Masterarbeit im Studiengang Angewandte Informatik

Variability-aware Software Defect Prediction

Chris Wieczorek

259109

wieczo@uni-hildesheim.de

Betreuer:
Prof. Dr. Klaus Schmid, SSE
M.Sc. Sascha El-Sharkawy, SSE

Arbeitsgruppe Software Systems Engineering • Institut für Informatik
Universität Hildesheim • Universitätsplatz 1 • D-31134 Hildesheim

Eigenständigkeitserklärung

Erklärung über das selbstständige Verfassen von "Variability-aware Software Defect Prediction"

Ich versichere hiermit, dass ich die vorstehende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der obigen Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem Fall durch die Angabe der Quelle bzw. der Herkunft, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet und anderen elektronischen Text- und Datensammlungen und dergleichen. Die eingereichte Arbeit ist nicht anderweitig als Prüfungsleistung verwendet worden oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Hildesheim, den July 10, 2020

Chris Wieczorek

Abstract

Software defects can cause tremendous costs at several stages of a software project due to the necessary time to find and correct the issues [JB11]. This issue can affect the flawlessness or even the success of a software project due to exceeded time limits, budgets, or just poor usability due to yet present bugs. Previous research has shown that especially method-level software defect prediction is the most meaningful approach to support the developers in charge and contribute to a reduction of defects and consequently the arising costs [SHAJ12]. This study aims to investigate the suitability of a variety of 692 static code metrics for method-level software defect prediction on the case of the Linux kernel. The used metrics include multiple novel variability-aware code metrics, which aim to provide useful approximations of the code's complexity that arose by implemented variability. Furthermore, additional variability metrics are used in order to weight some of the defined code metrics with respect to involved variability variables. A conducted literature review has shown that neither the use of variability-aware code metrics nor the use of variability metrics were applied and evaluated as input features for software defect prediction so far. This study examines the usefulness of the provided software metrics, but it also provides a comprehensive comparison of achieved prediction capabilities across the different types of software metrics. Guided by studied literature on software defect prediction research, two sets of experiments were conceptualized, performed, and precisely analyzed. The first set of experiments has built meaningful feature subsets of the provided software metrics according to their variability-awareness and whether they utilize an additional weighting by variability metrics or not. These feature subsets were used to evaluate the suitability of the individual types of metrics for software defect prediction. The second set of experiments was built upon the results and experimental environment of the first set of experiments. It aimed to combine the defined feature subsets in all possible ways to find the overall best predictions capabilities. In addition to finding the most suitable software metric combination for prediction purpose, these results served as a base to analyze and extract univariate information about particularly important software metrics for the prediction. The results indicate that measures of code variability are much better predictors for defects than traditional metrics like *LoC* or *McCabe's Cyclomatic Complexity*. Based on these findings, it is recommended that existing approaches of defect prediction should consider additional measures of code variability to improve their results. Common coding guidelines should pay additional attention to regulate and simplify implemented code variability in order to reduce the risk of software defects. Further research is necessary to develop new ways of measuring code variability across different programming languages and cross-validate the findings of this thesis on other software projects.

Contents

List of Figures

List of Tables

Source code index

1 Introduction

This chapter gives a general introduction to the topic of software defect prediction (SDP) and the content of this thesis.

SDP aims to explore patterns in software code that indicate high risks of software defects based on a certain number of specific observations. These observations might be static code metrics (SCM) or other information about a software module of arbitrary size. Software modules could be methods, functions, classes, files, or any other logically separated parts of a software project. The terms *method* and *function* depend on the type of used programming language. *Method* or *member function* is commonly used for functions that are associated to an object of a certain class. The term *function* is typically used for procedures that are not associated with an object like static code functions. Recent research in the area of SDP has primarily used the terminology of *method*-level SDP because it mostly examined software projects based on object-oriented programming languages. The data provided to this study is based on C-code, and hence it contains only functions. Even though the terminology is different, both describe a similar level of granularity. Thus, sometimes the term method is used in the context of other research or SDP in general, but the term function is used when referencing the content of data provided to this study. Both basically describe the same.

A software defect, fault, bug, or error describes any flaw or imperfection in a software product. The focus of this SDP study is on clear compilation errors. A code section is considered as buggy or defective if it contains code that causes the program, system, or application to behave incorrectly or unexpectedly.

1.1 Motivation

SDP is a powerful tool in software engineering, which can help to improve the quality of code. High numbers of software defects cause a high amount of expensive finding and correcting work, and hence increase the costs of a software project. Jones & Bonsignour reported that rising costs due to finding and fixing defects are even one of the most expensive activities in a software development project [JB11]. Shihab et al. reported that method-level software defect prediction is the most suitable approach to support developers because of its fine-grained granularity-level [SHAJ12]. Other approaches that predict defects on source-file-level or commit-level granularity, do not reduce the necessary effort to finally locate a defect significantly [SHAJ12]. Furthermore, Menzies et al. reported that defect prediction approaches may achieve higher probabilities to detect a defect than manually performed code review [MMT+10].

Even though research has found that method-level SDP can provide significant benefits to the software development process, it also stated that the status quo of defect prediction on method-level is an unsolved problem and severely needs further research to achieve groundbreaking results [LBMP08] [PPB18].

The software defect prediction approach in this thesis uses a variety of novel SCMs on method-level. Most of the used metrics are variability-aware and consider implemented code variability within functions. The Linux kernel was selected as a project to analyze because of its large size and the high amount of applied code variability due to its product line nature. Prior literature review has shown a severe gap in method-level defect prediction using such variability-aware code metrics. The majority of studies that conducted experiments of method-level defect prediction used traditional code metrics like McCabe's Cyclomatic Complexity [McC76], Halstead measures [H+77], or additional process metrics.

According to the best of my knowledge, an approach using measures of contained code variability in a software product line as input features for SDP has not been examined so far.

1.2 Goals

This section defines and describes the research questions of this thesis. The research questions are used to achieve the main goal of research while also investigating some specific assumptions and achieve clear answers to them.

The general aim of the study is to investigate the suitability of a variety of 692 software metrics. These metrics contain multiple novel variability-aware code metrics, which may provide useful approximations of code complexity due to considering and analyzing implemented code variability. The following paragraphs provide definitions and explanations of the examined research questions.

[RQ1] *Does the utilization of variability-aware code metrics achieve significantly better results than non-variability-aware code metrics?*

The first research question aims to examine the assumption that implemented code variability could increase the code complexity, and hence it could increase the difficulty for a developer to create new bug-free code. The provided metrics have to be partitioned into groups according to their variability-awareness to evaluate these groups individually. The variability-awareness indicates whether a code metric considers C-Pre-Processor (CPP) code or other methods to measure code variability. Any statistical superiority has to be validated by Welch-tests. The results relate to a certain set of suitable performance metrics, whereas one of these is considered as the primary metric. The most suitable performance metric will be ascertained as part of the literature review and further research. There are many possible metrics available, but not all are suitable for this sort of prediction task.

[RQ2] *Does the weighting of code metrics by additional variability metrics achieve significantly better results than non-weighted code metrics?*

This second research question aims to examine the assumption that an additional weighting of code metrics by variability metrics provides more accurate approximations of the code's complexity, and hence these metrics could produce more expressive input features than non-weighted ones. The provided metrics must be separated into groups according to their use of variability metrics to evaluate the different types of variability-awareness individually.

[RQ3] *Which software metrics are most suitable for software defect prediction?*

This third research question aims to extract univariate information about particularly important code metrics as input features for SDP on the Linux kernel. The ulterior motive of such information is to derive clear recommendations for common coding guidelines contributing to the reduction of software defects. The method of how to measure a code metric's suitability will be elaborated as part of the research.

[RQ4] *What is the best-performing configuration in this software defect prediction task?*

The final research question aims for the optimal configuration in this case of SDP. A complete configuration includes any steps of pre-processing, the choice of classification algorithm itself, and any algorithm-specific hyperparameters. Finally, it also includes the selection of input features. The performance of a model is measured by performance metrics.

These describe the prediction capability of a model from different perspectives. Depending on the prediction task, the optimal performance metric can vary.

1.3 Thesis Structure

The next Chapter 2 provides a variety of essential knowledge for this thesis. It is divided into multiple sections. The first section introduces the applied methodology in this study. The following sections introduce some fundamental terminology, concepts, algorithms, and evaluation methods of machine learning and data science. This chapter covers any necessary knowledge to understand the subsequent chapters of theoretical and practical work. Chapter 3 gives a summary of the conducted literature review, which was the first step of diving into the research area of SDP. The chapter starts with a very general introduction to the topic and becomes more specific afterward. It introduces some important and influential papers and discusses their findings and claims. It ends up with a summary of learned lessons from the review. A variety of information could be extracted from previous research, which helped to work properly in this thesis. The next Chapter 4 is the very first step of the applied project methodology. The chapter is meant to ensure a complete understanding of the use case and purpose of the project to be performed. Hence, it gives additional information about the reason, purpose, and environment of this study. Chapter 5 gives a detailed analysis of the provided data. The chapter is divided into three sections. The first section describes the content of the data. In this case, the data primarily contains a large set of software metrics. All these metrics are described in this section to ensure a clear understanding of what data are used in this study. The subsequent section describes the process of cleaning and aggregating the provided data to get a single useful dataset for further analyses and experiments. The final part of Chapter 5 is a summary of a conducted data exploratory analysis. Such an analysis is a common first step of a data science project to examine the data from a more data science point of view and reveal obstacles and helpful or problematic properties of the data. The next Chapter 6 describes the complete first set of experiments. This description includes any important information about the aimed goals, the concept of how to achieve them, the elaboration of necessary steps of pre-processing as well as details of the model construction, optimization, and final implementation. This first set of experiments is aimed to gain the necessary information to answer [RQ1], [RQ2], and [RQ3]. The last part of Chapter 6 is a comprehensive analysis of the results to prepare them in order to clearly answer the research questions later. The analysis sections typically end up with a list summarizing all important findings of the current analysis subsection concisely. Chapter 7 builds upon the findings of the previous set of experiments and follows the same schema as Chapter 6. It is aimed to gain the necessary information to answer [RQ3] and [RQ4]. Chapter 8 shows the final discussion of the results and findings of the previous work. It catches up on the findings of the analysis sections and puts them into the context of the defined research questions. It provides clear answers to the questions as well as some further interpretations and concerns about the findings. Furthermore, this chapter provides an aligning of own findings with the claims of other research. The last two parts of Chapter 8 describe the limitations and threats to the validity of this study. These are parts of research that were beyond the scope of this thesis and could not be covered in this study. The threats to validity are typical threats that arise with empirical research, the use of data collected from a certain number of experiments, and the application of statistical methods and tests.

2 Background

This chapter introduces some general background knowledge about the used terms and concepts within this thesis. The first section introduces the applied methodology of *CRISP-DM*, which served as a baseline to accomplish this data science project. The next sections provide a brief introduction to some fundamental terms, concepts, and algorithms of machine learning. These include explanations of the general types of prediction tasks addressed by *supervised learning* algorithms, the theory, and terminology of *bias and variance* as well as *over- and under-fitting*. The explained algorithms correspond to the applied classification algorithms in this study. These explanations only describe the general working of the algorithms on a top-level and do not cover detailed descriptions of special cases or all recent advances. References are given for more detailed descriptions and definitions. The final section provides an introduction to several performance metrics being relevant in this thesis.

2.1 CRISP-DM Methodology

CRISP-DM[1] (Cross Industry Standard Process for Data Mining) is a standardized methodology for structured accomplishments of data mining projects. It was developed in 1996 by an incorporation of multiple enterprises. The methodology is one of the most widely used models in this domain. It consists of six different phases, which will be introduced in the following paragraphs. Figure 2.1 shows the complete process model of CRISP-DM. This process model does not proceed sequentially but can include multiple repetitions of certain phases.

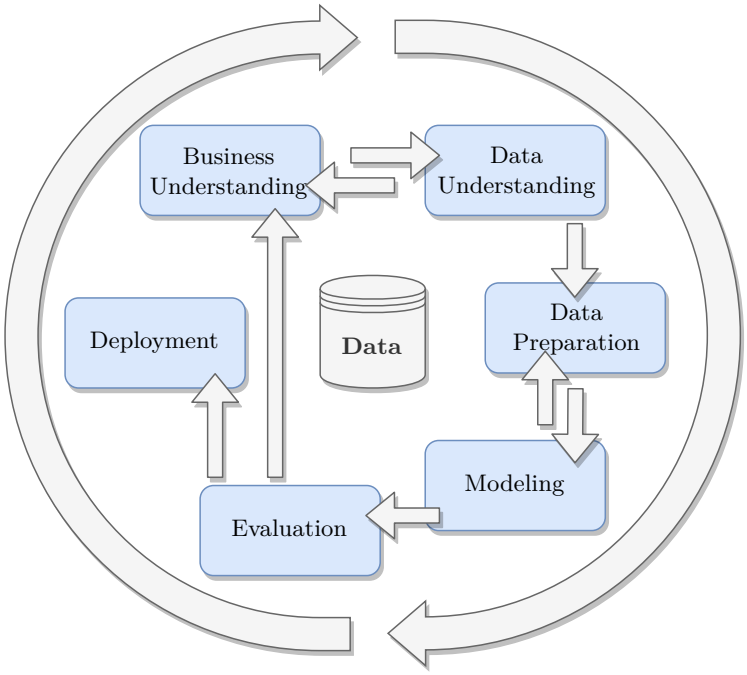


Figure 2.1: CRISP-DM process model.

The first phase of the model is *Business Understanding* and focuses on defining and understanding the project's objectives, purposes, and requirements. This knowledge can be used to form a coarse-grained preliminary project plan on how to achieve those objectives.

[1]Detailed information about CRISP-DM available under: http://www.CRISP-DM.eu

The second phase is called *Data Understanding* and includes processes meant to get familiar with the data. Those processes are a comprehensive interpretation of the meaning of the data and a first data exploration. The previous interpretation and understanding of the meaning can help to formulate hypotheses and assumptions to examine as part of the data exploration. The next phase of *Data Preparation* covers any activities meant to construct but also improve the quality of the dataset. Those activities can include data aggregation, data cleaning as well as other methods of data pre-processing like *resampling* or *normalization*. Hence, many of those activities are strongly related to the subsequent phase of *Modeling*. This phase consists of any activities meant to build models like *model construction* or *hyperparameter optimization* but also a subsequent *model validation*. The validation is realized by a defined set of experiments, which is conducted according to a certain strategy. The modeling is followed by the phase of *Evaluation*, which covers any kind of reviewing activities based on the results of conducted experiments. The final stage is the *Deployment*, which differs from project to project. It could either include a final implementation to run a certain model on live data or just the preparation of a project report.

2.2 Regression and Classification

The two most common types of problems addressed by *supervised learning* algorithms are *regression* and *classification* problems. At first, supervised learning describes a group of algorithms used to learn patterns from a set of labeled samples. The problems differ in the type of their *target variables* $y \in Y$. The target variable is also called *label*, *dependent variable*, or in the case of classification, *class*. In classification, as well as in regression tasks, a set of *input features*, an *input vector*, *predictors*, or *independent variables* are denoted as $x = (x_1, ..., x_n)$. They are used to predict the corresponding target variable y. Any algorithm utilizes a certain approach in order to learn or train a *decision boundary* or *mapping function* $f(x)$. The prediction of a model is denoted as $\hat{y}$ and computed by $\hat{y} = f(x)$. A model learned according to any algorithm is typically called *regressor*, *classifier*, *model*, or *estimator*. In the case of a regression task, the target variables y are typically considered as continuous values $Y \subset \mathbb{R}$. In the case of classification, the labels are either *nominal* or *ordinal* categorical values $Y \subset \mathbb{N}_0$. The most algorithms provide continuous values even in classification tasks. Those continuous values typically correspond to probabilities that a sample belongs to a certain class. They must be converted into discrete numbers in order to provide final label predictions. Those conversions are typically performed by applying a simple threshold function.

2.3 Bias, Variance, Over- & Under-fitting

This section introduces the terms of *bias* and *variance*. Both are properties a model can have. These terms are helpful to better understand the subsequent theory of *over-* and *under-fitting*. Both concepts play an important role across several parts of this thesis.

As a first step, the total error of a prediction model is defined as given in Equation 2.1.

$$Error_{total} = Error_{bias} + Error_{variance} + Error_{irreducible} \tag{2.1}$$

The $Error_{bias}$ and the $Error_{variance}$ are considered as *reducible errors*, which are primarily caused by poor modeling. The $Error_{irreducible}$ cannot be reduced. It is like a measure of included noise in the data. If $Error_{bias}$ and $Error_{variance}$ approach zero, there are only miss-classifications due to data issues left (e.g. anomalies or wrong-labeled samples).

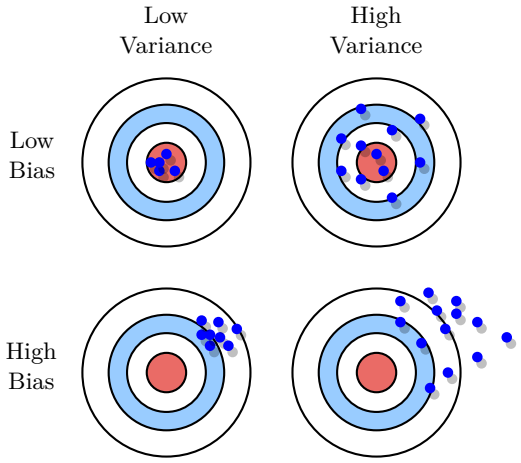


Figure 2.2: Graphical explanation of bias and variance [FR12].

$Error_{bias}$ basically means errors due to an oversimple model. A very high $Error_{bias}$ could be caused by a model utilizing only a small number of input features but having many general assumptions about the underlying pattern. The result is a rather bad performance at training time due to an under-complex decision boundary. This under-complexity is caused by the high number of assumptions without considering the actual input features of a sample. If the training performance is already bad, the testing performance cannot become any better, but also not much worse (i.e. the performance does not vary much). A model having low bias does not have many assumptions about the data, it really pays attention to the input. Figure 2.2 illustrates the result of high and low biases. The red dot in the middle corresponds to the true label of a test sample. The several blue dots correspond to multiple predictions for the same test sample from a similar model trained on slightly different training data. The predictions of a model of high $Error_{bias}$ shift away from the true label because of their poor mapping functions due to its oversimplicity caused by the high number of assumptions. The common conceptual definition of the $Error_{bias}$ is:

> *The error due to bias is taken as the difference between the expected prediction of a model and the true label which one is trying to predict.* [FR12]

The $Error_{variance}$ can be interpreted as the error caused by a model's sensitivity to training data. A very high $Error_{variance}$ could happen when a model utilizes a large number of input features and does not make any assumptions about the data. This can cause significant changes in learned patterns depending on the provided training samples. The result of not making any assumptions is a good performance on training data but a sharp drop in performance on testing data. This happens due to over-complex recognized patterns that are barely generalizable. This over-complexity could be caused by high attention to a high number of input features. A model of low $Error_{variance}$ does not learn significantly different patterns when trained on slightly different training samples. The over-complex recognized patterns also cover several types of noise. A model could have considered uninformative or even misleading features, paid attention to anomalies (i.e. samples that are not representative for the underlying pattern) or considered confusing errors of measurements. An over-complex model strongly relies on the exact data it gets, even though these data are not of high quality. A high variance of predictions is illustrated as the dispersion of the blue dots in Figure 2.2. In the case of such a high variance, the dots are far apart, whereas, in the case of low variance, the dots are very close to each other. The common conceptual definition of $Error_{variance}$ is:

The error due to variance is taken as the variability of a model's predictions for a given test sample. [FR12]

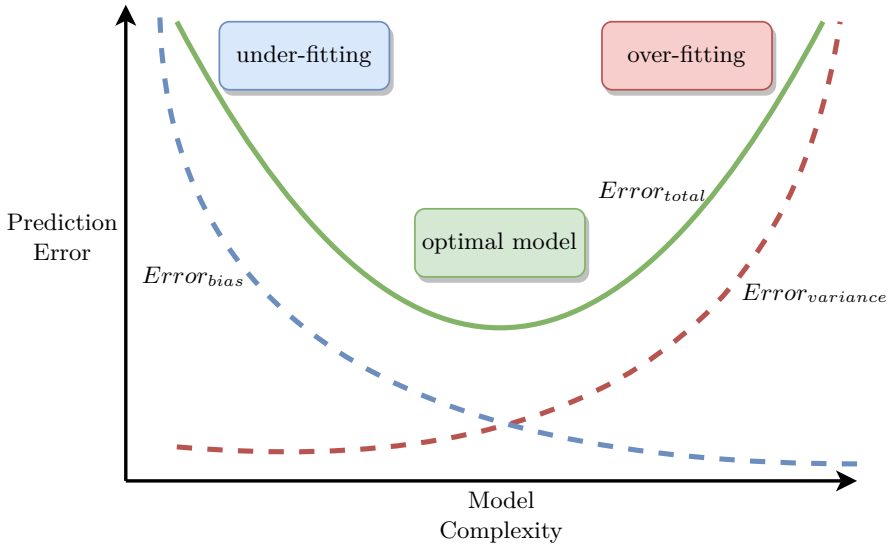
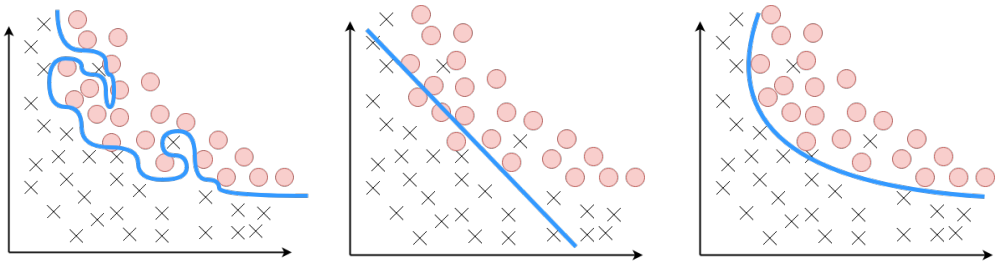


Figure 2.3: Graphical explanation of the bias-variance-trade-off [FR12].

The *Bias-Variance-Trade-Off* describes the interplay of both types of errors. Figure 2.3 illustrates this. A high $Error_{bias}$ typically comes along with a low $Error_{variance}$ because if a model makes many assumptions about an underlying pattern, it acts quite insensitive to the details of a sample (i.e. it is biased). Hence, it does not react with heavy changes in the recognized patterns if the exact training data changes. A high $Error_{variance}$ typically comes along with a low $Error_{bias}$ because if the model pays much attention to the actual input of a sample, it does not need to make any oversimplified assumptions about the data. This means there is an obvious interplay between both properties. A low $Error_{bias}$ means a model is not oversimple due to allowing some assumptions about the data. A low $Error_{variance}$ means a model has not learned over-complex patterns, which are barely generalizable. Hence, the amount of $Error_{bias}$ and $Error_{variance}$ depends on the complexity of learned patterns.

This paragraph introduces the terms *over-fitting* and *under-fitting*. Over-fitting basically means that a model has learned over-complex patterns and has lost its generalization ability (i.e. it has decreased its $Error_{bias}$ while it has increased its $Error_{variance}$). The model has moved away from being a good predictor of unseen data to an expert in the interpolation of seen data. The model has not recognized the general underlying pattern and does not have well-suited assumptions. This leads to an (almost) zero-biased model. The fundamental reason is that the model has become too complex. A model can also underfit, which means it oversimplifies the underlying pattern. This results in poor predictions for seen and unseen data. The reason is that the model has become too simple. Figure 2.4a and 2.4b show how over-fitted and under-fitted decision boundaries could look like in two-dimensional feature space. The appropriate decision boundary shown in Figure 2.4c pays attention to its input features but does also make necessary assumptions about the data to keep it simple. This allows the model to recognize the underlying pattern by the input features while ignoring non-representative samples contrary to its assumptions.

The *Bias-Variance Dilemma* happens when trying to minimize the $Error_{bias}$ and $Error_{variance}$ simultaneously because they affect each other. Both properties start to induce a reducible error at a certain point but also contribute to improving the testing performance. The



(a) Over-fitting (b) Under-fitting (c) Appropriate fitting

Figure 2.4: Differently fitted decision boundaries.

Dilemma is the process of finding the right balance between both properties while minimizing the $Error_{total}$.

A perfect model has minimized its $Error_{bias}$ and its $Error_{variance}$ with respect to the $Error_{total}$ on testing data. This results in making a few assumptions about the data (i.e. low bias) while considering only important signals (i.e. low variance) of a sample. This could also be described as a model that is not learning an oversimple decision boundary (i.e. not under-fitted) while also not learning an over-complex decision boundary (i.e. not over-fitted). Hence, both terminologies describe a similar state of a model from different perspectives. An over-fitted model has a very low $Error_{bias}$ but a high $Error_{variance}$ because it has learned an over-complex decision boundary for a simple problem. An under-fitted model has a low $Error_{variance}$ but a high $Error_{bias}$ because it has learned an oversimple decision boundary.

Typical reasons for high $Error_{bias}$ or high $Error_{variance}$ are given in the following list:

- **Inappropriate number of input features** A model might have insufficient important features available (i.e. it is forced to make many assumptions due to lacking information), or a model might have too many poor features available (i.e. a model does not make any well-suited assumptions but pays attention to noisy input).
- **Inappropriate algorithmic complexity** A problem might require a non-linear decision boundary of a high polynomial degree, but the used classifier provides only linear capabilities (i.e. it is forced to make many assumptions due to lacking capabilities). On the other side, an algorithm could be provided with too high complexity that it learns unnecessarily complicated boundaries for simple problems (i.e. a model does not need to make any well-suited assumptions because it can learn over-complex boundaries).
- **Inappropriate training time** A model could have learned good patterns, but constant fine-tuning led to over-complexity over time (i.e. a model had good assumptions initially but discarded them later). On the other side, a model's training could be aborted before it has learned a proper decision boundary (i.e. a model was changing from oversimple assumptions to good patterns but was interrupted too early).

2.4 Naive Bayes

Naive Bayes (NB) is a supervised learning algorithm that makes predictions based on prior probabilities and likelihoods (i.e. conditional probabilities). In contrast to other machine learning algorithms, NB does not need to learn for multiple iterations. The name *naive* Bayes stems from the naive assumption about conditional independence of all features $x = (x_1, ..., x_n)$. This assumption allows the classifier to ease the computation of posterior

probabilities because $P(x_i \mid y, x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$ is naively considered as $P(x_i \mid y)$. The algorithm applies *Bayes' Theorem* to compute posterior probabilities for all labels $y \in Y$ given a feature vector x. The final prediction $\hat{y}$ corresponds to the highest computed posterior probability of label $y \in Y$. Hence, it follows the classification rule given in Equation 2.2.

$$\hat{y} = \underset{y}{\operatorname{argmax}} P(y \mid x_1, ..., x_n) \tag{2.2}$$

The posterior probability $P(y \mid x_1, ..., x_n)$ can be computed using the *Bayes' Theorem* given in Equation 2.3. It gives a probability estimation according to Equation 2.4.

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) \cdot P(B \mid A)}{P(B)} \tag{2.3}$$

$$P(y \mid x_1, ..., x_n) = \frac{P(y) \cdot P(x_1, ..., x_n \mid y)}{P(x_1, ..., x_n)} \tag{2.4}$$

Considering the naive assumption of feature independence, Equation 2.4 can be simplified to the final Equation 2.5.

$$P(y \mid x_1, ..., x_n) = \frac{P(y) \cdot \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, ..., x_n)} \tag{2.5}$$

In order to determine the likelihood of continuous features x where $x \in \mathbb{R}^n$, a Gaussian kernel is applied and assumes a normal distribution, which is shown in Equation 2.6, where *exp* denotes the exponential function $x \to e^x$. σ_y and μ_y denote mean and standard deviation of a continuous variable $x_i \in x$ for a given target $y \in Y$. The abbreviation *NBG* denotes the NB algorithm applying the Gaussian kernel.

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \cdot exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.6}$$

2.5 Decision Tree

Decision trees are another group of supervised learning algorithms to come from a set of input features $x = (x_1, ..., x_n)$ to a prediction of the corresponding target variable y. Decision trees are applicable to regression and classification problems. One of the most common algorithms of decision tree learning is CART (Classification And Regression Tree) [BFSO84]. Other common learning approaches are *ID3* [Qui86], *C4.5* [Qui14], or *MARS* [Fri91].

A decision tree is typically structured from upside down and starts at its *root node* as shown in Figure 2.5. The following nodes are considered as *internal nodes* that are connected to other nodes above and below. The lowest nodes are considered as *leaf nodes* and correspond to the final predictions of a tree. The connections between nodes are named as *branches*. Each non-leaf node applies a test on a certain feature at a certain value (e.g. $x_3 <= 30$). Each branch corresponds to the outcome of a nodes test. Whereas the root node yet contains the entire dataset, and any nodes below contain only subsets of the data depending

on the performed tests above. Hence, any node and any performed test divides the data until the algorithm converges.

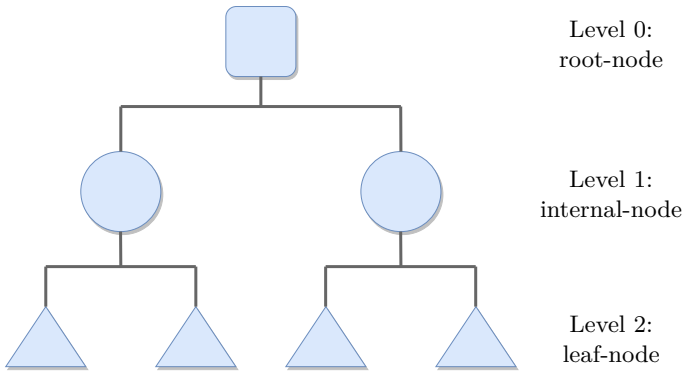


Figure 2.5: Schematic structure of a decision tree.

Multiple algorithms are used to learn decision trees. All share a similar approach of (1) propose multiple splits, (2) evaluate all of them considering a certain *splitting criterion*, (3) select the best split, and (4) repeat until convergence. The individual algorithms differ in their approaches to create potential splits to evaluate, their applied splitting criteria, and their definition of the point where the algorithm converges. The following paragraph describes how a decision tree is learned using *Gini Impurity*.

The *Gini Index* is applied as the splitting criterion in order to enforce the tree to minimize the likelihood that a randomly chosen sample of a node would be wrongly classified, if the classification would be done randomly, according to the class distribution of the node's subset [RS04]. This means the algorithm strives to achieve as many *pure* leaf nodes as possible because the likelihood of wrong classifications on randomly chosen samples is zero on pure nodes. Hence, the likelihood of wrong classification corresponds to $(1 - likelihood\ of\ correct\ classification)$, which is given in Equation 2.7, where p_i denotes the probability that a sample belongs to a certain class i, and Y represents all possible labels within the dataset.

$$Gini\ Index = 1 - \sum_{i=1}^{Y} (p_i)^2 \tag{2.7}$$

Gini Index yields values between zero and one, where zero denotes the best possible value (i.e. zero likelihood of the wrong classification for randomly chosen samples) and a pure leaf node. A value of one would denote a subset of randomly distributed samples of various classes. A value of 0.5 denotes a node of equally distributed samples of multiple classes. The final prediction of a tree typically corresponds to the probability of a certain label equal to the class distribution of the final leaf node. The computation of the *Gini Index* is an important part of the *Gini Impurity* approach, but the algorithm includes some further steps that are shown in Algorithm 1.

In the case of learning a regression tree, the algorithm works quite similarly. The splitting criterion changes to a certain loss or error function like MSE (Mean Squared Error) or MAE (Mean Absolute Error) as shown in Equation 2.8 and 2.9, where k denotes the total number of samples within a node's subset.

Algorithm 1 Classification tree learning using Gini Impurity

```
for all proposed_splits do
    for all branches do
        nodeSize ← number of samples in this branch's node / number of samples in parent node
        Gini Index ← Compute Gini Index of this branch's node
        Weighted Gini Index ← Gini index · nodeSize
    end for
    Σ weighted Gini Index ← ∑ weighted Gini Index each branch
end for
Best split ← argmin Σ weighted Gini Index
```

$$MSE(y, \hat{y}) = \frac{1}{k} \sum_{j=1}^{k} (y_j - \hat{y}_j)^2 \tag{2.8}$$

$$MAE(y, \hat{y}) = \frac{1}{k} \sum_{j=1}^{k} |y_j - \hat{y}_j| \tag{2.9}$$

The prediction $\hat{y}$ becomes the average of all labels of the samples contained in a leaf node. The best split is always considered as the one providing the lowest error measure.

2.6 Random Forest

Random Forest (RF) [Bre01] is an advanced ensemble technique based on decision trees. Instead of learning a single tree, it learns plenty of uncorrelated trees, lets all of them make predictions, and takes advantage of the simple concept of trusting the majority. The RF algorithm applies two key concepts to learn multiple *uncorrelated* decision trees. On the one hand, all learned trees use a randomly selected input feature subset, and on the other hand, all trees are learned on a randomly selected subset of training samples. These two concepts are the reason why it is called *random* forest. The random sampling of data points is also known as *subsampling with replacement* or *bootstrapping*. The procedure of bootstrapping the data, learning multiple models, and averaging the individual predictions in order to form a final ensemble prediction is known as *bootstrap aggregating* or *bagging*. Hence, RF is a bagging algorithm, but it even extends this concept by randomly selected input feature subsets. These concepts provide RFs to be more robust to outliers and noise, and less prone to over-fitting [Bre01].

Figure 2.6 shows a simplified diagram of the RF algorithm. On the first level, (1) the entire training dataset is bootstrapped to provide training data for the individual trees of the ensemble. (2) Each of these trees utilizes a random selection of features to learn according to the algorithm described in Section 2.5. Once the trees are learned, (3) each tree makes predictions on the testing data. The final step is that (4) all predictions are averaged to the final ensemble prediction. Figure 2.6 illustrates that all trees of the forest are arranged horizontally, and hence there are no dependencies among the trees. This leads to a high parallelizability allowing a very efficient computation.

2.7 Gradient Boosting Trees

Gradient Boosting (GB) [Fri01] [Fri02] is another approach typically utilizing decision trees to form an ensemble and improve the prediction capability of a classifier. It is an advancement of the first boosting algorithm *AdaBoost* (Adaptive Boosting) [FS+96] [FS95],

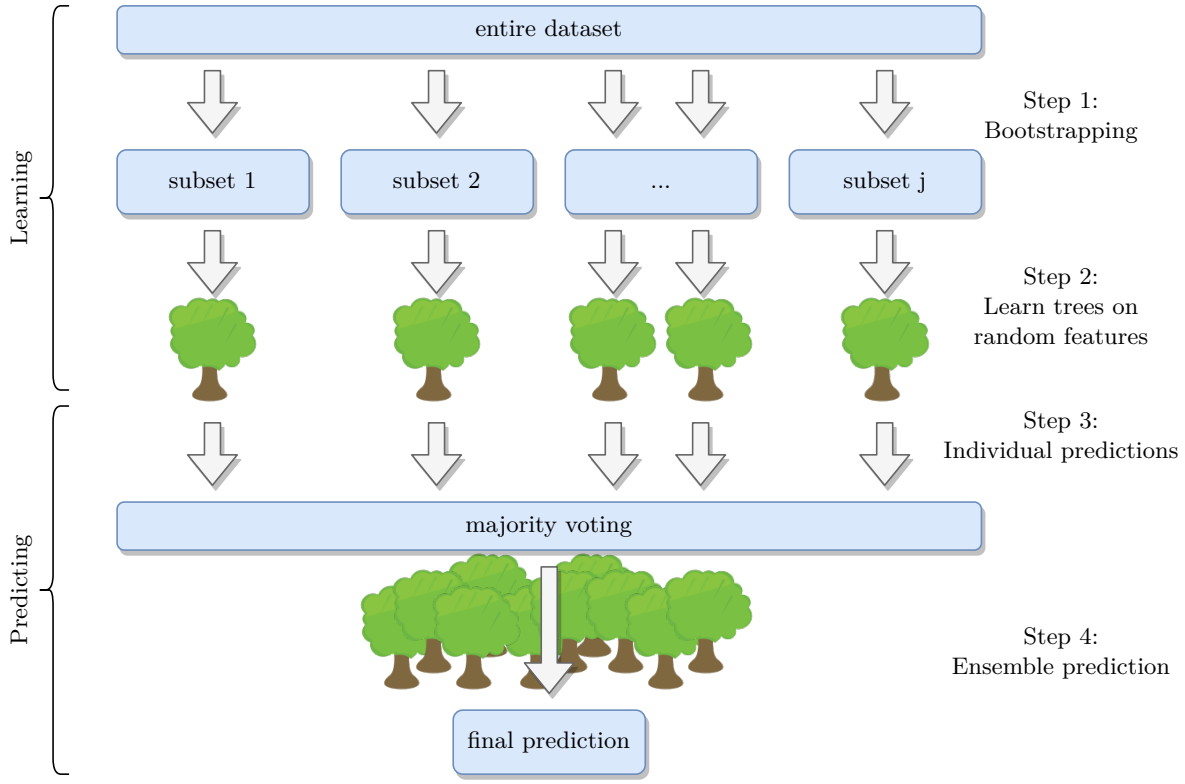


Figure 2.6: Schematic structure of a random forest.

which allows the use of a variety of loss functions to compute the *residuals*. The learned trees within a GB ensemble are called *weak learners*.

In contrast to the previously described RF approach, GB does not use bagging but *boosting*. Boosting basically means learning an ensemble of multiple models but in a sequential manner. Boosting generally tries to build new models that learn from the previous model's mistakes, which are called *residuals* and are computed as given in Equation 2.10. This is the reason why GB-based models are badly parallelizable by nature.

$$residual = actual\ y - current\ prediction \tag{2.10}$$

Figure 2.7 shows a simplified diagram of the GB algorithm. It basically consists of five steps. At first, (1) train a single decision tree, (2) apply the learned tree to predict training samples and (3) calculate the tree's residual and define it as the new target. (4) This procedure is repeated multiple times. Once all trees of the ensemble are learned, it is able to make predictions. Therefore, (5) all testing samples pass the entire ensemble sequentially, and the last tree outputs the final prediction. Hence, the ensemble is built by adding trees that are learned in order to rectify the mistakes made by the previous tree. Due to this, the ensemble needs to use regression trees instead of classification trees because the targeted residuals of a previous tree are real values and no discrete class labels anymore.

2.8 Artificial Neural Networks

The term artificial neural network (NN) encompasses plenty of networks based on a similar concept. This section introduces the *fully connected network* (FCN), *feedforward network*, or *multilayer perceptron*, which was initially proposed in [Ros58]. NNs generally consist

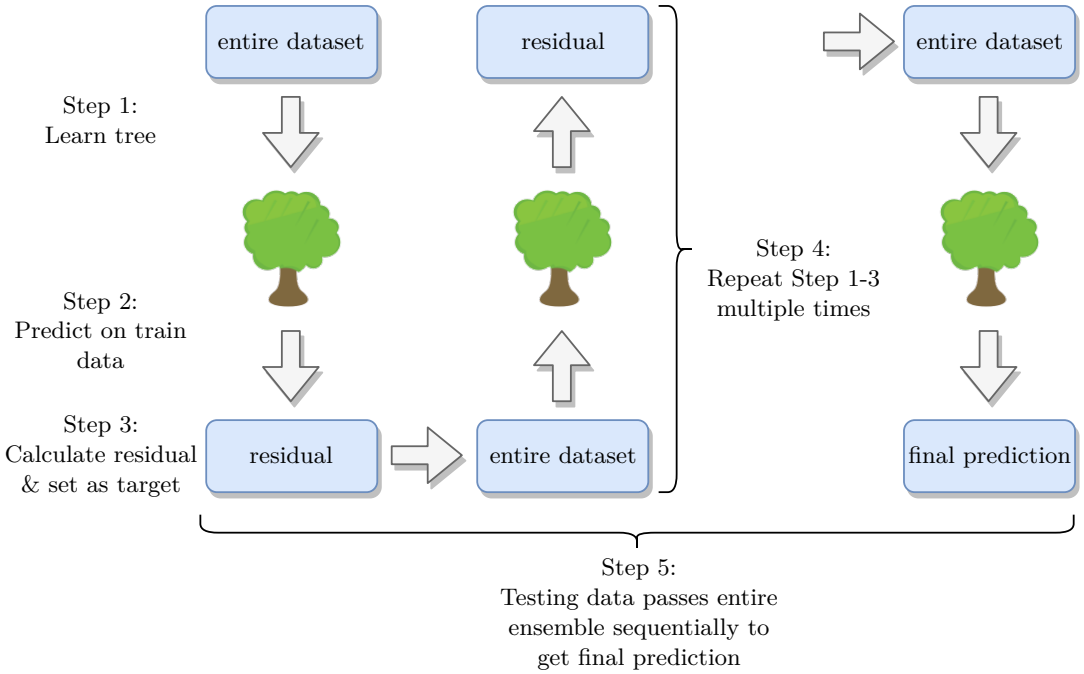


Figure 2.7: Schematic structure of a gradient boosting classifier using decision trees.

of many simple units called *neurons*. These are interconnected by weighted links to form complex structures.

A single neuron works quite simple. It sums up the arriving weighted inputs and processes this sum by an additional function called *activation function*. Figure 2.8 shows a single neuron fed by three inputs x_1, x_2, x_3, and weighted by corresponding coefficients, denoted as w_1, w_2, w_3.

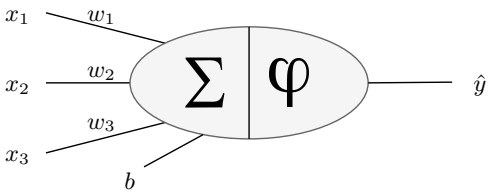


Figure 2.8: Single neuron neural network with bias.

Furthermore, an additional *bias* value b_1 is added to the sum. This value allows the neuron to better fit any pattern without exclusively relying on the neuron's inputs. The resulting sum is processed by the activation function φ and computes the final output, which is denoted as $\hat{y}$. Equation 2.11 and 2.12 describe the computation of the neuron's output.

$$\Sigma = \sum_{i=1}^{j} x_i w_i + b \tag{2.11}$$

$$\hat{y} = \varphi(\Sigma) \tag{2.12}$$

The use of activation functions enables the network to learn non-linear decision boundaries. Two commonly used activation functions are the *sigmoid function* and the *ReLu function*. Both are defined as shown in Equation 2.13 and 2.14.

$$sigmoid(x) = \sigma(x) = \frac{1}{1 + e^{-(x)}} \tag{2.13}$$

$$ReLu(x) = max(0, x) \tag{2.14}$$

The sigmoid function provides values $\in [0, 1]$, which makes it useful when predicting probabilities (e.g. in a classification task), and Relu provides values $\in [0, \infty]$, which makes it useful in the case of a regression task. In the case that the regression task requires negative values as well, a *linear function* is applicable.

FCNs typically contain multiple *layers* with multiple neurons each layer to learn complex decision boundaries. Figure 2.9 shows a small FCN architecture of a single *input layer*, a single *hidden layer*, and a single *output layer*.

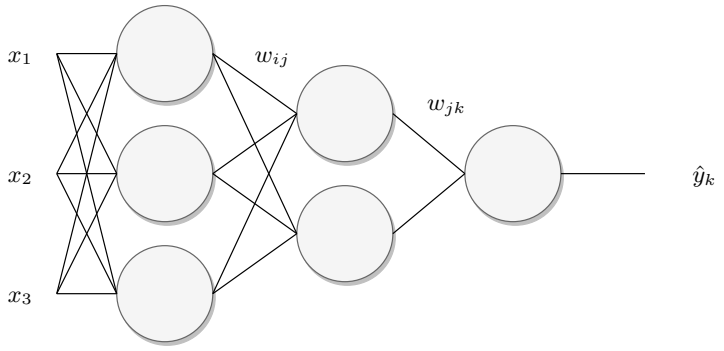


Figure 2.9: FCN architecture.

Any of the shown neurons works as described previously. The used activation functions can differ across the layers. The network is considered as *fully connected* because any neuron from a previous layer is connected to any neuron on an immediately subsequent layer.

In order to *train* the FCN, a *loss function* is used. Loss functions compare the actual networks output $\hat{y}$ with the ground truth y, and provide a measure of distance. The network is adjusting all its weights to minimize a certain loss function. In the case of binary classification, the loss function to minimize is *binary cross-entropy*, which is defined as shown in Equation 2.15, where m denotes the number of all available training samples in this training iteration and $\hat{y}_n$ is the network's prediction for a sample n.

$$L_{BCE}(y, \hat{y}) = \sum_{n=1}^{m} y_n \log(\hat{y}_n) - (1 - y_n) \log(1 - \hat{y}_n) + \lambda R(w) \tag{2.15}$$

The latter term $\lambda R(w)$ adds a *regularization term* to the loss function, which would lead to an increased loss once the learned weights become too high. This term forces the model to keep its weights as low as possible in order to avoid over-fitting and enforce simplicity.

In order to minimize the loss function, the FCN applies an algorithm called *stochastic Gradient Descent* (SGD) and derives gradients for all of its weights. These gradients can be interpreted as pointing in the direction where the given loss function *increases* most rapidly. Hence, the negative gradients point in the direction where the given loss function

decreases most rapidly. All weights are moved in the computed directions in order to minimize the overall loss. Each derivation uses a subset of samples that are applied to the loss function, and hence each training iteration or training step changes the weights to better fit the provided set of samples. This set of samples is called *batch* or *mini-batch*. Equation 2.16 shows the update rule of a single training iteration according to the vanilla gradient descent algorithm, where α denotes the *learning rate*. This learning rate determines the step size to move in the direction of the negative gradients.

$$w_i = w_{i-1} - \alpha \frac{\partial L}{\partial w} \tag{2.16}$$

SGD considers only a subset of samples in a training iteration to simplify the computation and reduce memory consumption by a lot. This also lets the algorithm conducting multiple training iterations within one single *epoch*. One epoch is defined as one complete pass of the entire training data. An NN is commonly trained for many epochs, and hence the network sees all training samples multiple times. On each training iteration, the algorithm changes its weights to better fit the targetted labels of the current mini-batch. Nowadays, FCNs are typically trained using more sophisticated *optimizers* like Adam [KB14] or RMSProp[2]. An optimizer basically means the applied update rule for a single training iteration.

2.9 Performance Metrics

This section provides definitions of the most common performance metrics and evaluation methods in machine learning and statistics.

2.9.1 Confusion Matrix

The confusion matrix is not a performance metric but a common evaluation method to get insight into a model's prediction capability. It provides precise information about how a model predicts, what it is capable of, and what type of samples cause difficulties. In the case of a binary classification problem, the matrix looks like Figure 2.10. The confusion matrix is the foundation for some further measures like *accuracy*, *precision*, *recall*, or *F1-scores*. The four different abbreviations in Figure 2.10 mean *true positives* (TP), *false positives* (FP), *false negatives* (FN), and *true negatives* (TN). False positives are also considered as *type I errors*, and the false negatives are also considered as *type II errors*. The meaning of these values is straightforward, e.g. if a model predicted an actual defect as a defect, the prediction is *true* on a *positive* sample, hence it hits a *true positive*. The remaining values follow the same schema, respectively.

2.9.2 Accuracy

The perhaps most common performance metric is accuracy. Accuracy provides a measure of correctly predicted labels. It only provides the proportion of true predictions among the overall number of classifications. The accuracy does not indicate specific weaknesses or strengths of a model. To get a deeper insight, further measures are necessary. The metric is computed according to the formula given in Equation 2.17.

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.17}$$

[2]RMSProp is an unpublished adaptive optimizer proposed by Geoff Hinton in his lecture. Available under: https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf

		actual class	
		defective	healthy
predicted class	defective	TP	FP
	healthy	FN	TN

Figure 2.10: Confusion matrix of a binary classification problem.

Accuracy does not serve as a robust measure in the case of imbalanced class distribution. If a test set contains only 100 positive samples but 10,000 negative samples, accuracy could lead to wrong conclusions about prediction capabilities. The model could simply predict a negative label for any sample and would achieve an accuracy of about 99%. This value would mean an almost perfect model although it is clearly not.

2.9.3 Precision

Precision is another metric, which is based on the confusion matrix. It provides information about what proportion of all positive predictions is correct. A high precision could indicate that a model found one very reliable pattern to recognize positive samples. This does not necessarily imply a good capability in recognizing positives generally. A model could behave very reserved and predict only positive labels for a minor share of all positive samples. This could lead to a very high precision even though the model has only considered a minority of positive samples. Hence, precision is only a single perspective on the performance and does not indicate any prediction capability if considered solely. Further measures are necessary to get an accurate idea about the model's performance. Precision is computed as shown in Equation 2.18.

$$precision = \frac{TP}{TP + FP} \tag{2.18}$$

2.9.4 Recall

Recall or *sensitivity* is an important addition to precision. As already mentioned, the precision does not provide sufficient information to get an idea about the model's capability if applied solely. The combination of precision and recall provides much better insight. The recall is the proportion of overall found positive samples and is computed according to Equation 2.19.

$$recall = \frac{TP}{TP + FN} \tag{2.19}$$

This measure is necessary to assess precision the right way and vice versa. It is not hard to achieve a high precision regardless of the recall. The classification threshold could be set to 0.99, which would cause the model to only predict a positive label when it is quite sure about this. This is likely to achieve high precision at the expense of recall. Only the consideration of recall as an addition to the precision would reveal the poor performance.

On the other side, a model is likely to achieve a high recall if the classification threshold would be set to 0.01. This would cause a model predicting a positive label if it would have the slightest indication for that. This, in turn, would increase the recall at the expense of precision. Again, only the consideration of both, precision and recall, would reveal the poor performance. As a conclusion, precision and recall are poor performance indicators if used alone, but they provide a good insight into prediction capability if they are used in combination.

2.9.5 F1-Score

The F1-score, *F-measure*, or *Fβ-score* is a performance metric based on achieved measures of recall and precision. The previous subsection already pointed out the trade-off between precision and recall and the importance of considering both values at the same time. The F1-score is the harmonic mean of precision and recall, and it is a specialization of the more generic Fβ-score. The F1-score is computed as given in Equation 2.20, which corresponds to an Fβ-score using $\beta = 1$. The more general measure of Fβ-score is defined as given in Equation 2.21.

$$F1\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.20}$$

$$F\beta\text{-}score = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} \tag{2.21}$$

While the F1-score gives equal priority to precision and recall, the Fβ-score can shift the priority in favor of one of both measures. The β denotes the more-importance of recall over precision, which can be meaningful depending on the purpose of a certain classifier. If $\beta = 2$, the recall would get twice the amount of importance than precision.

2.9.6 AUC-ROC

The AUC-ROC (Area Under the Curve - Receiver Operating Characteristics) [HM82] is a non-threshold-dependent performance metric. While all previously introduced metrics depend on the defined classification threshold, the AUC-ROC does not. It considers a variety of thresholds and tries to approximate the overall prediction capability. AUC simply means the computation of an integral below a function. ROC corresponds to a plot of a model's *True Positive Rates* (TPR) against its *False Positive Rates* (FPR) while considering multiple thresholds. Hence, the ROC-curve is a graphical evaluation method. An example plot is shown in Figure 2.11.

TPR and FPR are computed as given in Equation 2.22 and 2.23.

$$TPR = recall = \frac{TP}{TP + FN} \tag{2.22}$$

$$FPR = \frac{FP}{FP + TN} \tag{2.23}$$

TPR is equal to the previously introduced measure of recall. FPR indicates how much a model tends to produce false alarms. Hence, there is a clear interplay between TPR and FPR. This interplay is illustrated by the ROC-curve. The AUC-ROC is computed to provide an easier interpretable measure of the curve itself.

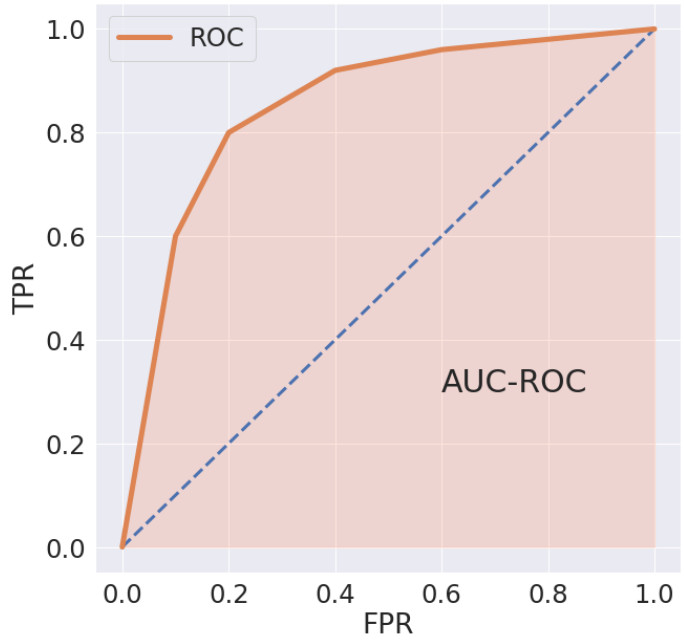


Figure 2.11: ROC-curve.

Figure 2.11 shows a typical ROC-plot of an arbitrary model. The plot will be generated by computing TPR and FPR on a range of different thresholds. These resulting points will be interpolated to get the final curve. A random prediction in the case of binary classification would result in a straight diagonal as shown in the plot. A diagonal line would result in an AUC-ROC of 0.5. A perfect classifier would achieve a TPR of 1.0 at an FPR of 0.0 at any threshold. Such a curve results in an AUC-ROC value of 1.0. An optimal threshold configuration, according to the ROC-curve, maximizes the TPR while minimizing the FPR. This means a value as large as possible on the y-axis and a value as low as possible on the x-axis.

The next chapter provides a summary of the conducted literature review. The gained information from this process served as a baseline to design appropriate experiments described in the following chapter.

3 Related Work

This chapter provides an overview of the research in the area of SDP. The first section starts with a very general introduction to different aspects and areas of SDP. The following section provides a more in-depth analysis of method-level SDP. Both sections provide a review of several influential and related research papers. The final part of this chapter is a summary of the learned lessons from the literature review.

3.1 General Software Defect Prediction

An early and important case study about SDP was conducted by Lessmann et al. [LBMP08]. The paper provided a comprehensive framework for SDP research and implemented 22 classifiers applied to multiple publicly available datasets from the NASA[3] repository. This repository contains ten different datasets of a variety of SCMs or *product metrics*. All these datasets belong to software products of different sizes. The metrics include the most traditional ones like lines of code, *Halstead attributes*, or *McCabe's Cyclomatic Complexity*, among others. All these metrics were generated on a *module-level* granularity. The sizes of datasets range from very small (125 modules) to moderate (9,537 modules). The shares of defective samples range from 0.51% to 48.8%. As a result of this case study, Lessmann et al. recommended using AUC-ROC as an evaluation metric because it separates the predictive performance from class and cost distributions, which are project-specific characteristics that may be unknown or change. They also concluded the suitability of metric-based SDP and emphasized the need for further research in the area of software metrics and other explanatory variables to extract more useful information from software products. They also stated that the best-performing models seemed to be RFs, support-vector-machines (SVMs), and NNs.

One year later, Catal and Diri provided another comprehensive overview [CD09] about SDP and how it is tackled nowadays. They conducted a systematic literature review and put some facts straight. The review was mainly focused on the type of applied software metrics, software metrics' granularity and the type of used datasets (i.e. whether they are publicly available, private, or unknown). They categorized the software metrics and prediction experiments according to their scopes or granularity-levels as follows: *method-level*, *class-level*, *component-level*, *file-level*, *process-level*, and *quantitative-level*. Quantitative-level metrics mean measurements like CPU utilization or disk usage during runtime. They considered 74 papers in their review, and none of them utilized any kind of variability-aware software metrics. Because many of the reviewed papers used private datasets, it is not easy to keep track of the number of contained software product lines at this point. It is rather unlikely that the datasets did not contain any software product line since papers like [KCH+90] [KJD02] from 1990 had driven the research and development of software product lines by a lot. This was of interest because software product lines typically make use of code-variability on a large scale.

In 2012 Hall et al. conducted another systematic literature review [HBB+12] and considered 208 defect and fault prediction studies. The number of comprehensively reviewed studies was 36 because many papers did not provide sufficient information about their applied approaches. In addition to [CD09], Hall et al. reported about the achieved performances of implemented algorithms in the reviewed papers. The review made use of precision, recall, and F1-score to report the achieved prediction capabilities. They also reported AUC-ROC as a performance metric partially. Whereas the majority of papers treated SDP as a classification problem, they also reviewed a small number of case studies

[3] Available under: http://promise.site.uottawa.ca/SERepository/datasets-page.html

treating SDP as a regression problem. Those papers predicted numbers of defects contained in a module, and hence they used error measures like MAE or MSE. Due to that, it was not possible to convert those results into comparable measures of classification approaches. Hence, in order to achieve comparable results, it is recommended to treat SDP as a classification task. Hall et al. analyzed the applied prediction algorithms, the used sort, size, and maturity of datasets and the considered software metrics as input features in the studies. They provided a very comprehensive overview of the reviewed papers and gave a good insight into the ongoing research on this topic. They categorized the metrics of the case studies in product metrics (i.e. traditional SCMs about size and complexity of code sections), *process metrics* (i.e. information about previous changes and defects in modules within the repository), *developer metrics* (i.e. socio-technical metrics to profile developers in charge) and the use of the *text of source code* itself. The source code was used to extract additional properties (e.g. the number of used &&-operators in a module). The best results were achieved by approaches using combinations of SCMs, process metrics, and source code text by Shivaji et al. [SWJAK09]. These experiments were performed on file-level granularity. Models that applied only SCMs performed worse [CBK09] [DP02] [KM08] [KS04] [MK10] [MPS08], which seemed to corroborate the superiority of process metrics. The use of socio-technical metrics about developers could partially improve the performance of models [BNG+09]. Papers that applied *feature selection algorithms* improved their results as well in most cases [SWJAK09] [KGS10] [BNG+09]. Hall et al. also concluded that at this point, relatively simple models like NB or logistic regression performed best [HBB+12]. This is a bit contrary to the findings of Lessmann et al. in [LBMP08]. Hall et al. reported that SVM-based models and NNs performed worse than expected [HBB+12]. These two techniques achieved the second and third best predictive performances in [LBMP08]. This could be caused by the fact that these more complex models are highly dependent on the selected hyperparameters. Hence, they strongly suffer from bad configurations. Provided comparisons and achieved values are not presented here because they seem to be too far-fetched and not comparable due to major differences in the tasks and taken approaches. The reviewed papers differ from this thesis in terms of their prediction-granularities, types of applied software metrics, sizes, qualities, and balances of the datasets. Anyway, Hall et al. stated that there is basically no general answer to the question about the right modeling technique [HBB+12]. It depends on the individual problem in terms of data, the developer's expertise in modeling, and the effort spent on hyperparameter optimization. They also stated that they did not find any evidence about a correlation between the granularity-level of predictions and performances. Hence, method-level defect prediction seems to be a meaningful approach. Finally, they concluded that some papers did not treat the imbalance of the dataset optimally. This might have harmed the training processes as well as the reported results. The entire literature review does only contain a single paper applying defect prediction on the Linux kernel [KB08]. This paper treated SDP as a regression problem and was aimed towards defect prediction in software versioning. So neither the results nor the use case is comparable to this thesis. Furthermore, the literature review did not mention a single paper utilizing variability-aware software metrics for defect prediction.

Most of the papers performing and analyzing SDP are focused on a coarse-grained prediction-granularity like class-level or even higher. Many papers achieved good results in SDP and found important and generally applicable information, e.g. the correlation of complexity metrics and software defects [ZPZ07]. But there is a severe need for more fine-grained SDP research [PPB18]. The study from Shihab et al. [SHAJ12] found that the research in SDP has got a little off track because developers, as potentially supported actors by SDP, consider class or module-level defect prediction as way too coarse-grained for practical

usage. In the case of such high granularity predictions the prone sections are too large to significantly reduce the amount of examination effort to ensure a module does not contain any defect [GDPG12]. Owed to the fact that large classes and modules are more likely to be bug-prone [KDPGA12], the effort to identify the particularly prone code sections inside a module is moderate to high [BBM96] [GFS05] [OWB05].

Finally, none of the reviewed papers applied variability-aware software metrics to SDP.

3.2 Method-level Software Defect Prediction

This section provides a review of literature explicitly addressing method-level SDP. This thesis aims for method-level SDP, and hence further investigation in this area could provide valuable information. These information can help in the conceptualization and evaluation of experiments as part of this thesis.

The first papers that were focused on fine-grained SDP on a method-level granularity are [MGF06] and [TBTM10]. Papers like [HMK12] [GDPG12] and finally [PPB18] deepened the research on this granularity-level. Hata et al. [HMK12] primarily focused on process metrics, whereas Giger et al. [GDPG12] used an additional set of product metrics in combination with process metrics. Giger et al. reported very good performances like achieved F1-scores of 0.86 and AUC-ROC values of 0.86 as well. They found a superiority of process metrics over product metrics and claimed that the use of only product metrics does not provide adequate results. Pascarella et al. criticized the applied validation approach in [GDPG12] and reproduced the experiments in order to validate the results. Additionally, Pascarella et al. conducted all experiments with a more realistic evaluation approach. The defect prediction experiments were applied to 22 Java projects. The results of Pascarella et al. show that the achieved performances by process metrics drop immediately as soon as their reworked and realistic validation strategy was applied. The achieved performances were very similar across product metrics, process metrics, and the combination of both. Table 3.1 shows achieved performances from Pascarella et al., where S indicates the use of SCMs, P indicates the use of process metrics, and S & P the combination of both.

Table 3.1: Median performances achieved in [PPB18].

Algorithm	F1-score			AUC-ROC		
	S	P	S & P	S	P	S & P
Baysian Network	.59	.60	.61	.53	.52	.53
J48	.62	.62	.63	.51	.51	.51
Random Forest	.64	.61	.63	.52	.51	.52
SVM	.62	.58	.62	.53	.53	.53

The results of conducted experiments by Pascarella et al. disproved any superiority of process metrics over product metrics. Even the combination of both did only achieve minimally better results. Hence, Pascarella et al. stated that method-level bug prediction is a non-solved area that requires further research [PPB18]. In terms of pre-processing, Pascarella et al. applied smote-resampling to cope with the imbalanced dataset and *correlation-based feature selection* [Hal99] to reduce multicollinearity among the data. Any comparisons across different datasets must be done carefully. Especially the F1-score strongly depends on the class balance of the test dataset. Generally, performances should be only compared among predictions using the same data. The applied SCMs in the research of Giger et al. and Pascarella et al. are shown in Table 3.2. All but three of them are also contained within the provided data of this study. The research did not report any particular importance of a metric.

Table 3.2: Static code metrics for SDP used in [PPB18].

Metric	Description
FanIN	# of methods that reference a given method
FanOUT	# of methods referenced by a given method
LocalVar	# of local variables in the body of a method
Parameters	# parameters in the declaration
CommentToCodeRatio	Ratio of comments to source code (line based)
CountPath	# of possible paths in the body of a method
Complexity	# McCabe's Cyclomatic Complexity of a method
execStmt	# of executable source code statements
maxNesting	Maximum nested depth of all control structures

Lessons learned The following list provides a summary of the most important findings of the literature review. These should be considered while designing experiments as part of this study.

a) **Prediction-granularity** The method-level defect prediction might be the most supportive and efficient prediction-granularity for software developers because of the significantly reduced effort when reviewing prone code sections [HMK12].

b) **Type of prediction task** It is recommended to treat SDP as a classification task in order to achieve easier interpretable results. Especially in the case of more fine-grained defect predictions like method-level SDP, it makes much more sense to treat the task as a classification because it is rather unlikely that functions contain multiple errors.

c) **Software metrics** Traditional SCMs seemed to perform poorly when they were applied solely. The use of variability-aware code metrics for SDP seems to be a novel approach and could provide further improvements to SCM-based method-level SDP.

d) **Prediction algorithms** There is no general answer to the question about the best classification algorithm or modeling technique. Hence, a variety of well-established algorithms should be selected and validated.

e) **Performance metrics** Most performance metrics are poor indicators of a model's performance if considered solely. A good approach is to select a set of recommended performance metrics. All of them should be recorded, reported, and analyzed in order to get a good insight into a model's prediction capability. The set of evaluation metrics should include precision, recall, F1-score, and AUC-ROC measure. Additional metrics could be considered as well. The recording of multiple performance metrics could also allow a better comparison to other experiments.

f) **Availability of the dataset** It is highly recommended to apply SDP on publicly available datasets. In the case of self-mined data, the data should be published to ensure availability. Several papers did not provide the used data, which has prevented further research or external validation of these projects.

g) **Data preparation** It is essential to comprehensively describe and explain all steps of cleaning and pre-processing. These information are important for the external validation of the results and the taken approach.

h) **Class imbalance** The highly imbalanced class distribution of the dataset is an issue that should be addressed in most cases. The appropriate method depends on the actual classification algorithm. Some algorithms suffer more from that than others.

Pascarella et al. and Giger et al. applied smote-resampling [CBHK02] according to the recommendations of [Cha09] to cope with this class imbalance. The use of smote-resampling provided promising results and should be considered within this study as well.

i) **Validation** Consider recommendations of [TMHM16] in order to perform an appropriate *k-fold Cross-Validation* (k-fold CV) using ten folds. 10-fold CV is the most widely used method of validation in the reviewed literature [GDPG12] [Gra12] [HMK12] [PD07] [JPT16] [WSS14] [SMWO10] [SWHJ14].

The next chapter gives an additional introduction to the business case of this project. It provides further information about the environment and purpose of this study and is the first stage of the applied CRISP-DM methodology.

4 Business Understanding

This chapter provides additional information to the business case of this project. The following paragraphs briefly describe a part of the ongoing research of the SSE group of the University of Hildesheim. Furthermore, the goal, purpose, and project environment is described more precisely.

The SSE research group examines several aspects of software product line engineering, among others. One particular characteristic of a software product line is the fact that it realizes code variability on a large scale. In order to implement this variability, a variety of techniques can be applied (e.g. *parametric polymorphism*, *make-file-based variability*, or *conditional compilation*). The different techniques take into account at different *binding times* (e.g. *configuration time*, *compilation time*, or *run time*) and provide different kinds of *variability-granularity* (e.g. make-file-based variability is a more coarse-grained technique than the conditional compilation of a single statement).

Ongoing research of SSE addresses how the complexity of code can be measured and how this complexity relates to code proneness. The focus of this research is on software product lines. There are various approaches to measure or approximate the complexity of a piece of code. One way of assigning code complexity is to use software metrics. There are a lot of well-established software metrics available that are used to measure specific complexity characteristics of a certain code section.

The main goal of this thesis is to examine the suitability of a variety of novel variability-aware code metrics. The assumption is that the high amount of code variability contributes to an increased code complexity, which in turn could contribute to an increased defect proneness. The complexity increases because developers need to consider and understand code, which is variable and depends on several variability variables. There might be functions that work differently under certain circumstances or variables that are only available conditionally. Hence, a developer has to implement new code while considering existing code under multiple conditions. Thus, sophisticated measures of this variability could provide useful approximations of the complexity and may serve as meaningful predictors in SDP.

As mentioned, one approach to realize code variability is conditional compilation. In the case of C-based projects, the CPP is used to analyze the code prior to the C-compiler. The CPP-directives are implemented using special CPP-code. These directives are used to include libraries or decide about what parts of source code to compile. Traditional SCMs basically ignore this kind of code and only analyze compiler code.

Hence, the measured source code does not really correspond to the conditionally compiled source code. Thus, some software metrics do not provide proper measures and only give poor approximations of the actual complexity of the source code. These metrics are not variability-aware. The provided data contain multiple metrics, which measure the contained variability within code exclusively, but it also contains metrics that provide combined measures of complexity due to variability and traditional complexity measures. The present software metrics in the provided data are a part of the metrics defined in [ESKS19].

The number of metrics within the provided data is 692. The number of totally defined metrics in [ESKS19] is about 23,000. All metrics were mined from the Linux kernel by

MetricHaven[4]. The Linux kernel serves as an ideal showcase due to its large size, the high number of developers, and the high amount of implemented code variability.

One reason why it is worth examining the correlation of code complexity and code proneness is to provide guidelines or recommendations for developers about what to avoid or obey in order to reduce the number of defects. There are already tools available that work as part of an IDE and continuously estimate the current risk of a defect in a certain code section. This shall help developers to keep their code simple and reduce the risk of defects.

The correcting of software defects is a very time-consuming and expensive activity of a software development project. Jones & Bonsignour reported it is one of the most expensive ones [JB11]. Particularly when considering that the costs for correcting a defect drastically increase with the project's progress. While the costs for a bug-fix are quite low at the prototyping stage, it becomes way more expensive at a later time in a project. This happens because the fix of a fundamental software defect could imply major changes in additional components based on that. Hence, it is worth spending the right amount of effort and attention on the emerging complexity of code in order to reduce the high costs due to finding and correcting defects.

In order to assess the suitability of the provided metrics, SDP experiments will be conducted. A good performance can be used as an indicator of the provided input features' suitability. Subsequent analyses of achieved results shall provide insights into the usefulness of certain subsets of metrics.

The next chapter gives a first introduction to the data. The first part introduces the meaning of the metrics, and the subsequent part provides a data exploratory analysis revealing some interesting characteristics.

[4]Available under: https://github.com/KernelHaven/MetricHaven

5 Data Understanding

This chapter introduces the used dataset in this study. In the first part of the chapter, the software metrics being part of the dataset are explained. The second part explains the procedure of aggregating, cleaning, and merging the data to prepare one useful dataset for all further analyses and experiments. The third part of this chapter provides the results of a data exploratory analysis, which is meant to reveal interesting characteristics of the dataset.

In the area of software engineering and especially on the domain of software product line engineering, the term *feature* typically means a configurable property of an application or a software system. A feature typically corresponds to a certain *code variability* at a *variation point*. As introduced in Section 2.2, the term feature also relates to the input features or observations that are provided to a model in order to make predictions. Hence, the meaning of *feature* differs between the domain of data science and software engineering.

In order to avoid any confusion due to ambiguous terminology, this paragraph defines the meaning of important terms distinctly. A configurable property of an application is considered as *variability*. Each variability of code is denoted as a *variation point* that considers at least one *variability variable* in the corresponding CPP-condition. A *feature* always describes a measurement belonging to a single *data point* or *sample*. The terms *data point* and *sample* are used synonymously and refer to a record of the dataset, which corresponds to a code function. If the term feature in the context of code variability could not be avoided (e.g. due to referencing names from other papers), it is distinctly denoted as a *software feature*. A collection of *samples* is always denoted as a subset or dataset.

A good understanding of the meaning of data is always an essential first step in each data science project. It helps the scientist to identify potential for faults or risks within the data but also in revealing any latent potential for further information. For instance, a timestamp or source path could provide meaningful information. In order to make features like timestamps or source paths useful, they need to be converted previously. This conversion assumes some sort of expert knowledge about the data to be aware of the interesting part of the raw feature. In the case of a timestamp, this could be an extracted weekday or year. In the case of a source path, this could be the name of a folder on a specific level of hierarchy. Hence, a good data understanding and domain knowledge are beneficial to extract the important fraction of information while discarding any useless or even confusing information. An important step in getting domain knowledge is presented by Section 5.1. Section 5.3 provides an exploratory analysis meant to reveal important properties of the dataset.

The provided data for this thesis consists of two different datasets. The first dataset is denoted as the *baseline* and encompasses all 692 software metrics generated for all functions of the entire Linux kernel V4.9.211. All functions were measured, and hence the samples are neither labeled as *healthy* nor *defective* at this point.

The entire Linux kernel encompasses around 89,000 commits, 56,000 source files, 475,000 functions and 22,000,000 lines of code. Due to the consideration of implemented conditional compilation, realized by CPP-code, the dataset contains 479,199 samples. It also contains *redundant samples*. These are samples that show the same function name but different software metrics. These redundant samples can occur since the header of a function can be nested within conditionals of CPP-code. Listing 5.1 illustrates this. As a result, the dataset would contain two samples referring `functionA` and may show different software metrics.

```
#IFDEF ASINTEGER
  int functionA(int a)
#ELIF ASFLOAT
  float functionA(float a)
#ENDIF
  {
    ...
  }
```

Listing 5.1: Nested function header in a CPP-condition.

5.1 Software Metrics

This section provides explanations about all software metrics contained in the dataset.

The software metrics count 692 different variants of nine different code metric families. These families are *Lines and Statement Counts of Code*, *Features per Function*, *Number of Code Blocks*, *Tangling Degree*, *Cyclomatic Complexity*, *Nesting Depth*, *Fan-In/Out*, *Recursive Fan-In/Out* and *Undisciplined Pre-Processor Usage*. Any metric can be clearly assigned to one of these families. Code metric families share the same method of measurement but can consider different parts of a function. For instance, the traditional lines of code (LoC) metric counts any line of C-code in a function. The lines of feature code (LoF) metric counts any line of C-code, which is surrounded by at least one CPP-condition. It counts all conditionally compiled lines of C-code. Both metrics utilize the same method of measurement but have different views on the function. Any CPP-directive is indicated by a leading `#`.

The majority of code metrics that consider CPP-code can be combined with global *variability metrics*. These additional variability metrics try to approximate the associated complexity of a certain variability variable in a specific way. Therefore, individual weights are computed for any occurring variability variable involved in a measurement. Hence, any considered variability variable is weighted individually depending on a certain variability metric. Each of these variability metrics represents a property of a variability variable on a global scope. This means in order to get a single weight, it is necessary to analyze the entire project according to a specific variability metric and with respect to a single variability variable.

The following subsections introduce the code metrics at first and the global variability metrics subsequently.

5.1.1 Code Metrics

This section provides descriptions of used code metrics based on the GitHub-repository of MetricHaven. Additional details can be found in [ESKS19]. The code metric variants are grouped by their metric families. The term *function to analyze* (FTA) denotes a certain function that is analyzed by MetricHaven.

5.1.1.1 Lines and Statement Counts of Code

LoC and Statements Counts of Code (SCoC) are likely to be part of the most common software metrics overall. These metrics measure the size and correspond to the number of lines of code of a specific code section. The dataset includes six different variants of this metric family. The variants differ in terms of the considered types of code they measure. Table 5.1 provides an overview of the individually implemented metric variants.

Table 5.1: Lines and Statement Counts of Code.

Metric Variant	Description
SCoC	Measures the number of statements of C-code within the FTA. No matter if they are surrounded or not by any CPP-condition
SCoF	Measures the number of statements of C-code within the FTA only if they are surrounded by any CPP-condition
PSCoF	$\frac{SCoC}{SCoF}$
LoC	Measures the number of lines of C-code within the FTA. No matter if they are surrounded or not by any CPP-condition
LoF	Measures the number of lines of C-code within the FTA only if they are surrounded by any CPP-condition
PLoF	$\frac{LoF}{LoC}$

5.1.1.2 Software-Features per Function

This metric family measures the number of used variability variables within a specific scope. The five different variants of this metric family consider different scopes of code and try to assess the configuration complexity of a code module. The original name of this metric is *features per function* and was modified to avoid any confusion due to ambiguous terminology. Table 5.2 provides an overview of the individually implemented metric variants.

Table 5.2: Software-Features per Function variants.

Metric Variant	Description
INTERNAL	Measures the number variability variables inside the FTA
EXTERNAL	Measures the number variability variables outside of the FTA but inside the source file
ALL	Measures the number of variability variables inside the source file of the FTA
EXTERNAL + BUILD	Measures the number of variability variables outside the FTA but inside the source file plus the used variability variables during the build process
ALL + BUILD	Measures the number of variability variables inside the source file of the FTA plus the used variability variables during the build process

5.1.1.3 Number of Code Blocks

This metric family measures the number of variation points (VPs) inside an FTA. VPs are exclusively available for CPP-code and describe any sort of construct to implement code variability. This typically means conditional compilation implemented by arbitrary `#IFDEF` statements. There are two variants implemented which differ in the method of counting. Table 5.3 provides an overview of both metric variants.

5.1.1.4 Tangling Degree

This metric family measures the number of variability variables used in CPP-code within the FTA. If a variability variable appears multiple times in VPs, each one increments the tangling degree. This is the difference between tangling degree and variability variables per

Table 5.3: Number of Code Blocks variants.

Metric Variant	Description
Block as One	Measures all connected sub-parts of a block (i.e. `#IFDEF`, `#ELIF` or `#ELSE`) as one single block
Separate Partial Blocks	Measures all connected sub-parts of a block (i.e. `#IFDEF`, `#ELIF` or `#ELSE`) individually

function. If any contained CPP-code only tests a single variability variable, the tangling degree equals to the number of *Code Blocks x Separate Partial Blocks*. Table 5.4 provides an overview of the individually implemented metric variants.

Table 5.4: Tangling Degree variants.

Metric Variant	Description
TD_{All}	Measures all used variability variables in CPP-code within the FTA, even if they appear multiple times
$TD_{Visible}$	Measures all used variability variables in CPP-code within the FTA, even if they appear multiple times, plus #ELSE statements

5.1.1.5 Cyclomatic Complexity

The Cyclomatic Complexity metric family is based on *McCabe's Cyclomatic Complexity* and measures the linearly independent control flow paths of the control flow graph within the FTA. The implemented variants consider different types of code, either only C-code, CPP-code, or a combination of both. Table 5.5 provides an overview of the individually implemented metric variants.

Table 5.5: Cyclomatic Complexity variants.

Metric Variant	Description
McCabe	Measures the number of control structures of C-Code within the FTA while ignoring any CPP-code. This is equal to the original metric defined in [McC76]
CC on VP	Measures the number of control structures of CPP-code within the FTA while ignoring any C-code
McCabe + CC on VP	Measures the number of control structures while considering C-code and CPP-code within the FTA

5.1.1.6 Nesting Depth

This metric family measures the depth of nested code elements within the FTA. There are six different variants implemented, which differ in the parts of code they consider and in the methods of counting. Table 5.6 provides an overview of the individually implemented metric variants.

5.1.1.7 Fan-In/-Out

This metric family measures different types of interaction between functions and tries to approximate the coupling of a function by analyzing its invocations. The metric family contains multiple metrics that differ in the *call directions*, *variability scopes*, and *localities*. The call direction distinguishes between *in-going* and *out-going*. The variability scope indicates what type of code is considered (i.e. only C-code or C-code and CPP-code)

Table 5.6: Nesting Depth variants.

Metric Variant	Description
$ND_{Code,max}$	Measures the maximal nesting depth of C-code within the FTA while ignoring any CPP-code
$ND_{Code,avg}$	Measures the average nesting depth of C-code within the FTA while ignoring any CPP-code. The average is computed by summing up all C-code statements nesting depth and divide it by the number of statements (SCoC) within the FTA
$ND_{VP,max}$	Measures the maximal nesting depth of C-code within the FTA while considering only CPP-code
$ND_{VP,avg}$	Measures the average nesting depth of C-code within the FTA while considering only the CPP-code. The average is computed by summing up all C-code statements nesting depth and divide it by the number of statements (SCoC) within the FTA
$ND_{Code \cup VP,max}$	Measures the maximal nesting depth of C-code while considering C-code and CPP-code
$ND_{Code \cup VP,avg}$	Measures the average nesting depth of C-code while considering C-code and CPP-code. The average is computed by summing up all C-code statements nesting depth and divide it by the number of statements (i.e. SCoC) within the FTA

under certain circumstances. The locality determines the extent of code to analyze in order to measure the metric (i.e. only the source file or the entire software project). Table 5.7 provides an overview of the individually implemented metric variants.

Table 5.7: Fan-In/-Out variants.

Metric Variant	Description
Classical Fan-In locally	Measures how often the FTA is invoked from inside of its source file while ignoring any CPP-conditions
Classical Fan-In globally	Measures how often the FTA is invoked from anywhere in the code while ignoring any CPP-conditions
Classical Fan-Out locally	Measures how many functions of the same source file the FTA invokes while ignoring any CPP–conditions
Classical Fan-Out globally	Measures how many other functions of the entire code the FTA invokes while ignoring any CPP-conditions
Conditional Fan-In locally	Measures how often the FTA is invoked from inside its source file while only considering function calls nested in CPP-conditions
Conditional Fan-In globally	Measures how often the FTA is invoked from anywhere in the code while only considering function calls nested in CPP-conditions
Conditional Fan-Out locally	Measures how many functions of the same source file the FTA invokes while only considering function calls nested in CPP-conditions
Conditional Fan-Out globally	Measures how many functions of the entire code the FTA invokes while only considering function calls nested in CPP-conditions
Degree Centrality Fan-In locally	Measures the number of variability variables (+ 1) of surrounding CPP-conditions when the FTA is invoked from another function inside the same source file

Degree Centrality Fan-In globally	Measures the number of variability variables (+ 1) of surrounding CPP-conditions when the FTA is invoked from another function anywhere in the code
Degree Centrality Fan-Out locally	Measures the number of variability variables (+ 1) of surrounding CPP-conditions for invocations of functions from inside the same source file
Degree Centrality Fan-Out globally	Measures the number of variability variables (+ 1) of surrounding CPP-conditions for invocations of functions from anywhere in the code
Degree Centrality Fan-Out No Stubs	Corresponds to previously described degree centrality variants while ignoring invocations of empty function stubs (i.e. functions with empty body)
Degree Centrality Fan-Out Ignore External VPs	Corresponds to previously described degree centrality variants while ignoring invocations surrounded by external `#IFDEFs`

5.1.1.8 Recursive Fan-In/-Out

This metric family provides recursive variants of the Fan-In/-Out metrics shown in Table 5.7. While the non-recursive variants only count all invocations, the recursive variants introduce additional weights for each invocation. This means the simple variants only measure the invocations of or by the FTA on the very top-level. The recursive variants analyze any invocation of or by the FTA to the full depth. Hence, an invocation of a function, which also makes multiple invocations, adds a higher value to *Recursive Fan-Out* than an invocation of a function, which does not invoke any functions itself. The assumption is that modifications in functions have higher risks of a defect when these functions are used in multiple parts of the project. Modifications of functions that are only used in a single part of code could be less prone to defects since it is less likely that they cause negative side effects due to unconsidered dependencies.

5.1.1.9 Undisciplined Pre-Processor Usage

This metric family measures undisciplined usage of the Pre-Processor (UPU) according to [MRG+17]. This means C-code statements that are interrupted by CPP-code. Listing 5.2 shows an example of UPU where a C-code control structure is interrupted by a CPP-condition. A second example is shown in Listing 5.3. In this case, a `return`-statement is interrupted by a CPP-condition.

Table 5.8: Undisciplined Pre-Processor Usage variants.

Metric Variant	Description
Undisciplined Pre-Processor Usage	Measures any interrupted C-code control-structure (i.e. as shown in Listing 5.2) and any interrupted C-code statement (i.e. as shown in Listing 5.3)

```
#IFDEF CONDITION
  if (var > 0) {
#ENDIF
  action1();
#IFDEF CONDITION
  } else {
#ENDIF
  action2();
}
```

Listing 5.2: Interrupted C-code control structure by a CPP-condition.

```
...
return
#IFDEF NEGATION
  -1 *
#ENDIF
result;
```

Listing 5.3: Interrupted C-code statement by a CPP-condition.

5.1.2 Variability Metrics

This section provides an overview of the additionally applied variability metrics. These variability metrics serve as weights, which are used for a subset of previously introduced code metrics. These variability metrics are computed by an analysis of the entire code or variability models. A variability metric is always related to a certain variability variable.

5.1.2.1 Scattering Degree

This metric measures the scattering of a certain variability variable. The assumption is that those variability variables that control very scattered parts of code could cause higher risks of defects because a developer needs to keep in mind very scattered parts of a project. It is easier to implement code using variability variables that exclusively control code sections in the current source file. It is more complex to implement code using variability variables that control multiple parts of code, scattered over the entire project. Table 5.9 provides an overview of the individually implemented metric variants.

Table 5.9: Scattering Degree variants.

Metric Variant	Description
SD_{VP}	Measures the number of VPs that have at least one reference to the current variability variable
SD_{File}	Measures the number of different source files that have at least one VP using the current variability variable

5.1.2.2 Software-Feature Size

This metric is computed in a similar way as the scattering degree. Whereas the scattering degree measures the number of VPs or files using a certain variability variable, the software-feature size measures the number of statements (i.e. ScoC) that are controlled by a certain variability variable. The original name of this metric is *feature size* and was modified to avoid any confusion due to ambiguous terminology. Table 5.10 provides an overview of the individually implemented metric variants.

5.1.2.3 Software-Feature Definition Distance

This metric measures the distance on file-level between a variability variable used in the FTA and the variability model, which has defined it. The assumption is similar to the scattering degree's. The risk of a defect may increase when dependencies become further apart. In this metric, the distance between a variability variables definition and its usage is measured. This metric tries to approximate the coherence of a certain variability variable. Low values indicate high coherence. The original name of this metric is *feature definition*

Table 5.10: Software-Feature Size variants.

Metric Variant	Description
Positive Size	Measures the number of statements (i.e. SCoC) controlled by a certain variability variable while ignoring any negated appearances of the variability variable
Total Size	Measures the number of statements (i.e. SCoC) controlled by a certain variability variable while considering any negated appearances of the variability variable

distance and was modified to avoid any confusion due to ambiguous terminology. Table 5.11 provides an overview of the individually implemented metric variants.

Table 5.11: Software-Feature Definition Distance variants.

Metric Variant	Description
Software-Feature Definition Distance	Measures the length of the shortest path from the location of the variability model defining a certain variability variable to the location of the FTA. If the variability model is in the same folder as the FTA a value of zero is returned

5.1.2.4 Cross-Tree Constraint Usage

This metric measures the number of associated constraints of a certain variability variable. Therefore, the defining variability model is analyzed. This metric also tries to approximate the coherence of a certain variability variable, while low values indicate high coherence. Table 5.12 provides an overview of the individually implemented metric variants.

Table 5.12: Cross-Tree Constraint Usage variants.

Metric Variant	Description
Incoming Connections	Measures the number of variability variables defining a constraint of a certain variability variable
Outgoing Connections	Measures the number of references of a certain variability variable in any constraints
All Connections	Incoming Connections + Outgoing Connections

5.1.2.5 Connectivity of Software-Feature

This metric measures associations of a certain variability variable to others. These associations are based on different properties that are extracted from the defining tree-based variability model of a variability variable. The original name of this metric is *connectivity of feature* and was modified to avoid any confusion due to ambiguous terminology. Table 5.13 provides an overview of the individually implemented metric variants.

Table 5.13: Connectivity of Software-Feature variants.

Metric Variant	Description
Children	Measures the number of direct children (i.e. variability variables underneath) of a certain variability variable
CoC	Measures the number of associated variability variables while considering parents (i.e. variability variables above), children and other variability variables associated by any constraint of a certain variability variable

5.1.2.6 Software-Feature Types

This metric measures variability variables according to their data type. Therefore, a set of pre-defined weights for each data type must be defined. The original name of this metric is *feature types* and was modified to avoid any confusion due to ambiguous terminology. There are no different variants defined for this metric.

5.1.2.7 Software-Feature Hierarchies

This metric measures the hierarchical level of definition within the variability model that defines a certain variability variable. A pre-defined set of weights is applied to consider the individual types differently. The different types of levels are separated into *top-level* (i.e. no parent variability variable), *intermediate-level* (i.e. parent and children variability variable), and *leaf-level* (i.e. no children variability variable). The original name of this metric is *feature hierarchies* and was modified to avoid any confusion due to ambiguous terminology. Table 5.14 provides an overview of the individually implemented metric variants.

Table 5.14: Software-Feature Hierarchies variants.

Metric Variant	Description
Hierarchy Types	Measures a certain type of hierarchy-level of a certain variability variable within its defining variability model
Nesting Level	Measures the nesting depth of a certain variability variable within its defining variability model

5.2 Data Aggregation

This section describes the procedure of building a meaningful and cleaned dataset to be used in the exploratory analysis of Section 5.3. The provided data consist of two different datasets. A part of the data is spread into multiple small files that must be aggregated.

5.2.1 Cleaning of the Defects

The initially provided data consists of two different datasets. This section describes the process of cleaning the dataset of *defects*. These data are provided as 55 individual CSV-files. All these files contain subsets of samples corresponding to the occurred compilation-problems of a certain month. The months range from March 2013 to December 2017, which is about five years. The raw dataset is available under https://github.com/SSE-LinuxAnalysis/MetricResults. Each of these samples is described by all selected software metrics and some additional information indicating the type of occurred problem, a timestamp and the location of the function.

Step 1 After reading all CSV-files containing defects, a dataset of 13,661 samples and 699 features was built. In the first step of cleaning, all samples that were not of type `ERROR` have been removed. This included the removal of 2,700 samples of type `WARNING`, 1,453 samples of type `SPARSE`, 944 samples of type `NOTE`, and 308 samples of type `UNKNOWN`. The resulting dataset contained 8,256 samples of type `ERROR` yet.

Step 2 The second step of cleaning has removed all unnecessary information from the dataset. This included the removal of `Type, Line No., Commit, Repository, Date`, and the addition of a feature, which indicates the class label `y`. All samples in this dataset got label one, which indicates a defect. Afterward, the dataset contained 694 features and the class label `y`. The features include 692 software metrics, `Source File`, and `Element`. The `Source File` denotes the location, and the `Element` denotes the name of the function.

Step 3 The third step of cleaning has removed all duplicated samples based on the remaining features. It resulted in a dataset containing 6,231 samples. These are the final samples of defects used for any further analysis and predictions.

5.2.2 Cleaning of the Baseline

In contrary to the defects, the baseline was provided as a single CSV-file containing all data. Those data represented all functions of the entire Linux kernel at a certain point in time.

Step 1 The provided CSV-file contained 432,518 samples and 695 features. The features contained 692 software metrics, `Source File, Line No.`, and `Element`. The first step of cleaning removed all unnecessary features. This only included the removal of `Line No`. Furthermore, the label `y` was added. At this temporary step, all contained samples got value zero as their label.

Step 2 The second step of cleaning has removed all duplicated samples based on the remaining features. The final dataset contained 432,235 samples.

5.2.3 Merging both Datasets

Both previously cleaned datasets were merged into one large dataset containing all relevant samples. In order to build a meaningful overall dataset, some steps of preparation were necessary. This procedure is described in the following paragraphs.

Figure 5.1 shows an abstract Venn-diagram of the two datasets. The baseline was much larger than the defects data. At this time, the baseline contained not only the healthy samples but also a part of defects, and all were temporarily labeled as healthy. In the first step of merging, all samples that appeared in the baseline and in the defects were removed from the baseline. This intersection included 3,693 samples. These samples could be distinctly identified using `Source File` and `Element`.

The defects are not a proper subset of the baseline because the data contain defects from more than only one point in time. There are defects from about five years. Hence, the defects contain functions that caused problems in the past but were totally removed from the project in the meantime. Hence, these samples were not part of the baseline anymore, but they are still informative for the purpose of SDP. Thus, these samples were kept in the dataset. The number of defects that were not part of the baseline is 2,538.

After the intersection was removed, both datasets were finally merged. The resulting dataset contains all relevant samples and corresponds to the set described by Equation 5.1. It finally contains 434,773 samples in total.

$$\text{Final Dataset} = (\text{Baseline} \setminus (\text{Baseline} \cap \text{Defects})) \cup \text{Defects} \tag{5.1}$$

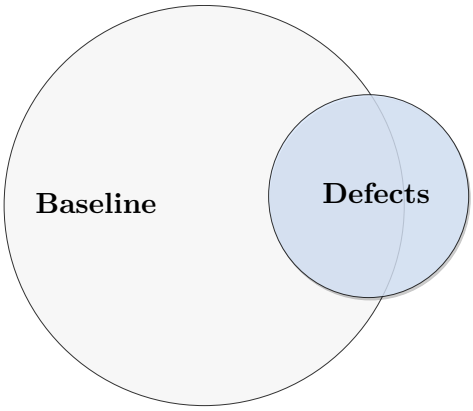


Figure 5.1: Venn-diagram of the provided data.

5.3 Data Exploratory Analysis

This section provides and discusses the results of the performed exploratory analysis of the data. This process was meant to get a first idea and discover important properties of the dataset. Those information helped in constructing adequate model configurations.

5.3.1 Class Balance

A common first step of data exploration is to examine the class balance. A class imbalance is a typical problem in SDP, but there are many possible approaches to address such an issue. Figure 5.2 illustrates the class distribution of this dataset. There are 428,542 samples labeled as *healthy* and 6,231 samples labeled as *defective*. This is a severe imbalance of roughly 1:69. The problem of imbalance will be discussed in detail in Chapter 6.

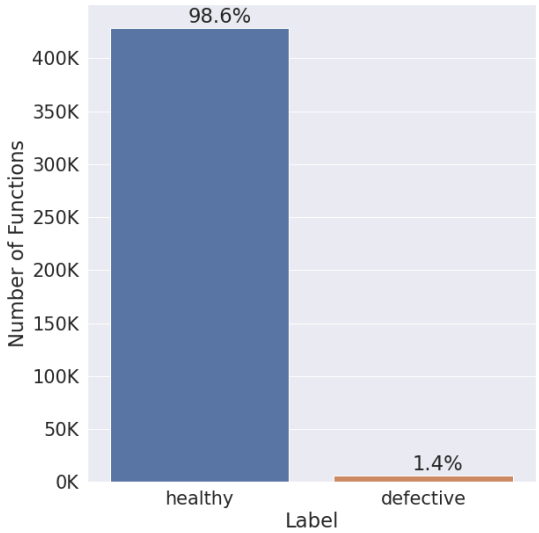


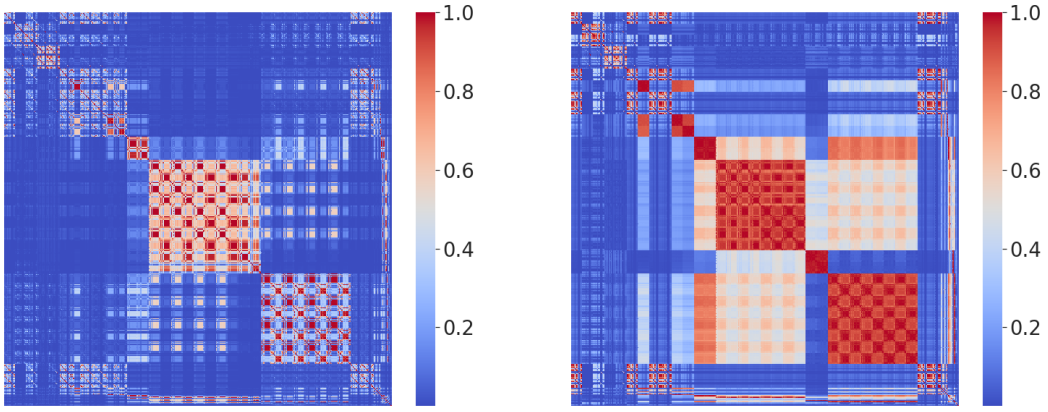
Figure 5.2: Class distribution of the dataset

5.3.2 Correlation & Data Distribution

Another basic step of data exploration is to examine the presence of correlation among the input features. A high amount of correlation among features can complicate the interpretation of feature importances because it can lead to shared (and hence reduced) importance values extracted by algorithms [GMSP17]. Furthermore, perfectly correlated features do

not contribute to a model's performance because the information is redundantly available. Thus, those correlated features can be removed from the data because one feature is only a linear projection of another. This only applies for perfectly correlated features according to the *Pearson correlation*, which measures the collinearity of features. Correlation is present if two features move in the same or opposing directions by a consistent (linear) amount. Two features moving in the same direction indicate a positive correlation, and two features moving in opposing directions indicate a negative correlation. A second correlation measure is the *Spearman correlation*. While the Pearson correlation is meant to be used for continuous features, the Spearman correlation is also applicable to ordinally scaled discrete values. It measures the monotonic relationship between variables. Hence, the Spearman correlation is less restrictive and is already satisfied if two variables move in the same or opposing direction monotonically regardless of a consistent amount of increase or decrease. In contrary to the Pearson correlation, a perfectly correlated feature, according to Spearman, could still provide additional information to its counterpart feature.

Figure 5.3 shows two heatmaps representing the absolute correlations among all features. Figure 5.3a shows a heatmap using Pearson correlation and 5.3b shows a heatmap using Spearman correlation.



(a) Heatmap of Pearson correlation.

(b) Heatmap of Spearman correlation.

Figure 5.3: Heatmaps of the correlations among the software metrics.

The absolute correlation coefficients are shown in the figure because it is less important whether a correlation is positive or negative, but its magnitude matters. Hence, providing the absolute values simplifies the diagram without omitting any important information. Both heatmaps show a significant amount of correlation among the features. The labels were omitted due to the high number of features. As supposed, the Spearman correlation tends to show higher and even more correlations among features than the Pearson correlation. This is owed to the less restrictive correlation criterion. In general, both correlations show similar results. The rectangular box pattern within both heatmaps along the diagonal correspond to features of the same metric family. This makes sense because the software metrics within one family only differ slightly. They generally share the same method of measurement. In order to identify redundant information among features, the Pearson correlation is more meaningful. In fact, there are 7,830 highly correlated unique feature combinations with a correlation coefficient $r > 0.9$. The total number of unique

Table 5.15: Descriptive statistics of the most related metrics to the target variable.

	Mean		Median		Mode		Max	
Class	H	**D**	H	**D**	H	D	**H**	D
LoC	26.12	**52.02**	16	**29**	4	4	**5127**	4431
SCoC	14.36	**27.80**	8	**16**	1	1	**2764**	2146
McCabe + CC on VPs x Feature distance	3.72	**6.56**	2	**4**	1	1	**301**	191

feature combinations is 239,432 ($\frac{\text{Number of features}^2}{2}$). Furthermore, there are 47 perfectly correlated unique feature combinations ($r = 1.0$). These combinations are shown in Table A.1 in the Appendix.

Besides the correlation among input features, it is a common step in an exploratory analysis to examine the correlation between features and the target variable. This is quite a meaningful and straightforward analysis in the case of regression problems (i.e. numerical target variable), but it becomes complicated in the case of classification tasks. Such correlation analyses try to approximate the importance and suitability of available features as predictors in a univariate fashion.

In the case of binary classification, the outcome is limited to two labels: *healthy* and *defective*. These labels can be represented as *zero* and *one*. It is also possible to interpret a numerical meaning to these numbers, the number of defects. This fact allows computing correlations in a partially meaningful manner. Considering that the Pearson correlation tests the presence of collinearity between two variables, an absolute correlation coefficient of one is certainly not possible when testing input features and the target variable. Nevertheless, it is a common practice to compute the Pearson correlation for continuous input features and target variables (presumed the labels are at least ordinally scaled). Even though the resulting correlation coefficients are not clearly interpretable, the ranking of correlation coefficients is. Hence, the correlation does not provide precise information about the amount of association between a feature and the label. Still, it points to features that are worth to be examined in more depth. Table A.2 in the Appendix provides an overview of the 50 highest achieved univariate correlations between the input features and the target variable. `LoC`, `SCoC`, and `McCabe + CC on VPs x Feature distance` show the strongest relation to the target variable. Thus, Figure 5.4 shows the data distributions of these metrics. The distributions are illustrated by kernel density estimates and a prior removal of outliers according to the *robust Z-score method* [IH93] using medians instead of means. The use of medians is generally more robust in the presence of outliers, and hence it has provided the best suitable method of outlier detection.

The kernel density estimations show very similar distributions once visualized. Especially Figure 5.4a and 5.4b look very similar. This is owed to the fact that `LoC` and `SCoC` metrics are highly correlated ($r \approx 0.905$). Both metrics show higher densities at lower values on healthy samples than on defect samples. This suggests `LoC` and `SCoC` tend to have higher values on defects. Table 5.15 shows some additional descriptive statistics, whereas the higher values of both classes are highlighted in bold.

The means and medians of `LoC` and `SCoC` are higher on defects, which would strengthen the mentioned assumption. The mode of both metrics is equal for both classes. This means that both metrics have the same most occurring values in both classes. This fact relatives the significance of the initially mentioned assumption about higher `LoC` and `SCoC` values on defects. Furthermore, the table shows that the healthy samples have a higher maximum value than the defects, which finally comprises the assumption.

Figure 5.4c shows the density estimate of `McCabe + CC on VPs x Feature distance`. The distribution of the healthy samples looks like a one-sided normal distribution (with gaps due to only integers and a small range of values), whereas the defective samples show an almost uniform distribution with only a very slight decrease to the right side. Even though both class' density estimates look quite differently, this metric also shows the same properties as the previous two. Higher mean and median on defects but same mode values and a higher maximum value on healthy samples. Finally, neither the plots nor the descriptive statistics could exhibit some clear and reliable differences. Both classes show differences, but it is questionable that they have any causality or statistical relevance. Hypothesis tests were not considered as meaningful at this point due to the imbalance of classes, the presence of outliers, and noisy distributions.

The univariate examination of relations between input features and the target variable did not yield any clear information. The question about more or less important features must be addressed by methods of machine learning.

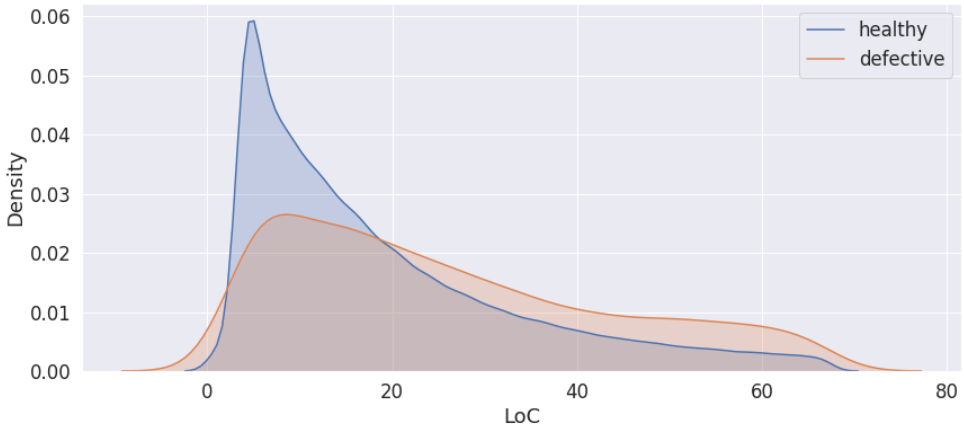
5.3.3 Locations

The last part of the data exploration examines the locations of functions. Location means the place in the project where the source file of a function is stored. This location can be extracted from the `Source File` column of each record. The aim of this analysis is to examine whether there are locations having higher defective rates than others.

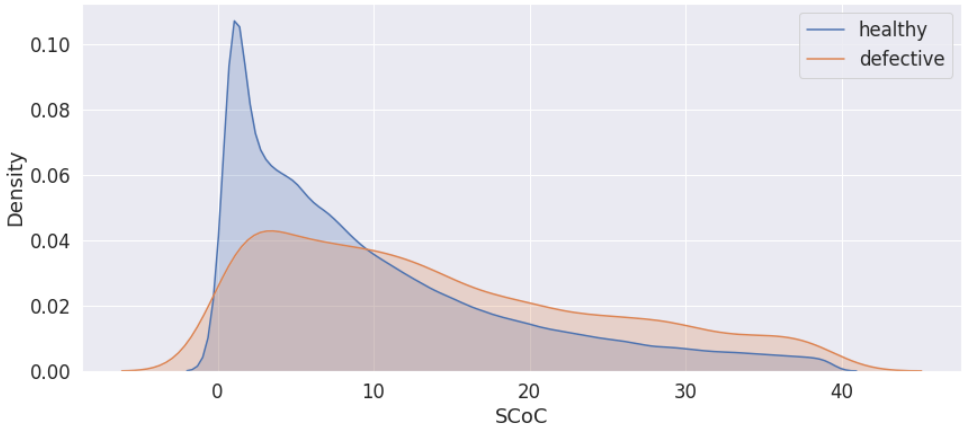
On the top level of the hierarchy, the Linux kernel is structured into 17 different locations (*drivers*: 64.34 %, *arch*: 13.12 %, *fs*: 6.26 %, *net*: 6.19 %, *sound*: 5.07 %, *kernel*: 2.19 %, remaining: < 1 %). Only the *kernel* location shows an increased defective rate of 4.10 % (defective rate overall: 1.43 %, *drivers*: 1.27 %, *arch*: 1.71 %, *fs*: 1.28 %, *net*: 1.25 %, *sound*: 1.03 %, *kernel* 4.10 %, remaining: statistically irrelevant due to insufficient samples).

If locations are extracted on a lower level, there are multiple locations with increased defective rates. Analysis on a lower level means that the location is extracted from the `Source File` path till the second level of the hierarchy. Statistically irrelevant locations due to insufficient samples ($< 1{,}000$) have been merged and were denoted as *Others*. Figure 5.5 shows the defective rates of the remaining 92 locations. The dashed line shows the overall defective rate. The figure suggests that there are many locations being significantly more or less prone to defects than others. This information could contribute to prediction performance in later experiments. A detailed overview of the defective rates within the locations is shown in Table A.3 in the Appendix.

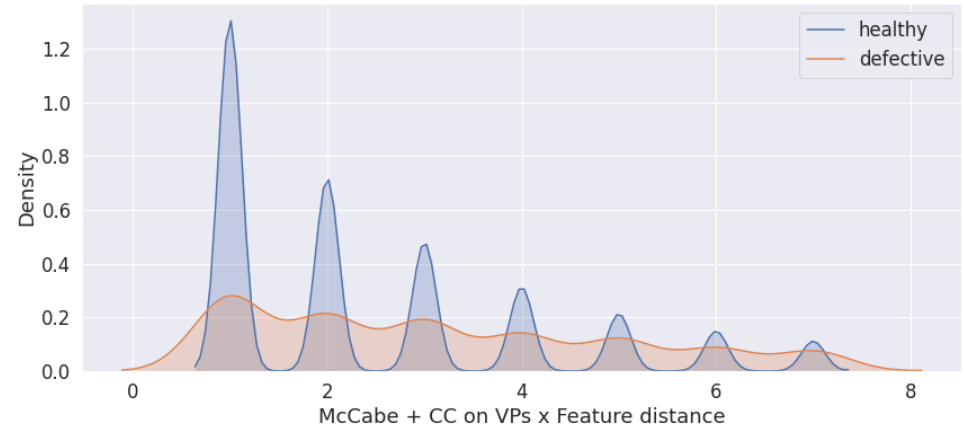
The following chapter describes the first set of experiments contributing to answering [RQ1], [RQ2], and [RQ3] partially. The experiments test the prediction capabilities of several feature subsets to assess the suitability of individual types of software metrics. The precisely defined goal, the concept, the models' construction, and implementation details as well as the final analysis of results are part of the next chapter.



(a) Kernel density estimation of LoC.



(b) Kernel density estimation of SCoC.



(c) Kernel density estimation of McCabe + CC on VPs x Feature distance.

Figure 5.4: Kernel density estimates of the strongest related input features to the target variable.

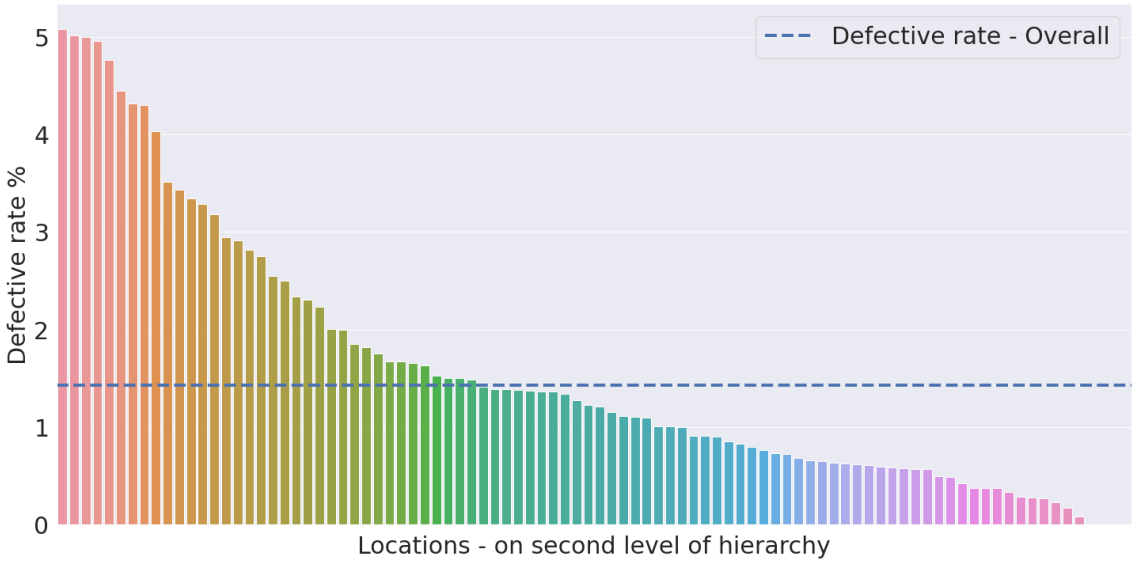


Figure 5.5: Defective rates within locations.

6 Software Metric Subset Comparison Experiments

This chapter provides a detailed description of the first set of experiments. The repetitive phases of *Data Preparation* and *Modeling* were refined to provide a more suitable guideline to describe and document this procedure step-by-step. The phases of Data Preparation and Modeling were subdivided into six sub-phases. In order to examine the given research questions, two sets of experiments had to be performed. These sets of experiments aimed for certain sub-goals contributing to answering the superior research questions. Each set of experiments was designed, performed, and evaluated according to the sub-phases shown in Figure 6.1.

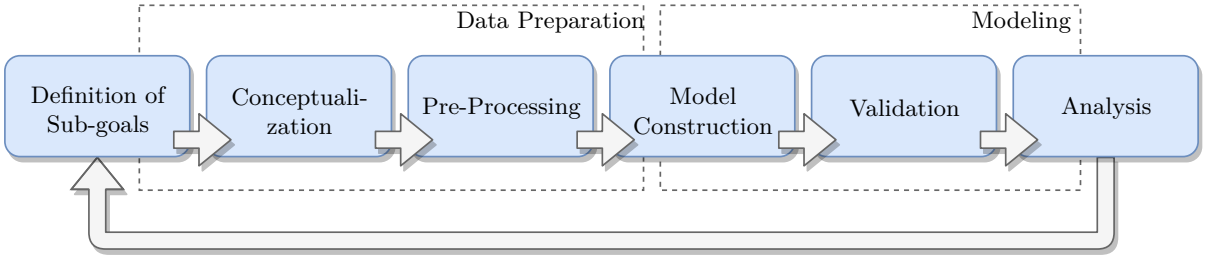


Figure 6.1: Refined procedure of Data Preparation and Modeling.

The first phase is the *definition of sub-goals*. The subsequent phase of *conceptualization* defines and elaborates the approach of how to achieve these goals. The third phase of *pre-processing* includes an examination of present data issues or obstacles and the proposal of the most suitable solutions on how they can be overcome. Once the required steps of pre-processing are addressed, the *model construction* phase can start. This phase includes the typical steps of building an architecture and optimizing hyperparameters. As soon as the models are constructed, the *validation* phase begins. This includes the execution of the defined experiments, which means training and testing plenty of models under a pre-defined set of configurations. The final phase of *analysis* presents and reviews the results but does not include final answers to research questions or derivations of conclusions yet. The subsequent Chapters 8 and 9 put the results in the context of the research questions and derive conclusions from that.

This chapter is structured analogously to the introduced stages of the previous paragraph, whereas Section 6.5 provides additional information and details on how the pre-processing, the models, and the validation routine were implemented.

6.1 Goal

The first set of experiments aimed to evaluate the prediction capabilities of different software metric subsets. The definition and description of these metric subsets used as input features will be given in Section 6.2. The metrics were grouped according to their *variability-awareness* and *scope*. All feature subsets were validated using the same set of pre-defined hyperparameters in order to improve the comparability. The entire procedure of validating all feature subsets is denoted as the *validation routine*. The obtained results from this validation routine served as the foundation for addressing [RQ1] and [RQ2]. Research question [RQ3] was partially addressed by this set of experiments because the software metrics were not assessed individually but in a coarse-grained manner using metric groups.

The hyperparameters are divided into *general hyperparameters* and *algorithm-specific hyperparameters*. The general hyperparameters can be applied to any sort of algorithm and basically address steps of pre-processing. Algorithm-specific hyperparameters are only applicable to one sort of algorithm and basically contain a classifier's architecture, applied kernel, or other settings. A complete selection of hyperparameters is denoted as *configuration*. Subsequently to the model construction stage, a single shared and meaningful algorithm-specific configuration was defined for each applied classification algorithm. This configuration was shared over all feature subsets. The definition of a meaningful but shared configuration might not lead to the best possible results but to an applicable and comparable compromise. The use of uniform algorithm-specific configurations was also owed to limited computational capacities.

6.2 Concept

This section provides a detailed description of the followed approach to achieve the desired results. This includes explanations and justifications of all decisions. The following paragraphs will describe the approach in detail and focus on certain parts of the concept.

Feature Subsets Research questions [RQ1] and [RQ2] require information about any superiority of one type of software metric over another. Hence, a variety of feature subsets was defined according to the software metrics' types. These types depend on the metrics' *variability-awareness* and *scope*. Section 5.1 has already introduced code metrics that consider variability as well as code metrics that ignore any variability. Any code variability is realized by the utilization of CPP-code and variability models in this case. Code metrics that consider variability are denoted as *variability-aware*. Code metrics that do not consider variability are denoted as *non-variability-aware*. The variability-aware metrics were further subdivided into *pure-variability-aware* metrics and *combined-variability-aware* metrics. The pure-variability-aware metrics focus on describing only the variability and ignore any C-code, whereas combined-variability-aware metrics consider CPP-code and C-code for measurements. Finally, all introduced code metrics of Section 5.1.1 can be clearly assigned to one of these three feature subsets according to its variability-awareness.

Besides the variability-awareness of metrics, the different scopes must be examined as well. Therefore, two different scopes were defined. At first, the *local scope*, which means that a code metric does not utilize any variability metrics, which were introduced in Section 5.1.2. These variability metrics weight any considered variability variable individually. Once a code metric utilizes those variability metrics, the entire software project must be analyzed. Therefore, the use of an additional weighting by variability metrics is considered as the *global scope*. Obviously, only variability-aware metrics can utilize variability metrics.

Consequently, all the provided software metrics, including those that utilize additional variability metrics, can be assigned to the following five metric subsets:

- Local-scoped non-variability-aware metrics (LNV)
- Local-scoped pure-variability-aware metrics (LPV)
- Local-scoped combined-variability-aware metrics (LCV)
- Global-scoped pure-variability-aware metrics (GPV)
- Global-scoped combined-variability-aware metrics (GCV)

All contained software metrics in the feature subsets are shown in Tables A.4, A.5, A.6, A.7 and A.8. A special case of the global-scoped metrics must be considered. The global-scoped feature subsets are not disjointly separable from the local-scoped ones. In case there is no global-scoped version of a code metric available (i.e. the code metric cannot be weighted by additional variability metrics), the local-scoped version of this metric is included in the corresponding global-scoped subset anyway. The reason is that [RQ2] aims to test for a potential improvement due to the extension of the scope (i.e. from local-scoped to global-scoped). Hence, discarding one sort of code metric entirely due to the lack of a global-scoped version would omit one specific perspective (i.e. realized by a certain code metric) on the functions. Thus, it is possible that a model performs worse due to omitting this perspective. This case of missing an important metric entirely could result in a decreased performance. If the performance is negatively affected for that reason, it is not possible anymore to clearly recognize an improvement or deterioration caused by the extended scope. The extend of the scope could have increased the model's performance, while the absence of a certain code metric could have decreased it. Rather than discarding the local-scoped metric (that has no corresponding global-scoped version), it is included as well. This allows to still assess changes in performance with regards to an extended scope. The model does not lack any perspectives on functions, and hence any in- or decrease in performance can be ascribed to the extended scope.

Classification Algorithms One of the most general hyperparameters to determine was the selection of meaningful algorithms used to perform defect prediction. The reviewed literature, which was presented in Section 3.2 and the performed data exploratory analysis of Section 5.3, served as the foundation for this decision. The reviewed literature reported different types of algorithms as the best choices for classification in SDP. Nevertheless, no consensus could be recognized. Hence, a variety of different classification algorithms was considered. This allowed a direct comparison of each classifier and the finding of the most suitable solutions for this specific problem. For that purpose, the following collection of algorithms was selected and considered as promising due to recommendations of the reviewed literature, personal experience, and additional research on state-of-the-art classification approaches on tabular data.

- Fully Connected Network (FCN)
- Random Forest Classifier (RF)
- Gradient Boosting Classifier (GB)
- Naive Bayes Classifier (NB)

Hyperparameter Optimization Before any other data processing, an *outer validation dataset* of 10% of the samples was separated. This outer validation dataset was used for any sort of model construction and hyperparameter optimization purposes. The usage and separation of the initial dataset are illustrated in Figure 6.2. The *Holdout-strategy* was chosen for the sake of computational cost. A more extensive approach using a cross-validation-based grid- or random-search was considered as infeasible. The high number of algorithms and feature subsets as well as the large size of the dataset combined with multiple general and algorithm-specific hyperparameters to consider would exceed the available computational resources. The initial model construction and hyperparameter optimization led to a single architecture and model-specific set of hyperparameters each algorithm.

Besides the algorithm-specific hyperparameters, a set of meaningful general hyperparameters was selected. All these general hyperparameters were validated extensively by the final validation routine. Section 6.3 provides detailed explanations about any considered general hyperparameters in terms of pre-processing steps. The individual subsections of Section 6.4 provide details about all selected algorithm-specific hyperparameters.

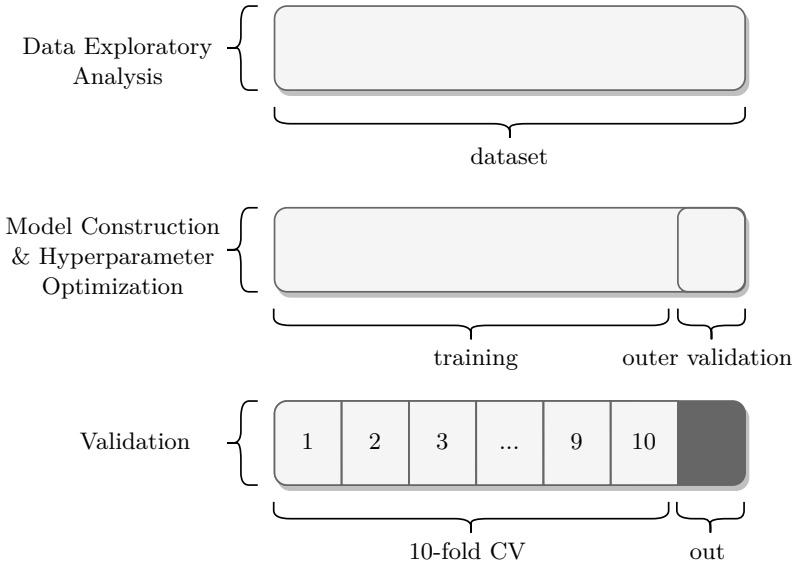


Figure 6.2: Usage of the dataset.

Performance Measures Before any model could be constructed or optimized, a reasonable measure of performance had to be defined. This metric was used for any sort of model validation. The foundation of the decision about a suitable set of performance metrics was the literature review presented in Section 3.2 as well as some additional research. A variety of performance metrics was selected in order to allow the best possible comparability and insight into a model's prediction capability.

The literature has reported the importance of providing results in a comprehensive manner. Hence, all recorded performance metrics will be presented for any model validation. This is a record of AUC-ROC, F1-score, recall, and precision. Even though all of these metrics are reported continuously, the F1-score served as the primary performance indicator for the model construction and optimization. All other metrics served as essential parts to better understand a model's prediction capability.

Validation Strategy The *k-fold CV* strategy was selected to provide robust and less biased results. The goal of this set of experiments was to obtain information about the general suitability of pre-defined feature subsets (i.e. software metric subsets). Hence, a Holdout-strategy on a randomly selected test set would not be sufficiently reliable because the performance strongly depends on the selected test samples. This bias can either produce optimistic or pessimistic results. The CV-approach provides multiple results of different test sets. These results can be combined to one robust measure, which is less biased to any of the specific test sets. The number of folds k was set to ten, and the number of CV repetitions was set to one. Due to increasing computationally costs, the number of repetitions could not be increased. A higher number of repetitions could still improve the robustness of the final validation score of a configuration. The CV approach was also

performed in multiple previous papers that examined SDP [GDPG12] [Gra12] [HMK12] [PD07] [JPT16] [WSS14] [SMWO10] [SWHJ14].

A stratified CV approach was selected to retain the original class balance and achieve a realistic final validation score. Because most performance metrics are strongly affected by the class distribution, the subsequent combining of obtained metrics under different distributions would entirely mess up the scores. The evaluation of each inner 10-fold iteration was based on a single changing fold for testing purposes (*Leave-One-Out* strategy). The outer validation dataset was removed prior to this validation. This also contributes to the avoidance of optimistic results because the models were built and specifically optimized with respect to the performance on this outer validation dataset.

A set of defined seeds ensured the same pseudo-random fold splits across all configurations to validate. This did not affect the randomness of a particular inner 10-fold iteration at all. It just allowed providing the same circumstances to all configurations. Each configuration to validate used the same ten different fold splits. Without this, a comparison across any configurations of models would mean comparing different models, which were trained and tested on different data. Hence, such a comparison would not provide meaningful results. Section 6.5.3 describes the implementation of the applied validation strategy in detail.

Figure 6.3 illustrates the composition of training and testing data according to a k-fold CV approach. The value of k determines the number of created folds, which in turn determines the number of needed iterations to include all possible training and testing compositions. The gray diagonal shows how the currently applied testing fold in the inner iterations changes step-wise. A stratified CV approach ensures each fold of each iteration has the same class distribution as the unified dataset. Hence, regardless of the k-fold iteration, any composition has also the same class distribution as the unified dataset. Because the training and testing composition changes in each iteration and steps of pre-processing are typically based on only training data, a CV approach also implies a repetitive pre-processing of data at each single inner k-fold iteration. Any processing prior to k-fold splitting would violate the indispensable separation of training and testing data and compromise any results.

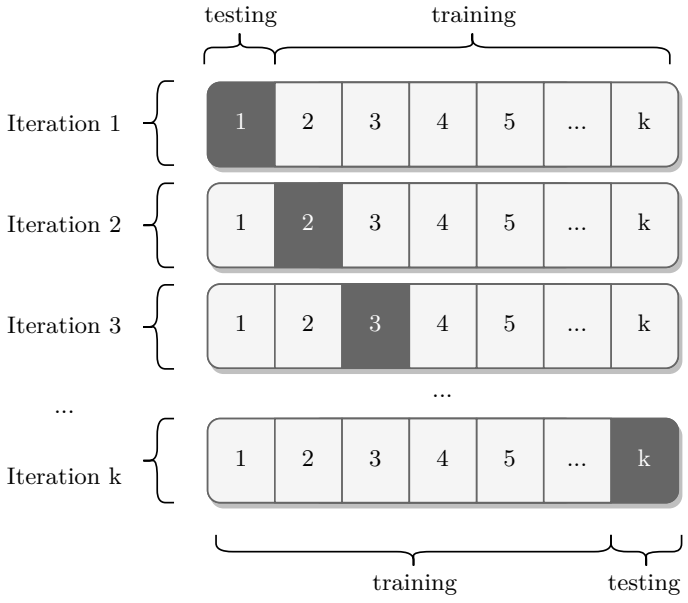


Figure 6.3: k-fold Cross-Validation.

6.3 Pre-Processing

This section describes the individual issues of the data that need to be addressed in order to achieve adequate results from the validation. The necessary steps of pre-processing differ from project to project and depend on the classification algorithm and the original condition of the data. Commonly, the raw dataset has some properties that could prevent algorithms from learning effective patterns. In the case of these data, these issues are primarily a *high class imbalance*, a very *high number of input features*, and *different feature scales*. The following subsections address each of these problems, explain them, and present the proposed solution to overcome them. Each subsection is structured in a first paragraph describing *the problem* and a second one describing *the proposed solution*. The solutions approach is at first described on a conceptual level, and subsequent paragraphs explain the chosen techniques in detail. The last part of each subsection provides a brief overview of *considered alternatives* that were implemented and tested but did not offer promising results.

6.3.1 High Number of Features

The high number of features can cause algorithms to perform worse than on a lower number. The bias and variance theory of Section 2.3 introduced information about how and why this can happen.

6.3.1.1 The Problem

It is likely that multiple features contain noisy and non-representative information. These features can cause the model to learn over-complex and barely generalizable patterns, which makes the model very sensitive to its training data. These things can finally result in over-fitted models and decrease the prediction capabilities due to an increased $Error_{variance}$. The provided data have a severe risk of over-fitting.

6.3.1.2 The proposed Solutions

A variety of measures was applied to address the risk of over-fitting. The risk of an *inappropriate algorithmic complexity* and an *inappropriate number of input features* are strongly related to each other and were primarily addressed by model-specific sorts of *regularization* and a *dimensionality reduction*. The chosen approach of dimensionality reduction is *Recursive Feature Elimination* via random forests. An appropriate algorithmic complexity can make an algorithm more robust to over-fitting caused by high numbers of input features, and hence it is a kind of preventative measure. The dimensionality reduction directly addresses the number of features and try to reduce it without losing something useful.

Dimensionality Reduction Classification algorithms are differently prone to over-fitting. Ideally, a dataset should only contain important features to reduce the risk of over-fitting and computational costs. In practice, it is not trivial to analyze which features are the important ones. The importance of features in a multi-variate manner depends on other included features. Hence, it is not necessarily meaningful to assess the importance of features in a univariate fashion. Methods of dimensionality reduction were considered to reduce the number of input features and the emerging risk of over-fitting. Dimensionality reduction encompasses different techniques to reduce the number of features. It is divided into two different sub-groups. On the one hand, *feature selection* approaches and, on the other hand, *feature extraction* approaches.

Feature Selection The *feature selection* approaches try to select the best-suited feature subset for a specific prediction problem. The group of feature selection approaches is again subdivided into three further groups [GE03]. The first group is considered as the *filter methods*. They work independently from the classification algorithm and are applied prior to the classification algorithm (e.g. by examining the correlation of features with the target variable). Filter methods apply a certain method to estimate the suitability of given features and remove those who seem to not contribute to the predictions. This leads to the problem that it does not remove features that harm the performance because they contribute. Hence, approximating the importance according to the contribution is not necessarily advantageous. The second type is considered as *wrapper methods*. These are applied in combination with a classification algorithm and allow more complexity in the assessment of importances. Standard approaches of wrapper methods are *forward- or backward-selection*. Wrapper methods are the most expensive approaches because they include repetitive evaluations by the classification algorithms. The third type of feature selection is considered as *embedded methods*. These techniques act as integrated parts of the classification algorithms (e.g. L1-regularization), which makes them rather cheap in terms of computational cost. They are more expensive than filter methods because they are a constant part of the algorithm but much cheaper than wrapper methods. Embedded methods are not available for all classification algorithms.

Feature Extraction *Feature extraction* approaches do not examine the actual features and decide about to use or discard them. They derive new synthetic features intended to be more informative and non-redundant while not losing important information. In some cases, even a loss of information can lead to an improved performance (e.g. due to an additional removal of noise). Common methods of feature extraction are *Principal Component Analysis* (PCA) or *Auto-Encoders* (AE).

Regularization *Regularization* means manifold methods used to prevent models from learning over-complex patterns. This can be achieved either by limiting algorithmic complexity or by penalizing high complexity. The first approach decreases the models' capability of learning complex decision boundaries, whereas the latter approach enforces a model only to increase the complexity if it provides a major benefit. Applied techniques of regularization will be described in the respective classifier's subsection of Section 6.4. The term *regularizing effect* is commonly used for any measures taken in order to reduce the final model's complexity. This can be also achieved by modifying the provided data. Hence, any type of dimensionality reduction has a regularizing effect.

Recursive Feature Elimination via Random Forests The technique of *Recursive Feature Elimination* (RFE) was introduced by Guyon et al.[GWBV02]. The authors have originally developed the RFE algorithm utilizing feature importances based on learned coefficients from SVMs. The algorithm works straightforward in a few steps. The first step is (1) the training of a classifier considering the current feature subset. Afterward, (2) the computed feature importances serve to rank the features according to their contribution. As the last step of each iteration, (3) the algorithm removes the n least important features from the provided subset and repeats this procedure until no features are left. A continuous performance validation after each training process allows a subsequent analysis of the performance progression throughout the entire RFE algorithm. The most suitable feature subset can be chosen based on the obtained performances during the procedure. This basic concept of successively removing the least important features is similar to a

Backward-search algorithm and can easily be modified in order to utilize any sort of feature ranking method, for instance, RFs. They were also successfully applied as part of RFE in [GMSP17]. Even though feature importances can show noisy information when extracted from highly correlated features [TL11] [GMSP17], RFE seemed to provide the most meaningful results of dimensionality reduction methods anyway. The error of feature importance approximation due to highly correlated features is considered as the *feature correlation bias* [GMSP17]. RF was chosen as the baseline classification algorithm for RFE because it seemed to outperform other algorithms already at model construction and optimization stages, and no other algorithm is immune to the feature correlation bias either.

The alternative *Non-Recursive Feature Elimination* algorithm was not considered due to its much higher proneness to the feature correlation bias [GMSP17]. The feature importance values of correlated features are pessimistically biased because they partially share the importance value over all correlated features [GMSP17]. When considering this on the actual dataset, this could cause that at some point, an informative feature ζ could be removed because of its least importance. This importance may have been assessed too low because another feature that is highly correlated to ζ is still present and got the lion's share of importance. Once there are no highly correlated features left, the importance is not shared anymore. The presence of feature correlation bias must be considered anytime when analyzing feature importances. The removal of an informative feature would cause a drop in the model's validation performance, which would be recognized in the subsequent analysis of the RFE algorithm. Only this analysis is decisive for the resulting reduced feature subset. Hence, the risk of discarded informative features in the resulting reduced feature subsets can be considered as rather low due to the manual inspection of the results.

The RFE algorithm was applied in the early stages of model construction and optimization. It was utilized to create additional feature subsets that might provide better performances as the original ones. Hence, RFE was executed on all defined feature subsets introduced in Section 6.2. This produced five additional features subsets, one RFE-version for each original feature subset. Additionally, RFE was also applied on all features (i.e. denoted as `ALL` feature subset) and has provided an additional `ALL_RFE` feature subset. All emerged RFE feature subsets correspond to the manually selected best-performing feature subset during the RFE algorithm. This exact procedure will be explained in Section 6.5.2. The main goal of the algorithm was to reduce the number of features (and hence the risk of over-fitting and high $Error_{variance}$) successively until the performance shows any drop. A performance drop indicates the removal of important information from the data. Besides just reducing the number of features without having performance decreases, an improvement of performance was expected. Some algorithms have mechanics that work better when there is no or less correlation present. It was also expected that at some point, the performance drops because the algorithm was forced to remove the least important feature continuously, even if all unimportant features were already discarded.

6.3.1.3 Considered Alternatives

A variety of multiple approaches were considered, implemented, and tested in order to improve the performance and reduce the risk of over-fitting. AE- and PCA-based feature extraction approaches were tested as part of the pre-processing. These approaches were used to provide a more meaningful feature representation to be used in the subsequent classification algorithms. Vanilla AEs [Kra91] and special Denoising AEs [VLBM08] [VLL+10] for tabular data were implemented and evaluated. Besides AE-based approaches, PCA was evaluated to reduce the number of features while also removing any feature correlation.

Furthermore, correlation-based feature selection was tested to identify high correlated features and discard one of the pairs. *Anomaly Detection* via AEs was also evaluated in order to identify samples that do not follow the meaningful pattern of training data and confuse the prediction algorithms. Those samples were removed before the data was fed to the supervised classifiers. None of these implementations provided sufficiently promising improvements to take place in the extensive validation run.

6.3.2 Imbalanced Class Distribution

This section addresses the problem of the highly imbalanced dataset. The first part explains why this can prevent models from learning meaningful patterns, and the subsequent sections propose solutions how this issue can be overcome. The final section lists alternative solutions that were considered.

6.3.2.1 The Problem

A high class imbalance can prevent some algorithms from learning a balanced decision boundary. Sometimes a class imbalance can lead to an algorithm being biased to the majority class due to its superior prior probability [JK19]. In these cases a model is not able to learn how to recognize the minority class. Some algorithms are more prone to this than others. The following paragraphs describe the effects of class imbalance on the individual classification algorithms.

The NN or FCN suffers the most from an imbalanced training dataset. An NN is trained using a certain cost or loss function. This means the algorithm adjusts its weights in order to minimize this loss with respect to the currently provided training samples (i.e. the mini-batch). The optimization algorithm computes gradients of the loss function with respect to every single weight.

The magnitudes of adjustments are substantially determined by how large the gradients are. The magnitudes of gradients are strongly determined by the amount of computed loss in the actual training batch. If the current training batch was predicted quite well, the loss becomes small, which in turn leads to small gradients. This is intuitive because a low loss typically indicates good predictions and just the need for some fine-tuning. Considering a case of a class imbalance of about 1:69 as it is in this dataset, the gradients become very small far before the model has learned a meaningful pattern to recognize the minority class. The very high loss at the beginning of training due to randomly initialized weights forced the model very quickly to predict the right labels for samples of the majority class. As soon as the model predicts the correct labels for the majority class ($\approx 99\%$ of samples), the gradients diminish immediately. Hence, the model stops its effective training, although it has not learned any useful pattern to recognize the minority samples yet. The minority class has a too small impact on the losses, and hence the model always tends to be biased towards the majority class. This class is accountable for the way larger proportion of the loss. If the model trains continuously in order to learn the pattern of the minority class, it is likely to over-fit before it has learned such a pattern (i.e. inappropriate training time). Anand et al. have examined and shown this inferiority of the minority class's gradients when training networks [AMMR93]. If a model would utilize a single-sample SGD, the result would be the same. The NN adjusts its weights 69 times to better fit the majority class and a single time to fit the minority class. The model is again likely to over-fit before it has learned any useful pattern to recognize minority class labels.

Tree-based models tend to behave more robust to imbalanced class distributions but are not immune to that at all. A learned model on highly imbalanced data is likely to be biased to the majority class. But in contrary to an NN-based model, a tree-based algorithm is not

completely unable to learn meaningful patterns. The effect of imbalance or corresponding countermeasures has to be analyzed in detail. It depends on the applied splitting criteria and precise training data.

Naive Bayes is robust to class imbalance. It applies prior probabilities to make predictions. These prior probabilities basically correspond to the number of training samples having a certain feature value, divided by the number of all samples of this certain class. Hence, the computations are made, based on only the priors of a single class. The prior probabilities of the classes are never combined or mixed, and hence the majority class does not mess up the computation or priors of the minority class.

6.3.2.2 The proposed Solutions

Potential solutions to overcome the issues caused by an imbalanced class distribution are *resampling* approaches or *cost modifications*. Resampling approaches try to even out the class balance by either adding samples to the minority class or removing samples from the majority class. Those resampling approaches are only applied to the training data in order to enable the model to learn a more balanced decision boundary. The test data always remains in its original condition because the test data are used to test the model under realistic circumstances. The resampling approaches can be subdivided into *over-sampling* and *under-sampling* approaches. Over-sampling approaches do either replicate available samples or create additional synthetic samples. Under-sampling approaches risk dropping meaningful information from the training data, which can harm the training process. Random over-sampling of available data risks to over-fit a model because it learns from the same samples multiple times. The approach of cost modifications does not change the samples. Still, it aims to force the model to pay more attention to the minority class by inducing additional weight to miss-classifications of these samples. The resampling approach was selected to address the issue of class imbalance because multiple papers reported positive results using this method. Furthermore, resampling is very generic and can be applied to arbitrary algorithms the same way (even though not all algorithms benefit from all resampling approaches).

Random Over-sampling The first selected resampling technique was random over-sampling. This method selects samples from the minority class randomly and duplicates them. The duplication of samples is performed until a particular class distribution has been reached. This desired distribution is variable and must be treated as a hyperparameter. Early model validations have shown the most positive effect when classes were utterly balanced. Several other publications reported positive results using over-sampling [PD07] [Tan15] [KMM+07].

Smote-resampling The second selected resampling technique is *Synthetic Minority Over-sampling Technique* (smote) [CBHK02]. The primary difference between smote-resampling and random over-sampling is that it adds additional synthetic samples instead of just replicating existing samples. The algorithm tries to produce other samples following the same pattern as the original ones. Therefore, (1) the algorithm selects a random sample from the minority class. Afterward, (2) it computes this sample's k nearest neighbors, while k is typically set to five. In the next step, (3) the algorithm selects one of these neighbors randomly, and finally (4) it creates a synthetic sample at a random point on the line (i.e. in feature space) between both samples. Therefore, both samples are considered as feature vectors corresponding to certain points in a multidimensional space. This approach

was applied multiple times in SDP [WLT16] [KMM+07] [PD07]. Especially Pelayo & Dick reported very successful results owed to smote-resampling[PD07].

6.3.2.3 Considered Alternatives

A variety of different class distributions was considered while evaluating the resampling techniques. Equal size of both classes after resampling has performed best reliably. Besides over- and smote-sampling, under-sampling was also tested. It performed constantly worse than both other approaches. This might be caused by the information loss due to discarding about 98% of samples of the majority class. Under-sampling has been reported as applied technique in multiple publications [GBD+10] [Gra12] [YLX+15] [SMWO11] [WSS14] [Tan15] [KMM+07]. It is likely that under-sampling can provide reasonable results if applied on datasets with a less severe class imbalance. Finally, multiple hybrid resampling configurations were tested. Hybrid means that over- and under-sampling methods were combined to achieve a little less extreme resampling. For instance, the utilization of under-sampling to shrink the majority class a bit and then over-sample the minority class until an even distribution has been reached.

6.3.3 Different Feature Scales

This section describes the problem of different feature scales. It follows the same structure as the previous sections. The first subsection explains the problem and the potential consequences. The second section provides a description of the proposed solution on how to overcome the issue, and the final one provides an overview of other considered alternatives.

6.3.3.1 The Problem

Different feature scales are a common issue of tabular datasets. In real-life data, individual features measure different observations, typically based on different feature scales. The number of `LoC` ranges from one to 5,127, whereas the values of the `LoC Comment Ratio` range from zero to one. Depending on the classification and optimization algorithm, this can lead to dominating features. Especially distance-based algorithms suffer from different feature scales. This happens because two values of an arbitrary feature x_1 showing values between zero and one are likely to have less distance than two values of a feature x_2, which shows values between one and 5,127. If those distances contribute to the prediction or learning, it is likely that some features dominate others.

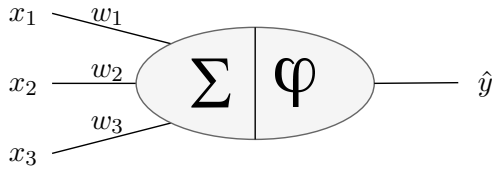


Figure 6.4: Single neuron neural network.

Another prominent algorithm that suffers badly from different feature scales is the NN. Figure 6.4 shows a single neuron network with three input features $x = (x_1, x_2, x_3)$ and corresponding weights $w = (w_1, w_2, w_3)$. The neurons input Σ is computed as shown in Equation 6.1.

$$\Sigma = \sum_{i=1}^{\forall x_i \in x} x_i w_i \tag{6.1}$$

Hence, the feature scales have a direct impact on the neuron's input sum Σ. The weights are typically randomly initialized following the same distribution. Thus, if a feature x_2 has larger values in magnitude, its contribution to Σ dominates others. In order to achieve an equal contribution to Σ, some weights must move dramatically. This compensation leads to learned weights on different scales. This is a severe problem because larger weights lead to high sensitivity to input data, which can contribute to over-fitting. Furthermore, different weight scales prevent any interpretation of coefficients in terms of feature importance assessments, and it also prevents the model from applying a balanced and appropriate regularization. This problem applies to regularization methods, which are based on additional regularization terms as part of the cost function to minimize as shown in Equation 6.2.

$$J(y, \hat{y}) = L(y, \hat{y}) + \lambda R(w) \tag{6.2}$$

These are, for example, L1-, L2- or L1-L2-regularization. The problem is that the regularization term forces the model to avoid high magnitudes of weights and try to keep all weights as low as possible. If weights on different feature scales are necessary in order to just even out the contribution to a neurons input Σ caused by different feature scales, the weights become regularized to different degrees. This could simply prevent the model from learning the right value of a certain weight just because it would lead to a too high regularization term of a certain high-scaled input feature.

Furthermore, Equation 6.3 shows that the vanilla SGD update rule learns the weights using a uniform learning rate α to control the step size of all weights equally.

$$w_i^{\text{new}} = w_i^{\text{old}} - \alpha \frac{\partial J}{\partial w_i} \tag{6.3}$$

This presumes that the gradients of all weights are roughly the same size. Considering that the gradient of a weight $\frac{\partial J}{\partial w_i}$ is computed as shown in Equation 6.4 and the final gradient $\frac{\partial \Sigma}{\partial w_i}$ is computed as shown in Equation 6.5, the input feature scales have direct impact on the magnitude of any weight's gradients.

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \varphi} \cdot \frac{\partial \varphi}{\partial \Sigma} \cdot \frac{\partial \Sigma}{\partial w_i} \tag{6.4}$$

$$\frac{\partial \Sigma}{\partial w_i} = x_i \tag{6.5}$$

If the input features are on different feature scales, the gradients become differently large as well. This would require multiple learning rates α to control the step size of all weights individually. NNs are prone to suffer from different feature scales because the scales have a significant impact on the optimization algorithm and predictions itself.

The positive effects of input normalization were also empirically studied by Lecun et al.[LBOM12]. Normalization is always reported as positively affecting the performance and decreasing the converging time of the optimization algorithm in the domain of deep learning. The use of normalization was even extended to approaches that constantly normalize the network's activations on each layer called *batch normalization* [IS15].

Tree-based models and NB behave scaling-invariant. Trees are split based on sample sizes and resulting class distributions. The different feature scales or the certain splitting values do not affect the prediction or optimization algorithm directly. NB does also not suffer from different feature scales. It makes predictions based on prior probabilities. The probabilities are not affected by the scaling of features because each feature's prior is computed individually. Whereas the feature's distribution does affect the algorithm, the features scaling does not.

6.3.3.2 The proposed Solutions

The issue of different feature scales will be addressed by applying normalization before fitting the data. Several normalizations and standardization techniques were tested at model construction and optimization stages. The L2-normalization method seemed to work best on NNs. NBs and tree-based models did not show significant changes. For a more unified setup across the models, normalization was applied to all algorithms. Even though it did not improve the models, it did not harm them either.

L2-Normalization L2-normalization computes the L2-norm for each feature based on the appearances of values in the training data. The computation of the norm is shown in Equation 6.6.

$$\|x_m\|_2 = \sqrt{\sum_{i=1}^{j} x_{m,i}^2} \tag{6.6}$$

Where x_m is an arbitrary feature of the feature vector $x = (x_1, x_2, x_3, ..., x_n)$, and n is the number of totally contained features. The index i iterates all training samples until the total number of samples j has been reached. Once the feature norms were computed, any samples' feature values are divided by the corresponding L2-norm as shown in Equation 6.7.

$$x'_m = \frac{x_m}{\|x_m\|_2} \tag{6.7}$$

Hence, the normalization outputs a vector x'_m which is one if all samples' values of feature x_m are squared and summed as shown in Equation 6.8.

$$\sum_{i=1}^{j} x'^2_{m,i} = 1 \tag{6.8}$$

The normalized values of $x'_{m,i}$ are in a range of $[0, 1]$.

6.3.3.3 Considered alternatives

Besides L2-normalization, *L1-normalization*, *MinMax-scaling*, and *standardization* were tested as well. L1-normalization works the same way as L2-normalization as given in 6.7, but it uses the L1-norm as defined in Equation 6.9.

$$\|x_m\|_1 = \sum_{i=1}^{j} |x_{m,i}| \tag{6.9}$$

MinMax-scaling rescales all values of a feature to $[0, 1]$ where the highest original value of the features is scaled to one, and the lowest value is scaled to zero. This sort of rescaling is computed as given in Equation 6.10.

$$x'_m = \frac{x_m - \min(x_m)}{\max(x_m) - \min(x_m)} \tag{6.10}$$

Standardization shifts the values of x_m to have zero mean and unit variance. Therefore, the mean $\bar{x}_m$ is subtracted from the original x_m and finally divided by the standard deviation σ_{x_m} as shown in Equation 6.11.

$$x'_m = \frac{x_m - \bar{x}_m}{\sigma_{x_m}} \tag{6.11}$$

6.4 Model Construction

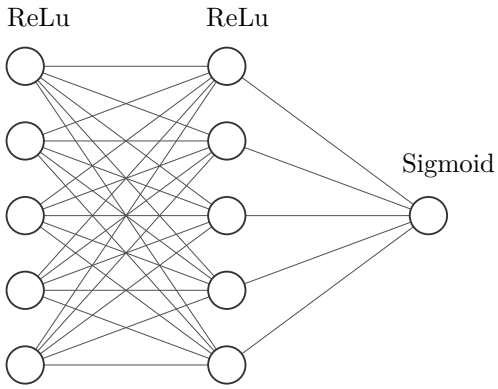
This section describes the process of model construction. This includes the finding of suitable architectures and other algorithm-specific hyperparameters. The explanations of chosen configurations are limited to the most important hyperparameters.

6.4.1 Fully Connected Network

The first model that was built is the FCN. FCNs are the prominent choice for non-time-series-based NNs on tabular data. The present prediction task is a standard binary classification problem. This implies the use of *binary cross-entropy* or *log loss* as the loss function. The architecture and other algorithm-specific hyperparameters were tuned using the outer validation set and the `ALL` feature subset as input features. This was done to ensure that the found architecture is sufficiently complex for any of the pre-defined feature subsets. A uniform architecture never provides the best possible results, but it led to a manageable computational effort. It is a compromise to find an architecture that is not under-complex for any feature subset. The optimization of the architecture was performed manually. The architecture was initialized with a single layer and just a very few neurons. At first, the number of neurons was successively increased until no further performance improvement could be achieved anymore. In the next step, a new layer was added. The neurons of this layer were again successively increased until the performance has stopped improving. This procedure was followed until neither additional layers nor additional neurons provided any further performance advantage.

The best performance and a reasonable amount of computational costs were achieved at 512 neurons on the input layer and on the first hidden layer. A single neuron on the output layer was defined because the final prediction is a single value. The required prediction is a probability that a sample belongs to the positive class (i.e. is a defect). This implies the use of the sigmoid activation function on the output layer because it gives outputs in the range of $[0, 1]$. The activation functions on input and hidden layers were changed from an initially used sigmoid function to ReLu functions. This change has provided additional improvements in performance and convergence speed. The final architecture is shown in Figure 6.5.

The applied optimization algorithm is the enhanced mini-batch SGD variant called *Adam* (Adaptive Momentum Estimation) [KB14]. The main advances of Adam compared to vanilla SGD are the use of momentum and adaptive learning rates. Momentum tries to minimize the risk of overshooting and slow convergence. It allows to change the weights fast if dramatic adjustments are needed, but it also reduces the learning rate if the weights



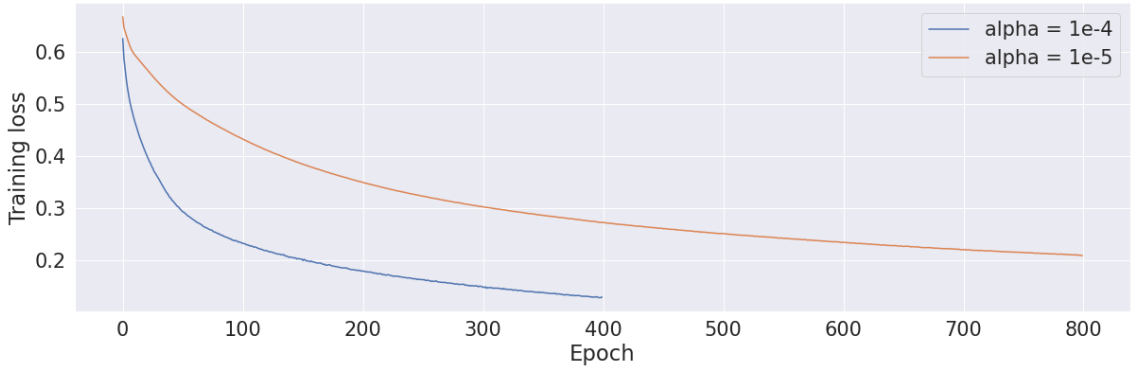
Input layer $\in \mathbb{R}^{512}$ Hidden layer $\in \mathbb{R}^{512}$ Output layer $\in \mathbb{R}^{1}$

Figure 6.5: Optimized FCN architecture.

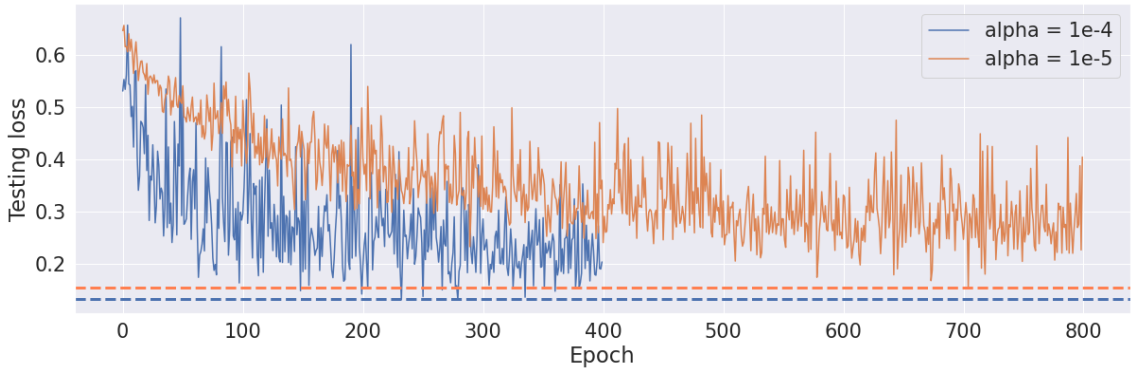
get close to the targeted local minimum of the cost function. Furthermore, the algorithm allows learning individual learning rates for each weight. The initial learning rate was set to *alpha*= 1e-4. The exponential momentum decays and the stability constant were chosen according to recommendations in [KB14] with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon =$ 1e-7. Several other parameters were considered during hyperparameter optimization. Besides Adam, RMSProp[5] and vanilla SGD were considered, but Adam has outperformed them reliably. The batch size was set to 128. A variety of other values was tested because the batch size has a major impact on the convergence speed and hence the runtime of the optimization algorithm. The batch size determines how many training samples are considered at a single training iteration within an epoch. Hence, a smaller batch size implies the need for more iterations to complete a single epoch. A larger batch size is desired to reduce the training time, but this also requires a higher amount of memory. This is due to the fact that the gradients' computation becomes more complex if more samples are considered at once. Larger batch sizes led to decreased performances, while smaller batch sizes did not lead to further improvements but longer training durations. Considered sizes were 32, 64, 256, 512, 1024, 2048, but 128 seemed to perform best and show a reasonable trade-off between performance and convergence speed. L2-regularization was chosen as a regularization method on all layers. The regularization term's coefficient was set to $\lambda = 0.01$. Besides L2-regularization, L1-, L1-L2-regularization, and *Dropout* [SHK+14] were considered but did not provide further improvements. A fixed number of epochs to train a network was chosen because the training process did not seem to converge smoothly, and hence an early stopping strategy was not an appropriate choice. Figure 6.6 shows the history of a training process under continuous validation on the outer validation dataset.

While Figure 6.6a shows the smoothly decreasing training error of two different learning rates ($\alpha =$ 1e-4 and $\alpha =$ 1e-5), Figure 6.6b shows the two heavily oscillating testing errors of the same training run. The oscillation also happened when the learning rate was decreased by magnitudes. Figure 6.6b shows that the decreased learning rate did not stabilize the training process but led to a worse achieved testing performance even if it was trained for twice the number of epochs. The horizontal dashed lines indicate the minimal loss values.

[5]RMSProp is an unpublished adaptive optimizer proposed by Geoff Hinton in his lecture. Available under: https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf



(a) Training errors of the FCN.



(b) Testing errors of FCN.

Figure 6.6: Loss curves of the constructed FCN at different learning rates.

The network was implemented that it reports the best achieved AUC-ROC measure and F1-score of the entire training run. Both values can be reported independently of each other. This means the network can report its best AUC-ROC measure from training epoch 200 while it also reports its best achieved F1-score from epoch 350. In contrary to that, the best achieved F1-score determines the reported values of precision and recall. These measures are associated with its F1-score, and hence it is not meaningful to provide maximized values of these measures independently. The implementation was done using TensorFlow 2.1. The general hyperparameters provided to the FCN in the final validation routine are L2-normalization and over- as well as smote-resampling. The FCN was not validated without using a resampling approach because this led to very poor performances. Reasons for that were given in Section 6.3.2.

6.4.2 Naive Bayes Classifier

The NB classifier is a simple probabilistic technique to predict labels. Due to its strong assumption of independence, the computations are quite simple. The algorithm does not require many algorithm-specific hyperparameters. A Gaussian kernel was applied as described in Equation 2.6 in order to compute prior probabilities of the numerical features. One additional hyperparameter named *var_smoothing* had to be set. It supported the computation stability for variances and was set to its default value of 1e-9. Despite the robustness of NB to imbalanced class distributions, the algorithm was validated using over- and smote-resampling as well as without any resampling. This was done to simply observe the effect of resampling manually. The applied implementation is based on the sklearn library.

6.4.3 Random Forest Classifier

The RF classifier is an ensemble technique based on a number of uncorrelated decision trees that perform a majority voting about which class to assign. In order to perform this majority voting, a set of trees must be learned. All trees are learned by a randomly selected subset of features and samples. The trees are learned in an ordinary fashion as described in Section 2.6. Decision trees come up with several hyperparameters determining the size and the shape of a tree. Furthermore, an ensemble technique comes along with additional hyperparameters controlling the composition of the ensemble itself. Finally, an RF ends up with plenty of different hyperparameters to be defined. The majority of these were kept on their default values according to sklearn. Considered hyperparameters at the model construction stage were the *number of estimators*, the *splitting criterion*, and the *maximal depth* of learned trees. The number of trees was set to 500. Figure 6.7 shows the changes in performance and runtime durations on an increasing number of estimators.

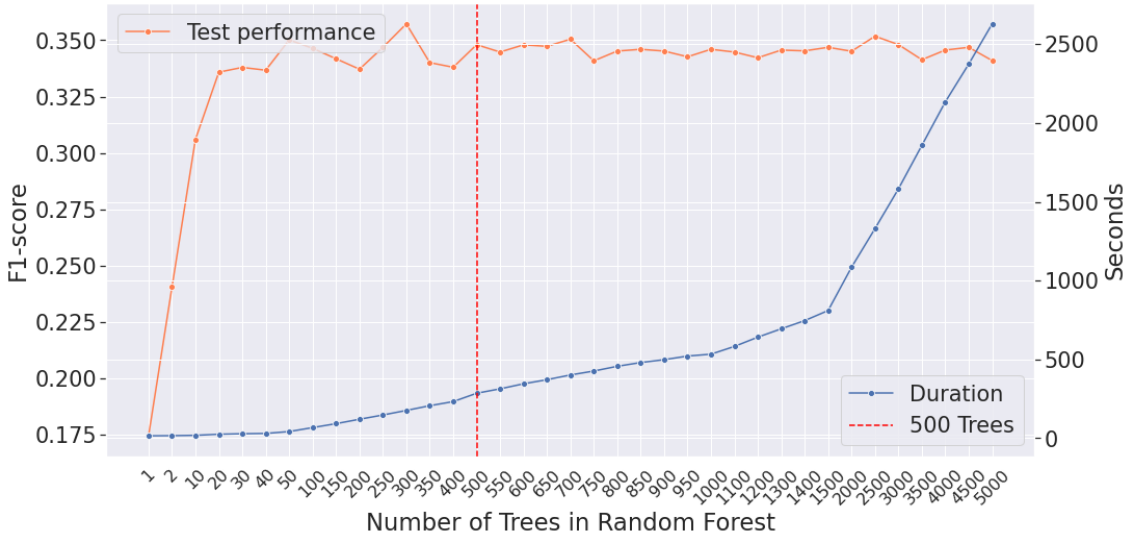


Figure 6.7: Optimization of the number of trees within the RF model.

Lower numbers of estimators seem to show slightly more fluctuations in performance, whereas the performance remains quite stable once the number of 500 trees is exceeded. This number also shows a reasonable runtime duration. A high number of estimators does generally not contribute to over-fitting but just stops improving the prediction capability at some point. Even a number of 5,000 estimators did not harm the performance at all but led to unreasonable runtime durations. The used training data for this plot were resampled by smote and used the `ALL` feature subset. This combination provides the highest complexity of data. Hence, the number of 500 estimators should be sufficiently high for any other feature subset or resampling approach.

The Gini Impurity was defined as the splitting criterion for all learned trees. The maximal depth of trees was not limited. The model optimization stage has shown that any tested limitation decreased the performance. The final validation routine of this algorithm considered over- and smote-resampling as well as no resampling. The applied implementation is based on the sklearn library.

6.4.4 Gradient Boosting Classifier

GB is the second tree-based classification algorithm and also the second ensemble technique. It provides the same hyperparameters as other tree-based models as well as some

additional ensemble specific hyperparameters. One of the most important hyperparameters to tune is the maximal depth of learned trees. In contrary to RF, a limitation of the maximal depth set to ten provided the best results. A possible reason for that is that RFs are very robust to over-fitting due to its bootstrap aggregating approach, which learns all estimators using random subsets of features and samples. Hence, the trees are already significantly less complex than they would be if learned using all samples and features. This is how GB works, and hence it requires additional measures of regularization (like limiting the trees' depths). Another important hyperparameter of GB is the number of estimators. This parameter is strongly associated with another hyperparameter denoted as *early stopping*. The early stopping criterion is realized by a number of allowed iterations (i.e. adding further estimators to the ensemble) without recognized improvement. It seemed to be a good combination to set the number of estimators rather high while preventing the model from over-fitting by the definition of such an early stopping criterion. The maximal number of estimators was set to 150, and the number of allowed iterations without improvements was set to ten. This allowed the algorithm to keep learning for many iterations but stop on convergence and before over-fitting happens. The learning rate of the GB algorithm was set to 0.1. The final validation routine considered over- and smote-resampling as well as no resampling for the GB algorithm. The applied implementation is based on the sklearn library.

6.5 Implementation

This section provides a detailed description of the implementation of the preparation pipeline, the RFE algorithm, and the final validation routine. The preparation pipeline is the component that takes the raw data, performs all configured steps of pre-processing, and returns the prepared data to be used as input in the classification algorithms. The RFE algorithm makes use of the preparation pipeline to provide additional feature subsets. The final validation routine takes all feature subsets and remaining configuration parameters and validates all of them according to the defined validation strategy introduced in Section 6.2.

6.5.1 Preparation Pipeline

The first implemented component is the preparation pipeline because it is used by the RFE algorithm as well as the final validation routine. The preparation pipeline takes the raw data as input and applies all defined steps of pre-processing according to Section 6.3. The pipeline contains the steps of (1) selecting the desired feature subset, (2) normalizing all data based on the training data, (3) removing *zero-important features*, (4) applying resampling and finally (5) returning the training and testing dataset in the appropriate data type. These steps must be performed in the proper sequence in order to preserve the validity of results and do not harm the prediction performance.

The pipeline is provided with a training and testing dataset, which in turn is returned by the 10-fold CV component. The first part of the pipeline extracts the desired feature subset from the provided data. Once the feature subset was selected from training and testing data, the normalization method is fitted. In the case of L2-normalization, this means the L2-norm is computed based on only the training data. Once the norm is computed, the normalization is applied to both, training and testing data. The pipeline takes a parameter that determines the desired sort of normalization method. Even though only L2-normalization is used within the validation routine, the implementation was done in a parametric fashion. An additional step of pre-processing, which was considered within the preparation pipeline, is the subsequent removal of zero-important features. These are features that turned out to be useless. Early investigations at model construction and

optimization stages have shown that some of the features show a minimal variance. In fact, the entire dataset has 434,772 samples that show the value zero in these features, and only a single sample that has value one. Obviously, these are no meaningful information to learn any patterns from. The affected features are listed in Table 6.1.

Table 6.1: Zero-importance features.

Metric
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)
Visible-TD x Feature Type(hex=1)
ALL Vars per Function x Feature Type(hex=1)
VP ND_Avg x Feature Type(hex=1)
EXTERNAL Vars per Function x Feature Type(hex=1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)
Full-TD x Feature Type(hex=1)
INTERNAL Vars per Function x Feature Type(hex=1)
VP ND_Max x Feature Type(hex=1)
CC on VPs x Feature Type(hex=1)

The pipeline takes a parameter determining whether these features shall be removed or not. Another parameter determines the resampling method to be used in this validation. It is essential that the resampling happens after the split into training and testing data and after the normalization. If the resampling would happen before the data were split, this could lead to similar samples in both subsets. This would severely compromise any results. The performances would become too optimistic because a model would be trained and tested on partly the same samples. Furthermore, the resampling happens after normalization because smote-resampling utilizes feature spaces and distances. Hence, different feature scales would severely affect this computation. The resampling is only applied on training data, whereas the testing data remain in its original shape. The testing data mimic a real case. If these data would be resampled to a synthetically equilibrated distribution, the performance measures would not provide meaningful real case estimations. The final step of the preparation pipeline is the execution of some minor implementation-specific data type conversions. It outputs completely prepared data which can be fed to the classification algorithm.

6.5.2 Recursive Feature Elimination

This section describes the implementation of the RFE algorithm in detail. RFE is originally considered as greedy wrapping feature selection method utilizing any supervised classification algorithm. The algorithm was used to produce additional feature subsets, which were also validated extensively as part of the final validation routine. This made the RFE algorithm acting like a filter-based feature selection method. The RFE algorithm makes use of the preparation pipeline in order to pre-process the data for the classification algorithm. The basic RFE algorithm was previously introduced in Section 6.3.1. This RFE implementation utilizes RF as its classification algorithm, which makes the result slightly biased in favor of RFs. Algorithm 2 shows the logical implementation of it. The subsequent analysis of all collected results of this algorithm served as a baseline to define the additional RFE feature subsets.

The algorithm used the same training and testing data, which were already used at model construction. The algorithm was applied on all pre-defined feature subsets described in Section 6.2 (i.e. `LNV`, `LPV`, `LCV`, `GPV`, `GCV`) and additionally on the `ALL` feature subset, which includes all software metrics. The algorithm iterates all provided feature subsets sequentially and reduces each feature subset until only a single feature is left. Any kind of

Algorithm 2 Recursive Feature Elimination via random forest classification

```
1: train ← dataset \ outerValidationSet
2: test ← outerValidationSet
3:
4: for all featureSubsets do
5:     featureSubset_ ← featureSubset
6:     while len(featureSubset_) > 1 do
7:         train_, test_ ← PreparationPipeline(train, test, featureSubset_, params)
8:         RF_ ← initialize(hps).train(train_)
9:
10:        saveResults(evaluate(RF_.predict(test_)))
11:        saveFeatureImportances(RF_.getFeatureImportances())
12:
13:        featureSubset_ ← featureSubset_.removeLeastImportantFeature()
14:    end while
15: end for
```

temporary object within the algorithm is denoted by a tailing `_`. In the first step of each inner iteration, the algorithm utilizes the preparation pipeline to pre-process the data. The *params* argument provided to the preparation pipeline includes any hyperparameters determining the desired steps of pre-processing. The RFE algorithm applied L2-normalization and no resampling. No resampling was used in order to avoid inducing an additional bias towards any resampling technique. Furthermore, the RFE algorithm does not aim for best performance but towards revealing insights into the performance history throughout the algorithm. RF was used as the classification algorithm because it behaves more robust to imbalanced class distribution than other algorithms and achieved meaningful results at reasonable computational costs at the model construction stage. After the data were prepared by the pipeline, a fresh RF classifier is initialized in each iteration. This RF model is trained on the prepared training data and predicts on testing data subsequently. The performance results are saved persistently for each iteration. Furthermore, the feature importances are extracted and stored on each iteration as well. The final step of each inner iteration is to analyze the feature importances and remove the least important one.

RFE Results This section presents the results of the performed RFE algorithm. Each plot shows the performance history of all recorded performance metrics throughout the entire algorithm. The vertical dashed grey line in each plot shows the iteration of the highest F1-score. This point of each plot determines the resulting RFE feature subset to be validated extensively. Two plots show an additional purple vertical line, which was defined manually because the point of the highest F1-scores did not seem to be the most promising one.

The first Figure 6.8a shows the performance history of the `LNV` feature subset. The plot shows a quite constant performance history. The point of the highest F1-score seems to be a good choice because, at this point, no performance metric has decreased yet. The next Figure 6.8b shows the performance history of the `LPV` feature subset. The plot shows an initially constant performance history but a slight decrease in AUC-ROC before the point of the highest F1-score has been reached. Hence, the decisive iteration for the RFE feature subset was manually adjusted to a previous point in order to omit the decrease in AUC-ROC within the resulting subset. The next Figure 6.8c shows the performance history of the `LCV` feature subset. The plot shows a rather constant history, and the point of the highest F1-score has been reached before the algorithm shows the slightest decrease in AUC-ROC measure. Hence, the point of the highest F1-score was considered as a meaningful choice. Figure 6.8d shows the performance history of the `GPV` feature subset.

The plot shows quite constant performances. The point of the highest F1-score has been reached rather early, while no considerable changes happened at this point. No significant increase happened before this point, and no decrease happened after this point. Hence, the decisive iteration for the RFE feature subset was manually moved some iterations further. This iteration shows a very similar performance than all previous ones and a significant decrease in performance afterward. Hence, it seems that this iteration provides an as-small-as-possible feature subset of a reasonable performance without any necessary information lost yet. Figure 6.8e shows the performance history of the `GCV` feature subset. The plot shows a quite constant, perhaps a slightly increasing performance while removing many features. The iteration of the highest F1-score is considered as meaningful because the subsequent iteration shows a significant drop in AUC-ROC and F1-score as well. Figure 6.8f shows the performance history of the `ALL` feature subset. The plot shows a considerable increase while constantly removing features. On a very late iteration, the plot shows a significant performance increase of F1-score and AUC-ROC measure. This sharp increase is followed by a sharp drop. The iteration of the highest F1-score seems to provide a significantly improved performance and a dramatic decrease in the number of features. The application of the RFE algorithm could reduce the number of contained features by a lot while it kept up or even improved the achieved performances. In order to validate the generalizability of these results, the final analyses of this set of experiments consider a comparison of achieved performance improvements at this point and achieved performance improvements at the extensive validation routine. The contained features of each RFE feature subset are shown in Tables 6.2, 6.3, 6.4, 6.5, 6.6, and 6.7.

The computation of feature importances The used sklearn library implemented the feature importance as described in [BFSO84]. The feature importance of a learned tree is calculated as the total decrease in a node's impurity, weighted by the probability of reaching that node. This probability is approximated by the proportion of samples reaching the node. Hence, a feature becomes more important if it is used on very top-level splits that hit a large part of all samples. In the case of an ensemble like RF, the importances are computed on each tree and are finally averaged over all.

Preliminary Findings

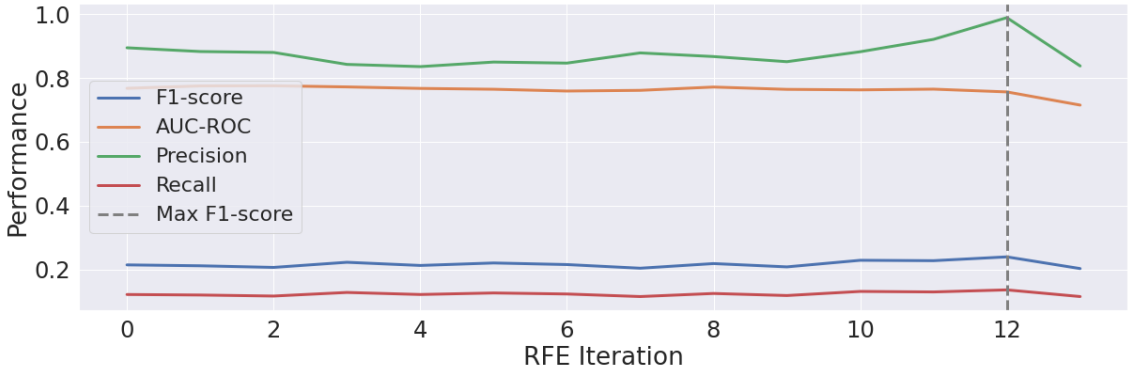
1. *The achieved F1-score of the LNV feature subset was improved by 11.68% with 3 out of 15 features (80% of features were removed).*

2. *The achieved F1-score of the LPV feature subset was improved by 6.65% with 5 out of 20 features (75% of features were removed).*

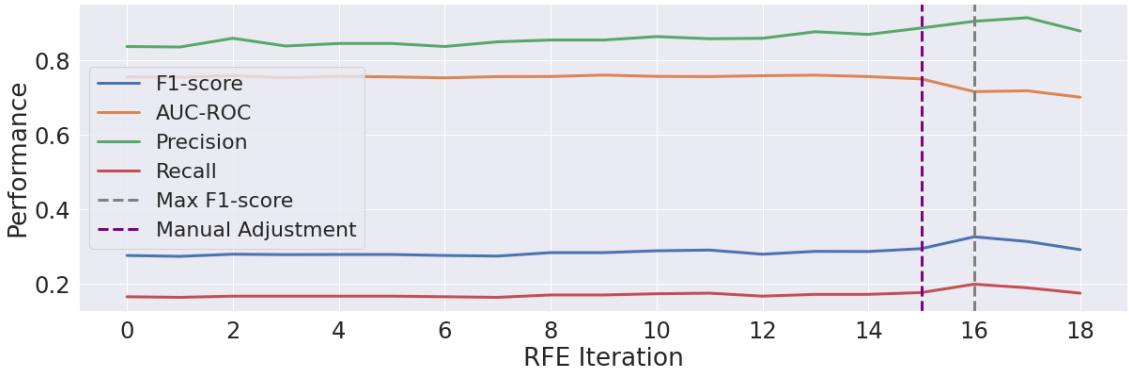
3. *The achieved F1-score of the LCV feature subset was improved by 14.78% with 9 out of 28 features (68% of features were removed).*

4. *The achieved F1-score of the GPV feature subset was improved by 1.26% with 20 out of 200 features (90% of features were removed).*

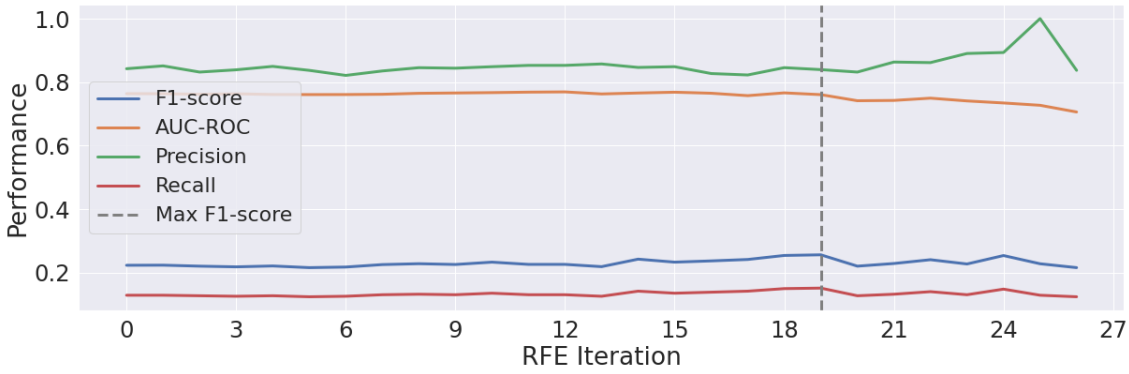
5. *The achieved F1-score of the GCV feature subset was improved by 38.57% with 9 out of 442 features (98% of features were removed).*



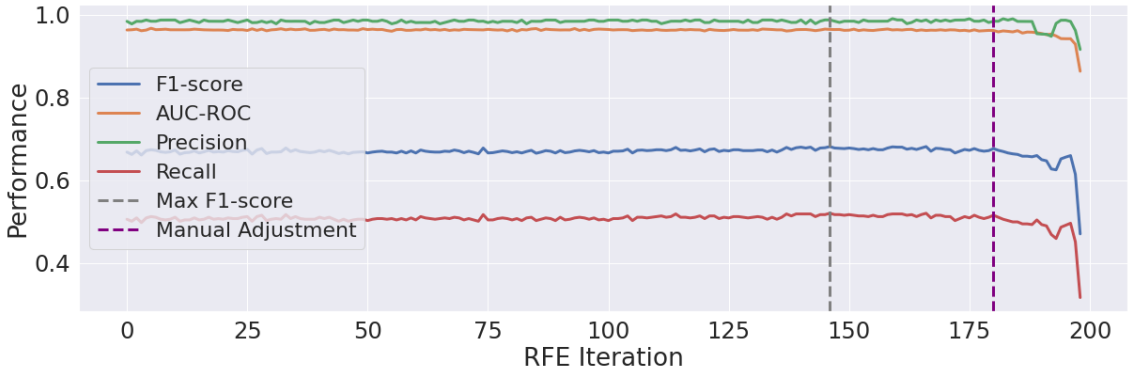
(a) Results of the LNV feature subset.



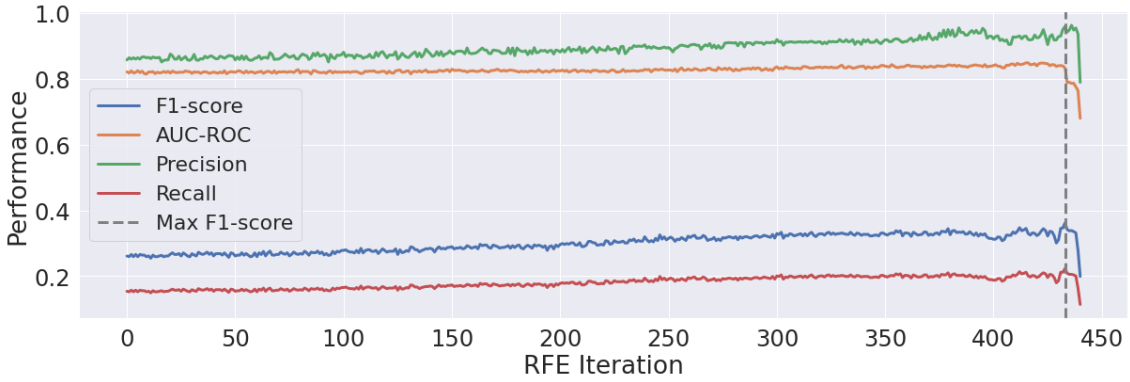
(b) Results of the LPV feature subset.



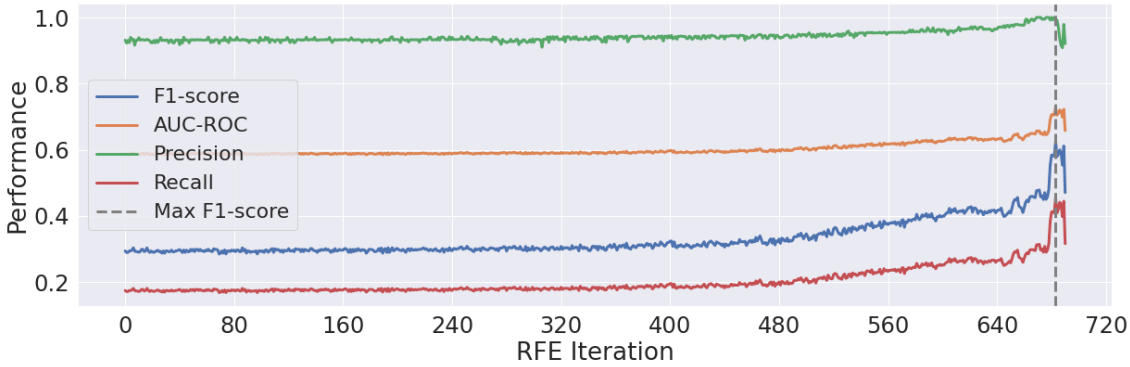
(c) Results of the LCV feature subset.



(d) Results of the GPV feature subset.



(e) Results of the GCV feature subset.



(f) Results of the ALL feature subset.

Figure 6.8: The results of the RFE algorithm for each feature subset.

6. *The achieved F1-score of the ALL feature subset was improved by 112.11% with 9 out of 692 features (99% of features were removed).*

Table 6.2: Software metrics in the LNV_RFE feature subset.

Metric
EV Classical Fan-In(global)
EV Classical Fan-In(local)
EV Classical Fan-Out(global)

Table 6.3: Software metrics in the LPV_RFE feature subset.

Metric
ALL_WITH_BUILD_VARS Vars per Function
EV VP Fan-In(global)
EV VP Fan-In(local)
EV VP Fan-Out(global)
EV VP Fan-Out(local)

Table 6.4: Software metrics in the LCV_RFE feature subset.

Metric
Combined ND_Avg
DC Fan-Out(global x No Stubs)
EV DC Fan-In(global)
EV DC Fan-In(local)
EV DC Fan-Out(global x No Stubs)
EV DC Fan-Out(global x No ext. VPs)
EV DC Fan-Out(global)
EV DC Fan-Out(local x No Stubs x No ext. VPs)
EV DC Fan-Out(local)

6.5.3 Validation Routine

This section describes the procedure of the final validation routine. It implements the validation strategy introduced in Section 6.2. It conducts the validation of all defined classification algorithms, feature subsets, and remaining hyperparameters like resampling methods. The validation routine stores any performance metrics of each learned and validated model. The resulting validation scores are the averages over all of the ten iterations of a single 10-fold CV run. The term *10-fold CV iteration* or *inner 10-fold CV iteration* denotes one out of ten training and testing passes of a model on the different fold compositions. The term *10-fold CV run* or *outer 10-fold CV iteration* encompasses all inner iterations.

The validation routine aimed to obtain reliable, robust, and non-biased information about the suitability of individual configurations. This suitability is measured by the pre-defined variety of performance metrics. The 10-fold CV strategy was defined as the most meaningful approach. The implementation of the validation routine is described by Algorithm 3.

As a first step of the algorithm, the outer validation set must be removed because it was already used to optimize the algorithms. The use of these data as part of the validation routine would provide too optimistic results. The routine iterates all pre-defined

Table 6.5: Software metrics in the GPV_RFE feature subset.

Metric
ALL_WITH_BUILD_VARS Vars per Function x ALL_CTCR
ALL_WITH_BUILD_VARS Vars per Function x COC
ALL_WITH_BUILD_VARS Vars per Function x Hierarchy Levels
ALL_WITH_BUILD_VARS Vars per Function x INCOMIG_CONNECTIONS
ALL_WITH_BUILD_VARS Vars per Function x NUMBER_OF_CHILDREN
ALL_WITH_BUILD_VARS Vars per Function x OUTGOING_CONNECTIONS
ALL_WITH_BUILD_VARS Vars per Function x POSITIVE_SIZES
ALL_WITH_BUILD_VARS Vars per Function x SD_FILE
ALL_WITH_BUILD_VARS Vars per Function x SD_VP
ALL_WITH_BUILD_VARS Vars per Function x TOTAL_SIZES
EV VP Fan-In(global)
EV VP Fan-In(local)
EV VP Fan-Out(global)
EXTERNAL_WITH_BUILD_VARS Vars per Function x ALL_CTCR
EXTERNAL_WITH_BUILD_VARS Vars per Function x COC
EXTERNAL_WITH_BUILD_VARS Vars per Function x INCOMIG_CONNECTIONS
EXTERNAL_WITH_BUILD_VARS Vars per Function x NUMBER_OF_CHILDREN
EXTERNAL_WITH_BUILD_VARS Vars per Function x POSITIVE_SIZES
EXTERNAL_WITH_BUILD_VARS Vars per Function x SD_VP
EXTERNAL_WITH_BUILD_VARS Vars per Function x TOTAL_SIZES

Table 6.6: Software metrics in the GCV_RFE feature subset.

Metric
EV DC Fan-In(local) x Feature distance
EV DC Fan-In(local) x Hierarchy Types(0-0-1)
EV DC Fan-In(local) x Hierarchy Types(0-1-0)
EV DC Fan-In(local) x NUMBER_OF_CHILDREN
EV DC Fan-In(local) x OUTGOING_CONNECTIONS
EV DC Fan-In(local) x POSITIVE_SIZES
EV DC Fan-In(local) x SD_FILE
EV DC Fan-In(local) x TOTAL_SIZES
EV DC Fan-Out(global) x Feature distance

Table 6.7: Software metrics in the ALL_RFE feature subset.

Metric
ALL_WITH_BUILD_VARS Vars per Function x POSITIVE_SIZES
ALL_WITH_BUILD_VARS Vars per Function x TOTAL_SIZES
EV DC Fan-In(local) x Feature distance
EV DC Fan-In(local) x NUMBER_OF_CHILDREN
EV DC Fan-In(local) x POSITIVE_SIZES
EXTERNAL_WITH_BUILD_VARS Vars per Function x COC
EXTERNAL_WITH_BUILD_VARS Vars per Function x INCOMIG_CONNECTIONS
EXTERNAL_WITH_BUILD_VARS Vars per Function x POSITIVE_SIZES
EXTERNAL_WITH_BUILD_VARS Vars per Function x TOTAL_SIZES

Algorithm 3 The feature subset comparison experimental routine

```
1: dataset ← dataset \ outerValidationSet
2:
3: for all predictionAlgorithms do
4:     for all featureSubsets do
5:         for all samplingMethods do
6:             for 0, n do # outer 10-fold iterations
7:                 k-folds ← StratifiedKFold(k, dataset)
8:                 for all train_fold, test_fold in k-folds do # inner 10-fold iterations
9:                     train, test ← PreparationPipeline(train_fold, test_fold, featureSubset,
   params)
10:
11:                    model ← predictionAlgorithm.initialize(hps).train(train)
12:                    saveResults(evaluate(model.predict(test)))
13:
14:                    if model is tree-based then
15:                        saveFeatureImportances(model.getFeatureImportance())
16:                    end if
17:                end for
18:            end for
19:        end for
20:    end for
21: end for
```

hyperparameters in a grid-based fashion. The implementation allows setting an additional parameter n, which determines the number of outer 10-fold CV iterations to perform on each configuration. However, this number had to be set to one. A higher number of outer 10-fold CV iterations would exceed the computational resources. In each of the outer 10-fold CV iterations, the provided dataset is split into ten folds according to the stratified approach described in Section 6.2. These folds provide ten different training and testing data compositions to validate. The first step of each validation starts with the application of the preparation pipeline in order to perform all necessary steps of pre-processing. The subsequent step is the initialization and training of the respective classifier. Each classification algorithm was built and optimized to have a single set of algorithm-specific hyperparameters, which were described in Section 6.4. Those parameters are applied and shared over all configurations for each classifier. Once the model was trained, it can be evaluated, and all performance metrics, as well as some other information about the configuration, are saved persistently. In the case that the actual classification algorithm is a tree-based one, the feature importances are extracted and saved as well.

Figure 6.9 shows a schematic summary of the implementation. It wraps up how the validation was performed and which parameters were validated. The outer cycle represents the grid-based iteration of the given hyperparameters. The intermediate cycle represents the $n = 1$ outer 10-fold iterations, and the innermost cycle shows the inner $k = 10$-fold iterations. Every single iteration trains and tests a certain configuration, and the final validation scores are provided as averaged measures for each outer cycle (i.e. for each configuration).

Any configuration to validate used L2-normalization and the removal of zero-important features. NB-, RF- and GB-based configurations were validated on over- and smote-resampling as well as without resampling. NN-based configurations were only validated using over- and smote-resampling. The final validation scores are defined as arithmetic means of each performance metric over all $k = 10$ iterations. In the case of a value $n > 1$, the validation scores are defined as the arithmetic mean over all k inner iterations and all n outer iterations.

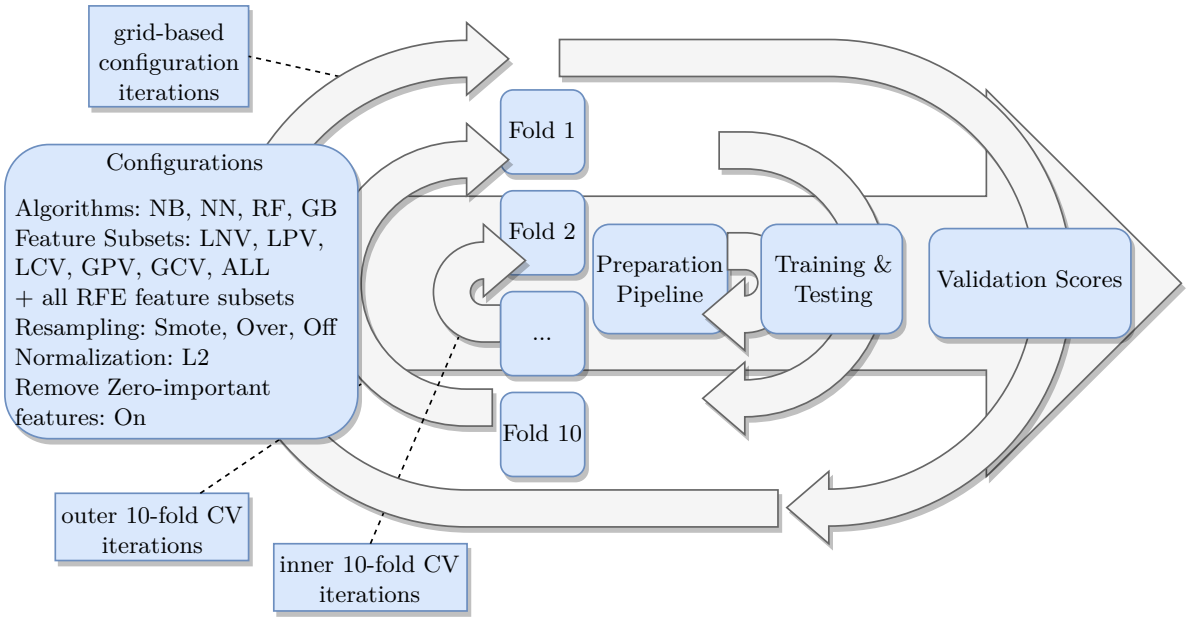


Figure 6.9: Validation routine setup.

6.6 Analysis

This section provides and discusses the results of the first set of experiments. Table A.9 in the Appendix shows all achieved validation scores of each recorded performance metric. The table is ordered according to achieved F1-scores. Those scores are based on the arithmetic mean (∅) over all iterations of a respective configuration. The duration is the summation (Σ) over all inner 10-fold iterations each configuration. It includes the training and testing time each iteration.

6.6.1 Comparison of Feature Subsets

This section shows a particular view on the results in order to address the question about the superiority of specific feature subsets over others. Table 6.8 presents an overview of the best achieved performances and its corresponding configuration for each feature subset. The best performance was determined by the highest achieved F1-score.

Table 6.8: Best achieved validation scores of each feature subset.

Subset	Model	Sampling	Precision ∅	Recall ∅	F1 ∅	AUC-ROC ∅
GPV_RFE	RF	Off	0.977432	0.506960	0.667468	0.959948
GPV	RF	Off	0.973382	0.501605	0.661919	0.960251
ALL_RFE	RF	Off	0.976596	0.422435	0.589675	0.955442
ALL	GB	Over	0.355559	0.600401	0.446027	0.924765
GCV_RFE	RF	Smote	0.417775	0.313305	0.357548	0.734410
LPV_RFE	RF	Off	0.891265	0.174754	0.291786	0.744306
LPV	GB	Off	0.731670	0.169583	0.274936	0.723002
GCV	RF	Off	0.897166	0.155492	0.264803	0.820075
LCV_RFE	RF	Off	0.854152	0.142116	0.243438	0.765221
LNV_RFE	RF	Off	0.939705	0.133560	0.233659	0.747704
LNV	RF	Smote	0.339268	0.159770	0.216925	0.705323
LCV	RF	Off	0.845451	0.110912	0.195932	0.765334

Table 6.9: Performance comparison between local-scoped non-variability-awareness (LNV_RFE) and all others.

From	To	± Precision	± Recall	± F1	Welch-test F1	± AUC-ROC
LNV_RFE	LPV_RFE	- 5.15%	+ 30.84%	+ 24.88%	✓	- 0.45%
LNV_RFE	LCV_RFE	- 9.10%	+ 6.41%	+ 4.19%	✗	+ 2.34%
LNV_RFE	GPV_RFE	+ 4.01%	+ 279.57%	+ 185.66%	✓	+ 28.39%
LNV_RFE	GCV_RFE	- 55.54%	+ 134.58%	+ 54.02%	✓	- 1.78%

The `GPV` and `GPV_RFE` feature subsets surpassed all others. Both achieved an approximately 12.25% higher F1-score than the `ALL_RFE` feature subset. The `ALL_RFE` feature subset achieved an about 32% higher F1-score than its original version. Below the `GPV` and `ALL` versions, `GCV_RFE` achieved the next highest F1-score. The original version of `LPV` and `GCV` reached a quite similar F1-score. Hence, the local-scoped pure-variability-aware metrics produced similar performances as the global-scoped combined-variability-aware metrics. The lowest F1-scores were achieved by the `LNV` and `LCV` feature subsets, whereas the `LNV` subset also shows the worst AUC-ROC measure. The best feature subset of `GPV_RFE` performed about 240% better than the worst feature subset of `LCV`.

6.6.2 Non-Variability-Awareness vs. Variability-Awareness

This section shows a certain view on the results in order to address the question about the superiority of variability-awareness over non-variability-awareness. Table 6.9 presents a comparison of achieved performances of the best non-variability-aware feature subset (`LNV_RFE`) and all others.

The second Table 6.10 presents a comparison of the feature subset that achieved the overall best validation scores (`GPV_RFE`) and all others. In order to provide a more robust insight and less cherry-picked results, the best version of each feature subset was chosen in all cases (i.e. either the RFE feature subset or the original one). In order to provide statistically validated results about superiority or inferiority of variability-awareness, *Welch's t-tests* were performed.

Welch's t-test for unequal variances The Welch's t-test for unequal variances or simply *Welch-test* was selected because it does not assume the same variances of both samples under comparison. Even in the case of very similar variances performs the Welch-test as robust as the *Student's t-Test* [Rux06]. The Welch-tests were applied to the F1-scores of the models. The level of significance α was set to 0.05. The null hypothesis H_0 states the means of given samples are equal. If a computed p-value was $\leq \alpha$, the null hypothesis was rejected. If H_0 was rejected, it means the validation scores of given samples are statistically unequal, and therefore one sample is either significantly better or worse than the other. This is indicated by a positive or negative difference between both samples as shown in the comparison tables. Whether H_0 could be rejected is indicated by a ✓. If a Welch-test was not passed, and H_0 could not be rejected; this is indicated by an ✗.

The results of experiments show that variability-awareness does not necessarily improve the prediction performance. The non-variability-aware software metrics achieved similar F1-scores as combined-variability-aware metrics on a local scope (`LNV_RFE` vs. `LCV_RFE`; H_0 not rejected with p-value ≈ 0.267). The non-variability-aware metrics achieved significantly worse F1-scores than pure-variability-aware metrics on a local (`LNV_RFE` vs. `LPV_RFE`; H_0 rejected with p-value $\approx$ 5.19e-05) and on a global scope (`LNV_RFE` vs. `GPV_RFE`; H_0 rejected

with p-value ≈ 1.79e-19). The combined-variability-aware metrics achieved significantly higher F1-scores only on a global scope (`LNV_RFE` vs. `GCV_RFE`; H_0 rejected with p-value ≈ 5.75e-11).

Table 6.10: Performance comparison between global-scoped pure-variability-awareness (GPV_RFE) and all others.

From	To	± Precision	± Recall	± F1	Welch-Test F1	± AUC-ROC
GPV_RFE	LNV_RFE	-3.86%	- 73.65%	- 64.99%	✓	- 22.11%
GPV_RFE	LPV_RFE	- 8.82%	- 65.53%	- 56.28%	✓	- 22.46%
GPV_RFE	LCV_RFE	- 12.61%	- 71.97%	- 63.53%	✓	- 20.29%
GPV_RFE	GCV_RFE	- 57.26%	- 38.20%	- 46.43%	✓	- 23.49%

Table 6.10 shows that the global-scoped pure-variability-aware metrics surpassed any other metric subset. This finding was validated by Welch-tests in all cases. This metric subset achieved higher F1-scores than non-variability-aware metrics (`GPV_RFE` vs. `LNV_RFE`; H_0 rejected with p-value ≈ 1.16e-18), pure-variability-aware metrics on a local scope (`GPV_RFE` vs. `LPV_RFE`; H_0 rejected with p-value ≈ 2.34e-15) and combined-variability-aware metrics on a local (`GPV_RFE` vs. `LCV_RFE`; H_0 rejected with p-value ≈ 9.15e-22) as well as on a global scope (`GPV_RFE` vs. `GCV_RFE`; H_0 rejected with p-value ≈ 4.06e-18).

Findings

1. *Non-variability-awareness did not outperform any variability-aware feature subset.*
2. *Pure-variability-awareness on a global scope outperformed any other variability-awareness on any scope significantly.*

6.6.3 Local Scope vs. Global Scope

This section shows a certain view on the results in order to address the question about the superiority of global scopes over local scopes. Table 6.11 shows the improvements achieved by changing from local-scoped to global-scoped metrics.

Table 6.11: Performance comparison between local- and global-scoped software metrics.

From	To	± Precision	± Recall	± F1	Welch-Test F1	± AUC-ROC
LPV_RFE	GPV_RFE	+ 9.66%	+ 190.10%	+ 128.75%	✓	+ 28.97%
LCV_RFE	GCV_RFE	- 51.09%	+ 120.46%	+ 46.87%	✓	- 4.03%

Because only variability-aware software metrics can be collected on a global scope, there is no global version of the non-variability-aware software metrics subset available. As in previous analyses, it was always the better version of each feature subset used (i.e. either the RFE feature subset or the original one). Welch-tests were performed as previously described in Section 6.6.2.

The performance of pure-variability-aware software metrics increased by about 129% (`LPV_RFE` vs. `GPV_RFE`; H_0 rejected with p-value ≈ 2.34e-15) in the achieved F1-score. The performance of combined-variability-aware software metrics increased by about 47% (`LCV_RFE` vs. `GCV_RFE`; H_0 rejected with p-value ≈ 4.68e-11) in terms of F1-score. Hence, the extension of the scopes led to significantly improved results in all cases.

Findings

3. *Changing the scope from local to global improved the achieved F1-scores in all cases significantly.*

6.6.4 Comparison of Classification Algorithms

This section shows a certain view on the results in order to address the question about any superiority of certain classification algorithms over others. The ulterior motive was that the removal of poor-performing classification algorithms could decrease the computational costs of subsequently performed experiments. Table 6.12 presents an overview of the best achieved performances of each applied classification algorithm, regardless of configurations. It is difficult to assess the general suitability of an algorithm because it always depends on all applied hyperparameters. Hence, basically no classification algorithm is generally superior over others, but it is likely that depending on a certain task and available data, one algorithm is able to achieve better results than others. Every algorithm made predictions for any considered configuration of the remaining hyperparameters. This allowed assessing the suitability of the algorithms in this specific task using the pre-defined algorithm-specific and general hyperparameters.

Table 6.12: Best achieved validation scores per classification algorithm.

Model	Subset	Sampling	Precision ∅	Recall ∅	F1 ∅	AUC-ROC ∅
RF	GPV_RFE	Off	0.977432	0.506960	0.667468	0.959948
GB	GPV_RFE	Off	0.878049	0.477176	0.617969	0.910329
NN	ALL	Smote	0.224262	0.262657	0.239804	0.756442
NBG	LNV	Off	0.038435	0.125534	0.058670	0.635658

The results show that NB-based models performed worst. No configuration using the Bayesian algorithm achieved an F1-score above 0.06 (best F1-score of a Bayesian model is 0.05867 on the `LNV` feature subset without resampling). NNs did not perform well in this task either. They only achieved about half as good F1-scores as GBs and RFs. The RF algorithm performed best overall. It even surpassed the second-best algorithm GB significantly. This was validated by a Welch-Test according to the description given in Section 6.6.2 (H_0 rejected with p-value $\approx$ 2.1e-05).

Table 6.8, which shows the best configurations of each feature subset, only contains RF models (10 models) and GB models (2 models). There is no NN or Bayesian model that performed better than RFs or GBs on any feature subset. Hence, all models applying NNs or Bayesian approaches did not really contribute to the analysis.

The results suggest that the RF models performed particularly well if applied on small- or moderate-sized feature subsets. RFs surpassed GBs on ten out of 12 feature subsets as shown in Table 6.8. The GB algorithm seems to handle the high amount of correlation among the `ALL` feature subset better than the RF algorithm. A reason for this could be that the strength of RFs arises with the majority voting of as many as possible uncorrelated decision trees. A high correlation among a feature subset increases the correlation between the learned trees, which reduces the reliability of the final majority voting. This could have happened because the contained trees within the RF ensemble utilized similar information (due to correlated features) for their predictions even though they picked random features. Once the `ALL` feature subset was reduced by the RFE algorithm, RF achieved higher F1-scores than GB on it.

Findings

4. *Random forests performed best on the vast majority of feature subsets.*
5. *Random forests achieved the best validation performance overall.*
6. *Bayesian models and neural networks did not achieve any meaningful validation performances.*

6.6.5 RFE vs. normal Feature Subsets

This section presents a certain view on the results in order to address the question of whether the RFE algorithm could improve the final validation performances similarly to the improvements at the Modeling stage. Table 6.13 shows a comparison of all original feature subsets and their corresponding RFE feature subsets. In order to validate any performance changes, Welch-tests were performed as described in Section 6.6.2.

Table 6.13: Performance improvements from the original to the RFE-reduced feature subsets.

From	To	± Precision	± Recall	± F1	Welch-Test F1	± AUC-ROC	± Number of Features
LNV	LNV_RFE	+ 176.98%	- 16.40%	+ 7.71%	✗	+ 6.01%	- 80%
LPV	LPV_RFE	+ 21.81%	+ 3.05 %	+ 6.13%	✓	+ 2.95%	- 75%
LCV	LCV_RFE	+ 1.03%	+ 28.13%	+ 24.25%	✓	- 0.01%	- 68%
GPV	GPV_RFE	+ 0.42%	+ 1.07%	+ 0.84%	✗	- 0.03%	- 90%
GCV	GCV_RFE	- 53.43%	+ 101.49%	+ 35.02%	✓	- 4.03%	- 98%
ALL	ALL_RFE	+ 174.67%	- 29.64%	+ 32.21%	✓	+ 3.32%	- 99%

The achieved F1-scores of the `LNV` (`LNV` vs. `LNV_RFE`; H_0 not rejected with p-value $\approx$ 0.051) and `GPV` (`GPV` vs. `GPV_RFE`; H_0 not rejected with p-value $\approx$ 0.38) feature subsets did not improve significantly. The achieved F1-scores of the `LPV` (`LPV` vs. `LPV_RFE`; H_0 rejected with p-value $\approx$ 0.15), `LCV` (`LCV` vs. `LCV_RFE`; H_0 rejected with p-value $\approx$ 8.6e-06), `GCV` (`GCV` vs. `GCV_RFE`; H_0 rejected with p-value $\approx$ 1.1e-08) and `ALL` (`ALL` vs. `ALL_RFE`; H_0 rejected with p-value $\approx$ 1.68e-13) feature subsets improved significantly.

The results show that the RFE algorithm could significantly improve the F1-scores in four cases. The RFE algorithm led to improved F1-scores on `LNV` and `GPV` feature subsets, but this could not be validated by Welch-tests, and hence it is not statistically significant. Not a single feature subset achieved a decreased F1-score on its corresponding RFE feature subset. In order to assess the benefit of the applied RFE algorithm, the high number of removed features has to be taken into account as well. Hence, even a non-significant improvement is an advantage when considering that the number of contained features was reduced by 80% (`LNV` to `LNV_RFE`) and 90% (`GPV` to `GPV_RFE`) respectively. The statistically improved feature subsets achieved higher scores while also reducing the number of features up to 99% (`ALL` to `ALL_RFE`). The high number of removed features by the RFE algorithm allows an easier interpretation regarding the software metrics' importances.

Findings

7. *The RFE algorithm provided significantly improved F1-scores while considering up to 99% fewer features.*
8. *The RFE algorithm provided significant improvements in F1-scores in four out of six cases.*

9. *The RFE algorithm did not provide any inferiority due to reduced features in any case.*

6.6.6 Comparison of Resampling Methods

This section presents a certain view on the results in order to address the question about the superiority of one sampling method over another. The highest achieved F1-score overall did not utilize any resampling. Table 6.8 shows that no resampling was used in nine out of twelve of the best-performing configurations of each feature subset. Table 6.14 shows the overall best achieved F1-scores of each applied resampling method. The best achieved F1-score using no resampling is significantly higher than the one of smote-resampling (H_0 rejected with p-value $\approx$ 1.96e-11) and over-sampling. Smote-resampling did not achieve a significantly higher F1-score than over-sampling (H_0 not rejected with p-value $\approx$ 0.08).

Table 6.14: Best achieved validation scores per resampling method.

Sampling	Model	Subset	Precision ∅	Recall ∅	F1 ∅	AUC-ROC ∅
Off	RF	GPV_RFE	0.977432	0.506960	0.667468	0.959948
Smote	RF	GPV	0.403128	0.638015	0.493644	0.941005
Over	RF	ALL_RFE	0.452437	0.494298	0.471758	0.915004

10. *Training without resampling led to the significantly best performance overall.*

6.6.7 Training Durations

This section presents a certain view on the results in order to assess the efficiency and overall computational costs of the models. For the sake of simplicity, the computational costs were defined as the average duration time needed for training and testing a single model within the validation routine (i.e. a single inner 10-fold iteration).

The needed time for model validation differed across the classification algorithms, feature subsets, and resampling approaches. Table 6.15 shows averaged durations for individual configurations of this first set of experiments. All durations were up-rounded to full numbers of minutes. Even though the computational costs should not become a first priority concern, it is worth observing these parameters because, in the case of a grid-based validation routine, a single additional algorithm to validate can lead to a substantially increased overall duration time.

The RF models constantly show short durations for validation. The RF approach benefited from its highly parallelizable nature and the high number of available CPU cores of the experimental environment, which will be described in Section 6.7. This led to a maximal averaged training and testing duration of six minutes, which gives a total of 60 minutes for an entire 10-fold CV run of a single configuration. The RF durations increased slightly according to the number of features within a feature subset. The non-RFE feature subsets show longer durations than the corresponding RFE versions generally. Smote- and over-sampling caused the algorithm to take significantly longer for training and testing. This is probably owed to the fact that both resampling methods lead to twice the number of training samples to analyze while learning the trees. RFs have shown constantly reasonable durations. In contrary to the tree-based classifier, NNs show rather constant durations over all feature subsets. This is probably caused by the fact that each NN was trained for a fixed number of epochs, regardless of the number of input features. The increased durations of the NNs for `ALL`, `ALL_RFE`, and `GCV` feature subsets were caused by the fact that these had to be validated on different machines than the other feature subsets. The longer durations

Table 6.15: Average validation durations of single inner 10-fold CV iterations in minutes.

Algorithm	GB			NBG			NN		RF		
Sampling	Off	Over	Smote	Off	Over	Smote	Over	Smote	Off	Over	Smote
Feature Subset											
ALL	130	1000	1102	1	1	1	225	201	3	5	5
ALL_RFE	6	22	27	1	1	1	114	116	1	1	1
GCV	62	843	917	1	1	1	166	150	3	5	6
GCV_RFE	2	22	28	1	1	1	76	62	1	2	2
GPV	44	149	172	1	1	1	69	70	1	2	2
GPV_RFE	8	34	41	1	1	1	62	62	1	1	1
LCV	4	59	63	1	1	1	60	59	1	2	2
LCV_RFE	1	14	25	1	1	1	60	59	1	1	2
LNV	2	41	45	1	1	1	52	52	1	1	1
LNV_RFE	1	10	12	1	1	1	59	59	1	1	1
LPV	3	21	24	1	1	1	60	59	1	1	1
LPV_RFE	1	7	8	1	1	1	56	61	1	1	1

happened on machines that did not have GPUs available. The shortest averaged training and testing duration of an NN-based model was about 52 minutes, which is still nine times longer than the longest averaged learning and testing duration of the RF-based models. The GB-based models show by far the longest durations. This is probably caused by their sequential nature and still enforced by high numbers of input features. The exact longest duration of a single entire 10-fold CV run was 660,620 seconds and is shown in Table A.9. This corresponds to approximately seven and a half days of computational time. This is an approximately 218 times longer duration than the longest RF-based validation duration. Any NB-based model shows negligible durations of less than one minute. The tree-based models also show longer durations on smote-resampled training data than on over-sampled training data. This could be caused by more complex data, which is likely to require the algorithm to perform more splits until convergence. This entire set of experiments took over 1,311 hours of computation overall. This computation effort was parallelized as much as possible and finally took approximately seven days for one pass.

Findings

11. *Random forests achieved the best performances overall while also showing the best trade-off between performance and computational time.*

6.6.8 Interpreting the Models Performance

This section provides a more practical analysis of the achieved performance metrics. The use of performance metrics is essential in order to evaluate and compare the quality of models. Still, those numbers do not really provide a clear understanding of a model's prediction capability under deployed circumstances. The performances of a few meaningful models were analyzed, and corresponding confusion matrices were reconstructed in order to achieve a more understandable performance interpretation. The confusion matrix, as described in Section 2.9, provides clear numbers regarding the different types of mistakes made by a model.

In the case of SDP, it might be more important to find as many prone functions as possible. It is not that problematic if a reasonable number of healthy functions were miss-classified as defects. The final classification of a model basically corresponds to the output of a simple threshold function for a passed continuous value. This continuous value could be considered as the actual model's prediction, which can be interpreted as a probability. In

the case of SDP, this probability could be interpreted as the "risk of a defect". Hence, a test sample that is actually labeled as healthy could still indicate a high risk of a defect. Thus, the *False Positive Rate* is rather negligible, and a high recall could be considered as preferable. Only considering recall as performance metric would lead to poor classifiers since simply classifying anything as defective would achieve perfect recall measures. A meaningful solution to prioritize the recall, but still exclude dumb classifiers, is to adjust the F1-score. The F1-score can be generalized to the Fβ-score as described in Section 2.9.5. The Fβ-score allows to increase or decrease the weight of recall according to the value of β. Table 6.16 shows the top ten models in terms of achieved F2-scores. The F2-score was computed according to Equation 6.12.

$$F2\text{-}score = (1 + 2^2) \cdot \frac{precision \cdot recall}{(2^2 \cdot precision) + recall} \tag{6.12}$$

Table 6.16: Top ten models according to achieved F2-scores.

Model	Subset	Sampling	Precision ∅	Recall ∅	F1 ∅	F2 ∅	AUC-ROC ∅
RF	GPV_RFE	Smote	0.357662	0.674045	0.467152	0.572497	0.946616
	GPV	Smote	0.403128	0.638015	0.493644	0.570993	0.941005
	GPV_RFE	Off	0.977432	0.506960	0.667468	0.560897	0.959948
	GPV	Off	0.973382	0.501605	0.661919	0.555400	0.960251
GB	ALL	Over	0.355559	0.600401	0.446027	0.527076	0.924765
	GPV_RFE	Off	0.878049	0.477176	0.617969	0.524988	0.910329
	GPV	Off	0.874358	0.474130	0.614222	0.521666	0.906361
RF	GPV	Over	0.349952	0.576320	0.435383	0.510206	0.928854
	GPV_RFE	Over	0.326170	0.587555	0.419393	0.506300	0.928519
GB	GPV	Over	0.230018	0.684737	0.344278	0.490570	0.926524

The use of F2-score allows to assess the prediction capability of a model considering achieved recall and precision but prefer models in favor of high recall values. Table 6.16 shows that the use of the F2-score would provide another best-performing configuration. As explained earlier, the use of resampling forces a model to learn a more balanced pattern. This does not necessarily lead to a finally better performance in any measure, but it typically ends up in a totally different way of making predictions. The RF-based models using smote-resampling have learned a more balanced pattern, which led them to achieve higher recalls but for the expense of precision.

All models were validated by the 10-fold CV approach described in Section 6.2 and 6.5.3. The overall data used for the final validation routine included 391,295 samples. It contained 385,687 healthy and 5,608 defective samples. In order to apply the 10-fold validation on any model, the used data were partitioned into ten different folds in a stratified manner. This gave ten distinct folds of about 38,568 healthy and 560 defective samples each. Nine of these folds were merged, perhaps resampled, and used for training. The single remaining fold served as testing data. Hence, all performance metrics are based on testing sets of about 39,128 samples.

The following paragraphs show an inspection of the model of the highest achieved F1-score and the model of the highest achieved F2-score. The model of the highest F1-score used the `GPV_RFE` feature subset, no resampling, and the RF algorithm. It achieved a precision of 0.977432, a recall of 0.506960, an F1-score of 0.667468, and an F2-score of 0.560897. The formulas that are shown in Equations 6.13 to 6.14 were derived and applied to reconstruct the confusion matrices of Figure 6.10. The model of the highest achieved

F1-scores found 283 (TPR ≈ 50%) of the 560 defects. The model predicted a defective label for 7 healthy functions and did not recognize 277 actual defects. The model predicted a correct healthy label for 38,561 (TNR ≈ 99.9%) out of 38,568 healthy functions in total. The corresponding confusion matrix is shown in Figure 6.10a.

		actual class	
		defective	healthy
predicted class	defective	283	7
	helathy	277	38,561

(a) Confusion matrix of the configuration that achieved the highest F1-score (RF, GPV_RFE and no resampling).

		actual class	
		defective	healthy
predicted class	defective	378	679
	healthy	182	37,889

(b) Confusion matrix of the configuration that achieved the highest F2-score (RF, GPV_RFE and smote-resampling).

Figure 6.10: Confusion matrices of the best-performing configurations.

$$TP = number\ of\ defective\ samples \cdot recall \tag{6.13}$$

$$FN = number\ of\ defective\ samples - TP \tag{6.14}$$

$$precision = \frac{TP}{TP + FP} \Leftrightarrow FP = \frac{-precision \cdot TP + TP}{precision} \tag{6.15}$$

$$TN = number\ of\ healthy\ samples - FP \tag{6.16}$$

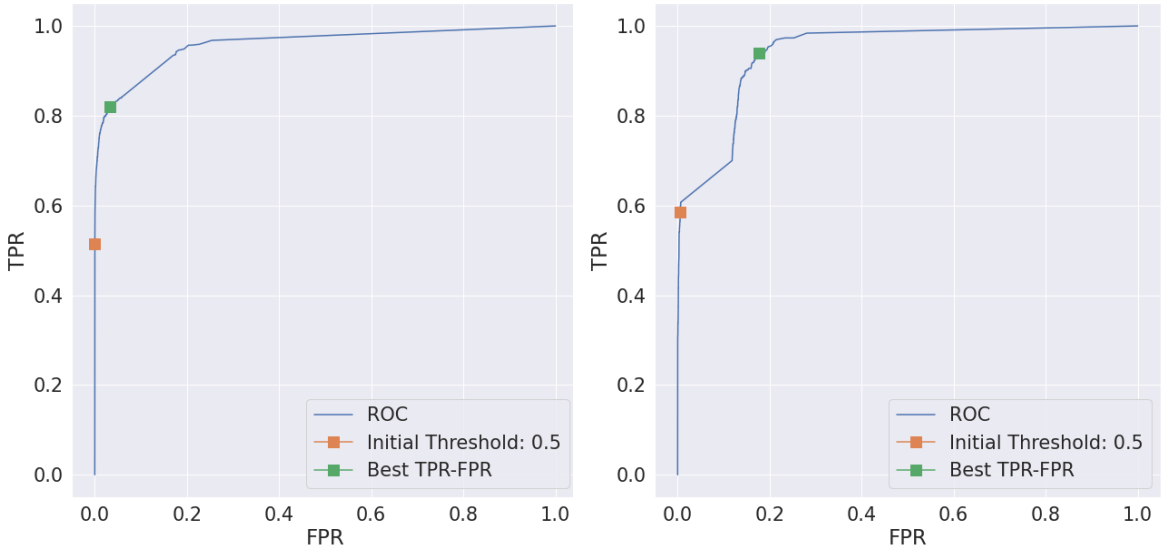
The model performed well in recognizing healthy functions but had problems in identifying defects. It has only found roughly half the number of all defects within the test data. Hence, the matrix suggests that it is easier for the model to recognize healthy functions than to recognize defects. It seems that the model found a reasonable pattern to recognize many defects, but this pattern only covers a part of all defects. Thus, it has very high precision but only a recall of about 0.5. Furthermore, it seems that the model behaves strongly biased towards the majority class. This is probably owed to the lack of applied resampling. Anyway, it achieved the best performance in terms of F1-score as well as the AUC-ROC measure.

The application of a complex smote-resampling approach to even out the classes led to a completely different performance and the model of the highest achieved F2-score. It achieved a precision of 0.357662, a recall of 0.67405, an F1-score of 0.467152, but an F2-score of 0.572497. Figure 6.10b shows the corresponding confusion matrix of this configuration. The model found 378 (TPR ≈ 67%) of the 560 defects, and it miss-classified 679 healthy functions as defects. The model did not recognize 182 defects and classified them as healthy. Finally, the model found 37,889 (TNR ≈ 98%) out of the 38,568 healthy functions. This model is less biased towards the majority class, which led to its significantly higher recall but also to its significantly higher number of FPs. While the first model behaved more reserved at recognizing defects, the second model did not. As a result, it could find way more defects but at the expense of precision. The previous model tended to classify a sample as healthy in doubt and predicted defect labels only if it was very sure about its prediction.

6.6.9 Interpreting the AUC-ROC Measure

This section provides some more interpretation and discussion about the meaning and implications of the AUC-ROC measures. The AUC-ROC metric is a single-value measure based on the ROC-curve, and it corresponds to its computed integral. A performance metric in terms of a plotted curve is neither easy comparable nor interpretable. Hence, the area under the curve is computed and presented as a performance metric.

In order to obtain analyzable plots and measures, fresh models were trained and tested as part of this analysis. Those models were configured as stated in the text. They used one randomly chosen inner 10-fold CV iteration of the original validation routine as the baseline for the applied training and testing data.



(a) ROC-curve of the configuration that achieved the highest F1-score (RF, GPV_RFE and no resampling).

(b) ROC-curve of the configuration that achieved the highest F2-score (RF, GPV_RFE and smote-resampling).

Figure 6.11: ROC-curves of the best-performing configurations.

Figure 6.11a shows the corresponding ROC-curve to the previously analyzed model of the highest achieved F1-score. Figure 6.11b shows the corresponding ROC-curve to the previously analyzed model of the highest achieved F2-score. The plots show that at a certain point, any increase in the TPR is accompanied by an increase in FPR. The closer a curve follows the left and top border, the better is the model's prediction capability. Both curves move quite close to the desired borders. The orange squares illustrate the models' initially chosen classification thresholds of 0.5. The left plot shows that the initial threshold could be easily moved in order to increase the TPR without any major increases in the FPR. The right plot shows that any further movement of the threshold in order to increase the TPR would be accompanied by an increase in FPR. These might be the consequences of learning a tree based on a balanced training dataset or a skewed training dataset. The green square in each plot illustrates the optimal threshold to maximize the performance according to Equation 6.17 and 6.18. It maximizes the TPR while considering the FPR.

$$Optimal\ threshold_{ROC} = \underset{i\ \in\ thresholds}{\operatorname{argmax}} (Sensetivity_i + Specificity_i) \tag{6.17}$$

$$\Leftrightarrow Optimal\ threshold_{ROC} = \underset{i\ \in\ thresholds}{\operatorname{argmax}} (TPR_i - FPR_i) \tag{6.18}$$

The optimal classification threshold of Figure 6.11a is 0.029 and would achieve a precision of 0.2711, a recall of 0.8196, an F1-score of 0.4075, and an F2-score of 0.5835. This would be a significantly decreased F1-score but an increased F2-score. The optimal classification threshold of Figure 6.11b is 0.03 and would achieve a precision of 0.0716, a recall of 0.0939, an F1-score of 0.1331, and an F2-score of 0.2744. This would be a significantly decreased F1- and F2-score. Hence, this threshold optimization method might not be suitable in this case. The problem is that optimization according to TPR and FPR implicitly assumes equal distribution of both classes. A TPR of 0.5 encompasses 280 miss-classifications while an FPR of 0.5 encompasses 19,284 miss-classification. Hence, this kind of optimization is not meaningful because it gives the same weight to both rates, even though the number of affected samples differs significantly.

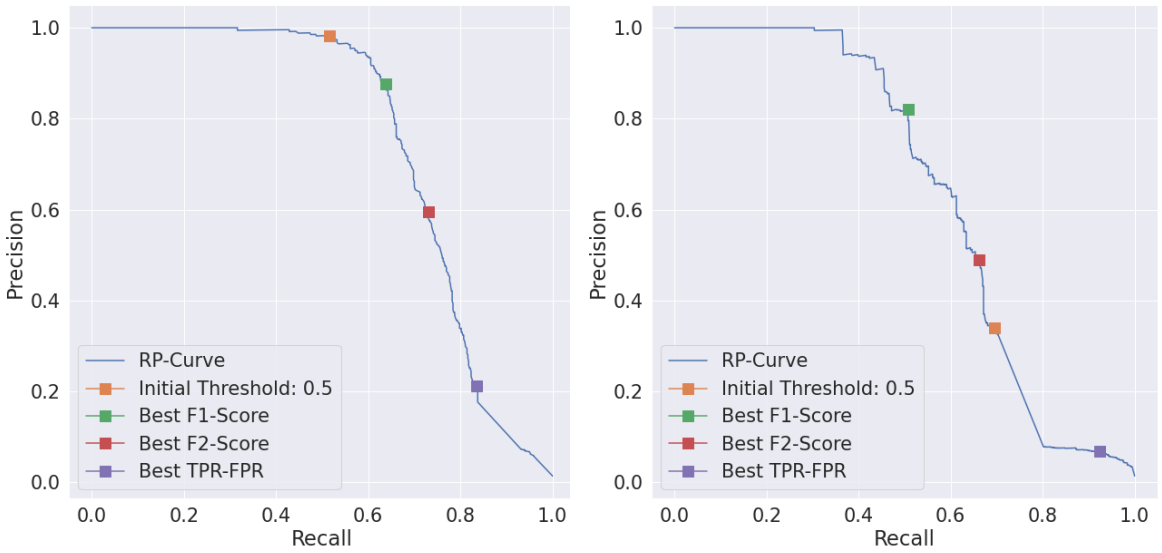
6.6.10 Interpreting the Precision-Recall-Curve

This section provides an additional analysis of the models' precision and recall values. These values can be computed for a variety of different classification thresholds of a single learned classifier. These multiple measures can be plotted to a curve in order to achieve good insight into a model's prediction capability. Like ROC-curves, those plots are also commonly used in order to spot optimization potential by adjustments of the classification threshold. Such a curve is called precision-recall-curve (PR-curve). Because models basically predict probabilities instead of discrete labels, it is possible to compute performance metrics for plenty of thresholds without performing any re-training.

Figure 6.12 shows PR-curves of the best-performing configurations. Both plots show colored squares illustrating the optimized thresholds with respect to a particular metric. The green and the red square show optimized thresholds according to Equation 6.19 and β-values of one and two, respectively.

$$Optimal\ threshold_{PR} = \underset{i\ \in\ thresholds}{\operatorname{argmax}} (1+\beta^2) \cdot \frac{precision_i \cdot recall_i}{(\beta^2 \cdot precision_i) + recall_i} \tag{6.19}$$

The initial threshold of Figure 6.12a is strongly skewed to a high precision, while the initial threshold of Figure 6.12b is a little more skewed to the a high recall. This illustrates the consequences of balancing by resampling approaches as already described in Section 6.6.8. Optimization of the first model could increase its F1-score by 9.54% from 0.6753 to 0.7397. An optimization of this model with respect to the F2-score could increase this measure by 22.09% from 0.5753 to 0.7024. The optimization of the right plot could increase the achieved F1-score by 37.42 % from 0.4562 to 0.6269 or its F2-score by 7.81 % from 0.5737 to 0.6185. After optimization, the left model (without resampling) would outperform the other (smote-resampled) in both F-measures. The purple square in both plots illustrates the performance when optimizing the threshold according to the previously introduced method of Equation 6.17 or 6.18. In both cases, this threshold is strongly skewed to a high recall. These analyses have shown that some sort of threshold optimization can provide further potential to improve a model's testing performance.



(a) PR-curve of the configuration that achieved the highest F1-score (RF, GPV_RFE and no resampling).

(b) PR-curve of the configuration that achieved the highest F2-score (RF, GPV_RFE and smote-resampling).

Figure 6.12: PR-curves of the best-performing configurations.

Findings

12. *The optimization of the classification thresholds provides further potential for improvements in F1- and F2-scores.*

6.7 Experimental Environment

The experiments were performed on multiple machines in parallel. Two virtual machines provided by the University of Hildesheim and several virtual machines based on Google-Cloud[6] were utilized. The two virtual machines provided by the University have 40 and 46 CPU cores and 315GB and 373GB RAM, respectively. Both ran Ubuntu 16.04.6 LTS OS, Python 3.6, and TensorFlow 2.1.

The Google-Cloud-based machines provided individual hardware resources on demand. They were primarily used for the deep learning tasks because of available GPUs. In order to train the majority of NNs, machines of four CPU cores, 24 GB RAM, and an NVIDIA Tesla T4 GPU were used. All machines ran an optimized version of Debian 9 and came with pre-installed Python 3.5 and TensorFlow 2.1. Up to four Google-Cloud-based machines were used in parallel to speed up the entire validation routine.

The next chapter provides a comprehensive description of the second set of experiments aimed to answer [RQ3] and [RQ4]. The chapter is structured like this and provides information on all parts of the experiments.

[6]Google Cloud or Google Cloud Platform is a suite of multiple cloud computing services. Among others, it offers specialized computing engines for machine and deep learning. https://cloud.google.com/

7 Software Metric Subset Combination Experiments

This chapter provides a detailed description of the second set of experiments. The chapter is structured according to the previously introduced refined methodology of Section 6. The first Section 7.1 defines and explains the aimed goal of this set of experiments. The subsequent Section 7.2 describes and introduces the general concept, which was followed to achieve the defined goal. Section 7.3 provides a description of the applied steps of pre-processing. Section 7.4 describes the applied models and their algorithm-specific hyperparameter configurations. Section 7.5 gives some additional details on the exact implementation of the introduced concept, and Section 7.6 provides a comprehensive analysis of the experiments' results in order to address [RQ3] and [RQ4]. The final Section 7.7 provides details about the experimental environment, which was used to conduct the experiments.

7.1 Goal

The second set of experiments aimed to find the most suitable software metric combination for SDP in this case. This set of experiments has validated all possible combinations based on the pre-defined software feature subsets, which were introduced in Section 6.2. A more fine-grained validation of all possible software metric combinations was not possible due to computational infeasibility. Such a grid-based approach to validate all combinations of the 692 software metrics would include 2^{692} combinations. 2^{692} gives approximately 2.05e+208, which would mean about 3.9e+202 (or 3.9e+178 quadrillion) years of computation when considering one minute for training and testing of a complete 10-fold CV run. The most suitable feature subset combinations served as baseline for a more in-depth analysis of the individual importance of contained software metrics. The obtained results from this validation and the subsequent analysis served as the baseline to answer [RQ3] and [RQ4].

7.2 Concept

This section provides a detailed description of the chosen approach to achieve the desired results. This includes explanations of the general decisions. The following paragraphs describe the approach in detail and focus on a certain aspect.

Feature subsets The applied feature subsets were reused from the previous set of experiments. A detailed description and definition were given in Section 6.2. The feature subsets are groups of software metrics defined according to each metric's variability-awareness and scope. In addition to the five originally pre-defined feature subsets, corresponding RFE feature subsets were also included. Furthermore, the `ALL_RFE` feature subset was included as well. The including of the `ALL` feature subset would not be meaningful because it corresponds to the union of all others and would not change when combined with any other feature subset. This gave a final set of the following eleven feature subsets: `LNV`, `LNV_RFE`, `LPV`, `LPV_RFE`, `LCV`, `LCV_RFE`, `GPV`, `GPV_RFE`, `GCV`, `GCV_RFE`, `ALL_RFE`.

Combinations This set of experiments considered any possible feature subset combination based on the previously defined ones. Because some of the software metric groups are subsets of other groups (e.g. `LNV_RFE` $\subset$ `LNV`), a mechanism was developed in order to avoid redundant validations. For instance, the considered eleven feature subsets would give $2^{11} = 2048$ possible combinations. A validation of `LNV x LNV_RFE` is not necessary

because it would give the same feature subset as `LNV`. This mechanism could have reduced the number of combinations from 2048 to 377.

Classification Algorithms RF was selected as the only applied classification algorithm because of its good performances in previous experiments and its good computational efficiency. The RF-based models achieved the best validation results on the most feature subsets and still have shown reasonable runtimes. The high amount of computational costs, emerging by the combination of feature subsets, did not allow to validate more than a single classification algorithm. It would not have been feasible to apply a GB-based algorithm on this high number of experiments.

Hyperparameter Optimization No additional hyperparameter optimization was performed for this set of experiments. The algorithm-specific configuration of the RF algorithm was reused from the previous set of experiments. This configuration was already optimized and achieved reasonable performances on a variety of feature subsets as described in Section 6.6.

Performance Measures All of the previously selected performance metrics were recorded on this set of experiments. This includes the recording of recall, precision, F1-score, and AUC-ROC. The *Maximal F1-score* and the corresponding *maximal F1 precision* and *maximal F1 recall* were recorded additionally. The maximal F1-score is computed as described in Section 6.6.10.

Validation Strategy In order to compute robust and reliable performance metrics, the same stratified 10-fold CV strategy was applied as in the previous set of experiments. This approach was considered due to its superiority over a simple hold-out strategy and was already applied in a variety of other SDP research papers. Additional details were given in Sections 6.2 and 6.5.3.

7.3 Data Pre-Processing

The same dataset as in the previous set of experiments was used. Hence, the data have shown the same issues to address. These issues were the high number of features that are also highly correlated, the imbalanced class distribution, and different feature scales. The reduced RFE feature subsets were considered to address the issue of the high number of features. The imbalance of class distribution was addressed by the selection of RF as the classification algorithm because it is more robust to this imbalance than other classifiers and the application of over-, and smote-resampling. The previous set of experiments has shown that resampling caused the RF algorithm to learn a more balanced and less biased pattern, but learning without resampling still achieved the best validation scores. No general superiority of one resampling method over the other was recognized. As the result, over-, smote-, and no resampling were considered in the validation routine. L2-normalization was applied to all experiments.

7.4 Model Construction

This section provides details about the chosen classification algorithm. This primarily means the description of algorithm-specific hyperparameters. No additional hyperparameter optimization was applied as part of this set of experiments. The optimized configuration from the previous set of experiments was completely reused. The RF algorithm was applied with the following hyperparameters: The number of trees within the ensemble was set to 500. No maximal depth of the trees was defined. The Gini Impurity was used as the splitting criterion. A detailed overview of evaluated parameters as well as explanations of the mentioned parameters was given in Section 6.4.3.

7.5 Implementation

This section provides some details about the implementation of the mechanism used to avoid redundant feature subset validations. Any other components, like the preparation pipeline or the validation routine itself, were reused from the previous set of experiments. Algorithm 4 shows how the unique feature subsets were built.

Algorithm 4 Building of unique feature subset combinations

```
1: featureSubsets ← LNV, LNV_RFE, LPV, LPV_RFE, LCV, LCV_RFE, GPV, GPV_RFE, GCV,
   GCV_RFE, ALL_RFE
2: powerset ← getPowerSet(featureSubsets)
3: uniqueFeatureSubsetsList ← ∅
4:
5: for all set in powerset do
6:     if set is not equal to any set in uniqueFeatureSubsetList then
7:         uniqueFeatureSubsetList.append(set)
8:     end if
9: end for
```

The algorithm works straightforward and starts with generating the *powerset* of the provided $featureSubsets$. This powerset contains all 2048 combinations of the feature subsets including the empty set. The empty set is not considered as meaningful feature subset and is removed immediately. In the next step, an empty list denoted as $uniqueFeatureSubsetsList$ is initialized to store the unique feature subset combinations. Then the algorithm starts to iterate all contained combinations in the *powerset* and continuously tests if the $uniqueFeatureSubsetList$ already contains a similar feature subset. If it does not, the current *set* is appended to the $uniqueFeatureSubsetList$. Once the algorithm was performed successfully, it reduced the number of feature subset combinations from 2048 to 377.

7.6 Analysis

This section provides an analysis of the obtained results from this set of experiments. The first analysis examines all achieved performances of the feature subset combinations and provides some first findings. The subsequent subsection provides a specific analysis in order to rank the combinations. The third subsection provides some more detailed and interpretable insights into the achieved performances, and the final analysis provides the findings with regards to an individual assessment of the software metrics.

7.6.1 General Performance Analysis

This set of experiments validated all possible combinations of the six provided pre-defined feature subsets. This process included 377 combinations after the reduction, which was described in Section 7.5. All combinations were validated on three different resampling

methods and used the 10-fold CV strategy. This ended up with learning and evaluating 11,310 RFs. An overview of the best achieved performances of each feature subset combination is shown in Table 7.1. Only the record of the sampling method, which achieved the highest F1-score, is presented for each feature subset combination. This was done for the sake of clarity. The *S* column indicates the applied resampling method, where an N means *no resampling*, S means *smote-resampling*, and an O means *over-sampling*. The last column, called *n*, indicates the number of contained software metrics in the feature subset. The columns *Prec* and *Rec* show the configurations precision and recall measures. The shown *max_F1* values were computed at runtime and follow the optimization approach described in Section 6.6.10. The table only shows RFE-based feature subset combinations. Previous analyses described in Section 6.6.5 have shown that the RFE feature subsets performed significantly better than their corresponding non-RFE feature subsets in four out of six cases. The RFE feature subsets performed similarly to the original feature subsets in the remaining two cases. Hence, discarding the non-RFE feature subset combinations from Table 7.1 maintained the clarity of the table without cherry-picking or discarding any findings. The entire record of results is provided digitally. The table is ordered descending by the achieved initial F1-scores.

Table 7.1: Achieved validation scores of all RFE-based feature subset combinations of the second set of experiments.

Subset	*S*	Prec ∅	Rec ∅	F1 ∅	max_F1 ∅	AUC -ROC ∅	*n*
LPV_RFE x GPV_RFE	N	0.975	0.507	0.666	0.728	0.960	22
GPV_RFE	N	0.975	0.506	0.666	0.728	0.961	20
LPV_RFE x GPV_RFE x ALL_RFE	N	0.990	0.479	0.646	0.740	0.967	25
GPV_RFE x ALL_RFE	N	0.990	0.477	0.643	0.740	0.968	23
LNV_RFE x LPV_RFE x GPV_RFE	N	0.984	0.472	0.638	0.733	0.966	25
LNV_RFE x GPV_RFE x ALL_RFE	N	0.991	0.465	0.633	0.744	0.968	26
LNV_RFE x GPV_RFE	N	0.985	0.466	0.632	0.731	0.966	23
LNV_RFE x LPV_RFE x GPV_RFE x ALL_RFE	N	0.992	0.461	0.629	0.741	0.969	28
LNV_RFE x GPV_RFE x GCV_RFE	S	0.629	0.624	0.627	0.644	0.953	32
LNV_RFE x LPV_RFE x GPV_RFE x GCV_RFE	S	0.636	0.616	0.626	0.639	0.952	34
GPV_RFE x GCV_RFE	N	0.990	0.451	0.620	0.742	0.968	29
LPV_RFE x GPV_RFE x GCV_RFE	N	0.990	0.450	0.619	0.740	0.968	31
ALL_RFE	N	0.975	0.424	0.591	0.677	0.955	9
LPV_RFE x ALL_RFE	S	0.573	0.609	0.590	0.639	0.936	14
LPV_RFE x LCV_RFE x GPV_RFE x ALL_RFE	S	0.609	0.554	0.580	0.629	0.949	34
LNV_RFE x LCV_RFE x GPV_RFE x ALL_RFE	S	0.590	0.562	0.575	0.630	0.950	35
LCV_RFE x GPV_RFE x ALL_RFE	S	0.582	0.565	0.572	0.634	0.950	32
LNV_RFE x LPV_RFE x ALL_RFE	N	0.991	0.401	0.571	0.695	0.960	17
LNV_RFE x LPV_RFE x LCV_RFE x GPV_RFE x GCV_RFE	S	0.576	0.557	0.566	0.618	0.951	43
LPV_RFE x LCV_RFE x GPV_RFE	S	0.577	0.556	0.566	0.622	0.947	31
LNV_RFE x LCV_RFE x GPV_RFE x GCV_RFE	S	0.568	0.562	0.565	0.624	0.952	41
LCV_RFE x GPV_RFE	S	0.565	0.564	0.564	0.625	0.947	29
LNV_RFE x LCV_RFE x GPV_RFE	S	0.568	0.558	0.562	0.624	0.947	32
LCV_RFE x GPV_RFE x GCV_RFE	S	0.558	0.568	0.562	0.627	0.952	38
LPV_RFE x LCV_RFE x GPV_RFE x GCV_RFE	S	0.567	0.557	0.562	0.621	0.951	40
LNV_RFE x LPV_RFE x LCV_RFE x GPV_RFE x ALL_RFE	S	0.568	0.556	0.561	0.622	0.950	37
LNV_RFE x LPV_RFE x LCV_RFE x GPV_RFE	S	0.567	0.555	0.560	0.619	0.947	34
LPV_RFE x GCV_RFE x ALL_RFE	N	0.989	0.382	0.551	0.683	0.959	20
GCV_RFE x ALL_RFE	S	0.533	0.561	0.546	0.606	0.922	15
LNV_RFE x ALL_RFE	N	0.986	0.376	0.545	0.677	0.955	12

Continued on next page

Subset	S	Prec ∅	Rec ∅	F1 ∅	max_F1 ∅	AUC -ROC ∅	n
LNV_RFE x LPV_RFE x GCV_RFE x ALL_RFE	N	0.991	0.356	0.524	0.675	0.958	23
LNV_RFE x GCV_RFE x ALL_RFE	N	0.985	0.355	0.522	0.662	0.953	18
LPV_RFE x LCV_RFE x ALL_RFE	S	0.438	0.502	0.467	0.543	0.931	23
LPV_RFE x LCV_RFE x GCV_RFE x ALL_RFE	S	0.430	0.511	0.466	0.540	0.933	29
LNV_RFE x LPV_RFE x LCV_RFE x ALL_RFE	S	0.433	0.507	0.466	0.537	0.932	26
LNV_RFE x LPV_RFE x LCV_RFE x GCV_RFE x ALL_RFE	S	0.427	0.507	0.463	0.535	0.932	32
LCV_RFE x GCV_RFE x ALL_RFE	S	0.378	0.486	0.425	0.507	0.924	24
LNV_RFE x LCV_RFE x GCV_RFE x ALL_RFE	S	0.372	0.489	0.422	0.508	0.925	27
LNV_RFE x LCV_RFE x ALL_RFE	S	0.366	0.483	0.416	0.507	0.922	21
LCV_RFE x ALL_RFE	S	0.348	0.485	0.405	0.503	0.923	18
LNV_RFE x LPV_RFE x GCV_RFE	N	0.981	0.248	0.395	0.499	0.857	17
LNV_RFE x GCV_RFE	S	0.508	0.315	0.389	0.404	0.738	12
LPV_RFE x GCV_RFE	N	0.987	0.239	0.384	0.495	0.857	14
GCV_RFE	S	0.426	0.311	0.359	0.386	0.733	9
LNV_RFE x LPV_RFE x LCV_RFE x GCV_RFE	N	0.965	0.219	0.356	0.464	0.851	26
LPV_RFE x LCV_RFE x GCV_RFE	N	0.973	0.215	0.352	0.464	0.852	23
LNV_RFE x LCV_RFE x GCV_RFE	N	0.959	0.197	0.326	0.428	0.834	21
LNV_RFE x LPV_RFE	N	0.972	0.191	0.319	0.413	0.807	8
LNV_RFE x LPV_RFE x LCV_RFE	N	0.936	0.192	0.318	0.398	0.817	17
LCV_RFE x GCV_RFE	N	0.955	0.191	0.318	0.421	0.832	18
LPV_RFE x LCV_RFE	N	0.915	0.176	0.295	0.372	0.807	14
LPV_RFE	N	0.889	0.175	0.292	0.336	0.745	5
LNV_RFE x LCV_RFE	N	0.912	0.156	0.267	0.335	0.788	12
LCV_RFE	N	0.853	0.141	0.242	0.304	0.765	9
LNV_RFE	N	0.942	0.133	0.232	0.314	0.745	3

The results that are shown in Table 7.1 corroborate the ranking of non-combined feature subsets as described in Section 6.6.1. The feature subset that achieved the highest F1-scores if applied solely is again `GPV_RFE`. The second-best is `ALL_RFE`, followed by `GCV_RFE`, and the worst F1-scores were achieved by `LPV_RFE`, `LCV_RFE`, and `LNV_RFE`. The slight superiority in the overall ranking of `LPV_RFE x GPV_RFE` over `GPV_RFE` can be considered as noise (`LPV_RFE x GPV_RFE` vs. `GPV_RFE`; H_0 not rejected with p-value ≈ 0.9502). Hence, the results of the table suggest that the application of the `GPV_RFE` leads to the overall best initial F1-scores (without any subsequent classification threshold optimization). It statistically surpassed the next best feature subset combination of `LPV_RFE x GPV_RFE x ALL_RFE` (`GPV_RFE` vs. `LPV_RFE x GPV_RFE x ALL_RFE`; H_0 rejected with p-value $\approx$ 4.97e-03).

Besides the initially achieved F1-scores, the table shows maximal achievable F1-scores once the classification thresholds were optimized. While the `GPV_RFE` feature subset achieved higher initial F1-scores than `LPV_RFE x GPV_RFE x ALL_RFE` and `GPV_RFE x ALL_RFE`, its maximal F1-score is lower. The same applies to AUC-ROC measures. This could mean that additional metrics from the mentioned combinations provided the model with additional capabilities but led to a slight over-fitting, which reduced its initial F1-score. The additional capabilities could be utilized once the classification thresholds were optimized. This acted as a post-training countermeasure against over-fitting. Because the vast majority of examined literature described in Chapter 3 applied and reported achieved results on default thresholds, the initial F1-scores remains the primary performance metric used to assess prediction capability.

All combinations from the top of Table 7.1 to the `ALL_RFE` feature subset solely, utilized the `GPV_RFE` feature subset. This strongly suggests that these metrics are quite essential

to achieve the best possible performances. This assumption is corroborated by presented initial F1-scores, maximal F1-scores, and AUC-ROC measures. No combination achieved an initial F1-scores ≥ 0.6 or a maximal F1-score ≥ 0.7 without containing the `GPV_RFE` features. On the other side, there is no feature subset combination containing the `GPV_RFE` feature subset and achieved an initial F1-score ≤ 0.56 or a maximal F1-score ≤ 0.6. These observations also corroborate the strong assumption of the best prediction capabilities by `GPV_RFE`.

There are many combinations that do not contain the `GPV_RFE` feature subset but the `ALL_RFE` feature subset and achieved reasonable results. Table 6.7 in the previous Section 6.5.2 shows all contained software metrics in the `ALL_RFE` feature subset. It contains six out of 20 software metrics of the `GPV_RFE` feature subset. These include variants of `EXTERNAL_WITH_BUILD_VARS` and `ALL_WITH_BUILD_VARS`. Hence, the `GPV_RFE` only differs from `ALL_RFE` in that it contains some further variants of `EXTERNAL_WITH_BUILD_VARS` and `ALL_WITH_BUILD_VARS` as well as `EV VP Fan-In(global)`, `EV VP Fan-In(local)` and `EV VP Fan-Out(global)`. This suggests that the probably second-best prediction capability of `ALL_RFE` could be primarily owed to the contained global-scoped pure-variability-aware metrics. These global-scoped pure-variability-aware metrics within the `ALL_RFE` feature subset were selected by the RFE algorithm. Hence, the RFE algorithm determined them as one of the most meaningful metrics overall. This also corroborates the strong assumption about the best prediction capabilities of the `GPV_RFE` features. The remaining features within the `ALL_RFE` feature subset are three metrics from the `GCV_RFE` feature subset. Hence, the `ALL_RFE` feature subset contains the perhaps most suitable metrics of `GPV_RFE` and `GCV_RFE`. This could easily lead to the assumption that `ALL_RFE` should be superior to `GPV_RFE` and `GCV_RFE`, but the results show the opposite. `GPV_RFE` is superior to `ALL_RFE` (H_0 rejected with p-value $\approx$ 2.11e-09). This demonstrates nicely that the use of the RFE algorithm as a feature selector does not find the globally optimal features. Still, it only optimizes iteratively in a local (i.e. greedy) fashion. The lowest F1-scores were achieved by combinations without either `ALL_RFE` and `GPV_RFE`.

The best local-scoped feature subset combination was `LNV_RFE x LPV_RFE`. It achieved an initial F1-score of 0.318921 and a maximal F1-score of 0.412602. This is still significantly worse than any global-scoped feature subset solely (`LNV_RFE x LPV_RFE` vs. `GPV_RFE`; H_0 rejected with p-value $\approx$ 1.71e-15) (`LNV_RFE x LPV_RFE` vs. `GCV_RFE`; H_0 rejected with p-value $\approx$ 6.18e-03).

Findings

13. *No arbitrary local-scoped feature subset combination achieved meaningful results (initial F1-score $\geq$ 0.4).*
14. *The GPV_RFE feature subset surpassed all feature subset combinations (according to the achieved initial F1-scores) significantly.*
15. *The RFE algorithm has selected a mix of global-scoped pure- and combined-variability-aware code metrics. It did not select any local-scoped or non-variability-aware metric.*

7.6.2 Feature Subset Combination Ranking

This section presents the findings of additional analyses meant to rank the feature subset combinations and describes why such a ranking was finally considered as not meaningful. Tables 7.2 and 7.3 provide simplified overviews of all RFE-based combinations without `LPV_RFE`, `LCV_RFE`, and `ALL_RFE`. These subsets were discarded because pure-variability-awareness and combined-variability-awareness are still represented by their superior global-scoped versions, and the `ALL_RFE` feature subset does not represent a specific type of software metrics. Table 7.2 shows the best records according to achieved initial F1-scores for each feature subset and is ordered by F1-scores descending. Table 7.3 shows the best records according to achieved maximal F1-scores for each feature subset and is ordered by maximal F1-scores descending.

Table 7.2: Best achieved validation scores each RFE-based feature subset combination according to achieved initial F1-scores and without ALL_RFE, LPV_RFE and LCV_RFE.

Subset	S	Precision ∅	Recall ∅	F1 ∅	max_F1 ∅	AUC -ROC ∅	n
GPV_RFE	N	0.975	0.506	0.666	0.728	0.961	20
LNV_RFE x GPV_RFE	N	0.985	0.466	0.632	0.731	0.966	23
LNV_RFE x GPV_RFE x GCV_RFE	S	0.629	0.624	0.627	0.644	0.953	32
GPV_RFE x GCV_RFE	N	0.990	0.451	0.620	0.742	0.968	29
LNV_RFE x GCV_RFE	S	0.508	0.315	0.389	0.404	0.738	12
GCV_RFE	S	0.426	0.311	0.359	0.386	0.733	9
LNV_RFE	N	0.942	0.133	0.232	0.314	0.745	3

Table 7.3: Best achieved validation scores each RFE-based feature subset combination according to achieved optimized F1-scores and without ALL_RFE, LPV_RFE and LCV_RFE.

Subset	S	Precision ∅	Recall ∅	F1 ∅	max_F1 ∅	AUC -ROC ∅	n
GPV_RFE x GCV_RFE	N	0.990	0.451	0.620	0.742	0.968	29
LNV_RFE x GPV_RFE x GCV_RFE	N	0.991	0.438	0.607	0.737	0.968	32
LNV_RFE x GPV_RFE	N	0.985	0.466	0.632	0.731	0.966	23
GPV_RFE	N	0.975	0.506	0.666	0.728	0.961	20
LNV_RFE x GCV_RFE	N	0.975	0.225	0.365	0.453	0.840	12
GCV_RFE	N	0.976	0.214	0.351	0.439	0.834	9
LNV_RFE	N	0.942	0.133	0.232	0.314	0.745	3

Table 7.2 corroborates all previous assumptions about the superiority of `GPV_RFE`, but Table 7.3 does not. Even though the `GPV_RFE` feature subset achieved the highest initial F1-score, it did not achieve the highest maximal F1-score. While it is listed as the top subset in Table 7.2, it is listed as the fourth-best subset in Table 7.3. The order of Table 7.3 would remain the same if the AUC-ROC measure would be used as the index. This means that depending on the applied performance metrics, the best feature subset combination differs. Anyway, both tables confirm the significant superiority of combinations containing the `GPV_RFE` feature subset and the inferiority of `GCV_RFE` and `LNV_RFE`. This is highlighted by a sharp drop in any performance metric once the `GPV_RFE` feature subset is entirely excluded from a combination (i.e. the three bottom records of both tables). A generally valid ranking of combinations is not meaningful due to the dependence of the

applied performance metric. The tables suggest that more comprehensive feature subset combinations lead to reduced initial F1-scores but an increased maximal F1-score and AUC-ROC measure. This hypothesis could not be validated by a complete analysis of all results.

While `GPV_RFE` and `LPV_RFE x GPV_RFE` achieved the highest initial F1-score of the shown combinations, `GPV_RFE x GCV_RFE` significantly outperformed `GPV_RFE` according to their maximal F1-scores (H_0 rejected with p-value $\approx$ 0.037) and AUC-ROC measures (H_0 rejected with p-value $\approx$ 2.81e-04). `LNV_RFE x GPV_RFE x GCV_RFE` significantly outperformed `GPV_RFE` according to their AUC-ROC measures (H_0 rejected with p-value $\approx$ 1.73e-03) but not according to maximal F1-scores (H_0 not rejected with p-value $\approx$ 0.188). `GPV_RFE x GCV_RFE` and `LNV_RFE x GPV_RFE x GCV_RFE` achieved similar results in maximal F1-score (H_0 not rejected with p-value $\approx$ 0.306) and AUC-ROC measure (H_0 not rejected with p-value $\approx$ 0.834).

`LNV_RFE x GPV_RFE x ALL_RFE` achieved the highest maximal F1-score overall with a value of 0.744235. `GPV x LNV_RFE x ALL_RFE` achieved the highest AUC-ROC measure overall with a value of 0.968735. `LNV_RFE x LPV_RFE x GPV_RFE x ALL_RFE` achieved the second-best AUC-ROC measure overall and the best AUC-ROC measure over all RFE-based feature subset combinations with a value of 0.968589. The complete record of all results is provided digitally.

Findings

16. *LPV_RFE x GPV_RFE and GPV_RFE achieved the highest initial F1-scores overall.*
17. *LNV_RFE x GPV_RFE x ALL_RFE achieved the highest maximal F1-score overall.*
18. *LNV_RFE x LPV_RFE x GPV_RFE x ALL_RFE achieved the highest AUC-ROC measure over all RFE-based feature subsets.*

7.6.3 Interpretation of best-performing Configuration

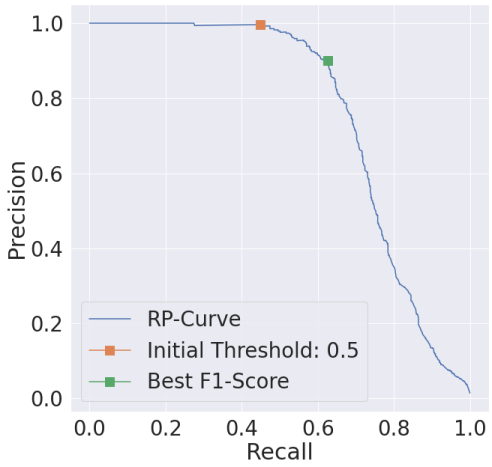
This section provides a brief analysis of the model's performance that achieved the highest maximal F1-score. The goal of this analysis is to provide a better understanding of the achieved prediction capability as it was introduced in Section 6.6.8 of the previous set of experiments.

As previously mentioned in Section 7.6.1, the highest maximal F1-score overall was achieved by `LNV_RFE x GPV_RFE x ALL_RFE`, and hence this model was selected for some further analyses. Figure 7.1a shows the reconstructed confusion matrix of this model. The reconstruction was performed according to the explanation of Section 6.6.8 and the formulas of Equations 6.13, 6.14, 6.15, and 6.16. This matrix was computed based on the 10-fold inner iteration that achieved the median maximal F1-score of this configuration. It achieved a maximal F1-score of 0.741803 in iteration two of the 10-fold CV run. The maximal F1-score was achieved on a classification threshold of 0.17. The corresponding precision is 0.872289, and the recall is 0.645276.

The confusion matrix shows significantly improved predictions compared to both confusion matrices of Figure 6.10 in Section 6.6.8. Figure 7.1b shows the PR-curve of this model and illustrates that it is capable of achieving decent precision and recall values on certain thresholds. The green square on the blue line indicates the maximal F1-score, and the

		actual class	
		defective	healthy
predicted class	defective	361	52
	helathy	199	38,516

(a) Confusion matrix.



(b) PR-curve.

Figure 7.1: Confusion matrix and PR-Curve of the configuration that achieved the highest maximal F1-score (RF, LNV_RFE x GPV_RFE x ALL_RFE and no resampling).

orange square indicates the initially achieved F1-score at a threshold of 0.5. The squares on the curve show again that omitting a resampling method induces a strong bias towards healthy labels, but a post-training adjustment of the classification threshold can lead to more balanced predictions. The model found 361 out of 560 defects. It predicted a defective label for 52 healthy functions and did not recognize 199 actual defects. The model predicted a correct healthy label for 38,516 healthy functions in total.

7.6.4 Analysis of individual Software Metrics' Importances

This section provides an analysis of individual feature importances. Decision trees and other tree-based algorithms are considered to be well interpretable. Machine learning libraries provide functions to obtain individual feature importances of learned tree-based models based on a feature's impurity decrease as described in Section 6.5.2. Feature importances were stored over all sets of experiments of any single tree-based model. While these feature importance values have served as baseline for the RFE algorithm, which was used to successively remove the least important features, it is not hesitatingly useful for a precise assessment of importances or rankings of features. The reason for that is the feature correlation bias due to the high multicollinearity among the data, which was already described in Section 6.3.1.2. The RFE algorithm has already made use of this kind of computed feature importances, but it only removed the single feature of least importance on each iteration. It did not utilize more sophisticated information from that. Furthermore, any decision made by the algorithm was verified on a subsequent validation on the outer validation dataset. Hence, the RFE algorithm utilized these feature importances despite the present feature correlation bias. The fact that it only extracted a quite simple information (i.e. what was the least important feature in this RFE iteration) and the continuous validation have made the algorithm to perform quite robust though the presence of feature correlation bias. Hence, the extracted feature importances were sufficiently robust to be used in the RFE algorithm but not to be used for precise and individual importance assessments of contained features within the final RFE feature subsets.

The GPV_RFE feature subset contains 20 code metrics that stem from only two different code metric families. These are several types of ALL_WITH_BUILD_VARS Vars per Function and EXTERNAL_WITH_BUILD_VARS Vars per Function as well as EV VP Fan-In(global), EV VP Fan-In(local), and EV VP Fan-Out(global). The first two metrics belong to the *Software-Feature per Function* metric family described in Section 5.1.1.2 and the second three metrics belong to the *Recursive Fan-In/-Out* metric family described in Section 5.1.1.8. Code metric variants typically share the same method of measurement, and hence they are correlated to a certain degree. This prevents from a reliable assessment of the metrics' importance within one code metric family, and it does not allow a reliable assessment of the importance of the different variability metrics either. The feature correlation bias falsifies the importance values too much.

Even though a detailed importance assessment within the feature subsets was not possible without further ado, the RFE algorithm has already provided a variety of information. The following paragraphs will take up the results and findings of Sections 6.5.2 and 6.6.5 and provide some additional analyses aimed to get some insights into the suitability of individual code metrics. The complete lists of contained code metrics within the RFE feature subsets were shown in Tables 6.2, 6.3, 6.4, 6.5, 6.6, and 6.7.

The LNV feature subset originally contained 15 different code metrics of five different code metric families. After the RFE reduction, the resulting LNV_RFE feature subset only contained three code metrics of one single code metric family. The RFE algorithm and the subsequent analysis of the results yielded EV Classical Fan-Out(global), EV Classical Fan-In(global), and EV Classical Fan-In(local) as the most meaningful metrics to be used in the RFE feature subset.

The LPV feature subsets originally contained 20 metrics of seven different code metric families in total. After the RFE reduction, the resulting LPV_RFE feature subset only contained five code metrics of two different metric families. The RFE algorithm yielded all four variants of EV VP Fan-In/-Out (local and global) and ALL_WITH_BUILD_VARS Vars per Function as the most meaningful metrics. Hence, the algorithm found all variants of the recursive VP Fan-In/-Out metrics more important than the non-recursive variants. Furthermore, the algorithm kept all available variants of this specific metric family. This happened only in this feature subset.

The LCV feature subset originally contained 28 metrics of six different code metric families in total. After the RFE reduction, the resulting LCV_RFE feature subset only contained nine code metrics of three code metric families. The RFE algorithm yielded recursive variants of DC Fan-In/-Out (local and global) as part of the most meaningful features. Besides six recursive DC Fan-In/-Out variants, the subset contains a single non-recursive DC Fan-Out variant as well as Combined ND_Avg as single nesting depth metric. The algorithm kept specific variants of the recursive DC Fan-In, DC Fan-Out locally and globally but due to the feature correlation bias, the exact selection of the variants (i.e. EV DC Fan-In (global) or EV DC Fan-In (global x No Stubs), etc.) should not be considered as reliable.

The GPV feature subset originally contained 200 code metrics of seven code metric families in total. These metrics include a variety of variability-aware code metrics, which were weighted by 19 different variability metrics. The RFE algorithm yielded a feature subset of 20 metrics of two different families. The metrics contain three out of four recursive VP Fan-In and Fan-Out variants. Only EV VP Fan-Out (global) was not kept by the algorithm. Furthermore, multiple weighted variants of the ALL_WITH_BUILD_VARS Vars per Function and EXTERNAL_WITH_BUILD_VARS Vars per Function were kept. The algorithm's decision about which weighted variants too keep should not be considered as

reliable due to the feature correlation bias. These metrics have led to one of the best prediction performances overall. The LPV feature subset had all these recursive variants of VP Fan-In and VP Fan-Out available. In contrast to the GPV feature subset, the LPV feature subset does not contain any weighted variants of the ALL_WITH_BUILD_VARS Vars per Function or EXTERNAL_WITH_BUILD_VARS Vars per Function metrics available but the unweighted ones. The original GPV as well as the GPV_RFE feature subset achieved a much better performance than LPV and LPV_RFE, which was validated in Section 6.6.3. Table 6.11 has shown the significant improvements in prediction capabilities of pure-variability-aware code metrics once the scope was extended to global. This paragraph shows that this improvement is likely owed to the application of variability metrics on ALL_WITH_BUILD_VARS Vars per Function and EXTERNAL_WITH_BUILD_VARS Vars per Function because these are the only difference between LPV_RFE and GPV_RFE. Furthermore, the algorithm preferred the metric variants that also considered variability variables during the build process (i.e. _WITH_BUILD_VARS) over the others in all cases.

The GCV feature subset originally contained 442 metrics of six code metric families in total. The metrics also include a variety of variability-aware code metrics, which were weighted by 19 different variability metrics. The RFE algorithm yielded a feature subset of nine metrics of a single code metric family. The feature subset only contains eight weighted variants of EV DC Fan-In(local) and a single weighted variant of EV DC Fan-Out(global). The GCV_RFE feature subset was also considered as superior over the LCV and LCV_RFE feature subset as described by Section 6.6.3. Still, the improvement was not that large as the one of pure-variability-aware code metrics. Nevertheless, the weighting of the recursive DC Fan-In and DC Fan-Out metrics improved the achieved F1-scores significantly.

The ALL feature subset originally contained 692 metrics of all nine code metrics families. The RFE algorithm yielded a feature subset of only nine metrics of two metric families. The feature subset contains six metrics that also appear in the GPV_RFE feature subset. These are weighted variants of ALL_WITH_BUILD_VARS Vars per Function and EXTERNAL_WITH_BUILD_VARS Vars per Function. The remaining three metrics of ALL_RFE also appear in the GCV_RFE feature subset and are weighted variants of EV DC Fan-In(local). All metrics contained in the ALL_RFE feature subset are also contained in another RFE feature subset, which corroborates the findings of the RFE algorithm.

Further analyses and the mentioning of the variability metrics which were kept by the RFE algorithm were omitted on purpose. The high correlation among different variability metrics on a certain code metric variant made this kind of decision insufficiently reliable. The exact weighted metric variants that were kept by the RFE algorithm can be obtained from Tables 6.2, 6.3, 6.4, 6.5, 6.6, and 6.7. The RFE algorithm chose the POSITIVE_SIZES variability metric in six out of seven cases where this metric was available. It also selected TOTAL_SIZES as variability metric in five out of seven cases. These variability metrics have the highest occurrence, but this still cannot be considered as solid findings due to the high feature correlation bias.

Findings

19. *Recursive variants of all Fan-In/-Out metrics were generally preferred over the non-recursive variants by the RFE algorithm.*

20. *The weighting by additional variability metrics on Software-Features per Function metrics improved the entire prediction capability significantly.*

21. *The Software-Features per Function metrics that additionally consider used variability variables of the build process, were generally preferred by the RFE algorithm.*
22. *Weighted variants of EXTERNAL_WITH_BUILD_VARS Vars per Function and ALL_WITH_BUILD_VARS Vars per Function have strongly contributed to the overall best achieved prediction performances.*
23. *Weighted variants of code metrics were generally preferred over non-weighted ones by the RFE algorithm.*
24. *The RFE algorithm has selected only global-scoped pure- and combined-variability-aware code metrics from ALL feature subset. It did not select any local-scoped or non-variability-aware metric.*

7.7 Experimental Environment

This set of experiments was exclusively performed on the two virtual machines provided by the University of Hildesheim. No other machines were employed. Because only RF-based models were trained and tested in this set of experiments, the virtual machines provided optimal conditions with their high number of CPU cores. A more detailed description of both machines was given in Section 6.7.

The next Chapter provides the final discussion of the results and findings. This chapter is meant to put all findings in context to the defined research questions. Furthermore, the discussion aligns the findings of this study with the findings of the reviewed literature and describes the limitations and potential threats to validity.

8 Discussion

This chapter provides a final discussion about the findings of this thesis and gives clear answers to the defined research questions. Furthermore, the findings of this thesis are aligned with the results of other reviewed literature. The final parts of this chapter describe the limitations and threats to the validity of the findings.

8.1 Answering the Research Questions

The main goal of this thesis was to gain information about the suitability of different types of SCMs for the application of SDP. The ulterior motive behind SDP is to support developers in producing bug-free code and reduce the number of defects in a software project. Software defects cause costs due to expensive troubleshooting. Consequently, this amount of costs can be reduced if the number of defects is reduced.

Previous research has produced a large variety of novel SCMs [ESKS19]. A subset of these metrics was applied and collected from the Linux kernel and finally provided as data to this work. This dataset was the baseline for the analysis of the metrics' suitability. Four research questions were defined and investigated in order to assess and understand the suitability of the provided data. The defined research questions of Section 1.2 are finally addressed and answered in this section. These answers are based on the elaborated and summarized findings of prior analysis sections of Chapters 6 and 7. The analysis sections already presented and interpreted the results of the conducted experiments precisely. Each subsection of the analyses was summarized by a listing of the concrete findings at the end of each subsection. At this point, the listed findings are used to finally address the research questions and provide clear answers.

[RQ1] *Does the utilization of variability-aware code metrics achieve significantly better results than non-variability-aware code metrics?*

[RQ1] was addressed by the first set of experiments. Section 6.6.2 presented a clear comparison of the achieved performances of non-variability-aware and variability-aware metrics. The section clearly found that non-variability-aware code metrics did not outperform any other feature subset, whereas three out of four variability-aware feature subsets outperformed non-variability-aware code metrics significantly. Furthermore, Section 6.6.2 has shown that pure-variability-aware code metrics achieved by far the best results overall.

The ulterior motive of [RQ1] was to investigate the assumption that implemented code variability could increase the complexity of code, and hence it could increase the difficulty of creating bug-free code. Thus, a lot of metrics were defined to measure the code variability because this variability might be a useful indicator to recognize defects. The results of conducted experiments have confirmed this assumption, and show that measures of code variability are very suitable as features used to predict software defects. The use of traditional SCMs, as in the local-scoped non-variability-aware feature subset, was not able to achieve reasonable performances if applied solely.

[RQ2] *Does the weighting of code metrics by additional variability metrics achieve significantly better results than non-weighted code metrics?*

[RQ2] was also addressed by the first set of experiments. Section 6.6.3 presented a clear comparison of the achieved performances of local-scoped (i.e. no additional weighting by variability metrics) and global-scoped (i.e. additional weighting by variability metrics) metrics. The results show that the utilization of variability metrics improved the performances of all variability-aware metric subsets significantly. Furthermore, the applied

feature selection algorithm generally preferred the weighted code metrics over the non-weighted metrics.

The ulterior motive of [RQ2] was to investigate the assumption that an additional weighting of code metrics by variability metrics could provide a more accurate approximation of the code's complexity. The results of the conducted experiments have confirmed this assumption. The additional weighting provided better results in SDP in all cases. The weighted code metrics are considered as the best suitable features in this SDP study. The use of unweighted variability-aware code metrics did not achieve significantly better results than non-variability-aware metrics. Only the weighting by variability metrics made them superior over traditional SCMs.

[RQ3] *Which software metrics are most suitable for software defect prediction?*

[RQ3] was partially addressed by the feature selection algorithm, the first and the second set of experiments. The question requires concrete information regarding the suitability of specific code metrics. This is not trivial to answer because the question aims for univariate information and methods of machine learning typically learn multivariate patterns. Hence, a univariate evaluation of input features becomes difficult in most cases. Even though tree-based algorithms provide techniques to extract precise feature importances, the strong presence of multicollinearity among the data did not allow a sufficiently reliable extraction of them. Instead, Section 7.6.4 provided an analysis aimed for such an individual feature evaluation based on the results of the applied feature selection algorithm and other obtained validation scores. This analysis found that especially recursive variants of conditional Fan-In/-Out (i.e. `EV VP Fan-In/-Out`) and Degree Centrality Fan-In/-Out (i.e. `EV DC Fan-In/-Out`) as well as weighted variants of Software-Features per Function metrics considering variability variables at the build process (i.e. `ALL_WITH_BUILD_VARS Vars per Function` and `EXTERNAL_WITH_BUILD_VARS Vars per Function`) primarily contributed to the best achieved prediction capabilities. The non-weighted variants of variability-aware code metrics did not provide sufficient information to achieve reasonable results in conducted experiments. Traditional non-variability-aware SCMs like LoC or McCabe's Cyclomatic Complexity did not provide sufficient information to achieve reasonable results either.

[RQ4] *What is the best-performing configuration in this software defect prediction task?*

[RQ4] aimed for the most suitable model configuration of this SDP task. The complete configuration of a model includes multiple hyperparameters. These are algorithm-specific hyperparameters as well as general hyperparameters. The most suitable general hyperparameters that achieved the best results are the use of RF as the classification algorithm, the use of L2-normalization, and no use of prior resampling methods on the training data. These settings turned out to perform well in the first set of experiments, and thus they were also applied on the second set of experiments. The detailed algorithm-specific hyperparameters were optimized and precisely explained in Section 6.4.3. Another important hyperparameter is the selection of the applied features to the classification algorithm. The second set of experiments aimed to find the most suitable feature subset combination overall. The analysis of these experiments corroborated that the metrics of `GPV_RFE` are essential to achieve the best possible performances. Further combinations of these metrics with local-scoped non-variability-aware metrics or global-scoped combined-variability-aware code metrics could still improve the prediction capabilities a bit. The very detailed analysis of results is given in Sections 7.6.1 and 7.6.2.

8.2 Aligning the results with reviewed literature

This section aligns and compares the results of this work to the findings and claims of the previously reviewed literature, which was presented in Chapter 3. Lessmann et al. found in their early research about SDP that RFs and NNs performed best [LBMP08]. The results of this study match their findings partially. While RFs also performed best in this case, NNs did not achieve any reasonable results. This is a little contrary to the findings of Lessmann et al. In fact, the suitability of classification algorithms is difficult to generalize. The used datasets vary much, although they are on a similar domain of SDP.

Furthermore, RFs are a very common choice for real tabular datasets because they are quite robust to over-fitting and not that hard to tune. NNs are more like the opposite of this. They are quite prone to over-fitting if no countermeasures are considered, and they are hard and costly to tune. Hence, especially NNs are strongly dependent on the spent effort in optimization and the condition of provided data. A generalization of the suitability of NNs in a certain case of tabular data, is not meaningful. Hall et al. found that especially simple models like NB and Linear Regression achieved good performances. The results of this thesis contradict this claim entirely. Simple models like NB achieved no adequate performances at all. This fact is probably also owed to the condition of the provided dataset. Ultimately, it is not meaningful to recommend a certain algorithm based on only the domain of a classification task. Algorithms and steps of pre-processing always depend on provided data. Even though the results of the research from Giger et al. [GDPG12] were criticized and their finding of the superiority of process metrics over product metrics were revised by Pascarella et al. [PPB18], both papers reported that the use of only SCMs is not suitable to perform adequate SDP. This finding is partially in line with the results of this thesis because the used SCMs in their research are similar to the applied local-scoped non-variability-aware code metrics in this study. The analyses of Sections 6.6.2 and 7.6.1 have shown exactly this. The application of only local-scoped non-variability-aware code metrics is not suitable to perform adequate SDP. On the other side, the results of this thesis contradict to the general claim that SCMs are not suitable for SDP because any metric of this study is an SCM. But a variety of the used metrics of this study are variability-aware and have an extended scope due to the additional weighting by variability metrics. In fact, all the used metrics are static ones, and some of them achieved reasonable results, presumed more than only local-scoped non-variability-aware code metrics were applied. Any comparisons of reported performances from other research and the results of this thesis are not considered as meaningful. The datasets differ too much, and hence achieved performances are not reliably comparable.

8.3 Limitations

This section describes the limitations of this study. These limitations could not be covered at this point within this study. Most of them will be taken up again in the next chapter's outlook section.

Though the thesis answered the defined research questions and provided meaningful information and results to the research of SDP, there are several limitations of this study. The generalizability of the results is limited by the fact that most variability-aware code metrics used in this study are based on analyzed CPP-code. Hence, most of these metrics are only available for C-based software projects, which prevents to test this approach on non-C-based software projects. Most studies of the reviewed literature performed SDP on Java-based projects.

It was not possible to compare the findings and results of this thesis with other studies on the same software project. Such a comparison would require other method-level SDP experiments applied to the Linux kernel. The found and reviewed literature included only a single study that performed SDP on the Linux kernel but with a too different purpose, and hence it was not comparable.

The study explained and analyzed all conducted experiments, and the obtained results, very comprehensively. Due to that, it was possible to provide several findings and draw conclusions about the suitability of certain types of metrics. In addition to that, this study has found some information about the suitability of specific software metrics. A detailed assessment of all metrics was not possible because of the strong presence of multicollinearity and the resulting feature correlation bias. The interpretability of sophisticated classification algorithms is an ongoing research topic called *interpretable machine learning* and is not trivial. A clear interpretation of machine learning models is a challenging task even on simple datasets but rather more on complex ones. The same applies to the assessment of variability metrics. The analysis sections of the experiments were designed reserved regarding the assessment of individual variability metrics purposely. This decision was made to avoid speculation, over-interpretation, or the presentation of unreliable results. The high feature correlation bias did not allow to extract sufficiently reliable information about an individual assessment of variability metrics.

The last limitation of this study is owed to typical computational limitations. A limited amount of time and computational power did not allow a more extensive hyperparameter optimization, especially on very costly algorithms like NNs or GBs. The NN-based experiments were primarily conducted on costly Google-Cloud machines. Hence, the rising costs were a clear limiting factor to this.

8.4 Threats to Validity

This section provides a brief description of potential threats to the validity of this study. These threats are typical statistical risks that could invalidate the findings of this study. The following paragraphs describe those risks and give a short estimate of the severity.

One factor, which was already raised in Section 8.3, is that only a single software project was analyzed. Hence, the findings of this study are not yet cross-project-validated by additional software projects besides the Linux kernel. This does not invalidate the findings, but an additional cross-project-validation could still corroborate them. Nevertheless, the Linux kernel is an extensive software project and is continuously developed by plenty of different software developers. Furthermore, the collected defects of the dataset stem from a period of about five years. These factors shrink the risk of a biased sample to a minimum but do not exclude it.

A substantial part of this thesis was based on the results of the RFE feature selection algorithm. The results of this greedy algorithm were not extensively cross-validated due to limited computational resources. Even though the procedure was repeated multiple times and the suitability of resulting RFE feature subsets were validated by an appropriate hold-out validation strategy, a 10-fold CV approach on every single iteration of the RFE algorithm could still have increased its reliability. This was not considered because such an approach would still have increased the amount of computational costs and durations beyond the limit of available resources.

The applied validation routine of both sets of experiments has performed 10-fold CVs in order to provide reliable results. This approach could have been extended to an n times 10-fold CV strategy. The conducting of multiple repetitions of the entire 10-fold CV run

could still have increased the reliability of results but also the computational costs. These increasing computational costs would have been beyond the limits of this study.

Any important findings were validated by performed Welch-tests. These tests applied a significance threshold α in order to reject the null hypothesis. This threshold was set to 0.05, which is a commonly used and appropriate value. Nevertheless, a lower value of α would still have increased the necessary confidence to reject the null hypothesis.

9 Conclusion

This chapter provides the conclusion of this thesis and is divided into three sections. The first section points out the final fundamental conclusion based on the findings of this thesis. The second section gives some practical recommendations and an outlook of future research based on the conclusion of this work. The last section wraps up the research contribution of this study.

9.1 General Conclusion

The study aimed to evaluate the suitability of a variety of novel variability-aware SCMs. Besides the general analysis of the metrics' suitability, a comparison between the novel variability-aware code metrics and traditional non-variability-aware metrics was of special interest. Based on the conducted experiments and the subsequent analyses of results, it can be concluded that especially global-scoped pure-variability-aware code metrics are superior to any local-scoped or non-variability-aware code metrics. The extension of the variability-aware metrics' scope by an additional weighting via variability metrics improved the expressiveness regarding software defects significantly. The measures of implemented code variability by pure-variability-aware code metrics achieved constantly best performances and were a substantial component to achieve reasonable prediction capabilities. A few individual global-scoped variability-aware code metrics turned out to be most useful in this SDP study. The results have shown the usefulness and suitability of variability-aware SCMs for method-level SDP. The addition of metrics to measure code variability provides the potential to set new benchmarks in SDP because the approximated complexity due to variability serves as a suitable indicator for defects. Hence, existing approaches of SDP should pay more attention to measures of code variability than on traditional measures of code complexity. The most reasonable classification algorithm was an RF classifier using 500 estimators.

9.2 Recommendation & Outlook

Based on the findings and conclusions of this study, ongoing research in the area of SDP should consider additional measures of code variability in their studies. A variety of papers were reviewed and summarized as part of the related work chapter. The majority of them strived to improve their achieved SDP performances by considering further process metrics, socio-technical metrics, or other techniques that analyze the source code itself. None of them has clearly examined the usefulness of software metrics that measure the present code variability. This research has shown that exactly these metrics could provide further improvements to many existing SDP approaches. This study has primarily analyzed the usefulness of variability-aware code metrics based on the conditional compilation by the CPP. Further research should investigate additional possibilities to measure the present code variability within the source code of other languages than C-based ones. Another important topic of future research should aim towards more interpretable methods of machine learning to derive precise and univariate recommendations or guidelines for developers. These guidelines could provide additional potential to reduce the risk and number of software defects within a software project, which in turn provides a further reduction of costs. At this point, the conclusions of this study indicate that it would be meaningful to shift the focus of existing guidelines from dictating the style and standards of normal source code to regulating and guiding the implemented code variability.

9.3 Contribution

This study has contributed to filling a gap in method-level SDP research using variability-aware code metrics. The related work chapter has shown that there was a severe lack of this consideration. This research clearly shows the superiority of variability-aware code metrics over traditional non-variability-aware metrics. This should point future research to consider such metrics in their approaches. While several papers have claimed that method-level SDP is still a non-solved task and severely requires further research [LBMP08] [PPB18], this study has contributed to exactly this research area and pointed out a new potential for existing approaches. The consideration of metrics like the used SCMs in this thesis could provide further improvements to existing SDP approaches, which in turn could contribute to the reduction of software defects. This finally ends up in a small contribution to the industrial successes of software projects.

A Appendix

Table A.1: Perfectly Spearman-correlated input features.

Feature Combinations	
ALL Vars per Function x Feature Type(hex=1)	ALL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)
	CC on VPs x Feature Type(hex=1)
	Full-TD x Feature Type(hex=1)
	VP ND_Max x Feature Type(hex=1)
	Visible-TD x Feature Type(hex=1)
ALL Vars per Function x Feature Type(integer=1, int=1)	ALL_WITH_BUILD_VARS Vars per Function x Feature Type(integer=1, int=1)
ALL Vars per Function x Feature Type(string=1)	ALL_WITH_BUILD_VARS Vars per Function x Feature Type(string=1)
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)	CC on VPs x Feature Type(hex=1)
	Full-TD x Feature Type(hex=1)
	VP ND_Max x Feature Type(hex=1)
	Visible-TD x Feature Type(hex=1)
CC on VPs x Feature Type(hex=1)	Full-TD x Feature Type(hex=1)
	VP ND_Max x Feature Type(hex=1)
	Visible-TD x Feature Type(hex=1)
CC on VPs x Feature Type(integer=1, int=1)	Visible-TD x Feature Type(integer=1, int=1)
CC on VPs x Feature Type(string=1)	Full-TD x Feature Type(string=1)
	Visible-TD x Feature Type(string=1)
DC Fan-In(global) x Feature Type(hex=1)	Classical Fan-In(global)
DC Fan-In(local) x Feature Type(hex=1)	Classical Fan-In(local)
DC Fan-Out(global x No Stubs) x Feature Type(hex=1)	DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(global x No ext. VPs) x Feature Type(hex=1)	Classical Fan-Out(global)
DC Fan-Out(global) x Feature Type(hex=1)	Classical Fan-Out(global)
	DC Fan-Out(global x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(local x No Stubs) x Feature Type(hex=1)	DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(local x No ext. VPs) x Feature Type(hex=1)	Classical Fan-Out(local)
DC Fan-Out(local) x Feature Type(hex=1)	Classical Fan-Out(local)
	DC Fan-Out(local x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(global x No Stubs) x Feature Type(hex=1)	EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(global) x Feature Type(hex=1)	EV DC Fan-Out(global x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(local x No Stubs) x Feature Type(hex=1)	EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(local) x Feature Type(hex=1)	EV DC Fan-Out(local x No ext. VPs) x Feature Type(hex=1)
EXTERNAL Vars per Function x Feature Type(integer=1, int=1)	EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(integer=1, int=1)

Continued on next page

Feature Combinations	
EXTERNAL Vars per Function x Feature Type(string=1)	EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(string=1)
Full-TD x Feature Type(hex=1)	Visible-TD x Feature Type(hex=1)
Full-TD x Feature Type(string=1)	Visible-TD x Feature Type(string=1)
INTERNAL Vars per Function x Feature Type(hex=1)	ALL Vars per Function x Feature Type(hex=1)
	ALL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)
	CC on VPs x Feature Type(hex=1)
	Full-TD x Feature Type(hex=1)
	VP ND_Max x Feature Type(hex=1)
	Visible-TD x Feature Type(hex=1)
INTERNAL Vars per Function x Feature Type(integer=1, int=1)	VP ND_Max x Feature Type(integer=1, int=1)
INTERNAL Vars per Function x Feature Type(string=1)	CC on VPs x Feature Type(string=1)
	Full-TD x Feature Type(string=1)
	Visible-TD x Feature Type(string=1)
VP ND_Max x Feature Type(hex=1)	Full-TD x Feature Type(hex=1)
	Visible-TD x Feature Type(hex=1)

Table A.2: Fifty highest Pearson-correlated software metrics to the target variable.

Input feature	Correlation Coefficient
LoC	0.078877
SCoC	0.070451
McCabe + CC on VPs x Feature distance	0.066771
McCabe + CC on VPs x Hierarchy Levels	0.066494
McCabe + CC on VPs x Feature Type(bool=1)	0.064626
McCabe + CC on VPs	0.064625
McCabe + CC on VPs x Hierarchy Types(0-0-1)	0.064081
McCabe + CC on VPs x Hierarchy Types(0-1-0)	0.063661
McCabe + CC on VPs x Hierarchy Types(1-0-0)	0.063385
McCabe + CC on VPs x Feature Type(tristate=1)	0.063298
McCabe	0.063197
McCabe + CC on VPs x Feature Type(hex=1)	0.063197
McCabe + CC on VPs x Feature Type(string=1)	0.063196
McCabe + CC on VPs x Feature Type(integer=1, int=1)	0.063195
McCabe + CC on VPs x NUMBER_OF_CHILDREN	0.062995
McCabe + CC on VPs x OUTGOING_CONNECTIONS	0.062312
Combined ND_Max x Feature distance	0.056348
Combined ND_Max x Hierarchy Levels	0.055466
Combined ND_Max x Feature Type(bool=1)	0.051720
Combined ND_Max	0.051387
Combined ND_Max x Hierarchy Types(0-0-1)	0.050458
Combined ND_Max x Hierarchy Types(0-1-0)	0.048920
Combined ND_Max x Hierarchy Types(1-0-0)	0.048032
Combined ND_Max x Feature Type(tristate=1)	0.047978
Classic ND_Max	0.047487
Combined ND_Max x Feature Type(hex=1)	0.047486
Combined ND_Max x Feature Type(string=1)	0.047485
Combined ND_Max x Feature Type(integer=1, int=1)	0.047478
Visible-TD	0.045217
Full-TD	0.044288
Visible-TD x Feature Type(bool=1)	0.044218
No. int. blocks x BLOCK_AS_ONE	0.044088
CC on VPs	0.044040
CC on VPs x Feature Type(bool=1)	0.043716
No. int. blocks x SEPARATE_PARTIAL_BLOCKS	0.043697
Full-TD x Feature Type(bool=1)	0.043040
INTERNAL Vars per Function	0.042797
DC Fan-Out(local)	0.041465
INTERNAL Vars per Function x Feature Type(bool=1)	0.041376
DC Fan-Out(local) x Feature Type(bool=1)	0.041217
DC Fan-Out(local) x Hierarchy Types(0-0-1)	0.040926
DC Fan-Out(local x No Stubs)	0.040657
DC Fan-Out(local x No Stubs) x Feature Type(bool=1)	0.040411
DC Fan-Out(local) x Feature distance	0.040201
DC Fan-Out(local x No Stubs) x Hierarchy Types(0-0-1)	0.040174
DC Fan-Out(local x No ext. VPs)	0.040019
DC Fan-Out(local x No ext. VPs) x Feature Type(bool=1)	0.039833
Visible-TD x Feature distance	0.039602
DC Fan-Out(local) x Hierarchy Types(1-0-0)	0.039599
DC Fan-Out(local x No ext. VPs) x Hierarchy Types(0-0-1)	0.039573

Table A.3: Defective rates within the locations on the second level of hierarchy.

Location up to level 2	Defective rate	Location up to level 2	Defective rate
drivers/base	5.079156	fs/Others	1.208934
kernel/Others	5.015353	drivers/spi	1.155751
mm/Others	5.000000	net/sched	1.114413
mm	4.957176	drivers/md	1.103931
arch/arm64	4.761905	arch/Others	1.099580
arch/tile	4.446064	net/netfilter	1.009749
drivers/gpio	4.320666	drivers/s390	1.005587
kernel	4.301458	drivers/tty	0.999345
lib	4.032258	drivers/media	0.912896
drivers/pci	3.518164	security/Others	0.909526
fs/btrfs	3.429061	drivers/net	0.901534
drivers/mmc	3.341794	drivers/power	0.850547
drivers/irqchip	3.290130	drivers/mfd	0.828609
drivers/iommu	3.184282	drivers/crypto	0.796813
arch/sparc	2.946200	drivers/regulator	0.764526
net/core	2.915082	drivers/scsi	0.728563
Others	2.815013	fs/nfs	0.727691
block	2.748585	drivers/input	0.684199
sound/soc	2.551574	drivers/ata	0.662879
drivers/Others	2.499774	drivers/platform	0.654253
fs	2.338308	fs/xfs	0.637233
drivers/dma	2.308239	arch/s390	0.629178
arch/x86	2.229878	arch/m68k	0.616197
drivers/i2c	2.007205	drivers/watchdog	0.609081
arch/mips	1.996458	drivers/infiniband	0.595877
crypto	1.853035	sound/core	0.587372
drivers/block	1.816557	sound/Others	0.575080
net/ipv6	1.755379	arch/alpha	0.573301
drivers/usb	1.677187	arch/ia64	0.568475
drivers/pinctrl	1.671542	fs/nfsd	0.494234
drivers/mtd	1.656196	net/bluetooth	0.486914
arch/arm	1.634646	lib/Others	0.420168
drivers/gpu	1.528685	drivers/rtc	0.375134
drivers/staging	1.506148	drivers/hwmon	0.373349
crypto/Others	1.503759	drivers/target	0.373134
drivers/char	1.484938	net/sunrpc	0.334225
arch/powerpc	1.417411	drivers/video	0.285055
drivers/iio	1.392405	drivers/edac	0.278035
drivers/clk	1.387283	arch/sh	0.272331
drivers/hid	1.380898	fs/ocfs2	0.225734
drivers/misc	1.374964	drivers/ide	0.172563
kernel/trace	1.368363	sound/pci	0.080537
net/mac80211	1.361032	block/Others	0.000000
drivers/acpi	1.341782	sound/isa	0.000000
net/ipv4	1.273101	drivers/isdn	0.000000
net/Others	1.223266	virt/Others	0.000000

Table A.4: Software metrics in the LNV feature subset.

Metric	Metric
Classic ND_Avg	EV Classical Fan-Out(global)
Classic ND_Max	EV Classical Fan-Out(local)
Classical Fan-In(global)	LoC
Classical Fan-In(local)	LoC Comment Ratio
Classical Fan-Out(global)	McCabe
Classical Fan-Out(local)	SCoC
EV Classical Fan-In(global)	SCoC Comment Ratio
EV Classical Fan-In(local)	

Table A.5: Software metrics in the LPV feature subset.

Metric	Metric
ALL Vars per Function	INTERNAL Vars per Function
ALL_WITH_BUILD_VARS Vars per Function	No. int. blocks x BLOCK_AS_ONE
CC on VPs	No. int. blocks x SEPARATE_PARTIAL_BLOCKS
EV VP Fan-In(global)	VP Fan-In(global)
EV VP Fan-In(local)	VP Fan-In(local)
EV VP Fan-Out(global)	VP Fan-Out(global)
EV VP Fan-Out(local)	VP Fan-Out(local)
EXTERNAL Vars per Function	VP ND_Avg
EXTERNAL_WITH_BUILD_VARS Vars per Function	VP ND_Max
Full-TD	Visible-TD

Table A.6: Software metrics in the LCV feature subset.

Metric	Metric
Combined ND_Avg	EV DC Fan-Out(global x No Stubs)
Combined ND_Max	EV DC Fan-Out(global x No ext. VPs)
DC Fan-In(global)	EV DC Fan-Out(global)
DC Fan-In(local)	EV DC Fan-Out(local x No Stubs x No ext. VPs)
DC Fan-Out(global x No Stubs x No ext. VPs)	EV DC Fan-Out(local x No Stubs)
DC Fan-Out(global x No Stubs)	EV DC Fan-Out(local x No ext. VPs)
DC Fan-Out(global x No ext. VPs)	EV DC Fan-Out(local)
DC Fan-Out(global)	LoF
DC Fan-Out(local x No Stubs x No ext. VPs)	McCabe + CC on VPs
DC Fan-Out(local x No Stubs)	PLoF
DC Fan-Out(local x No ext. VPs)	PSCoF
DC Fan-Out(local)	SCoF
EV DC Fan-In(global)	Undisciplined CPP
EV DC Fan-In(local)	
EV DC Fan-Out(global x No Stubs x No ext. VPs)	

Table A.7: Software metrics in the GPV feature subset.

Metric
ALL Vars per Function x ALL_CTCR
ALL Vars per Function x COC
ALL Vars per Function x Feature Type(bool=1)
ALL Vars per Function x Feature Type(hex=1)
ALL Vars per Function x Feature Type(integer=1, int=1)
ALL Vars per Function x Feature Type(string=1)
ALL Vars per Function x Feature Type(tristate=1)
ALL Vars per Function x Feature distance
ALL Vars per Function x Hierarchy Levels
ALL Vars per Function x Hierarchy Types(0-0-1)
ALL Vars per Function x Hierarchy Types(0-1-0)
ALL Vars per Function x Hierarchy Types(1-0-0)
ALL Vars per Function x INCOMIG_CONNECTIONS
ALL Vars per Function x NUMBER_OF_CHILDREN
ALL Vars per Function x OUTGOING_CONNECTIONS
ALL Vars per Function x POSITIVE_SIZES
ALL Vars per Function x SD_FILE
ALL Vars per Function x SD_VP
ALL Vars per Function x TOTAL_SIZES
ALL_WITH_BUILD_VARS Vars per Function x ALL_CTCR
ALL_WITH_BUILD_VARS Vars per Function x COC
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(bool=1)
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(integer=1, int=1)
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(string=1)
ALL_WITH_BUILD_VARS Vars per Function x Feature Type(tristate=1)
ALL_WITH_BUILD_VARS Vars per Function x Feature distance
ALL_WITH_BUILD_VARS Vars per Function x Hierarchy Levels
ALL_WITH_BUILD_VARS Vars per Function x Hierarchy Types(0-0-1)
ALL_WITH_BUILD_VARS Vars per Function x Hierarchy Types(0-1-0)
ALL_WITH_BUILD_VARS Vars per Function x Hierarchy Types(1-0-0)
ALL_WITH_BUILD_VARS Vars per Function x INCOMIG_CONNECTIONS
ALL_WITH_BUILD_VARS Vars per Function x NUMBER_OF_CHILDREN
ALL_WITH_BUILD_VARS Vars per Function x OUTGOING_CONNECTIONS
ALL_WITH_BUILD_VARS Vars per Function x POSITIVE_SIZES
ALL_WITH_BUILD_VARS Vars per Function x SD_FILE
ALL_WITH_BUILD_VARS Vars per Function x SD_VP
ALL_WITH_BUILD_VARS Vars per Function x TOTAL_SIZES
CC on VPs x ALL_CTCR
CC on VPs x COC
CC on VPs x Feature Type(bool=1)
CC on VPs x Feature Type(hex=1)
CC on VPs x Feature Type(integer=1, int=1)
CC on VPs x Feature Type(string=1)
CC on VPs x Feature Type(tristate=1)
CC on VPs x Feature distance
CC on VPs x Hierarchy Levels
CC on VPs x Hierarchy Types(0-0-1)
CC on VPs x Hierarchy Types(0-1-0)
CC on VPs x Hierarchy Types(1-0-0)
CC on VPs x INCOMIG_CONNECTIONS
CC on VPs x NUMBER_OF_CHILDREN
CC on VPs x OUTGOING_CONNECTIONS
CC on VPs x POSITIVE_SIZES
CC on VPs x SD_FILE
CC on VPs x SD_VP
CC on VPs x TOTAL_SIZES

Continued on next page

Metric
EV VP Fan-In(global)
EV VP Fan-In(local)
EV VP Fan-Out(global)
EV VP Fan-Out(local)
EXTERNAL Vars per Function x ALL_CTCR
EXTERNAL Vars per Function x COC
EXTERNAL Vars per Function x Feature Type(bool=1)
EXTERNAL Vars per Function x Feature Type(hex=1)
EXTERNAL Vars per Function x Feature Type(integer=1, int=1)
EXTERNAL Vars per Function x Feature Type(string=1)
EXTERNAL Vars per Function x Feature Type(tristate=1)
EXTERNAL Vars per Function x Feature distance
EXTERNAL Vars per Function x Hierarchy Levels
EXTERNAL Vars per Function x Hierarchy Types(0-0-1)
EXTERNAL Vars per Function x Hierarchy Types(0-1-0)
EXTERNAL Vars per Function x Hierarchy Types(1-0-0)
EXTERNAL Vars per Function x INCOMIG_CONNECTIONS
EXTERNAL Vars per Function x NUMBER_OF_CHILDREN
EXTERNAL Vars per Function x OUTGOING_CONNECTIONS
EXTERNAL Vars per Function x POSITIVE_SIZES
EXTERNAL Vars per Function x SD_FILE
EXTERNAL Vars per Function x SD_VP
EXTERNAL Vars per Function x TOTAL_SIZES
EXTERNAL_WITH_BUILD_VARS Vars per Function x ALL_CTCR
EXTERNAL_WITH_BUILD_VARS Vars per Function x COC
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(bool=1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(hex=1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(integer=1, int=1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(string=1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature Type(tristate=1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Feature distance
EXTERNAL_WITH_BUILD_VARS Vars per Function x Hierarchy Levels
EXTERNAL_WITH_BUILD_VARS Vars per Function x Hierarchy Types(0-0-1)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Hierarchy Types(0-1-0)
EXTERNAL_WITH_BUILD_VARS Vars per Function x Hierarchy Types(1-0-0)
EXTERNAL_WITH_BUILD_VARS Vars per Function x INCOMIG_CONNECTIONS
EXTERNAL_WITH_BUILD_VARS Vars per Function x NUMBER_OF_CHILDREN
EXTERNAL_WITH_BUILD_VARS Vars per Function x OUTGOING_CONNECTIONS
EXTERNAL_WITH_BUILD_VARS Vars per Function x POSITIVE_SIZES
EXTERNAL_WITH_BUILD_VARS Vars per Function x SD_FILE
EXTERNAL_WITH_BUILD_VARS Vars per Function x SD_VP
EXTERNAL_WITH_BUILD_VARS Vars per Function x TOTAL_SIZES
Full-TD x ALL_CTCR
Full-TD x COC
Full-TD x Feature Type(bool=1)
Full-TD x Feature Type(hex=1)
Full-TD x Feature Type(integer=1, int=1)
Full-TD x Feature Type(string=1)
Full-TD x Feature Type(tristate=1)
Full-TD x Feature distance
Full-TD x Hierarchy Levels
Full-TD x Hierarchy Types(0-0-1)
Full-TD x Hierarchy Types(0-1-0)
Full-TD x Hierarchy Types(1-0-0)
Full-TD x INCOMIG_CONNECTIONS
Full-TD x NUMBER_OF_CHILDREN
Full-TD x OUTGOING_CONNECTIONS
Full-TD x POSITIVE_SIZES
Full-TD x SD_FILE

Continued on next page

Metric
Full-TD x SD_VP
Full-TD x TOTAL_SIZES
INTERNAL Vars per Function x ALL_CTCR
INTERNAL Vars per Function x COC
INTERNAL Vars per Function x Feature Type(bool=1)
INTERNAL Vars per Function x Feature Type(hex=1)
INTERNAL Vars per Function x Feature Type(integer=1, int=1)
INTERNAL Vars per Function x Feature Type(string=1)
INTERNAL Vars per Function x Feature Type(tristate=1)
INTERNAL Vars per Function x Feature distance
INTERNAL Vars per Function x Hierarchy Levels
INTERNAL Vars per Function x Hierarchy Types(0-0-1)
INTERNAL Vars per Function x Hierarchy Types(0-1-0)
INTERNAL Vars per Function x Hierarchy Types(1-0-0)
INTERNAL Vars per Function x INCOMIG_CONNECTIONS
INTERNAL Vars per Function x NUMBER_OF_CHILDREN
INTERNAL Vars per Function x OUTGOING_CONNECTIONS
INTERNAL Vars per Function x POSITIVE_SIZES
INTERNAL Vars per Function x SD_FILE
INTERNAL Vars per Function x SD_VP
INTERNAL Vars per Function x TOTAL_SIZES
No. int. blocks x BLOCK_AS_ONE
No. int. blocks x SEPARATE_PARTIAL_BLOCKS
VP Fan-In(global)
VP Fan-In(local)
VP Fan-Out(global)
VP Fan-Out(local)
VP ND_Avg x ALL_CTCR
VP ND_Avg x COC
VP ND_Avg x Feature Type(bool=1)
VP ND_Avg x Feature Type(hex=1)
VP ND_Avg x Feature Type(integer=1, int=1)
VP ND_Avg x Feature Type(string=1)
VP ND_Avg x Feature Type(tristate=1)
VP ND_Avg x Feature distance
VP ND_Avg x Hierarchy Levels
VP ND_Avg x Hierarchy Types(0-0-1)
VP ND_Avg x Hierarchy Types(0-1-0)
VP ND_Avg x Hierarchy Types(1-0-0)
VP ND_Avg x INCOMIG_CONNECTIONS
VP ND_Avg x NUMBER_OF_CHILDREN
VP ND_Avg x OUTGOING_CONNECTIONS
VP ND_Avg x POSITIVE_SIZES
VP ND_Avg x SD_FILE
VP ND_Avg x SD_VP
VP ND_Avg x TOTAL_SIZES
VP ND_Max x ALL_CTCR
VP ND_Max x COC
VP ND_Max x Feature Type(bool=1)
VP ND_Max x Feature Type(hex=1)
VP ND_Max x Feature Type(integer=1, int=1)
VP ND_Max x Feature Type(string=1)
VP ND_Max x Feature Type(tristate=1)
VP ND_Max x Feature distance
VP ND_Max x Hierarchy Levels
VP ND_Max x Hierarchy Types(0-0-1)
VP ND_Max x Hierarchy Types(0-1-0)
VP ND_Max x Hierarchy Types(1-0-0)
VP ND_Max x INCOMIG_CONNECTIONS

Continued on next page

Metric
VP ND_Max x NUMBER_OF_CHILDREN
VP ND_Max x OUTGOING_CONNECTIONS
VP ND_Max x POSITIVE_SIZES
VP ND_Max x SD_FILE
VP ND_Max x SD_VP
VP ND_Max x TOTAL_SIZES
Visible-TD x ALL_CTCR
Visible-TD x COC
Visible-TD x Feature Type(bool=1)
Visible-TD x Feature Type(hex=1)
Visible-TD x Feature Type(integer=1, int=1)
Visible-TD x Feature Type(string=1)
Visible-TD x Feature Type(tristate=1)
Visible-TD x Feature distance
Visible-TD x Hierarchy Levels
Visible-TD x Hierarchy Types(0-0-1)
Visible-TD x Hierarchy Types(0-1-0)
Visible-TD x Hierarchy Types(1-0-0)
Visible-TD x INCOMIG_CONNECTIONS
Visible-TD x NUMBER_OF_CHILDREN
Visible-TD x OUTGOING_CONNECTIONS
Visible-TD x POSITIVE_SIZES
Visible-TD x SD_FILE
Visible-TD x SD_VP
Visible-TD x TOTAL_SIZES

Table A.8: Software metrics in the GCV feature subset.

Metric
Combined ND_Avg x ALL_CTCR
Combined ND_Avg x COC
Combined ND_Avg x Feature Type(bool=1)
Combined ND_Avg x Feature Type(hex=1)
Combined ND_Avg x Feature Type(integer=1, int=1)
Combined ND_Avg x Feature Type(string=1)
Combined ND_Avg x Feature Type(tristate=1)
Combined ND_Avg x Feature distance
Combined ND_Avg x Hierarchy Levels
Combined ND_Avg x Hierarchy Types(0-0-1)
Combined ND_Avg x Hierarchy Types(0-1-0)
Combined ND_Avg x Hierarchy Types(1-0-0)
Combined ND_Avg x INCOMIG_CONNECTIONS
Combined ND_Avg x NUMBER_OF_CHILDREN
Combined ND_Avg x OUTGOING_CONNECTIONS
Combined ND_Avg x POSITIVE_SIZES
Combined ND_Avg x SD_FILE
Combined ND_Avg x SD_VP
Combined ND_Avg x TOTAL_SIZES
Combined ND_Max x ALL_CTCR
Combined ND_Max x COC
Combined ND_Max x Feature Type(bool=1)
Combined ND_Max x Feature Type(hex=1)
Combined ND_Max x Feature Type(integer=1, int=1)
Combined ND_Max x Feature Type(string=1)
Combined ND_Max x Feature Type(tristate=1)
Combined ND_Max x Feature distance
Combined ND_Max x Hierarchy Levels

Continued on next page

Metric
Combined ND_Max x Hierarchy Types(0-0-1)
Combined ND_Max x Hierarchy Types(0-1-0)
Combined ND_Max x Hierarchy Types(1-0-0)
Combined ND_Max x INCOMIG_CONNECTIONS
Combined ND_Max x NUMBER_OF_CHILDREN
Combined ND_Max x OUTGOING_CONNECTIONS
Combined ND_Max x POSITIVE_SIZES
Combined ND_Max x SD_FILE
Combined ND_Max x SD_VP
Combined ND_Max x TOTAL_SIZES
DC Fan-In(global) x ALL_CTCR
DC Fan-In(global) x COC
DC Fan-In(global) x Feature Type(bool=1)
DC Fan-In(global) x Feature Type(hex=1)
DC Fan-In(global) x Feature Type(integer=1, int=1)
DC Fan-In(global) x Feature Type(string=1)
DC Fan-In(global) x Feature Type(tristate=1)
DC Fan-In(global) x Feature distance
DC Fan-In(global) x Hierarchy Levels
DC Fan-In(global) x Hierarchy Types(0-0-1)
DC Fan-In(global) x Hierarchy Types(0-1-0)
DC Fan-In(global) x Hierarchy Types(1-0-0)
DC Fan-In(global) x INCOMIG_CONNECTIONS
DC Fan-In(global) x NUMBER_OF_CHILDREN
DC Fan-In(global) x OUTGOING_CONNECTIONS
DC Fan-In(global) x POSITIVE_SIZES
DC Fan-In(global) x SD_FILE
DC Fan-In(global) x SD_VP
DC Fan-In(global) x TOTAL_SIZES
DC Fan-In(local) x ALL_CTCR
DC Fan-In(local) x COC
DC Fan-In(local) x Feature Type(bool=1)
DC Fan-In(local) x Feature Type(hex=1)
DC Fan-In(local) x Feature Type(integer=1, int=1)
DC Fan-In(local) x Feature Type(string=1)
DC Fan-In(local) x Feature Type(tristate=1)
DC Fan-In(local) x Feature distance
DC Fan-In(local) x Hierarchy Levels
DC Fan-In(local) x Hierarchy Types(0-0-1)
DC Fan-In(local) x Hierarchy Types(0-1-0)
DC Fan-In(local) x Hierarchy Types(1-0-0)
DC Fan-In(local) x INCOMIG_CONNECTIONS
DC Fan-In(local) x NUMBER_OF_CHILDREN
DC Fan-In(local) x OUTGOING_CONNECTIONS
DC Fan-In(local) x POSITIVE_SIZES
DC Fan-In(local) x SD_FILE
DC Fan-In(local) x SD_VP
DC Fan-In(local) x TOTAL_SIZES
DC Fan-Out(global x No Stubs x No ext. VPs) x ALL_CTCR
DC Fan-Out(global x No Stubs x No ext. VPs) x COC
DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(bool=1)
DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(integer=1, int=1)
DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(string=1)
DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(tristate=1)
DC Fan-Out(global x No Stubs x No ext. VPs) x Feature distance
DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Levels
DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Types(0-0-1)
DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Types(0-1-0)

Continued on next page

Metric
DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Types(1-0-0)
DC Fan-Out(global x No Stubs x No ext. VPs) x INCOMIG_CONNECTIONS
DC Fan-Out(global x No Stubs x No ext. VPs) x NUMBER_OF_CHILDREN
DC Fan-Out(global x No Stubs x No ext. VPs) x OUTGOING_CONNECTIONS
DC Fan-Out(global x No Stubs x No ext. VPs) x POSITIVE_SIZES
DC Fan-Out(global x No Stubs x No ext. VPs) x SD_FILE
DC Fan-Out(global x No Stubs x No ext. VPs) x SD_VP
DC Fan-Out(global x No Stubs x No ext. VPs) x TOTAL_SIZES
DC Fan-Out(global x No Stubs) x ALL_CTCR
DC Fan-Out(global x No Stubs) x COC
DC Fan-Out(global x No Stubs) x Feature Type(bool=1)
DC Fan-Out(global x No Stubs) x Feature Type(hex=1)
DC Fan-Out(global x No Stubs) x Feature Type(integer=1, int=1)
DC Fan-Out(global x No Stubs) x Feature Type(string=1)
DC Fan-Out(global x No Stubs) x Feature Type(tristate=1)
DC Fan-Out(global x No Stubs) x Feature distance
DC Fan-Out(global x No Stubs) x Hierarchy Levels
DC Fan-Out(global x No Stubs) x Hierarchy Types(0-0-1)
DC Fan-Out(global x No Stubs) x Hierarchy Types(0-1-0)
DC Fan-Out(global x No Stubs) x Hierarchy Types(1-0-0)
DC Fan-Out(global x No Stubs) x INCOMIG_CONNECTIONS
DC Fan-Out(global x No Stubs) x NUMBER_OF_CHILDREN
DC Fan-Out(global x No Stubs) x OUTGOING_CONNECTIONS
DC Fan-Out(global x No Stubs) x POSITIVE_SIZES
DC Fan-Out(global x No Stubs) x SD_FILE
DC Fan-Out(global x No Stubs) x SD_VP
DC Fan-Out(global x No Stubs) x TOTAL_SIZES
DC Fan-Out(global x No ext. VPs) x ALL_CTCR
DC Fan-Out(global x No ext. VPs) x COC
DC Fan-Out(global x No ext. VPs) x Feature Type(bool=1)
DC Fan-Out(global x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(global x No ext. VPs) x Feature Type(integer=1, int=1)
DC Fan-Out(global x No ext. VPs) x Feature Type(string=1)
DC Fan-Out(global x No ext. VPs) x Feature Type(tristate=1)
DC Fan-Out(global x No ext. VPs) x Feature distance
DC Fan-Out(global x No ext. VPs) x Hierarchy Levels
DC Fan-Out(global x No ext. VPs) x Hierarchy Types(0-0-1)
DC Fan-Out(global x No ext. VPs) x Hierarchy Types(0-1-0)
DC Fan-Out(global x No ext. VPs) x Hierarchy Types(1-0-0)
DC Fan-Out(global x No ext. VPs) x INCOMIG_CONNECTIONS
DC Fan-Out(global x No ext. VPs) x NUMBER_OF_CHILDREN
DC Fan-Out(global x No ext. VPs) x OUTGOING_CONNECTIONS
DC Fan-Out(global x No ext. VPs) x POSITIVE_SIZES
DC Fan-Out(global x No ext. VPs) x SD_FILE
DC Fan-Out(global x No ext. VPs) x SD_VP
DC Fan-Out(global x No ext. VPs) x TOTAL_SIZES
DC Fan-Out(global) x ALL_CTCR
DC Fan-Out(global) x COC
DC Fan-Out(global) x Feature Type(bool=1)
DC Fan-Out(global) x Feature Type(hex=1)
DC Fan-Out(global) x Feature Type(integer=1, int=1)
DC Fan-Out(global) x Feature Type(string=1)
DC Fan-Out(global) x Feature Type(tristate=1)
DC Fan-Out(global) x Feature distance
DC Fan-Out(global) x Hierarchy Levels
DC Fan-Out(global) x Hierarchy Types(0-0-1)
DC Fan-Out(global) x Hierarchy Types(0-1-0)
DC Fan-Out(global) x Hierarchy Types(1-0-0)
DC Fan-Out(global) x INCOMIG_CONNECTIONS

Continued on next page

Metric
DC Fan-Out(global) x NUMBER_OF_CHILDREN
DC Fan-Out(global) x OUTGOING_CONNECTIONS
DC Fan-Out(global) x POSITIVE_SIZES
DC Fan-Out(global) x SD_FILE
DC Fan-Out(global) x SD_VP
DC Fan-Out(global) x TOTAL_SIZES
DC Fan-Out(local x No Stubs x No ext. VPs) x ALL_CTCR
DC Fan-Out(local x No Stubs x No ext. VPs) x COC
DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(bool=1)
DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(integer=1, int=1)
DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(string=1)
DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(tristate=1)
DC Fan-Out(local x No Stubs x No ext. VPs) x Feature distance
DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Levels
DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Types(0-0-1)
DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Types(0-1-0)
DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Types(1-0-0)
DC Fan-Out(local x No Stubs x No ext. VPs) x INCOMIG_CONNECTIONS
DC Fan-Out(local x No Stubs x No ext. VPs) x NUMBER_OF_CHILDREN
DC Fan-Out(local x No Stubs x No ext. VPs) x OUTGOING_CONNECTIONS
DC Fan-Out(local x No Stubs x No ext. VPs) x POSITIVE_SIZES
DC Fan-Out(local x No Stubs x No ext. VPs) x SD_FILE
DC Fan-Out(local x No Stubs x No ext. VPs) x SD_VP
DC Fan-Out(local x No Stubs x No ext. VPs) x TOTAL_SIZES
DC Fan-Out(local x No Stubs) x ALL_CTCR
DC Fan-Out(local x No Stubs) x COC
DC Fan-Out(local x No Stubs) x Feature Type(bool=1)
DC Fan-Out(local x No Stubs) x Feature Type(hex=1)
DC Fan-Out(local x No Stubs) x Feature Type(integer=1, int=1)
DC Fan-Out(local x No Stubs) x Feature Type(string=1)
DC Fan-Out(local x No Stubs) x Feature Type(tristate=1)
DC Fan-Out(local x No Stubs) x Feature distance
DC Fan-Out(local x No Stubs) x Hierarchy Levels
DC Fan-Out(local x No Stubs) x Hierarchy Types(0-0-1)
DC Fan-Out(local x No Stubs) x Hierarchy Types(0-1-0)
DC Fan-Out(local x No Stubs) x Hierarchy Types(1-0-0)
DC Fan-Out(local x No Stubs) x INCOMIG_CONNECTIONS
DC Fan-Out(local x No Stubs) x NUMBER_OF_CHILDREN
DC Fan-Out(local x No Stubs) x OUTGOING_CONNECTIONS
DC Fan-Out(local x No Stubs) x POSITIVE_SIZES
DC Fan-Out(local x No Stubs) x SD_FILE
DC Fan-Out(local x No Stubs) x SD_VP
DC Fan-Out(local x No Stubs) x TOTAL_SIZES
DC Fan-Out(local x No ext. VPs) x ALL_CTCR
DC Fan-Out(local x No ext. VPs) x COC
DC Fan-Out(local x No ext. VPs) x Feature Type(bool=1)
DC Fan-Out(local x No ext. VPs) x Feature Type(hex=1)
DC Fan-Out(local x No ext. VPs) x Feature Type(integer=1, int=1)
DC Fan-Out(local x No ext. VPs) x Feature Type(string=1)
DC Fan-Out(local x No ext. VPs) x Feature Type(tristate=1)
DC Fan-Out(local x No ext. VPs) x Feature distance
DC Fan-Out(local x No ext. VPs) x Hierarchy Levels
DC Fan-Out(local x No ext. VPs) x Hierarchy Types(0-0-1)
DC Fan-Out(local x No ext. VPs) x Hierarchy Types(0-1-0)
DC Fan-Out(local x No ext. VPs) x Hierarchy Types(1-0-0)
DC Fan-Out(local x No ext. VPs) x INCOMIG_CONNECTIONS
DC Fan-Out(local x No ext. VPs) x NUMBER_OF_CHILDREN
DC Fan-Out(local x No ext. VPs) x OUTGOING_CONNECTIONS

Continued on next page

Metric
DC Fan-Out(local x No ext. VPs) x POSITIVE_SIZES
DC Fan-Out(local x No ext. VPs) x SD_FILE
DC Fan-Out(local x No ext. VPs) x SD_VP
DC Fan-Out(local x No ext. VPs) x TOTAL_SIZES
DC Fan-Out(local) x ALL_CTCR
DC Fan-Out(local) x COC
DC Fan-Out(local) x Feature Type(bool=1)
DC Fan-Out(local) x Feature Type(hex=1)
DC Fan-Out(local) x Feature Type(integer=1, int=1)
DC Fan-Out(local) x Feature Type(string=1)
DC Fan-Out(local) x Feature Type(tristate=1)
DC Fan-Out(local) x Feature distance
DC Fan-Out(local) x Hierarchy Levels
DC Fan-Out(local) x Hierarchy Types(0-0-1)
DC Fan-Out(local) x Hierarchy Types(0-1-0)
DC Fan-Out(local) x Hierarchy Types(1-0-0)
DC Fan-Out(local) x INCOMIG_CONNECTIONS
DC Fan-Out(local) x NUMBER_OF_CHILDREN
DC Fan-Out(local) x OUTGOING_CONNECTIONS
DC Fan-Out(local) x POSITIVE_SIZES
DC Fan-Out(local) x SD_FILE
DC Fan-Out(local) x SD_VP
DC Fan-Out(local) x TOTAL_SIZES
EV DC Fan-In(global) x ALL_CTCR
EV DC Fan-In(global) x COC
EV DC Fan-In(global) x Feature Type(bool=1)
EV DC Fan-In(global) x Feature Type(hex=1)
EV DC Fan-In(global) x Feature Type(integer=1, int=1)
EV DC Fan-In(global) x Feature Type(string=1)
EV DC Fan-In(global) x Feature Type(tristate=1)
EV DC Fan-In(global) x Feature distance
EV DC Fan-In(global) x Hierarchy Levels
EV DC Fan-In(global) x Hierarchy Types(0-0-1)
EV DC Fan-In(global) x Hierarchy Types(0-1-0)
EV DC Fan-In(global) x Hierarchy Types(1-0-0)
EV DC Fan-In(global) x INCOMIG_CONNECTIONS
EV DC Fan-In(global) x NUMBER_OF_CHILDREN
EV DC Fan-In(global) x OUTGOING_CONNECTIONS
EV DC Fan-In(global) x POSITIVE_SIZES
EV DC Fan-In(global) x SD_FILE
EV DC Fan-In(global) x SD_VP
EV DC Fan-In(global) x TOTAL_SIZES
EV DC Fan-In(local) x ALL_CTCR
EV DC Fan-In(local) x COC
EV DC Fan-In(local) x Feature Type(bool=1)
EV DC Fan-In(local) x Feature Type(hex=1)
EV DC Fan-In(local) x Feature Type(integer=1, int=1)
EV DC Fan-In(local) x Feature Type(string=1)
EV DC Fan-In(local) x Feature Type(tristate=1)
EV DC Fan-In(local) x Feature distance
EV DC Fan-In(local) x Hierarchy Levels
EV DC Fan-In(local) x Hierarchy Types(0-0-1)
EV DC Fan-In(local) x Hierarchy Types(0-1-0)
EV DC Fan-In(local) x Hierarchy Types(1-0-0)
EV DC Fan-In(local) x INCOMIG_CONNECTIONS
EV DC Fan-In(local) x NUMBER_OF_CHILDREN
EV DC Fan-In(local) x OUTGOING_CONNECTIONS
EV DC Fan-In(local) x POSITIVE_SIZES
EV DC Fan-In(local) x SD_FILE

Continued on next page

Metric
EV DC Fan-In(local) x SD_VP
EV DC Fan-In(local) x TOTAL_SIZES
EV DC Fan-Out(global x No Stubs x No ext. VPs) x ALL_CTCR
EV DC Fan-Out(global x No Stubs x No ext. VPs) x COC
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(bool=1)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(integer=1, int=1)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(string=1)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature Type(tristate=1)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Feature distance
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Levels
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Types(0-0-1)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Types(0-1-0)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x Hierarchy Types(1-0-0)
EV DC Fan-Out(global x No Stubs x No ext. VPs) x INCOMIG_CONNECTIONS
EV DC Fan-Out(global x No Stubs x No ext. VPs) x NUMBER_OF_CHILDREN
EV DC Fan-Out(global x No Stubs x No ext. VPs) x OUTGOING_CONNECTIONS
EV DC Fan-Out(global x No Stubs x No ext. VPs) x POSITIVE_SIZES
EV DC Fan-Out(global x No Stubs x No ext. VPs) x SD_FILE
EV DC Fan-Out(global x No Stubs x No ext. VPs) x SD_VP
EV DC Fan-Out(global x No Stubs x No ext. VPs) x TOTAL_SIZES
EV DC Fan-Out(global x No Stubs) x ALL_CTCR
EV DC Fan-Out(global x No Stubs) x COC
EV DC Fan-Out(global x No Stubs) x Feature Type(bool=1)
EV DC Fan-Out(global x No Stubs) x Feature Type(hex=1)
EV DC Fan-Out(global x No Stubs) x Feature Type(integer=1, int=1)
EV DC Fan-Out(global x No Stubs) x Feature Type(string=1)
EV DC Fan-Out(global x No Stubs) x Feature Type(tristate=1)
EV DC Fan-Out(global x No Stubs) x Feature distance
EV DC Fan-Out(global x No Stubs) x Hierarchy Levels
EV DC Fan-Out(global x No Stubs) x Hierarchy Types(0-0-1)
EV DC Fan-Out(global x No Stubs) x Hierarchy Types(0-1-0)
EV DC Fan-Out(global x No Stubs) x Hierarchy Types(1-0-0)
EV DC Fan-Out(global x No Stubs) x INCOMIG_CONNECTIONS
EV DC Fan-Out(global x No Stubs) x NUMBER_OF_CHILDREN
EV DC Fan-Out(global x No Stubs) x OUTGOING_CONNECTIONS
EV DC Fan-Out(global x No Stubs) x POSITIVE_SIZES
EV DC Fan-Out(global x No Stubs) x SD_FILE
EV DC Fan-Out(global x No Stubs) x SD_VP
EV DC Fan-Out(global x No Stubs) x TOTAL_SIZES
EV DC Fan-Out(global x No ext. VPs) x ALL_CTCR
EV DC Fan-Out(global x No ext. VPs) x COC
EV DC Fan-Out(global x No ext. VPs) x Feature Type(bool=1)
EV DC Fan-Out(global x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(global x No ext. VPs) x Feature Type(integer=1, int=1)
EV DC Fan-Out(global x No ext. VPs) x Feature Type(string=1)
EV DC Fan-Out(global x No ext. VPs) x Feature Type(tristate=1)
EV DC Fan-Out(global x No ext. VPs) x Feature distance
EV DC Fan-Out(global x No ext. VPs) x Hierarchy Levels
EV DC Fan-Out(global x No ext. VPs) x Hierarchy Types(0-0-1)
EV DC Fan-Out(global x No ext. VPs) x Hierarchy Types(0-1-0)
EV DC Fan-Out(global x No ext. VPs) x Hierarchy Types(1-0-0)
EV DC Fan-Out(global x No ext. VPs) x INCOMIG_CONNECTIONS
EV DC Fan-Out(global x No ext. VPs) x NUMBER_OF_CHILDREN
EV DC Fan-Out(global x No ext. VPs) x OUTGOING_CONNECTIONS
EV DC Fan-Out(global x No ext. VPs) x POSITIVE_SIZES
EV DC Fan-Out(global x No ext. VPs) x SD_FILE
EV DC Fan-Out(global x No ext. VPs) x SD_VP
EV DC Fan-Out(global x No ext. VPs) x TOTAL_SIZES

Continued on next page

Metric
EV DC Fan-Out(global) x ALL_CTCR
EV DC Fan-Out(global) x COC
EV DC Fan-Out(global) x Feature Type(bool=1)
EV DC Fan-Out(global) x Feature Type(hex=1)
EV DC Fan-Out(global) x Feature Type(integer=1, int=1)
EV DC Fan-Out(global) x Feature Type(string=1)
EV DC Fan-Out(global) x Feature Type(tristate=1)
EV DC Fan-Out(global) x Feature distance
EV DC Fan-Out(global) x Hierarchy Levels
EV DC Fan-Out(global) x Hierarchy Types(0-0-1)
EV DC Fan-Out(global) x Hierarchy Types(0-1-0)
EV DC Fan-Out(global) x Hierarchy Types(1-0-0)
EV DC Fan-Out(global) x INCOMIG_CONNECTIONS
EV DC Fan-Out(global) x NUMBER_OF_CHILDREN
EV DC Fan-Out(global) x OUTGOING_CONNECTIONS
EV DC Fan-Out(global) x POSITIVE_SIZES
EV DC Fan-Out(global) x SD_FILE
EV DC Fan-Out(global) x SD_VP
EV DC Fan-Out(global) x TOTAL_SIZES
EV DC Fan-Out(local x No Stubs x No ext. VPs) x ALL_CTCR
EV DC Fan-Out(local x No Stubs x No ext. VPs) x COC
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(bool=1)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(integer=1, int=1)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(string=1)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature Type(tristate=1)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Feature distance
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Levels
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Types(0-0-1)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Types(0-1-0)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x Hierarchy Types(1-0-0)
EV DC Fan-Out(local x No Stubs x No ext. VPs) x INCOMIG_CONNECTIONS
EV DC Fan-Out(local x No Stubs x No ext. VPs) x NUMBER_OF_CHILDREN
EV DC Fan-Out(local x No Stubs x No ext. VPs) x OUTGOING_CONNECTIONS
EV DC Fan-Out(local x No Stubs x No ext. VPs) x POSITIVE_SIZES
EV DC Fan-Out(local x No Stubs x No ext. VPs) x SD_FILE
EV DC Fan-Out(local x No Stubs x No ext. VPs) x SD_VP
EV DC Fan-Out(local x No Stubs x No ext. VPs) x TOTAL_SIZES
EV DC Fan-Out(local x No Stubs) x ALL_CTCR
EV DC Fan-Out(local x No Stubs) x COC
EV DC Fan-Out(local x No Stubs) x Feature Type(bool=1)
EV DC Fan-Out(local x No Stubs) x Feature Type(hex=1)
EV DC Fan-Out(local x No Stubs) x Feature Type(integer=1, int=1)
EV DC Fan-Out(local x No Stubs) x Feature Type(string=1)
EV DC Fan-Out(local x No Stubs) x Feature Type(tristate=1)
EV DC Fan-Out(local x No Stubs) x Feature distance
EV DC Fan-Out(local x No Stubs) x Hierarchy Levels
EV DC Fan-Out(local x No Stubs) x Hierarchy Types(0-0-1)
EV DC Fan-Out(local x No Stubs) x Hierarchy Types(0-1-0)
EV DC Fan-Out(local x No Stubs) x Hierarchy Types(1-0-0)
EV DC Fan-Out(local x No Stubs) x INCOMIG_CONNECTIONS
EV DC Fan-Out(local x No Stubs) x NUMBER_OF_CHILDREN
EV DC Fan-Out(local x No Stubs) x OUTGOING_CONNECTIONS
EV DC Fan-Out(local x No Stubs) x POSITIVE_SIZES
EV DC Fan-Out(local x No Stubs) x SD_FILE
EV DC Fan-Out(local x No Stubs) x SD_VP
EV DC Fan-Out(local x No Stubs) x TOTAL_SIZES
EV DC Fan-Out(local x No ext. VPs) x ALL_CTCR
EV DC Fan-Out(local x No ext. VPs) x COC

Continued on next page

Metric
EV DC Fan-Out(local x No ext. VPs) x Feature Type(bool=1)
EV DC Fan-Out(local x No ext. VPs) x Feature Type(hex=1)
EV DC Fan-Out(local x No ext. VPs) x Feature Type(integer=1, int=1)
EV DC Fan-Out(local x No ext. VPs) x Feature Type(string=1)
EV DC Fan-Out(local x No ext. VPs) x Feature Type(tristate=1)
EV DC Fan-Out(local x No ext. VPs) x Feature distance
EV DC Fan-Out(local x No ext. VPs) x Hierarchy Levels
EV DC Fan-Out(local x No ext. VPs) x Hierarchy Types(0-0-1)
EV DC Fan-Out(local x No ext. VPs) x Hierarchy Types(0-1-0)
EV DC Fan-Out(local x No ext. VPs) x Hierarchy Types(1-0-0)
EV DC Fan-Out(local x No ext. VPs) x INCOMIG_CONNECTIONS
EV DC Fan-Out(local x No ext. VPs) x NUMBER_OF_CHILDREN
EV DC Fan-Out(local x No ext. VPs) x OUTGOING_CONNECTIONS
EV DC Fan-Out(local x No ext. VPs) x POSITIVE_SIZES
EV DC Fan-Out(local x No ext. VPs) x SD_FILE
EV DC Fan-Out(local x No ext. VPs) x SD_VP
EV DC Fan-Out(local x No ext. VPs) x TOTAL_SIZES
EV DC Fan-Out(local) x ALL_CTCR
EV DC Fan-Out(local) x COC
EV DC Fan-Out(local) x Feature Type(bool=1)
EV DC Fan-Out(local) x Feature Type(hex=1)
EV DC Fan-Out(local) x Feature Type(integer=1, int=1)
EV DC Fan-Out(local) x Feature Type(string=1)
EV DC Fan-Out(local) x Feature Type(tristate=1)
EV DC Fan-Out(local) x Feature distance
EV DC Fan-Out(local) x Hierarchy Levels
EV DC Fan-Out(local) x Hierarchy Types(0-0-1)
EV DC Fan-Out(local) x Hierarchy Types(0-1-0)
EV DC Fan-Out(local) x Hierarchy Types(1-0-0)
EV DC Fan-Out(local) x INCOMIG_CONNECTIONS
EV DC Fan-Out(local) x NUMBER_OF_CHILDREN
EV DC Fan-Out(local) x OUTGOING_CONNECTIONS
EV DC Fan-Out(local) x POSITIVE_SIZES
EV DC Fan-Out(local) x SD_FILE
EV DC Fan-Out(local) x SD_VP
EV DC Fan-Out(local) x TOTAL_SIZES
LoF Comment Ratio
McCabe + CC on VPs x ALL_CTCR
McCabe + CC on VPs x COC
McCabe + CC on VPs x Feature Type(bool=1)
McCabe + CC on VPs x Feature Type(hex=1)
McCabe + CC on VPs x Feature Type(integer=1, int=1)
McCabe + CC on VPs x Feature Type(string=1)
McCabe + CC on VPs x Feature Type(tristate=1)
McCabe + CC on VPs x Feature distance
McCabe + CC on VPs x Hierarchy Levels
McCabe + CC on VPs x Hierarchy Types(0-0-1)
McCabe + CC on VPs x Hierarchy Types(0-1-0)
McCabe + CC on VPs x Hierarchy Types(1-0-0)
McCabe + CC on VPs x INCOMIG_CONNECTIONS
McCabe + CC on VPs x NUMBER_OF_CHILDREN
McCabe + CC on VPs x OUTGOING_CONNECTIONS
McCabe + CC on VPs x POSITIVE_SIZES
McCabe + CC on VPs x SD_FILE
McCabe + CC on VPs x SD_VP
McCabe + CC on VPs x TOTAL_SIZES
PLoF
PSCoF
SCoF Comment Ratio

Continued on next page

Metric
Undisciplined CPP

Table A.9: Achieved validation scores of the first set of experiments.

Model	Subset	Sampling	Precision ∅	Recall ∅	F1 ∅	F2 ∅	AUC-ROC ∅	Duration (s) Σ
RF	GPV_RFE	Off	0.977432	0.506960	0.667468	0.560897	0.959948	240
	GPV	Off	0.973382	0.501605	0.661919	0.555400	0.960251	378
GB	GPV_RFE	Off	0.878049	0.477176	0.617969	0.524988	0.910329	4516
	GPV	Off	0.874358	0.474130	0.614222	0.521666	0.906361	25937
RF	ALL_RFE	Off	0.976596	0.422435	0.589675	0.476484	0.955442	225
GB	ALL_RFE	Off	0.804594	0.437596	0.566684	0.481446	0.893432	3257
RF	GPV	Smote	0.403128	0.638015	0.493644	0.570993	0.941005	816
	ALL_RFE	Over	0.452437	0.494298	0.471758	0.484832	0.915004	457
	GPV_RFE	Smote	0.357662	0.674045	0.467152	0.572497	0.946616	557
GB	ALL	Over	0.355559	0.600401	0.446027	0.527076	0.924765	599513
RF	GPV	Over	0.349952	0.576320	0.435383	0.510206	0.928854	739
GB	ALL	Smote	0.769285	0.294049	0.425176	0.335408	0.892922	660620
		Off	0.632263	0.315633	0.420656	0.350615	0.877553	77434
RF	GPV_RFE	Over	0.326170	0.587555	0.419393	0.506300	0.928519	468
	GCV_RFE	Smote	0.417775	0.313305	0.357548	0.329538	0.734410	797
		Off	0.975537	0.214339	0.351281	0.253931	0.832393	328
GB	GPV	Over	0.230018	0.684737	0.344278	0.490570	0.926524	89266
	GPV_RFE	Over	0.229618	0.684385	0.343573	0.489677	0.927539	20269
	GPV	Smote	0.236046	0.614657	0.340838	0.464938	0.907369	102699
RF	ALL	Smote	0.700687	0.195790	0.305493	0.228605	0.859971	2655
		Over	0.659668	0.197577	0.303836	0.229701	0.890299	2544
GB	ALL_RFE	Over	0.194770	0.641946	0.298552	0.439300	0.914160	12947
RF	LPV_RFE	Off	0.891265	0.174754	0.291786	0.208138	0.744306	89
GB	LPV_RFE	Off	0.754961	0.173685	0.282226	0.205255	0.707214	545
	GCV_RFE	Off	0.615692	0.177607	0.275103	0.206916	0.704755	954
	LPV	Off	0.731670	0.169583	0.274936	0.200267	0.723002	1353
RF	ALL	Off	0.952371	0.159595	0.273214	0.191436	0.895785	1432
	LPV	Off	0.843709	0.160131	0.268632	0.190973	0.742307	120
GB	GPV_RFE	Smote	0.168128	0.652821	0.266872	0.412897	0.913820	24322
RF	GCV	Off	0.897166	0.155492	0.264803	0.186239	0.820075	1392
GB	GCV	Off	0.483383	0.170119	0.251366	0.195361	0.761028	36745
RF	LCV_RFE	Off	0.854152	0.142116	0.243438	0.170496	0.765221	210
NN	ALL	Smote	0.224262	0.262657	0.239804	0.252391	0.756442	120565
		Over	0.197283	0.300460	0.235787	0.269707	0.769569	134595
RF	LNV_RFE	Off	0.939705	0.133560	0.233659	0.161175	0.747704	104
GB	GCV_RFE	Smote	0.177617	0.330604	0.229947	0.280660	0.725932	16767
RF	GCV_RFE	Over	0.187010	0.271757	0.221257	0.248904	0.707837	678
GB	LNV_RFE	Off	0.544613	0.138196	0.220032	0.162333	0.680750	419
RF	LNV_RFE	Smote	0.185786	0.266941	0.218237	0.244697	0.687119	240
	LNV	Smote	0.339268	0.159770	0.216925	0.178566	0.705323	593
	GCV	Over	0.210084	0.208272	0.209096	0.208583	0.780776	2803
GB	GCV	Over	0.137496	0.401572	0.204723	0.289895	0.785013	505358
RF	GCV	Smote	0.201370	0.208272	0.204074	0.206404	0.758004	3020
	LCV	Off	0.845451	0.110912	0.195932	0.134203	0.765334	304
	LNV	Off	0.888343	0.107525	0.191641	0.130421	0.766042	208
GB	LCV_RFE	Off	0.438304	0.117512	0.185098	0.137601	0.719586	309
	LCV	Off	0.369107	0.111449	0.171034	0.129487	0.733784	1974
	GCV	Smote	0.114563	0.322581	0.168902	0.236332	0.744885	549651
RF	LNV	Over	0.185443	0.151034	0.166379	0.156800	0.718721	460
NN	GPV	Over	0.102617	0.403878	0.162857	0.252651	0.769259	41202
		Smote	0.104967	0.366980	0.162154	0.242439	0.767492	41870
RF	LCV	Smote	0.134114	0.200425	0.160652	0.182347	0.691292	864
GB	LNV	Off	0.371376	0.100039	0.157467	0.117118	0.726729	1186
RF	LCV	Over	0.149645	0.162799	0.155860	0.159928	0.717603	750
GB	GCV_RFE	Over	0.099189	0.346295	0.154056	0.230800	0.722874	13192
RF	LNV_RFE	Over	0.118066	0.202751	0.145772	0.173269	0.637200	228

Continued on next page

Model	Subset	Sampling	Precision ∅	Recall ∅	F1 ∅	F2 ∅	AUC-ROC ∅	Duration (s) Σ
NN	GPV_RFE	Over	0.088510	0.413871	0.145498	0.237566	0.797109	36665
GB	LNV	Smote	0.174088	0.124290	0.144533	0.131592	0.657901	26514
NN	GPV_RFE	Smote	0.086691	0.407092	0.142743	0.233509	0.791522	37023
RF	LCV_RFE	Over	0.104609	0.202205	0.137804	0.170312	0.705082	518
NN	ALL_RFE	Over	0.084154	0.375526	0.137317	0.221405	0.779656	68123
RF	LPV_RFE	Over	0.104324	0.292062	0.136367	0.175861	0.665348	188
GB	LNV_RFE	Smote	0.089577	0.286735	0.136003	0.198115	0.680684	7164
	LPV_RFE	Over	0.092723	0.347164	0.135240	0.193230	0.695734	4127
NN	ALL_RFE	Smote	0.082014	0.334694	0.131353	0.206138	0.759832	69048
	GCV	Smote	0.086052	0.249101	0.126666	0.178445	0.698008	89748
		Over	0.083450	0.262480	0.125925	0.182340	0.704995	99564
GB	LPV	Over	0.077978	0.345750	0.125251	0.198110	0.705323	12450
RF	LPV	Over	0.082672	0.261764	0.122503	0.174028	0.656243	264
GB	LCV	Over	0.074315	0.348789	0.122460	0.200423	0.718167	35253
	LNV	Over	0.073544	0.347188	0.121350	0.198982	0.723267	24281
RF	LCV_RFE	Smote	0.075277	0.299750	0.120286	0.187650	0.708530	819
NN	GCV_RFE	Smote	0.075713	0.267663	0.116851	0.175088	0.704344	37085
		Over	0.072420	0.297263	0.115897	0.181973	0.709923	45456
GB	LCV	Smote	0.073411	0.225038	0.110652	0.159149	0.661551	37265
RF	ALL_RFE	Smote	0.054766	0.779779	0.102341	0.213754	0.925762	590
GB	LCV_RFE	Over	0.057688	0.379101	0.100123	0.179246	0.727112	8156
NN	LPV	Over	0.049389	0.226282	0.080816	0.131109	0.646460	35657
		Smote	0.048815	0.221832	0.079797	0.129204	0.645304	35149
GB	LNV_RFE	Over	0.043721	0.349862	0.077550	0.145012	0.679576	5805
NN	LPV_RFE	Smote	0.047937	0.155851	0.073097	0.107017	0.602953	36006
GB	ALL_RFE	Smote	0.037950	0.861802	0.072697	0.161316	0.887932	16167
NN	LPV_RFE	Over	0.047691	0.147641	0.071843	0.103562	0.603450	33482
	LCV	Over	0.037546	0.360377	0.067812	0.131577	0.684418	35507
RF	LPV	Smote	0.038715	0.428867	0.067436	0.127821	0.674641	293
NN	LNV	Smote	0.037054	0.337734	0.066683	0.128312	0.678428	30951
		Over	0.036441	0.367507	0.066138	0.129671	0.685488	30612
GB	LCV_RFE	Smote	0.035217	0.517656	0.065922	0.138265	0.725798	14544
RF	LPV_RFE	Smote	0.037359	0.467198	0.065680	0.128180	0.694089	227
NN	LCV	Smote	0.035967	0.362889	0.065326	0.128185	0.679386	35395
	LCV_RFE	Over	0.035163	0.403702	0.064633	0.130083	0.689836	35845
	LNV_RFE	Smote	0.034426	0.388563	0.063181	0.126739	0.681050	34871
	LCV_RFE	Smote	0.034121	0.399797	0.062715	0.126369	0.684076	35220
	LNV_RFE	Over	0.033262	0.430998	0.061725	0.126893	0.688011	35224
NBG	LNV	Off	0.038435	0.125534	0.058670	0.085978	0.635658	0
GB	LPV_RFE	Smote	0.028686	0.507847	0.054304	0.116991	0.695579	4656
NBG	LNV	Over	0.030703	0.233589	0.053904	0.098955	0.635629	0
	LCV	Over	0.033221	0.157802	0.053888	0.086778	0.628518	0
GB	LPV	Smote	0.028397	0.506955	0.053782	0.115994	0.694139	13968
NBG	LNV	Smote	0.028656	0.368743	0.052582	0.105659	0.637809	0
	LPV	Off	0.037218	0.087912	0.052291	0.069083	0.578862	0
	LCV	Off	0.033758	0.103954	0.050704	0.072850	0.628482	0
		Smote	0.027195	0.291865	0.048188	0.090391	0.629725	0
	GCV	Over	0.026456	0.319511	0.047504	0.091760	0.614154	40
		Off	0.026668	0.288492	0.047233	0.088642	0.614216	20
	LPV	Over	0.026247	0.139440	0.043814	0.073694	0.579165	0
		Smote	0.022849	0.203639	0.041084	0.078835	0.580243	0
	LPV_RFE	Over	0.022636	0.124288	0.038294	0.065469	0.554135	0
		Smote	0.022192	0.126250	0.037744	0.065134	0.553004	0
	LNV_RFE	Over	0.022346	0.111271	0.037204	0.061917	0.541781	0
		Smote	0.022215	0.113588	0.037120	0.062192	0.546559	0
	GCV_RFE	Smote	0.020741	0.142478	0.036198	0.065494	0.566223	0
		Over	0.020552	0.144439	0.035983	0.065483	0.558740	0
	GPV_RFE	Off	0.021626	0.085419	0.034510	0.053715	0.556034	0

Continued on next page

Model	Subset	Sampling	Precision ∅	Recall ∅	F1 ∅	F2 ∅	AUC-ROC ∅	Duration (s) Σ
	LCV_RFE	Smote	0.017713	0.679181	0.034415	0.079326	0.597321	0
		Over	0.019986	0.200397	0.034185	0.060345	0.599294	0
	GCV_RFE	Off	0.021290	0.082206	0.033692	0.051978	0.558834	0
	LPV_RFE	Off	0.021392	0.067942	0.032536	0.047333	0.553606	0
	GPV_RFE	Over	0.015339	0.784948	0.030090	0.071132	0.555969	0
		Smote	0.015117	0.796182	0.029671	0.070250	0.555502	0
	ALL_RFE	Over	0.015029	0.824532	0.029520	0.070037	0.542592	0
		Smote	0.015016	0.830238	0.029499	0.070015	0.543878	0
		Off	0.026418	0.031921	0.028883	0.030624	0.542919	0
	ALL	Over	0.014432	0.984668	0.028447	0.068163	0.523834	85
		Off	0.014369	0.988233	0.028327	0.067898	0.512125	40
	GPV	Smote	0.014359	0.993761	0.028309	0.067872	0.501211	10
	ALL	Smote	0.014348	0.984132	0.028283	0.067786	0.500619	78
	GPV	Off	0.014344	0.997504	0.028281	0.067818	0.500123	10
		Over	0.014340	0.996969	0.028273	0.067799	0.499795	11
	GCV	Smote	0.014324	0.980208	0.028236	0.067667	0.512406	40
	LNV_RFE	Off	0.021791	0.032455	0.025523	0.028989	0.541812	0
	LCV_RFE	Off	0.022184	0.029424	0.024813	0.027148	0.598714	0

Bibliography

[AMMR93] Anand, Rangachari ; Mehrotra, Kishan G. ; Mohan, Chilukuri K. ; Ranka, Sanjay: An improved algorithm for neural network classification of imbalanced training sets. In: *IEEE Transactions on Neural Networks* 4 (1993), Nr. 6, S. 962–969

[BBM96] Basili, Victor R. ; Briand, Lionel C. ; Melo, Walcélio L: A validation of object-oriented design metrics as quality indicators. In: *IEEE Transactions on software engineering* 22 (1996), Nr. 10, S. 751–761

[BFSO84] Breiman, Leo ; Friedman, Jerome ; Stone, Charles J. ; Olshen, Richard A.: *Classification and regression trees*. CRC press, 1984

[BNG+09] Bird, Christian ; Nagappan, Nachiappan ; Gall, Harald ; Murphy, Brendan ; Devanbu, Premkumar: Putting it all together: Using socio-technical networks to predict failures. In: *2009 20th International Symposium on Software Reliability Engineering* IEEE, 2009, S. 109–119

[Bre01] Breiman, Leo: Random forests. In: *Machine learning* 45 (2001), Nr. 1, S. 5–32

[CBHK02] Chawla, Nitesh V. ; Bowyer, Kevin W. ; Hall, Lawrence O. ; Kegelmeyer, W P.: SMOTE: synthetic minority over-sampling technique. In: *Journal of artificial intelligence research* 16 (2002), S. 321–357

[CBK09] Caglayan, Bora ; Bener, Ayse ; Koch, Stefan: Merits of using repository metrics in defect prediction for open source projects. In: *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development* IEEE, 2009, S. 31–36

[CD09] Catal, Cagatay ; Diri, Banu: A systematic review of software fault prediction studies. 36 (2009), Nr. 4, 7346–7354. `http://dx.doi.org/10.1016/j.eswa.2008.10.027`. – DOI 10.1016/j.eswa.2008.10.027. – ISSN 09574174

[Cha09] Chawla, Nitesh V.: Data mining for imbalanced datasets: An overview. In: *Data mining and knowledge discovery handbook*. Springer, 2009, S. 875–886

[DP02] Denaro, Giovanni ; Pezzè, Mauro: An empirical evaluation of fault-proneness models. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002* IEEE, 2002, S. 241–251

[ESKS19] El-Sharkawy, Sascha ; Krafczyk, Adam ; Schmid, Klaus: MetricHaven - More Than 23,000 Metrics for Measuring Quality Attributes of Software Product Lines. In: *Proceedings of the 23rd International Systems and Software Product Line Conference* Bd. B, - ACM, 2019, S. 25–28

[FR12] Fortmann-Roe, Scott: Understanding the bias-variance tradeoff. In: *I: URl: http://scott. fortmann-roe. com/docs/BiasVariance. html (hämtad 2017-09-29)* (2012)

[Fri91] Friedman, Jerome H.: Multivariate adaptive regression splines. In: *The annals of statistics* (1991), S. 1–67

[Fri01] Friedman, Jerome H.: Greedy function approximation: a gradient boosting machine. In: *Annals of statistics* (2001), S. 1189–1232

[Fri02] Friedman, Jerome H.: Stochastic gradient boosting. In: *Computational statistics & data analysis* 38 (2002), Nr. 4, S. 367–378

[FS95] Freund, Yoav ; Schapire, Robert E.: A desicion-theoretic generalization of on-line learning and an application to boosting. In: *European conference on computational learning theory* Springer, 1995, S. 23–37

[FS+96] Freund, Yoav ; Schapire, Robert E. u. a.: Experiments with a new boosting algorithm. In: *icml* Bd. 96 Citeseer, 1996, S. 148–156

[GBD+10] Gray, David ; Bowes, David ; Davey, Neil ; Sun, Yi ; Christianson, Bruce: Software defect prediction using static code metrics underestimates defect-proneness. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2010. – ISBN 978–1–4244–6916–1, 1–7

[GDPG12] Giger, Emanuel ; D'Ambros, Marco ; Pinzger, Martin ; Gall, Harald C.: Method-level bug prediction. In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* ACM, 2012, S. 171–180

[GE03] Guyon, Isabelle ; Elisseeff, André: An introduction to variable and feature selection. In: *Journal of machine learning research* 3 (2003), Nr. Mar, S. 1157–1182

[GFS05] Gyimothy, Tibor ; Ferenc, Rudolf ; Siket, Istvan: Empirical validation of object-oriented metrics on open source software for fault prediction. In: *IEEE Transactions on Software engineering* 31 (2005), Nr. 10, S. 897–910

[GMSP17] Gregorutti, Baptiste ; Michel, Bertrand ; Saint-Pierre, Philippe: Correlation and variable importance in random forests. In: *Statistics and Computing* 27 (2017), Nr. 3, S. 659–678

[Gra12] Gray, David: Software Defect Prediction Using Static Code Metrics: Formulating a Methodology. (2012), S. 201

[GWBV02] Guyon, Isabelle ; Weston, Jason ; Barnhill, Stephen ; Vapnik, Vladimir: Gene selection for cancer classification using support vector machines. In: *Machine learning* 46 (2002), Nr. 1-3, S. 389–422

[H+77] Halstead, Maurice H. u. a.: *Elements of software science*. Bd. 7. Elsevier New York, 1977

[Hal99] Hall, Mark A.: Correlation-based feature selection for machine learning. (1999)

[HBB+12] Hall, T. ; Beecham, S. ; Bowes, D. ; Gray, D. ; Counsell, S.: A Systematic Literature Review on Fault Prediction Performance in Software Engineering. 38 (2012), Nr. 6, 1276–1304. `http://dx.doi.org/10.1109/TSE.2011.103`. – DOI 10.1109/TSE.2011.103. – ISSN 0098–5589, 1939–3520

[HM82] Hanley, James A. ; McNeil, Barbara J.: The meaning and use of the area under a receiver operating characteristic (ROC) curve. In: *Radiology* 143 (1982), Nr. 1, S. 29–36

[HMK12] Hata, Hideaki ; Mizuno, Osamu ; Kikuno, Tohru: Bug prediction based on fine-grained module histories. In: *Proceedings of the 34th International Conference on Software Engineering* IEEE Press, 2012, S. 200–210

[IH93] Iglewicz, Boris ; Hoaglin, David C.: *How to detect and handle outliers.* Bd. 16. Asq Press, 1993

[IS15] Ioffe, Sergey ; Szegedy, Christian: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: *arXiv preprint arXiv:1502.03167* (2015)

[JB11] Jones, Capers ; Bonsignour, Olivier: *The economics of software quality.* Addison-Wesley Professional, 2011

[JK19] Johnson, Justin M. ; Khoshgoftaar, Taghi M.: Survey on deep learning with class imbalance. In: *Journal of Big Data* 6 (2019), Nr. 1, S. 27

[JPT16] Jimenez, Matthieu ; Papadakis, Mike ; Traon, Yves L.: Vulnerability Prediction Models: A Case Study on the Linux Kernel. In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2016. – ISBN 978–1–5090–3848–0, 1–10

[KB08] Kastro, Yomi ; Bener, Ayşe Basar: A defect prediction method for software versioning. In: *Software Quality Journal* 16 (2008), Nr. 4, S. 543–562

[KB14] Kingma, Diederik P. ; Ba, Jimmy: Adam: A method for stochastic optimization. In: *arXiv preprint arXiv:1412.6980* (2014)

[KCH+90] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Novak, William E. ; Peterson, A S.: Feature-oriented domain analysis (FODA) feasibility study / Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. 1990. – Forschungsbericht

[KDPGA12] Khomh, Foutse ; Di Penta, Massimiliano ; Guéhéneuc, Yann-Gaël ; Antoniol, Giuliano: An exploratory study of the impact of antipatterns on class change-and fault-proneness. In: *Empirical Software Engineering* 17 (2012), Nr. 3, S. 243–275

[KGS10] Khoshgoftaar, Taghi M. ; Gao, Kehan ; Seliya, Naeem: Attribute selection and imbalanced data: Problems in software defect prediction. In: *2010 22nd IEEE International Conference on Tools with Artificial Intelligence* Bd. 1 IEEE, 2010, S. 137–144

[KJD02] Kang, K.C. ; Jaejoon Lee ; Donohoe, P.: Feature-oriented product line engineering. 19 (2002), Nr. 4, 58–65. `http://dx.doi.org/10.1109/MS.2002.1020288`. – DOI 10.1109/MS.2002.1020288. – ISSN 0740–7459

[KM08] Kaur, Arvinder ; Malhotra, Ruchika: Application of random forest in predicting fault-prone classes. In: *2008 International Conference on Advanced Computer Theory and Engineering* IEEE, 2008, S. 37–43

[KMM+07] Kamei, Yasutaka ; Monden, Akito ; Matsumoto, Shinsuke ; Kakimoto, Takeshi ; Matsumoto, Ken-ichi: The effects of over and under sampling on fault-prone module detection. In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)* IEEE, 2007, S. 196–204

[Kra91] Kramer, Mark A.: Nonlinear principal component analysis using autoassociative neural networks. In: *AIChE journal* 37 (1991), Nr. 2, S. 233–243

[KS04] Khoshgoftaar, Taghi M. ; Seliya, Naeem: Comparative assessment of software quality classification techniques: An empirical case study. In: *Empirical Software Engineering* 9 (2004), Nr. 3, S. 229–257

[LBMP08] Lessmann, Stefan ; Baesens, Bart ; Mues, Christophe ; Pietsch, Swantje: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. In: *IEEE Transactions on Software Engineering* 34 (2008), Nr. 4, S. 485–496

[LBOM12] LeCun, Yann A. ; Bottou, Léon ; Orr, Genevieve B. ; Müller, Klaus-Robert: Efficient backprop. In: *Neural networks: Tricks of the trade*. Springer, 2012, S. 9–48

[McC76] McCabe, Thomas J.: A complexity measure. In: *IEEE Transactions on software Engineering* (1976), Nr. 4, S. 308–320

[MGF06] Menzies, Tim ; Greenwald, Jeremy ; Frank, Art: Data mining static code attributes to learn defect predictors. In: *IEEE transactions on software engineering* 33 (2006), Nr. 1, S. 2–13

[MK10] Mende, Thilo ; Koschke, Rainer: Effort-aware defect prediction models. In: *2010 14th European Conference on Software Maintenance and Reengineering* IEEE, 2010, S. 107–116

[MMT+10] Menzies, Tim ; Milton, Zach ; Turhan, Burak ; Cukic, Bojan ; Jiang, Yue ; Bener, Ayşe: Defect prediction from static code features: current results, limitations, new approaches. In: *Automated Software Engineering* 17 (2010), Nr. 4, S. 375–407

[MPS08] Moser, Raimund ; Pedrycz, Witold ; Succi, Giancarlo: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: *Proceedings of the 30th international conference on Software engineering* ACM, 2008, S. 181–190

[MRG+17] Medeiros, Flávio ; Ribeiro, Márcio ; Gheyi, Rohit ; Apel, Sven ; Kästner, Christian ; Ferreira, Bruno ; Carvalho, Luiz ; Fonseca, Baldoino: Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. In: *IEEE Transactions on Software Engineering* 44 (2017), Nr. 5, S. 453–469

[OWB05] Ostrand, Thomas J. ; Weyuker, Elaine J. ; Bell, Robert M.: Predicting the location and number of faults in large software systems. In: *IEEE Transactions on Software Engineering* 31 (2005), Nr. 4, S. 340–355

[PD07] Pelayo, Lourdes ; Dick, Scott: Applying Novel Resampling Strategies To Software Defect Prediction. In: *NAFIPS 2007 - 2007 Annual Meeting of the North American Fuzzy Information Processing Society*, IEEE, 2007. – ISBN 978–1–4244–1213–6 978–1–4244–1214–3, 69–72

[PPB18] Pascarella, Luca ; Palomba, Fabio ; Bacchelli, Alberto: Re-evaluating method-level bug prediction. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* IEEE, 2018, S. 592–601

[Qui86] Quinlan, J. R.: Induction of decision trees. In: *Machine learning* 1 (1986), Nr. 1, S. 81–106

[Qui14] Quinlan, J R.: *C4. 5: programs for machine learning*. Elsevier, 2014

[Ros58] Rosenblatt, Frank: The perceptron: a probabilistic model for information storage and organization in the brain. In: *Psychological review* 65 (1958), Nr. 6, S. 386

[RS04] Raileanu, Laura E. ; Stoffel, Kilian: Theoretical comparison between the gini index and information gain criteria. In: *Annals of Mathematics and Artificial Intelligence* 41 (2004), Nr. 1, S. 77–93

[Rux06] Ruxton, Graeme D.: The unequal variance t-test is an underused alternative to Student's t-test and the Mann–Whitney U test. In: *Behavioral Ecology* 17 (2006), Nr. 4, S. 688–690

[SHAJ12] Shihab, Emad ; Hassan, Ahmed E. ; Adams, Bram ; Jiang, Zhen M.: An industrial study on the risk of software changes. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* ACM, 2012, S. 62

[SHK+14] Srivastava, Nitish ; Hinton, Geoffrey ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan: Dropout: a simple way to prevent neural networks from overfitting. In: *The journal of machine learning research* 15 (2014), Nr. 1, S. 1929–1958

[SMWO10] Shin, Yonghee ; Meneely, Andrew ; Williams, Laurie ; Osborne, Jason A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. In: *IEEE transactions on software engineering* 37 (2010), Nr. 6, S. 772–787

[SMWO11] Shin, Yonghee ; Meneely, Andrew ; Williams, Laurie ; Osborne, Jason A.: Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. 37 (2011), Nr. 6, 772–787. `http://dx.doi.org/10.1109/TSE.2010.81`. – DOI 10.1109/TSE.2010.81. – ISSN 0098–5589

[SWHJ14] Scandariato, Riccardo ; Walden, James ; Hovsepyan, Aram ; Joosen, Wouter: Predicting vulnerable software components via text mining. In: *IEEE Transactions on Software Engineering* 40 (2014), Nr. 10, S. 993–1006

[SWJAK09] Shivaji, Shivkumar ; Whitehead Jr, E J. ; Akella, Ram ; Kim, Sunghun: Reducing features to improve bug prediction. In: *2009 IEEE/ACM International Conference on Automated Software Engineering* IEEE, 2009, S. 600–604

[Tan15] Tan, Ming: Online Defect Prediction for Imbalanced Data. (2015), S. 51

[TBTM10] Tosun, Ayşe ; Bener, Ayşe ; Turhan, Burak ; Menzies, Tim: Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. In: *Information and Software Technology* 52 (2010), Nr. 11, S. 1242–1257

[TL11] Toloşi, Laura ; Lengauer, Thomas: Classification with correlated features: unreliability of feature ranking and solutions. In: *Bioinformatics* 27 (2011), Nr. 14, S. 1986–1994

[TMHM16] Tantithamthavorn, Chakkrit ; McIntosh, Shane ; Hassan, Ahmed E. ; Matsumoto, Kenichi: An empirical comparison of model validation techniques for defect prediction models. In: *IEEE Transactions on Software Engineering* 43 (2016), Nr. 1, S. 1–18

[VLBM08] Vincent, Pascal ; Larochelle, Hugo ; Bengio, Yoshua ; Manzagol, Pierre-Antoine: Extracting and composing robust features with denoising autoencoders. In: *Proceedings of the 25th international conference on Machine learning*, 2008, S. 1096–1103

[VLL+10] Vincent, Pascal ; Larochelle, Hugo ; Lajoie, Isabelle ; Bengio, Yoshua ; Manzagol, Pierre-Antoine: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. In: *Journal of machine learning research* 11 (2010), Nr. Dec, S. 3371–3408

[WLT16] Wang, Song ; Liu, Taiyue ; Tan, Lin: Automatically learning semantic features for defect prediction. In: *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, ACM Press, 2016. – ISBN 978–1–4503–3900–1, 297–308

[WSS14] Walden, James ; Stuckman, Jeff ; Scandariato, Riccardo: Predicting vulnerable components: Software metrics vs text mining. In: *2014 IEEE 25th international symposium on software reliability engineering* IEEE, 2014, S. 23–33

[YLX+15] Yang, Xinli ; Lo, David ; Xia, Xin ; Zhang, Yun ; Sun, Jianling: Deep Learning for Just-in-Time Defect Prediction. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*, IEEE, 2015. – ISBN 978–1–4673–7989–2, 17–26

[ZPZ07] Zimmermann, Thomas ; Premraj, Rahul ; Zeller, Andreas: Predicting defects for eclipse. In: *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)* IEEE, 2007, S. 9–9