to choose the best algorithm automatically for a given data set. The goal is to find the perfect black box eliminating the human in the loop. Also, preprocessing methods should be evaluated for each data set and algorithm separately. In machine learning, preprocessing methods are used to transform a data set. For example, when a data set contains categorical attributes such as music genre which can take the values pop, rock and classic, you may want to convert these attributes to binary values. This method will ensure that the data set can be handled by more algorithms. Another example is to scale numerical attributes to values between 0 and 1. This can be useful for algorithms where distance is important, for example K-Nearest Neighbours. The used preprocessing methods for this study are discussed in chapter 5.2.

The goal of this study is to develop a program that automatically selects the algorithm with the highest accuracy on a given data set. There is only a limited time budget available to search for the best algorithm. Because of this limited time budget, some smart strategies should be applied in order to find the best algorithm as quickly as possible. These strategies will be discussed in this study.

Chapter 3

Related work

The Auto-WEKA project is the first project that combines both problems of model selection and hyper-parameter optimization into one search space. This project shows that this optimization problem can be solved by a fully automated approach. All 39 WEKA classification algorithms are considered. For this project the open source package WEKA is used, which was written in Java. This package has some limitations, for example it does not take scalability into account. Also, optimizing the preprocessing methods is not investigated during this project [3].

Another project which combines model selection and hyper-parameter optimization in one search space is Hyperopt-Sklearn. This project also takes preprocessing methods into account. Hyperopt-Sklearn is a Python package for automatic algorithm configuration of machine learning algorithms provided by Scikit-learn. This project uses a library called Hyperopt, which can be used for serial and parallel optimization over search spaces. Hyperopt is used to describe a search space over possible configurations of Scikit-learn components, including preprocessing and classification modules. The search space used in the project includes six preprocessing methods and seven classification algorithms. The hyper-parameters of the preprocessing methods as well as the classification algorithms are optimized. The algorithms used for optimization are random search, annealing and TPE [8].

Lastly, a study by the National Institute of Astrophysics, Optics and Electronics [10], considers the use of Particle Swarm Optimization (PSO) for full model selection. In this study, full model selection is defined as follows: “given a pool of preprocessing methods, feature selection and learning algorithms, to select the combination of these that obtains the lowest classification error for a given data set; the task also includes the selection of hyper-parameters for the considered methods.”

Many researches were done about hyper-parameter optimization. Different strategies were explored to optimize hyper-parameters. Two common used strategies are grid search and random search. Both strategies will be discussed further in chapter 5.3. Other strategies are exhaustive enumeration, hill-climbing, beam search, genetic algorithms, experimental design approaches, sequential parameter optimization, racing algorithms and combinations of fractional experimental design and local search [7].

For model selection many strategies were explored in the literature. One example is to use Hoeffding Races. The idea of this algorithm is to first let all algorithms participate in a race. At each point in the algorithm, a random point from the test set is selected. The error at that point for all algorithms is calculated. The algorithms with the worst error are eliminated from the race. The Hoeffding algorithm is repeated until there is one algorithm left. In this way, algorithms that perform badly are dropped early in the race [9].

Chapter 4

CodaLab challenge

4.1 General information

For this study we participate in an online challenge[1]. It is a supervised machine learning challenge in which data present themselves as input-output pairs $\{x, y\}$ and the goal is to solve regression and classification problems. The objective of this challenge is to find, for a given combination of data sets, task, metric of evaluation, available computational time, the combination of methods and hyper-parameter setting that is best suited. The difficulty of the challenge is that each data set spans a range of difficulties, including missing values, sparsity, different tasks, different scoring metrics and categorical variables. These difficulties will be discussed further in chapter 4.4.

4.1.1 Rounds and phases

The challenge consists of 6 rounds and in each round 5 data sets are made available. Some data sets may be recycled from other challenges. Only the Master round includes completely new data.

The 6 rounds are:

1. **Preparation**: different tasks, sparse and full matrices, some missing values, some categorical variables, small and large number of features.

2. **Novice**: only binary classification problems, no missing data, no categorical variables, balanced classes, moderate number of features, sparse and full matrices, noise.

[1]https://www.codalab.org/competitions/2321

3. **Intermediate**: multi-class and binary classification problems, unbalanced classes, some missing values, some categorical variables, large number of features.

4. **Advanced**: all types of classification problems, up to 300,000 features.

5. **Expert**: classification and regression problems, all difficulties.

6. **Master**: classification and regression problems, all difficulties, completely new data sets.

Each round consists of three different phases:

- **Tweakathon**: the program can be improved by tweaking the code on the data sets and running the program on your own system. In this phase the program does not need to be constrained by a time budget. Code and results can be submitted. The leaderboard scores are based on the validation set results.

- **Final**: no code submission. The program is evaluated on the test set. The results of the Tweakathon phase are revealed. To participate in this phase it is required to submit code in the Tweakathon phase.

- **AutoML**: submitted code is "blind tested" in limited time on the CodaLab platform. The program is evaluated on the test set which contains data that is never seen before. To participate in this phase it is required to submit code in the Tweakathon phase of the previous round.

Difficulties are introduced from round to round, cumulating all the difficulties from the previous rounds plus new ones.

For this study we used the 5 data sets from the first round (Preparation). We did not participate online as a consequence of postponed deadline dates. Hence we did not compare the performance of our program with other participants of the challenge.

4.1.2 Code submission

Participants can submit their code and results. The code will be executed on the servers of the challenge and will be evaluated on unknown data sets. The scores of all participants will be shown on a leaderboard. The participants with the highest scores can win prizes. The total prize pool is 30,000 USD [2].

4.2 Python program

A skeleton program is provided by the challenge that is written in Python, using the machine learning library Scikit-learn[2]. The sample code uses ensemble algorithms which improve over time by adding more base learners. The hyper-parameters are not optimized and also the sample code does not include preprocessing methods.

4.2.1 Scikit-learn

Scikit-learn is a Python library that supports many machine learning application areas. It provides implementations of many well known machine learing algorithms. Its interface is easy to use and tightly integrated with the Python language. It differs from other machine learning tools for the following reasons: it is distributed under the BSD license, it incorporates compiled code for efficiency, it depends only on numpy and scipy and it focuses on imperative programming. It also incorporates the C++ libraries LibSVM and LibLinear that provide reference implementations of Support Vector Machines (SVMs) and generalized linear models.

The library does not only include algorithms. It also includes cross validation methods and scoring metrics. Additionally, it supports model selection methods. For example `GridSearchCV` can be used to try different values for hyper-parameters on a specified parameter grid and determine what works best by maximizing a score [4].

4.2.2 Skeleton program

The skeleton program consists of a few separate files:

- `data_io.py`: a file that organizes the input and output of data. This program is also used to create, move and delete directories.
- `data_manager.py`: this file loads and saves data easily with a cache and generates a directory in which each key is a feature (name, format, number of features, etc). It reads information that is specified in the information file.
- `data_converter.py`: a file that performs various data conversions, for example converting binary targets to a numeric vector.
- `models.py`: the constructor selects a model based on the data information passed as argument. It fits the model on a data set and predicts the target variables.

[2]http://scikit-learn.org/stable/

- `run.py`: the main program. It keeps track of the time spent, reads the data set, executes algorithms on a data set that are specified in `models.py` and creates prediction files.
- `libscores.py`: the calculations of the scoring metrics are specified in this file.
- `score.py`: it reads the prediction files created by `run.py`, compares the prediction file with the solution file and calculates a score based on the specified scoring metric.

4.3 Tasks

Each data set has a different task. Four different tasks can be distinguished: regression, binary classification, multi-class classification and multi-label classification. These tasks will be discussed below. Also, a special approach to deal with multi-class and multi-label classification will be discussed. This approach is called the *One-vs-all classifier*.

4.3.1 Regression

For regression we want to predict a numeric value. Assume that we want to predict the total income of a person in one year. We have a data set that contains the attributes age and education. Based on these attributes we want to predict the value of the target attribute *income*. One possible regression model is the linear regression model. In this model, the relationship is represented by

$$I = a + bA + cE$$

where I is income, A is age and E is education. The parameter a reflects a default income, parameter b reflects the effect of age on income and parameter c reflects the effect of education on income. We need to determine the values of these parameters. This is an example of a linear regression model, but many other types of models are possible.

The task of regression is to determine an estimate of these parameters, based on the information contained in the data set. Many lines can be drawn by the formula. The goal is to find the line that fits the instances contained in the data set as well as possible. Regression chooses the line where the sum of squared errors is at a minimum. The error can be defined as the vertical distance between the real target value and the predicted target value of an instance. In this example this is a linear line, but the line can be of any form.

In figure 4.1 below an example of a regression line is shown. The vertical lines represent the distances between the instances and the regression line.

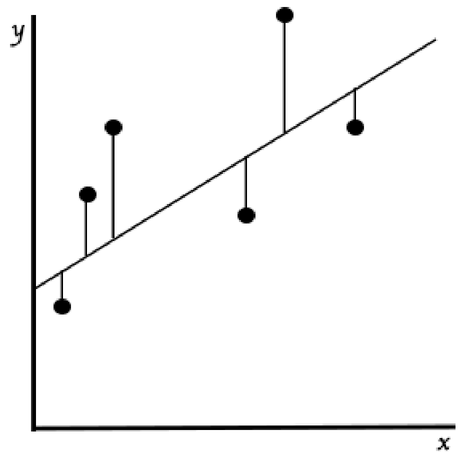

Figure 4.1: Regression: a linear regression model

4.3.2 Binary classification

Each instance in a data set can belong to one out of two classes. For example, when we want to predict if an email contains spam, the target value can be 0 (no spam) or 1 (spam).

4.3.3 Multi-class classification

In multi-class classification, an instance belongs to one of possibly more than two classes. The instance can be assigned to one and only one label at the same time. For example, when we want to predict what type of fruit an instance belongs to (pear, apple, banana), there are more than two classes that can be distinguished. An instance can be a pear or apple, but not both at the same time.

In figure 4.2 below a comparison of binary and multi-class classification is shown. With binary classification, an instance can be a circle or cross. With multi-class classification, an instance can be a square, cross or triangle.[3]

4.3.4 Multi-label classification

An instance can belong to more than one class at the same time. An instance is assigned to a set of labels. For example, a document can have more than one

[3]Source: http://www.holehouse.org/mlclass/06_Logistic_Regression.html

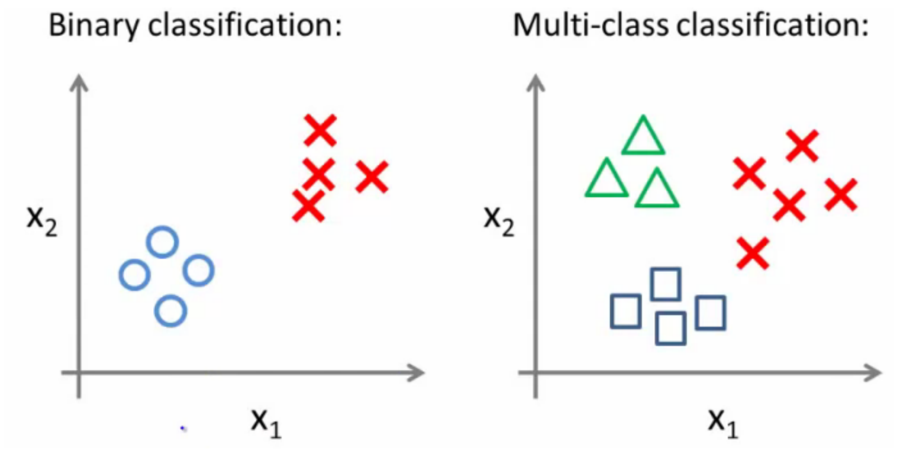

Figure 4.2: Binary classification vs. Multi-class classification

subject. It can be assigned to religion, politics and history at the same time.

In figure 4.3 below an example of multi-label classification is shown. It shows that an instance can belong to more than one music genre at the same time.[4]

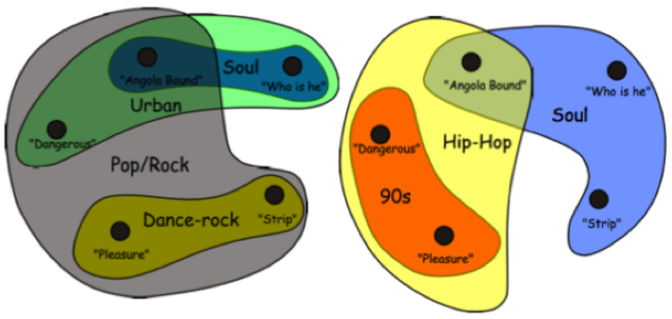

Figure 4.3: Multi-label classification

4.3.5 One-vs-all classifier

For multi-class and multi-label classification, an approach to solve these problems is to consider the problem as a collection of binary classification problems. This method is called *One-vs-all*. N different binary classifiers are trained. Each classifier is trained to distinguish the instance in a single class from the instances in all remaining classes. To classify a new instance, the N classifiers are run. For

[4]Source: http://musicmachinery.com/2009/10/28/ismir-oral-session-5-tags/

multi-class classification, the classifier that produces the largest value is chosen [11]. This value represents the probability that an instance belongs to a certain class and is produced by many binary classification algorithms. For multi-label classification, every classifier predicts a different target attribute. The approach is also called *Binary Relevance*. Binary relevance is often criticized for ignoring the correlation between labels. Also, the training complexity is linear to the number of labels k [12].

In our study we use this approach for all algorithms with a multi-label classification task. Multi-class classification is inherently supported by most algorithms.

In Scikit-learn a module called *One-Vs-The-Rest* implements the One-vs-all method. This module is used to deal with multi-label classification problems.

4.4 Data sets

In total 20 data sets were used for this study. 5 data sets were used from the first round of the challenge and the other 15 data sets were collected from other resources.

4.4.1 Data sets from the challenge

The data sets that are provided by the challenge come from a wide variety of domains including medical diagnosis, speech recognition, text classification and protein structure prediction. The 5 data sets that were used in the Preparation phase are:

- **Adult**: the prediction task is to determine whether a person makes over 50K a year, based on census data from 1994.
- **Cadata**: LibSVM dataset from Statlib house prices (1997).
- **Digits**: a data set carved out of the MNIST data set from LeCun, Cortes and Burges (1998) of handwritten digits, extracted from a larger benchmark collected by the US National Institute of Standards and Technologies.
- **Dorothea**: data set prepared for the NIPS 2003 feature selection challenge from one of the KDD Cup 2001 tasks.
- **Newsgroups**: the 20 Newsgroups data set (1997) used for text categorization.

4.4.2 Data sets from other resources

For this study 15 other data sets were collected from the UCI Machine Learning repository [5], OpenML [6] and for multi-label data sets the KEEL data set repository [7] was used.

In order to compare the results of our program with other results, most data sets that were gathered from the UCI Machine Learning repository are also available on OpenML. This online platform collects and organizes results from many data sets online. A large amount of experiments are run for different data sets using algorithms from the WEKA package. This makes it possible to compare our own results with results of others.

The 15 data sets that were collected are:

- **Electricity**: data set collected from the Australian New South Wales Electricity Market. The class label identifies the change of the price relative to a moving average of the last 24 hours.
- **Nursery**: a data set that was derived from a hierarchical decision model originally developed to rank applications for nursery schools.
- **Kropt**: a game data set collected from OpenML.
- **Spambase**: a collection of spam e-mails and non-spam e-mails.
- **Splice**: splice junctions are points on a DNA sequence at which 'superfluous' DNA is removed during the process of protein creation in higher organisms. The problem posed in this data set is to recognize, given a sequence of DNA, the boundaries between exons (the parts of the DNA sequence retained after splicing) and introns (the parts of the DNA sequence that are spliced out).
- **Kin8nm**: a data set concerned with the forward kinematics of an 8 link robot arm. This is a variant of the original data set, which is known to be highly non-linear and medium noisy.
- **Wind**: daily average wind speeds for 1961 - 1978 at 12 synoptic meteorological stations in the Republic of Ireland.
- **Stock**: a data set that contains daily stock prices from January 1988 through October 1991, for ten aerospace companies.

[5]https://archive.ics.uci.edu/ml/datasets.html
[6]http://www.openml.org
[7]http://sci2s.ugr.es/keel/multilabel.php

- **Houses**: a data set containing information from all the block groups in California from the 1990 Census. A block group on average includes 1425.5 individuals living in a geographically compact area.
- **Sick**: a data set containing thyroid disease records.
- **Diabetes**: a Pima Indians Diabetes Database. All patients are females at least 21 years old of Pima Indian heritage.
- **Hepatitis**: a data set containing information about hepatitis. The objective is to predict if a patient is going to die.
- **Yeast**: a data set that contains information about a set of yeast cells. The task is to determine the localization site of each cell.
- **Mediamill**: contains data from a generic video indexing problem where each item can belong to one or more classes.
- **Emotions**: a music data set that contains 72 music features for 593 songs that are categorized into one or more out of 6 classes of emotions.

Transforming data sets
Some data sets that were collected from these resources had to be transformed before they could be used. For example, all nominal attributes that were represented as a string were converted to binary values to make them work on all algorithms. Unique identifiers were deleted from the data sets. Also, some data sets were represented in the ARFF format that is used for WEKA. They had to be converted to the format that was used in the skeleton program of the challenge. For some data sets, the order of the records was shuffled to remove possible patterns in the data.

4.4.3 Training, validation and test set

The data sets that were provided by the challenge were split into separate sets. Each data set contains two separate sets: a training set and validation set. The training set was used for training a model. We used *10-fold cross validation* to train the model. This approach splits the training set into ten folds. Nine folds are used for training and one fold is used for testing. This is repeated ten times, whereby the folds are rotated. So in the end each fold is used for testing. Using this approach, a score on the training set was calculated for each algorithm.

In order to evaluate as many algorithms as possible, a timeout was set on the 10-fold cross validation process for each algorithm. When the training set was large, a sample of the training set was used to train a model. These adjustments are further explained in chapter 6.1.

In figure 4.4 below the *k*-fold cross validation method is illustrated [8]. In this case, *k* is 10. The white folds are used for training and the gray folds are used for testing.

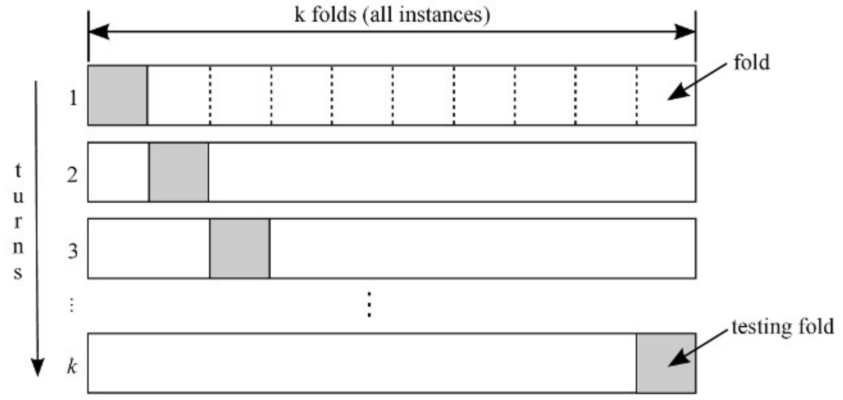

Figure 4.4: *k*-fold cross validation

The training and validation sets were labeled, meaning the solutions are known. In this way a score could be calculated for data that was not seen before. We used the validation set to calculate the score for the best algorithm. The best algorithm was chosen based on the score of the 10-fold cross validation approach on the training set.

For the 15 data sets that were collected from other resources, only one main labeled data set was available (i.e. it was not split into separate training and validation sets). We have partitioned the data set into two mutually exclusive subsets called a training and validation set. This method is called the *holdout method.* A common ratio is to use 2/3 of the data as the training set and the remaining 1/3 as the validation set. We have slightly used this guideline and have split this data set into 60% for training and 40% for validation. Randomly, records from the data set were added to the training and validation set. The holdout method is a pessimistic estimator as only a portion of the data is used for training. Fewer validation set instances means that the confidence interval for the accuracy will be wider [5]. The holdout method was combined with the 10-fold cross validation method for the training set as described above.

[8]Source: http://cse3521.artifice.cc/classification-evaluation.html

We have not used the test sets that were provided by the challenge. These test sets were unlabeled so they could not be used to evaluate an algorithm.

4.4.4 Information file

Each data set that was provided by the challenge contains an information file. In this file the characteristics of the data set are specified. When there is no information file available, the Python program will extract the characteristics from the data itself. For the data sets that were collected from other resources, we have created the information file manually.

The information file contains the following information:

- **Task** (task): the task for which the problem is solved. This can be one out of 4 values: regression, binary classification, multi-class classification or multi-label classification.
- **Target type** (target_type): the type of the target variable. This can be numerical or binary.
- **Feature type** (feat_type): the type of the attribute variables. This can be numerical, categorical, binary or mixed. The type for each variable can be specified separately in a feature type file.
- **Metric** (metric): the metric used to calculate the score. For all available scoring metrics, see chapter 4.5.
- **Number of features** (feat_num): the number of attribute variables, so the number of columns in the data matrix.
- **Number of target values** (target_num): number of values to predict. For regression this is always 1.
- **Number of labels for classification problems** (label_num): number of labels a target variable can accept. For regression this is always 1.
- **Number of training examples** (train_num)
- **Number of validation set examples** (valid_num)
- **Number of test set examples** (test_num)
- **Existence of categorical variables** (has_categorical): the presence or absence of categorical variables in the data set. It can take values 1 (for yes) and 0 (for no).

- **Existence of missing values** (has_missing): the presence or absence of missing values in the data set. It can take values 1 (for yes) and 0 (for no).
- **The representation of the data matrix** (is_sparse): if the data set is in sparse or dense format. It can take values 1 (for sparse) and 0 (for dense).
- **Time budget in seconds** (time_budget)

For the data sets from other resources, the time budget was set to 400 seconds.

See Appendix A for a full overview of all data sets and their characteristics.

4.5 Scoring metrics

The skeleton program provided by the challenge implements some scoring metrics that are used to evaluate the performance of the algorithms on a data set. Each data set has its own metric, which is specified in the information file. The score is computed based on the prediction files that predict the target values on the training and validation set. For regression problems this target value is a continuous numeric coefficient y_i. For binary classification problems, the target value is a single binary indicator y_i in $\{0, 1\}$. For multi-class and multi-label classification problems, the target value is a vector of binary indicators $[y_{ik}]$ in $\{0, 1\}$. Multi-label problems are treated as multiple binary classification problems and are evaluated by the average of the scores of each binary classification problem.

In the challenge, the scores that are computed are normalized. Because of the fact that this normalization resulted in some negative scores for multi-label data sets, we decided to remove the normalization for all scoring metrics.

The available scoring metrics are:

- **R2**: used for regression problems. It is the slope of the regression line and is calculated as follows: R2 = 1 - MSE / VAR where MSE is the mean squared error and VAR is the variance. The MSE is calculated with the formula

$$MSE = \sum_{n=1}^{N} \frac{(y_{s,n} - y_{p,n})^2}{N}$$

where N is the number of instances in the data set, $y_{s,n}$ is the solution for every instance and $y_{p,n}$ is the prediction for every instance. The VAR is calculated with the formula

$$VAR = \sum_{n=1}^{N} \frac{(y_{s,n} - \overline{y_s})^2}{N}$$

where $\overline{y_s}$ is the mean of the solution over all instances.

- **BAC**: balanced accuracy, which is the average of sensitivity and specificity. This is also called class-wise accuracy. Sensitivity is calculated with the formula

  $$sensitivity = \frac{\max\left(1e - 15, tp\right)}{\max\left(1e - 15, (tp + fn)\right)}$$

  where tp are the true positives and fn are the false negatives. Specificity is calculated with the formula

  $$specificity = \frac{\max\left(1e - 15, tn\right)}{\max\left(1e - 15, (tn + fp)\right)}$$

  where tn are the true negatives and fp are the false positives.

  If the task is multi-class classification, then the BAC is equal to sensitivity. For every class i a BAC is calculated (BAC_i). For the other tasks, the BAC is calculated with the formula

  $$BAC = 0.5 * (sensitivity + specificity)$$

  The BAC is averaged over all classes with the formula

  $$BAC = \sum_{c=1}^{C} \frac{BAC_c}{C}$$

  for every class c.

  For multi-label problems, the class-wise accuracy is averaged over all classes. For multi-class classification problems, the predictions are binarized by selecting the class with the largest score before computing the class-wise accuracy.

- **AUC**: area under the ROC curve which is used for ranking and binary classification problems. The ROC curve shows the sensitivity on the vertical axis and (1 - specificity) on the horizontal axis.

  ROC curves rely on prediction algorithms that return a probability that an example belongs to a class; they are constructed by iterating over all possible thresholds on this probability. By setting a threshold on a probability, we can split the classes into negatively and positively defined classes. Based on this threshold we decide if an instance is classified as positive or negative.

The AUC score is defined by calculating the area under the ROC curve. The average AUC score over all classes is computed with the formula

$$AUC = 2 * \overline{AUC} - 1$$

where $\overline{AUC}$ is

$$\overline{AUC} = \sum_{i=c}^{C} \frac{AUC_c}{C}$$

for every class c.

- **F1**: the harmonic mean of precision and recall. Precision is calculated with the formula

  $$precision = \frac{tp}{(tp + fp)}$$

  where tp are the true positives and fp are the false positives. Recall is calculated as

  $$recall = \frac{tp}{(tp + fn)}$$

  where fn are the false negatives.

  The F1-score is calculated with the formula:

  $$F1 = \frac{(recall * precision)}{arithmetic\_mean}$$

  The $arithmetic\_mean$ is calculated with the formula

  $$arithmetic\_mean = 0.5 * \max(1e - 15, (recall + precision))$$

  Then the average over all classes is taken. So

  $$F1 = \sum_{c=1}^{C} \frac{F1_c}{C}$$

  for every class c.

- **PAC**: probabilistic accuracy that is calculated with the formula

  $$PAC = \sum_{i=c}^{C} \frac{\exp(-CE)}{C}$$

  for every class c where CE is the cross entropy, also called log loss. This loss function is used in logistic regression. It is defined as the negative log-likelihood of the true labels given a probabilistic classifier's predictions.

4.6 Time budget

For every data set from the challenge a time budget is specified in the information file. This is to make sure that all participants have the same amount of time available to execute their program. When uploading the code to the CodaLab platform, the execution time is limited to the time budget. When running the code on an own platform, the execution time does not need to be constrained by the time budget. The unit of the time budget is seconds. For the 15 data sets that we collected from other resources we set the time budget to 400 seconds.

Chapter 5

Existing methods in Scikit-learn

In this chapter we describe existing methods implemented in Scikit-learn that we have used for our study. Different types of algorithms, preprocessing methods and hyper-parameter optimization methods are discussed.

5.1 Portfolio of algorithms

For this study different types of algorithms were selected. They are all implemented in the Scikit-learn library. In this chapter, each algorithm will be described briefly.

5.1.1 Linear models

Different types of linear models were evaluated. A linear model depends on its parameters in a linear way. The models are described below.

- *Ordinary Least Squares*: this algorithm fits a linear model on a data set, also called *Linear Regression*. The goal is to minimize the sum of squares of the instances in the data set. The linear model consists of coefficients $w = (w_1, ..., w_p)$. The values of these coefficients should be optimized in order to find the best linear model.

- *Ridge Regression*: this algorithm improves the Ordinary Least Squares regression. It is an iterative process that shrinks coefficients, so the model becomes more stable which leads to a better prediction performance. It forces parameters of a model to not take too large values. However, it does not set coefficients to 0 so it does not give an easier interpretable model than the model that is produced by Ordinary Least Squares [31].

- *Lasso*: this algorithm also tries to improve the Ordinary Least Squares regression. It shrinks some coefficients to improve the prediction accuracy. It also sets some coefficients to 0 in order to provide an interpretable model. So in fact it does both continuous shrinkage and automatic variable selection simultaneously. It produces a sparse model [31].

- *Elastic Net*: this algorithm tries to improve the performance of the Lasso algorithm. Like Lasso, it does continuous shrinkage and automatic variable selection simultaneously. It can also select groups of correlated attributes. This algorithm can be useful for sparse models and when there are multiple attributes which are correlated with one another [32].

- *Logistic Regression*: this algorithm is a linear model for classification rather than regression. A single outcome variable $Y_i(i = 1, ..., n)$ follows a Bernoulli probability function. It takes on the value 1 with probability π_i and 0 with probability $1 - \pi_i$. Then π_i varies over the observations as an inverse logistic function of a vector x_i [33]. This logistic function can be formulated as

$$\pi_i = \frac{1}{1 + e^{-x_i \beta}}$$

- *Bayesian Ridge Regression*: this algorithm is similar to the classical *Ridge Regression* algorithm. The difference is that this algorithm estimates a probabilistic model of the regression problem. In a Bayesian model, the hyper-parameters are given prior distributions. The values of the hyper-parameters are estimated from the data [34].

5.1.2 Ensemble methods

An ensemble method uses a set of classifiers whose individual results are combined in some way (for example by a majority vote) to classify new examples. Ensembles often perform much better than individual classifiers. This is because individual algorithms can get stuck in local optima. When using an ensemble method, the algorithm runs from different starting points and this may provide a better approximation and reduce the chance to get stuck. Also, an algorithm can be viewed as searching a space of hypotheses and the goal is to identify the best hypothesis. When the training set is too small, the algorithm can find many hypotheses that all have the same accuracy on the training data. In order to find the best hypothesis, an ensemble method can average the votes of all individual classifiers and reduce the risk of choosing the wrong classifier [27].

Two types of ensemble methods can be distinguished: **Bagging** and **Boosting**. They both rely on resampling techniques to obtain different training sets for each of the classifiers.

For the Bagging method each classifier is trained on a random redistribution of the training set. Each training set is generated by randomly drawing, with replacement, N examples. N is the size of the original training set. It is possible that some instances are repeated and some instances are left out in the resulting training set. Each classifier is generated with a different random sampling of the training set. The predictions of each classifier are averaged. Bagging is effective on algorithms where small changes in the training set result in large changes in predictions. An example of such an algorithm is a decision tree [28].

For the Boosting method a series of classifiers is produced. The generated training set of each classifier is based on the performance of the earlier classifier(s) in the series. Instances that were incorrectly predicted by previous classifiers are chosen more often than instances that were correctly predicted. In this way more weight is given to incorrectly predicted instances and the goal is to produce new classifiers which improve the prediction accuracy of these instances [28].

Examples of Bagging methods are the *Bagging Tree* classifier and *Random Forests* classifier. Examples of Boosting methods are the *AdaBoost* classifier and the *Stochastic Gradient Boosting* classifier. For the *Bagging Tree* and *AdaBoost* classifier an estimator can be specified. The default estimator is a decision tree. We have focused on this default estimator as decision trees work well with ensemble methods and they have a fast training speed.

Different types of ensemble methods were used:

- *Bagging Trees*: several training sets are created by resampling with replacement. For each training set a tree is grown to maximum size and the result is averaged. In this way the variance component of the output error is reduced. The part of the data that is used for the samples is called the “in-bag” data and the part that is left out is called the “out-of-bag” data. When the individual trees differ a lot from each other, then averaging will improve the results [29].
- *Random Forests*: a large number of trees are grown (500 to 2000). Each tree is grown with a randomized subset of predictors. To find the best split at each node, a randomly chosen subset of the total number of predictors is chosen. The trees are grown to maximum size and the result is averaged across all trees. The “out-of-bag” samples can be used to calculate an unbiased error rate. Because of the large number of trees, overfitting is nearly impossible [29].

- *Stochastic Gradient Boosting*: many trees are built from "pseudo"-residuals, which is the gradient of the loss function of the previous tree. A tree is built from a random subsample of the data at each iteration without replacement. Advantages of using part of the data are that it improves the computation speed and the prediction accuracy. It also prevents overfitting. Gradient boosting is resistant to outliers [30].

- *AdaBoost*: the AdaBoost classifier maintains a set of weights over the training instances. In each iteration the weighted error should be minimized. The weights are updated in each iteration. A heavier weight is placed on the instances that were misclassified and a lighter weight is placed on the instances that were correctly classified. In this way, subsequent iterations focus on wrongly classified instances and a more difficult learning problem is constructed. The final classifier combines all individual classifiers by a weighted vote. A stronger weight is assigned to the classifiers that have a high accuracy [27].

5.1.3 Support Vector Machines

A Support Vector Machine (SVM) has a robust performance with respect to sparse and noisy data. Therefore it is often used for text categorization and protein function prediction.
When an SVM is used for classification, it finds the optimal hyperplane that separates two classes. It chooses the hyperplane that leaves the largest margin between the samples of the two classes. A parameter called the *kernel* can be set when there is no linear separation possible. The kernel can be linear or non-linear. It can automatically realize a non-linear mapping to a feature space [19].
Classification success of an SVM does not depend on the dimensions of the input space. Therefore an SVM is robust against the *curse of dimensionality.*
For multi-class data sets, the one-against-one approach is used [20].

In figure 5.1 below an example of a hyperplane of an SVM is shown [1].

5.1.4 Stochastic Gradient Descent

A Stochastic Gradient Descent can be used well for large-scale and sparse machine learning problems.

Let the *empirical risk* be an error function to measure the training set performance and we want to minimize this. For a normal *gradient descent*, the empirical

[1]Source: http://docs.opencv.org/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

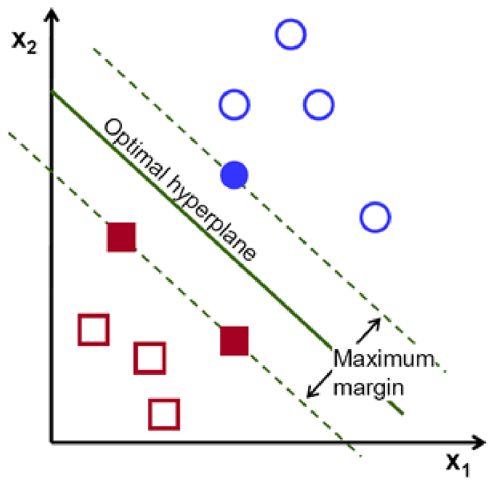

Figure 5.1: Hyperplane of an SVM

risk is minimized by updating weights in each iteration on the basis of the gradient of the empirical risk. This algorithm achieves linear convergence when the initial weight is close enough to the optimum and when the gain is sufficiently small. Stochastic Gradient Descent is a simplification of this algorithm. Instead of computing the gradient of the empirical risk exactly, each iteration estimates the gradient by randomly picking an example. The stochastic algorithm does not need to remember which examples were picked in previous iterations. The convergence speed of Stochastic Gradient Descent is limited by the noisy approximation of the true gradient [26].

The Stochastic Gradient Descent algorithm can be applied to linear classifiers such as SVM and Lasso. For the best results, the data should have zero mean and unit variance.

This algorithm supports multi-class classification by using a one-vs-all approach.

5.1.5 Naive Bayes

The Naive Bayes algorithm assumes that all attributes of the instances are independent of each other given the class. This assumption is often false. However, the Naive Bayes algorithm performs well on classification tasks. It is mostly used for document classification and spam filtering. The algorithm works well when there are a large number of attributes. Because of the independence assumption, the parameters for each attribute can be learned separately. This simplifies learning a lot. A Naive Bayes algorithm runs fast as it requires only one pass through the data if all attributes are discrete. It is robust to irrelevant attributes [24].

Two types of Naive Bayes algorithms were selected:

- *Gaussian Naive Bayes*: for this algorithm it is assumed that numeric attributes obey a Gaussian distribution (also called a normal distribution). This means that within each class the values of numeric attributes are normally distributed. Such a distribution can be written in terms of mean (μ) and standard deviation (σ). The probability that the attributes are Gaussian is computed. The formula used is:

  $$p(X = x|C = c) = g(x; \mu_c, \sigma_c)$$

  where

  $$g(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

  The mean and standard deviation are estimated using maximum likelihood [25].

- *Bernoulli Naive Bayes*: this type of algorithm is appropriate for tasks that have a fixed number of attributes. Instances should be represented by a vector of binary attributes. For example it can be used for text classification where each attribute is a word. The binary values indicate which words occur and do not occur in a document [24].

5.1.6 Nearest Neighbours

The Nearest Neighbours algorithm classifies instances based on the class of their nearest neighbours. When taking more than one neighbour into account, the algorithm is referred to as the k-Nearest Neighbours algorithm. The k-Nearest Neighbours algorithm consists of two stages. First, the nearest neighbours are determined using a distance metric. The distance between the instance and the neighbours is calculated using Euclidean distance. Second, the class of the instance is determined based on the classes of the nearest neighbours. The class that is assigned to the instance is the majority class of the nearest neighbours [21].

5.1.7 Decision Trees

A Decision Tree is capable of breaking down a complex decision-making process into a collection of simpler decisions. Advantages of a Decision Tree are that they are fast and easy to understand. On the other hand, a Decision Tree can cause overfitting. This problem can occur when the depth of the Decision Tree is too large or the tree is too complex. This problem can be solved by methods such as pruning or setting a maximum depth.

A Decision Tree is built by recursive partitioning. The tree starts at the root and an attribute is chosen to split the data on using some criterion. The criterion used here can be the Gini index or the information gain. The process repeats recursively for each child. When the full tree is built, a pruning step is executed which reduces the size of the tree [22].

In figure 5.2 below an example of a Decision Tree is shown. It shows the root, the intermediate nodes and the leaf nodes [23].

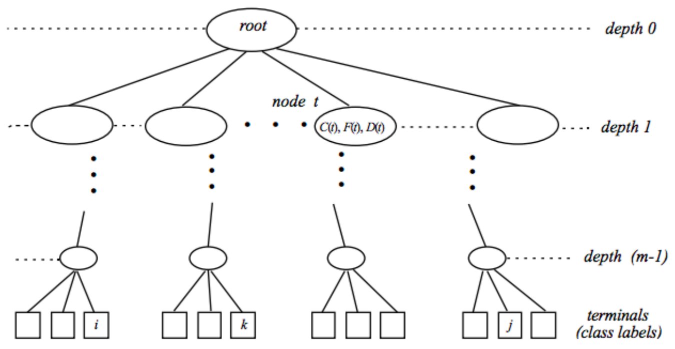

Figure 5.2: Decision Tree

5.2 Preprocessing methods

Many different types of preprocessing methods are available. Applying preprocessing methods on a data set can have different goals:

- to transform a data set. For example, categorical attributes can be converted to binary values.
- to make a data set work on different types of algorithms. For some algorithms it is required that all attributes are numerical.
- to improve the running time of an algorithm. For example, by removing attributes in a data set (dimensionality reduction), the data set will become smaller and the algorithm will run faster.

For this study, some of the preprocessing methods are selected on the basis of properties of the data set. These methods are described in section 5.2.1. For other preprocessing methods it is better to apply them on the basis of the type of algorithm that is evaluated. These methods are discussed in section 5.2.2.

5.2.1 General preprocessing methods

- **Missing values imputation**: it is possible that a data set contains missing values, which are defined as 'NaN'. Many algorithms do not know how to handle these missing values. Therefore they have to be replaced by some other value. There are different strategies to handle missing values. One standard strategy is to delete the entire row or column that contains missing values. The disadvantage is that information will be lost. A better strategy is to impute missing values. Strategies that have been evaluated for this study are:
  - *Replace by zero*: every missing value is replaced by zero.
  - *Replace by mean*: the missing value is replaced by the mean of the column or row to which it belongs. For this study the missing value is replaced by the mean of the column.
- **Dealing with categorical variables**: a data set can contain categorical variables, for example gender. These categorical variables can be represented as integers. For example, female can be represented as 1 and male can be represented as 0. These integers cannot be used directly for the algorithms, as these integers will be interpreted as ordered values. This is often not desired for categorical variables. A method called *One Hot Encoder* can be used to transform the categorical variables. The categorical variables with m possible values will be transformed into m binary features, with only one active.
- **Dimensionality reduction**: when many attributes are present in a data set, it can take a long time to run an algorithm. The goal of dimensionality reduction is to compress the size of the data set. Another goal is to remove the attributes that do not add much information. By removing these noisy attributes, the prediction accuracy of an algorithm can improve. Only the most important attributes will remain. Different strategies for dimensionality reduction were evaluated for this study:
  - *Principal Component Analysis (PCA)*: this method only works for dense matrices. The first principle component should have the largest possible variance. This is computed by choosing a vector where the projection

of the data points on this vector leads to an attribute with the highest possible variance. PCA computes a new attribute for each example by projecting the example on a principal component (a vector). In this way a new data set is created. The second component is computed under the constraint of being orthogonal to the first component. A new data set is then created in the same way. The other components are computed likewise [13].

  PCA works best when data is normalized first. This is because the axis is calculated based on the weights of the attributes. An attribute with a high standard deviation will have a higher weight for the calculation of an axis. If the data is normalized first, all attributes will have the same standard deviation and thus have the same weight.

- *Select K best*: the k features with the highest scores are kept, based on the scoring function chi-squared. The parameter k can be chosen freely. The chi-square test measures dependence between attributes. This test removes the attributes that are most likely to be independent of class. These attributes are irrelevant for classification. This method does not work for regression tasks because of chi-squared.

- *Feature selection KDD cup 2001*: this method only works for binary classification and sparse matrices. It was used by the KDD cup 2001 [17]. First, the algorithm selects the instances in the data set where the target value is 1. For each attribute separately, the number of true positives is calculated. True positives can be defined as where the predicted and the real value are both positive. The k attributes with the most number of true positives will remain. This parameter k can be chosen and is set to 1000 by default.

Random projection

For the above three dimensionality reduction methods we experimented with an algorithm in order to choose the number of attributes to keep. This random projection method is called the *Johnson-Lindenstrauss lemma.* This method estimates the minimum amount of attributes to keep. The number of attributes to keep depends on the size of the data set.

The Johnson-Lindenstrauss lemma shows that a set of n points in high dimensional Euclidean space can be mapped down into an $O(\log n/\epsilon^2)$ dimensional Euclidean space. The distance between any two points changes by only a factor of $(1 \pm \epsilon)$, for any $0 < \epsilon < 1$. So the distances are nearly preserved [18]. The amount of attributes that remain depends on the value of ϵ. This value is set to 0.1 by default.

5.2.2 Preprocessing methods based on algorithm

- **Normalize**: this preprocessing method scales the input vectors to unit norm. A normalized attribute can be described by the following formula:

$$f_n = f/\|f\|$$

  where f_n is the normalized attribute [14].

- **Scaling**:
  - *Standard Scaler*: this method standardizes attributes by removing the mean and scaling to unit variance. The transformed attribute $\bar{x}$ is calculated by the formula
  $$\bar{x} = \frac{x - \mu}{\sigma}$$
  where μ is the sample mean and σ is the sample standard deviation of that attribute [15].
  - *Min Max Scaler*: it standardizes attributes by scaling each attribute to a given range, in this case between 0 and 1. Suppose we have attribute x, the scaled value of x can be calculated by the following formula:
  $$\bar{x} = \frac{x - l}{u - l}$$
  where l is a lower bound and u is an upper bound. The value of $\bar{x}$ results in a value between 0 and 1 [16].

5.3 Hyper-parameter optimization

While grid search is the most used strategy for hyper-parameter optimization, it does not seem to be very efficient. Grid search is described as follows. Suppose we have a set of hyper-parameter variables (e.g., for neural networks it would be the learning rate, the number of hidden units, the strength of weight regularization, etc.). For grid search it is required to choose a set of values for each parameter $(L^{(1)}...L^{(K)})$. Every possible combination of values is tried, so the number of trials in a grid search is $S = \prod_{k=1}^{K} |L^{(k)}|$ elements. This product over K sets makes the search space very large, because the number of joint values grows exponentially with the number of hyper-parameters [6].

Bergstra and Bengio [6] describe another method called random search. This method is more efficient than grid search in highly dimensional spaces. As grid search tries out every possible combination, it carries out many trials that may be irrelevant for a particular data set. Bergstra and Bengio [6] show that random search is able to find models that are as good or better than grid search with a small fraction of the computation time. Random search obtains independent draws from a uniform density from the same configuration space as would be spanned by a regular grid. In contrast to grid search, not all parameter values are tried out, but only a part of the specified parameter distributions is sampled. In this way, random search achieves low effective dimensionality. The number of parameter settings that are tried out can be specified. When using random search, it is easy to expand the search space by adjusting the distribution for each parameter. However, the downside of this method is that it is possible that the best combination of hyper-parameters will not be found. By increasing the number of trials, the chance to find the best combination increases.

The algorithm for random search is described below.

Algorithm 1: RandomSearch(N, θ_o)

Outline of random search in parameter configuration space; θ_{inc} denotes the incumbent parameter configuration, *betterN* compares two configurations based on the first N instances from the training set.

```
   Input  : Number of runs to use for evaluating parameter configurations, $N$;
            initial configuration $\theta_o \in \Theta$.
   Output: Best parameter configuration $\Theta_{inc}$ found.
1  $\theta_{inc} \leftarrow \theta_o$;
2  while not TerminationCriterion() do
3      $\theta \leftarrow$ random $\theta \in \Theta$;
4      if betterN($\theta, \theta_{inc}$) then
5          $\theta_{inc} \leftarrow \theta$;
6  end
7  return $\theta_{inc}$
```

For this study a random search method, which is implemented in Scikit-learn, was used called `RandomizedSearchCV`. For every algorithm separately a parameter grid was created and the number of different trials to run was set. The parameter values can be numerical or categorical. For continuous parameters it is important to define a range of continuous values. For example, for a Stochastic Gradient Descent the learning rate can take on any value between 0 and 1.

The score of each parameter setting is calculated using 10-fold cross validation. The training set is randomly split into 10 mutually exclusive subsets of approximately equal size. The inducer is trained and tested 10 times; each time t $\in$

$\{1, 2, ..., k\}$, it is trained on $D \setminus D_t$ and tested on D_t. The cross-validation score is calculated using the r2 score for regression problems and the f1 score for classification problems [5].

When the training set was large, a random sample of the training set was used for random search. A training set is considered large when the number of training instances is larger than 10,000 or when the number of attributes is larger than 50,000. If the number of training instances is larger than 10,000, a sample of 10% was used. When the number of training instances is smaller than 10,000 but the number of features is larger than 50,000, a sample of 50% was used. These percentages are only applied when at least 5000 instances of the total training set remain. When the number of instances that remain is less than 5000 and when the original number of instances is larger than 5000, a sample of 5000 instances of the training set is taken.

See Appendix B for an overview of all hyper-parameter settings for each algorithm.

Chapter 6

Proposed method

In this chapter we describe the experiments that we carried out on the existing methods described in Chapter 5.

First we will describe our model selection strategy (section 6.1). For some steps in this strategy we have carried out experiments. For example applying preprocessing methods on a data set is part of our strategy. For the 5 data sets that were provided by the first round of the challenge, we did some experiments on preprocessing methods. The purpose of these experiments was to determine which preprocessing methods work best for a given data set or algorithm. This is discussed in section 6.2.
Another part of our strategy is to execute the different algorithms in a particular order. Therefore we carried out an experiment that considers the order in which the algorithms are evaluated for all 20 data sets. As a consequence of a fixed time budget, we want to evaluate the algorithms that run fast and in general provide a high score first. Therefore we have tried to optimize the order of the algorithms. This is discussed in section 6.3.

6.1 Automatic model selection strategy

An obvious method to search through a space of algorithms is to use a brute force approach. This means that for every algorithm the accuracy is calculated and the best one is picked. The time to find the accuracy of a particular algorithm is proportional to the size of the training set *train*. Suppose that the search space consists of a finite number of algorithms *algo*, then the amount of work required is at least $O(train \times algo)$, which is expensive [9]. The amount of work can be even higher as not all algorithms are linear in the size of the data.

If we would create a search space of all algorithms combined with all possible hyper-parameter values for each algorithm, this would take a lot of computational time as the search space will be incredibly large. Alternatively, we can try each algorithm with their default hyper-parameters, calculate the accuracy for each algorithm on the training set and eliminate the algorithms with the lowest scores. We can then continue with the best algorithms and optimize their hyper-parameters. This would save a lot of computational time as the search space would be much smaller by only optimizing the best algorithms. This strategy is implemented for the program of this study. It can be compared to Hoeffding Races in which the worst algorithms are eliminated early in the process [9].

The strategy to determine which algorithm performs best for a given data set can be described as follows:

Step 1: Apply general preprocessing steps based on characteristics of the data set

Step 2: Apply preprocessing steps based on the evaluated algorithm

Step 3: Train and calculate scores of all algorithms in a particular order with their default hyper-parameters on the training set using 10-fold cross validation

Step 4: Save the 3 algorithms with the best scores

Step 5: Use random search to optimize the hyper-parameters of the 3 best algorithms and calculate scores using 10-fold cross validation on the training set

Step 6: Save the best algorithm and calculate the score on the validation set

For steps 1, 2 and 3 we carried out experiments in order to optimize these steps to get the best result. In the next chapter, we will evaluate how well the results of these experiments work on a large number of data sets.

In step 2, we use grid search to determine the best combination of preprocessing steps based on the evaluated algorithm. For every combination that is tried, we set a timeout of 10 seconds.

In step 3, the algorithms are executed in a particular order which is based on the experiments. The set of algorithms tried for a given data set depends on the task. For example, for regression a different set of algorithms will be tried than for binary classification. For some algorithms, the default hyper-parameters were adapted a little bit. See Appendix E for the adapted default hyper-parameters that were used.

In step 3 we set a timeout of 60 seconds for calculating the score on the training set using 10-fold cross validation for each algorithm. We decided to set a timeout because of the fact that we want to try as many algorithms as possible. When it takes too long for an algorithm to run, we terminate the process and try another algorithm.

In step 5 we set a timeout to the remaining time of the time budget specified in the information file. When there is no time left, the process is terminated.

In step 6 we did not set a timeout. The time it takes to calculate the final score on the validation set is not limited.

In steps 3 and 5, the r2 metric is used to calculate the scores on the training set for regression tasks. For the other tasks the f1 metric is used.

In step 3 and 5, if the training set is large a sample of the training set is used. A training set can be defined as large when the number of training instances is larger than 10,000 or when the number of attributes is larger than 50,000. When the training set contains more than 10,000 instances, 10% of the training set is used for training the model. When the number of attributes is larger than 50,000, 50% of the training set is used. Otherwise the whole training set is used. These percentages are only applied when at least 5000 instances of the total training set remain. When the number of instances that remain is less than 5000 and when the original number of instances is larger than 5000, a sample of 5000 instances of the training set is taken. If the original training set contains less than 5000 instances, then the original amount is used.

In step 6, the scoring metric used for calculating the score on the validation set is specified in the information file of the data set (which is in most cases the r2 or the f1 score).

6.2 Experiment 1: Preprocessing methods

For the 5 data sets that were provided by the challenge, we did some experiments on the preprocessing methods. The preprocessing methods are split into two parts:

1. Preprocessing methods that are executed based on characteristics of the data set.
   Three categories can be distinguished:

   - **Missing values**: we compare the performance of *replacing by zero* with *replacing by the mean of the column*. Only the data set *Adult* contains missing values. Therefore we use this data set for this experiment.
   - **Categorical variables**: only the data set *Adult* contains categorical variables. We use *One Hot Encoder* to deal with these variables.
   - **Dimensionality reduction**: we apply a dimensionality reduction strategy when the number of features is equal to or larger than 1400. Only the data sets *Digits*, *Newsgroups* and *Dorothea* have more than 1400 features. Therefore we use these data sets to experiment with different dimensionality reduction methods.
     - *PCA*: this strategy only works for dense data. We evaluate the *Digits* data set for this method. We experiment with two different methods to determine the number of features to keep. The first method is to set the number of features to keep to 4% of the original number of features in the data set. For the second method we use the *Johnson Lindenstrauss lemma* to determine the number of features to keep.
     - *Select K Best*: we use this strategy when the data set is sparse and when the task is multi-class or multi-label classification. We evaluate the *Newsgroups* data set for this method. We experiment with two different methods to select the number of features to keep. The first method sets the number of features to keep to 1000. For the second method we use the *Johnson Lindenstrauss lemma* to determine the number of features to keep.
     - *Feature selection KDD cup 2001*: this method only works for binary classification and sparse matrices. We use the *Dorothea* data set to evaluate this method. We experiment with two different methods to select the number of features to keep. The first method sets the number of features to keep to 1000. For the second method we use the *Johnson Lindenstrauss lemma* to determine the number of features to keep.

2. Preprocessing methods that are executed based on the algorithm that is evaluated.
   For all algorithms, we experiment with the following preprocessing methods:

   - **Normalize**: scale the input vectors to unit norm.

- **Scaling**: we experiment with two different methods for each algorithm: *Standard Scaler* and *Min Max Scaler.*

For the preprocessing methods that are applied based on characteristics of the data set (1) we do the following for all algorithms. We compare the two methods for missing values with each other. We also compare the two methods for determining the number of features to keep for the three dimensionality reduction methods with each other. Based on the data set that is evaluated, we apply these preprocessing methods. For example, for the *Dorothea* data set only the two methods for *Feature selection KDD cup 2001* are applied.

For the preprocessing methods that are based on the algorithm that is evaluated (2), we try different combinations of normalizing and one method of scaling with each other for every data set and every algorithm. These combinations are evaluated together with the applied preprocessing methods based on characteristics of the data set on each algorithm (1). So for both types of preprocessing methods we use the same approach: evaluating the different types of methods for each data set and every algorithm separately. For the *Dorothea* and *Newsgroups* data sets we evaluate combinations of *Normalizing* and *Standard Scaler* with each other on every algorithm. For the other 3 data sets we evaluate combinations of *Normalizing* and *Min Max Scaler* on every algorithm. For this part of the experiment we set the methods for *Select K best* and *Feature selection KDD cup 2001* to 1000 features to keep. For *PCA* we set the number of features to keep to 4%. So, for example for the data set *Newsgroups*, we evaluate different combinations of *Normalizing*, *Standard Scaler* and *Select K best* with 1000 features with each other for every algorithm.

For every experiment we measure the time spent and the score on the validation set. We only use the 5 data sets from the challenge for the experiments. So only these data sets have seen the validation set. The other 15 data sets that are not used for the experiments have not seen the validation set. When the time to execute the experiment takes longer than 10 minutes, the experiment is terminated. We then document the time spent and the score with a '**-**'. Sometimes an error occurs. For example, for the *SGD Classifier* scaling is sometimes necessary. We then document the time spent and score with '**error**'.

For a full overview of the experiments for each data set, see Appendix C.

6.3 Experiment 2: Order of executing the algorithms

Because of the restricted time budget for each data set, we want to execute the algorithms with their default hyper-parameters in a particular order. The goal is to first execute the algorithms that give a high score on the training set and run fast.

For all 20 data sets that we have used for this study, we measure the time it takes to run each algorithm for a specific data set. We also measure the score on the training set. The algorithms that are executed for a particular data set depends on the task of the data set (regression, binary classification, multi-class classification or multi-label classification). Therefore the 20 data sets can be split into four groups (table 6.1).

Regression	**Binary classification**	**Multi-class classification**	**Multi-label classification**
Cadata	Dorothea	Digits	Adult
Kin8nm	Electricity	Newsgroups	Yeast
Wind	Spambase	Nursery	Mediamill
Houses	Sick	Kropt	Emotions
Stock	Diabetes	Splice	
	Hepatitis		

Table 6.1: Data sets per task

6.4 Results of the experiments

6.4.1 Experiment 1.1: Preprocessing methods based on characteristics of the dataset

- **Missing values**: based on the results of the experiments, we can conclude that *replace by the mean of the column* gives better results than *replace by zero*. Therefore we decide to use this strategy.

- **Categorical variables**: if there are categorical variables in the data set, we always apply *One Hot Encoder*.

- **Dimensionality reduction**: for all three methods, setting the number of features to 1000 gives better results than using the *Johnson Lindenstrauss lemma*. Most of the time the *Johnson Lindenstrauss lemma* results in more than 1000 features and therefore the time spent increases a lot. Also, we cannot conclude that this method always improves the score on the validation set. Therefore we decide to set the number of features to keep to 1000 for all three methods.

Of course the results of these experiments can be overfitted on these 5 data sets. Therefore they are not necessarily the best settings for other data sets.

6.4.2 Experiment 1.2: Preprocessing methods based on the algorithm that is evaluated

Based on the experiments, we cannot conclude which preprocessing methods work best for every algorithm. It depends on the data set that is used.

Therefore we decide to implement a search space using *grid search*. For every data set, all combinations of *Normalize*, *Standard Scaler* and *Min Max Scaler* are evaluated for each algorithm. The best combination of preprocessing steps is determined when the algorithms are executed with their default hyper-parameters. We use 10-fold cross validation to calculate a score for every combination on the training set. When the task is regression, we use the r2 metric to assign a score. For all other tasks we use the f1 metric. For every combination that is tried we set a timeout of 10 seconds. The best combination for each algorithm is then saved inside a dictionary. This combination is executed during other steps in the program (during random search and calculating the final score on the validation set).

When the training set is large, a random sample of the training set is used for grid search. A training set is considered large when the number of training instances is larger than 10,000 or when the number of attributes is larger than 50,000. If the number of training instances is larger than 10,000, a sample of 10% is used. When the number of training instances is smaller than 10,000 but the number of features is larger than 50,000, a sample of 50% is used. These percentages are only applied when at least 5000 instances of the total training set remain. When the number of instances that remain is less than 5000 and when the original number of instances is larger than 5000, a sample of 5000 instances of the training set is taken.

6.4.3 Experiment 2: Order of executing the algorithms

Based on the experiment, we created a certain order in which the algorithms are executed for each task separately. We assign ranks to each algorithm to create the order.

Calculation of the order using ranks
We first order the algorithms for each data set by the **score** on the training set. We give each algorithm a rank. Rank 1 is applied to the algorithm with the highest score on the training set. We then compute the average rank of the algorithms for all data sets that belong to a certain task. For example, we compute the average rank of the *Random Forest* classifier for the data sets that belong to the task *binary classification*. We then order the algorithms based on this average rank.

For example, the list of ranks for the *Random Forest* classifier for the data sets that have task *binary classification* looks like this:

[Dorothea: 5; Electricity: 2; Spambase: 8; Sick: 3; Diabetes: 6; Hepatitis: 1;]

The average rank (*avg_rank*) of the *Random Forest* classifier for the task *binary classification* is then

$$avg\_rank = \sum_{d \in Datasets} rank_d$$

where d are the data sets. In this example the average rank is

$$avg\_rank = \frac{5+2+8+3+6+1}{6} = 4,17$$

We order the algorithms based on this average rank.

We repeat the same computations for the **execution time**. We first order the algorithms for each data set by the time spent on the training set. Rank 1 is applied to the algorithm with the shortest execution time. We then compute the average rank of the algorithms for all data sets that belong to a certain task. Lastly, we order the algorithms based on this average rank.

For each task we now have an average rank of each algorithm for the score and execution time. For example, for *binary classification* the list of average ranks for the score is represented by

[GaussianNB: 1,7; BernoulliNB: 2,8; Logistic Regression: 3,3; Decision Tree Classifier: 4,3; KNeighbours Classifier: 4,6]

We then convert these average ranks to absolute ranks. So the list will now look like

(1): [GaussianNB: 1; BernoulliNB: 2; Logistic Regression: 3; Decision Tree Classifier: 4; KNeighbours Classifier: 5]

We do the same for the execution time. For example, the list of absolute ranks for execution time looks like

(2): [Decision Tree Classifier: 1; GaussianNB: 2; KNeighbours Classifier: 3; Logistic Regression: 4; BernoulliNB: 5]

So for each task we now have two lists with absolute ranks (for score and execution time). The next step is to compute the final absolute rank for each algorithm by combining both lists. We first compute the average rank (avg_rank) for each algorithm (alg) by the formula

$$avg\_rank(alg) = \frac{abs\_rank(score) + abs\_rank(time)}{2}$$

By combining (1) and (2) we now get the following list of average ranks:

(3): [GaussianNB: 1,5; BernoulliNB: 3,5; Logistic Regression: 3,5; Decision Tree Classifier: 2,5; KNeighbours Classifier: 4]

We order this list and convert the average ranks to absolute ranks. The final list will look like

(4): [GaussianNB: 1; Decision Tree Classifier: 2; BernoulliNB: 3; Logistic Regression: 4; KNeighbours Classifier: 5;]

List (4) represents the order in which the algorithms will be executed for the task *binary classification*. We repeat the same computations for the other tasks.

For all results of this experiment, see Appendix D.

Chapter 7

Results

We compared the results of our program with three other programs. First we applied some necessary preprocessing steps.

7.1 Preprocessing

When missing values are present in the data set, we imputed these missing values with the mean of the column. We decided to use this strategy because it gave the best results on the *Adult* data set during the experiments on the 5 data sets provided by the challenge. From these 5 data sets, only the *Adult* data set has missing values so we could only evaluate the different strategies on this data set. For the other 15 data sets, the *Sick* and *Hepatitis* data sets have missing values. We decided to use the same strategy for all 3 data sets. In order to be able to compare the performance of our program with the other programs on a fair basis, we keep the strategy for missing values the same for all programs.

7.2 Programs

- **Random Forest program**: a *Random Forest* classifier with the default hyper-parameters is executed. We used three variants of the *Random Forest* classifier. For binary and multi-class classification we used the standard *Random Forest* classifier. For regression we used a *Random Forest* regressor. For multi-label classification, we wrapped the *Random Forest* classifier inside a *One-vs-all classifier*.

- **Decision Tree program**: a *Decision Tree* classifier with the default hyper-parameters is executed. We used three variants of the *Decision Tree* classifier. For binary and multi-class classification we used the standard *Decision Tree* classifier. For regression we used a *Decision Tree* regressor. For multi-label classification, we wrapped the *Decision Tree* classifier inside a *One-vs-all classifier.*

- **Skeleton program**: this program is provided by the challenge and uses ensemble methods. They improve over time by adding more base learners. The base learners increase exponentially, starting with one base learner. They are increased until the time spent is equal to the time budget. The default hyper-parameter settings are used.

  The choice of the ensemble method is made based on this decision tree:

  - If the task is regression, use the *Random Forest* regressor.
  - If categorical features are present in the data set, use the *Random Forest* classifier.
  - If the data set is sparse, use the *Bagging* classifier with *Bernoulli Naive Bayes* as base estimator.
  - In all other cases, use the *Gradient Boosting* classifier.

  In case the task is multi-label classification, we wrap the ensemble method inside a *One-vs-all classifier.*

For all programs we simply train the model on the training set and then calculate a score on the validation set. For training the model we set a timeout. For the 5 data sets provided by the challenge, the timeout is equal to the time budget that is specified in the information file. For the other 15 data sets the timeout is set to 400 seconds. For calculating a score on the validation set we do not set a timeout.

When a timeout occurred on a data set while training the model, we document the score and execution time with **0**. For detailed results of our program and the three competing programs see Appendix F.

7.3 Comparison

7.3.1 Comparison on validation score

When we compare all four programs on validation score, we get the following graph (figure 7.1). On the y-axis the score on the validation set is shown. This score is based on a scoring metric (for example the r2 metric for regression tasks and the f1 metric for other tasks). The scoring metric used is different for every data set and is specified in the information file. On the x-axis the 20 data sets are shown.

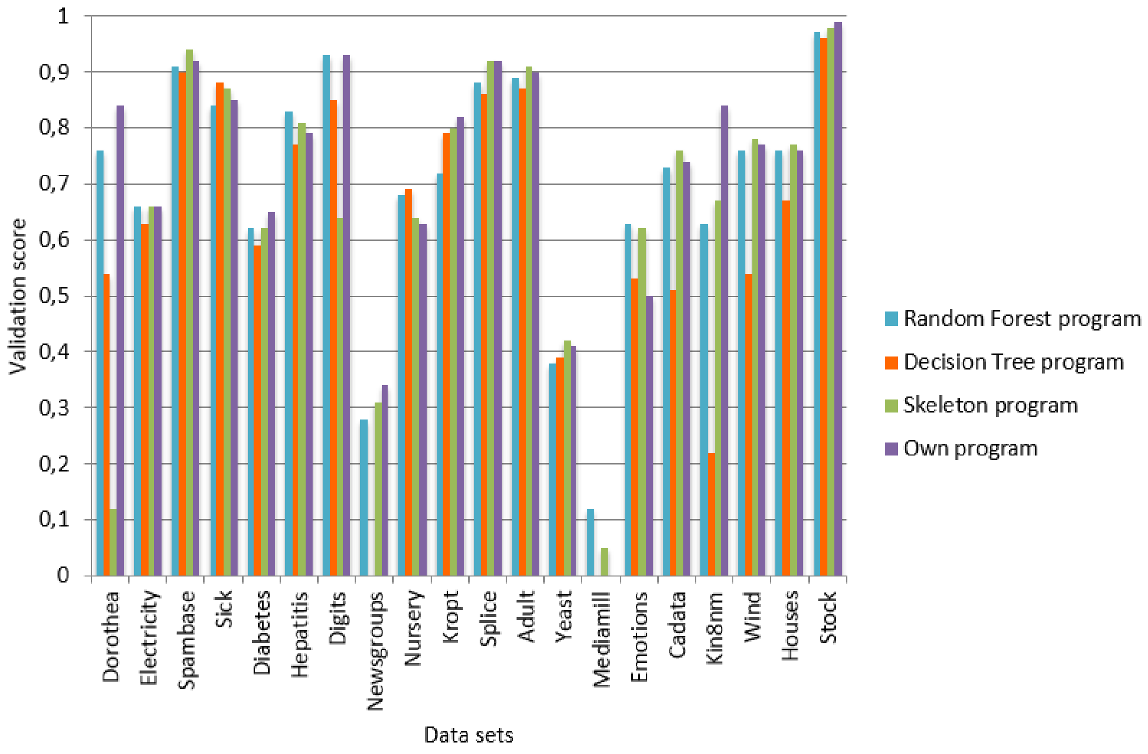

Figure 7.1: Comparison of all programs on validation score

For the data sets *Dorothea*, *Diabetes*, *Newsgroups*, *Kropt*, *Kin8nm* and *Stock* our program produces the highest scores. The algorithms that are chosen for these data sets are respectively the *Random Forest* classifier, *Logistic Regression*, *Random Forest* classifier, *Bagging* classifier, *SVR* and *KNeighbours* regressor. When we analyze the hyper-parameters of the *Random Forest* classifiers for the *Dorothea* and *Newsgroups* data sets (this is the best algorithm for these data sets when using our program) and compare them with the default hyper-parameters that are used for the Random Forest program, the difference is that the number of estimators

is set to 30 for our program. For the Random Forest program the number of estimators is set to 10. Therefore the *Random Forest* classifiers that are chosen as the best algorithm for the *Dorothea* and *Newsgroups* data sets give a better score, because more trees are built so there is a higher chance that a better tree is found. For the data sets *Spambase*, *Adult*, *Yeast*, *Cadata*, *Wind* and *Houses* the skeleton program produces the highest scores. The algorithms that are chosen for these data sets are respectively the *Gradient Boosting* classifier, *Random Forest* classifier, *Gradient Boosting* classifier, *Random Forest* regressor, *Random Forest* regressor and *Random Forest* regressor. For the *Adult* data set the *Random Forest* classifier was also chosen for the Random Forest program and our own program. When we compare the hyper-parameters for all three programs, we see that the number of estimators is set to 10 for the Random Forest program, the number of estimators is set to 30 for our own program and it is set to 128 for the skeleton program. For our program the *Min Max scaler* was applied as preprocessing step. Because of the high number of estimators, the skeleton program produces the highest score. For the *Cadata* data set the *Random Forest* regressor was chosen for our own program and the Random Forest program as well. When we compare the hyper-parameters, we see that the number of estimators is set to 256 for the skeleton program, it is set to 10 for the Random Forest program and it is set to 30 for our own program. For our own program the hyper-parameter "criterion" is set to "mse". We can conclude that for this data set the high number of estimators results in a high score for the skeleton program.
For the data sets *Hepatitis*, *Mediamill* and *Emotions* the Random Forest program produces the best scores. For the data sets *Sick* and *Nursery* the Decision Tree program produces the best scores.
For the data set *Electricity*, the Random Forest program, skeleton program and our own program produce the highest scores. For all three programs the *Random Forest* classifier is chosen. When we compare the *Random Forest* classifier for all three programs, the number of estimators for the skeleton program is set to 1024. For the Random Forest program, the number of estimators is set to 10 and for our own program the number of estimators is set to 30. *Min Max Scaler* has been applied as preprocessing step for our program. For this data set the number of estimators does not have any influence. A possible explanation for this is that the best decision tree is already found with a small number of estimators.
For the *Digits* data set, both the Random Forest program and our own program produce the highest scores. For both programs the *Random Forest* classifier is chosen. For the data set *Splice*, the skeleton program and our own program produce the highest scores. The algorithms that are chosen for these two programs are the *Random Forest* classifier and *Bagging* classifier respectively.

The data sets that get the highest scores with our program (including the data sets that get the same highest score with another program) have the following data set tasks: 3 binary classification data sets, 4 multi-class classification data sets and 2 regression data sets. For the skeleton program the distribution is 2 binary classification data sets, 1 multi-class classification data set, 2 multi-label classification data sets and 3 regression data sets. For the Random Forest program the distribution is 2 binary classification data sets, 1 multi-class classification data set and 2 multi-label classification data sets. For the Decision Tree program the distribution is 1 binary classification data set and 1 multi-class classification data set.

So we can conclude that for 6 out of 20 data sets, our own program gives the best results. In 6 out of 20 data sets, the skeleton program provides the best scores. In 3 out of 20 data sets, the Random Forest program gives the highest scores. In 2 out of 20 data sets, the Decision Tree program gives the highest scores. In 1 out of 20 data sets, the Random Forest program, the skeleton program and our own program give the highest scores. In 1 out of 20 data sets, the Random Forest program and our own program give the best scores. In 1 out of 20 data sets, the skeleton program and our own program give the best scores.

To conclude, in most of the cases our own program performs the best. When only using our program, we get the highest scores for 9 out of 20 data sets.

7.3.2 Comparison on execution time in seconds

When we compare the execution time of all programs and for each data set, we get the following graph (figure 7.2). On the y-axis the execution time in seconds is shown. On the x-axis the 20 data sets are shown.

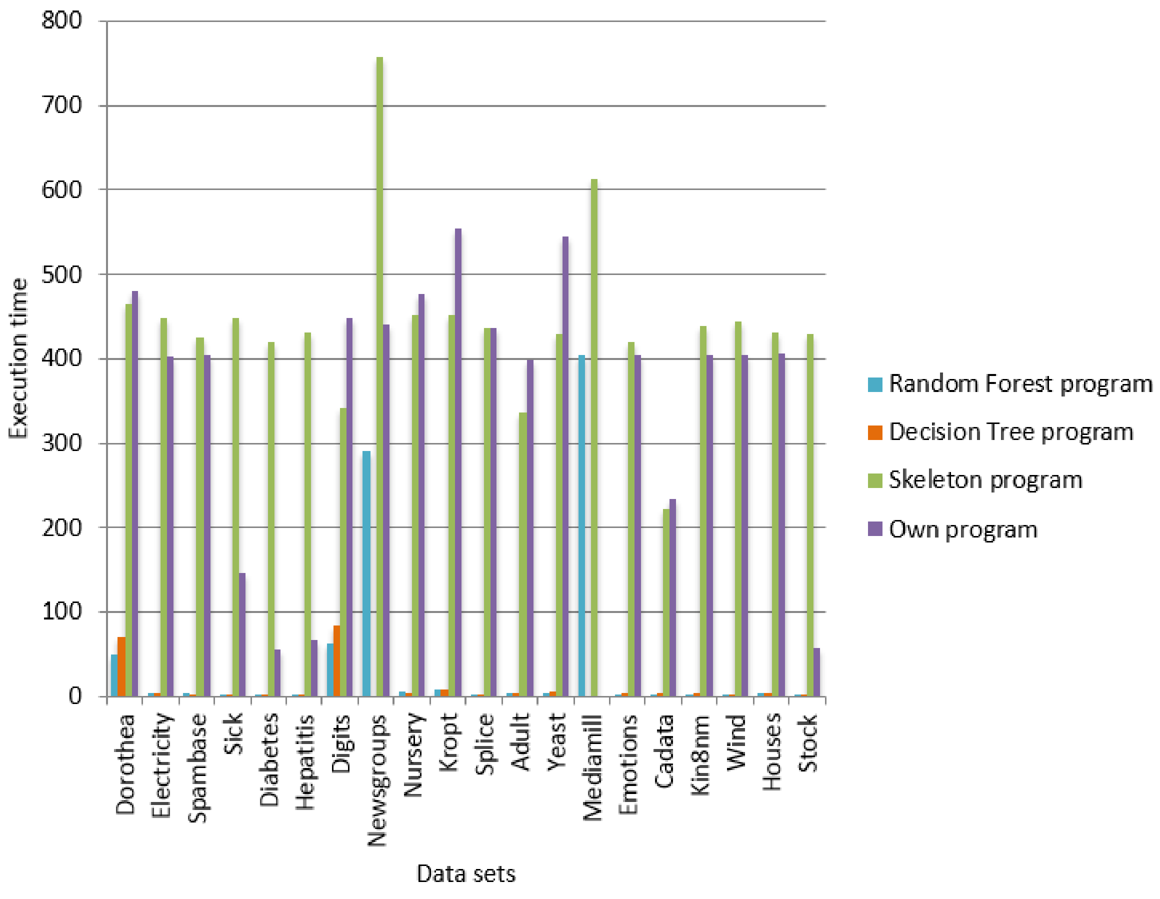

Figure 7.2: Comparison of all programs on execution time

For 10 out of 20 data sets, the Random Forest program gives the shortest execution time. For the other 10 out of 20 data sets, the Decision Tree program gives the shortest execution time. When we look at the tasks of the data sets for which the Decision Tree program needs the shortest time, the distribution is as follows. For 4 binary classification data sets the Decision Tree program needs the shortest time. The Decision Tree program is also faster for 3 multi-class classification data sets, 1 multi-label classification data set and 2 regression data sets. The Random Forest program is faster for 2 binary classification data sets, 2 multi-class classification data sets, 3 multi-label classification data sets and 3 regression data sets.

Both the skeleton program and our own program need a much longer execution time than the two other programs for all data sets. This is logical as the skeleton program keeps adding more base learners until there is no time left. Our own program searches through a search space for the preprocessing methods and hyper-parameter settings. Also it evaluates many different algorithms. For 7 out of 20 data sets, our program needs more time compared to the skeleton program. In 1 out of 20 data sets, the execution time exceeds the time budget for our program. In 12 out of 20 data sets, the skeleton program needs more time compared to our program.

Our program needs more time for most multi-class and multi-label classification data sets. The 8 data sets that need more time for our program (including the data set that exceeds the time budget) compared to the skeleton program exists of 3 multi-class classification data sets, 3 multi-label classification data sets, 1 binary classification data set and 1 regression data set.
For the skeleton program, most data sets that need more time compared to our program have a binary classification or regression task. The 12 data sets that need more time exists of 5 binary classification data sets, 4 regression data sets, 2 multi-class classification data sets and 1 multi-label classification data set.

7.3.3 Trade-off between execution time and validation score (1/2)

In the four scatter plots below (figure 7.3 until 7.6) all 20 data sets for each program separately are plotted. On the x-axis the execution time in seconds is shown on a logarithmic scale. On the y-axis the score on the validation set is shown (based on a specific scoring metric which is specified in the information file). Each color represents another data task. The red color represents the binary classification task, the purple color represents the regression task, the green color represents the multi-label classification task and the blue color represents the multi-class classification task. The labels of the data points represent the data sets. The optimal location of the data points would be in the top left corner. That position represents a high score on the validation set and a short execution time. When the time budget exceeded for a certain data set, then this data set is not plotted.

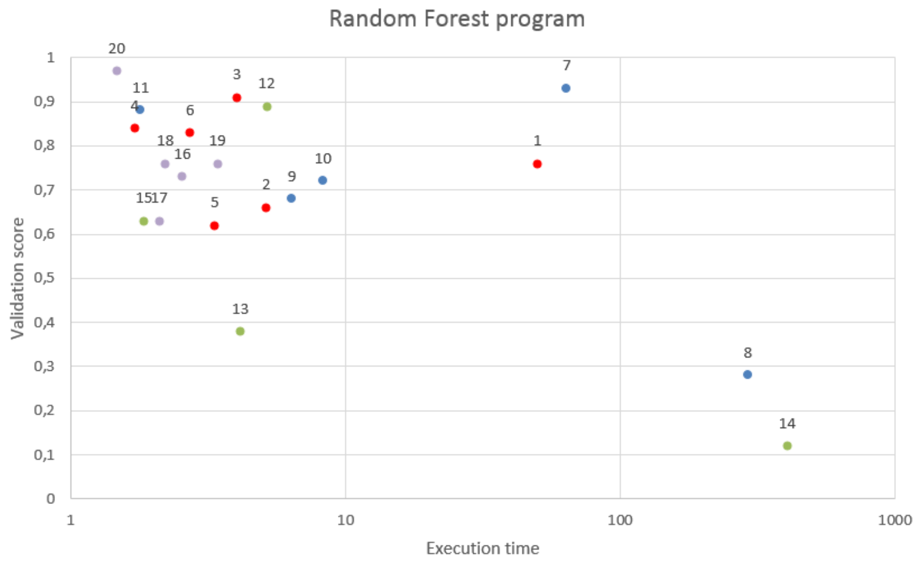

Figure 7.3: Trade-off between execution time and validation score for the Random Forest program

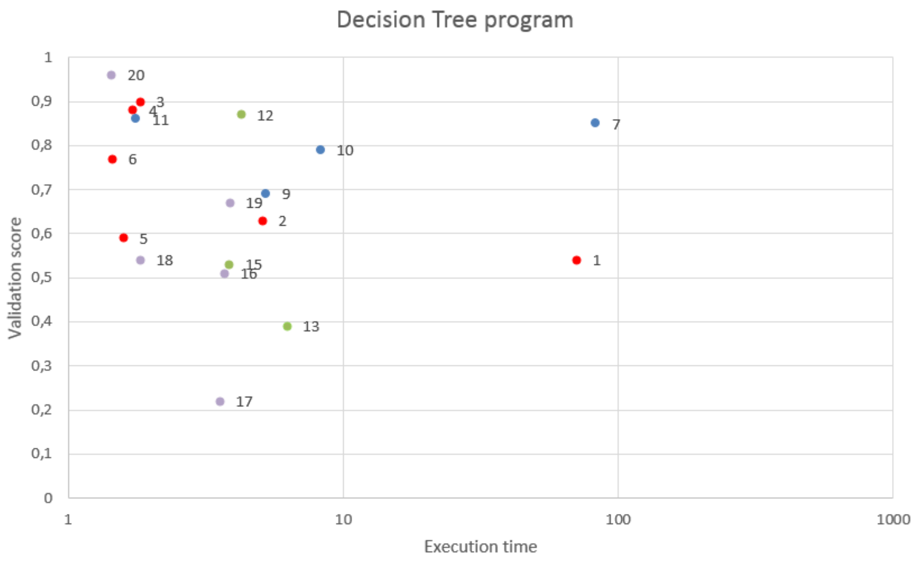

Figure 7.4: Trade-off between execution time and validation score for the Decision Tree program

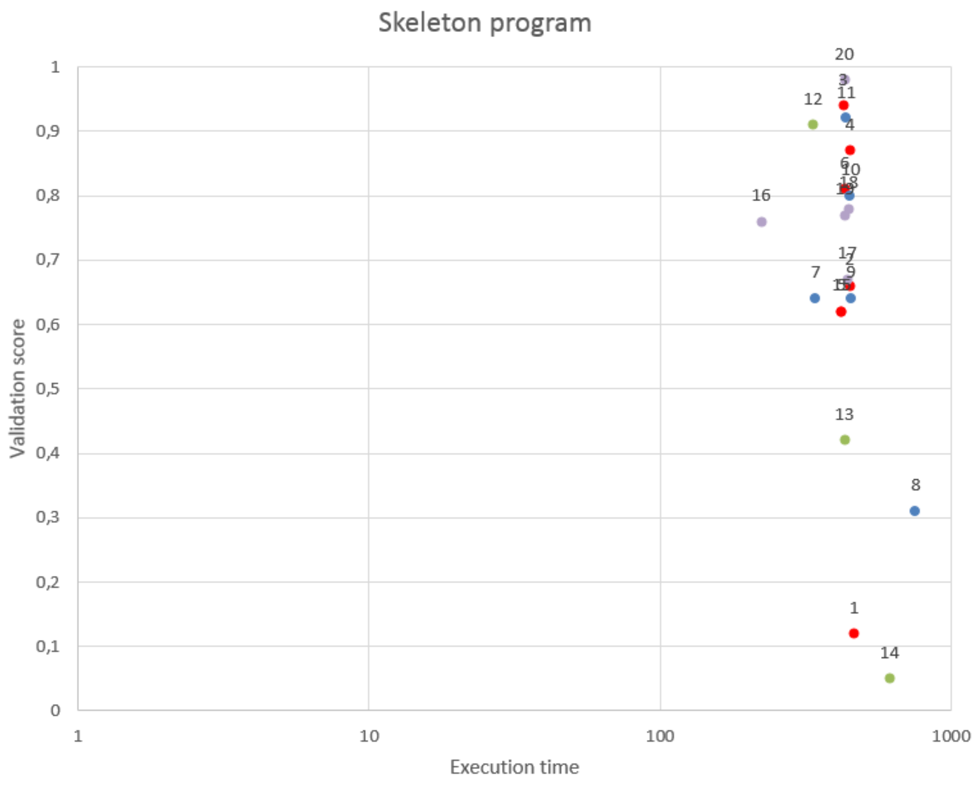

Figure 7.5: Trade-off between execution time and validation score for the skeleton program

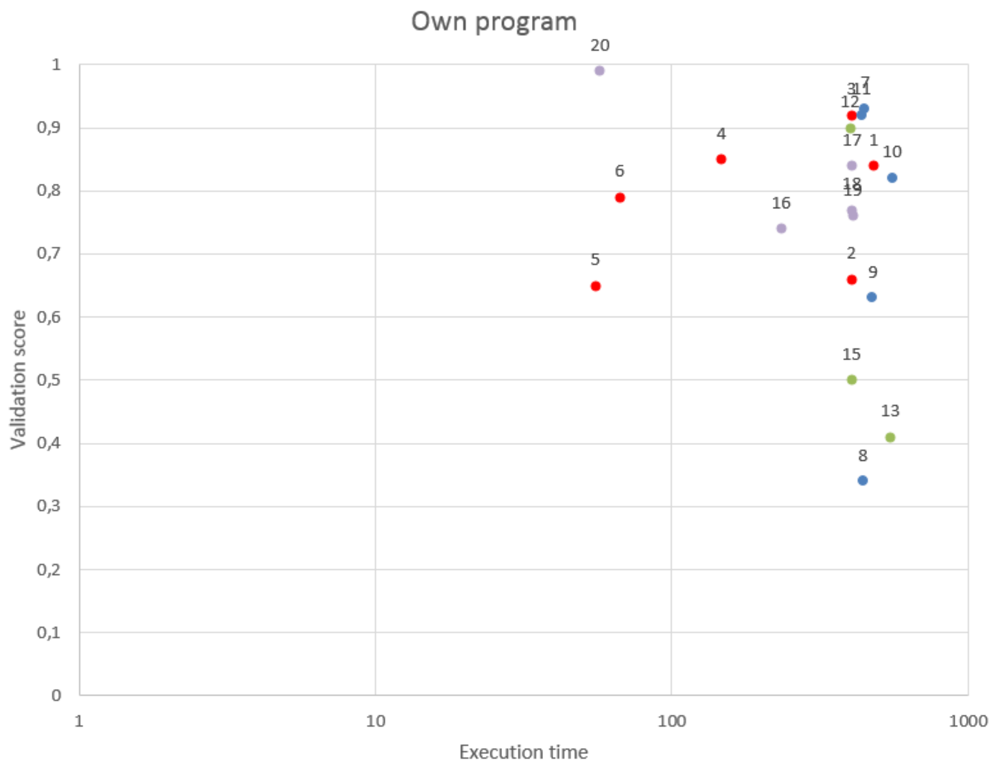

Figure 7.6: Trade-off between execution time and validation score for our own program

When we look at the scatter plot of the Random Forest program, we can see that most data points are located in the top left corner. For most of the data sets the execution time lies between 1 and 10 seconds. This is the case for all regression data sets. For the data sets *Newsgroups* and *Mediamill* more time is needed. This is possibly because of the large number of attributes for the *Newsgroups* data set and the large number of target attributes for the *Mediamill* data set. For most data sets the validation score lies between 0,6 and 1,0. This is the case for all binary classification and regression data sets.

When we look at the scatter plot of the Decision Tree program, we can see that the data points are more distributed compared to the Random Forest program. For most of the data sets the execution time lies between 1 and 10 seconds. This is the case for all regression and multi-label classification data sets. For the data sets *Dorothea* and *Digits* the execution time is longer. A possible explanation for this is that both data sets have a large number of attributes. For most data sets the validation score lies between 0,5 and 1,0. This is the case for all binary classification and multi-class classification data sets.

When we look at the scatter plot of the skeleton program, we can see that the execution time varies between 100 and 1000 seconds. We can see that most data sets have around the same execution time. This is because we keep adding base learners until there is no time left. For most of the data sets the validation score lies between 0,6 and 1,0. This is the case for all regression data sets. Compared to the Random Forest and Decision Tree program, the skeleton program needs a much longer execution time for all data sets.

When we look at the scatter plot of our own program, we can see that the execution time varies more compared to the skeleton program. This is especially the case for the binary classification data sets. The execution time varies more because our program stops when the whole process of evaluating all algorithms, optimizing the preprocessing steps and optimizing the hyper-parameters is finished. Because of this it is possible that our program finishes while there is still some time left. For the skeleton program we keep adding base learners so this process will continue until there is no time left. Especially for small data sets our program runs fast. We can see that the *Stock*, *Diabetes* and *Hepatitis* data sets need less time. This is because they have a small number of attributes and a small number of training instances and validation instances. For most of the data sets the execution time varies between 100 and 1000 seconds. For most of the data sets the validation score lies between 0,6 and 1,0. This is the case for all binary classification and regression data sets.

We can conclude that the scatter plots of the Random Forest and Decision Tree program look alike. But when we take a closer look we can see some differences. Most of the data points of the Random Forest program are located close together in the top left corner. The data points of the Decision Tree program are a little bit more spread out across different validation scores. We can also see that the scatter plots of the skeleton program and our own program look alike. However, the validation scores of the skeleton program are more distributed compared to the validation scores of our own program. There is more variation in the execution times of our own program compared to the skeleton program.

7.3.4 Trade-off between execution time and validation score (2/2)

In the graph below (figure 7.7) we plotted a line for every data set. The line of each data set is created by plotting the execution time and validation score for all four programs. So each line is created by four data points. The squares represent the

Random Forest program, the triangles represent the Decision Tree program, the circles represent the skeleton program and the crosses represent our own program. On the x-axis the execution time in seconds is shown. On the y-axis the score on the validation set is shown (based on a specific scoring metric which is specified in the information file).

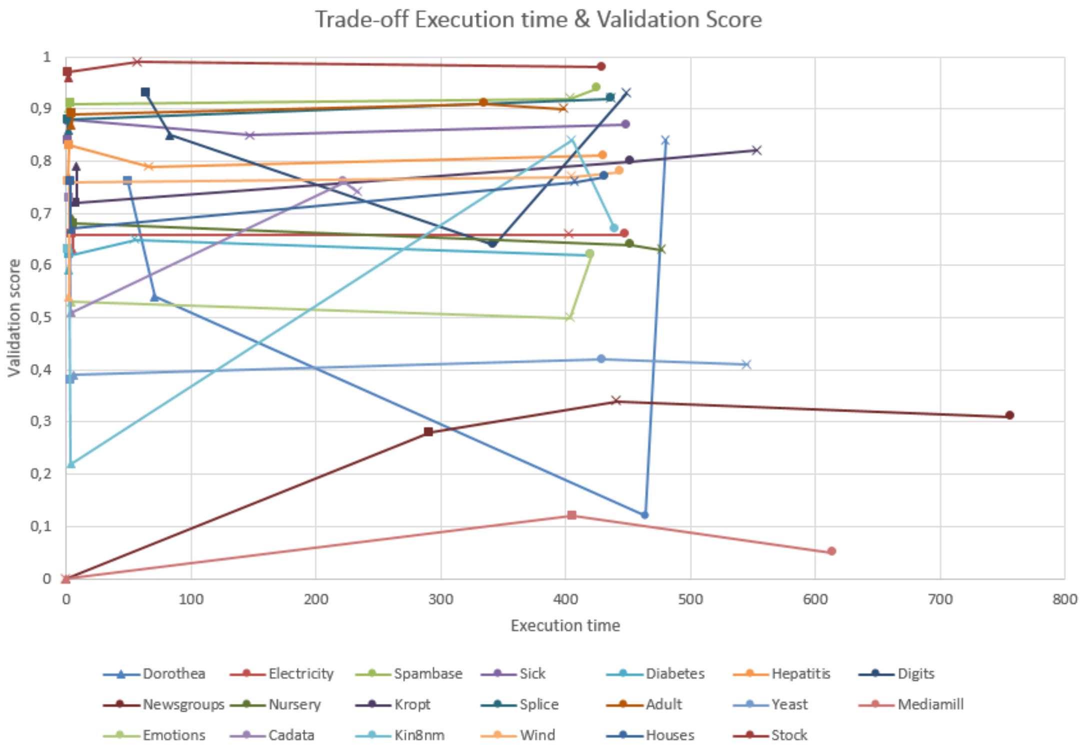

Figure 7.7: Trade-off between execution time and validation score for each data set

In this graph we can see if there is any relation between execution time and validation score. For example we would see many linearly rising lines if the validation score increases when the execution time for a program is longer. This is not always the case. For some data sets the validation score decreases when the execution time increases. For example for the data set *Digits* the Random Forest program gives a higher score (0,93) and shorter execution time (63,93 seconds) compared to the Decision Tree (score of 0,85 in 83,12 seconds) and skeleton program (score of 0,64 in 342,54 seconds). There are some lines that increase linearly. When looking at the data we can see that this is the case for 4 data sets (*Electricity*, *Spambase*, *Splice* and *Wind*). For example for the *Wind* data set the skeleton program has the longest execution time (444,09 seconds) and highest score (0,78). For this data set there is a difference in validation score of 0,24 compared to the Decision Tree program (a score of 0,54) with a much shorter execution time (1,83 seconds).

7.3.5 Comparison on type of algorithms

For our program, all data sets evaluated a *Decision Tree* classifier and *Random Forest* classifier, except for the data sets *Yeast*, *Mediamill* and *Cadata*. For the data sets *Yeast* and *Cadata* there was only enough time to evaluate the *Random Forest* classifier. For the data set *Mediamill* no algorithm was evaluated because of a timeout on the first evaluated algorithm (*Random Forest* classifier).

The distribution of algorithms that were chosen as best algorithms for our program is represented as follows:

- 5 times *Random Forest* classifier
- 5 times *Bagging* classifier
- 2 times *SVR* (Support Vector Machine regressor)
- 2 times *KNeighbors* classifier
- 1 time *Logistic Regression*
- 1 time *GaussianNB* (Gaussian Naive Bayes)
- 1 time *Random Forest* regressor
- 1 time *Bagging* regressor
- 1 time *KNeighbors* regressor

We can conclude that there is quite some variation in the choice of algorithm. There is not one algorithm that will always give the highest score on all data sets. When we join the regressors and classifiers of the same algorithm (for example the *Random Forest* classifier and *Random Forest* regressor), we can see that the *Random Forest* algorithm and *Bagging* algorithm are chosen most often. The *Random Forest* algorithm is chosen 6 times and the *Bagging* algorithm also 6 times. These two algorithms cover more than half of all data sets.

A possible explanation for the fact that the *Random Forest* and *Bagging* algorithm are chosen most often is the following. Compared to an algorithm such as a Support Vector Machine, the hyper-parameters of these two algorithms do not have to be set to a certain value in order to get a good score. For a Support

Vector Machine the hyper-parameter "C" is critical and should be set to a certain value. Our strategy tries all algorithms with their default hyper-parameters and then keeps the three best algorithms. Because of this, it is possible that a Support Vector Machine is eliminated at this step in the process. The consequence is that we do not know if the Support Vector Machine would have provided a good score when the hyper-parameters would have been optimized.

Chapter 8

Conclusion

On the basis of this study we can draw some conclusions.

First of all, we discovered that the influence of preprocessing methods is different for each algorithm and data set. It is not possible to use the same preprocessing methods for all algorithms or all data sets in order to get the best results. For example for an *SGD Classifier* it can be important to scale the attributes, while for a *Random Forest* classifier scaling does not have any influence on the performance. The best way to discover which methods work best on a given algorithm and data set, is to try all different combinations and save the combination with the best result. Presumably it is possible to eliminate some combinations for a certain algorithm based on common knowledge. For example for a *K-Nearest Neighbours* algorithm it is obvious to normalize the attributes as distance is an important property.

Secondly we can conclude that our program does not always give the best validation score on a data set. For 9 out of 20 data sets we get the highest score with our program. These data sets mostly have a multi-class or regression task. For some data sets, the skeleton program, Decision Tree program or Random Forest program give a better result. The Decision Tree program will only provide the best score for 2 out of 20 data sets.

Thirdly, when we compare all four programs on execution time, we can see that our program and the skeleton program need a much longer time compared to the Random Forest program and Decision Tree program. The Decision Tree program needs a shorter time for most binary classification data sets, the Random Forest program is faster for most multi-label data sets. When we compare our program and the skeleton program, we can see that our program needs more time for most multi-class and multi-label data sets. The skeleton program needs more time for most binary classification and regression data sets.

We can also conclude that random subsampling of the data when the data set is large is a good method to evaluate an algorithm in a relatively short time. By randomly picking instances from the data set we can create a sample. In this way we do not have to look at all instances to evaluate an algorithm so the algorithm will run much faster.

Moreover, we can conclude that setting a timeout when searching through a search space is a good strategy. When many algorithms need to be evaluated, we would like to terminate an algorithm that is running very slow. In this way we do not waste too much time when evaluating a particular algorithm.

When we look at the trade-off between execution time and validation score for each data set and program separately, we cannot conclude that there is a clear relation between execution time and validation score. For some data sets, the programs that need a longer execution time do not necessarily provide a higher validation score. This is only the case for 4 data sets.

Furthermore, when we analyze the type of algorithm that is chosen by our program as the best algorithm for each data set, we can conclude that the choice depends on the data set itself. There is not one algorithm that will always give the best result for all data sets. The algorithms that are chosen most often by our program are the *Random Forest* and *Bagging* algorithm.

8.1 Future work

Our study can be extended in many different ways.

First of all, we used random search instead of exhaustive grid search to optimize the hyper-parameters of an algorithm. Experiments can be carried out in order to compare the performance of both methods.

For the experiments with the preprocessing methods and optimizing the order of the algorithms, the separation of the training and validation set could be improved. When trying different combinations of preprocessing methods on the 5 data sets, we calculated a score on the validation set. We used this score to decide which preprocessing methods work best. This method can cause overfitting, because the data sets have seen the validation set so the decision is not based on unseen data. It would have been better to keep a separate validation set and to

use this set to evaluate the results of the experiments.
This is also the case for the experiment about optimizing the order of the algorithms. We calculated a rank for each algorithm based on the score on the validation set and the execution time. A better method would be to evaluate the final order of the algorithms on an unseen data set.

Furthermore, we did not optimize all hyper-parameters of the preprocessing methods. For example, for PCA a search space can be created over the hyper-parameters *n_components* and *whiten*. Furthermore, we did not evaluate all preprocessing methods that are implemented in Scikit-learn. For example we did not look at the influence of *Tf-idf* for data sets based on text classification.

Also, we did not include all algorithms implemented in Scikit-learn. For example we did not implement the *Extremely Randomized Trees* algorithm.

Moreover, we based the choice of algorithms that are tried for a particular data set on the task of the data set. For example, for binary classification data sets we tried other algorithms than for regression data sets. Of course the choice of algorithms can be based on many other properties of a data set. For example, the choice can be based on the number of attributes or number of instances that are present in a data set. Also, it can be based on the type of attributes, for example categorical or numerical attributes.

Another part of this study that can be investigated further is the model selection strategy. We used a brute force approach by first trying all algorithms with their default hyper-parameters. Then we used a strategy similar to Hoeffding Races in which the worst algorithms are eliminated early in the process. There are many other possible strategies to decide which algorithm will give the best result on the validation set. For example, a Particle Swarm Optimization (PSO) strategy can be investigated.

In this study we made some assumptions. For example, we set the time budget of the data sets that were not provided by the challenge to 400 seconds. We did not experiment with other amounts. Maybe some data sets need a longer execution time in order to get better results. This can be investigated by experimenting with different amounts. Also, we set timeouts for evaluating the different algorithms. For example, for evaluating each algorithm with their default hyper-parameters we set a timeout of 60 seconds. Experiments can be carried out for setting the right amount of seconds (or not setting a timeout at all).

Moreover, this study can be extended by evaluating many more data sets. We only evaluated 20 data sets for this study.

Appendices

A Data sets

Name	**Task**	**Metric**	**Time**	**Categorical**	**Missing**	**Sparse**	**Target type**	**Feat type**	**Feat num**	**Target num**	**Label num**	**Train num**	**Valid num**
Dorothea	Binary	auc	400	0	0	1	Binary	Binary	100000	1	2	800	350
Electricity	Binary	f1	400	1	0	0	Binary	Mixed	8	1	2	27188	18124
Spambase	Binary	f1	400	0	0	0	Binary	Numerical	57	1	2	2761	1840
Sick	Binary	f1	400	1	1	0	Binary	Mixed	29	1	2	2264	1508
Diabetes	Binary	f1	400	0	0	0	Binary	Numerical	8	1	2	461	307
Hepatitis	Binary	f1	400	1	1	0	Binary	Mixed	19	1	2	93	62
Digits	Multiclass	bac	300	0	0	0	Categorical	Numerical	1568	10	10	15000	20000
Newsgroups	Multiclass	pac	300	0	0	1	Numerical	Numerical	61188	20	20	13142	1877
Nursery	Multiclass	f1	400	1	0	0	Categorical	Categorical	8	5	5	7776	5184
Kropt	Multiclass	f1	400	1	0	0	Categorical	Categorical	6	18	18	16834	11222
Splice	Multiclass	f1	400	1	0	0	Categorical	Categorical	60	3	3	1914	1276
Adult	Multilabel	f1	300	1	1	0	Binary	Mixed	24	3	3	34190	4884
Yeast	Multilabel	f1	400	0	0	0	Binary	Numerical	103	14	14	1451	966
Mediamill	Multilabel	f1	400	0	0	0	Binary	Numerical	120	101	101	26345	17562
Emotions	Multilabel	f1	400	0	0	0	Binary	Numerical	72	6	6	356	237
Cadata	Regression	r2	200	0	0	0	Numerical	Numerical	16	1	0	5000	5000
Kin8nm	Regression	r2	400	0	0	0	Numerical	Numerical	8	1	0	4916	3276
Wind	Regression	r2	400	0	0	0	Numerical	Numerical	14	1	0	3945	2629
Houses	Regression	r2	400	0	0	0	Numerical	Numerical	8	1	0	12384	8256
Stock	Regression	r2	400	0	0	0	Numerical	Numerical	9	1	0	570	380

Table 1: Datasets

B Hyper-parameter optimization

Classifier	**Optimized parameters**	**Values**	**Number of iterations**
AdaBoostClassifier	learning rate	[0,1]	15
BaggingClassifier	max features, max samples	[0,1]; [0,1]	15
BernoulliNB	alpha	[0,1]	15
DecisionTreeClassifier	criterion, splitter, max features, min samples split, min samples leaf	gini, entropy; best, random; [0,1]; [1,11]; [1,11]	30
GaussianNB	-	-	-
GradientBoostingClassifier	learning rate, max features, min samples split, min samples leaf	[0,1]; [0,1]; [1,11]; [1,11]	30
KNeighborsClassifier	n neighbors, weights, algorithm	[1,11]; uniform, distance; auto, ball tree, kd tree, brute	15
LogisticRegression	C	expon(scale = 100)	15
RandomForestClassifier	max features, min samples split, min samples leaf	[0,1]; [1,11]; [1,11]	30
SGDClassifier	alpha, l1 ratio, loss, penalty	10ˆ-[1,7]; [0,1]; log, modified huber; l1, l2, elasticnet	15
SVC	C, kernel, gamma	expon(scale = 100); linear, poly, rbf; expon(scale=0.1)	5
DecisionTreeRegressor	max features, min samples split, min samples leaf	[0,1]; [1,11]; [1,11]	240
SVR	C, kernel, gamma	expon(scale = 100); linear, poly, rbf; expon(scale=0.1)	5
AdaBoostRegressor	learning rate, loss	[0,1]; linear, square, exponential	30
BaggingRegressor	max features, max samples	[0,1]; [0,1]	15
RandomForestRegressor	max features, min samples split, min samples leaf	[0,1]; [1,11]; [1,11]	30
GradientBoostingRegressor	loss, learning rate, max features, min samples split, min samples leaf	ls, lad, huber, quantile; [0,1];[0,1]; [1,11]; [1,11]	30
KNeighborsRegressor	n neighbors, algorithm	[1,11]; auto, ball tree, kd tree, brute	15
Linear Regression	fit intercept, normalize	true, false; true, false	4
BayesianRidge	alpha 1, alpha 2, lambda 1, lambda 2	10ˆ-[1,7]; 10ˆ-[1,7]; 10ˆ-[1,7]; 10ˆ-[1,7]	15
SGDRegressor	alpha, l1 ratio, loss, penalty	10ˆ-[1,7]; [0,1]; squared loss, huber; l1, l2, elasticnet	15
Ridge	alpha, solver, fit intercept, normalize	[0,1]; auto, svd, cholesky, lsqr, sparse_cq; true, false; true, false	15
Lasso	alpha, fit intercept, normalize	[0,1]; true, false; true, false	15
ElasticNet	alpha, l1 ratio, fit intercept, normalize	[0,1]; [0,1]; true, false; true, false	15

Table 2: Hyper-parameter optimization settings

C Experiments

Classifier	**Preprocessing**	**Validation score**	**Time spent**
AdaBoostClassifier	feature selection (1000), standard scaler	0.6185	219.08
AdaBoostClassifier	feature selection (1000)	0.6185	227.52
AdaBoostClassifier	**normalize, feature selection (1000)**	0.6185	43.40
AdaBoostClassifier	normalize, feature selection (1000), standard scaler	0.6185	47.21
AdaBoostClassifier	feature selection Johnson Lindenstrauss	0.5814	92.33
BaggingClassifier	**normalize, feature selection (1000), standard scaler**	0.865	43.91
BaggingClassifier	normalize, feature selection (1000)	0.800	61.62
BaggingClassifier	feature selection (1000)	0.817	65.13
BaggingClassifier	feature selection (1000), standard scaler	0.808	66.25
BaggingClassifier	feature selection Johnson Lindenstrauss	0.7806	113.28
BernoulliNB	**normalize, feature selection (1000), standard scaler**	0.7014	43.87
BernoulliNB	feature selection (1000), standard scaler	0.7014	47.41
BernoulliNB	normalize, feature selection (1000)	0.7014	50.56
BernoulliNB	feature selection (1000)	0.7014	54.90
BernoulliNB	feature selection Johnson Lindenstrauss	0.53	67.16
DecisionTreeClassifier	**normalize, feature selection (1000), standard scaler**	0.48	51.01
DecisionTreeClassifier	normalize, feature selection (1000)	0.40	51.02
DecisionTreeClassifier	feature selection (1000), standard scaler	0.423	52.50
DecisionTreeClassifier	feature selection (1000)	0.3674	52.87
DecisionTreeClassifier	feature selection Johnson Lindenstrauss	0.4819	79.16
GaussianNB	**normalize, feature selection (1000)**	0.7118	41.80
GaussianNB	feature selection (1000), standard scaler	0.7118	42.52
GaussianNB	feature selection (1000)	0.7118	63.16
GaussianNB	normalize, feature selection (1000), standard scaler	0.7118	84.39
GaussianNB	feature selection Johnson Lindenstrauss	0.5739	140.28
GradientBoostingClassifier	**normalize, feature selection (1000), standard scaler**	0.73	41.15
GradientBoostingClassifier	feature selection (1000)	0.7340	482.02
GradientBoostingClassifier	feature selection (1000), standard scaler	-	-
GradientBoostingClassifier	normalize ,feature selection (1000)	-	-
GradientBoostingClassifier	feature selection Johnson Lindenstrauss	-	-
KNeighborsClassifier	feature selection (1000)	0.117	43.46
KNeighborsClassifier	normalize, feature selection (1000)	0.117	43.82
KNeighborsClassifier	**feature selection (1000), standard scaler**	0.203	43.96
KNeighborsClassifier	normalize, feature selection (1000), standard scaler	0.203	44.57
KNeighborsClassifier	feature selection Johnson Lindenstrauss	0.00	62.25
LogisticRegression	normalize, feature selection (1000)	0.778	42.76
LogisticRegression	feature selection (1000)	0.778	42.94
LogisticRegression	**feature selection (1000), standard scaler**	0.822	44.93
LogisticRegression	normalize, feature selection (1000), standard scaler	0.822	56.18
LogisticRegression	feature selection Johnson Lindenstrauss	0.792	63.55
RandomForestClassifier	normalize, feature selection (1000)	0.8245	39.84
RandomForestClassifier	**feature selection (1000), standard scaler**	0.8511	40.34
RandomForestClassifier	normalize, feature selection (1000), standard scaler	0.848	48.57
RandomForestClassifier	feature selection (1000)	0.8741	68.15
RandomForestClassifier	feature selection Johnson Lindenstrauss	0.7977	51.08
SGD Classifier	**feature selection (1000), standard scaler**	0.807	65.38
SGD Classifier	normalize, feature selection (1000)	0.7714	75.48
SGD Classifier	feature selection (1000)	0.7714	819.88
SGD Classifier	normalize, feature selection (1000), standard scaler	0.807	874.67
SGD Classifier	feature selection Johnson Lindenstrauss	0.789	178.89
SVC	**normalize, feature selection (1000), standard scaler**	0.842	47.33
SVC	feature selection (1000), standard scaler	0.842	654.40
SVC	feature selection (1000)	0.848	70.05
SVC	normalize, feature selection (1000)	0.848	70.89
SVC	feature selection Johnson Lindenstrauss	0.8430	66.40

Table 3: Experiments Dorothea data set

Classifier	**Preprocessing**	**Validation score**	**Time spent**
AdaBoostClassifier	**PCA (4 %)**	0.5916	71.53
AdaBoostClassifier	PCA (4 %), Normalization	0.5916	76.76
AdaBoostClassifier	PCA (4 %), Min Max Scaler	0.5917	77.55
AdaBoostClassifier	PCA (4 %), Min Max Scaler, Normalization	0.5917	81.15
AdaBoostClassifier	PCA Johnson Lindenstrauss	0.5598	368.47
BaggingClassifier	**PCA (4 %)**	0.8897	151.58
BaggingClassifier	PCA (4 %), Min Max Scaler, Normalization	0.892	158.13
BaggingClassifier	PCA (4 %), Min Max Scaler	0.888	163.87
BaggingClassifier	PCA (4 %), Normalization	0.8939	173.34
BaggingClassifier	PCA Johnson Lindenstrauss	-	-
BernoulliNB	**PCA (4 %)**	0.6919	65.03
BernoulliNB	PCA (4 %), Min Max Scaler, Normalization	0.00	72.06
BernoulliNB	PCA (4 %), Normalization	0.6919	85.75
BernoulliNB	PCA (4 %), Min Max Scaler	0.00	97.57
BernoulliNB	PCA Johnson Lindenstrauss	0.6963	107.80
DecisionTreeClassifier	**PCA (4 %)**	0.78953	91.21
DecisionTreeClassifier	PCA (4 %), Min Max Scaler	0.7886	92.38
DecisionTreeClassifier	PCA (4 %), Min Max Scaler, Normalization	0.7889	93.11
DecisionTreeClassifier	PCA (4 %), Normalization	0.7922	94.64
DecisionTreeClassifier	PCA Johnson Lindenstrauss	0.7351	340.65
GaussianNB	**PCA (4 %), Normalization**	0.85107	61.68
GaussianNB	PCA (4 %), Min Max Scaler	0.85107	71.46
GaussianNB	PCA (4 %), Min Max Scaler, Normalization	0.85107	73.68
GaussianNB	PCA (4 %)	0.85107	76.30
GaussianNB	PCA Johnson Lindenstrauss	0.3297	126.39
GradientBoostingClassifier	**PCA (4 %)**	0.8614	202.57
GradientBoostingClassifier	PCA (4 %), Normalization	0.8612	211.30
GradientBoostingClassifier	PCA (4 %), Min Max Scaler, Normalization	0.8616	217.92
GradientBoostingClassifier	PCA (4 %), Min Max Scaler	0.8615	225.95
GradientBoostingClassifier	PCA Johnson Lindenstrauss	-	-
KNeighboursClassifier	**PCA (4 %)**	0.95711	250.08
KNeighboursClassifier	PCA (4 %), Normalization	0.95711	261.62
KNeighboursClassifier	PCA (4 %), Min Max Scaler	0.89685	299.87
KNeighboursClassifier	PCA (4 %), Min Max Scaler, Normalization	0.89685	342.88
KNeighboursClassifier	PCA Johnson Lindenstrauss	-	-
LogisticRegression	**PCA (4 %), Min Max Scaler**	0.8695	105.28
LogisticRegression	PCA (4 %), Min Max Scaler, Normalization	0.8695	112.16
LogisticRegression	PCA (4 %)	0.8733	455.20
LogisticRegression	PCA (4 %), Normalization	0.8734	756.92
LogisticRegression	PCA Johnson Lindenstrauss	-	-
RandomForestClassifier	**PCA (4 %), Min Max Scaler, Normalization**	0.92192	67.43
RandomForestClassifier	PCA (4 %)	0.91875	74.67
RandomForestClassifier	PCA (4 %), Min Max Scaler	0.920508	81.86
RandomForestClassifier	PCA (4 %), Normalization	0.91935	90.38
RandomForestClassifier	PCA Johnson Lindenstrauss	0.7219	158.67
SGDClassifier	**PCA (4 %), Min Max Scaler, Normalization**	0.8668	118.72
SGDClassifier	PCA (4 %), Min Max Scaler	0.8668	120.71
SGDClassifier	PCA (4 %), Normalization	error	error
SGDClassifier	PCA (4 %)	error	error
SGDClassifier	PCA Johnson Lindenstrauss, Min Max Scaler	-	1412.35
SVC	**PCA (4 %), Min Max Scaler, Normalization**	0.911	272.02
SVC	PCA (4 %), Min Max Scaler	0.893	499.03
SVC	PCA (4 %), Normalization	0.00	635.40
SVC	PCA (4 %)	0.00	784.42
SVC	PCA Johnson Lindenstrauss	-	-

Table 4: Experiments Digits data set

Classifier	**Preprocessing**	**Validation score**	**Time spent**
OneVSRest AdaBoostClassifier	Missing values (replace by 0), OHE, Normalize	0.7974	15.08
OneVSRest AdaBoostClassifier	Missing values (replace by 0), OHE, Min Max Scaler	0.7974	15.77
OneVSRest AdaBoostClassifier	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.7974	15.87
OneVSRest AdaBoostClassifier	Missing values (replace by 0)	0.7974	17.20
OneVSRest AdaBoostClassifier	Missing values (replace by 0), OHE	0.7974	17.40
OneVSRest AdaBoostClassifier	**Missing values (replace by mean)**	0.7982	13.44
OneVSRest BaggingClassifier	Missing values (replace by 0), OHE, Normalize	0.8072	76.68
OneVSRest BaggingClassifier	Missing values (replace by 0), OHE, Min Max Scaler	0.8061	80.27
OneVSRest BaggingClassifier	Missing values (replace by 0)	0.8045	82.70
OneVSRest BaggingClassifier	Missing values (replace by 0), OHE	0.8042	87.53
OneVSRest BaggingClassifier	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.8092	92.57
OneVSRest BaggingClassifier	**Missing values (replace by mean)**	0.8077	66.86
OneVSRest BernoulliNB	Missing values (replace by 0), OHE, Normalize	0.7235	3.52
OneVSRest BernoulliNB	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.7339	3.61
OneVSRest BernoulliNB	Missing values (replace by 0), OHE, Min Max Scaler	0.7339	3.65
OneVSRest BernoulliNB	Missing values (replace by 0), OHE	0.7235	5.66
OneVSRest BernoulliNB	Missing values (replace by 0)	0.7235	6.55
OneVSRest BernoulliNB	**Missing values (replace by mean)**	0.7230	2.29
OneVSRest DecisionTreeClassifier	Missing values (replace by 0)	0.7364	57.28
OneVSRest DecisionTreeClassifier	Missing values (replace by 0), OHE, Min Max Scaler	0.7377	7.75
OneVSRest DecisionTreeClassifier	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.736	9.54
OneVSRest DecisionTreeClassifier	Missing values (replace by 0), OHE, Normalize	0.7341	9.65
OneVSRest DecisionTreeClassifier	Missing values (replace by 0), OHE	0.736	9.85
OneVSRest DecisionTreeClassifier	**Missing values (replace by mean)**	0.7383	5.19
OneVSRest GaussianNB	Missing values (replace by 0), OHE, Min Max Scaler	0.7463	3.45
OneVSRest GaussianNB	Missing values (replace by 0), OHE, Normalize	0.7463	3.48
OneVSRest GaussianNB	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.7463	3.49
OneVSRest GaussianNB	Missing values (replace by 0)	0.7463	4.87
OneVSRest GaussianNB	Missing values (replace by 0), OHE	0.7463	5.82
OneVSRest GaussianNB	**Missing values (replace by mean)**	0.7475	2.30
OneVSRest GradientBoostingClassifier	Missing values (replace by 0), OHE, Min Max Scaler	0.8102	23.01
OneVSRest GradientBoostingClassifier	Missing values (replace by 0), OHE, Normalize	0.8102	24.97
OneVSRest GradientBoostingClassifier	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.8102	25.56
OneVSRest GradientBoostingClassifier	Missing values (replace by 0)	0.8102	28.35
OneVSRest GradientBoostingClassifier	Missing values (replace by 0), OHE	0.8102	28.58
OneVSRest GradientBoostingClassifier	**Missing values (replace by mean)**	0.8099	20.33
OneVSRest KNeighboursClassifier	Missing values (replace by 0), OHE	0.6536	19.44
OneVSRest KNeighboursClassifier	Missing values (replace by 0)	0.6535	22.74
OneVSRest KNeighboursClassifier	Missing values (replace by 0), OHE, Normalize	0.6535	34.28
OneVSRest KNeighboursClassifier	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	-	-
OneVSRest KNeighboursClassifier	Missing values (replace by 0), OHE, Min Max Scaler	-	-
OneVSRest KNeighboursClassifier	**Missing values (replace by mean)**	0.6535	14.92
OneVSRest LogisticRegression	Missing values (replace by 0), OHE	0.7250	11.33
OneVSRest LogisticRegression	Missing values (replace by 0), OHE, Normalize	0.7275	12.92
OneVSRest LogisticRegression	**Missing values (replace by 0), OHE, Min Max Scaler**	0.7451	4.78
OneVSRest LogisticRegression	Missing values (replace by 0)	0.7248	40.44
OneVSRest LogisticRegression	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.7451	5.17
OneVSRest LogisticRegression	Missing values (replace by mean)	0.7295	35.45
OneVSRest RandomForest	Missing values (replace by 0), OHE, Min Max Scaler	0.8058	20.88
OneVSRest RandomForest	Missing values (replace by 0), OHE, Normalize	0.8066	22.18
OneVSRest RandomForest	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.8062	22.21
OneVSRest RandomForest	Missing values (replace by 0)	0.8082	23.37
OneVSRest RandomForest	Missing values (replace by 0), OHE	0.8059	29.61
OneVSRest RandomForest	**Missing values (replace by mean)**	0.8058	17.96
OneVSRest SGDClassifier	Missing values (replace by 0), OHE	0.731	11.20
OneVSRest SGDClassifier	Missing values (replace by 0), OHE	0.731	17.80
OneVSRest SGDClassifier	Missing values (replace by 0), OHE, Normalize	0.7310	4.48
OneVSRest SGDClassifier	Missing values (replace by 0), OHE, Min Max Scaler	0.7439	4.97
OneVSRest SGDClassifier	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	0.7439	5.14
OneVSRest SGDClassifier	Missing values (replace by 0)	0.731	8.44
OneVSRest SGDClassifier	**Missing values (replace by mean)**	0.731	2.63
OneVSRest SVC	Missing values (replace by 0), OHE	-	-
OneVSRest SVC	Missing values (replace by 0), OHE, Normalize, Min Max Scaler	-	-
OneVSRest SVC	Missing values (replace by 0), OHE, Normalize	-	-
OneVSRest SVC	Missing values (replace by 0), OHE, Min Max Scaler	-	-
OneVSRest SVC	Missing values (replace by mean)	-	-

Table 5: Experiments Adult data set

Classifier	**Preprocessing**	**Validation score**	**Time spent**
AdaBoostRegressor	Min Max Scaler	0.4281	13.40
AdaBoostRegressor	-	0.4214	6.64
AdaBoostRegressor	**Normalize**	0.4834	8.06
AdaBoostRegressor	Normalize, Min Max Scaler	0.4526	8.31
BaggingRegressor	**-**	0.7482	14.45
BaggingRegressor	Min Max Scaler	0.7509	15.58
BaggingRegressor	Normalize	0.7480	16.87
BaggingRegressor	Normalize, Min Max Scaler	0.7551	18.31
BayesianRidge	**-**	0.6408	2.98
BayesianRidge	Min Max Scaler	0.6408	3.43
BayesianRidge	Normalize	0.6408	3.84
BayesianRidge	Normalize, Min Max Scaler	0.6408	4.50
DecisionTreeRegressor	Normalize	0.5048	101.01
DecisionTreeRegressor	Normalize, Min Max Scaler	0.511	101.25
DecisionTreeRegressor	Min Max Scaler	0.5030	102.06
DecisionTreeRegressor	-	0.5028	102.53
DecisionTreeRegressor	**Normalize, Min Max Scaler**	0.5215	92.74
ElasticNet	Min Max Scaler	0.0368	4.71
ElasticNet	**Normalize**	0.599	4.78
ElasticNet	Normalize, Min Max Scaler	0.036	5.05
ElasticNet	-	0.5995	5.15
GradientBoostingRegressor	**Normalize**	0.6479	172.80
GradientBoostingRegressor	Normalize, Min Max Scaler	0.6479	174.60
GradientBoostingRegressor	-	0.6479	189.44
GradientBoostingRegressor	Min Max Scaler	0.6479	220.83
KNeighborsRegressor	Normalize, Min Max Scaler	0.398	18.67
KNeighborsRegressor	-	0.075	5.23
KNeighborsRegressor	Normalize	0.075	5.88
KNeighborsRegressor	**Min Max Scaler**	0.398	6.72
KNeighborsRegressor	Normalize, Min Max Scaler	0.398	7.14
Lasso	**-**	0.6408	4.63
Lasso	Normalize	0.6408	4.92
Lasso	Min Max Scaler	0.6408	5.03
Lasso	Normalize, Min Max Scaler	0.6408	5.49
Linear Regression	**-**	0.6408	2.36
Linear Regression	Min Max Scaler	0.6408	4.59
Linear Regression	Normalize	0.6408	4.65
Linear Regression	Normalize, Min Max Scaler	0.6408	5.43
RandomForestRegressor	Normalize	0.7473	15.61
RandomForestRegressor	Normalize, Min Max Scaler	0.74566	16.18
RandomForestRegressor	**-**	0.7526	16.40
RandomForestRegressor	Min Max Scaler	0.7425	17.23
Ridge	**-**	0.4037	4.42
Ridge	Min Max Scaler	0.4037	7.54
Ridge	Normalize	0.4037	8.04
Ridge	Normalize, Min Max Scaler	0.4037	9.07
SGDRegressor	**Normalize, Min Max Scaler**	0.3308	4.43
SGDRegressor	Min Max Scaler	0.3308	5.78
SGDRegressor	Normalize	error	error
SGDRegressor	-	error	error
SVR	Min Max Scaler	-0.0538	121.28
SVR	**Normalize, Min Max Scaler**	-0.0538	6.64
SVR	Normalize	-	-
SVR	-	-	-

Table 6: Experiments Cadata data set

Classifier	**Preprocessing**	**Validation score**	**Time spent**
AdaBoostClassifier	**standard scaler, select K best (1000)**	0.0085	84.44
AdaBoostClassifier	normalize, select K best (1000)	0.0082	85.61
AdaBoostClassifier	select K best (1000)	0.0082	87.67
AdaBoostClassifier	normalize, standard scaler, select K best (1000)	0.0085	94.02
AdaBoostClassifier	select K best Johnson Lindenstrauss	0.0082	334.73
BaggingClassifier	**normalize, select K best (1000)**	0.2332	405.33
BaggingClassifier	select K best (1000)	0.2278	411.90
BaggingClassifier	normalize, standard scaler, select K best (1000)	0.2345	482.78
BaggingClassifier	standard scaler, select K best (1000)	0.2334	484.62
BaggingClassifier	select K best Johnson Lindenstrauss	-	-
BernoulliNB	select K best (1000)	0.2571	70.01
BernoulliNB	normalize, select K best (1000)	0.2571	84.13
BernoulliNB	**normalize, standard scaler, select K best (1000)**	0.2921	84.77
BernoulliNB	standard scaler, select K best (1000)	0.2921	87.27
BernoulliNB	select K best Johnson Lindenstrauss	-	-
DecisionTreeClassifier	**normalize, standard scaler, select K best (1000)**	0.1717	101.34
DecisionTreeClassifier	standard scaler, select K best (1000)	0.1743	114.49
DecisionTreeClassifier	select K best (1000)	0.0070	87.73
DecisionTreeClassifier	normalize, select K best (1000)	0.0047	90.53
DecisionTreeClassifier	select K best Johnson Lindenstrauss	0.0084	348.48
GaussianNB	**standard scaler, select K best (1000)**	0.1677	83.83
GaussianNB	normalize, standard scaler, select K best (1000)	0.1677	84.72
GaussianNB	normalize, select K best (1000)	0.1542	85.76
GaussianNB	select K best (1000)	0.1542	87.80
GaussianNB	select K best Johnson Lindenstrauss	0.2797	456.53
GradientBoostingClassifier	**select K best (1000)**	0.1501	361.06
GradientBoostingClassifier	normalize, standard scaler, select K best (1000)	0.1373	392.84
GradientBoostingClassifier	normalize, select K best (1000)	0.1498	412.17
GradientBoostingClassifier	standard scaler, select K best (1000)	0.1376	417.99
GradientBoostingClassifier	select K best Johnson Lindenstrauss	-	-
KNeighborsClassifier	select K best (1000)	0.0383	185.59
KNeighborsClassifier	normalize, select K best (1000)	0.0383	223.07
KNeighborsClassifier	**standard scaler, select K best (1000)**	0.2607	72.35
KNeighborsClassifier	normalize, standard scaler, select K best (1000)	0.2607	74.62
KNeighborsClassifier	select K best Johnson Lindenstrauss	0.062	204.78
LogisticRegression	select K best (1000)	0.3777	162.44
LogisticRegression	normalize, select K best (1000)	0.3777	188.62
LogisticRegression	normalize, standard scaler, select K best (1000)	0.2977	82.94
LogisticRegression	standard scaler, select K best (1000)	0.2977	86.38
LogisticRegression	**select K best Johnson Lindenstrauss**	0.4838	169.10
RandomForestClassifier	**select K best (1000)**	0.3087	77.57
RandomForestClassifier	normalize, select K best (1000)	0.3073	80.60
RandomForestClassifier	normalize, standard scaler, select K best (1000)	0.2457	82.45
RandomForestClassifier	standard scaler, select K best (1000)	0.2765	94.14
RandomForestClassifier	select K best Johnson Lindenstrauss	0.3826	157.77
SGDClassifier	normalize, standard scaler, select K best (1000)	0.293	77.96
SGDClassifier	select K best (1000)	0.3063	80.68
SGDClassifier	standard scaler, select K best (1000)	0.293	83.72
SGDClassifier	normalize, select K best (1000)	0.3063	86.53
SGDClassifier	**select K best Johnson Lindenstrauss**	0.4090	77.66
SVC	**normalize, standard scaler, select K best (1000)**	0.2266	215.94
SVC	standard scaler, select K best (1000)	0.2264	220.90
SVC	standard scaler, select K best (1000)	0.2264	220.90
SVC	normalize, select K best (1000)	-	-
SVC	select K best (1000)	-	-
SVC	standard scaler, select K best Johnson Lindenstrauss	0.2595	534.28

Table 7: Experiments Newsgroups data set

D Order of executing algorithms for each task

Regression	**Binary classification**	**Multi-class classification**	**Multi-label classification**
Linear Regression	RandomForestClassifier	DecisionTreeClassifier	KNeighborsClassifier
KNeighborsRegressor	DecisionTreeClassifier	RandomForestClassifier	RandomForestClassifier
Lasso	LogisticRegression	LogisticRegression	LogisticRegression
Ridge	BernoulliNB	GaussianNB	GaussianNB
RandomForestRegressor	GradientBoostingClassifier	BaggingClassifier	SGDClassifier
BayesianRidge	KNeighborsClassifier	KNeighborsClassifier	BernoulliNB
DecisionTreeRegressor	GaussianNB	BernoulliNB	AdaBoostClassifier
GradientBoostingRegressor	AdaBoostClassifier	SGDClassifier	GradientBoostingClassifier
ElasticNet	BaggingClassifier	GradientBoostingClassifier	DecisionTreeClassifier
BaggingRegressor	SGDClassifier	AdaBoostClassifier	BaggingClassifier
SGDRegressor	SVC	SVC	SVC
AdaBoostRegressor			
SVR			

Table 8: Order of executing algorithms for each task

E Default Hyper-parameters

Classifier	**Adapted default hyper-parameters**
AdaBoostClassifier	n estimators = 30
BaggingClassifier	n estimators = 30
BernoulliNB	
DecisionTreeClassifier	
GaussianNB	
GradientBoostingClassifier	n estimators = 30 (5 for multi-class)
KNeighborsClassifier	
LogisticRegression	
RandomForestClassifier	n estimators = 30
SGDClassifier	n iter = $10^6/train\_num$, loss = log
SVC	probability = True
DecisionTreeRegressor	
SVR	
AdaBoostRegressor	n estimators = 30
BaggingRegressor	n estimators = 30
RandomForestRegressor	n estimators = 30
GradientBoostingRegressor	n estimators = 30, warm start = True
KNeighborsRegressor	
Linear Regression	
BayesianRidge	normalize = True
SGDRegressor	warm start = True
Ridge	fit intercept = True, normalize = True
Lasso	normalize = True, warm start = True
ElasticNet	normalize = True, warm start = True

Table 9: Default hyper-parameters used

F Results of programs

Name	**Time (seconds)**	**Score**	**Algorithm**
Dorothea	479,82	0,84	Random Forest Classifier
Electricity	403,04	0,66	Random Forest Classifier
Spambase	403,66	0,92	Bagging Classifier
Sick	147	0,85	Bagging Classifier
Diabetes	55,24	0,65	Logistic Regression
Hepatitis	66,65	0,79	Kneighbors Classifier
Digits	448,89	0,93	Random Forest Classifier
Newsgroups	441,26	0,34	Random Forest Classifier
Nursery	476,73	0,63	Bagging Classifier
Kropt	553,59	0,82	Bagging Classifier
Splice	436,47	0,92	Bagging Classifier
Adult	399,12	0,9	Random Forest Classifier
Yeast	545,82	0,41	Kneighbors Classifier
Mediamill	0	0	0
Emotions	403,79	0,5	GaussianNB
Cadata	233,62	0,74	Random Forest Regressor
Kin8nm	405,28	0,84	SVR
Wind	404,52	0,77	SVR
Houses	407,24	0,76	Bagging Regressor
Stock	57,23	0,99	Kneighbors Regressor

Table 10: Results own program

Name	**Time (seconds)**	**Score**	**Algorithm**
Dorothea	49,99	0,76	RandomForest Classifier
Electricity	5,15	0,66	RandomForest Classifier
Spambase	4,03	0,91	RandomForest Classifier
Sick	1,71	0,84	RandomForest Classifier
Diabetes	3,35	0,62	RandomForest Classifier
Hepatitis	2,72	0,83	RandomForest Classifier
Digits	63,93	0,93	RandomForest Classifier
Newsgroups	291,53	0,28	RandomForest Classifier
Nursery	6,39	0,68	RandomForest Classifier
Kropt	8,29	0,72	RandomForest Classifier
Splice	1,8	0,88	RandomForest Classifier
Adult	5,17	0,89	RandomForest Classifier
Yeast	4,15	0,38	RandomForest Classifier
Mediamill	405,42	0,12	RandomForest Classifier
Emotions	1,85	0,63	RandomForest Classifier
Cadata	2,55	0,73	RandomForest Regressor
Kin8nm	2,1	0,63	RandomForest Regressor
Wind	2,21	0,76	RandomForest Regressor
Houses	3,42	0,76	RandomForest Regressor
Stock	1,47	0,97	RandomForest Regressor

Table 11: Results Random Forest program

Name	**Time (seconds)**	**Score**	**Algorithm**
Dorothea	70,6	0,54	DecisionTree Classifier
Electricity	5,09	0,63	DecisionTree Classifier
Spambase	1,83	0,9	DecisionTree Classifier
Sick	1,72	0,88	DecisionTree Classifier
Diabetes	1,59	0,59	DecisionTree Classifier
Hepatitis	1,45	0,77	DecisionTree Classifier
Digits	83,12	0,85	DecisionTree Classifier
Newsgroups	0	0	DecisionTree Classifier
Nursery	5,25	0,69	DecisionTree Classifier
Kropt	8,28	0,79	DecisionTree Classifier
Splice	1,77	0,86	DecisionTree Classifier
Adult	4,25	0,87	DecisionTree Classifier
Yeast	6,24	0,39	DecisionTree Classifier
Mediamill	0	0	DecisionTree Classifier
Emotions	3,86	0,53	DecisionTree Classifier
Cadata	3,7	0,51	DecisionTree Regressor
Kin8nm	3,56	0,22	DecisionTree Regressor
Wind	1,83	0,54	DecisionTree Regressor
Houses	3,87	0,67	DecisionTree Regressor
Stock	1,44	0,96	DecisionTree Regressor

Table 12: Results Decision Tree program

Name	**Time (seconds)**	**Score**	**Algorithm**
Dorothea	464,37	0,12	Bagging Classifier
Electricity	448	0,66	Random Forest Classifier
Spambase	425,16	0,94	Gradient Boosting Classifier
Sick	449	0,87	Random Forest Classifier
Diabetes	420,37	0,62	Gradient Boosting Classifier
Hepatitis	430,78	0,81	Random Forest Classifier
Digits	342,54	0,64	Gradient Boosting Classifier
Newsgroups	756,47	0,31	Bagging Classifier
Nursery	452,22	0,64	Random Forest Classifier
Kropt	451,59	0,8	Random Forest Classifier
Splice	436,49	0,92	Random Forest Classifier
Adult	335,52	0,91	Random Forest Classifier
Yeast	429,69	0,42	Gradient Boosting Classifier
Mediamill	614,04	0,05	Gradient Boosting Classifier
Emotions	420,53	0,62	Gradient Boosting Classifier
Cadata	222,35	0,76	Random Forest Regressor
Kin8nm	439,39	0,67	Random Forest Regressor
Wind	444,09	0,78	Random Forest Regressor
Houses	431,48	0,77	Random Forest Regressor
Stock	429,38	0,98	Random Forest Regressor

Table 13: Results Skeleton program

Bibliography

[1] Mitchell, T. M. (2006). *The discipline of machine learning* (Vol. 17). Carnegie Mellon University, School of Computer Science, Machine Learning Department.

[2] Guyon, I., Bennett, K., Cawley, G., Escalante, H. J., Escalera, S., Ho, T. K., Macia, N., Ray, B., Saeed, M., Statnikov, A., & Viegas, E. *Design of the 2015 ChaLearn AutoML Challenge.*

[3] Thornton, C., Hutter, F., Hoos, H., & Leyton-Brown, K. (2012). *Auto-WEKA: Automated selection and hyper-parameter optimization of classification algorithms.* CoRR, abs/1208.3719.

[4] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). *Scikit-learn: Machine learning in Python.* The Journal of Machine Learning Research, 12, 2825-2830.

[5] Kohavi, R. (1995, August). *A study of cross-validation and bootstrap for accuracy estimation and model selection.* In Ijcai (Vol. 14, No. 2, pp. 1137-1145).

[6] Bergstra, J., & Bengio, Y. (2012). *Random search for hyper-parameter optimization.* The Journal of Machine Learning Research, 13(1), 281-305.

[7] Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). *ParamILS: an automatic algorithm configuration framework.* Journal of Artificial Intelligence Research, 36(1), 267-306.

[8] Komer, B., Bergstra, J., & Eliasmith, C. (2014). *Hyperoptsklearn: Automatic hyperparameter configuration for scikitlearn.* In ICML workshop on AutoML.

[9] Maron, O., & Moore, A. W. (1993). *Hoeffding races: Accelerating model selection search for classification and function approximation.* Robotics Institute, 263.

[10] Escalante, H. J., Montes, M., & Sucar, L. E. (2009). *Particle swarm model selection.* The Journal of Machine Learning Research, 10, 405-440.

[11] Rifkin, R., & Klautau, A. (2004). *In defense of one-vs-all classification.* The Journal of Machine Learning Research, 5, 101-141.

[12] Tai, F., & Lin, H. T. (2012). *Multilabel classification with principal label space transformation.* Neural Computation, 24(9), 2508-2542.

[13] Abdi, H., & Williams, L. J. (2010). *Principal component analysis.* Wiley Interdisciplinary Reviews: Computational Statistics, 2(4), 433-459.

[14] Ekenel, H. K., & Stiefelhagen, R. (2006, June). *Analysis of local appearance-based face recognition: Effects of feature selection and feature normalization.* In Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on (pp. 34-34). IEEE.

[15] Jain, A.K., & Dubes, R.C. *Algorithms for Clustering Data*, Prentice-Hall advanced reference series, 1988.

[16] Aksoy, S., & Haralick, R. M. (2001). *Feature normalization and likelihood-based similarity measures for image retrieval.* Pattern recognition letters, 22(5), 563-582.

[17] Cheng, J., Hatzis, C., Hayashi, H., Krogel, M. A., Morishita, S., Page, D., & Sese, J. (2002). *KDD Cup 2001 report.* ACM SIGKDD Explorations Newsletter, 3(2), 47-64.

[18] Dasgupta, S., & Gupta, A. (1999). *An elementary proof of the Johnson-Lindenstrauss lemma.* International Computer Science Institute, Technical Report, 99-006.

[19] Furey, T. S., Cristianini, N., Duffy, N., Bednarski, D. W., Schummer, M., & Haussler, D. (2000). *Support vector machine classification and validation of cancer tissue samples using microarray expression data.* Bioinformatics, 16(10), 906-914.

[20] Kapp, M. N., Sabourin, R., & Maupin, P. (2012). *A dynamic model selection strategy for support vector machine classifiers.* Applied Soft Computing, 12(8), 2550-2565.

[21] Cunningham, P., & Delany, S. J. (2007). *k-Nearest neighbour classifiers.* Multiple Classifier Systems, 1-17.

[22] Kohavi, R. (1996, August). *Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid.* In KDD (pp. 202-207).

[23] Safavian, S. R., & Landgrebe, D. (1991). *A survey of decision tree classifier methodology.* IEEE transactions on systems, man, and cybernetics, 21(3), 660-674.

[24] McCallum, A., & Nigam, K. (1998, July). *A comparison of event models for naive bayes text classification.* In AAAI-98 workshop on learning for text categorization (Vol. 752, pp. 41-48).

[25] John, G. H., & Langley, P. (1995, August). *Estimating continuous distributions in Bayesian classifiers.* In Proceedings of the Eleventh conference on Uncertainty in artificial intelligence (pp. 338-345). Morgan Kaufmann Publishers Inc.

[26] Bottou, L. (2010). *Large-scale machine learning with stochastic gradient descent.* In Proceedings of COMPSTAT'2010 (pp. 177-186). Physica-Verlag HD.

[27] Dietterich, T. G. (2000). *Ensemble methods in machine learning.* In Multiple classifier systems (pp. 1-15). Springer Berlin Heidelberg.

[28] Opitz, D., & Maclin, R. (1999). *Popular ensemble methods: An empirical study.* Journal of Artificial Intelligence Research, 169-198.

[29] Prasad, A. M., Iverson, L. R., & Liaw, A. (2006). *Newer classification and regression tree techniques: bagging and random forests for ecological prediction.* Ecosystems, 9(2), 181-199.

[30] Moisen, G. G., Freeman, E. A., Blackard, J. A., Frescino, T. S., Zimmermann, N. E., & Edwards, T. C. (2006). *Predicting tree species presence and basal area in Utah: a comparison of stochastic gradient boosting, generalized additive models, and tree-based methods.* ecological modelling, 199(2), 176-187.

[31] Tibshirani, R. (1996). *Regression shrinkage and selection via the lasso.* Journal of the Royal Statistical Society. Series B (Methodological), 267-288.

[32] Zou, H., & Hastie, T. (2005). *Regularization and variable selection via the elastic net.* Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67(2), 301-320.

[33] King, G., & Zeng, L. (2001). *Logistic regression in rare events data.* Political analysis, 9(2), 137-163.

[34] Neal, R. M. (1997). *Monte Carlo implementation of Gaussian process models for Bayesian regression and classification.* arXiv preprint physics/9701026.