Intelligent Code Inspection using Static Code Features
An approach for Java

Irene Moriggl

School of Engineering
Blekinge Institute of Technology
Box 520
SE - 372 25 Ronneby
Sweden
This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Irene Moriggl 860404-P496
E-mail: irmo09@student.bth.se

University advisor(s):
Dr. Stefan Axelsson
School of Computing

School of Engineering
Blekinge Institute of Technology
Box 520
SE - 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : +46 457 271 25
Effective defect detection is still a hot issue when it comes to software quality assurance. Static source code analysis plays thereby an important role, since it offers the possibility for automated defect detection in early stages of the development. As detecting defects can be seen as a classification problem, machine learning is recently investigated to be used for this purpose. This study presents a new model for automated defect detection by means of machine learners based on static Java code features. The model comprises the extraction of necessary features as well as the application of suitable classifiers to them. It is realized by a prototype for the feature extraction and a study on the prototype’s output in order to identify the most suitable classifiers. Finally, the overall approach is evaluated in a using an open source project. The suitability study and the evaluation show, that several classifiers are suitable for the model and that the Rotation Forest, Multilayer Perceptron and the JRip classifier make the approach most effective. They detect defects with an accuracy higher than 96%. Although the approach comprises only a prototype, it shows the potential to become an effective alternative to nowadays defect detection methods.

**Keywords:** Java, Static Source Code Analysis, Machine Learning, Automated Defect Detection.
Contents

Abstract i

1 Introduction 1

2 Background 2
  2.1 Software Quality and Static Source Code Analysis ............ 2
  2.2 Machine Learning ........................................ 3
  2.3 Learning on Source Code Properties .......................... 4
  2.4 JavaCC - the Java Compiler Compiler ....................... 5
  2.5 WEKA Framework and ARFF ................................. 5

3 Related Work 8
  3.1 Defect Prediction Models ................................. 8
  3.2 Defect Detection Models .................................. 9

4 Problem Description 10

5 JJParse Prototyping 12
  5.1 Methodology .............................................. 12
  5.2 Prototype Implementation ................................. 12
    5.2.1 The Choice of WEKA ............................... 12
    5.2.2 Code Representation ............................... 13
    5.2.3 Reduced Feature Set ............................... 15
    5.2.4 Focused Faults ................................... 15
    5.2.5 JJParse .......................................... 16

6 Suitability Study and Evaluation 19
  6.1 Design ..................................................... 19
  6.2 Settings ................................................... 20
    6.2.1 Instance Generation for Faulty Java Operations ........ 20
  6.3 Suitability Study ........................................ 21
    6.3.1 Measures ............................................ 21
    6.3.2 Experiment Execution ............................... 21
    6.3.3 Results and Observations ........................... 22
List of Figures

5.1 Methodology ......................................................... 13
5.2 Simplified Execution Flow of JJParse ............................. 18

6.1 Average Accuracy per Category ................................. 22
6.2 Best Performing Machine Learners w.r.t Accuracy ............. 23
6.3 Average FP Rate per Category ..................................... 23
6.4 Best Performing Machine Learners w.r.t FP Rate ............... 24
6.5 Average FN Rate per Category ..................................... 25
6.6 Best Performing Machine Learners w.r.t FN Rate ............... 26
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Statistical Significance of the Learners’ ACC</td>
<td>27</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance of the Classifiers on the Test Set</td>
<td>28</td>
</tr>
<tr>
<td>A.1</td>
<td>Performance of All Used Machine Learners from the Suitability Experiment</td>
<td>36</td>
</tr>
<tr>
<td>B.1</td>
<td>Choice of the Most Suitable Learners for C Code</td>
<td>37</td>
</tr>
<tr>
<td>B.2</td>
<td>Evaluation of the Most Suitable Learners on the Test Set from C Code</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Faults exist since the first software system was written. Since then, developers try to catch as many bugs as possible to get their systems right. During the past decades several methodologies and techniques have been proposed to facilitate fault detection. Recently, machine learning gets more and more attention for this purpose. Combined with source code analysis it is supposed to provide effective defect prediction and detection models. From the former are known already a bunch of models which can compete with traditional prediction approaches. The latter instead are still under development.

The idea behind machine learning based defect detection is the classification of code into faulty or correct. The classification is done based on a set of already classified code examples. The advantage compared to the common pattern matching approach is a classifier’s ability to learn, i.e. its ability to generalize from examples to similar cases.

The purpose of this study is to provide and assess such a machine learning based defect detection approach for Java source code. It shall show that the underlying model is feasible and that the approach can be effective using machine learners. Furthermore, it shall show that machine learning in general is worth to be done in the field of source code analysis.

So the basic idea is to statically analyze Java source code by means of a newly implemented prototype. It shall describe the Java code at a certain granularity in terms of code features. Existing machine learners shall then be used to classify the code based on these extracted features. Unfortunately, not all existing machine learners perform equally well on a certain feature set. So their suitability must be assessed first. Having the most suitable ones, the entire approach shall finally be assessed for its efficiency.

The following two chapters provide the necessary background and related works to this study. The aim and objectives together with the research questions are provided in the problem description. This follows the proposed solution with details about the used research methodology and about the implemented prototype. Thereafter, the suitability assessment and the evaluation of the presented defect detection approach are described. And after the limitations and future works, the study is concluded.
Chapter 2

Background

First of all, this work is supposed to contribute to a software’s quality by providing an alternative to today’s static source code analysis approaches. The enhancement is supposed to be realized by the application of machine learning techniques to code features. Since the work’s contribution takes the form of a prototype for code feature extraction on Java code, some existing tools have been used for its implementation and evaluation: JavaCC and WEKA. More details on the underlying concepts and used tools follow in the sections of this chapter.

2.1 Software Quality and Static Source Code Analysis

Developing high quality software is desired but even for experts it is a complex task. Each stage in the development life cycle has its own difficulties which have to be mastered in order to achieve the goal. The testing phase of the life cycle can thereby require most effort. The amount of code written for testing purposes can easily comprise more than 1/2 of the total code base. And usually do developers spend much more time in quality assurance than planned. So it can make up 30-60 percent of the overall development time.

Obviously, the problem faced during the testing phase is buggy source code. Besides missing or wrong functionalities the problem comprises the presence of faults. The latter category is probably the most tricky one. Faults left in the code cause the software to perform badly or to crash. Unfortunately, they can be well hidden and the complete absence of such faults can never be proved as the computational theory and its halting problem[33] shows. However, analyzing and testing the software in an appropriate way can guarantee an acceptable quality level. There are many techniques for doing so - black box or white box, from unit to system testing. The description of all would certainly go beyond the scope of this work. Therefore it is limited to the approach upon which this work is based, namely static source code analysis.

Static source code analysis has been introduced about 30 years ago by M. E. Fagan as part of his software inspection process [7][8]. It falls into the cate-
Chapter 2. Background

gory of white box testing approaches, where testers require insights into a software’s source code. Originally, during code inspection, experts or developers read through the code and try to manually identify as many faults as possible. Those faults are then recorded for fixing. The advantage of static analysis is its early applicability since it does not require the software to run. But it is time consuming and usually annoying. Therefore it is nowadays automated by static analysis tools. Generally, there are two main types of tools. There are tools, which analyze code to compute metrics as fault indicators like Understand [30]. Then there are tools like FindBugs [25], which try to directly detect faults.

Such tools make usually use of pattern matching for identifying faults in the source code or in its binary format. Each tool has its own collection of rules describing their fault patterns. A very common pattern is for instance the use of the assignment operator instead of a relational one in an if condition; or simply the use of an uninitialized variable. While the tools analyse the code, they search for occurrences of such patterns. Each match is then reported as a problem. The analysis can be done in different ways. Common approaches are control flow analysis, data flow analysis and analysis through finite state machines.

Although many static analysis tools are used in practice, they have pitfalls in reporting problems. On one hand, they identify faults even if there may not be any. On the other hand, they may not report a fault at all. The former are called false positives, the latter false negatives. Usually the tools trade a lower false positive rate for a higher false negative rate.

2.2 Machine Learning

The dream of creating humanly thinking, intelligent machines was already dreamed in ancient history and since it attracts many. A machine can already be seen to be intelligent, if it is able to react to a given input in the correct way, even though it has never seen exactly that input before. The correct behaviour depends on the problem to be solved. It can be the classification, association or clustering of the input. The input is also called instance and is described by a set of properties. Depending on these properties, a machine learning algorithm is able to solve the instance specific problem and allows the machine to react accordingly. As with humans, learning is the key factor to achieve such an intelligence. Learning or training\textsuperscript{1} a machine can be done in two different ways: supervised or unsupervised.

Supervised learning is characterized by providing input/output pairs and is used for classification problems. The machine learner gets several instances as input together with their corresponding output class. Using those pairs, a so called learning scheme is produced to classify new instances. A learning scheme

\textsuperscript{1}Some prefer the one some the other term; but within this work they are meant to be synonyms in the context of machine learning
Chapter 2. Background

tells how new instances are classified. It can consist for instance of a mathematical
function or of a set of rules. Unsupervised learning instead is used for clustering.
The machine learner gets just the instances and produces a cluster based on
similarities of the instance properties.

Furthermore, a machine learner can be trained in two different ways: either
by batch learning or by active learning. In the former case, only a predefined
set of instances, called training set, is used to establish the classifier. Instead in
the latter case feedback from each instance is used to continuously update the
classifier. Which approach can be used or is more suitable depends highly on the
machine learning algorithm.

All approaches, supervised or unsupervised with batch or with active learning,
make use of different machine learners with their own weaknesses and strengths.
What is called a machine learner is nothing else than an algorithm which builds
and uses a learning scheme to solve a particular problem. Many have been im-
plemented during the last decades and are well known in fields like game playing,
speech recognition, computer vision or expert systems. Only supervised learners
are of interest for this work, because it tries to solve the defect detection problem
by classification rather than by clustering. And for now, only batch learning is
used, leaving active learning for future work, since it can be based on the former.
Moreover, this work makes just use of existing machine learners without concern-
ing much of their details. So for more details on how the learners exactly work
please refer to the comprehensive book referred by [40].

As it is the case with static code analysis tools, the machine learners are
mainly evaluated in literature by measuring their accuracy, false positive and
false negative rates.

2.3 Learning on Source Code Properties

The success of machine learning leads to its application in more and more re-
search areas. So it comes also to the field of quality assurance, especially source
code analysis. First, machine learners are just used to establish defect prediction
models. Then, more recently, they are also used for establishing defect detection
models. Both types rely on the extraction of some kind of code properties, upon
which machine learners can be executed.

The prediction models are usually based on so called software metrics. These
metrics are mathematically computed during the static code analysis and measure
quantitatively a property of the code. It is believed, that these metrics charac-
terize code modules like a class or a function so that they can be classified into
fault prone or not [28]. Frequently used metrics are for instance lines of code
and Cyclomatic Complexity. Models based on these kind of code properties try
to guess, where faults may be hidden in the code in order to allow a better re-
source distribution when it comes to testing the whole software. Defect detection
models instead try to directly identify faults in the code. By making use of machine learners to classify code parts, the process is supposed to be more effective regarding accuracy, false positives and false negatives.

2.4 JavaCC - the Java Compiler Compiler

Since this work is based on statically analyzing Java source code, there is a need for parsing it in order to extract the necessary information. The Java Compiler Compiler has become a popular tool for generating parsers and seems to be the right thing. Based on a grammar specification it generates a Java program, which is able to recognize implementations of the specified grammar.

A grammar specification describes the syntax of a language using parsing rules, terminal symbols and nonterminal symbols [15]. Parsing rules are constructed using nonterminal symbols on the left hand side and terminal and/or nonterminal symbols on the right hand side. JavaCC converts each parsing rule into a Java method, which handles the right hand side of the rule. Thus, all JavaCC generated parsers proceed in a top down fashion. In contrast, bottom up approaches would try to reduce the right hand side symbols to their nonterminal on the left hand side.

Besides parser generation, JavaCC also provides other useful facilities like tree or documentation generators as described in detail in [4]. However, here it is just used to generate a parser from an adapted Java language specification.

2.5 WEKA Framework and ARFF

The Waikato Environment for Knowledge Analysis (WEKA) framework provides machine learning algorithms and data processing tools. It makes it easy to apply state-of-the-art machine learning practices to given datasets. It allows to pre-process and visualize datasets, compare different machine learning techniques and statistically evaluate the performance of machine learners on any given dataset [40].

In order to realize all these features, a format for representing datasets in a standardized manner has been introduced by the creators of the WEKA framework. It is called attribute relation file format, short ARFF. In an ARFF, the instances of a dataset are represented by rows, whereas the features describing the instances are represented by columns. The preamble of the ARFF lists the features and their format. The format can either be a set of nominal values, numeric or string. However, there are some machine learners not accepting numeric or string attributes. Usually such learners can not be used at all or ignore the corresponding attributes. For these cases, WEKA provides tools for the discretization of numeric values or for filtering values from the dataset. Thanks to
Chapter 2. Background

the ARFF and pre-processing tools, it is possible to run the different machine learners on the same dataset and thus to compare their performances.

The WEKA framework was developed at the University of Waikato in New Zealand. Its main purpose is to provide a comprehensive collection of data mining techniques for learning and experimenting on datasets. But it also offers a command line interface, through which it could be integrated into real-life programs. The provided data mining tools comprise regression, classification, clustering, association rule mining, and attribute selection. For each category, implementations of different algorithms can be found. Most important to mention are the classification algorithms covered by the framework. Some classifiers are widely used in the field of machine learning. The following listing comes from the WEKA classifier categorization as it is described in [40]. This categorization is used also within this work:

**Bayesian:** e.g. Naïve Bayes, Bayesian Networks; these learners are based on mathematical expressions for conditional probabilities. It is supposed, that these learners can work well with missing data. However, some Bayesian learners assume that the features describing an instance are independent.

**Trees:** e.g. J4.8, Best First Tree; as the category name indicates, these learners use decision trees as learning scheme. The built trees can be pruned for performance reasons. The nodes of the tree represent the features upon which the learner decides. Here probabilities can be introduced. The leaves instead represent the output class. In some cases, the leaves are exchanged by further classification algorithms as it is the case for instance for the Naïve Bayes Tree (NBTree) or the Classification And Regression Tree (CART).

**Rules:** e.g. ZeroRule, DecisionTable; these learners instead compute a set of rules, through which instances can be classified. Decision trees can usually be represented also as a set of rules.

**Functions:** e.g. Multilayer Perceptron, Linear Regression; all these learners use mathematical equations (other than those for conditional probabilities) in order to build their learning schemes.

**Lazy:** e.g. IB1, KStar; in this category fall the nearest neighbour algorithms. They do not directly build any learning scheme. Instead, they store the training instances. When it comes to the classification of new instances, a distance function to the training instances is computed. The output class is then the one of the nearest neighbours. This approach is also known as instance based classification.

**Misc:** in this category only two classifiers are present: HyperPipes and Voting Feature Intervals (VFI). Both are similar to the lazy classifiers with the difference that they compress instances into regions or intervals. This makes them much faster but also much more
approximate.

**Meta:** e.g. Bagging, AdaBoostM1; all these learners make use of traditional learners as described above and enhance them. The enhancement usually takes place by training and running different classifiers in parallel and combining then their results.
Chapter 3

Related Work

Both, source code analysis and machine learning, are well studied research areas for themselves. But the combination of both was introduced only recently as the a priori performed systematic literature review shows (see appendix C). The following sections present the most important studies, which deal with this newly created field of study. The first section talks about defect prediction models, the second about defect detection models.

3.1 Defect Prediction Models

Initially, pure mathematical models are used for predicting faulty code modules from software metrics. But then in 1999, Fenton et al. suggests in [9] Bayesian Belief Networks for defect prediction based on software metrics in order to get more generic prediction models. From then on, several studies like [38] are conducted. They all show that a variety of machine learners are well suited for establishing accurate prediction models.

The studies [6], [11] and [36] investigate Case Based Reasoning (CBR). The latter compares CBR to other techniques and shows on a military system, that it can obtain along with a tree learner the best results.

The studies [41], [32] and [12] focus on Artificial Neural Networks (ANNs). The first study successfully applies ANNs to the NASA project KC1. The second study compares the new ANN solution of the authors to traditional approaches. And the third study compares ANNs to Support Vector Machines (SVM). Both approaches obtain good results: 72% and 87% accuracy respectively. To a similar result for SVMs comes also the study presented in [5] while applying SVMs to 4 NASA datasets written in C, C++ or Java.

Tree approaches instead are investigated by the studies [13] and [39]. The former uses random forests and gets generally an accuracy of more than 75%; in most cases however more than 85%. The latter gets similar results by using decision trees.

Khoshgoftaat and Seliya compared many learners based on metrics from 4 releases of a large legacy telecommunication system. In [19] the authors compare 6 commonly used approaches. Among them also a CBR technique, an ANN and
two tree approaches. The winning technique regarding accuracy is here one of the tree approaches. In [20] instead, they repeat the study with slightly different learners. But in this second study, the main focus is on the investigation of the cost of misclassification, i.e. the cost of false positives and false negatives. They obtain the result that the expected cost of misclassification alone does not allow them to rank the compared approaches. Decision trees along with Naive Bayes and 5 other machine learning techniques are used also by Lounies et al. in [23] in order to verify that software metrics can predict software quality.

All the mentioned studies do not make a distinction between code and design metrics. [18] and [31] investigate the performance of machine learners by taking this distinction into account. [24] and [14] consider change metrics as well. And [27] uses even measurements from the inspection process (total number of faults found during inspection, minimum/maximum/average number of faults found by each reviewer, standard deviation of each reviewer’s findings, and so on) with the slightly different goal to predict the number of remaining faults after an inspection.

3.2 Defect Detection Models

Besides building fault prediction models, researchers recently began to build detection models using machine learning algorithms. The case study [3] presents a dynamic code analysis approach for C and Java code in which faulty code and its correct versions are trained. The authors use SVMs and trees to discover faults in new running code.

Kreimer instead uses a static analysis approach in [21] to detect design faults in Java code. The trained tree learner is based on design metrics and correctly identifies design faults with an accuracy of about 90%. Song et al. also use static analysis in [34] but they try to identify patterns, which are similar to previously found faults. For doing so, they use an association rule mining (ARM) algorithm. Based on the found patterns, they can predict defects with an accuracy of 96.6%. However, their main goal in this study is to predict the effort for correcting the faults.

Instead, a neural networks approach for detecting defects is presented in [17]. While evaluating this approach using a realistic stock trading system, the detection reaches an accuracy of 95%. Finally, in [1] Kacan and Sidlauskas present a prototype, which is able to identify faults in C code by making use of a distance function in combination with a k-nearest neighbour algorithm. This prototype transforms source code into text upon which the distance function is computed for classifying the code. The results given by the prototype were surprising and in a further step they could be validated. As far as known, this prototype represents the most similar work to the one presented here.
Chapter 4

Problem Description

Nowadays automated source code analysis has become very valuable and is performed in practice. But its generally high false positive and false negative rates in defect detection limit the trust in the analysis. In certain cases the false alarms can even lead to the insertion of additional faults as shown by Baca in [2]. Such problems are tried to be overcome by the application of machine learning to source code analysis.

Initial successful tries have already been made as pointed out in chapter 3. However, most of the studies deal with defect prediction. And there are only few defect detection approaches based on static source code analysis and machine learning. Thus, further investigation is required in order to come up with possible alternative approaches making the static source code analysis as effective as possible.

This work is therefore intended to present such an alternative approach for the Java programming language. It shall show, that the given approach is feasible by providing a prototype implementation for appropriately parsing the Java code and applying machine learning algorithms to the prototype’s output. It shall show, that the approach has the potential to become an effective defect detector by evidencing a high accuracy and at the same time an acceptable false positive and false negative rate for at least one machine learner. Meeting these goals, the study shall also implicitly strengthen the belief that machine learning can effectively be used for fault detection.

So to realize these goals, a prototype for extracting code properties is implemented. Then, for a set of faults, the prototype is executed on open source projects. Subsequently, several machine learners are trained and evaluated on the prototype’s data to get the most applicable ones with respect to the Java properties.

In order to make sure, that these objectives are achieved, the following research questions need to be answered within the context of this work.

1. Can Java code features be selected for an effective machine learning and how?
Chapter 4. Problem Description

(a) Of which form are these code features?
(b) What are the main features classifying code into faulty or not?

2. How can the Java code features be transformed and represented in a learner’s input format?

3. What are the most effective machine learners for these Java code features and the resulting model?

(a) What is their accuracy?
(b) What is their false positive rate?
(c) What is their false negative rate?
This chapter presents the methodology used for this work as well as details on the prototype implementation for the Java feature extraction.

5.1 Methodology

As an a priori work and first step toward a new model for performing defect detection through machine learning, a systematic literature review was conducted. The aim was to generally assess the applicability of machine learning algorithms to source code analysis (see appendix C).

Based on the results of the review, the necessary insights could be gathered to define the objectives and research questions of this study as they are given in chapter 4. Starting from them, the prototype for extracting features from source code was implemented. While doing so, the focus was on a subset of possible faults which are described in the subsequent section.

Then a study was designed and performed in order to assess the suitability of several machine learners based on a training set with extracted features from the prototype. In this step, also the need for some code properties was investigated by comparing the learners’ performance on the whole and on a reduced feature set. After that, the here presented defect detection approach is evaluated using an independent test set. Figure 5.1 sketches the performed steps.

5.2 Prototype Implementation

During prototyping, some important choices were made. These are explained in the following subsections. Then the prototype itself is described.

5.2.1 The Choice of WEKA

As already mentioned, the aim of this work is not to implement own machine learners but to exploit the power of the existing ones for effective defect classification. So, one of the first questions was where to get the appropriate machine
learners. While performing the systematic literature review, it was noticed, that researchers frequently used WEKA for the machine learning purposes. WEKA is a reliable framework and offers a large amount of classifiers. Most of the classifiers encountered during the systematic literature review are available and preconfigured. Moreover, the framework is open source and can be easily downloaded from [26]. These and the advantages given in the section 2.5 makes WEKA well suited for this work.

5.2.2 Code Representation

As WEKA’s classifiers work with ARFF files, the output of the prototype is of this form, too. This makes the interoperability with the framework much easier. But this implies also, that each code instance is represented as one row only. So, properties describing code instances must be found, extracted and put into the resulting ARFF file as such. Only then the machine learners are able to classify the code instances in terms of rows as faulty or not.

Now, what are these code instances? To keep the prototype as intuitive as possible, a code instance is a collection of properties describing a Java operation. Such operations are for example variable assignments, method calls and relational or mathematical expressions. Of course do different operations have different properties. But the result of the prototype is one single ARFF file, where all instances must be described by the same set of properties. Consequently, the
union of the properties per operation is taken and properties which do not apply for a certain operation type are marked as missing for all its instances.

In total the prototype extracts more than 150 properties per operation. These are roughly composed as follows:

**trace:** name of the Java file and line number to locate the operation (if it is a fault, it can be highlighted as such in future works)

**control:** looping or branching information

**main variable:** name, access modifiers, type, status information of the main variable

**aux. variable:** name, access modifiers, type, status information of an auxiliary variable

**operation:** type of the operation, operator

**method:** name, access modifiers, parameter information

**parameters:** name, access modifiers, type, status information of the first 3 parameters passed to a method

Listing 5.1 shows an excerpt of a final ARFF file produced by the prototype. It starts with the definition of the code features. Then follow the code instances. This final ARFF file holds the standard feature set to which the WEKA machine learners are applied.

```plaintext
@RELATION code

@attribute Line numeric
@attribute Time { end, middle }
@attribute ControlType { none, looping, branching, [...] }
@attribute Control { none, for, if, else, switch, while, [...] }
@attribute V1_ID { si, stations, remoteManager, [...] }
@attribute V1_Access { public, protected, private, default }
@attribute V1_Static { true, false }
@attribute V1_Final { true, false }
@attribute V1_Operation { none, assignment, relational, math, [...] }
@attribute V1_Operator { none, =, !=, <, post-−−, −−, &&, ||, [...] }
@attribute V2_ID { length, i, #literal, [...] }
@attribute V2_Access { public, protected, private, default }
@attribute V2_Static { true, false }
@attribute V2_Final { true, false }
@attribute MethName { close, add, equals, [...] }
@attribute M_Access { public, private, protected, default }
@attribute M_Static { true, false }
@attribute M_Abstract { true, false }
@attribute P1_ID { #mp, upload_limiter, b64Key, [...] }
@attribute P1_Access { public, private, protected, default }
@attribute P1_Static { true, false }
@attribute P1_Final { true, false }

@DATA
159,middle,none,manager,default,false,false[..],member_access[..],7,7,7,7,7[..],addRateLimiter,public,false,false[..],upload_limiter,private,false,false[..]
...
345,middle,branching,if,if,if,if,else,switch,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,while,While
5.2.3 Reduced Feature Set

From the literature about machine learning it is known, that the crucial point is the feature set on which a learner is trained. A learner’s performance is tightly related to the quality of the feature set. On one hand, missing data and irrelevant features can easily confuse a learner so that it classifies instances incorrectly. On the other hand, the more features are in the set the longer it may take to build the learning scheme and to classify an instance [40].

There are of course some learners, which are somehow resistant to missing data or irrelevant features or which perform still fast with a large amount of features. But those might not always be the best ones. Therefore, a reduced feature set is considered besides the standard one when it comes to the suitability study and evaluation of the presented approach.

In the reduced set, possibly irrelevant features should be eliminated. So first of all file names and line numbers are dismissed when it comes to the classification process (they are also dismissed in the standard set, since the trace information does not matter at all at this stage). Also the names of the main and auxiliary variables and of the 3 parameters should never be a reason for classifying an instance. Similarly the real types of the parameters may be dismissed, although in certain circumstances they could be of interest for future works.

Besides the explicitly mentioned features, some others are dismissed as well so that the reduced set contains only 127 relevant features\(^1\). This reduction should also be enough to note slight improvements in execution time. So only the problem of missing data remains within this study and should be tackled in future works.

5.2.4 Focused Faults

Up to now, the set of properties describing the code instances is aligned to a subset of possible faults. For further faults it may be, that the set must be expanded. The most annoying faults are definitely those, which cause a running program to crash. So primarily faults rising a runtime exception are chosen. To be precise, those possibly causing null pointer, class cast, out of memory and index out of bounds exceptions. Moreover there are considered two common mistakes in Java, which even experts can carelessly introduce in programs. Firstly, the mismatch between the method call equals and the relational operators == or !=. Secondly, reassigning variables without using their old value. These two fault types are also tracked by defect detectors like FindBugs [25] and seem thus to be important.

\(^1\)In future works, the reduction of more features could be an option to determine the minimal feature set.
Special Case: Null Pointer Exceptions

Null pointer exceptions are by far the most common runtime errors in Java programs. They can rise in many different situations. This fault category is divided into 3 subcategories:

1. Accessing a member on a variable, which has not been initialized yet.

2. Accessing a member on a variable, which may be null but is not explicitly checked for being so.

3. Adding a variable, which is or may be null, to a collection.

The result is in either case the same, but the representation of the fault in the ARFF file differs from case to case. Therefore they are treated as separate types of faults.

5.2.5 JJParse

The prototype for extracting the code properties is called JJParse. It is written in Java. The code parser itself is generated using JavaCC, which was downloaded from [16]. The used Java language grammar is an adapted version of the open source grammar available at [10]. Originally, the grammar allowed the parser just to go through the code and to check for syntactical errors. The listing 5.2 shows an excerpt of how the grammar is adapted to extract the needed code properties for the Java operations. More precisely, it shows the grammar for parsing a literal and how thereby the corresponding variable is constructed. Everything between curly brackets are adaptations made to the grammar. The same is true for the return values of the methods. The JavaCC compiler generates from these 3 definitions 3 methods of the parser.

```java
Variable Literal(): {
    String val = null;
    Variable var = new Variable();
    var.id = "#literal";
    var.status = Status.INITIALIZED;
    var.operation = Operation.NONE;
    var.isLiteral = true;
}

    |<INTEGER_LITERAL>( var.newval = token.image; var.type = Type.INT; )
    |<FLOATING_POINT_LITERAL>( var.newval = token.image; var.type = Type.FLOAT; )
    |<CHARACTER_LITERAL>( var.newval = token.image; var.type = Type.CHAR; )
    |<STRING_LITERAL>( var.newval = token.image; var.type = Type.STRING; )
    |val = BooleanLiteral() ( var.newval = val; var.type = Type.BOOLEAN; )
    |val = NullLiteral() ( var.newval = val; var.type = Type.NULL_LITERAL; )
}

    var.oldval = var.newval;
    return var;
}

String BooleanLiteral(): {
{
    "true" | "false"
    return token.image;
}
}
```
JJParse starts from a java file and a method. It works in two steps. First, the given file is pre-parsed. This means, the imports are resolved and all discovered, user defined classes together with their instance variables and methods are registered in the JJParse’s core class. For doing so, JJParse creates and populates the data structures for classes, methods and variables while pre-parsing the files. So they can be found whenever they are needed to be updated.

Then in the second step, the method is parsed. While parsing, JJParse follows the execution flow in order to update the status information of the involved variables. So, whenever a method call is encountered, JJParse resolves the corresponding class, searches for the method and parses it before continuing on the other one. Furthermore, instance variables and ancestor instance variables are resolved in order to always have the updated status information. If class members can not be found, as it is the case for those coming from external libraries, they are treated as unknown.

Whenever a Java operation is completed, its describing properties are combined to a comma separated line and appended to the output file. After completion of parsing the initial method, the properties of all static variables are appended to the output file as well. This is done in order to identify memory problems, which could lead to the out of memory exceptions.

Null pointer and reassignment problems instead can be identified thanks to JJParse’s tracking of the status and the checks of variables. The extracted type information helps to identify possible class cast exceptions and together with operation information it helps also to identify equals/"==" problems.

Although JJParse follows the execution flow, it is very hard to work with the size of arrays and collections. So there was a need to find an easier way for the prototype to identify index out of bound problems. The solution to this is up to now, that not the size itself is part of the outputted code features, but whether the code checks the size before accessing an array member.

Besides these fault describing properties, there are several others outputted for each operation. There would even be the possibility to address semantic faults which a compiler usually checks (e.g. reassigning to a variable which is declared as final).

The figure 5.2 roughly sketches how JJParse works.
Figure 5.2: Simplified Execution Flow of JJParse
Chapter 6

Suitability Study and Evaluation

This chapter describes the suitability study and the evaluation of the presented approach. The former assesses the suitability of the machine learners based on the prototype’s code features. The latter evaluates the prototype together with the best performing learners to see, how good the approach performs on realistic data. Both make use of WEKA to facilitate the execution and to the introduction of careless mistakes as much as possible.

6.1 Design

Since this work can be seen as a feasibility study, a one shot case study should be enough for the suitability assessment and the evaluation in order to show, that the presented approach can work. A one shot case study is not an experiment in the proper sense, because it lacks of randomization. It rather belongs to the pre-experimental studies. It consists of a treatment and an observation of some subjects to determine an effect. There is neither an observation before the treatment, nor a control group. The table 6.1 shows how the general design is mapped in this case.

\[
\begin{array}{ccc}
X &=& \text{Treatment} \\
O &=& \text{Observation} \\
\text{Subjects} &=& \text{Operations (Faulty/Correct Code Instances)}
\end{array}
\]

Table 6.1: One Shot Case Study

The advantage of a control group would be the possibility to determine, whether the effect really comes from the treatment, and not from the evolution of the subjects over time (aka maturation). But since the subjects in this case are Java operations represented by appropriate machine learning instances and the latter are not changed once added to the training or test sets, the maturation is assumed not to be a problem.
6.2 Settings

For both tasks, suitability assessment and evaluation, a training set with code instances was needed to train the machine learners. Up to now, the training set contains about 650 distinct instances. These instances come from 5 open source Java projects. Their source code was downloaded from sourceforge [35]. The projects are of different size and belong to different domains:

1. Vuze (Azureus) - P2P file sharing client; about 254,000 weekly downloads;
2. SweetHome3D - Desktop application for interior design with 3D preview; about 50,000 weekly downloads;
3. FullSync - File synchronization application; about 800 weekly downloads;
4. JWhoisServer 0.3.2.1 - Whois lookup and domain name search server; about 50 weekly downloads;
5. TrainPlanner - Mobile application for finding public transport routes; 1 weekly download;

Vuze was chosen, because of its popularity and size. Furthermore it provided access to the commit logs. The other projects were chosen without having any special relevance for the work. They were the first suitable projects to be used in combination with the prototype implementation of this work.

6.2.1 Instance Generation for Faulty Java Operations

Besides correct Java code, the training set must contain also faulty code examples. Vuze provides access to the commit logs [22]. So, in order to get realistic faulty code instances from Vuze, these commit logs were investigated. On the other projects fault injection was performed. Thereby served the commit log faults as guidelines to keep the injected faults also realistic. In each project one fault was injected for each fault category presented in 5.2.4. For each faulty code instance, the corresponding correct version was extracted as well by running the prototype twice: once on the original code and once after the fault injection. Then the two output files were automatically compared so that only the distinct code instances remained. From them, the corresponding instances were taken, respectively marked as faulty and correct and added to the training set. This procedure was followed for each single fault to avoid that the faults influence each other.

After collecting the pairs of faulty and correct code instances from the projects, they were duplicated and slightly changed to new instances by replacing values of some properties to other values which still make sense. The idea behind this duplication is to make a learner more sensitive to the properties which really matter to classify an instance as faulty.
6.3 Suitability Study

The probably de facto standard way of assessing a machine learner’s predictive performance on a limited data set is n-fold cross validation. At this, the data set is randomly partitioned into n subsets. n-1 subsets are used to train the classifier, 1 subset - the so called holdout - to test it. This is done n times, so that each subset is used once for testing. The computed performance measures are then averaged over the n runs. This study uses 10-fold cross validation as it is commonly used in the machine learning research area. About the correct number of folds is still discussed, but “extensive studies on numerous datasets, with different learning techniques, have shown that 10 is about the right number of folds to get the best estimates for the predictive performance” [40]. And although the random partitioning represents a possible bottleneck what procedures like leave one out cross validation could overcome, it is still preferred here, because it is affordable w.r.t. the computational expense and mitigates the possible bias in the representativeness of faulty and non faulty instances in the holdout.

6.3.1 Measures

According to the a priori performed systematic literature review are the following three performance measures most used:

- **Accuracy**: The percentage of correctly classified instances w.r.t the total number of instances.
- **False Positive Rate**: The percentage of instances classified as faulty w.r.t the total number of correct instances
- **False Negative Rate**: The percentage of instances classified as correct w.r.t the total number of faulty instances

They were chosen for this study as well, so that the results could eventually be related to other studies.

6.3.2 Experiment Execution

WEKA allows 10-fold cross validation and provides all three measures. Since the code instances can contain missing values, not all machine learners provided by WEKA are applicable. Therefore they were filtered out. The study was done in two steps. First all the applicable learners were evaluated one after the other on all code features (except line number and file name). Then the same were evaluated on a reduced feature set as described in 5.2.3. The next subsection presents the most interesting results. More details about the performance of the learners can be found in the appendix A.
6.3.3 Results and Observations

In total 69 machine learning algorithms were evaluated. Only the best performing ones are presented in the following charts. From these results, a small subset of machine learners is put together. This subset is afterwards used in the overall evaluation.

Accuracy

On average the learners reached an accuracy over 75%. This is an acceptable value in literature. Especially tree learners perform well on the given training set. The figure 6.1 shows that reducing the features increases the accuracy slightly for almost all types of learners. Only rule learners perform worse, since some of them based a lot on variable IDs and type information. Furthermore it seems, that trees and meta learners can handle large feature sets like the training set very well. They do not increase a lot by reducing features. Bayesian learners and nearest neighbour algorithms instead would probably work much better with less features.

The two figures in 6.2 confirm these observations. They list the learners with the most accuracy rate for the total feature set and the reduced one, respectively. Most learners above 90% belong to tree or meta learners and no Bayesian learner was even able to reach 85%. Among the tree learners perform Rotation Forests based on J48 trees, Logistic Model Trees (LMT) and Best-First decision Trees (BFT) best. Classification via Regression and AdaBoost Best-First decision trees work best as meta learners. Also interesting is here the Multilayer Perceptron. It achieves in both cases the highest rate at all. Notably is also JRip. This rule learner performs well on all features.
Chapter 6. Suitability Study and Evaluation

The average false positive rates of the different categories is not surprising. As shown in the figure 6.3, they conform more or less to the accuracy rates. Only the nearest neighbour classifiers show a higher false positive rate compared to the Bayesian classifiers. The figure shows also, that reducing the feature set decreases in most cases the false positive rate on average.

The average false positive rate for the evaluated learners is almost 30%. This is way too high for defect detection. Such a high rate either leads to introducing more faults by updating correct code or to skipping many real faults. However, some of the learners perform much better than the average and can reach almost 3%.

Comparing the performance of the classifiers on the whole and on the reduced feature set, it can be said, that the reduction of the features is an advantage in
almost all cases. The figures in 6.4 demonstrate this graphically for the learners with a false positive rate under 10%. Here are the meta classifiers (J48 Rotation Forests or AdaBoost) slightly better than the tree learners (LMT or BFTrees). And again performs the Multilayer Perceptron very good; as well as JRip on the whole feature set.

![Figure 6.4: Best Performing Machine Learners w.r.t FP Rate](image)

**False Negative Rate**

Usually, when people talk about defect detection, the false negative rate has minor importance. But obviously it must not be too high for the used learners. Otherwise the approach becomes everything else than efficient. Therefore this rate is considered here, too.

The figure 6.5 shows the average false negative rate of the learners’ categories. The rate is with a value of 20% about 10% lower than the false positive rate. For the tree and meta learners it is even below the average. This is an acceptable situation and conforms with the observations about the accuracy of these learners.

Very interesting in this case is the Lazy category, i.e. the nearest neighbour algorithms. They perform quite badly w.r.t the false positive rate but show the best false negative rate. This is most probably because the faulty instances differ only slightly from their correct version. This way many correct instances are mistakenly identified as faulty and so only few faulty ones are missed. This situation is not desirable. Thus this category of learners is totally dismissed when evaluating the overall approach. Similarly the Bayesian classifiers and those belonging to the Misc category. They show a very low accuracy and high false positive and negative rates. Therefore they are not suitable for the prototype’s output as well.

Similar to the other two measures, the figure 6.5 additionally shows, that the learners perform better by reducing the feature set. This is in some way obvious;
unnecessary information just confuses the learner when it builds its model and rises so the chance of misclassifying the instances. But the interesting fact, what these comparisons show, is that the functional, rule, meta and tree classifiers are generally more resistant to irrelevant data than the classifiers from the other categories.

The figures in 6.6 instead list the learners with the smallest false negative rates individually. Quite many are below 10\%, especially on the reduced feature set. Most of the learners, which perform best w.r.t accuracy and false positives, show also low false negatives.

As a result of this study, the most suitable learners for the given type of instances come from the trees, meta, rules and functions categories. The absolutely best functional classifier is by far the Multilayer Perceptron. The best rule classifiers instead are NNge and JRip (although the latter is convincing only on all features). From the trees are the LMT and BFT most suitable. The Functional Tree (FT) and the simple Classification and Regression Tree (SimpleCART) can also be considered. And as meta learners are J48 Rotation Forests, AdaBoost BFT and Classification via Regression most applicable to the prototype’s output.

6.4 Statistical Significance

The above mentioned 11 most suitable learners got excellent accuracies (above 90\%) on the reduced feature set. To confirm, that these values are not by chance and that the corresponding learners are significantly better than the simplest one, a significance test at a 0.05 level is done for 10\(^1\) of them.

\(^1\)The Multilayer Perceptron could not be considered yet due to its high time consumption when running on the given dataset.
WEKA offers the so called experimenter for such purposes. The ZeroRule learner is taken as reference. This learner always predicts the mode, i.e. if there are more correct than faulty instances in the training set, it classifies everything as correct. Classifying always to the same class is the simplest and usually the worst thing, one can do. In this case, as the training set contains almost a balanced number of faulty and correct instances, the ZeroRule learner predicts with an almost 50:50 chance. Now, taking this simplest learner as baseline, a Paired T-Test can be performed.

To establish the needed samples for each of the learners, the 10-fold cross validation as described in the previous section was performed 10 times per learner using WEKA on the training set with the reduced features. This resulted in a dataset of 1,100 independent runs; 100 per learner. Again using WEKA, the paired T-test is then performed on these samples. The exact details about this T-test can be found in the book referenced by [40].

The table 6.2 summarizes the results of this test. It shows for each learner the mean and the standard deviation in the accuracy. As expected, all 10 learners are significantly better.

### 6.5 Evaluation

The suitability study provides the learners, which work best on the given training set. But remember that the training set is a manually gathered collection of instances out of many different runs of the prototype. So if these learners are
trained with the full training set, how do they perform on real data from one single run? Answering this question shows, whether the model (prototype combined with some of the most suitable learners) goes in the right direction to successfully and effectively detect faults in source code.

In order to perform this evaluation, the prototype was run once on a method of one of the 5 open source project. Faults were injected before the run to get in total 38 faulty code instances. Since the prototype is not yet fully adapted to all situations the resulting output had to be finalized by removing incomplete and unnecessary instances. Finally, the test set is composed of 38 (known) faulty and 2,316 correct instances. This makes in total 2,354 instances. Then the few classifiers are trained and tested one after the other using WEKA and the same configurations from the suitability study. Furthermore the reduced feature set was used, since the feature reduction showed a performance improvement for most of the classifiers.

### 6.5.1 Results

The table 6.3 summarizes the results in terms of accuracy, false positive and false negative rates for each classifier used to detect the faulty operations in the test set. The classifiers are increasingly sorted by their accuracy and false positive rate.

While looking at these results must be kept in mind, that there may still be faulty instances among the 2,316 correct ones. Only the known 38 faulty operations were searched in the test set and marked as faulty. The others are just assumed to be correct.

The table shows that 6 out of 11 classifiers achieve a very high accuracy. Although the other 5 classifiers reach a very low false negative rate, they are not acceptable for defect detection. Their false positive rates are clearly too high.
Table 6.3: Performance of the Classifiers on the Test Set

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Category</th>
<th>Correct</th>
<th>Incorrect</th>
<th>ACC</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rot.Forest-J48</td>
<td>Meta</td>
<td>2322</td>
<td>32</td>
<td>98.64</td>
<td>0.012</td>
<td>0.105</td>
</tr>
<tr>
<td>Multil.Perc.</td>
<td>Functions</td>
<td>2275</td>
<td>79</td>
<td>96.64</td>
<td>0.034</td>
<td>0.026</td>
</tr>
<tr>
<td>JRip</td>
<td>Rules</td>
<td>2275</td>
<td>79</td>
<td>96.64</td>
<td>0.034</td>
<td>0.026</td>
</tr>
<tr>
<td>FT</td>
<td>Trees</td>
<td>2253</td>
<td>101</td>
<td>95.71</td>
<td>0.044</td>
<td>0.000</td>
</tr>
<tr>
<td>LMT</td>
<td>Trees</td>
<td>2145</td>
<td>209</td>
<td>91.12</td>
<td>0.090</td>
<td>0.026</td>
</tr>
<tr>
<td>Classif.ViaRegr.</td>
<td>Meta</td>
<td>2129</td>
<td>225</td>
<td>90.44</td>
<td>0.094</td>
<td>0.211</td>
</tr>
<tr>
<td>BFTree-UnPr.</td>
<td>Trees</td>
<td>1828</td>
<td>526</td>
<td>77.66</td>
<td>0.225</td>
<td>0.132</td>
</tr>
<tr>
<td>AdaB.-BFTree</td>
<td>Meta</td>
<td>1706</td>
<td>648</td>
<td>72.47</td>
<td>0.278</td>
<td>0.132</td>
</tr>
<tr>
<td>BFTree-PostPr.</td>
<td>Trees</td>
<td>1648</td>
<td>706</td>
<td>70.01</td>
<td>0.303</td>
<td>0.132</td>
</tr>
<tr>
<td>SimpleCart</td>
<td>Trees</td>
<td>1641</td>
<td>713</td>
<td>69.71</td>
<td>0.306</td>
<td>0.132</td>
</tr>
<tr>
<td>NNge</td>
<td>Rules</td>
<td>1132</td>
<td>1222</td>
<td>48.09</td>
<td>0.525</td>
<td>0.184</td>
</tr>
</tbody>
</table>

The J48 based Rotation Forest performs surprisingly well. Its false positive rate is excellent. JRip is also surprising, especially because the reduced feature set was used for this evaluation. JRip and the Multilayer Perceptron can both keep a relatively low false positive as well as false negative rate. Interesting is also the Functional Tree classifier. It was able to identify all faulty instances. But its false positive rate near 5% is correspondingly higher than for the former learners.

Interesting in the other sense is the BFT based AdaBoost classifier. In the suitability study it was among the winning machine learners, but on the test data it performs quite badly. Rätsch et al. showed in [29] that AdaBoost algorithms can lead to over-fitted models. And over-fitting causes misclassification due to the missing ability of generalization. This could easily be the reason for the bad performance on the test set.
Chapter 7

Validity

There are some facts, which limit the validity of this study. The following sections explain them briefly.

7.1 Internal Validity

For establishing the suitability study and the evaluation there was the need to gather data from real projects. The chosen projects are subjective to the author. Also the types of the faults detected by the model are chosen subjectively. Similarly the feature sets and classifiers.

7.1.1 Unknown Faults in Test Set

As mentioned in the previous chapter, the evaluation just assumes that all 2,316 instances marked as correct are indeed so. But this might not be the case for all, since they were not explicitly checked. This fact may distort the measurements given in the table 6.3 in several ways. Let us assume there are more than 38 faulty instances in the test set. Then there are two situations:

1. A classifier did not classify any hidden faulty instance.
2. A classifier was able to classify at least one hidden faulty instance.

The first scenario implies, that the accuracy decreases while the false positive and negative rates increase. The second scenario instead implies, that the accuracy increases, while the false positive and negative rates decrease. The results of this work must therefore be observed with caution. To validate them, further investigation is necessary.

7.2 External Validity

The projects used for the work are all open source projects. None of them is a real industrial system. Although they are chosen in a way, that they come from different domains, they are not representative for all possible projects. Moreover
all analyzable projects must be written in Java. The results of this work are thus not generalizable w.r.t the language of a project.

The implemented prototype is limited up to now to certain type of faults. They are limited to those appearing most critical to the author. But there are many more, which must be covered for making the model applicable in real-life situations. The results are thus also not generalizable w.r.t the detectable faults.

The learners used for this work come exclusively with the WEKA framework. There may be other algorithms working well or better on such data sets as used within this work. Furthermore, the results of the study may change with the size of the training and feature set. So the best classifiers are for sure best only within the context of this work.
The presented approach for defect detection seems to have the potential to be effective. But it is realized only by a prototype implementation for extracting the code features. This part needs to be expanded such that more faults can be detected. If the evaluation of an advanced implementation provides similar results, then there is also the possibility to generalize them w.r.t the detectable faults.

The prototype covers only the feature extraction part, but the machine learners must be run manually on the prototype’s output. Since WEKA offers a command line interface, the prototype could easily be expanded in such a way, that it includes an interface to WEKA and the suitable machine learners.

The code operations classified as faulty could then also be visualized to the programmers. The possibility to mark a faulty classified code operation as correct and feed it back to the classifier would be an option, too.

In order to achieve the ability to generalize w.r.t the type of analyzable projects, the prototype could be expanded to cover other programming languages. Moreover, experiments and evaluations on industrial projects could be conducted to see, to which extent the results conform to these applications.

Having a complete tool for the presented model, it could finally be compared to nowadays bug finders. There is definitively the need to know, how the fully-developed version of this approach performs w.r.t other defect detection approaches like FindBugs or commercial applications.

Furthermore, the complete tool should be evaluated according to its efficiency, i.e. resource consumption, to check the feasibility of running it in practice. Up to now, only the effectiveness of the prototype implementation was evaluated in terms of defect detection rates without explicitly checking memory or time consumption.

It was noticed, that most of the learners built their learning schemes very fast and classified the test set with about 2,000 instances in few minutes. But there
are some exceptions. For example the Multilayer Perceptron took on the reduced feature set more than 12 hours in order to build its scheme and classify the test instances.

It would also be interesting to see, whether the presented results are reproducible with a much larger training set. For the suitability study and the evaluation there have been used about 650 instances. This is a very small amount compared to common machine learning experiments. A larger training set could increase the accuracy of some learners. Experiments on the feature set instead could reveal further classifiers suitable for defect detection.
Machine learning is used in many research areas. Only recently it is combined with source code analysis with the goal to perform an effective defect detection. This work presents a new approach for detecting defects in Java source code based on existing machine learners.

The approach consists of two steps. The first one deals with the questions on how source code features can be extracted and on how they can be represented, so that machine learners can classify Java code instances into faulty or correct. To realize this part, a prototype has been implemented. It is based on the Java Compiler Compiler to analyze Java code. Among other code features, it extracts those through which 8 common types of faults can be identified. It represents the features then as code instances in an attribute relation file format (ARFF). This format is used by all classifiers of the well known WEKA framework, through which the existing learners are provided.

The second step instead deals with another question: Which of the existing machine learners are most suitable for these code instances so that the approach becomes most effective? So using WEKA, a suitability study and an evaluation of the approach were conducted in order to answer this third question. Both base on code instances from 5 open source projects. The suitability study assesses over 70 classifiers based on accuracy, false positive and false negative rates. Whereas the evaluation assesses the presented approach by applying the most suitable learners to a separate test set of features.

The suitability study shows, that 11 learners are suitable to classify the code instances by performing 10-fold cross validations on a training set. Best performing is the Multilayer Perceptron with an accuracy of 97.03% on all extracted features and 97.19% on a reduced feature set. The rule learner JRip and the Logistic Model Tree (LMT) have a lower accuracy than the Multilayer Perceptron but on all features they show the same and even a smaller false positive rate, scoring 3.7% and 3.1% respectively. On the reduced feature set has the J48 based Rotation Forest the smallest false positive rate with 3.4%. All these learners show a low false negative rate as well, being them below 4.5%. The Best First Tree (BFT), Simple Classification and Regression Tree (SimpleCart), Functional Tree (FT), the meta learners AdaBoostM1 and Classification Via Regression and the
rule learner NNge perform also still acceptable from the accuracy point of view. The performance of these learners is even statistically validated.

The results of the evaluation instead show, that the Java code instances can be detected with an accuracy of 98.64% and a false positive rate of 1.2% by using the J48 based Rotation Forest. The Multilayer Perceptron, JRip and FT classifiers achieve also good results.

Although the prototype is not fully developed, this study shows, that the presented approach has the potential to be an effective alternative to nowadays defect detection approaches. But this study has several limitations and shows also, that future works are necessary to go there.

Nevertheless, this work contributes to the current body of knowledge by confirming the belief that it is possible to effectively detect defects by combining automated static code analysis with machine learning. It contributes also by presenting a new approach in this area together with a prototype implementation. Finally, the study doesn’t only show, that the presented approach is feasible but also reveals those machine learners, which work best for defect detection in the given context.
### Appendix A

#### Suitability Results

The following table A.1 summarizes the results of the suitability study for all used machine learners. The data is sorted by classifier name. It shows for the standard and for the reduced feature set the learner’s accuracy, false positive and false negative rates.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Feature Set</th>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaB.-BFTree</td>
<td>Meta</td>
<td>93.125</td>
<td>0.071</td>
</tr>
<tr>
<td>AdaBoostM1-Dec.Stump</td>
<td>Meta</td>
<td>62.813</td>
<td>0.477</td>
</tr>
<tr>
<td>ADTree</td>
<td>Trees</td>
<td>71.563</td>
<td>0.269</td>
</tr>
<tr>
<td>AODE</td>
<td>Bayesian</td>
<td>69.531</td>
<td>0.325</td>
</tr>
<tr>
<td>AODEr</td>
<td>Bayesian</td>
<td>75.000</td>
<td>0.276</td>
</tr>
<tr>
<td>Attr.Sel.Classif.</td>
<td>Meta</td>
<td>70.000</td>
<td>0.368</td>
</tr>
<tr>
<td>Bagging-REPTree</td>
<td>Meta</td>
<td>87.656</td>
<td>0.152</td>
</tr>
<tr>
<td>BayesNet-5MAEst</td>
<td>Bayesian</td>
<td>66.250</td>
<td>0.464</td>
</tr>
<tr>
<td>BayesNet-SimpleEst.</td>
<td>Bayesian</td>
<td>62.656</td>
<td>0.415</td>
</tr>
<tr>
<td>BTTree-PostPr.</td>
<td>Trees</td>
<td>91.563</td>
<td>0.090</td>
</tr>
<tr>
<td>BTTree-UnPr.</td>
<td>Trees</td>
<td>91.406</td>
<td>0.096</td>
</tr>
<tr>
<td>ClassBal.ND-J48</td>
<td>Meta</td>
<td>76.563</td>
<td>0.266</td>
</tr>
<tr>
<td>Classif.ViaClust.</td>
<td>Meta</td>
<td>50.625</td>
<td>0.418</td>
</tr>
<tr>
<td>Classif.ViaRegr.</td>
<td>Meta</td>
<td>93.438</td>
<td>0.065</td>
</tr>
<tr>
<td>Conj.Rule</td>
<td>Rules</td>
<td>60.625</td>
<td>0.622</td>
</tr>
<tr>
<td>Dagging</td>
<td>Meta</td>
<td>72.344</td>
<td>0.393</td>
</tr>
<tr>
<td>DataNearBal.ND-J48</td>
<td>Meta</td>
<td>76.563</td>
<td>0.266</td>
</tr>
<tr>
<td>Dec.Stump</td>
<td>Trees</td>
<td>60.625</td>
<td>0.607</td>
</tr>
<tr>
<td>dec.Table-BFS</td>
<td>Rules</td>
<td>81.094</td>
<td>0.260</td>
</tr>
<tr>
<td>dec.Table-Generics</td>
<td>Rules</td>
<td>81.094</td>
<td>0.207</td>
</tr>
<tr>
<td>Decorate-J48</td>
<td>Meta</td>
<td>80.000</td>
<td>0.241</td>
</tr>
<tr>
<td>DTNB</td>
<td>Rules</td>
<td>84.219</td>
<td>0.192</td>
</tr>
<tr>
<td>END-J48</td>
<td>Meta</td>
<td>76.563</td>
<td>0.266</td>
</tr>
<tr>
<td>EnsembleSel.-J48</td>
<td>Meta</td>
<td>80.625</td>
<td>0.223</td>
</tr>
<tr>
<td>Filt.Classif.-J48</td>
<td>Meta</td>
<td>76.563</td>
<td>0.266</td>
</tr>
<tr>
<td>FT</td>
<td>Trees</td>
<td>89.844</td>
<td>0.093</td>
</tr>
<tr>
<td>HyperPipes</td>
<td>Misc</td>
<td>62.031</td>
<td>0.724</td>
</tr>
<tr>
<td>IB1</td>
<td>Lazy</td>
<td>60.625</td>
<td>0.437</td>
</tr>
<tr>
<td>IBk</td>
<td>Lazy</td>
<td>64.219</td>
<td>0.505</td>
</tr>
<tr>
<td>J48</td>
<td>Trees</td>
<td>76.563</td>
<td>0.266</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Machine Learner</th>
<th>Type</th>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48Graft</td>
<td>Trees</td>
<td>ACC</td>
<td>76.563</td>
</tr>
<tr>
<td>JRip</td>
<td>Rules</td>
<td>96.250</td>
<td>0.037</td>
</tr>
<tr>
<td>KStar</td>
<td>Lazy</td>
<td>52.656</td>
<td>0.647</td>
</tr>
<tr>
<td>LADTree</td>
<td>Trees</td>
<td>72.344</td>
<td>0.300</td>
</tr>
<tr>
<td>LBR</td>
<td>Lazy</td>
<td>83.438</td>
<td>0.214</td>
</tr>
<tr>
<td>LMT</td>
<td>Trees</td>
<td>96.563</td>
<td>0.031</td>
</tr>
<tr>
<td>Logistic</td>
<td>Functions</td>
<td>86.250</td>
<td>0.155</td>
</tr>
<tr>
<td>LogitBoost-Dec.Table</td>
<td>Meta</td>
<td>70.781</td>
<td>0.307</td>
</tr>
<tr>
<td>LWL-Dec.Stump</td>
<td>Lazy</td>
<td>69.531</td>
<td>0.520</td>
</tr>
<tr>
<td>LWL-J48</td>
<td>Lazy</td>
<td>90.625</td>
<td>0.158</td>
</tr>
<tr>
<td>MultiBoostAB-J48</td>
<td>Meta</td>
<td>62.031</td>
<td>0.576</td>
</tr>
<tr>
<td>MultiClassClassif.-J48</td>
<td>Meta</td>
<td>86.250</td>
<td>0.155</td>
</tr>
<tr>
<td>Multil.Perc.</td>
<td>Functions</td>
<td>70.781</td>
<td>0.307</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>Bayesian</td>
<td>62.188</td>
<td>0.399</td>
</tr>
<tr>
<td>NBTree</td>
<td>Trees</td>
<td>91.250</td>
<td>0.099</td>
</tr>
<tr>
<td>ND-J48</td>
<td>Meta</td>
<td>76.563</td>
<td>0.266</td>
</tr>
<tr>
<td>NNge</td>
<td>Rules</td>
<td>90.625</td>
<td>0.111</td>
</tr>
<tr>
<td>OneR</td>
<td>Rules</td>
<td>62.500</td>
<td>0.570</td>
</tr>
<tr>
<td>Ord.ClassClassif.-J48</td>
<td>Meta</td>
<td>76.563</td>
<td>0.266</td>
</tr>
<tr>
<td>PART</td>
<td>Rules</td>
<td>85.000</td>
<td>0.217</td>
</tr>
<tr>
<td>Rand.Committee-Rand.Tree</td>
<td>Meta</td>
<td>66.563</td>
<td>0.313</td>
</tr>
<tr>
<td>Rand.Forest</td>
<td>Trees</td>
<td>75.469</td>
<td>0.257</td>
</tr>
<tr>
<td>Rand.SubSpace-REP.Tree</td>
<td>Meta</td>
<td>85.000</td>
<td>0.195</td>
</tr>
<tr>
<td>Rand.Tree</td>
<td>Trees</td>
<td>91.719</td>
<td>0.480</td>
</tr>
<tr>
<td>REP</td>
<td>Functions</td>
<td>54.888</td>
<td>0.430</td>
</tr>
<tr>
<td>REPTree</td>
<td>Trees</td>
<td>79.688</td>
<td>0.251</td>
</tr>
<tr>
<td>Ridor</td>
<td>Rules</td>
<td>76.694</td>
<td>0.319</td>
</tr>
<tr>
<td>Rot.Forest-J48</td>
<td>Meta</td>
<td>96.250</td>
<td>0.040</td>
</tr>
<tr>
<td>SimpleCart</td>
<td>Trees</td>
<td>91.250</td>
<td>0.093</td>
</tr>
<tr>
<td>SimpleLogistic</td>
<td>Functions</td>
<td>84.844</td>
<td>0.189</td>
</tr>
<tr>
<td>SMO-Norm.PolyK.</td>
<td>Functions</td>
<td>70.625</td>
<td>0.389</td>
</tr>
<tr>
<td>SMO-PolyK.</td>
<td>Functions</td>
<td>84.531</td>
<td>0.183</td>
</tr>
<tr>
<td>SMO-PUK</td>
<td>Functions</td>
<td>62.969</td>
<td>0.341</td>
</tr>
<tr>
<td>SMO-RBFK.</td>
<td>Functions</td>
<td>84.025</td>
<td>0.424</td>
</tr>
<tr>
<td>ThresholdSel.-Log.</td>
<td>Meta</td>
<td>84.688</td>
<td>0.186</td>
</tr>
<tr>
<td>VFI</td>
<td>Misc</td>
<td>60.313</td>
<td>0.393</td>
</tr>
<tr>
<td>VotedPerc.</td>
<td>Functions</td>
<td>59.375</td>
<td>0.384</td>
</tr>
<tr>
<td>Winnow</td>
<td>Functions</td>
<td>52.188</td>
<td>0.594</td>
</tr>
<tr>
<td>ZeroR</td>
<td>Rules</td>
<td>50.469</td>
<td>0.000</td>
</tr>
</tbody>
</table>

**Table A.1:** Performance of All Used Machine Learners from the Suitability Experiment
Appendix B

Complementary Study

At the same time with this study, Tribus [37] conducted a complementary one. It
regards the application of machine learners to extracted C code properties. The
following two sections give an overview of its outcome and relate it to the given
one.

B.1 Study Overview

The goal of the complementary study was to detect defects in C code by means
of machine learning. First a prototype was developed for extracting relevant
C code properties. Based on the prototype, features were extracted from open
source projects to establish a training set for WEKA machine learners. Also here
was a subset of fault categories considered; most notably allocation problems like
dereferencing null pointers and double-frees. Then a study was performed to get
the most suitable learners. Finally the model which consists of the prototype and
the most suitable learners was evaluated.

<table>
<thead>
<tr>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>corr</td>
</tr>
<tr>
<td>ADABoost(BFTree) Meta</td>
<td>96.774</td>
</tr>
<tr>
<td>Ibk (LinearNN) Lazy</td>
<td>96.573</td>
</tr>
<tr>
<td>LMT Tree</td>
<td>96.573</td>
</tr>
<tr>
<td>MultiLayer.Perc. Func.</td>
<td>96.573</td>
</tr>
<tr>
<td>NNge Rule</td>
<td>96.075</td>
</tr>
<tr>
<td>RotationForest Meta</td>
<td>95.987</td>
</tr>
<tr>
<td>Rand.Com.(RandT.) Meta</td>
<td>94.597</td>
</tr>
</tbody>
</table>

Table B.1: Choice of the Most Suitable Learners for C Code

The tables B.1 and B.2 show the outcomes of the suitability study and the
evaluation from the complementary study. The first one presents the 7 machine
learners, which performed best during the suitability study using 10-fold cross
validation. The second table instead presents the performance of the learners,
when they were trained and then applied to a separate test set. The measurements
are accuracy of classifying correctly and of classifying incorrectly, false negative
rate and false positive rate. Both, the full and the reduced feature sets, are
considered in the tables.
The similar goals and used methodology of the two studies are the basis for a reasonable comparison. In collaboration with the author of the complementary study, the subsequent conclusions were drawn.

<table>
<thead>
<tr>
<th></th>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>corr</td>
<td>incorr</td>
</tr>
<tr>
<td>Ibk (LinearNN)</td>
<td>98.434</td>
<td>0.064</td>
</tr>
<tr>
<td>Lazy</td>
<td>96.434</td>
<td>2.021</td>
</tr>
<tr>
<td>Rand.Com.(RandT.)</td>
<td>Meta</td>
<td>97.979</td>
</tr>
<tr>
<td>LMT</td>
<td>Tree</td>
<td>97.423</td>
</tr>
<tr>
<td>NNge</td>
<td>Rule</td>
<td>94.543</td>
</tr>
<tr>
<td>RotationForest</td>
<td>Meta</td>
<td>91.359</td>
</tr>
<tr>
<td>MultiLay.Perc.</td>
<td>Functions</td>
<td>77.160</td>
</tr>
</tbody>
</table>

Table B.2: Evaluation of the Most Suitable Learners on the Test Set from C Code

B.2 Conclusions

Despite that the feature set produced by the parser for C is quite different in its size and in the kind of attributes that are tracked, the underlying idea is in both cases the same and therefore the two studies should still be comparable. The differences arise mainly from the different possibilities the basic language in either the two cases offers. At the end the feature set in Java was approximately 3 times bigger in size than the one used for C.

Interesting results regarding the comparability of the works are on one hand the set of learners that perform well in both studies because from them it is possible to conclude about the suitability of some learners and about the method itself, on the other hand the differences which allow to conclude on the impact of the size of the feature set on some machine learning algorithms. Therefore one of the most obvious outcomes are the set of learners that could perform well in both cases, which are “LMT”, “RotationForest” and “MultiLayerPerceptron”. Especially the latter showed an excellent performance by being among the best in any evaluation of both works.

An even bigger similarity showed up in the performance of “ADABOOST” on a “BFTree”. In both works it was among the best performing learners in the suitability study, but when it came to the second evaluation its accuracy fell down to about 70%.

Among the observed differences, the most obvious and important one is that in the case of C the nearest neighbour algorithms like “Ibk” or “NNge” perform quite good in both, the suitability study and the evaluation, and therefore they are mentioned in the resulting tables as well. However this learners are not able to produce acceptable results for the Java output. This could be due to the larger feature set used in Java which seems not to be in favour of these kind of learners.

The results obtained for Java instead show that the rule based “JRip” and the “FT” tree are among the best performing learners. In the case of C, these
two learners performed so badly in the suitability study that they have not even been part of the second evaluation.

However the conjunctive results of the two works reduce a limitation by showing the suitability of the method for different kind of programming languages. They showed with their parsers that it is possible to implement the method in a prototype. And despite that their feature sets are different, a possible future work could be to unify and implement them into a complete tool.
Appendix C

Systematic Literature Review

As mentioned in chapter 3 and section 5.1, a systematic literature review was performed before this work was done. The review was conducted in collaboration with Tribus, the author of the complementary study presented in appendix B. It is titled “Applicability of Machine Learning Techniques to Source Code Analysis” and is provided in the following pages “as is”.

Abstract—Quality is probably the most important characteristic which a software system should own. In this context a systematic literature review was performed. Subject to this review are published papers talking about source code analysis using machine learning algorithms with the goal of detecting or predicting defects. It will be shown, that machine learning techniques can benefit to code analysis and perform at least as well as standard techniques. Moreover improvements and missing approaches are addressed.

I. INTRODUCTION

Nowadays more and more devices become controlled by some sort of software system. Almost independent from the application domain the range goes from funny toys up to critical systems. The more critical the system is the less bugs it should contain. However the final goal in all software development activities is to deliver high quality applications.

In spite of being impossible to omit the presence of bugs in source code [7] the aim of each developer should be to reduce their occurrences to a minimum. This seems to be a clear rule for a developer, whereas how to achieve it is not.

There have been proposed several techniques to reach the desired effect, which start from coding rules, over manual inspection or testing up to more sophisticated ways like tool supported error detection or prediction. The latter are exactly the focus of this study. We try to assess the defect prediction as well as detection models used to predict or identify faults in source code. To be more precise we concentrate just on those models that make use of machine learning algorithms to some extend.

Machine learning algorithms have been used in many different problem domains in the past, now its time to review how they can be used to identify faults in source code or in other words, how they can be used to achieve high quality software.

This train of thought brought us to the final question that we try to answer within this paper. In one sentence this question can be stated as follows:

1) Do machine learning techniques benefit to defect detection and defect prediction while performing source code analysis?
   a) How accurate do they benefit to the analysis?
   b) How can the analysis approaches be improved?
   c) Are there missing approaches?

During this study we will try to answer if the application of machine learning techniques improves the defect prediction or detection techniques by looking at published papers and cases studies. However, in order to answer this big question we will focus on answering the smaller ones. These are related to the extend of benefit they can gain, the ways how existing techniques could be extended and the identification of new approaches, new opportunities to work on.

Our hypothesis in this case says that using techniques of the machine learning field it should be possible to improve the reliability of a prediction model or a detection tool. If this hypothesis can hold or not will be an outcome of this study.

This paper is structured as follows. First of all we define our problem domain more in detail by giving a short background into the areas we are looking into. Then the methodology used to retrieve all the needed information and to formalize them is presented. Afterwords we present some of the most important papers found by our research more in detail. At the end there will be a discussion part stating the most important results together with a conclusion about the findings.

II. BACKGROUND

As already mentioned in the previous section the overall goal of each software development is to produce high quality software. There have been proposed several approaches to achieve this ideal.

One of these approaches is source code analysis, usually done as part of formal reviews [3], [4]. Using this approach one or more developers which should preferably not be the ones that wrote the source code, go through the code and try to identify memory leaks, buffer overflows or other flaws. Goal of such code analyses is finding possible defects without execution of the developed application. This has the advantage that they can be applied early in the development process, namely as soon as an algorithm, class, file or library is checked in. And so defects can be eliminated before they can influence other parts of the source code.

A formal review is a time consuming stage of the development process and companies are tempted to adapt it to save in time and effort. This might lead to inefficiency, as reported by the case study presented in [5]. However, partially
automating this stage can be a solution. The source code
analysis step gives an opportunity for automation. There exist
already source code analyzers, which can give developers hints
on where defects may reside within the code (see [6] for a
description and discussion of three tools). The analyzers either
detect directly defects by using pattern matching or provide
code metrics which can be used for building defect prediction
models.

Prediction models have been based mostly on mathematical
techniques like regressions or Markov chains. The problem
with these approaches is their accuracy in the prediction. The
major problem with pattern matcher instead is their rate of
false positives, i.e. the identified issues are not defects at all in
the given context. Through the application of machine learning
algorithms to the defect detection and defect prediction process
engineers try to go about these problems.

Machine learning has already been proved to be an impor-
tant discipline in several fields like bio-informatics, computer
vision and game playing. Although the machine learning
algorithms exist since decades, they are only recently applied
to investigate source code. First they were used just for
establishing defect prediction models. During the last decade,
plenty of such models were reported.

Then, more recently, machine learning techniques are also
used for analyzing the code itself in depth. Shabtai et al. [8]
for example provide appropriate machine learning algorithms
in order to classify code to be malicious or harmless based
on features selected from the code’s binary format. Tan et
al. [9] instead analyse the code to see whether the comments
match the meaning of the code itself. But there have been
carried out only few case studies in industry up to now, which
regard machine learning and source code analysis. So research
is going on to validate the applicability of existing machine
learning algorithms to the source code inspection process.
Researchers try to discover the most accurate defect detection
and defect prediction models and try to apply those to more
practical projects in industry.

III. RESEARCH METHODOLOGY

This study is based on a systematic review. A systematic
review is a comprehensive literature review to answer a
research question based on evidence from prior researches.
Initially they were coming from the medical field but now
they are widely used. Performing a systematic review requires
3 main phases, whereby phase number 2 is in contrast to the
other two phases iterative within its sub-steps:
1) Planning: includes identifying the research question and
scheduling the activities to be performed.
2) Conducting: includes literature search, review and ana-
lysis.
3) Documenting: includes presenting and validating the
research.

A detailed explanation of a systematic review and a com-
prehensive description of its process can be found in [10]. The
outline of our review process instead is shown in figure 1. The
second phase on how we conducted our review is explained
more in detail in the following paragraphs.

Subject of this review are published papers which focus on
source code analysis using machine learning with the goal of
finding or predicting defects in source code. Thereby the type
of analysis is not constrained and could therefore be performed
either dynamically or statically. However, the papers must have
been accessible in full text without any charge in order to be
considered here.

This is one reason, why Compendex and Inspec were used
as the search databases for the published papers, which are re-
viewed in this study. The used infrastructure\(^1\) allowed to search
both databases at once, what made them preferable as well.
But the main advantage of Compendex and Inspec represents
the databases size. Both contain millions of scientific papers
since 1969, including also plenty of papers which are related
to computer science. Furthermore, they cover the Springer
Link and IEEE Xplore articles and include this way the IEEE
conference papers, too. Other search engines like CiteSeerx
or SCOPUS were looked up for testing their appropriateness,
but in comparison with Compendex and Inspec they did not
provide the accuracy in finding the papers.

\(^{1}\text{BTH Library System}\)
The first attempt searching the databases using the search string “machine and learning and code and analysis” in all possible fields resulted in a set of over 900 papers. Lots of them were not concerned with fault prediction or detection at all. This fact strengthened our assumption, that there are much less relevant papers out there since the underlying research fields have been combined just recently.

Consequently, the literature search was started by narrowing the result set first as much as possible and then expanding it in size. The search strings were targeted in all but one case solely to title, abstract and subject. Furthermore, the results were sorted according to their relevance. This, because the reasonable number of irrelevant papers at the end of the result set given by the last search string was the termination criteria for the literature search. The summary of the search results can be found in table I and the search strings were composed as follows:

1) S1: (source OR code) AND analysis AND software AND (defects OR errors OR faults OR flaws) AND automated AND (“machine learning” OR “case based reasoning” OR “decision trees” OR Bayesian OR “neural network”)

2) S2: ((static AND analysis AND ”machine learning”) AND ((detection OR prediction) WN All fields) NOT (financial WN All fields)

3) S3: (quality AND software AND (“machine learning” OR “case based reasoning” OR “neural network” OR “decision tree”) AND (code OR source) AND analysis)

4) S4: (analysis and software and (defects or errors or faults or flaws) and (“machine learning”))

We considered a paper as potentially relevant, if it mentioned in the title and/or the abstract that it uses machine learning for building defect detection or prediction models. Furthermore, we included papers which were evaluating such models. Otherwise, we excluded a paper. Sometimes, papers looked promising by just looking at title and/or abstract. But after reading through, they either didn’t base their models on code analysis results or appeared to be not “academic” enough. In this case papers were excluded as well after agreement.

As shown in table I, 21 completely related and accessible papers were found in the databases. In addition to those, 13 other available papers have been selected from the references of the former. Besides these 34 papers, two master thesis works - [1] and [2] - are taken into account. After having collected papers we made a short summary. Moreover we collected for each data regarding the following properties:

- model type: prediction or detection
- machine learning type: tree, neural networks, case based reasoning etc.
- input for machine learners: code metrics, design metrics, process data etc.
- whether a model was evaluated on real projects or source code
- project domain used for model evaluation
- programming language of the projects

The data about these properties was then filled into spreadsheets in order to visually represent it afterwards. Unfortunately, the exact accuracy values for the machine learning models were rarely provided by papers. Thus, the level of accuracy - low, high, lower or higher than standard techniques - was only recorded in the summaries of the papers and could not be represented visually. However, on the basis of the summary and the spreadsheets we discussed the models and presented the results in this paper.

### IV. Literature Review

As seen in the methodology section above, the focus of this paper is neither machine learning nor source code analysis by its own, but the combination of this two areas. This means our interest is limited to all those papers which try to or show how it is possible to apply machine learning techniques to the field of source code analysis.

As this combination is quite new the number of published papers is small enough to analyze them all instead of limiting the study to the last few years.

The most important difference we encountered during the analysis of the paper is in our opinion the distinction between fault prediction and fault detection which we are going to see in detail within this section.

#### A. Fault prediction

Most of the papers found with our methodology fall into this category. In some sense all of them try to identify procedures, classes, files or modules as faulty or not without directly detecting the fault. A study performed by Fenton et al. [11] in 1999 criticized the models presented so far, called single-issue models and suggested instead the use of machine learning techniques (to be more precise a holistic model, using Bayesian Belief Networks) in order to get more generic models for predicting faults in software.

Most of these models are based on some sort of metric which allow to conclude about the content of the underlying part of code. Some of the metrics which are frequently used are Cyclomatic Complexity(CC), Halstead complexity or simply Lines Of Code(LOC).

Turhan and Kutlubay showed in the case study [12] that it is possible to establish accurate defect prediction and cost estimation models by using machine learning techniques based on software metrics. Similarly Lounis and Ait-Mehedine [13],

<table>
<thead>
<tr>
<th>Inspec</th>
<th>Compendex</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>S3</td>
<td>31</td>
<td>8</td>
</tr>
<tr>
<td>S4</td>
<td>52</td>
<td>10</td>
</tr>
</tbody>
</table>

**TABLE I**

**LITERATURE SEARCH RESULTS**
they also showed that it is possible to establish quality assessment models through machine learning techniques based on software metrics.

A general idea and another good entry point into studies performed over fault prediction models is the work done by Khoshgoftaar and Seliya [14] where they compare 6 of the most commonly used techniques for fault prediction in detail. They evaluated the various methods using four consecutive releases of a very large telecommunications system, which they don’t call by name (fortunately they don’t specify its size, but the language it is written in, which is PROTEL). The models under evaluation in this case are classification and regression trees in two versions (least square = CART-LS and least absolute deviation = CART-LAD), S-PLUS (data analysis system based on the language “S”), Case based reasoning (CBR), artificial neural networks (ANN), and multiple linear regression (MLR). The evaluation included product metrics like LOC, number of loops or number of calls and execution metrics like time into procedure. As a result they got, that classification and regression trees (in version CART-LAD) performed best whereas S-Plus performed worst on both metric sets.

The same authors redid their study one year later in [15] with slightly different models. In this case study they used CART, S-PLUS, Sprint-Sliq, C4.5, TreeDisc, CBR, and logistic regression. The subject of this study was once again the telecommunications software mentioned above. But the more interesting thing they added to this study is a new measure called “Expected cost of misclassification (ECM)” under which they tried to evaluate the models. This measure should reflect the cost of classifying a source code unit as faulty which is actually fault free (false-positive) together with the cost of classifying a faulty unit as fault free. At the end they recognize that considering only ECM as measure to evaluate the performance of a model is not enough to rank the models. Other factors such as simplicity in model-calibration, complexity of model-interpretation, and stability of models across releases should be considered as well. Even considering just ECM they wouldn’t be able to generalize their results, as they considered only one software system. Nevertheless they pointed out one crucial point in fault prediction systems, which is the number of false positives which is certainly an important factor when deciding over the quality of a prediction system.

One of the commonalities of the two studies is the usage of CBR to build prediction models. The usage of this technique is not limited to the two papers above, but there are some more researchers experimenting with them, even if most of them are done by Khosgoftaar or within his research team. Looking at the assessment of these studies, it is throughout positive. Ganesan and Khoshgoftaar [16] applied this approach successfully to 4 projects written in assembly language. In [17] CBR was successfully applied in a tool called SMART which was tested on a military software and in [18] CBR came out as the winning technique together with C4.5 decision trees. The study presented in [19] instead analyzed the algorithms needed to calculate the similarity between the case library and the actual case, coming to the result that the most easiest ones (like euclidean or Manhattan distance) are most times the best.

Another method that is taken quite often by the studies we analyzed is represented by Artificial Neural Networks (ANN). Yuming et al. [20] successfully applied ANNs to the KC1 project provided by NASA. They were able to show that some metrics are related to fault proneness, but unfortunately as they used only one project they lack of generalizability. In [21] Singh et al. compared traditional strategies to their new solution using ANNs. They were able to show that some metrics are related to fault proneness and therefore their algorithm performed well on medium sized systems. Gondra et al. in [22] tried to compare ANNs with Support Vector Machines (SVM) and got the great result of over 72% of correct classifications by the ANN and the even better result of over 87% of corrects by the SVMs.

SVMs are at least as good as other techniques in fault prediction. At least that’s the outcome of another study [23]. In the authors’ experiment they used 4 different NASA datasets written in 3 different languages (CM1 and PC1 written in C and KC1 and KC3 written in Java and C++). SVMs accuracy across releases should be considered as well. Even considering just ECM they wouldn’t be able to generalize their results, as they considered only one software system. Nevertheless they pointed out one crucial point in fault prediction systems, which is the number of false positives which is certainly an important factor when deciding over the quality of a prediction system.

One of the commonalities of the two studies is the usage of CBR to build prediction models. The usage of this technique is not limited to the two papers above, but there are some more researchers experimenting with them, even if most of them are done by Khosgoftaar or within his research team. Looking at the assessment of these studies, it is throughout positive. Ganesan and Khoshgoftaar [16] applied this approach successfully to 4 projects written in assembly language. In [17] CBR was successfully applied in a tool called SMART which was tested on a military software and in [18] CBR came out as the winning technique together with C4.5 decision trees. The study presented in [19] instead analyzed the algorithms needed to calculate the similarity between the case library and
study (static analysis tool, metric tool, repository and source code history). Despite that change metrics seem to perform well, their drawbacks become uncovered [36]. Change metrics are collected during development. As nowadays time-to-market decreases as well as the size of development teams collecting metrics is often neglected (or simply impossible because of buying COTS). Therefore they suggest a model where static analysis is used to detect faults and code coverage is used to determine which paths are executed and how often. Finally, a Bayesian belief network is used to combine these parameters and estimate the resulting software reliability.

A completely different approach is presented in [37]. The input data for this model is the result after an inspection (total number of faults found during inspection, minimum/maximum/average number of faults found by each reviewer, standard deviation of each reviewer’s findings, and so on). This metrics are used to train a neural network which at the end was able to predict the number of remaining faults in the software.

Having all these different techniques and approaches, there is still the open question on how to obtain the best results from them. The study in [38] showed, that predictions performed better if the classifier is trained only by the mayor faults and not the complete training data. But [39] showed that their model performed better if the dataset is about 80% fault free.

**B. Fault detection**

Despite that machine learning algorithms fit well into the models for defect prediction, there is another big area in which they could be used as well which is the defect detection. The major difference to the models presented above is that detection algorithms really try to identify an actual fault within the code, rather than mark a unit as possibly faulty.

The case study presented in [40] covers a dynamic code analysis of C and Java code. They used a SVM and DTs previously trained with faulty code and its corrected version to identify errors. It demonstrates benefits especially for C in the bug discovery process. A similar approach is shown in [41] where a prototype was developed which based on code metrics and C4.5 decision trees was able to correctly identify design errors with an accuracy of about 90%. In [42] the authors showed an approach using neural networks with back-propagation to detect fault pattern. They tested it using an implementation of a realistic stock trading system and reaching an accuracy of 95%.

Song et al. [43] used the association rule mining(ARM) machine learner to find patterns that are similar or related to previously-found errors. Defects were predicted with an accuracy of 96.6%. In addition, they calculated the effort to isolate and fix the fault compared to other techniques (PART, C4.5, and Naive Bayes). The ARM based isolation-effort prediction had an average accuracy of 93.9% (about 25% higher than the other machine learners). The fix-effort prediction’s average accuracy was 94.7% (about 23% higher).

There exist two approaches to train a machine learner: either batch learning where some predefined subset of the code is used to train the classifier or active learning, where the learner is trained by feedback with every new entered case. The latter was explored by Bowring et al. [44] who said that a program’s behaviour is modeled by its executions. One of the more practical approaches to this is shown in [45] where they developed a tool which is able to improve the performance of the code reviews. Incrementally trained, the tool was able at the end to generalize from a faulty “strcpy” to an incorrect “strcat” (The prototype language was C). The algorithm they used was the normalized comprehension distance in conjunction with k-nearest neighbours. Another practical application is described in [46] where they tried to automatically identify logical errors in student submissions by comparing them to a predefined master solution.

**V. FINDINGS**

First of all, it must be said that the majority of the papers regard defect prediction and only few present defect detection models. This fact must be taken into account when looking at our results.

The reviewed papers present a wide range of machine learning algorithms. Figure 2 shows the most popular classes, i.e. which have been used in at least two different papers. The tree algorithms including forests are most widely used. Reason for this is not only their accuracy but also their relatively easy interpretation. Nevertheless, also induction rule algorithms like IRule as well as support vector machine approaches are used.

All the used machine learning techniques are fed with metrics. Most models base on code metrics. But there are also cases, which make use of design metrics, process metrics and revision metrics. There are even models using behavioural and inspection information for predicting defects in the source code.

As the reviewed studies vary in their used techniques and metrics, they do in their selection of projects for evaluating the presented models. The majority bases the evaluation on the NASA’s metrics repository. This is a big source of information
containing metrics of different projects\textsuperscript{2}. Others either access company repositories or introduce data mining tools, which collect metrics from open source projects. Besides NASA metrics, there are several other domains involved in evaluating the prediction models as shown by figure 3. This indicates, that machine learning techniques are applicable across domains.

All projects considered here are written in several different programming languages. Since the NASA metrics are coming from C and C++ modules, those are the most representative languages. Surprisingly, there has also been used a legacy telecommunication system written in PROTEL. Figure 4 summarizes the used programming languages. This indicates, that machine learning techniques are also applicable across programming languages.

Due to all these different settings in building and evaluating the models, it is difficult to clearly answer the question on how accurate machine learning algorithms are when they are used to predict or detect faults in source code. However, the systematic review of the papers shows us, that given a certain setting the accuracy of the approaches lies generally between 60 and 95 percent. They got accuracies comparable to the standard techniques and in some cases the accuracy was even higher. Furthermore the false-positive rate can be held low.

VI. DISCUSSION

Looking at the results, we know now, that machine learning approaches are well applicable to source code analysis. Even though we were not able to get an overall accuracy rate for machine learning approaches, we have seen that they are performing at least as good as the standard techniques. But what about improvements and missing approaches in this research area?

From the prediction models’ point of view, there is less missing or to improve than with detection models, since the former have been studied already a lot. But as some reviewed studies indicate, the performance of some machine learning approaches depends on the domain and data properties of the data sets. So it would be interesting to have a model, whose parts are easily interchangeable. Meaning, that it provides several machine learning and classification mechanisms, several training sets for different domains and a configuration component for tuning the model according to the current operational settings. For example, Bayesian belief networks are favourable when data is sparse. Now let us assume, one wants to detect faults within an embedded system, from which code features are hard to extract and to interpret. In this case, the model could be configured with a BBN trained on data from the embedded system domain. If the domain or the data properties change, the model can then be just reconfigured. This way, it would be universally applicable and always yield the best possible results.

Although there are already many prediction models applying machine learning, there are only few fault detection models. These models do not cover as many programming languages, domains or machine learning algorithms as the former. So what is missing here, are detection models investigating the variations. Since there is an ongoing debate on whether common code metrics (size and complexity metrics, DIT ecc.) are really fault indicators, some of the existing detection approaches could be improved by not using them but using computed metrics instead, as done by [1]. This would increase the accuracy and lower the false positive rate of the detection models. Furthermore, a comprehensive comparison of the available detection models can also be useful for software testers, which have to decide on using one of them. However, despite all the different settings and flaws, the studies show that machine learning algorithms are able to produce comparable and even better results than standard techniques. This holds especially for studies, which present fault prediction models. These models can reach in almost all studies at least as high accuracy as the compared prediction models based on regression or other mathematical techniques. Consequently, machine learning techniques are applicable across domains and across programming languages. Similarly, this seems to be the case for defect detection models. The assumption that machine learning algorithms benefit to source code analysis

\textsuperscript{2}http://mdp.ivv.nasa.gov/index.html
is definitely strengthened. And thus, our general research question can be answered with a secure yes.

VII. THREATS TO VALIDITY

This systematic literature review is not without threats to internal and external validity. On one hand, the selection and evaluation of the reviewed papers are subjective to the authors. On the other hand, not all relevant papers may have been published and are therefore not considered here. Moreover, there are papers which are definitely relevant to the subject but which are not accessible to the authors, since they are not available in full text or have to be bought.

VIII. CONCLUSION

Since decades, researchers try to establish accurate defect detection and prediction models by analysing source code. Recently, machine learning algorithms have been used instead of the traditional techniques. Within the context of this article the benefit of machine learning to source code analysis for such models has been assessed. The assessment is performed by looking at the accuracy of the models as well as by identifying possible improvements and missing approaches.

For this purpose, a systematic review has been performed. Published, scientific papers which talk about source code analysis using machine learning with the goal of finding or predicting defects have been investigated deeply.

The analysis of the publications has shown that there are many suggested fault prediction models but only few fault detection models. Nevertheless, the papers have shown, that machine learning algorithms are well applicable to both. Most of the models use decision tree approaches, but also support vector machine algorithms, rule induction approaches and regression via classification techniques are used. Besides the variation in the algorithms, there are also variations in the data set features, domains and programming languages of the projects which are used for evaluating the models. This shows that machine learning algorithms are generally cross-domain and cross-programming language applicable. But it must be said that some algorithms are better in certain circumstances than in others.

However, the accuracy of the approaches is in general comparable to or even higher than with standard techniques. Due to the variations in algorithm and domain, it is on one hand difficult to compare the different presented models. On the other hand, it is possible to improve the fault prediction models and one possible approach is briefly mentioned within this review. Also, since there are just few fault detection models and thus not varying that much in algorithm or domain, there is an opportunity for improving and investigating new approaches for defect detection.

REFERENCES


References


References


<table>
<thead>
<tr>
<th><strong>Glossary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Active Learning:</strong> Feedback from each instance is used to continuously update a learning scheme</td>
</tr>
<tr>
<td><strong>ANN:</strong> Artificial Neural Network; a type of machine learning classifiers which are based on mathematical functions.</td>
</tr>
<tr>
<td><strong>ARFF:</strong> Attribute Relation File Format to represent instances for classification</td>
</tr>
<tr>
<td><strong>Batch Learning:</strong> A predefined set of instances is used for building a learning scheme</td>
</tr>
<tr>
<td><strong>CBR:</strong> Case Based Reasoning aka instance based classification</td>
</tr>
<tr>
<td><strong>Code Feature:</strong> A property of code, which in combination with other features allows the classification of the code</td>
</tr>
<tr>
<td><strong>CSV:</strong> Comma-Separated Values; used by the ARFF</td>
</tr>
<tr>
<td><strong>DT:</strong> Decision Tree; a type of machine learning classifiers which learning schemes are represented as trees.</td>
</tr>
<tr>
<td><strong>False Negative:</strong> An instance mistakenly evaluated to a negative answer; here to “no fault”, although it is a fault</td>
</tr>
<tr>
<td><strong>False Positive:</strong> An instance mistakenly evaluated to a positive answer; here to “fault”, although it is not a fault</td>
</tr>
<tr>
<td><strong>Fault Detection:</strong> The direct identification of a fault (aka bug) in source code</td>
</tr>
<tr>
<td><strong>Fault Prediction:</strong> The guess of where most faults reside in source code; used for optimizing resource distribution while testing</td>
</tr>
<tr>
<td><strong>Feature:</strong> A property of an instance based on which a machine learner makes decisions</td>
</tr>
<tr>
<td><strong>Forest:</strong> A collection of decision trees to enhance classification</td>
</tr>
<tr>
<td><strong>Instance:</strong> An input to a machine learner; described by features through which the learner can classify, cluster or associate.</td>
</tr>
<tr>
<td><strong>J4.8:</strong> A decision tree based on the C4.5 algorithm</td>
</tr>
<tr>
<td><strong>JavaCC:</strong> The Java Compiler Compiler; framework for parser generation</td>
</tr>
</tbody>
</table>
**JRip:** A rule based classifier based on the RIPPER algorithm

**Learning Scheme:** A set of instructions, telling how to solve a problem

**LMT:** Logistic Model Tree; a tree based classifier with linear logistic regression models at the leaves

**Machine Learner:** An algorithm, which builds a learning scheme based on instances to solve a problem

**Metric:** A property or attribute of an artifact; here of a piece of software

**Multilayer Perceptron:** A classifier, which belongs to the ANNs, in particular it is a SVM with a special kind of kernel function (sigmoid function)

**Null Pointer:** A reference without memory address assigned to it

**RIPPER:** Repeated Incremental Pruning to Produce Error Reduction; an algorithm for optimizing rule sets

**Rotation Forest:** A classifier, which consists of a decision tree as base classifier and baggs ensembles of random decision trees to a forest

**SVM:** Support Vector Machine; a type of machine learning classifiers which are based on mathematical functions (note: some ANNs are SVMs)

**Static Code Analysis:** The Analysis of a piece of software without running it

**Supervised Learning:** For classification; a learning scheme is built using instances and their output class

**True Negative:** An instance correctly evaluated to a negative answer; here to “no fault”

**True Positive:** An instance correctly evaluated to a positive answer; here to “fault”

**Unsupervised Learning:** For clustering; a learning scheme is built using instances only just of group the similar ones

**WEKA:** The Waikato Environment for Knowledge Analysis; framework providing state-of-the-art machine learners