Static Code Features for a Machine Learning based Inspection

An approach for C

Hannes Tribus

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):
Hannes Tribus 820919-P496
E-mail: hatr09@student.bth.se

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

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
Sweden
Abstract

Delivering fault free code is the clear goal of each developer, however the best method to achieve this aim is still an open question. Despite that several approaches have been proposed in literature there exists no overall best way. One possible solution proposed recently is to combine static source code analysis with the discipline of machine learning. An approach in this direction has been defined within this work, implemented as a prototype and validated subsequently. It shows a possible translation of a piece of source code into a machine learning algorithm’s input and furthermore its suitability for the task of fault detection. In the context of the present work two prototypes have been developed to show the feasibility of the presented idea. The output they generated on open source projects has been collected and used to train and rank various machine learning classifiers in terms of accuracy, false positive and false negative rates. The best among them have subsequently been validated again on an open source project. Out of the first study at least 6 classifiers including “MultiLayerPerceptron”, “Ibk” and “ADABost” on a “BFTree” could convince. All except the latter, which failed completely, could be validated in the second study. Despite that the it is only a prototype, it shows the suitability of some machine learning algorithms for static source code analysis.

Keywords: static source code analysis, machine learning, feature selection, fault detection
# List of Figures

3.1 Weka Explorer .................................................. 9
5.1 CMore: Abstract Execution Flow .............................. 16
5.2 Holmes: Abstract Execution Flow ............................ 19
5.3 Method .......................................................... 20
6.1 Accuracy (Full) .................................................. 25
6.2 Accuracy (Reduced) ............................................. 25
6.3 Average Accuracy per Category ............................... 26
6.4 False Positive (Full) ............................................ 27
6.5 False Positive (Reduced) ...................................... 27
6.6 Average False Positive per Category ......................... 27
6.7 False Negative (Full) .......................................... 28
6.8 False Negative (Reduced) ..................................... 28
6.9 Average False Negative per Category ....................... 28
List of Tables

3.1 Data Type Range .................................................. 8
6.1 Paired T-Test on “Accuracy” ................................. 29
6.2 Results from WEKA for the test experiment ............... 30
A.1 Results from WEKA for all tested machine learners .... 39
B.1 Results of Most Suitable Machine Learners ............... 40
B.2 Performance of the Classifiers on the Test Set ............ 41
# Contents

Abstract

1 Introduction

2 Related Works
   2.1 Fault Prediction
   2.2 Fault Detection

3 Background
   3.1 Static Source Code Analysis
   3.2 Types of Faults
      3.2.1 Buffer Overflows
      3.2.2 Memory Handling
      3.2.3 De-referencing a Nullpointer
      3.2.4 Control Flow
      3.2.5 Signed to Unsigned Conversion
   3.3 WEKA - Machine Learning
      3.3.1 Machine Learning Algorithms
      3.3.2 Attribute Relation File Format (ARFF)

4 Problem Description

5 Proposed Solution
   5.1 CMore
   5.2 Holmes
   5.3 Methodology
      5.3.1 Experimental Design

6 Results and Analysis
   6.1 Open Source Projects
   6.2 Suitability Analysis Results
      6.2.1 Accuracy
      6.2.2 False Positive Rate
      6.2.3 False Negative Rate
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.4 Statistical Significance</td>
<td>29</td>
</tr>
<tr>
<td>6.3 Project Evaluation Results</td>
<td>29</td>
</tr>
<tr>
<td>7 Limitations and Validity</td>
<td></td>
</tr>
<tr>
<td>7.1 Limitations</td>
<td>32</td>
</tr>
<tr>
<td>7.2 Threats to Internal validity</td>
<td>33</td>
</tr>
<tr>
<td>7.3 Threats to External validity</td>
<td>33</td>
</tr>
<tr>
<td>8 Future Works</td>
<td>34</td>
</tr>
<tr>
<td>9 Conclusion</td>
<td>36</td>
</tr>
<tr>
<td>A Experiment Results</td>
<td>38</td>
</tr>
<tr>
<td>B Comparison Results</td>
<td>40</td>
</tr>
<tr>
<td>B.1 Complementary Results</td>
<td>40</td>
</tr>
<tr>
<td>B.2 Conclusions</td>
<td>41</td>
</tr>
<tr>
<td>C Systematic Literature Review</td>
<td>43</td>
</tr>
<tr>
<td>References</td>
<td>51</td>
</tr>
<tr>
<td>Glossary</td>
<td>54</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“It has been just so in all my inventions. The first step is an intuition and comes with a burst, then difficulties arise. This thing that gives out and then that 'Bugs' as such little faults and difficulties are called show themselves and months of anxious watching, study and labor are requisite before commercial success - or failure - is certainly reached.”[29] (Thomas Alva Edison in a letter to Theodore Puskas (18 Nov. 1878))

Despite that Edison’s citation is related to his invention of a storage battery, it reflects pretty well the situation faced nowadays by a developer or a development team when it comes to the lengthily and often annoying process of discovering faulty statements within a piece of software. Computer scientists mostly agree that it will never be possible to deliver bug-free software [15], nevertheless it is important that developers aim for reaching that state, or as it is not reachable at all, come as close as possible by removing most of the faulty statements.

During the history of the development of software several different approaches have been proposed with more or less acceptable results. The range of them start from simple coding rules which focus on not introducing faults, over manual inspection or testing up to more sophisticated ways like tool supported error prediction or detection. Nowadays the latter gain more and more importance as they have the possibility of simplifying the work of the developer by pointing them to the places in the source code where the actual fault is supposed to be. Depending on how well these tools have been developed, they are able to find more or less complicated faults.

The quality gain achieved by using such tools can be the crucial factor for buying one software solution instead of a competing one, and of course the sales figures are the main driving factor for every company. Despite that the driver seems to be clear, the question about the best possible method to find faults in source code has no unique answer yet. Recently scientists start to think about having some sort of intelligent, teachable system, that is not only able to detect faults but to learn from the errors made in previous reasoning to improve its precision in future ones.

The algorithms that can perform such a task can be summarized as machine learning algorithms and exist already implemented and ready to use in some open
source and commercial products. The difficulty in here, and the crucial problem when it comes to combine source code analysis with machine learning is that those algorithms are not intended to work on source code, but on instances. Therefore the problem is how a piece of code can be translated into one or more instances which can then be fed into such a learning algorithm.

The goals of this project are the development of a feature selection model for a representative procedural language (it will be the C programming language for this work) and the implementation of an appropriate parser as preprocessing step for a machine learning based fault detection system. For the latter there will be an evaluation of the possible machine learning algorithms, which are compared and evaluated in terms of accuracy and false positive/negative rates. In order to perform this evaluation the parser will be used to create a data set of instances from various open-source projects.

To reach the goals stated above there is a need of answering at least to the following research questions:

- What are the relevant source code features to classify an instance into faulty or not?
- How can those features be transformed and represented in the machine learners input file format?
- How accurate is the resulting application depending on the machine learning algorithm used?
  - What is its false positive rate?
  - What is its false negative rate?

The following chapters will try to describe in detail the approach found during the research performed in this area. After this short introduction there will be a chapter containing some of the most important works related to this one. Before giving a more detail insight into the gap that this work tries to fill stated in the chapter “Problem Description” there will be a background chapter containing an overview of the most important affected topics. The chapters “Proposed Solution” and “Results” probably being the most interesting one contain detailed descriptions of the developed tools as well as the experimental design and the results of the experiments. Finally there will be “Conclusion” chapter that states the impact of the findings followed by some possible future works. The work is concluded with a list of references and an appendix containing detailed data obtained from the experiments.
Despite that the fields involved in this work, static code analysis and machine learning, are discussed in literature quite often, papers on the combination of these two fields are rare and therefore difficult to find. During the investigation performed to create the knowledge base needed for this work it became apparent that the works related to this one could be divided into two areas which are defect prediction and defect detection [33].

2.1 Fault Prediction

Most of the work done so far in this area deals with the prediction of faults in source code. The basic idea here is to extract properties or attributes from the code which allows to draw conclusions about the presence of one or more faults. Properties in this case could metrics like lines-of-code (LOC) or cyclomatic complexity (CC) or in the case of object orientation even number-of-children (NOC), depth of inheritance tree (DIT) and so on, known as “CK-Metrics” [11].

A study performed by Fenton et al. [10] 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. This was confirmed by the studies of Turhan and Kutlubay [34] and Lounis and Ait-Mehedine [22] who achieved an improvement in the precision of their models by adding a learner to it.

Some of the case studies found try to identify the best possible learning algorithm for this kind of application. So done for example by Ganesan and Khoshgoftaar in [12] and Khoshgoftaar and Seliya in [19] and [20] without coming to a unique result for every case. The most important thing in the latter study is that they introduce the notion of “Expected Cost of Misclassification (ECM)” which is crucial in every prediction (and also detection) system.

Despite that most of the works in the prediction area deal with the code metrics stated above, there exist some different approaches too. Challagulla in [5]
showed that similar results to the one using code metrics can be obtained by using design or change metrics. These metrics could be the number of times a file has been changed in its life, the expertise of the changer and so on. This is supported by Heckman and Williams in [14] and Moser et al. in [24]. Another interesting work in this area is the one performed by Jiang et al. [17] who compared the performance of design and code metrics in fault prediction models. They came to the conclusion that models using a combination of these metric sets outperform models that use either code or design metrics.

2.2 Fault Detection

Despite that the approaches mentioned above obtain more or less good results, they all try to conclude about the presence of errors by considering some kind of meta-values instead of taking the real source code and detect the faults present within. There exist just a few papers about the idea of having a fault detection method using static source code analysis in combination with machine learning and even less about a working tool, however the results obtained by them promise a very good precision and a low false positive/negative rates. The impact of the latter in a real world scenario has been investigated by Baca in [3].

The closest approach to the one performed within this work is the one from Sidlauskas et al. [2] where they successfully combined a machine learner with an interactive visualisation tool developed for that purpose. They showed that it is possible to train a learner with pattern extracted from source code in a way that it is able to adapt it to slightly different situations using the closest training instance in terms of normalized comprehension distance (NCD) computed by the framework they used. Incrementally trained, the tool was able at the end to generalize from a faulty “strcpy” to an incorrect “strcat” (The prototype language was C). Similarly Song et al. in [30] used the “association rule mining (ARM)” machine learner to find patterns that are similar or related to previously found errors.

Burn et al. showed in their case study [4] that support vector machines (SVM) and decision trees (DT), previously trained on faulty code and its corrected version, can be used to identify errors during program execution. Despite that their approach uses dynamic analysis instead of static, it shows that machine learning can benefit to fault detection. A similar approach has been published in [21] where the author describes the development of a tool that is able to detect design flaws during execution time. Finally in [16] the authors successfully applied neural networks with back propagation to detect pattern of faulty code.
Chapter 3

Background

As already mentioned above the task of this work is to find an approach to automatically detect faults within a piece of source code. In order to perform this task it is needed to have a look on how it is possible to achieve such an automation. On the other hand it is important to define what kind of faults or groups of faults it will focus on.

These and some other question will be answered within this section before focusing on the real problem in the section 4.

3.1 Static Source Code Analysis

Static code analysis is one possible procedure to ensure a certain level of quality within a piece of software. During this procedure the code has to pass a lot of formal tests in order to be considered “bug-free”. In reality this is not the case because each static analysis procedure can only be able to determine the presence of a fault and not ensuring the absence of bugs in source code.

Static code analysis can be performed very early in the development process, even before any module or unit tests can be launched. Therefore it can be used on libraries without the need of executing them. It can be done manually which is usually very time consuming or with tool support which comes in very handy when the source becomes very long and complex.

The appearance of static analysis can be very different. Probably the simplest form are pattern matching algorithms where the analyzer (tool or person) tries to identify possibly faulty code fragments based on a pattern “database”. Lexical analyzers improve the performance of simple pattern matcher by transforming the code into a series of tokens which can then be checked against a corresponding lexical pattern “database”. In this chain the next link would be parsing and abstract syntax tree (AST) analysis. The most important work to mention in this category is “lint” [18], a static analysis tool built in 1979. As a last possibility data flow analysis is mentioned in this section. Some types of faults such as buffer overflows and array indexing problems can only be found using a precise analysis of the data flow which is not always possible.
Chapter 3. Background

3.2 Types of Faults

This works aim is the creation of a method to detect faults within source code, however first there is a need to define what a fault is in this context. Despite that there exist also other kinds of faults only the ones described below will be considered in this work. The selection of the groups has been done according to their importance which means according to the harm they will cause and their possibility of being detectable by a computerized tool. Most of these groups have been adopted from the book [6].

3.2.1 Buffer Overflows

A buffer overflow or buffer overrun is one of the most serious faults in source code. It happens when a process stores data outside the location the programmer has set apart for it or so to say the amount of data to store into the buffer is bigger than the size of the buffer. The result of this operation is that the process overwrites some of its data or even the data of some other process, which could lead to the crash of the whole system.

In C/C++ a really common way to introduce this kind of fault is the standard string library with its potentially unsafe functions strcpy and strcat or using the functions to read or write, from or to a file (local or remote via sockets). These are only a few examples where and how buffer overflows can be caused. In fact this kind of fault is one of the most exploited vulnerabilities to overcome modern operating system security.

One example taken from the book is shown in listing 3.1

```c
void trouble () {  
    int a = 32; /*integer*/  
    char line [128]; /*character array*/  
    gets (line); /*read a line from stdin*/
}
```

**Listing 3.1:** Buffer Overflow

3.2.2 Memory Handling

This second kind of error refers to the way how memory is treated within the program, referring to the dynamic memory allocation and deallocation. In C/C++ the statements causing these operations are malloc(or calloc) and free.

The two main errors in this group are the use of the memory after it has been freed (null-pointer) and the freeing of memory that has already been freed before (double-free). Both of them can be used to exploit the system by a buffer overflow attack. A third problem, which is usually not that serious, but a waste of memory is whenever memory is allocated but not used subsequently.

A simple example found in the code is shown in Listing 3.2
3.2.3 De-referencing a Nullpointer

A very common but easy to discover fault is the de-referenciation of a null pointer. This fault occurs whenever a pointer to NULL is used like a pointer to a valid memory address, meaning that a pointer variable is used within the program before it has been assigned to a memory address or the location has been freed before as seen in 3.2.2. De-referencing such a pointer will cause a crash of the program.

3.2.4 Control Flow

This group contains mainly two types of errors. One is the failing of the return value check and the other is the failing of resource release.

Whenever a function returns its status, but it is not checked, the participant variables could be let in an undefined status. A consequence of this could be a de-referencing of a null pointer (seen in 3.2.3) in the following statements.

The problem in failing to release a resource is shown in listing 3.3. If this sample method will exit with one of the two “return NULL” statements, the space allocated for “buf” will not be accessible anymore in the code and is therefore lost. To be more precise this space is reserved but unusable by the process and not released until the termination of the process. Despite that this is not a real error in a normal program, it can be a serious problem for a process that runs during the whole up-time of the host machine.

```c
char* getBlock(int fd) {
    char* buf = (char*)malloc(BLOCK_SIZE);
    if (!buf) {
        return NULL;
    }
    if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
        return NULL;
    }
    return buf;
}
```

Listing 3.3: Resource Release
3.2.5 Signed to Unsigned Conversion

This kind of error can be very difficult for a programmer to detect because it can occur in an apparently checked situation as for example the one shown in listing 3.4.

```c
char src[10], dst[10];
int size = read_int();
if (size <= sizeof(dst)) {
    memcpy(dst, src, size);
}
```

Listing 3.4: Signed/Unsigned Conversion

In this case the value of “size” is checked with the “sizeof” command, so at the first sight there seems to be no possible error. However the expression giving value to “size” is a method that could return every possible integer and not only the positive ones. The method “memcpy” on the other hand is defined to accept an “unsigned int” as third parameter, which means that it expects a positive integer value. Passing a negative one would result in misinterpreting it as a very large positive integer.

However the representation of an integer is 32 bits, no matter if it is signed or unsigned. The difference exists in the interpretation of the bits itself, which means in the case of signed integer, that the leftmost bit is used for the sign (0 means positive; 1 means negative), whereas in the unsigned case it is used to extend the positive range of possible values (see table 3.1).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>32</td>
<td>-2,147,483,648 - 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>0 - 4,294,967,295</td>
</tr>
</tbody>
</table>

Table 3.1: Data Type Range

The real fault in this case is, that if “size” contains a negative value, the check with the “sizeof” command will not fail, but the “memcpy” function will receive a value much higher than the size of the buffers. This at the end will result in a buffer overflow.

3.3 WEKA - Machine Learning

WEKA is a collection of Machine Learning algorithms for data mining tasks, published as open source\(^1\) and written completely in Java. Despite that its purpose is mostly of educational nature, it is equipped with a rich set of functionality,

\(^1\)GNU General Public License
Chapter 3. Background

Figure 3.1: Weka Explorer

accessible through its built in graphical user interface shown in figure 3.1 or directly from any java program that includes its library. Therefore it is perfectly well suited even for more sophisticated tasks in real world problems.

Being open source the actual versions of the source code as well as the compiled versions of it for the most common platforms, including Windows, Mac OS X and Linux\(^2\), can be downloaded from its web page\(^{[25]}\).

The rich set of functionality provided by the WEKA system includes\(^{[35]}\):

- Data preprocessing and visualization
- Attribute selection
- Classification algorithms (OneR, Decision trees, Covering rules)
- Prediction algorithms (Naïve Bayes, Nearest neighbor, Linear models)
- Clustering (K-means, EM, Cobweb)
- Association rules
- Evaluation techniques

Despite that it offers so many different possibilities, the most important points of the above list for this project are the attribute selection and various classification and prediction algorithms. The later are used to analyze and classify/predict

\(^{2}\)The version used within this project is \textit{weka-3-6-2} for Mac OS X
on the input data, while the selection algorithms show which of the numerous features contained in the input vector are involved in the decision making process and which not.

Even though these points are the most important for the operational part of the work, there is a need of the preprocessing step to prepare the data for the classifiers as well as some evaluation techniques which allows to draw conclusions on the quality of the result. These two functionalities are not directly related to the machine learner itself, but they are needed for this work to justify the obtained results.

One of the most important advantages of WEKA over most of the other tools available is that the whole functionality of the system is available either over an intuitive graphical user interface or via an API which allows to be called by a normal (java) program. A second very important advantage that is used within this project is the input data format used by WEKA which is ARFF. Having one standardized file format for all the machine learning algorithms and all the other tasks that can be performed with WEKA it is possible to have one single input data set to try out all the possibilities offered. In this way it is possible within this project to figure out the best suited machine learner for the underlying problem of static source code analysis.

### 3.3.1 Machine Learning Algorithms

The WEKA framework categorizes its machine learning algorithms into 7 different categories, which are “Bayes”, “Trees”, “Rules”, “Functions”, “Lazy”, “Misc” and “Meta”. This section will not explain each algorithm present in the various categories, but it gives an overview based on these groups and mentions their most important algorithms. Detailed information on the algorithms can be obtained from the book [35].

**Bayes**

This category contains mainly the bayesian network and various types of na"ıve bayes networks. Basically a bayesian network is a directed acyclic graph whose nodes are variables, or in this case attributes, and the edges represent the conditional probability associated with them. The na"ıve versions of them are similar, but based on a strong independence assumption. This means that they assume that each feature in the set is independent from any other feature.

**Trees**

The most important trees implemented in WEKA are the “ADTree”, “Decision-Stump”, “LMT”, “RandomForest” and “J48”. The latter is the java implementation of the “C4.5” which is well known in literature [27]. Logistic model trees
Chapter 3. Background

(LMT) are trees that apply logistic regression on their leave level, whereas a decision stump is a tree with just one level. Finally the ADTrees are alternating decision trees, which alternate between decision levels (known values) and prediction levels (probabilities).

Rules

The algorithms present in this category include the “Decision Table”, which bases its decision on simple rule tables and “JRib” which implements the RIPPER [8] algorithm in Java. Furthermore it contains “NNge” which is a rule based implementation of the nearest neighbor algorithm.

Functions

This category contains mainly various algorithms based on a regression function like the linear or the logistic (These are called “Logistic”, “LinearRegression”). Both functions exist as “simple” versions (“SimpleLogistic” and “SimpleLinearRegression”). Beside these group of learners it contains also the neural network “MultiLayerPerceptron” which is based on back propagation.

Lazy

In WEKA “lazy” is basically another word for nearest neighbor. Therefore this category contains the nearest neighbor algorithms “KStar”, “Ib1” and “Ibk” where the latter two differ only in the number of neighbors taken into consideration, while the former is based on a generalized distance function.

Misc

This category contains only two classifiers which are “Hyperpipes” and “VFI” (Voting Feature Intervals). These methods try to group the attributes of the training instances into intervals or ranges. These groups are then used in a nearest neighbour like fashion to determine the results.

Meta

The classifiers present in this category are not completely new but they use one (or more) of the classifiers present in the categories above and try to enhance its performance by using techniques like bagging, boosting or simply by combining classifiers. The most important ones in here are “Bagging” and “AdaBoostM1”. The latter is a boosting algorithm, that can dramatically increase the performance of a classifier, but may lead to overfitting [28].
3.3.2 Attribute Relation File Format (ARFF)

The attribute relation file format is a standardized format to represent data together with its name and type. As this format is used to feed every learner present in the WEKA system it decouples data from the learner and creates in that way the possibility of testing multiple learners as well as selection algorithms on a single data file to determine the best suited method for the actual problem.

Each ARFF file contains two distinct sections, that are called header and data. The former contains the name of the relation and a list of attributes with their associated types, whereas the latter consists of the real data vectors stated in a CSV(Comma Separated Values) style. A short sample of such a file structure taken from the book[35] is shown in listing 3.5

```
@RELATION iris

@ATTRIBUTE sepal length NUMERIC
@ATTRIBUTE sepal width NUMERIC
@ATTRIBUTE petal length NUMERIC
@ATTRIBUTE petal width NUMERIC
@ATTRIBUTE class {Iris-setosa, Iris-versicolor, Iris-virginica}

@DATA
5.1,3.5,1.4,0.2, Iris-setosa
4.9,3.0,1.4,0.2, Iris-setosa
4.7,3.2,1.3,0.2, Iris-setosa
...
```

Listing 3.5: ARFF Sample

The header file in this case defines that the first four attributes are of type “NUMERIC” meaning that the values in the data-part of the file can only consist of numbers (or “?”), which in any case signals an unknown value) either floating points or integers. The last attribute is of type “nominal”, which means that its values can only consist of one of these stated in the attribute set.

There exist some other possible types such as “STRING” for example where every possible string can be its value. The selection of the types of the attributes determines the set of possible machine learners that can work with it. The type “STRING” for example is not possible for most of the learners simply because they cannot handle it. Other learners instead can not deal with missing values or have problems when they have to deal with “nominal” ones that either have too few or too much different possible values.
Chapter 4

Problem Description

The topics of the above sections are mainly the different types of faults, a machine learning framework (WEKA) and the analysis of static source code. Having this knowledge in mind it is time to define what the real problem, focused within this work, is.

Despite that, from the basic theory of computing we know that it is not possible to find and remove all bugs present in a reasonably large piece of software[15], each developer tries to reduce their presence to an acceptable minimum. The problem to face when searching for faults in source code is how this can be performed in the most efficient way.

Static source code analysis is one possible solution to this problem with all its pros and cons stated above. The most important advantage of this technique and therefore probably the reason why it has been chosen so often in literature and for this work too, is that it can be applied very early in the development process, namely as soon as there is some code accepted by the corresponding programming language compiler.

Even though it seems to be the best solution to reduce the number of bugs in source code, it is a very annoying and time consuming activity when it has to be performed manually. However when it can be done with the support of a well suited tool it can be done in a more efficient way and so contribute to the overall quality of the resulting software.

Nowadays the most widely used tools on the market are based on pattern matching, which means that they are scanning the source code trying to identify faulty patterns by comparing them to the ones present in their database. The problem with this approach is its inability to learn and adapt its behaviour whenever a classification fails. This behaviour can lead to high number of false positives which can highly devalue the usefulness of the tool (The impact of a relatively high false positive rate has been investigated by Baca in [3]).

That is actually one of the points for having support of a machine learning framework, which should be able to learn from previous classifications and therefore reduce the number of false positives while obtaining a high rate of true positives. The literature has shown in some papers that machine learning algorithms can contribute to the quality of the results of static source code analysis tools.
Chapter 4. Problem Description

The systematic literature review [33] performed prior to this work has shown that most of the work performed so far deals with the prediction of faults in source code using some kind of metrics (described in more detail in chapter 2 “related works”), while there exist just a few papers about fault detection. The latter is the area where this work is placed and the gap it tries to fill up is the absence of a well suited method or approach to integrate a machine learning classifier into the static analysis of source code.

The outcome of this work should be one possible solution of transforming a standard C-language source code file into a format usable within the machine learning framework. As stated already above this work will make use of the WEKA framework, so the output of this step will be in form of an ARFF-file. Having this conversion from source code to ARFF it should be possible to define what kind of attributes or features are needed to identify the target faults within the source and what kind of machine learning algorithms are really suitable for this kind of operation.

These two questions are really crucial for the present work, being the main drivers for the underlying investigation. In order to find the minimal or at least one minimal set of features needed to identify faults in source code the experiment has to be executed more than once. Starting with a large feature vector extracted from the source code. Using the outputs provided by WEKA it should be possible to understand which are the features that either are not used at all by the classifiers or the ones that are used but could lead to misclassification. Both of them will be removed in the subsequent iterations, which will hopefully increase the precision of the suitable learners.

To answer to the second question it is needed to evaluate the possibility of achieving a high fault detection rate with a small number of errors. This means that the evaluation has to take care of the precision and the false positive/negative rate of the resulting classifier to define its quality.
As seen in the chapter above the problem to face is to propose a solution for feeding a machine learner with the data obtained by static source code to create an intelligent fault detection method. This problem can be split into three subproblems which are:

1. How to transform source code into a suitable format for the machine learning classifier?
2. How to train that classifier? Which data has to be used to train it?
3. What are the features needed by the machine learner and which learner fits best to the problem?

These are the main problems faced during this work and the solutions found will be described in detail within this chapter. First it will present the prototypes developed for the final solution then the method in which they are utilized to produce the expected results.

Before coming to these sections there is one decision that had to be taken and needs to be justified before starting to talk about the real solution itself. The choice of the machine learning framework decides already about the file format of its input files and needs to be defined therefore in advance. As already expectable from the background chapter 3.3 the favorite choice was the WEKA data mining framework. The main reason why it has been chosen beside its great popularity in recent papers and its open source license was that it offers a wide variety of different learners. Furthermore all the learners can be used with the same kind of input file and using the same GUI which is already provided with the download of the framework.

5.1 CMore

The first and probably the most complex prototype developed for this solution is “CMore”. This program is able to decompose a program written in C into a set of basic functions and procedures. The basic idea behind its functionality is
Chapter 5. Proposed Solution

Figure 5.1: CMore: Abstract Execution Flow

that it should be able to follow the execution flow of a program and capture the state of all the variables involved. Having all the states it is possible to detect several kind of possible memory access faults which are null-pointer-references and misuse of memory (see 3.2 for details)

The state diagram shown in figure 5.1 sketches the basic steps performed by “CMore” to achieve this behaviour.

It mainly works in two steps, starting with a clean and divide preprocessing step followed by the real state tracking step. The former activity is performed by a set of classes of which the main one is called “FileWalker”. This class reads a whole file as a string of bytes into its memory, which is needed to remove every unused stuff from the file such as comments and macros\(^1\). Afterwards the input is subdivided into the various functions and procedures and stored into a central depository (called “Brain”) for further use in the next steps.

Despite that its main functionality is to keep track of all the possible functions and procedures, the repository contains a list of all known types, structures, constants and global variables within the analyzed file. Furthermore there exists the possibility of including other files into the execution flow, which in C is done like this: \texttt{#include "something.h"}. Whenever the “FileWalker” encounters such a statement it launches a new instance of itself which is then responsible to decompose the included file.

At the end of this step the central repository is aware of all the possible function names together with their parameter list and all the possible types, constants and globals.

Immediately after the preprocessing step the flow following step starts. In the context of “CMore” the main classes involved are the “FormalExecuter” and the

\(^1\) As this is still a prototype the expansion and analysis of macros was left over as a future work
“ExpressionMelter”. The latter is responsible to evaluate possible mathematical expressions and replace variable names with their according values whenever possible. The former class follows the execution flow and determines the nature of each statement, which could be in this case an assignment, a declaration, a function call or something else. This class has to be aware of the information gained and invoke the protocolling of the state of the line when needed.

Protocolling in this context means to invoke the writing to a file in the format the machine learning framework expects it. As seen in the background section 3.3 and shortly justified above the framework is WEKA and therefore the file format has to be an ARFF-file.

```
@RELATION c m o r e

@ATTRIBUTE operation {Functioncall, Assignment, Usage}

@ATTRIBUTE left_type {FILE, Function, header, char, int, Reference, Simple, void, ...}

@ATTRIBUTE left_isRef {true, false}
@ATTRIBUTE left_isNull {true, false}
@ATTRIBUTE left_isAss {true, false}
@ATTRIBUTE left_isUse {true, false}
@ATTRIBUTE left_isChk {true, false}

@ATTRIBUTE right_type {FILE, Function, header, char, int, Reference, Simple, void, ...}

@ATTRIBUTE right_isRef {true, false}
@ATTRIBUTE right_isNull {true, false}
@ATTRIBUTE right_isAss {true, false}
@ATTRIBUTE right_isUse {true, false}
@ATTRIBUTE right_isChk {true, false}

@ATTRIBUTE par1_type {FILE, Function, header, char, int, Reference, Simple, void, ...}

@ATTRIBUTE par1_isRef {true, false}
@ATTRIBUTE par1_isNull {true, false}
@ATTRIBUTE par1_isAss {true, false}
@ATTRIBUTE par1_isUse {true, false}
@ATTRIBUTE par1_isChk {true, false}

 [...] 

@ATTRIBUTE orig_stmt STRING
@ATTRIBUTE err_free {true, false}

@DATA
Assignment, int, ?, true, true, false, false, Reference, ?, true, false, false, true, false, ...
...
```

**Listing 5.1**: Snapshot of the final ARFF

Listing 5.1 shows a snapshot of the final ARFF file used to get the results for the experiment described below. Its structure is quite simple and simulates the typical structure of a possible statement. The first attribute states the kind of operation the statement under investigation adheres to, which could be either an “Assignment” a “Usage” or a “FunctionCall”. Depending on this attribute and the real statement, one or more of the groups stated below it (“left”, “right”, “par1”-“parX”) are filled. A group in such an ARFF file contains 7 attributes and is separated by a blank line in listing 5.1 to make them identifiable more easily. The last two attributes contain the real statement needed to identify the faulty line in the code and the “error-free” flag.
Chapter 5. Proposed Solution

The following examples should clarify the behaviour of the single groups and why they are filled according to the statement:

\[ a = b; \quad \text{-- > In this case “a” is used to fill the “left” and “b” to fill the “right” group} \]

\[ \text{aProcedure}(a); \quad \text{-- > In this case “aProcedure” is the “right” and “a” is used to fill “par1”} \]

\[ a = \text{aFunction}(b,c); \quad \text{-- > In this case “a” is “left”; “aFunction” is “right” and “b”, “c” are the two parameters} \]

The standardized structure of the ARFF file makes it easy later on to reconstruct a statement using the instances present in the file. However one statement can produce multiple instances in the result. The first case stated above would for example produce one “Usage” instance for variable “b” and then an “Assignment” instance for the assignment containing variables “a” and “b”.

One group of attributes in this file contains the following properties:

- **Type** it contains the real type of a variable, if that type is known.
- **Name** it is used to hold the name of the function in the case of a function call, or in some cases the name of the actual variable
- **isRef** Flag that tells if the variable is a pointer or a simple type
- **isNull** If the variable is a reference it tells if it points to NULL
- **isAss** Tells if the variable has been assigned to some value or address
- **isUse** Tells if the variable has been used in any statement before
- **isChk** Tells if the variables value has been checked before

5.2 Holmes

This program is not so complex and also not so important for the method itself, but it gains some importance when it comes to the task of training the classifier. “CMore”, the program described above, is not able to understand what an error is, but it is able to keep track of the execution flow and provides later steps with data. These steps now get only a list of instances and do not see the real source code anymore, but they have to identify some instances as faulty whereas others as fault-free. As the result of a conversion of a piece of source code into ARFF can easily lead to several thousands of instances this could be a very annoying process.
Figure 5.2: Holmes: Abstract Execution Flow

“Holmes” has been developed to simplify this process. It is able to perform a line wise comparison of two ARFF files and separate matching lines from the rest for any two ARFF files coming from “CMore”. The real purpose of having a program performing this task will be explained later in this chapter when the methodology is explained (see 5.3).

The basic execution steps of “Holmes” are shown in figure 5.2. At the beginning it reads two versions of an ARFF file into its memory and sorts them separately according to a predefined attribute consisting of the filename and the statement which lead to the creation of the line.

Having these two files arranged in that way it is possible for “Holmes” to determine if an instance (a line in the file) is present in either one of the two files or in both. The latter case then needs to be analyzed further to determine if they differ in some attributes or if the content of the line is exactly the same. The latter is actually the only case in which the instances can be considered to be the same, therefore “Holmes” will output them into a separate file. All the real differences get an attribute appended which determines their origin (the input file they are coming from).

As already mentioned above, the real purpose will be clearer after section 5.3, however the idea behind is to create the needed instances to train a machine learning classifier. Whenever this learning task should be performed there is a need of having faulty instances together with its corrected versions, which can be obtained from “CMore” described above in section 5.1. Despite that this seems to be enough at first sight, “CMore” is not searching for faults and will therefore output all the data it is expected to output for both, the faulty and the corrected version. These two output files will of course contain many duplicate
lines, as the correction of a fault does usually not cause the whole program to change. “Holmes” in this scenario simplifies the detection of the instances that are relevant for the training step.

5.3 Methodology

Now that the most important programs used during the work are introduced it comes to the experiments where they are used in conjunction to gain the results. The general structure of the process applied in order to get them is shown in figure 5.3 and described more in detail below.

Starting from the basic idea of combining modern machine learning algorithms with the field of static source code analysis the first step was to investigate what has already been analyzed and published and where there could be a possible gap to fill. In order to get this information a deep study of all the possible (and freely available) papers published in this area has been performed and summarized in a systematic literature review [33].

Despite that this review has been written prior to this thesis it can be seen as the first step and the source of the gap that this work tries to fill (The most important findings of the review have been stated in the related works chapter 2). From the results of this review the main objectives and its correlated research questions have been drawn (These are stated in the introduction chapter 1 and described more in detail in chapter 4 “problem description”).

At this point the problem has been clearly defined in terms of research questions and objectives and lacked of possible solutions. The proposed one in this case was the development of a program that is able to keep track of the status information of every variable in an execution flow. The prototype capable of
performing this task has been called “CMore” and is described in detail above.

The next step in this flow was to define a strategy to evaluate the possibilities behind this solution and to find possible improvements. Therefore the idea was to evaluate the suitability of the machine learning algorithms in this case by testing them on real open source projects. In order to be suitable for the purpose of the work the chosen projects needed to have some kind of bug-repository and their source has to be accessible via some common version control system\(^2\). Projects that fulfill these requirements can be found either on Sourceforge[31] or on Google Code[7].

Having these projects it was possible to create different “CMore” output files for the different versions on the version control system in order to obtain source containing a certain fault and code without the fault. Whenever it was not possible to locate the fault in the source code, or the bug repository did not provide enough information the solution was to inject a suitable fault directly in the source code to come up with the two different versions. These two versions were then used as inputs for “Holmes” to get the interesting instances out from the two files produced by “CMore” that really identify the fault. These instances have then been collected to obtain a huge file of instances needed in the next step (A more detailed description of “CMore” and “Holmes” can be found above in this chapter).

The result from the previous step was a huge file of instances that has then been used to train the machine learner. In the case of WEKA it was now possible to train many different learning algorithms with this file, and that is one of its advantages that made it suitable for this work. The WEKA output of the various learners has then been collected and analyzed more in detail. The results of this analysis will be shown in the results chapter 6.

To test if all the features in the set used for the analysis are needed it has been reduced by some hopefully unnecessary attributes and then re-evaluated using the WEKA framework. The results of this part of the experiment will be shown as well in the results chapter 6.

As a last step the best machine learners have been tested on the complete output of one of the open source projects previously used to generate the training set for the learners, to evaluate its performance on a real data set. This experiment should show that the classifiers trained are not working only in the suitability environment but also in a real world scenario.

### 5.3.1 Experimental Design

The main idea behind this study is to show that the presented approach works for the field of fault detection. In order to show this a test scenario will be created and subsequently analyzed and evaluated.

\(^2\text{SVN[32], CVS[9] or GIT[13]}\)
Chapter 5. Proposed Solution

X O

X = Treatment → fault detection
O = Observation → Accuracy, FP, FN
Subjects → Operations (Faulty/Correct Instances)

The best suited experimental design for this kind of evaluation is the “One Shot Case Study” whose structure is sketched above. This design starts directly with the treatment omitting the pre-observation/testing and postpones the testing to the end of the study.

In this case the treatment is the whole training procedure of the machine learners which includes the creation and manual classification of the instances used for training. As soon as the training is completed the suitability analysis is performed which returns the list of suitable learners by analyzing their accuracy, false positive rate and false negative rate. The project evaluation study follows the same procedure, however in this case it can be seen as a further testing or observation.
Chapter 6

Results and Analysis

In this chapter the most important results are shown in terms of tables and graphs. Due to space constraints, some of them are truncated to focus on the interesting parts for the conclusions drawn, but in any case a complete reference of the data obtained by the different experiments can be found in the appendix A. Wherever a truncation has been applied there will be a reference to the corresponding data set in the appendix.

From the methodology it is possible to see that two different experiments have been performed whose results will be explained in two different sections within this chapter. However before coming to the real results there will be a short introduction of the projects used to come to the results.

6.1 Open Source Projects

As already stated shortly in the methodology the best sources for projects fulfilling the requirements to be considered can be found either on Sourceforge[31] or on Google Code[7].

The requirements for the projects are the following:

Language The programs should be written entirely or at least mostly in C programming language

Availability The prototypes need real source code, so it should provide access to that using one of the following common version control systems SVN[32], CVS[9] or GIT[13]

Bug Repository The programs should provide access to a bug database if possible

Project Size As the programs used to extract the data are only prototypes the programs under investigation should be of small or medium size

During the search it was possible to come up with 6 candidates for the deeper investigation, which are:
Chapter 6. Results and Analysis

Algoview is a small program that is able to visualize a program as a graph (Sourceforge).

Boxc is a vector graphics language to simplify their creation (Sourceforge).

ClamAV is an anti-virus program developed for the Linux environment (Sourceforge).

PocketBook is an open source firmware for e-book readers (Sourceforge).

FalconHTTP is a HTTP server offering several parallelism technologies (Google Code).

Siemens Programs is a collection of programs with several versions where for educational purposes some faults have been introduced (Other [26]).

6.2 Suitability Analysis Results

From the programs stated above it has been possible to extract correct and faulty instances using the two programs “CMore” and “Holmes” as described in the methodology section (see 5.3).

In total it was possible to get 496 instances containing faulty and correct versions of instances and some correct instances reflecting the most important standard situations. The latter have been added in order to reduce the false alarms on correct standard situations in source code (important for the second experiment). The results of the various WEKA machine learning algorithms when trained with these instances are stated below.

According to the literature review performed beforehand the crucial values to measure the performance of a static analysis method are:

Accuracy which is the percentage of correctly classified instances

False Positive rate is the percentage of instances the method signals as faulty among the correct ones

False Negative rate is the percentage of instances signaled as fault free among the faulty ones

In order to obtain these measures the instances are loaded into WEKA using the WEKA explorer and evaluated against the possible machine learning algorithms. In order to get a representative evaluation a cross-validation with 10 folds has been chosen. This means that the whole data set is randomly divided into 10 different sets of which 9 are used to train and the remaining to test the classifier. This procedure is then repeated 10 times, so each set will be once the test set.
Chapter 6. Results and Analysis

The whole procedure has been performed twice to reduce the set of features involved in the process. The purpose of this should be to understand which features are really needed to identify the faults in the source. After the first run the log-data of the classifiers has been analyzed to understand which features are not used at all or which of them could probably lead to incorrect classifications when used in the decision process of the learners. Those features have been reduced in the second run, trying to improve the learners performance.

This experiment is intended to prove the suitability of certain machine learning algorithms to the underlying data set.

6.2.1 Accuracy

Despite that it took some time, WEKA made it quite easy to test and compare about 71 classifiers, which means all the classifiers that are applicable with the kind of data produced by “CMore”. The resulting ranking in terms of accuracy is shown in figure 6.1.

![Accuracy (Full)](figure6.1.png) ![Accuracy (Reduced)](figure6.2.png)

Figure 6.1: Accuracy (Full)  Figure 6.2: Accuracy (Reduced)

It is truncated to show only the ones that performed better than 90%, however the complete data set is listed in the appendix A, table A.1. In this graph among the 7 classifiers performing better than 95%, three are based on nearest neighbor algorithms (“Ib1”, “Ibk”, “NNge”), three are tree based (“LMT”, “RotationForest” and “ADABoost” performed on a “BFTree”) and the function based neural network “MultiLayerPerceptron”.

Despite that trees perform quite well on the given data set the output of the attributes they used showed most of the times that they tried to base their reasoning on the real data type of the involved variables. To test their reaction
Chapter 6. Results and Analysis

Figure 6.3: Average Accuracy per Category

and the one of the other learners in the second step of this experiment these features have been removed. The output in terms of accuracy of this experiment is shown in figure 6.2 (complete data set is listed in the appendix A, table A.1).

As expected the performance of the trees has been reduced and also the previously well performing “ADABOost” was not able to push it back to the top. However it was possible to increase the performance of the “MultiLayerPerceptron” to be the overall highest in the experiment together with the “NNge” nearest neighbor learner.

This impression is acknowledged by figure 6.3 which shows the average accuracy per category (These categories are taken from the categories used in WEKA to separate the various classifiers). It shows that almost all categories could improve by not taking in consideration the real name of the involved types, except the trees. The graph shows also that functions could not improve, but this is mainly due to a high loss in accuracy of one learner in this category called “SMO(Puk)" which decreased from almost 95% down to 66%. On the other hand one of the new best performing learners descends from this category too.

6.2.2 False Positive Rate

A similar impression can be gained from the false positive analysis results shown in figure 6.4 for the full feature set and figure 6.5 for the reduced one. Among the bests for the full data set there is still the nearest neighbor algorithm and the “MultiLayerPerceptron” together with the ADAboosted “BFTree” and some other trees like the “LMT” and a version of the “J48”. As before when using the reduced feature set the trees will become worse, but the nearest neighbor algorithms and the “MultiLayerPerceptron” gain.

This impression is again acknowledged by the averages per category which show that all except the trees could gain from the reduced feature set (see figure 6.6).

Interesting in this case is that the “NNge” nearest neighbor algorithm despite

1Sequential minimal optimisation using the “Puk” kernel (Support Vector Machine)
that it is among the best regarding accuracy it is not as good when it comes to false positive rate and also the reducing of the features can not significantly increase its performance. This in combination with its good accuracy in fact means that this classifier should have a low value when it comes to false negative rate

6.2.3 False Negative Rate

According to the results gained by Baca [3] it is very important for a static analysis tool to have a low false positive rate. According to that results if a tool signals to many false alarms the developer tend to ignore all signals or even worse, introduces new faults at the position the tool signals them.

On the other hand it is also important to have a low false negative rate which means that the tool finds most of the faults present in the code. A tool in this area needs to find an acceptable trade off between false positives and false negatives.

The results and the ranking of the analysis in terms of false positive rate are
Chapter 6. Results and Analysis

As already expected the best performing algorithm according to this measure is the “NNge” due to its quite bad performance in the false positive measure. However it is worth to notice that the trees perform quite well regarding this measure and most of them are able to gain from the reduced feature set. Unfortunately as they are not as good regarding accuracy it means that their average false positive rate is high.

Generally this rate suffers from the reducing of the feature set, as the number of learners being below 8% decreases. The average values of the various categories do not change so much due to the decreasing as shown in figure 6.9. The only real notable difference is in the category of the rule based learners, but they have not been part of the top learners before the reduction and they could not gain
enough to be among them after, except of course for “NNge” which could win this measures ranking.

### 6.2.4 Statistical Significance

Reading the sections above, there exist some learners that seem to dominate the rankings and therefore could be suitable for the project evaluation, which will be the subsequent step. In order to prove this intuition the WEKA framework offers the possibility to run significance tests on the various data sets.

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7)</td>
<td>53.73±0.60</td>
<td>93.94±3.78</td>
<td>95.91±2.76</td>
<td>95.02±3.18</td>
<td>94.27±3.08</td>
<td>94.21±3.16</td>
</tr>
</tbody>
</table>

- (1) rules.NNge
- (2) lazy.IBk
- (3) meta.AdaBoostM1
- (4) meta.RotationForest
- (5) meta.RandomCommittee
- (6) trees.LMT
- (7) rules.ZeroR

**Table 6.1:** Paired T-Test on “Accuracy”

In order to perform this test the “WEKA-Experimenter” has been loaded with the training data set used above and with the 6 learners that previously performed best together with the “ZeroR” as reference learner (“ZeroR” knows only one result; either it classifies every instance as false or every instance as true depending on which one is more frequently present in the training set). To be consistent with the experiment above the validation technique used is still 10-fold cross validation, independently for each of the 6 learners. This procedure is repeated 10 times each one independent from the other ones. These iterations are needed to randomize the creation of the folds, so the values created in each iteration are slightly different, which is reflected in standard deviation.

The accuracy value of the resulting data set has then been used to perform a T-test to obtain the significance of the values obtained above. The result of this analysis is listed in table 6.1 and confirms the intuition gained above. The table shows the average accuracy obtained by evaluation the 10 iterations and the according standard deviation (e.g. 95.91±2.76).

### 6.3 Project Evaluation Results

From the previous experiment it was possible to obtain the best performing machine learning algorithms. However those results have been achieved by using only

---

2In WEKA the tool able to perform this operation is called “Experimenter”

3The “MultiLayerPerceptron” has not been considered here due to its vast run time
well sorted instances to have pair of two or more instances showing the difference between correct and faulty instances.

The purpose of this experiment instead is to run the best performing machine learning algorithms against a complete data set obtained from the application of “CMore” to one of the open source projects listed above. The chosen project in this case was “FalconHTTP” from which “CMore” was able to extract over 4000 instances. Using fault injection 44 instances have been marked as faulty, the remaining ones are considered fault free\(^4\).

For this experiment the whole set of instances used for the first experiment has been used to train the six previously best performing classifiers and the new generated set of instances is used to test their performance. The experiment has been performed for both, the full and the reduced feature set. The results obtained from WEKA for these six learners is listed in table 6.2 sorted by their accuracy for the full feature set.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Category</th>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Corr</td>
<td>Incorr</td>
</tr>
<tr>
<td>Ibk (LinearNN)</td>
<td>Lazy</td>
<td>98,434</td>
<td>1,566</td>
</tr>
<tr>
<td>Rand.Com.(RandT.)</td>
<td>Meta</td>
<td>97,979</td>
<td>2,021</td>
</tr>
<tr>
<td>LMT</td>
<td>Tree</td>
<td>97,423</td>
<td>2,577</td>
</tr>
<tr>
<td>NNge</td>
<td>Rule</td>
<td>96,059</td>
<td>3,941</td>
</tr>
<tr>
<td>RotationForest</td>
<td>Meta</td>
<td>94,543</td>
<td>5,457</td>
</tr>
<tr>
<td>MultiLay.Perc.</td>
<td>Functions</td>
<td>91,359</td>
<td>8,641</td>
</tr>
<tr>
<td>AdaBoost(BFTree)</td>
<td>Meta</td>
<td>77,160</td>
<td>22,840</td>
</tr>
</tbody>
</table>

Table 6.2: Results from WEKA for the test experiment

The most important result gained from this experiment regarding the whole work described in this paper is that five out of the six tested classifiers showed an accuracy of over 90% which makes them applicable to real world problem data sets.

Despite that in the previous experiment it has been possible to observe a gain in accuracy and false positive rate by reducing the feature set, in this case only the “MultiLayerPerceptron” was able to profit. This learner augmented its accuracy from about 91% to over 99% reducing its false positive rate from over 8% to 0.5%. Important to notice in this table is the good performance of the “LMT” tree in both cases, whose performance in the previous experiment could not overcome the ones of “Ibk” or “NNge”. With its false positive rate of 2.5% in both cases it is among the overall top three learners for this experiment.

Probably the most surprising result from this experiment is the poor performance of the “ADABost” classifier\(^1\) with both feature sets. The real difference between the data set used for this experiment and the one used in the previous one (which is actually the training set for this experiment) is the relation between faulty and correct instances. The latter should exceed the former in a real world project, which means in other words that there should be way more

\(^{4}\)this is an assumption and can not be proved to be correct\(^{15}\)
correct statements in a project than faulty ones. Actually the test set contained over 4000 instances of which only 44 can be considered faulty. According to [35] and [28] boosting can in some cases dramatically increase the performance of the learner, but there is the risk of over fitting it, which could be the cause of this huge decrease.

As a last remark it is noticeable that the learners in both cases show a relatively high value for the false negative rate. This happened because if a learner was not able to identify four out of the 44 faulty instances its false negative rate is already on 0.083 (two out of 44 results in a false negative rate of 0.043).

Despite that only 44 are known to be faulty instances, not all of the fault free instances have been checked to be really fault free. so the rate of false negatives could differ in reality. This behaviour is originated by the assumption that the test program contained only the injected faults.
Chapter 7

Limitations and Validity

Despite that these results show, that this method could be a possible approach for future static analysis tools there exist some limitations and threats to the validity that need to be mentioned.

7.1 Limitations

The most important limitations come from the used programs, and their ability to track enough information. “CMore” being a prototype developed for this work is not yet able to discover enough information to allow the learners in the second step to identify the faults mentioned in the background section.

Up to now it is able to keep track of allocations and deallocations, meaning that it can handle every kind of memory access errors. So it is perfectly well suited to find double allocations or double free as well as null pointer de-referenciations for variables and access to unopened files. Furthermore it is able to deal with the most fault prone sting functions such as “strcat” and “strcpy”. The latter can cause buffer overflow faults in some cases, but to discover the real buffer overflow faults the prototype would need to keep track of the sizes of the buffers, which it is prepared to do, but up to now it has not been finished and therefore left over to future works. The same counts for the signed to unsigned conversion errors, which would be easier to discover than real buffer overflow errors, but in this first prototype version have not been handled.

Types of faults are a limitation of the prototype, however there exists a limitation regarding the method itself. In order to work properly it is necessary that the analysis starts at some head method like the “main”. The success of this approach highly depends on the completeness of the data it can get. In other words this means the more data it can collect the more precise the output will be. So if this method will ever be implemented in a tool there is a need of finding a way to analyze a smaller block, for example a function, where the tool supplies the information of the surrounding variables and constants, and the method just analyzes the basic block given to it.
7.2 Threats to Internal Validity

The first threat to internal validity comes from the selection of the projects. Despite that they come from several categories including drivers and graphical tools, they are all small or medium sized and so potentially not representative for all the possible projects.

A second threat regards the choice of the machine learners. Despite that all the applicable learners for the resulting data set have been tested and taken into consideration for the final results, their choice is limited to the learners provided by the WEKA machine learning framework. Even if it was possible to run the test data against about 70 learners there might exist others that have not been taken into consideration for this study.

In spite of using only the best performing machine learners in the second experiment could already be seen as possible threat, the more important observation is, that it has been done using only one single project. Probably the results obtained by using this project can not be generalized to all the open source C projects.

7.3 Threats to External Validity

Threats in this category refer to the question of the generalizability of the results. In this work it has been tried to prove that this method is applicable to programs written in the “C” programming language by using different learners and reevaluate the method on a real world program.

Despite that C is a very important and wide spread language its results can probably not be generalized to any other language. Most of the results will probably hold for other procedural languages but, when it comes to object oriented programming like “Java” or “C#”, or functional programming like “Haskell” separate studies have to be performed, and maybe the combined results of them and this work can be used to generalize it to all possible programming languages.
Chapter 8

Future Works

A list of possible future works can be directly deviated from the issues identified in the validity and limitations sections above. However there are some future work tasks that have not to deal with them and will be explained below.

The first and probably most important future work for the suitability regards the improvement of the “CMore” parser. It is already able to identify correctly most of the common situations in C-programs, but there is still room for improvements. The most important one in this context would be to add the value tracking ability in order for being able to identify buffer overflow faults.

Even though it was possible to extract the needed values using “CMore” there may exist other more sophisticated parsers like “ANTLR”\(^1\) that could be adapted to produce the desired output. A solution based on such a framework would probably be easier to adapt to new faults and other programming languages.

Despite that it has been shown to work with the chosen projects there is still a need to validate it with other open source projects and if possible even with industrial projects. Having such a deeper validation would strengthen the results obtained here.

Another possibility to show the performance of the presented method would be to compare the tools presented here against the state of the art tools present on the market. As the prototypes are not capable to detect all possible faults (as mentioned above) this comparison has to be limited to the faults that can be detected by all the involved tools, or they need to be postponed until the prototypes are extended.

Furthermore, as already seen in the validity section, this method works so far only for the procedural C programming language. The future work in this case would be to apply the method described within this work to programming languages of other kinds such as the object oriented or the functional ones. These results together with the ones presented here could lead to a more generalizable method that is not based on one programming concept.

In spite of having tested about 70 different machine learning algorithms, the present work deals only with their accuracy and false positive/negative rates but it totally ignores all the other properties important for an efficient and usable

\(^1\)http://www.antlr.org/
tool for fault detection. These properties include among others execution time, memory- and CPU-usage. The latter two are important to dimension a computer that is able to perform this task whereas the former is important to define the execution frequency. The less time consuming such a fault detection process is, the more often it can be executed which hopefully leads to a faster fault detection and removal time.

Objectively it could be observed that for example the “MultiLayerPerceptron” is much slower in its execution than the other 6 classifiers used in the validation experiment\(^2\) (This classifier was probably the slowest at all, however it was one of the best performing learners too), however during the experiments the measures that could prove this impression have not been taken. Before this method could be implemented into a valuable tool, these measures need to be taken at least for the best preforming classifiers, in order to find the best trade off between calculation power needed (nowadays probably not that important anymore), execution speed and detection performance (accuracy and false positive/negative rates).

The overall goal of such a work is of course to create a foundation for future fault detection tools. In order for this method to become implemented in a tool that does not need much interaction from a user there is a need of having a design study to know how to create a graphical user interface (GUI) that simplifies the selection of blocks to analyze and the visualization of the found faults. Furthermore it would need to offer the possibility of specifying values for some constants or possible value ranges for variables in order to improve the quality of the outcome. However the outcome of this work never focused on the development of any real tool, so the possibilities in this direction are pretty open.

\(^2\)see section 6.3
During this work a lot of interesting results came up and have been tried to describe in detail throughout the last chapters. Now at the end it is time to go a few steps back to have a deeper look on the research questions stated in the introduction and check if it was possible to answer them.

One of the research questions dealt with the problem of how source code could be transformed into a file format appropriate for a machine learner. The solution to this question presented in this work is the “CMore” parser and its underlying concept. This parser tries to follow the execution flow of a program and captures all the important activities to output them later on when needed in form of an input file for the WEKA tool.

The answer to this research question leads to another one dealing with the problem of what could be a possible feature vector to identify a code block as faulty or not. In order to find a minimal feature set, the experiment with WEKA has been performed two times. Once with the complete feature set obtained by the “CMore” parser and once with a reduced feature set.

From the results of this study it was possible to obtain a list of the best performing machine learning algorithms in terms of accuracy, false positive and false negative rate. It was possible to train 7 classifiers to show up an accuracy of over 95%, 5 to have a false positive rate of below 4% and 6 to have a false negative rate below 4% for the full feature set and 8 with accuracy over 95%, 9 with false positive below 4% and 4 respective for false negative on the reduced feature set.

The list (and graphs) obtained from it actually answer the last research question whose asking for these values. To validate the results a second experiment has been performed running the best 6 machine learning algorithms on a real world program. This one showed that 5 out of 6 could hold their expectations, while the one using “ADABoost” on a “BFTree” reduced its accuracy from over 96.77% in the first study down to below 77.15% for the full feature set and from 95.36% down to 73.43% for the reduced feature set (Similarly for false positive/negative rates).

Summarizing, the results of this work contribute in two points to the works published so far in this area. On one hand it verifies the suitability of machine
learning algorithms for the field of static source code analysis. This has been proved by pointing out some machine learning algorithms with very high precision and at the same time with a low false positive/negative rate.

In order to obtain this result about 70 different learners have been tested, which offers a comprehensive overview on the performance of the various learning algorithms. None of the papers found in the literature review and summarized in chapter 2 performed such a profound suitability test.

Furthermore this works introduces a new and apparently efficient way to transform source code into a standardized input format for a machine learning framework. Despite that it has not been implemented in a fully functional tool, the results obtained by the present work show the potential of the sketched solution, and its possibilities for future enhancements.
### Experiment Results

Table A.1 contains the complete data used in the analysis and shown in the results. This data is responsible for the graphs shown in the results section 6 showing averages and false positive rates.

<table>
<thead>
<tr>
<th>Method</th>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>corr</td>
<td>incorr</td>
</tr>
<tr>
<td>ADABoostM1 (BFTree)</td>
<td>Meta</td>
<td>96,774</td>
</tr>
<tr>
<td>ADABoostM1 (Decis.St.)</td>
<td>Meta</td>
<td>73,899</td>
</tr>
<tr>
<td>ADTree</td>
<td>Tree</td>
<td>79,052</td>
</tr>
<tr>
<td>AODE</td>
<td>Bayes</td>
<td>89,315</td>
</tr>
<tr>
<td>AODEEst</td>
<td>Bayes</td>
<td>91,734</td>
</tr>
<tr>
<td>AttributeSelectedClassifier</td>
<td>Meta</td>
<td>73,186</td>
</tr>
<tr>
<td>Bagging</td>
<td>Meta</td>
<td>87,298</td>
</tr>
<tr>
<td>BayesNet (BMAEst)</td>
<td>Bayes</td>
<td>76,210</td>
</tr>
<tr>
<td>BayesNet (SimpleEst)</td>
<td>Bayes</td>
<td>73,057</td>
</tr>
<tr>
<td>BFTree (PostPruned)</td>
<td>Tree</td>
<td>92,339</td>
</tr>
<tr>
<td>BFTree (UnPruned)</td>
<td>Tree</td>
<td>94,153</td>
</tr>
<tr>
<td>ClassViaClustering</td>
<td>Meta</td>
<td>49,798</td>
</tr>
<tr>
<td>ClassViaRegression</td>
<td>Meta</td>
<td>94,758</td>
</tr>
<tr>
<td>ClassBalancedND</td>
<td>Meta</td>
<td>75,403</td>
</tr>
<tr>
<td>ConjunctiveRule</td>
<td>Rule</td>
<td>54,637</td>
</tr>
<tr>
<td>DecisionTree</td>
<td>Meta</td>
<td>79,839</td>
</tr>
<tr>
<td>DataNearBalancedND</td>
<td>Meta</td>
<td>75,403</td>
</tr>
<tr>
<td>Decis.Stump</td>
<td>Tree</td>
<td>63,307</td>
</tr>
<tr>
<td>Decis.Tree (BestFirst)</td>
<td>Rule</td>
<td>87,500</td>
</tr>
<tr>
<td>Decis.Tree (Gen.Search)</td>
<td>Rule</td>
<td>87,097</td>
</tr>
<tr>
<td>Decorate</td>
<td>Meta</td>
<td>80,643</td>
</tr>
<tr>
<td>DTNB</td>
<td>Rule</td>
<td>89,113</td>
</tr>
<tr>
<td>EML</td>
<td>Meta</td>
<td>75,403</td>
</tr>
<tr>
<td>EnsembleSelection</td>
<td>Meta</td>
<td>82,460</td>
</tr>
<tr>
<td>FilterClassifier</td>
<td>Meta</td>
<td>75,403</td>
</tr>
<tr>
<td>FT</td>
<td>Tree</td>
<td>91,936</td>
</tr>
<tr>
<td>HyperPipes</td>
<td>Misc</td>
<td>50,202</td>
</tr>
<tr>
<td>Ibl</td>
<td>Lazy</td>
<td>95,161</td>
</tr>
<tr>
<td>J4L (LinearNN)</td>
<td>Lazy</td>
<td>96,573</td>
</tr>
<tr>
<td>J4S</td>
<td>Tree</td>
<td>75,403</td>
</tr>
<tr>
<td>J4SGraft</td>
<td>Tree</td>
<td>75,403</td>
</tr>
<tr>
<td>JRib</td>
<td>Rule</td>
<td>91,936</td>
</tr>
</tbody>
</table>

Continued on next page
### Appendix A. Experiment Results

#### Table A.1: Results from WEKA for all tested machine learners

<table>
<thead>
<tr>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>corr</strong></td>
<td><strong>incorr</strong></td>
</tr>
<tr>
<td>KStar</td>
<td>Lazy</td>
</tr>
<tr>
<td>LADTree</td>
<td>Tree</td>
</tr>
<tr>
<td>RBR</td>
<td>Lazy</td>
</tr>
<tr>
<td>LMT</td>
<td>Tree</td>
</tr>
<tr>
<td>Logistic</td>
<td>Funct.</td>
</tr>
<tr>
<td>LogitBoost</td>
<td>Meta</td>
</tr>
<tr>
<td>LWL (Decis.Stump)</td>
<td>Lazy</td>
</tr>
<tr>
<td>LWL (J48)</td>
<td>Lazy</td>
</tr>
<tr>
<td>MultiBoostAB</td>
<td>Meta</td>
</tr>
<tr>
<td>MultiClassClassifier</td>
<td>Meta</td>
</tr>
<tr>
<td>MultiLayerPerceptron</td>
<td>Funct.</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>Bayes</td>
</tr>
<tr>
<td>NBTree</td>
<td>Tree</td>
</tr>
<tr>
<td>ND</td>
<td>Meta</td>
</tr>
<tr>
<td>NNge</td>
<td>Rule</td>
</tr>
<tr>
<td>OneR</td>
<td>Rule</td>
</tr>
<tr>
<td>OrdinalClassifiers</td>
<td>Meta</td>
</tr>
<tr>
<td>PART</td>
<td>Rule</td>
</tr>
<tr>
<td>Rand.Comitee(Rand.Tree)</td>
<td>Meta</td>
</tr>
<tr>
<td>Rand.Forest</td>
<td>Tree</td>
</tr>
<tr>
<td>Rand.SubSpace(RepTree)</td>
<td>Meta</td>
</tr>
<tr>
<td>Rand.Tree</td>
<td>Tree</td>
</tr>
<tr>
<td>RBFNetwork</td>
<td>Funct.</td>
</tr>
<tr>
<td>RepTree</td>
<td>Tree</td>
</tr>
<tr>
<td>Rider</td>
<td>Rule</td>
</tr>
<tr>
<td>RotationForest</td>
<td>Meta</td>
</tr>
<tr>
<td>SimpleCart</td>
<td>Tree</td>
</tr>
<tr>
<td>SimpleLogistic</td>
<td>Funct.</td>
</tr>
<tr>
<td>SMO (Norm.PolyKernel)</td>
<td>Funct.</td>
</tr>
<tr>
<td>SMO (PolyKernel)</td>
<td>Funct.</td>
</tr>
<tr>
<td>SMO (Puk)</td>
<td>Funct.</td>
</tr>
<tr>
<td>SMO (RBFKernel)</td>
<td>Funct.</td>
</tr>
<tr>
<td>ThresholdSel. (Logistic)</td>
<td>Meta</td>
</tr>
<tr>
<td>VFI</td>
<td>Misc</td>
</tr>
<tr>
<td>VotedPerceptron</td>
<td>Funct.</td>
</tr>
<tr>
<td>Winnow</td>
<td>Funct.</td>
</tr>
<tr>
<td>ZeroR</td>
<td>Rule</td>
</tr>
</tbody>
</table>

**Average**

82,048 17,952 0.137 0.236 82,419 17,581 0.134 0.231
Appendix B

Comparison Results

Despite that this work tries to fill a gap which has not been addressed in that way before it is restricted to one single type of language. In order to be able to draw general conclusions on the suitability of machine learning algorithms for the field of static source code analysis a second complementary work has been performed which will be shortly introduced within this appendix chapter.

B.1 Complementary Results

The complementary work to the present one has been performed by Moriggl [23] and deals with object oriented programming languages on the example of Java instead of procedural ones. The projects under investigation are of course written in Java and the parser is completely different to be able to parse the peculiarities of the object oriented language. However, in order to be comparable the methodology, the machine learning framework and the idea behind the two works is basically the same.

<table>
<thead>
<tr>
<th></th>
<th>Full Features</th>
<th>Reduced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACC</td>
<td>FP</td>
</tr>
<tr>
<td>AdaB-BFTree</td>
<td>93.125</td>
<td>0.071</td>
</tr>
<tr>
<td>BF Tree-PostPr.</td>
<td>91.563</td>
<td>0.090</td>
</tr>
<tr>
<td>BF Tree-UnPr.</td>
<td>91.406</td>
<td>0.096</td>
</tr>
<tr>
<td>Classif.ViaRegr.</td>
<td>93.438</td>
<td>0.065</td>
</tr>
<tr>
<td>FT</td>
<td>89.844</td>
<td>0.093</td>
</tr>
<tr>
<td>JRip</td>
<td>Rules</td>
<td>96.250</td>
</tr>
<tr>
<td>LMT</td>
<td>Trees</td>
<td>96.563</td>
</tr>
<tr>
<td>Multil.Perc.</td>
<td>Functions</td>
<td>97.031</td>
</tr>
<tr>
<td>NNge</td>
<td>Rules</td>
<td>90.625</td>
</tr>
<tr>
<td>Rot.Forest-J48</td>
<td>Meta</td>
<td>96.250</td>
</tr>
<tr>
<td>SimpleCart</td>
<td>Trees</td>
<td>91.250</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>93.395</td>
</tr>
</tbody>
</table>

*Table B.1: Results of Most Suitable Machine Learners*

Out of the first study the best performing classifiers have been extracted and listed in the table B.1 with their corresponding accuracies and false positive/negative rates. The results presented here correspond to the ones found in the suitability study section of the results (see section 6.2).
Seamlessly to the procedure in this work the best performing classifiers have been tested on a real open source programs output. The results of this study is shown in table B.2 and correspond to the results found in section 6.3 “Project Evaluation Results”

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 “ADABost” 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

<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>

Table B.2: Performance of the Classifiers on the Test Set
in the case of C the nearest neighbour algorithms like “Ibk” or “NNge” perform quite good in both experiments 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 seams 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 first 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.
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 important 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 analysis.
3) Documenting: includes presenting and validating the research.

A detailed explanation of a systematic review and a comprehensive 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 reviewed in this study. The used infrastructure 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.
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 Kutubay 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 Khosgoftaar 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 telecommunication system, which they don’t call by name (unfortunately 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-Slig, C4.5, Treedisc, CBR, and logistic regression. The subject of this study was once again the telecommunication 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 recognized 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 Khosgoftaar [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 in these datasets was similar to the results above.

As already stated above, most of the papers presented in this section rely somehow on the expression force of some metrics. But what are the metrics that need to be used in order to obtain a good accuracy? Menzies et.al. in 2003 [24] compared some machine learners to understand which of them could be used in early development stages. As a result they got that using the proper metrics and a simple machine learner they could obtain the best results. Guo et.al. [25] with its random forest was able to obtain an accuracy of more than 75% (generally over 85%) based only on code metrics. As datasets they used 5 NASA projects written in different languages (CM1 (c), JM1 (c), KC1 (c++), KC2 (c++), PC1 (c)). Similarly in [26] where the authors used decision trees (and self-organizing maps for visualization) or in [27] where the the authors used the regression via classification approach. Briand et al. [28] were able to get accuracy of about 90% already in 1993 examining programs written in ADA. Lounies et.al. in [29] compared 7 machine learning techniques, containing decision trees and Naive-Bayes, to verify that code metrics can predict software quality. Beaver et.al. in [30] successfully applied Bayesian belief networks based on code metrics to predict quality. Similarly the study in [31], except that a dynamic analysis was performed.

However it is not all about code metrics, Shin in [32] showed that the same results obtained by code metrics can be obtained using design metrics (or a hybrid approach using code and design metrics). Challagulla et al. [33] went even a step further and said that size and complexity metrics are not enough to predict accuracy. This is supported by the approach presented by Moser et.al. in [34]. They showed that change metrics can be as good predictors for software quality as code metrics. They analyzed code repositories and commit messages to predict fault prone units. Similarly very recently the authors of [35] used code and change metrics for their
Despite that change metrics seem to perform well, their drawbacks become uncovered in [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 major 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 definitively 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


References


## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADABoostM1</strong></td>
<td>A boosting algorithm for nominal class values. Could increase dramatically the performance of a classifier, but sometimes overfits</td>
</tr>
<tr>
<td><strong>Allocation</strong></td>
<td>Reserving memory space</td>
</tr>
<tr>
<td><strong>ARFF</strong></td>
<td>Attribute Relation File Format</td>
</tr>
<tr>
<td><strong>AST</strong></td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td><strong>Back Propagation</strong></td>
<td>or error propagation is method to teach machine learners by comparing the real output with the desired one and adjust the learner according to it</td>
</tr>
<tr>
<td><strong>Boosting</strong></td>
<td>names a meta algorithm in machine learning based on the theory that a bunch of weak learning algorithms could in combination become a strong one</td>
</tr>
<tr>
<td><strong>Bug</strong></td>
<td>A common term used to name errors, failures and faults in software systems</td>
</tr>
<tr>
<td><strong>CSV</strong></td>
<td>Comma-Separated Values</td>
</tr>
<tr>
<td><strong>DAG</strong></td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td><strong>Detection</strong></td>
<td>In this case the direct identification of a fault in the source code</td>
</tr>
<tr>
<td><strong>Double-Free</strong></td>
<td>“free” command executed on an already freed pointer or null-pointer</td>
</tr>
<tr>
<td><strong>DT</strong></td>
<td>Decision Tree</td>
</tr>
<tr>
<td><strong>False Positive</strong></td>
<td>Something is evaluated to a positive answer, but in fact it is not (In this case: the process signals a fault where there is no)</td>
</tr>
<tr>
<td><strong>Failure</strong></td>
<td>The manifestation of a fault as a malfunction or system crash</td>
</tr>
<tr>
<td><strong>Fault</strong></td>
<td>A defect that causes reproducible malfunction of a software system</td>
</tr>
<tr>
<td><strong>GUI</strong></td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td><strong>Ibk</strong></td>
<td>K nearest neighbour algorithm, able to choose the best option among the k neighbours using cross validation</td>
</tr>
<tr>
<td><strong>Instance</strong></td>
<td>A case of occurrence of something (In this case: One line in an ARFF file)</td>
</tr>
<tr>
<td><strong>LMT</strong></td>
<td>“Logistic Model Trees” are classification trees with logistic</td>
</tr>
</tbody>
</table>
regression functions at the leaves.

**MultiLayerPerc.**: A classifier that uses neural networks and back propagation to produce the output

**NNge:** Nearest-neighbour-like algorithm which bases its decision on non-nested generalized examples (Rules)

**Null-Pointer:** A pointer without memory address assigned to it

**Overfitting:** is a situation where the learning has been performed to long or with to specific examples, so that the learner based its decisions on features that are not relevant to the target

**Prediction:** in this case names a guess where a fault in a piece of source code code be based on some kind of meta-values (metrics)

**RIPPER:** Repeated Incremental Pruning to Produce Error Reduction; an inductive rule based learner designed to minimize the error

**SVM:** Support Vector Machine

**SW Metric:** A property or attribute of, in this case, a piece of software

**True Positive:** Something is evaluated to a positive answer, and it is correct (In this case: the process signals an error where it really is)

**WEKA:** The Waikato Environment for Knowledge Analysis; a framework that provides a set of machine learners