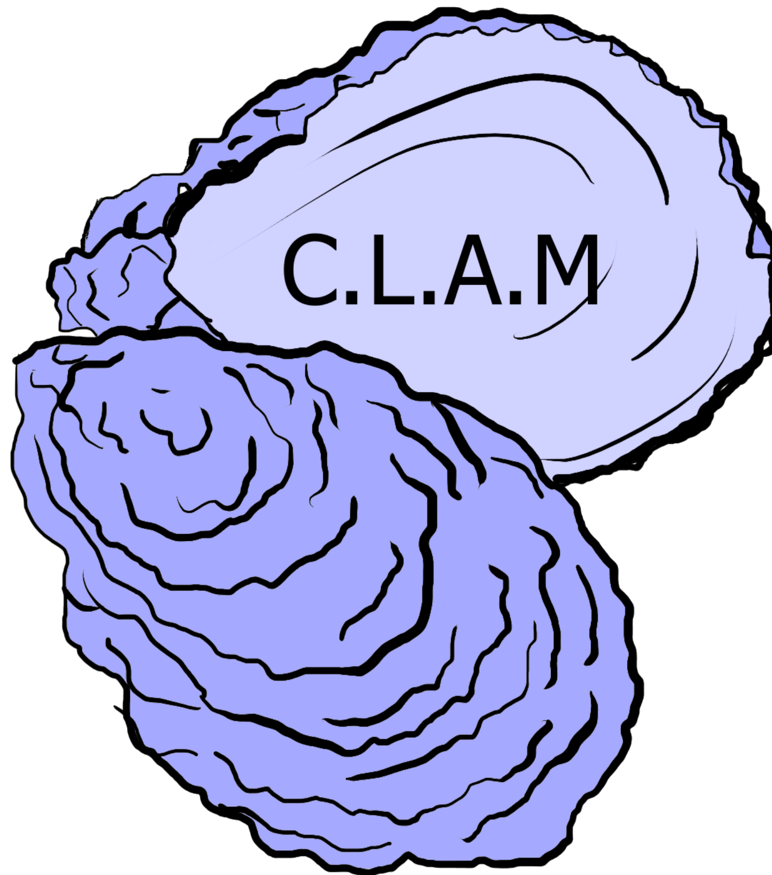

APPLYING MACHINE LEARNING ALGORITHMS TO DETECT LINES OF CODE CONTRIBUTING TO TECHNICAL DEBT

The C.L.A.M. Project

Bachelor of Science Thesis in Software Engineering and Management

FILIP ISAKOVSKI
RAFAEL ANTONINO SAULEO

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

The Author grants to University of Gothenburg and Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let University of Gothenburg and Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Using Machine Learning to detect Lines of Code contributing to Technical Debt

The C.L.A.M project [8].

FILIP ISAKOVSKI,
RAFAEL A. SAULEO,

© FILIP ISAKOVSKI, June 2018.
© RAFAEL A. SAULEO, June 2018.

Supervisor: Miroslaw Ochodek
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover:
A logo for the C.L.A.M. project which this paper covers.]

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Abstract. This paper shows the investigation of the viability of finding lines of code (LOC) contributing to technical debt (TD) using machine learning (ML), by trying to imitate the static code analysis tool SonarQube. This is approached by letting industry professionals choose the SonarQube rules, followed by training different classifiers with the help of CCFlex (a tool for training classifiers with lines of code), while using SonarQube as an oracle (a source of training sample data) which selects the faulty lines of code. The codebase consisted of a couple of proprietary software solutions, provided by Diadrom (a Swedish software consultancy company), along with open source software, such as ColourSharp [9]. The different classifiers were then analyzed for accuracy – compared against the oracle (SonarQube). The results of this paper demonstrate that using machine learning algorithms to detect LOC contributing to technical debt is a promising path that should be researched further. Within our chosen training parameters, the results show that increasing the percentage of LOC marked by the oracle brought increasingly better recall [7] values. The values increased more consistently than they did by just increasing the amount of LOC used for training. Furthermore, even though the precision is generally low within our parameters (meaning that the number of false positives is high), our classifiers still predicted many of the actually faulty LOC. These results are very promising when all of the training parameters are kept in mind. They show a lot of promise and open the gates to further exploration of this topic in the future.

Keywords: Technical Debt, Machine Learning, Static Code Analysis.

1 Introduction

1.1 Problem Statement

Software Engineering (SE) is a practice that systematically applies engineering to the development of software. As the software engineering profession matures, the software reliability, maintainability, and security requirements increase, with software becoming more pervasive in everyday life. Software quality (SQ) is one aspect investigated by SE, which deals with the software's capability of satisfying the stated (and implied) needs under specific conditions – however, this depends on the stakeholders' needs, wants, and expectations [23]. There is an interesting, still evolving concept of SQ, called technical debt, which deals with the consequences of the decisions that companies/developers have to make during the software development process.

Technical debt (TD) can be better explained as a software development concept that focuses on the costs that might arise because of certain decisions during software development. The definition of TD has been expanded over time, as initially, TD was defined just as bad code which is postponed to be fixed [12]. One aspect of its importance is presented in research that shows TD can cost companies up to $3.6 (USD) per line of code (LOC) [6]. Even the simplest software solutions can end up having thousands of LOC, which can accumulate to cause a lot of costs, failure to deliver etc.,

if the risk of TD is not mitigated during the development. It must be noted that TD can prove a worthy risk in the short term, which allows the software developers to deliver a software solution faster, but can have negative consequences in the long term [13]. Having in mind that TD can be potentially very expensive, many solutions have been developed with the purpose of detecting it in time. One way to detect TD is through the use of static code analysis tools. There are already a number of existing tools available (i.e. SonarQube [5], Gendarme [24], FindBugs [25]), that aim to deal with detecting LOC that contribute to TD. However, traditional static software quality analysis tools also have their limitations.

The first limitation that some of these tools have are the high false-positive detection rates. According to Jernej Novak et. al (2010) [26], the false-positive rates of software detection tools can even reach up to 50% in practice. On top of this, another issue exists – it's not very easy to modify the existing tools' fault recognition rules [27]. Different tools rely on different ways of implementing the rules, some through plugins and others through APIs. And even the simplest manner of extending the rules still requires the knowledge of how the tools work, and the ability to write a complex algorithm that will implement a fault detection rule. Adding to this, not all tools cover all coding standards, of which many exist – i.e. MISRA C [21], CERT C [28], Quantum Leaps [29] etc. Furthermore, these tools don't usually cover different internal company standards. Simply said, there is no single and definite standard of rules, and not all standards are covered by existing tools.

Having all of the above-mentioned limitations of static code analysis tools in consideration, we decided to investigate a different approach to static code analysis in order to overcome those limitations – through a piece of software that can learn how to find faulty LOC based on examples, instead of rules. So, instead of having to come up with rules on how to address a certain issue, developers could just "feed" the software with examples in order to achieve the same result. In order to achieve this, we have tried to investigate whether it is possible to solve the problem of detecting LOC contributing to TD with the use of Machine Learning (ML). So, instead of understanding or creating a complex algorithm, a developer would show examples of problematic code to the machine, which will try to find patterns in order to identify problematic code in the future.

Machine Learning (ML) is a powerful tool that can learn how to find patterns which map different inputs to different outputs. This concept introduces an opposite idea of creating and modifying rules - to build them (and even whole programs) based on examples (training data). Thus, it might help overcome the challenges that the current tools bring. However, it also introduces some drawbacks, since there are many algorithms that can be used for machine learning, all of which have their own positive and negative sides in certain situations [31].

In order to construct an ML-based tool for identifying LOC contributing to TD, we need a bigger program of studies investigating different issues – e.g., whether people would accept the different way of specifying the "rules" by providing examples. But in order to see if that issue is even worth further investigation, we need to start from this study, which tries to understand the possibility of constructing such a tool.

In this study we try to replicate the code analysis tool SonarQube - which if successful, could be further repeated for other tools. The study aims to answer whether it is feasible to create a code analysis tool by using ML, and whether the matter is worth of further exploration. More precisely, our investigation's objective is to discover if it's possible to use ML in technical debt and code quality analysis. Can ML be used to mimic and possibly improve code quality analysis? Can it become as good as SonarQube only by analyzing its output data, or will feedback from professional programmers be required for reaching this goal and possibly overcoming it?

To achieve this objective, we designed and performed an exploratory case study in collaboration with a Swedish software consultancy company – Diadrom, which provided us with the needed resources for our investigation. We decided to name this project C.L.A.M. (Code Learning Analysis Machine).

1.2 Outline

This paper is structured in the following way: In Section 2, we have described previous work related to the research, and the background for the research. In Section 3 the research study design and execution are described into detail, starting with the research methodology, data collection and analysis details, and all the way up to the conclusion. This is followed by the References, which are followed by the Appendix, which holds a lot of evaluation quality results.

2 Related Work and Background

Similar problems have been analyzed in both "Predicting Source Code Quality with Static Analysis and Machine Learning" [2] and "Using machine learning to design a flexible LOC counter" [3], with the former being a bit more related to the issue of code quality analysis with machine learning, but the latter being more related towards designing a flexible software engineering tool with machine learning. A similar way to analyze code using ML is presented in the work "A Case Study on Effectively Identifying Technical Debt" [15]. We have also identified two studies that helped us get a better grasp on what TD is and how to determine it effectively – "The Role of Technical Debt in Software Development" [13] and "A Case Study on Effectively Identifying Technical Debt" [14].

2.1 Predicting Source Code Quality with Static Analysis and Machine Learning

In "Predicting Source Code Quality with Static Analysis and Machine Learning" [2] the authors approach the issue by using public datasets, peer review and classification, in order to create the training data used for machine learning. The purpose was to find out whether it can be decided if a piece of code is "good" or "bad" by using machine learning. We're planning to build on this idea, but in a different direction - the industry. We want to find out whether we can replicate an already well-established tool in the

industry – SonarQube, to what extent that can be accomplished and to understand whether and to what extent it can be accepted by industry developers.

2.2 Using Machine Learning to Design a Flexible LOC Counter

The study “Using machine learning to design a flexible LOC counter” [3], shows a combined use of a software pipeline and machine learning in order to create a flexible line of code counter. The training data for the machine learning in this research was provided by industry professionals, who conducted the needed classification. This study resulted in the creation of CCFlex, which is one of the two main tools (the other one being SonarQube) that we’re going to use in our research. It will be used as the basis for machine learning in our research, as it can be trained with marked lines of code.

2.3 The Role of Technical Debt in Software Development

This study [13] focuses on researching technical debt – what it is, whether it plays a relevant role in software engineering, the causes of technical debt and the short and long-term effects of technical debt in software development.

The results of the study show that there are always several reasons for technical debt – it does not stem from one. It also shows that there are mainly positive effects to TD in the short-term (due to the benefit of a quicker software release), which produce negative effects in the long-term (due to the accumulation of bad decisions and practices). The artefact produced during this study was a technical debt management framework, used for describing different categories in TD management.

2.4 A Case Study on Effectively Identifying Technical Debt

A study which focuses on the identification of TD [14], by comparing items related to TD selected by humans, and ones selected by 3 different analysis tools. Then they compared the results and observed that the analysis tools and the developers found relevant TD-contributing items, however the developers usually found “defect debt”. Another observation they made was that static analysis tools overlook TD that is not caused by source code, but by other development artefacts.

2.5 Automated Analysis of Source Code Patches using Machine Learning Algorithms

In this work [15] a tool is presented, which can be used for the automated analysis of source code and branch differences. The tool uses ML for the analysis. The source code analysis using ML part of this work is an idea that is very similar to the way that ML is going to be used in our study (the CLAM project).

3 Research Design and Execution

3.1 Research Questions

Research Question 1: Can machine learning algorithms be used to build a code analyzer that will be able to detect lines of code contributing to technical debt?

RQ 1.1: What accuracy can be achieved in detecting LOC contributing to TD, when taking SonarQube as an oracle?

RQ 1.2: How does the number of labeled examples affect the accuracy of the ML-based tool for detecting LOC contributing to TD, when taking SonarQube as an oracle?

Purpose: Explore if it is possible to train a ML-based classifier to mimic SonarQube in detecting lines of code contributing to technical debt.

3.2 Research Methodology, Data Collection and Analysis

Our methodology focuses on conducting an exploratory case study [20], consisting of simulation experiments, with the goal of exploring whether it is possible to train ML-based classifiers to mimic SonarQube in detecting LOC which contribute to TD. We decided to conduct an exploratory case study, because it allowed us to perform exploratory experiments with ML classifiers, by exploring different parameters - i.e. different sample size, or a different marked LOC density, etc. (as explained better in the following paragraphs). We could then gather a lot of data from the experiments, which we could afterwards analyze and try to understand, with the purpose of understanding the accuracy of the classifiers and how the different parameters affect them.

Obtaining the samples. The first step, necessary for carrying out the experiments, was obtaining the codebase. For the data collection, we managed to obtain our codebase by collaborating with the software consultancy company Diadrom, which provided us with 2 proprietary software suites. Additionally, we also obtained code from ColourSharp [9], which is an open-source project. All of these suites were mainly written in C#.

In order to effectively decide which SonarQube rules to use during the training of the classifiers, we had a small workshop with professional developers from Diadrom, during which they agreed that we should use all of the TD-related rules in our experiments. After this, the software suites were compiled using MSBuild [10] and analyzed by SonarQube. When the analysis was over, we needed to mark the faulty LOC detected by SonarQube. This was accomplished by exporting the SonarQube results to CSV (comma-separated values) files, which contained the LOC that SonarQube found as faulty. Following this, bash scripts were made, which were used for copying the files with faulty code and marking the faulty lines of code inside of them. Now we had a pool of code that we could use in the training data samples.

After all of these steps, the data was ready to be processed by CCFlex. Lines of code were extracted manually, but randomly, from the class files. We did this while still trying to maintain some code structure - for example, if a marked line was a part of a function, the whole function was used in the sample. This was done for the purpose of making the sample data more diverse, while preserving its integrity in terms of code

structure. Our assumption was that this would help the classifiers to find better patterns for detecting faulty LOC. We prepared 12 different sets of data (samples) in this manner, and divided each one into 10 equal parts, which would be used for 10-fold cross-validation (as explained below) [32]. The data was now ready to be processed by CCFlex.

Training and testing the classifiers. CCFlex needed to be configured a bit before running it on the samples. Among the several types of configurations that it offered, we decided to configure which algorithms it should use for training. We chose the CART [16], RandomForest [17] and KNN [18] classifiers. The first two algorithms are decision tree based [16] [17], which use internal decision nodes for classifying. The KNN, on the other hand works through clustering the features of the data into an n-dimensional space and then predicts trough comparing the placement of the training data and the *testing data* (data which is used to test the classifier's prediction accuracy) on that space. We chose these algorithms because we could understand how they operate and they were also implemented into CCFlex. Additionally, we wanted to explore how similarly functioning algorithms compare against each other and against different ones. CCFlex also allows some aspects of the algorithms to be configured as well. We chose to explore how tweaking the depth of the decision trees and the number of nearest neighbors would affect the classifiers - whether it would make them better or worse.

During the training of the classifiers, we decided to do 36 runs (12 samples × 3 classifier configurations), in which the samples contained a different number of LOC and percentage of marked LOC, while the classifiers' configuration varied in the tree depth, or number of nearest neighbors (5, 7 and 9) in different runs. We did runs with 100, 500, 1000 and 2000 LOC, in combination with 10%, 25%, and a random (unobserved) percentage of marked LOC, all of which were combined with a varying classifier tree depth/no. of nearest neighbors, by using the 10-fold cross-validation technique. The 10-fold cross-validation technique requires 10 iterations of training and testing of the classifiers, for each of the 36 runs, or 360 iterations in total. This was performed by, firstly, splitting the samples into 10 equal subsamples. After that, in each of the 10 iterations, 9 subsamples of the data are used for training, and 1 subsample is used for the testing of the classifiers, in a manner in which each subsample is eventually used for both training and testing. The results from the testing of the subsamples were then averaged.

We limited the amount of LOC in the samples for several reasons. We wanted to see how viable it would be for developers in the real world to train the classifiers with a sample of a limited size. Even though developers might possibly have access to big amounts of data, if they were to mark their own faulty LOC (in order to train the classifier) it would take a lot of time to do it manually. Adding on to this, we wanted to investigate how different densities of faulty LOC in the sample would alter the results. This is because, although we have an oracle (SonarQube) which marks all the faulty LOC for us, developers who would try to train a classifier with their own rules wouldn't have an oracle that would provide them with "unlimited" examples of faulty LOC. That is why samples with varying size and density of faulty LOC were chosen. We also wanted to explore whether the classifiers can be improved by tweaking some of their settings, with our chosen number of LOC. In order to achieve this, as mentioned above,

we combined 3 different tweaks of the tree depth (for the CART and Random Forest algorithms) and number of nearest neighbors (for the KNN algorithm), with values of 5, 7 and 9.

Data evaluation. When all of the iterations of running CCFlex were completed - both training and testing the classifiers, we had the data from the testing. 'Testing' in this case means running a trained classifier on a subsample that contains LOC, and then comparing the predicted results against the results from SonarQube [31]. From this comparison, we would get 4 types of results: **a)** *true positive* - faulty lines marked as faulty by both the classifier and SonarQube; **b)** *false positive* - faulty lines marked as faulty by the classifier, but not by SonarQube; **c)** *true negative* - lines not marked as faulty by both the classifier and SonarQube; **d)** *false negative* - lines not marked as faulty by the classifier but marked as faulty by SonarQube. With this data, we were able to create confusion matrices [30] for each classifier, which allowed us to visualize and analyze the results from the tests. Furthermore, they allowed us to calculate the accuracy, recall [7], precision [7] and F-score of the results, which show us the performance of the classifiers. The accuracy represents the amount of correctly marked lines, the recall represents the number of true positives, the precision shows the number of false positives, while the F-score represents the balance between the precision and the recall. With this, we were able to evaluate the performance of the classifiers.

Along with all of the previously mentioned data, we also decided to measure how the representation of some of the features used in the training of the classifiers might affect the results. The **total number of features** extracted from the samples varied between **800** and **1150**. Most of these features were extracted using the bag of words model. We used ***all of these features*** in the ***classifier training***. Out of all of them, however, we ***analyzed the influence*** of only 2 features, to see whether and how they affect the results. These two features were the number of words and number of characters in every LOC. We calculated the mean, median and standard deviations of these two features in order to try to see whether there is any correlation between these features and the classifier predictions. The reason that we chose to observe these 2 features was because they stood out in the pool of a thousand other features and could be easily measured and compared.

3.3 Validity

To ensure the validity in this research, we tried to analyze the aspects of validity mentioned in the checklist designed by Runeson and Höst [11] for case studies. We have identified the several threats to validity to our research and have devised a plan on how to avoid or mitigate them in some cases.

In order to establish the construct validity, we will be using a Confusion Matrix [30], which is a common way to visualize and calculate the performance of algorithms in ML experiments. The use of such measures as Accuracy, Precision and Recall is a regular way of measuring the quality in this kind of scenario. In this way we will keep the construct validity's integrity.

We also devised several ways to mitigate the internal validity threats. First, the training samples were chosen randomly from a pool of data, to avoid sample bias. On top of this, experiments were conducted with different sample sizes for every algorithm, in order to be able to evaluate the performance of the system when classifying the results. Additionally, in order to further sustain the validity, member checking was done by the most competent developers' evaluation of SonarQube's rules. These developers were selected by Diadrom's CEO.

There are some possible external validity threats. For example, the research was conducted on code written in the C# programming language. This might make our research results not applicable to other languages, especially if they are not in the C-family of languages. Furthermore, for this research we used SonarQube as a point of reference which provides the data needed for the research – or in other words, an oracle. Depending on how other tools similar to SonarQube work, the study's results may end up being limited to SonarQube and limited to the languages that SonarQube can analyze. In addition to this, the developers from Diadrom evaluated the rules which affected the training data at one point in the research. Due to the coding standards utilized by the developers in the company, the results might end up different than they would if they were conducted in a company with different standards.

Finally, in order to mitigate or alleviate any reliability threats, we did our best to show every step during the experiments that we've conducted in our research, in a way that the study could be repeated with similar results if the same parameters are used.

3.4 Results

We trained the classifiers in several runs, using a different amount of marked and unmarked lines of code, and varying lines of code in general. We did different runs with 100, 500, 1000 and 2000 LOC. Each of them was tested with a different amount of marked lines: random percentage of marked LOC and set percentage of LOC – 10% and 25%. We also modified the classifiers' depth. Below we present the results split along two parameters: Features and Evaluation Quality.

Features. We used around 1000 features in the training of the classifiers. However we investigated 2 of them into more detail. The feature data that we focused on was the number of characters and number of words in each line. The average, median and standard deviation were calculated for the number of characters and number of words features.

Table 1. Features of the first run – random % of marked LOC.

Lines of code	Average No. of characters.	Median No. of characters	Stand. Dev. No. of char.	Average No. of words	Median No. of words	Stand. Dev. No. of words
100	26.36	21.5	25.18	2.1	1	2.26
500	32.53	32	28.76	2.56	2	2.71
1000	35.23	29	32.01	2.73	2	3.29
2000	33.85	30	29.58	2.61	2	2.73

Table 2. Features of the second run – 10% of marked LOC.

Lines of code	Average No. of characters.	Median No. of characters	Stand. Dev. No. of char.	Average No. of words	Median No. of words	Stand. Dev. No. of words
100	37.45	35.5	31.74	2.4	2	2.09
500	30.43	25	26.02	2.25	1.5	2.04
1000	37.45	31	32.9	2.78	2	3.14
2000	34.65	30	29.8	2.47	2	2.26

Table 3. Features of the third run – 25% of marked LOC.

Lines of code	Average No. of characters.	Median No. of characters	Stand. Dev. No. of char.	Average No. of words	Median No. of words	Stand. Dev. No. of words
100	36.71	25.5	30.67	2.4	1.5	2.01
500	36.53	35	30.34	2.78	2	2.51
1000	34.76	30	29.49	2.59	2	2.34
2000	36.52	52	29.88	2.88	2	2.38

Evaluation Quality. The confusion matrix method was used to evaluate the accuracy of the classifiers. This method shows the number of true positive, true negative, false positive and false negative lines predicted by the classifiers. This way, the Accuracy, F1 Score, Precision and Recall can be measured. In order to test whether the accuracy can be improved with a different configuration, the max. depth of the CART and RF algorithms was modified, along with the number of nearest neighbors of the KNN algorithm. The number of marked (faulty) lines in the training samples, was also a sample configuration which we experimented with. The tables below show charts of how the quality of the algorithms varied with different configurations of the samples (size, percentage of marked lines), and different configurations of the classifiers – tree depth/no. nearest neighbors of 5,7,9.

Table 4. A Chart Containing the Recall Values

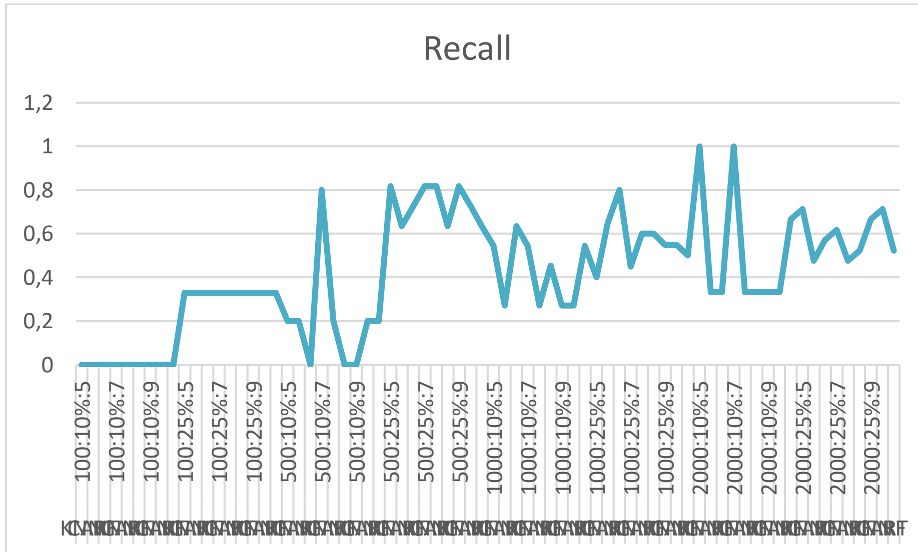

Table 4 represents the recall (no. of true positives) derived from the results of the classifier evaluation. A pattern can be noticed, according to which, the recall value is lower when the percentage of marked LOC in the samples is lower (10% vs 25%). Table 5, on the other hand, represents the precision values derived from the classifier evaluation. The precision, however, shows no pattern which demonstrates that its value is dependent on some parameter from the sample or a classifier configuration.

Table 5. A Chart Containing the Precision Values

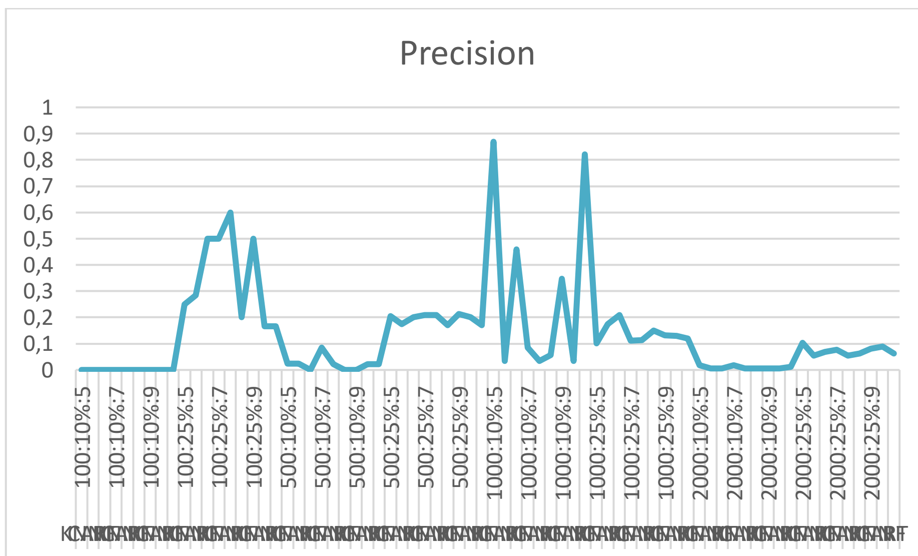

Table 6. A Chart Containing the Accuracy Values

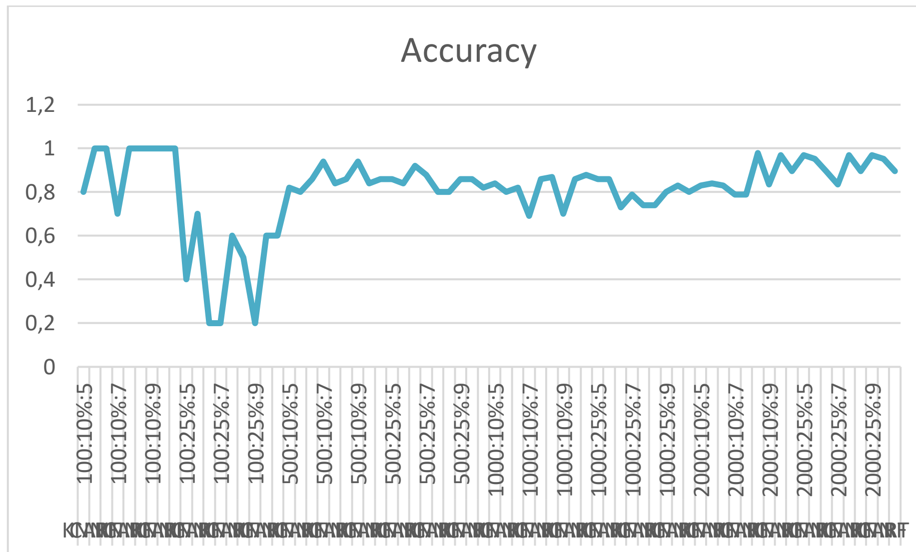

Table 6 represents the accuracy of the classifiers in different sample and classifier configurations. The accuracy seems to be the highest when the number of LOC and the percentage of marked LOC are at their highest configurations.

The confusion matrices for these results are presented in the Appendix of this document from Fig. 1 to Fig. 24, because of the size of the data. Those represent the average results that were observed during the classifier tests.

3.5 Discussion

We conducted this research, with the purpose of figuring out whether it is possible to use ML algorithms for building a code analyzer which is able to detect LOC contributing to technical debt. With the above results taken into consideration, many different aspects can be discussed, however we have decided to focus on several points that are correlated to our research questions. From the 800-1000 features that were used for training of the classifiers, we concentrated on measuring 2 of them for the investigation – the ***number of characters per LOC*** and the ***number of words per LOC***. This was for the purpose of finding out whether these features play a role in the accuracy of the trained algorithms. We calculated the *mean* and *median* of these features, per sample. This was done with the aim to try to answer RQ1.1. We initially thought that the length of the lines might play a role in the accuracy. For example, if the training files had too many short lines (i.e. parentheses only), we assumed that it might affect the confidence of the classifier when marking a line as faulty. From the investigation that we conducted based on the features that we chose to measure and compare (line length and no. of words per line) we can see that no matter what the number of LOC or marked percentage of LOC is, they are not correlated to the quality of the classifiers' predictions. This

is due to the fact that even though the measures of the accuracy of the classifiers' predictions vary quite a lot, we do not see any correlation between them and the representation of the features that we chose to examine.

Secondly, we looked at the number of marked lines per sample, with the purpose of answering RQ1.2. We can see that they definitely shake the numbers when training with 100 LOC, from the sample confusion charts in Fig. 1 – Fig. 6 of the appendix. From the chart in Table 5, we can also see a trend of the ***precision*** rising when training with 25% of marked lines, and it is lower when the classifiers are trained with 10% of marked faulty LOC. The precision doesn't seem affected by the size of the samples. The ***accuracy*** value seems unaffected by the number of marked LOC, but instead looks more affected by the number of total LOC per sample (the size of the sample).

The ***recall*** is a value which shows the ratio of correctly predicted positive observations to all of the other observations in a class (the percentage of correctly guessed faulty lines). It is generally higher when the number of marked LOC is higher - and is consistently so. So, when we have a higher percentage of marked LOC, we can see that the recall is consistently higher, of course compared to other results with the same total number of LOC. What is noteworthy to mention from the results, is that the percentage of marked LOC brought more consistently positive recall values than the total amount of LOC did. However, both of those affect the recall. From this we can say that the percentage of marked LOC in a training sample is very important, at least when the total sample is not bigger than 2000 LOC. The recall values are very important in our case, because they show the amount of the true positives that our classifiers predicted. Even though the precision is quite low (which gets low when the number of false positives is high), our classifiers still predicted many of the actually faulty LOC. These results are very promising, since they show that if developers wanted to teach a classifier how to find faulty code according to their own standards, they could plausibly get good results (but we **cannot guarantee** this, since more investigation is required in order to claim this) with only 250-500 LOC of examples of faulty code. In the future, we would like to try to see what kind of results could be gotten with up to 50% of marked LOC (equal number of marked and unmarked LOC), and how that would affect the recall and the precision.

The precision in our case is usually quite low. In our best-case scenarios, we have around 20-50% false positives. This means that even though we have a promisingly high percentage of true positives, the number of false positives is also quite high, which would be inconvenient for the developers. However, like we mentioned, according to Jernej Novak et. al (2010) [26], some static software analysis suites can show up to 50% of LOC marked as false positive, which means that our classifiers have room for improvement, but are not critically bad, especially since in some cases it might be more important to catch the *actually faulty* LOC (true positives), even if we have a lot of marked false positives. This might be so in cases where the faulty LOC is some critically bad piece of code that has to be found even at the cost of going through many falsely marked pieces of code. We see a problem, however, which is not consistently affected by the different parameters. The problem is that neither the size of the samples, the amount of marked LOC, nor the configuration of the classifiers show any pattern of affecting the precision in a positive or a negative manner. But we have to agree that

the precision has a lot of space for improvement, so in the future it would be a nice idea to see how the precision would be affected by bigger and more saturated samples.

The F-Score is a value which shows balance between the precision and the recall. Since its value is determined by both the precision and the recall values, we could not find any pattern which affects its value, because we don't see any pattern in the value of the precision. It would play a bigger role into the results if we had more consistent precision values, and would show the dynamics between different patterns, but unfortunately it doesn't mean a lot to us in this study, with the current results.

The Recall, Precision and F-Score values are all important values, very commonly more so than the Accuracy value, especially in cases when the sets of False Positive and False Negative values are very dissimilar [19]. However, it is important to have a high accuracy – in our case it should be optimally higher than the percentage of unmarked LOC. This is because we could easily get a high accuracy of 90%, by just marking all of the LOC as non-faulty, if we have a sample with 90% non-faulty LOC. The results from our experiments show a pretty good accuracy, especially when the percentage of faulty LOC and the total number of LOC are high. This shows promise that the results might be better if the investigation would continue with a larger and denser sample, or with different classifier configurations.

The last aspect that we explored was the configuration of the max depth of the CART and RF classifiers and the number of nearest neighbors of the KNN classifier. This was done in order to answer RQ1.1, and we did these different configurations with the goal of trying to increase the accuracy of the classifiers. While the depth doesn't show any considerable differences in the CART and RF classifiers, the results give quite an obvious difference when the number of nearest neighbors is changed. The rise of this number shows a drop in the amount of False Positive LOC marked by the KNN classifier, when the number of training LOC is 1000 or 2000. When this is the case, the accuracy of the LOC marked by KNN increases. However, we see that sometimes the recall is affected negatively when the number of nearest neighbors is increased, and the sample size is equal to, or larger than 500 LOC. This behavior is more noticeable at the sample size of 2000 LOC, even though sometimes the decrease is very small, so it might not be significant.

3.6 Conclusion

During the course of our experiments, we did not manage to completely mimic SonarQube in its functions. However, we did get positive and promising results. The high *recall* and high *accuracy* give hope that it might be possible to reliably detect LOC contributing to TD using ML in the future. Even though the *precision* remained quite low and unpredictable, our classifiers managed to predict 60-100% of the truly faulty lines correctly. This means that even though we had many false-positives, with a bigger size and saturation of the samples, and a different configuration of the classifiers, even better results might be achieved. We can use this information in order to answer RQ1.1 - the accuracy was quite high in the experiments, and the recall was also pretty good, with the precision being the only down point. Furthermore, even though we **used around 1000 features** for training, we have **investigated** and **measured** 2 of them,

whose values turned out to be unrelated to the accuracy of the predictions. The *mean* and *median* values of these 2 features do not affect the classifiers in our investigation, which means that ***different features should be measured*** if the investigation is continued in the future, in order to find out if some other features impact the accuracy of the classifiers.

In order to answer RQ1.2, we can say that the number (or percentage) of marked lines in the sample ***noticeably influences*** the classifiers' predictions, and also improves the classifiers' *accuracy* and *recall*. However, the only parameter that successfully affects the *precision* is the increased percentage of marked LOC per sample.

Something that should be done in a future investigation of the problem, is to try and measure whether some rules are easier to detect than others, and maybe to try training different classifiers to recognize only one rule. For now, we have identified different patterns in the *precision*, *recall* and *accuracy* values, which are affected by different configurations of the samples and classifiers. More configurations should be explored in the future.

So, can machine learning algorithms be used to build a code analyzer that will be able to detect lines of code contributing to technical debt? Our investigations show promising results, which means that we might be on a path to answer this question with a "Yes" in future explorations of the problem.

References

1. P. Bourque, R.E. Fairley: Guide to the Software Engineering Body of Knowledge, Version 3.0. IEEE Computer Society, 2014.
2. V. Barstad, M. Goodwin, T. Gjøsæter: Predicting Source Code Quality with Static Analysis and Machine Learning. Norsk Informatikkonferanse, 2014.
3. M. Ochodek, M. Staron, D. Bargowski, W. Meding, R. Hebig,: Using machine learning to design a flexible LOC counter. IEEE, 2017.
4. SonarQube Doc. Page, https://docs.sonarqube.org/display/SONAR/Documentation, last accessed 2018/03/26.
5. SonarQube About Page, https://www.sonarqube.org/about/, last accessed 2018/04/01.
6. CAST, The CRASH Report - 2011/12 (CAST Report on Application Software Health). CAST, 2011.
7. W. Koehrsen: Beyond Accuracy: Precision and Recall, https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c, last accessed 2018/04/03.
8. C.L.A.M - Code Learning Analysis Machine - The name that the authors gave to this project.
9. ColorSharp GitHub page, https://github.com/Litipk/ColorSharp, last accessed 2018/04/29
10. MSBuild Microsoft Website, https://msdn.microsoft.com/en-us/library/dd393574.aspx, last accessed 2018/04/29.
11. P. Runeson, M. Höst: Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering. Springer 2009.
12. P. Kruchten, R. L. Nord, I. Ozkaya: Technical Debt: From Metaphor: from Metaphor, to Theory, to Practice. IEEE, 2012.
13. J. Yli-Huumo: The Role of Technical Debt in Software Development. LUT, 2017
14. N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, C. Seaman: A Case Study on Effectively Identifying Technical Debt. ACM, 2013.

15. A. C. Lechtaler, J. C. Liporace, M. Cipriano, E. García, A. Maiorano, E. Malvacio, N. Tapia: Automated Analysis of Source Code Patches using Machine Learning Algorithms. SEDICI UNLP, 2015.
16. Scikit-Learn CART page, http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html, last accessed 2018/05/27.
17. Scikit-Learn RandomForest page, http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, last accessed 2018/05/27.
18. Scikit-Learn KNN page, http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, last accessed 2018/05/27.
19. Exilio Webpage, http://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/, last accessed 2018/05/27.
20. Zaidah Z.: Case study as a research method. In: Jurnal Kemanusiaan bil.9, 2007
21. Motor Industry Research Association: Guidelines for the use of the c language in vehicle based software. Motor Industry Research Association, Nuneaton, Warwickshire, 1998.
22. Bourque P. and Fairley R.E.: Guide to the Software Engineering Body of Knowledge, Version 3.0. In: IEEE Computer Society, 2014; www.swebok.org.
23. Ammann, P., & Offutt, J.: Introduction to software testing. In: Cambridge University Press, New York, 2008.
24. Mono Homepage: http://www.mono-project.com/docs/tools+libraries/tools/gendarme/, last accessed 2018/06/20.
25. FindBugs™ - Find Bugs in Java Programs Webpage, http://findbugs.sourceforge.net/, last accessed 2018/06/20.
26. Novak, J., Krajnc, A., & Žontar, R.: Taxonomy of static code analysis tools. In: Mipro 2010. [vol. 2], Telecommunications & Information = Telekomunikacije I Informacije, Vol. 2, Str. 169-173, (2010).
27. Svoboda, D., Flynn, L., Snavely, W., & 2016 IEEE Cybersecurity Development (SecDev). In: Static analysis alert audits: Lexicon & rules (pp. 37-44). In: IEEE. doi:10.1109/SecDev.2016.018, Boston, MA, USA 2016, Nov, 2016.
28. CERT C Coding Standard, 2016 Edition. Cert, 2016.
29. Quantum Leaps, LLC.: Application Note C/C++ Coding Standard, Document Revision J. Quantum Leaps, LLC, April, 2013.
30. Susmaga R.: Confusion Matrix Visualization. In: Kłopotek M.A., Wierzchoń S.T., Trojanowski K. (eds) Intelligent Information Processing and Web Mining. Advances in Soft Computing, vol 25. Springer, Berlin, Heidelberg, 2004.
31. Ademujimi T., Brundage M. P., Prabhu V. V.: A review of current machine learning techniques used in manufacturing diagnosis. In: Lödding H., Riedel R., Thoben KD., von Cieminski G., Kiritsis D. (eds) Advances in Production Management Systems, 2017.
32. Refaeilzadeh P., Tang L., Liu H.: Cross-Validation. Arizona State University, 2008.

Appendix

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	0	0	0	0	0	0
Actual Negative	2	8	0	10	0	10
	Acc. 0.8	Prec. 0	Acc. 1	Prec. 0	Acc. 1	Prec. 0
	Recall 0	F1 0	Recall 0	F1 0	Recall 0	F1 0

Fig. 1 – Confusion Matrix of classifiers trained with 100 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	0	0	0	0	0	0
Actual Negative	3	7	0	10	0	10
	Acc. 0.7	Prec. 0	Acc. 1	Prec. 0	Acc. 1	Prec. 0
	Recall 0	F1 0	Recall 0	F1 0	Recall 0	F1 0

Fig. 2 – Confusion Matrix of classifiers trained with 100 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	0	0	0	0	0	0
Actual Negative	0	10	0	10	0	10
	Acc. 1	Prec. 0	Acc. 1	Prec. 0	Acc. 1	Prec. 0
	Recall 0	F1 0	Recall 0	F1 0	Recall 0	F1 0

Fig. 3 – Confusion Matrix of classifiers trained with 100 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	1	2	2	1	1	2
Actual Negative	4	3	2	5	6	1
	Acc. 0.4	Prec. 0.25	Acc.0.7	Prec. 0.285	Acc. 0.2	Prec. 0.5
	Recall 0.333	F1 0.285	Recall 0.333	F1 0.4	Recall 0.333	F1 0.4

Fig. 4 – Confusion Matrix of classifiers trained with 100 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	1	2	1	2	1	2
Actual Negative	6	1	2	5	3	4
	Acc. 0.2	Prec. 0.5	Acc. 0.6	Prec. 1.66	Acc. 0.5	Prec. 0.2
	Recall 0.333	F1 0.4	Recall 0.333	F1 0.222	Recall 0.333	F1 0.25

Fig. 5 – Confusion Matrix of classifiers trained with 100 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	1	2	1	2	1	2
Actual Negative	6	1	2	5	2	5
	Acc. 0.2	Prec. 0.5	Acc. 0.6	Prec. 0.166	Acc. 0.6	Prec. 0.166
	Recall 0.333	F1 0.4	Recall 0.333	F1 0.222	Recall 0.333	F1 0.222

Fig. 6 – Confusion Matrix of classifiers trained with 100 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	1	4	1	4	0	5
Actual Negative	5	40	6	39	2	43
	Acc. 0.82	Prec. 0.024	Acc. 8	Prec. 0.025	Acc. 0.86	Prec. 0
	Recall 0.2	F1 0.043	Recall 0.2	F1 0.044	Recall 0	F1 0

Fig. 7 – Confusion Matrix of classifiers trained with 500 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	4	1	1	4	0	5
Actual Negative	2	43	4	41	2	43
	Acc. 0.94	Prec. 0.085	Acc. 0.84	Prec. 0.023	Acc. 0.86	Prec. 0
	Recall 0.8	F1 0.153	Recall 0.2	F1 0.042	Recall 0	F1 0

Fig. 8 – Confusion Matrix of classifiers trained with 500 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	0	5	1	4	1	4
Actual Negative	2	43	4	41	0	45
	Acc. 0.86	Prec. 0	Acc. 0.84	Prec. 0.023	Acc. 0.92	Prec. 0.021
	Recall 0	F1 0	Recall 0.2	F1 0.042	Recall 0.2	F1 0.039

Fig. 9 – Confusion Matrix of classifiers trained with 500 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

4 - 25%

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	9	2	7	4	8	3
Actual Negative	4	35	6	33	7	32
	Acc. 0.88	Prec. 0.204	Acc.0.8	Prec. 0.175	Acc. 0.8	Prec. 0.2
	Recall 0.8181	F1 0.327	Recall 0.636	F1 0.274	Recall 0.727	F1 0.313

Fig. 10 – Confusion Matrix of classifiers trained with 500 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	9	2	9	2	7	4
Actual Negative	5	34	5	34	5	34
	Acc. 0.86	Prec. 0.209	Acc. 0.86	Prec.0.209	Acc. 0.82	Prec. 0.170
	Recall 0.818	F1 0.333	Recall 0.818	F1 0.333	Recall 0.636	F1 0.269

Fig. 11 – Confusion Matrix of classifiers trained with 500 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	9	2	8	3	7	4
Actual Negative	6	33	7	32	5	34
	Acc. 0.84	Prec. 0.214	Acc. 0.8	Prec. 0.2	Acc. 0.82	Prec. 0.170
	Recall 0.818	F1 0.339	Recall 0.727	F1 0.313	Recall 0.636	F1 0.269

Fig. 12 – Confusion Matrix of classifiers trained with 500 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

1 - 10%

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	6	5	3	8	4	7
Actual Negative	26	63	6	83	6	83
	Acc. 0.69	Prec. 0.869	Acc. 86	Prec. 0.034	Acc. 87	Prec. 0.459
	Recall 0.545	F1 0.15	Recall 0.272	F1 0.061	Recall 0.636	F1 0.081

Fig. 13 – Confusion Matrix of classifiers trained with 1000 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

2 - 10%

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	6	5	3	8	5	6
Actual Negative	25	64	6	83	6	83
	Acc. 0.7	Prec. 0.085	Acc. 0.86	Prec. 0.034	Acc. 0.88	Prec. 0.056
	Recall 0.545	F1 0.148	Recall 0.272	F1 0.068	Recall 0.454	F1 0.101

Fig. 14 – Confusion Matrix of classifiers trained with 1000 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

3 - 10%

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	3	8	3	83	6	5
Actual Negative	6	83	6	8	2	67
	Acc. 0.86	Prec. 0.348	Acc. 0.86	Prec. 0.034	Acc. 0.73	Prec. 0.821
	Recall 0.272	F1 0.061	Recall 0.272	F1 0.061	Recall 0.545	F1 0.142

Fig. 15 – Confusion Matrix of classifiers trained with 1000 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

4 - 25%

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	8	12	13	7	16	4
Actual Negative	9	71	19	61	2	58
	Acc. 0.79	Prec. 0.101	Acc.0.74	Prec. 0.175	Acc. 0.74	Prec. 0.21
	Recall 0.4	F1 0.161	Recall 0.65	F1 0.276	Recall 0.8	F1 0.340

Fig. 16 – Confusion Matrix of classifiers trained with 1000 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	9	11	12	8	12	8
Actual Negative	9	71	9	71	12	68
	Acc. 0.8	Prec. 0.112	Acc. 0.83	Prec.0.114	Acc. 0.8	Prec. 0.15
	Recall 0.45	F1 0.18	Recall 0.6	F1 0.233	Recall 0.6	F1 0.24

Fig. 17 – Confusion Matrix of classifiers trained with 1000 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	11	9	11	9	10	10
Actual Negative	8	72	7	73	7	73
	Acc. 0.83	Prec. 0.132	Acc. 0.84	Prec. 0.130	Acc. 0.83	Prec. 0.120
	Recall 0.55	F1 0.213	Recall 0.55	F1 0.211	Recall 0.5	F1 0.194

Fig. 18 – Confusion Matrix of classifiers trained with 1000 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	3	0	1	2	1	2
Actual Negative	42	155	40	157	2	195
	Acc. 0.79	Prec. 0.018	Acc. 0.79	Prec. 0.006	Acc. 0.98	Prec. 0.005
	Recall 1	F1 0.037	Recall 0.333	F1 0.012	Recall 0.333	F1 0.010

Fig. 19 – Confusion Matrix of classifiers trained with 2000 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	3	0	1	2	1	1
Actual Negative	33	164	4	193	19	178
	Acc. 0.835	Prec. 0.0179	Acc. 0.97	Prec. 0.005	Acc. 0.895	Prec. 0.005
	Recall 1	F1 0.035	Recall 0.333	F1 0.010	Recall 0.333	F1 0.010

Fig. 20 – Confusion Matrix of classifiers trained with 2000 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	1	2	1	2	2	1
Actual Negative	4	193	11	186	20	177
	Acc. 0.97	Prec. 0.005	Acc. 0.953	Prec. 0.005	Acc. 0.895	Prec. 0.011
	Recall 0.333	F1 0.010	Recall 0.333	F1 0.010	Recall 0.666	F1 0.021

Fig. 21 – Confusion Matrix of classifiers trained with 2000 LOC and 10% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	15	6	10	11	12	9
Actual Negative	50	129	8	171	18	161
	Acc. 0.72	Prec. 0.104	Acc.0.905	Prec. 0.055	Acc. 0.865	Prec. 0.069
	Recall 0.714	F1 0.181	Recall 0.476	F1 0.99	Recall 0.571	F1 0.123

Fig. 22 – Confusion Matrix of classifiers trained with 2000 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 5.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	13	8	10	11	11	10
Actual Negative	21	158	8	171	14	165
	Acc. 0.855	Prec. 0.076	Acc. 0.905	Prec.0.055	Acc. 0.88	Prec. 0.062
	Recall 0.619	F1 0.135	Recall 0.476	F1 0.099	Recall 0.5236	F1 0.111

Fig. 23 – Confusion Matrix of classifiers trained with 2000 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 7.

	KNN Positive	KNN Negative	CART Positive	CART Negative	R.Forest Positive	R.Forest Negative
Actual Positive	14	7	15	6	11	10
Actual Negative	21	158	26	153	15	164
	Acc. 0.86	Prec. 0.081	Acc. 0.84	Prec. 0.089	Acc. 0.875	Prec. 0.628
	Recall 0.666	F1 0.145	Recall 0.714	F1 0.158	Recall 0.523	F1 0.112

Fig. 24 – Confusion Matrix of classifiers trained with 2000 LOC and 25% marked lines. The max depth of the CART and RF, and No. of nearest neighbors of KNN are 9.