Linnéuniversitetet
Kalmar Växjö

Master Thesis Project

Predicting Software Defectiveness by Mining Software Repositories

Author: Stanislav Kasianenko
Supervisor: Morgan Ericsson
Examiner: Welf Löwe
Reader: Francesco Flammini
Semester: HT 2017
Course Code: 4DV50E
Subject: Computer Science

Abstract

One of the important aims of the continuous software development process is to localize and remove all existing program bugs as fast as possible. Such goal is highly related to software engineering and defectiveness estimation. Many big companies started to store source code in software repositories as the later grew in popularity. These repositories usually include static source code as well as detailed data for defects in software units. This allows analyzing all the data without interrupting programing process. The main problem of large, complex software is impossibility to control everything manually while the price of the error can be very high. This might result in developers missing defects on testing stage and increase of maintenance cost. The general research goal is to find a way of predicting future software defectiveness with high precision. Reducing maintenance and development costs will contribute to reduce the time-to-market and increase software quality.

To address the problem of estimating residual defects an approach was found to predict residual defectiveness of a software by the means of machine learning. For a prime machine learning algorithm, a regression decision tree was chosen as a simple and reliable solution. Data for this tree is extracted from static source code repository and divided into two parts: software metrics and defect data. Software metrics are formed from static code and defect data is extracted from reported issues in the repository. In addition to already reported bugs, they are augmented with unreported bugs found on “discussions” section in repository and parsed by a natural language processor. Metrics were filtered to remove ones, that were not related to defect data by applying correlation algorithm. Remaining metrics were weighted to use the most correlated combination as a training set for the decision tree. As a result, built decision tree model allows to forecast defectiveness with 89% chance for the particular product. This experiment was conducted using GitHub repository on a Java project and predicted number of possible bugs in a single file (Java class). The experiment resulted in designed method for predicting possible defectiveness from a static code of a single big (more than 1000 files) software version.

Keywords: repository mining, software metric, correlation, defect, bug, natural language processing, Pearson coefficient, Breiman’s decision tree, machine learning.

Preface

I started my master study on September 2016 in Linnaeus University and have received a priceless experience since then.

As a preface, I want to thank my supervisor Morgan Ericsson. After choosing the topic of research, he helped me a lot with guiding through the field and with writing this thesis. I am also grateful for helping me with organizational matters as a responsible person for my exchange program.

Contents

1 Introduction

This thesis project is about researching a problem dictated by modern business directions, which is becoming classic in the machine learning research community: the task of predicting error rate based on existing data history. The problem of defectiveness estimation in software developing is solved by applying decision making algorithms to big data-like structures and extracting such data from source code of analyzed program [15]. This research considers the existing approaches of predicting the defect occurrences and go the complete path from processing the data, exported from the static repository history, to building theoretical prediction model and implementing a prototype prediction system for real data from real big software projects.

1.1 Background and Motivation

So far, different researchers have already proposed possible solutions to prevent future project defects and explained several reasons of software failures, such as financial, technical or social [1], [2]. As stated in the Standish Group Report, only 9%, 16.2%, 28% of all projects in large, medium and small companies respectively are successful. 61.5% of all projects in large companies were challenged on development or production stages [1]. According to another research, 93% of analyzed failed software projects had unrealistic expectations of delivery time and 81% were not estimated correctly [2]. It is important to predict future software defectiveness for increasing efficiency of managing and planning the testing of projects to allow developers to see which parts require more precise checking and to reduce amount of failures. Defectiveness estimation could lead to reducing projects' cost by helping developers to focus on problematic parts of the project.

Current situation in big software projects requires creating programs with as minimum errors as possible. And because currently running software, that is under continuous development, cannot be stopped for detailed analysis, the only instrument that is left is code repository. Given a set of repository data, training system can be configured [18]. This system will come to a point score that indicates the bug amount of the test data in a code segment using appropriate metrics.

The studied field has enough existing model building software for compressing raw code from repositories into evaluable structures as well as existing analytical software for creating more variables to study from previous structures. As for the motivation of this research, the current state of the research area will be described by the means of related studies in this field. Following three papers explain strategies and approaches used in this research.

"Integrating software repository mining" research [3] presents the Sambasore approach, that is based on ideas, technologies of software processing, and has a goal to speed up the integration of tools that support software engineering and the results of open source repositories mining. Using the approach described in this paper it is possible to help to move present developing state where engineers are still relying on the decisions from experience to a new scenario, where decision making is based on collective and historical background. This work helped with understanding developers needs in analyzing version and bug history as well as current project situation.

For the possible research strategies, "Mining Software Repositories – A Comparative Analysis" paper [4] describes a study of various instruments for mining software repositories, based on six existing criteria and three new, that are extensions of previous ones. This research work gave big advantage in this analysis

because it helps engineers to understand and compare various tools and services to quickly check a potential tool instead of depending on scientific «method at random» approach.

"Mining Internet-Scale Software Repositories" [5] is another useful resource for this study. It provided some valuable methods in mining of large-scale software repositories. The researched approach explains retrieving and reviewing programming code of scalable software by bringing together term-based information retrieval procedures with visual information from program structure. This allows to greatly improve software search and retrieval performance. The work provided information about several instruments and ways for preprocessing repository data for repository analysis.

"Quantitative Evaluation of Software Quality Metrics in Open-Source Projects" paper [36] describes how to validate software metrics in order to assure that metric-based software will be used efficiently. It is specifically applied to quantitative approaches and helped with constructing and validating this research's method. The paper also explains how validity and reliability evaluation should be applied to studies and calculations based on software metrics.

On the topic of common approaches that were used in other related studies, it can be said, that machine learning algorithms are applicable to defectiveness prediction tasks. In past years the number of studies, which showed machine learning algorithms, increased in comparison to other approaches. According to D. Wolpert, there is no universal algorithm, that can be used for all possible static software [6]. This makes impossible to use same prediction model on all software in general. As stated in "Reliability and Validity in Comparative Studies of Software Prediction Models", research approaches, that are more reliable, should be developed to be sure, that all comparative studies are accurate [7].

There are several studies regarding how software unit size is related to defects amount [8], [9], and they describe, that number of bugs decreases with decreasing size of unit. But there are also several studies [10], [11], showing that defects amount is related to so-called "optimal size" that is not infinitely large or is of the minimum size. From the third side there are two researches [12], [13] that explain the discovery of not particular relation between software unit's size and amount or density of defects in it. Regarding to latter works, one of goals of this research is to find which metrics correlate with defects and how in a certain scope.

The general motivations are to find new dependencies in code properties itself and to build prediction models for software (finding bugs during its lifetime to help determine future limitations) and to try applying this knowledge to other open source projects. New knowledge of this research should help with understanding current software problems and forecast future fails.

1.2 Problem Statement

The problem itself is finding undiscovered software defects, which can greatly increase project cost or delivery and maintenance time. This kind of bugs is only found at production stage when it is already too late to fix without great impact on the project's workflow. This situation could be avoided if developers or testers would know what part of software should be tested with greater precision. As stated in Standish Group Report, 17.2% of combined statistics for unrealistic time frame, expectations and incompetence led to project failure [1]. This percentage could be reduced with better defects discovering.

The obvious solution is to test every part of software in detail. But for modern big software this would increase development time resulting in unrealistic delivery

terms. This is why automatic approach is required in order to increase software quality. Safety-critical software is an exception to this developing and testing approach. Such systems are extensively tested and verified so they contain minimal number of bugs and have no defect history and are not taken into account in this research.

There are many related studies that were also discovering possibility of defectiveness estimation from software metrics. According to M. Suffian, it is possible to construct a regression model from software metrics to contribute to the testing stage [14]. In this research Six Sigma model was used to predict with 95% success chance. Another study shows two neural networks that were used for predicting with both having 95% chance for 8 different object-oriented software metrics [6]. This research also showed high relations between metrics and bugs.

First step for solving this problem is to retrieve metrics and bug data from source code. Next would be building and applying decision tree for predicting. To make this research applicable to the development process of modern software projects, basic data (software metrics, defects amount with localization, severity) should be gathered from open source software repositories. Since a part of bug data could be not reported yet but been discussed on forums, it is important to choose repository with discussions section for enhancing analyzed defects data. The main difference from existing researches is a complex approach featuring full way from open source repository and verbal discussion to ready model for using on a particular project or projects within same company, programing language etc. Also, even if it is possible to retrieve all existing metrics based on static source code they should be filtered, because not all of them are related to bug data.

The first research question on the way to predict bugs would be establishing relations between any properties that could be retrieved from the source code and bug history in the same repository. Without researching if such data correlate, there is no sense in building a prediction model. If metrics do not relate to defects amount, predicting such bugs would become a basic “coin flip” with 50/50 percent for occurrence/absence of a defect. Bug history should also be augmented with verbal discussions’ data. Second research question is to determine if same metrics correlate universally to defects amount across different projects. Since this question was answered in other studies, it is important to find relations in the scope of this research. After finding correlating parameters, they should be filtered. Filtration is done by removing metrics that do not correlate with defects amount to increase prediction chance. The process is performed by applying correlation algorithm to all possible metrics combination to find the highest correlation percentage one. After finding successful combination, next step is to find out if built model will show similar bug occurrence in source files, containing actual error history, using older version data.

R1	Do metrics correlate with defects amount?
R2	Do same metrics correlate with bugs across different projects equally?
R3	Can we predict number of defects in a software unit with regression models?

On the topic of expected answers to the research questions, the possible answers are:

- R1: Some part of the big variety of software metrics will hypothetically correlate with defects data, others are just filtered and not included in the decision tree constructing

- R2: Some part of the metrics will be correlating to certain extent with bugs across different projects. It is important to mention, that correlation similarity will hypothetically happen only among projects, that are written in the same programming language and are released not too long time between each other
- R3: As stated in [15], [16], it is possible to predict future defect occurrences, the only question is how precise would be trained decision tree in a given scope

1.3 Contributions and Results

Considering the fact, that estimating defectiveness in software projects is not a new task, this particular research is focused on designing a new, detailed way of preparing data and using machine learning to be usable for developers for focusing on specific software parts during testing stage. It is also centered around special ways of obtaining data from open source repositories, like natural language processing for detailing received defect data and shows possibility of applying same metrics relations for different software projects.

Presented projects in this research are analyzed only using last software version, because big products are mostly released in only a few versions as open source, which is not enough for precise prediction. In this case only last software version was used, but it is also possible to use version history if latter is available for multiple versions [17]. Using only last software version also helps to identify future bugs in newly written classes and files, that cannot be done using multiple versions approach because of training the tree only within same software unit but could be achieved with certain changes in machine learning algorithm. This research study considers only Java projects and is neither taking in account other object-oriented programing languages, nor including other types of languages, that may result in impossibility of using this predicting approach in said cases.

This study proved the existence of correlation between extracted metrics from static source code and software defects' history. It was also shown that applying same correlations to another project, even developed by same company within same programing language, is not possible. This means that same metrics do not correlate with defects amount across several projects and should be re-extracted again for different product. Also, applying natural language could enhance bug data, if developers' discussions are available and large-scaled. In the performed experiment adding weights to extracted metrics gave extra 2% to collective defects correlation and resulted in total of 80.88%. The decision tree that was built after these manipulations used 220 entities with bugs for training and 29 for predicting. The total number of 249 classes is based on number of classes with defects from studied software product. During experiment, predicted values were compared to original reported defects by a side-to-side comparison of defects amount predicted to real and coincided in 89% of all cases.

In general, this project resulted in designed approach for estimating defectiveness in big software that can be relatively simply used by developers to increase software quality on the stage of active developing and testing as well as supporting released products.

1.4 Target Group

The target group is people who are responsible for finding and fixing defects in developed software. Though the model itself has no simple UI and appear only as

a runtime digital structure, it can be used not only by developers, but also by managing staff. In the latter case the prediction model as well as repository data preprocessing would require additional handling from development team. This thesis approach has low practically applicable value for small projects, due to chosen predicting algorithm, that requires many classes for training purposes (>1000). It also can be used for learning and gathering statistics of the old software written decades ago or software that is too complex to read in regular way.

1.5 Report Structure

This report is structured as follows. In Background chapter the existing researches will be classified and reviewed. Scientific approach that is used to answer research questions will be described in Method. This chapter also contains a review for all pieces of built software in details. The following Result chapter will show all received results in details. Discussion chapter will explain all received results. In the Conclusions and Future Work part report will be ended with conclusions of gained knowledge and brief plans for future research will be made.

2 Background

2.1 Repository Mining

The most important theory to understand this research is the field of Repository Mining. Often, along with this name, the terms “Knowledge Discovery” and “Data Warehouse” are found [26]. The emergence of these terms, which are an integral part of Repository Mining, is associated with a new spiral in the development of tools and methods for processing and storing data. So, the goal of Repository Mining is to identify hidden rules and patterns in large (very large, considering the nature of “big data”) amounts of information.

The problem is that the data, that is constantly generating, increasing in volume faster than it could be processed manually. On average, a person, with the exception of some individuals, is not able to catch more than couple data relations even in small samples. But traditional statistics, which for a long time claimed to be the main tool for data analysis, is also often to appear to be too simple by its nature to solve problems from real life. It operates with averaged characteristics of the sample, which are often fictitious values (the average solvency of the client, when depending on the risk function or loss function you need to be able to predict the client's consistency and intentions, the average signal intensity, whereas you are interested in the characteristic features and background of signal peaks etc.).

Therefore, the methods of mathematical statistics are useful mainly to test pre-formulated hypotheses, whereas the definition of a hypothesis is sometimes quite a complex and time-consuming task. Modern Repository Mining technologies process information to automatically search for patterns that are the description of any fragments of heterogeneous multidimensional data. Unlike the operational analytical processing of data (OLAP) [27] in Repository Mining, the burden of formulating hypotheses and identifying unusual (unexpected) templates is passed from person to computer. Repository Mining - this is not one, but a combination of a large number of different methods for gaining knowledge. The choice of method often depends on the type of data available and on what information is tried to been obtained. Here, for example, some methods: association, classification, clustering, time series analysis and forecasting, neural networks [28], and so on.

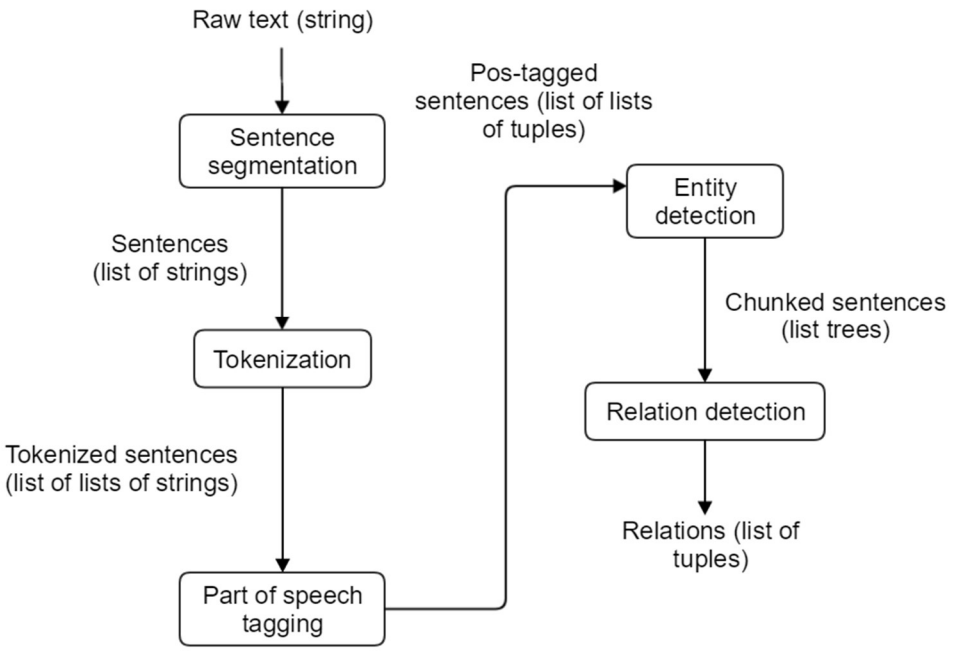
Here are the properties of the detected knowledge, more detailed:

- Knowledge must be new, previously unknown. The effort spent on discovering knowledge that is already known to the user does not pay off. Therefore, the value is represented by new, previously unknown knowledge
- Knowledge must be nontrivial. The results of the analysis should reflect the unobvious, unexpected patterns in the data that constitute the so-called hidden knowledge. The results that could be obtained in more simple ways (for example, visual viewing) do not justify the use of powerful Data Mining methods
- Knowledge should be practically useful. The knowledge found should be applicable, including the new data, with a sufficiently high degree of reliability. The usefulness is that this knowledge can bring a certain benefit in their application
- Knowledge should be accessible to a human’s understanding. The found patterns should be logically explainable, otherwise there is a possibility that they are random. In addition, the knowledge found should be presented in a manner that is understandable to a person

In Data Mining, models serve to represent the knowledge gained. Types of models depend on the methods of their creation. The most common are: rules, decision trees, clusters and mathematical functions.

2.2 Natural Language Processing

To make defect data more detailed it is reasonable to not only analyze bug reports but also to process human language in code discussions, that is to use natural language processing. The task of natural data clustering is the task of grouping objects into groups, similar on certain grounds - one of the fundamental issues of Repository Mining [29]. Often, data clustering has an applied nature and a list of areas in which clustering is applied are quite wide: text analysis, segmentation of images, forecasting of various events, marketing. Modern natural processing tasks have clustering as the first stage of data processing for the formation of groups of characteristics for which, in the future, other methods and models will be applied. It should be noted that the task of document clustering has much in common with task of classifying texts into a pre-created and pre-filled system of categories. Despite the preliminary similarity, clustering has a number of features that must be taken into account when solving problems. Contrary to well-studied and effective in practice methods of classification, approaches to solution of the problem of clustering are to some extent poor and limited practical applicability. The main reason for this difference is that the task of clustering is very difficult to formalize. While there are objective and sufficiently accurate methods for assessing quality 12 classification, the assessment of the quality of clustering, as a rule, is based on the expert's opinion and is difficult to express in numerical terms. In other words, one of the fundamental problems of text document clustering is evaluation of the quality of the results obtained, since there is no single, generally recognized and applicable in all cases, the method of evaluation. The typical algorithm of natural language processing is shown in Figure 2.1



.

Figure 2.1: Natural language processing algorithm.

In general, the clustering problem is based on proximity metrics [30]. The source documents are presented as a vector in the space of certain signs. Approaches for the formation of a vector of features can significantly differ from each other and metrics can be calculated in different ways and are a separate task. In the simplest case, each characteristic corresponds to the presence of a word or phrase in the source text. The value of the component can be also determined in various ways, for example, the component can be true (or 1) if the word / phrase in question is present in this text or zero in the opposite case; value can be calculated to find the number of occurrences of the word in question in the document (frequency of occurrence) or to be calculated by some other more complex formulas, for example, take into account the average occurrence of a specific word by the current set of text regarding the entire body of document. A measure of closeness between the texts in this case will be calculated as a scalar product between vectors.

2.3 Metrics and Correlation

All related discussed studies in this field used software metrics as the only measurable values from static source code. They are used to compare number of defects found in a software's unit to the software itself. The measure of comparison is correlation and is used to how which parameters are more related. Since quantitative methods have proven themselves in other areas, many computer science theorists and practitioners have tried to transfer this approach to software development. As Tom DeMarco said, "you cannot control what you cannot measure". In general, the use of metrics allows project and enterprise managers to study the complexity of a being developed or even already developed project, assess the scope of work, the style of the program being developed, and the efforts each developer spent to implement a particular solution. However, metrics can serve only as advisory characteristics, they cannot be fully guided, since in software development programmers are trying to minimize or maximize a measure for their program, and it can resort to tricks up to a decrease in program performance. In addition, if, for example, a programmer wrote a small number of lines of code or made a small number of structural changes, this does not mean that he did nothing, or it could mean that the program defect was very difficult to find. The latter problem, however, can partially be solved by using complexity metrics, because in a more complex program, the error is more difficult to find. As an example, the set of metrics used can include [31]:

- Bugs per line of code
- Code coverage
- Cohesion
- Comment density
- Coupling
- Cyclomatic complexity (McCabe's complexity)
- Halstead Complexity
- Maintainability index
- Number of classes and interfaces
- Number of lines of code
- Program execution time
- Program load time
- Program size (binary)
- Weighted Micro Function Points

It is worth to note that there is no universal metric. Any controlled metric characteristics of the program must be controlled either depending on each other or depending on the specific task, in addition, hybrid measures can be applied, but they also depend on simpler metrics and also cannot be universal. Strictly speaking, any metric is just an indicator that depends heavily on the language and style of programming, so no measure can be built into the absolute and make any decisions based only on it. Example of relations between attributes are shown in Figure 2.2. In this figure quality attributes are calculated based on a specific set of internal attributes (basic software metrics). After extracting metrics, they are analyzed to find any useful correlations. The most important goal of data analysis is the study of objectively existing connections between metrics. In the process of a relational study of these links, it is necessary to identify the cause-effect relationships between the indicators, i.e. how much the change in some indicators depends on the change in other indicators.

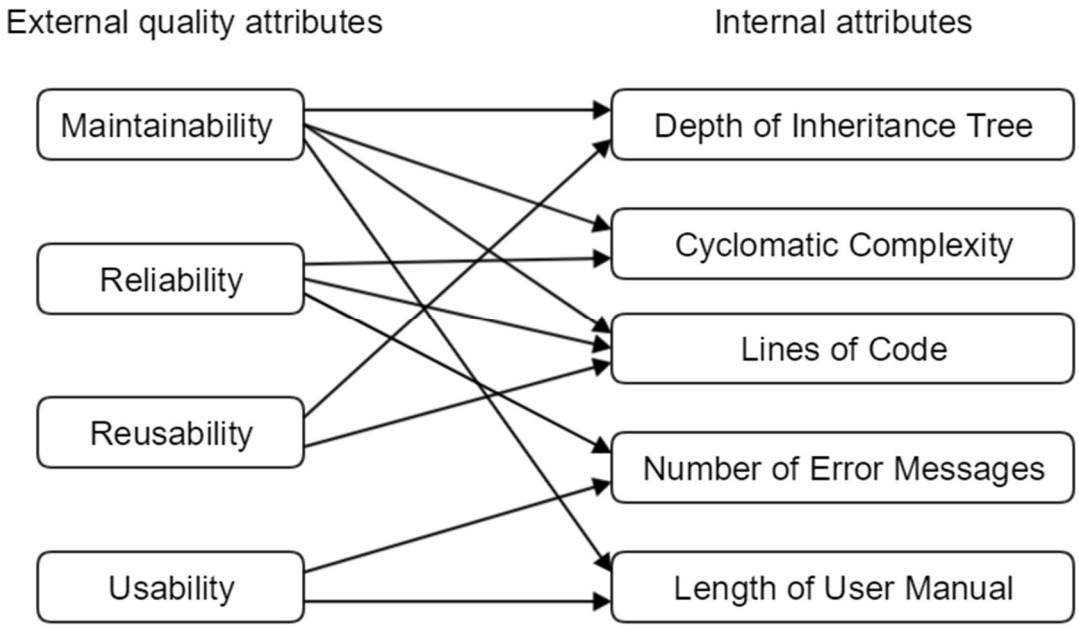


Figure 2.2: Types of software metrics.

Correlations could be connected with a link where the impact of individual factors manifests itself only as a tendency (on average) in the mass observation of actual data. Examples of correlation dependence may be the relationship between the size of the bank's assets and the amount of the bank's profit, the growth of labor productivity and the length of service of employees.

The simplest variant of the correlation dependence is the pair correlation, i.e. the relationship between two characteristics (productive and factorial or between two factorial). Mathematically, this dependence can be expressed as the dependence of the effective exponent y on the factor exponent x. Connections (graph edges) can be direct and reverse. In the first case, with an increase in the sign of x, the sign of y also increases, when the feedback with increasing sign of x decreases, the sign of y decreases. Metrics scatter plot example is shown in Figure 2.3.

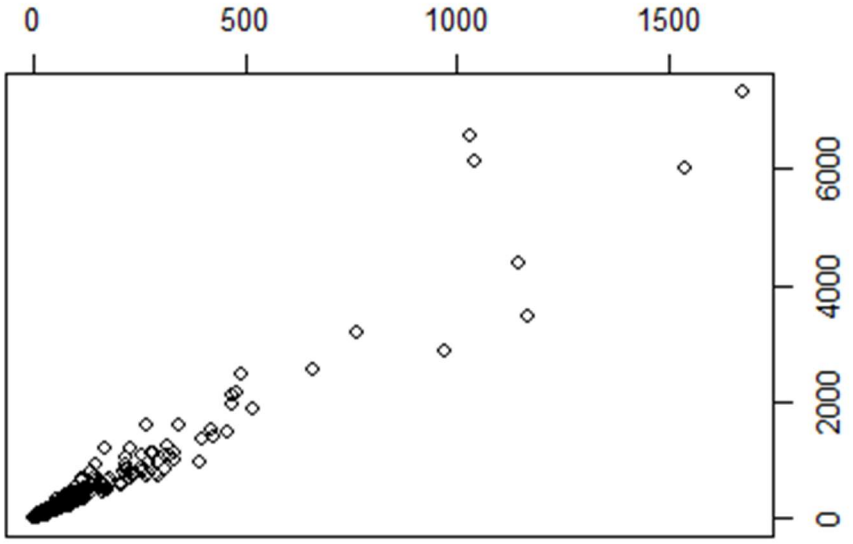


Figure 2.3: Software metrics Weighted Method Count (WMC) as X-axis and Lines of Code (LOC) as Y-axis scatter.

2.4 Decision Tree

Building decision trees from datamined software metrics and using them to predict future defects in development is not new and multiple approaches were already highlighted by many other researches [21]. A decision tree (also called a classification tree or regression tree) is a decision support tool used in statistics and data analysis for predictive models. The structure of the tree is "leaves" and "branches". On the edges (branches) of the decision tree, attributes are recorded, on which the objective function depends, in the leaves the values of the objective function are recorded, and in the remaining nodes - the attributes for which the cases differ. To classify a new case, you have to go down the tree to the leaf and output the corresponding value. Similar decision trees are widely used in data analysis. The goal of decision tree approach is to create a model that predicts the value of the

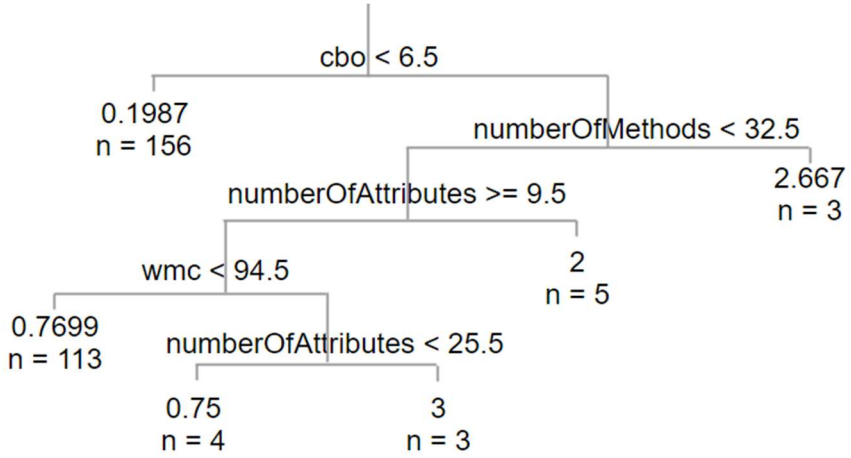


Figure 2.4: Part of Breiman's decision tree.

target variable based on several variables at the input. An example of ready-to-predict decision tree is shown in Figure 2.4. It has a metric with simple logic statement in each node and average amount of found bugs in leaves with number of

training sets ended up there. To predict a defect count, an entry (software unit with calculated values for each metric) is processing from the top of the tree and on each node follows left branch if the statement is true (metric value less or more than stated number). After reaching the leaf, the top number represent average amount of predicted bugs in a unit.

Each leaf represents the value of the target variable, changed during the movement from the root to the leaf. Each internal node corresponds to one of the input variables. The decision tree model can also be "learned" by dividing the original sets of variables into subsets, based on testing attribute values. This is a process that is repeated on each of the received subsets. The recursion is completed when the subset at the node has the same values of the target variable, so it does not add value to the predictions. The top-down induction of the decision tree (TDIDT) [32], is an example of an absorbing "greedy" algorithm and is by far the most common decision tree strategy for data, but this is not the only possible strategy. In data analysis, decision trees can be used as mathematical and computational methods to help describe, classify and summarize a set of data that can be written as follows: $(x, Y) = (x_{1,}x_{2,}x_{3\ldots}x_{k,}Y)$, where the dependent variable Y is the target variable that needs to be analyzed, classified and generalized. The vector x consists of the input variables $x_{1,}x_{2,}x_{3}$ etc., which are used to perform this task.

The decision trees used in Repository Mining [21] [22] [23] are of two main types:

- A tree for classifying when the predicted result is the class to which the data belongs
- A tree for regression, when the predicted result can be considered as a real number (for example, the price of the house, or the length of the patient's stay in the hospital)

The terms mentioned above were first introduced by Breiman et al. The listed types have some similarities (recursive construction algorithms), as well as some differences, such as the criteria for selecting a partition at each node [18].

Some methods allow you to build more than one decision tree (ensembles of decision trees):

- Bagging over decision trees, the earliest approach. Constructs several solution trees, repeatedly interpolating the data with a replacement (bootstrap), and as a consensus response gives the result of voting trees (their average forecast) [19]
- The "Random Forest" classifier is based on bagging, but in addition to it randomly selects a subset of characteristics in each node, in order to make the trees more independent
- Boolean over trees can be used for both regression and classification problems [20]. One implementation of the bootstrap over trees, the XGBoost algorithm, was repeatedly used by the winners of the data analysis competition
- "Rotation of the forest" - trees in which each decision tree is analyzed by the first application of the principal component method (PCA) to random subsets of input functions [21]

There are several reasons to use decision tree method for defectiveness estimation. It is simple in understanding and interpretation. People are able to interpret the results of the decision tree model after a brief explanation of how the tree works. Tree does not require preparation of data. Other techniques require the normalization of data, the addition of dummy variables, and the removal of missing data. Tree model is also able to work with both categorical and interval variables.

Other methods work only with data where only one type of variable is present. For example, the relationship method can be applied only to nominal variables, and the method of neural networks only on variables measured on an interval scale. Decision tree uses the “white box” model. If a certain situation is observed in the model, then it can be explained with the help of Boolean logic. An example of a “black box” can be an artificial neural network, since the results of this model cannot be easily explained. The tree also allows you to evaluate the model using statistical tests. These tests make possible to assess the reliability of the decision tree model, which is considered to be a reliable method. The method works well even if the initial assumptions included in the model were violated. This all together allows to work with a large amount of information without special preparatory procedures, like cleaning or sampling. This method does not require special equipment for working with large databases.

3 Method

3.1 Scientific Approach

Among two possible research methods, qualitative and quantitative, the quantitative method was chosen [34]. Despite having one of the research's goals to build a prediction model, qualitative is based on mainly gathered verbal data and is only useful on the earliest stages of similar works. The main goal of qualitative research is to give a complete, full description of the stated problem. Quantitative research is used in studies for calculating and classifying numerical data to express final observations, like resulted correlation values or prediction accuracy. That is the reason why quantitative method was chosen for this work.

3.2 Method Description

Figure 3.1 shows visualizing of the whole analysis process that is described below. For a better clarification of the process component (Figure 3.2) and activity (Figure

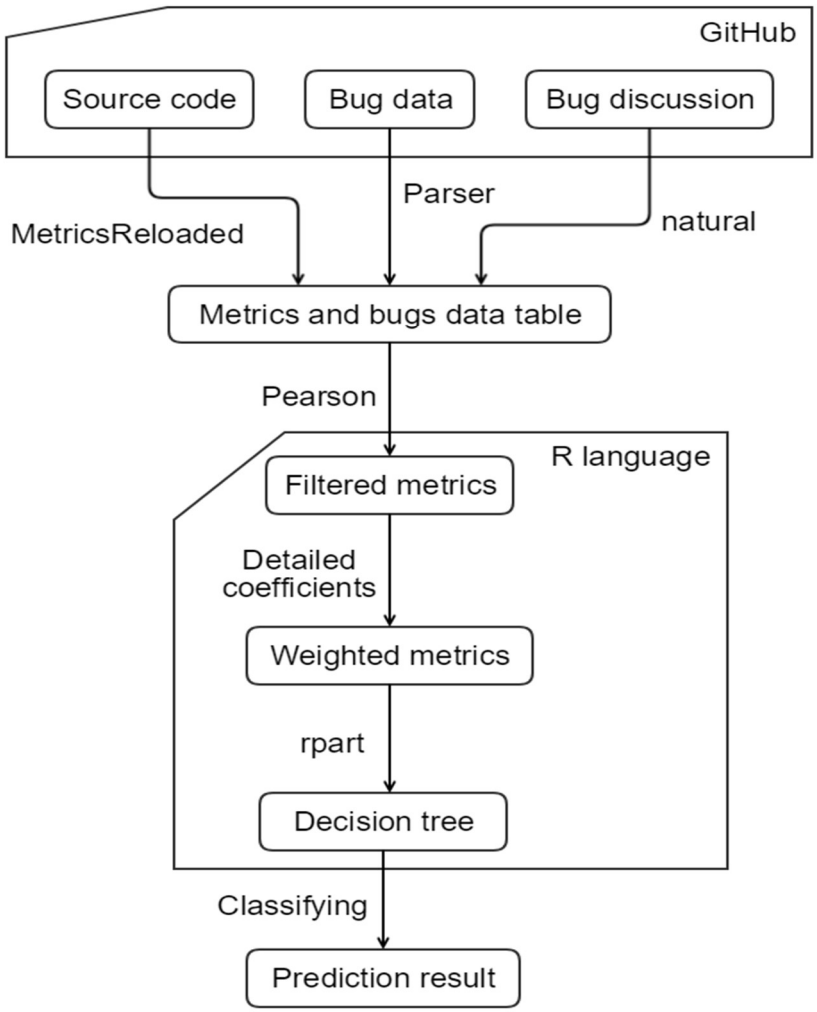


Figure 3.1: Research method structure.

3.3) diagrams are also presented. During this analysis, bug repository history data was used with simple as well as custom metrics to predict the defect density value

for a given project or a module. In this study, artificial neural networks and decision tree approaches were used to predict the defect density values for a testing data set.

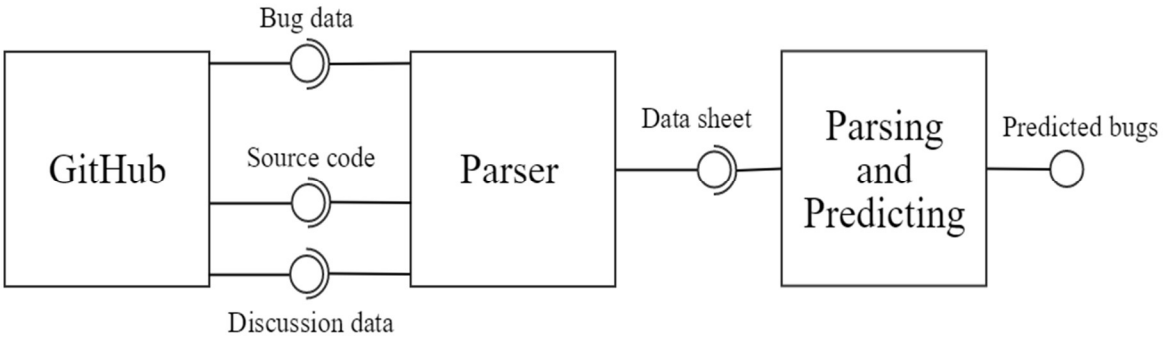


Figure 3.2: Component diagram.

Regression models and learning systems offer a general framework for representing functional mappings between a group of input variables and a group of output

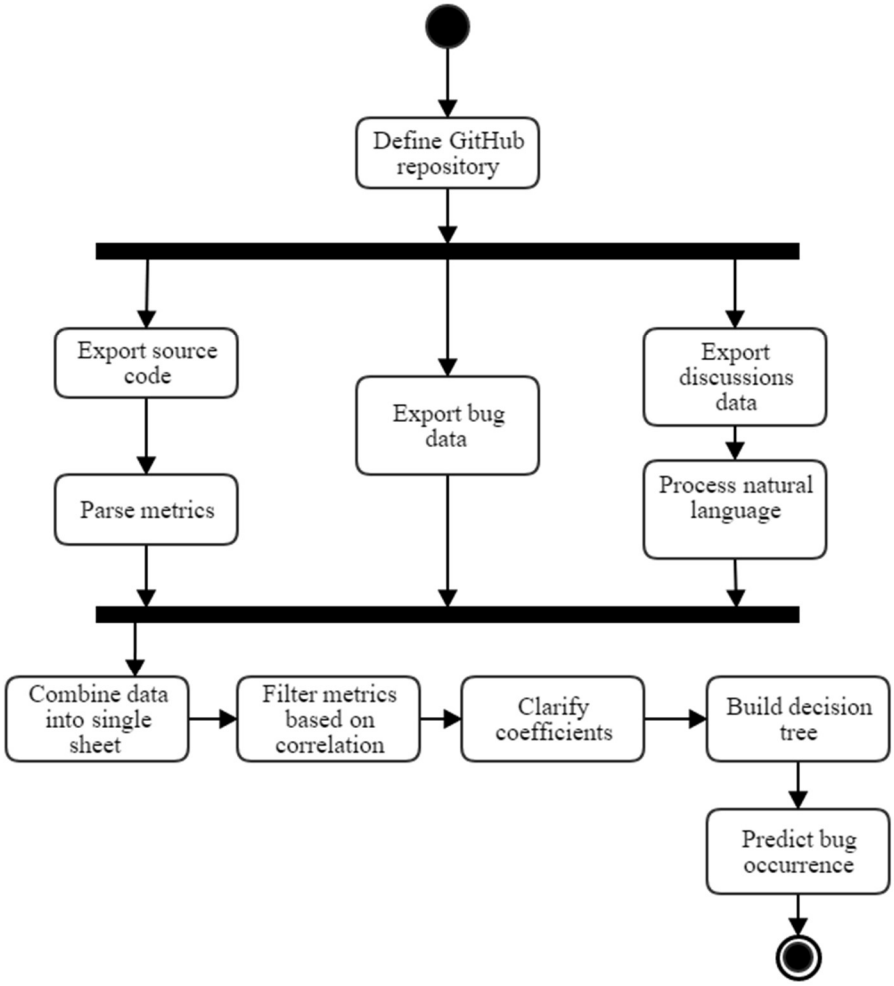


Figure 3.3: Activity diagram.

variables. This can be achieved by representing the function of many variables in terms of compositions of functions of one variable, that are referred to as activation functions. Decision trees are one among the most common approaches for each classification and regression kind predictions. They're generated based on specific

rules. Decision tree is a classifier in a tree structure. Leaf node is the outcome obtained. It's computed with relation to the current attributes. Decision node relies on an attribute, that branches for every potential outcome for that attribute. Decision trees are often thought as a sequence of queries, that results in a final outcome.

Every question depends on the previous question therefore this case ends up in a branching within the decision tree. Whereas generating the decision tree, the final goal is to reduce the common range of queries in every case. This task provides increase within the performance of prediction. Additionally, to overcome the over-fitting downside pruning to reduce the output variable variance within the validation data was used by choosing a less complicated tree than the one obtained once the tree building algorithm stopped, however one that's equally correct for predicting or classifying "new" observations. Within the regression kind prediction experiments regression trees were used which can be thought of as a variant of decision trees, designed to approximate real-valued functions rather than getting used for classification tasks. In the experiments, initially decision tree way to perform a regression-based prediction over the entire data set was applied.

To evaluate the quality and accuracy of conducted experiment with reference to the F-score [33], a formula based on data precision and recall was calculated with confusion matrix approach. The standard deviation of the data set is in fact the average harmonic of those measures. The F-measure reaches a maximum (“1”) in terms of completeness and accuracy and is close to “0” if one of the arguments is close to zero. The regression model predicts the presumably defected modules in a data set, besides it offers an estimation of the defect density within the module that's expected as defected. Therefore, the model helps to focus the efforts on specific suspected elements of the code so most amount of time and resources were saved during software developing.

class	CBO	DIT	LCOM	NOC	RFC	WMC
sk.controller.FieldController	6	1	1	0	58	34
sk.controller.Main	2	2	2	0	27	2
sk.controller.PickerController	5	1	1	0	52	61
sk.model.GameModel	7	1	1	0	24	28
sk.model.Ship	2	1	2	0	7	13
sk.Utils	4	1	1	0	5	1
sk.Utils.CellStates	4		0		0	0
sk.Utils.MoveStates	3		0		0	0
test.GameModelTest	4	1	1	0	13	8
test.ShipTest	1	1	2	0	7	3
Total						**150**
Average	3,80	1,12	1,10	0,00	19,30	15,00

Figure 3.4: MetricsReloaded sample analysis result.

Further in this chapter all research steps will be explained in details. For the most beginning a ready bug prediction data set [22], created especially for experimenting with error predicting, was used. It contains following software systems: Equinox Framework, Lucene and Mylyn. Those are big enough as well as complex in terms of possible metrics and rich with bugs. They helped to understand

the basic methods and flows of the whole process. Next task was to determine metrics and to retrieve their values. A free software called MetricsReloaded [35] which is a plugin for IDE Intellij Idea was used for this task. It gathers static metrics from source code of different languages and outputs in widely-used data types (XML or CSV), so it allows automating this part of process. Reason behind MetricsReloaded choice is that it is easy to integrate into popular development tool IntellijIdea and has enough different metrics to analyze. MetricsReloaded example output is shown in Figure 3.4. The primary goal for preparing data for this step is to extract it from source code repository and retrieve all the metrics. The most suitable repository to use is GitHub. It offers not only raw software data itself but also provides defect reports as well as project discussions that were used for correcting bug data by the means of natural language processor. The source code was retrieved by simple request or manual UI functionality, while defects data was received from public API by manually developed software. For this step GitHub public API was used. Starting with issues data by invoking GET request for API address "/repos/:owner/:repo/issues" defect reports were received. Using parameter "url" list of changed classes were received by using following address: "/repos/:owner/:repo/issues/:issueNO/files". Resulting list contained file name parameter "filename" that was extracted and stored in defects datasheet.

Next step was to compose all gathered information into one table-like structure to apply filtering algorithms. Because of having too many software metrics it would not only take long to process all the data by any software without using special computing hardware but also could interfere with prediction results making them inaccurate, filtering metrics was required. Filtering metrics means that Pearson correlation algorithm was used to determine which metrics were correlating with bug data and how this happened. For this step, all metrics are summed up into one value and compared with correlational algorithm to number of bugs. Filtering is done by finding a combination of metrics with highest correlation value. The summands are not only summed, but also subtracted from resulting value in order to determine if some of the metrics have reverse correlation. Basic example of how coefficient is reflecting data in this correlation algorithm is shown in Figure 3.5. This figure shows how coefficient is calculated among simple different data.

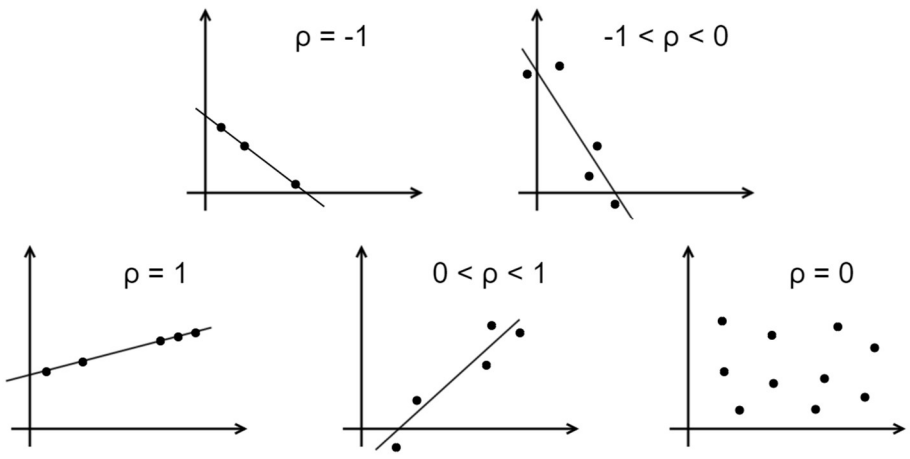


Figure 3.5: Simple diagrams with different correlation coefficient (ρ).

To enhance filtered metrics and determine to what extent they were related to defects data simple search algorithm was used. It applies weights to all metrics in all possible combinations to resolve in the highest correlation percentage. These weights were used for future prediction model building.

Next major step was to take bug repository and enhance defects data with results received from processing natural language in project's discussion. This was done by text parsing, because most of these reports contain phrases similar to "Fixed bug in Utils.java". Parsing the class allows calculating error amount for every file or lesser structure. As it is suggested in "Predicting Faults from Cached History" [23] such parsing can use so-called "trust model", increasing trust for bug message if it contains words like "fixed", "bug". This approach helped in finding extra non-reported defects by mentioned class names. These found mentions were added to bug data by incrementing amount per file (java class). Free open source library "natural" was used for the purpose of this step. This library is free and simple parsing of natural language into so-called "tokens" and is sufficient to the purpose of enhancing bug data. Tokenizing example highlighting the work of this library is shown in Figure 3.6. This figure shows sets of three code lines, each representing a library's feature example. First line shows name of method, second shows input line, and third shows output from the library after tokenizing. Similar to one of the previous steps, GitHub API was used for extracting comments to pull requests. To perform this, request was made to the GitHub's repository API, using URL "/repos/:owner/:repo/pulls/:pullNO/reviews/:reviewNO/comments", and received parameter "body", which is the comment itself, was passed to natural library. Library is tokenizing the string and is instructed to find words by the pattern "*.java" if other words "bug", "defect", "problem" were found and no words "solved", "resolved", "fixed" exist. At this point everything was ready for building prediction model.

For the further steps programing language R was be used. It is created specifically as a statistical software for data analysis and became the standard for programs in statistics field. It also contains needed libraries for building decision trees. By the means of R the prediction model was built. This model is a decision

```
tokenizer = new natural.TreebankWordTokenizer();
console.log(tokenizer.tokenize("my dog hasn't any fleas."));
// [ 'my', 'dog', 'has', 'n\'t', 'any', 'fleas', '.' ]

tokenizer = new natural.RegexpTokenizer({pattern: /\-/});
console.log(tokenizer.tokenize("flea-dog"));
// [ 'flea', 'dog' ]

tokenizer = new natural.WordPunctTokenizer();
console.log(tokenizer.tokenize("my dog hasn't any fleas."));
// [ 'my',  'dog',  'hasn',  '\'',  't',  'any',  'fleas',  '.' ]
```

Figure 3.6: Application of NLP library.

tree that was made by specific library "rpart" for R language. This library implements Classification And Regression Tree (CART) algorithms introduced by Leo Breiman in 1984 [18] and can be seen in Figure 2.4. Basic idea behind

regression tree was explained in Background chapter. Unlike common regression machine learning approaches, it requires training data but results in better quality. This software showed if any correlation between metrics and bug history exists. If none found, metrics need to be combined, and experiment repeated. After this it was required to find any relations, that could be applied to other projects. For example, apart from development environment Eclipse, another type of software could be used for analysis, like a Java EE application server JBoss. Testing models on other big programs showed if the design is universal and answered second research question.

Next step was defectiveness prediction itself. Constructed model was applied to a newer software version or generally to another part of the software, that was not included in learning subset, and produced prediction in terms of bug amounts and place. Predicted value needs to have at least 80% precision with real value to be considered successful. As suggested in "Predicting Defects for Eclipse" paper [24], quality model of prediction can be divided in 4 types: true positive (found bugs and they exist), true negative (didn't find bugs and they don't exist), false positive (found bugs and they don't exist) and false negative (didn't find bugs and they exist). Only the last type results in model failure, as false positive is just an extra safety in developing.

To check the research accuracy, F score [33], based on previous 4 score types, could be calculated. It shows the harmonic average between test's precision and recall. Figure 3.7 shows its formula.

$$F = 2 \cdot \frac{1}{\frac{1}{recall} + \frac{1}{presicion}} = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Figure 3.7: F measure formula.

For getting F measure first precision and recall are calculated using formula shown in Figure 3.8.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Figure 3.8: Precision and recall formulas.

To calculate defectiveness of a software estimated expected quality (EstQuality) as an implementation effectiveness metric and defect detection state (DetState) as a testing effectiveness metric are used [38]. Formulas for calculating these values are shown in Figure 3.9, where Size is total size of predicted code part in thousands of lines of code (KLoC), Defects is number of real defects and EstDefects is number of predicted defects.

$$EQ = \frac{Size}{EstDefects}$$

$$DetState = \frac{Defects}{EstDefects} \cdot 100\%$$

Figure 3.9: Estimated expected quality and defect detection state formulas.

After conducting an experiment by the described method, it is expected to have an approach that allows predicting future errors from project's version and bug history and helps developers to save time and effort creating big programs.

3.3 Ethical Considerations

Working with natural human language in the form of internet chat and software units which might contain developers' signatures questions ethical considerations of this research. Despite being a public data, all the names are not included in extracted data and will not be shown in this work.

4 Results

The first step of this research is choosing a project and retrieving software metrics. The analyzed project is written in Java, located on GitHub and has approximately 997 classes, which is just enough for research purpose. The project is called "eclipse.jdt.core" and is located in "eclipse" user repository. To extract metrics a plugin MetricsReloaded is used in IDE Intellij Idea. Software project is loaded into the development studio by direct repository link and is processed by the plugin. This altogether result in metrics data table. In this research all basic metrics provided by MetricsReloaded are extracted:

- Coupling Between Objects (CBO)
- Depth of Inheritance Tree (DIT)
- Number of other classes that reference a class (FANin)
- Number of other classes referenced by a class (FANout)
- Lack of Cohesion of Methods (LCOM)
- Number of Children (NOC)
- Number of Attributes (NOA)
- Lines of Code (LOC)
- Number of Methods (NOM)
- Response for Class (RFC)
- Weighted method count (WMC)

After getting all metric data defects are retrieved. This is done by GitHub API. Part of resulting defects data is shown in Figure 4.1.

BUGS	classname
1	org/eclipse/jdt/core/compiler/CompilationParticipant
1	org/eclipse/jdt/core/CompletionContext
1	org/eclipse/jdt/core/CompletionProposal
1	org/eclipse/jdt/core/CorrectionEngine
2	org/eclipse/jdt/core/dom/AnnotationBinding
1	org/eclipse/jdt/core/dom/ASTConverter
1	org/eclipse/jdt/core/dom/ASTParser
0	org/eclipse/jdt/core/dom/ASTRequestor
1	org/eclipse/jdt/core/dom/CharacterLiteral
0	org/eclipse/jdt/core/dom/ChildListPropertyDescriptor
1	org/eclipse/jdt/core/dom/DefaultBindingResolver
1	org/eclipse/jdt/core/dom/DocCommentParser
0	org/eclipse/jdt/core/dom/MemberRef
1	org/eclipse/jdt/core/dom/rewrite/ASTRewrite
0	org/eclipse/jdt/core/dom/SingleMemberAnnotation
1	org/eclipse/jdt/core/dom/StringLiteral
0	org/eclipse/jdt/core/dom/SuperFieldAccess
1	org/eclipse/jdt/core/dom/TypeBinding
2	org/eclipse/jdt/core/formatter/DefaultCodeFormatterConstants
1	org/eclipse/jdt/core/JavaConventions
4	org/eclipse/jdt/core/JavaCore
0	org/eclipse/jdt/core/jdom/DOMFactory
2	org/eclipse/jdt/core/NamingConventions

Figure 4.1: Received bug data with number of bugs in each class.

To enhance bug data natural language processing library “natural” is used. The result is shown in Figure 4.2 and is a list of classes, received after running “natural” library, that are merged with main defect datasheet. Every line of this data means that class contains extra undocumented bug, and such class should have number of bugs incremented. Note, that since big and reliable projects like Eclipse have low amount of submitted bugs data was shortened. This means that some of the classes that contain 0 bugs were not taken into account. To be more exact classes with 0 bugs were added in amount of 20% from bug-containing classes.

org/eclipse/jdt/internal/formatter/Scribe
org/eclipse/jdt/internal/formatter/DefaultCodeFormatterOptions
org/eclipse/jdt/internal/eval/CodeSnippetSingleNameReference
org/eclipse/jdt/internal/core/search/matching/DeclarationOfReferencedMethodsPattern
org/eclipse/jdt/internal/core/index/Index
org/eclipse/jdt/internal/core/BinaryMethod

Figure 4.2: Received NLP data.

After grouping and unifying all data together a total of 11 different metrics was received. In order to answer second research question, all previous steps are executed to gather the data of another software project “Eclipse PDE UI”. Correlations between same metric and bug amount in a file are compared to determine possibility of applying gained knowledge to other software products that are written in the same programing language. For the two similar products same metrics have different correlations between same metric and defect amount. Output scatter plots are shown on Figure 4.3 and Figure 4.4.

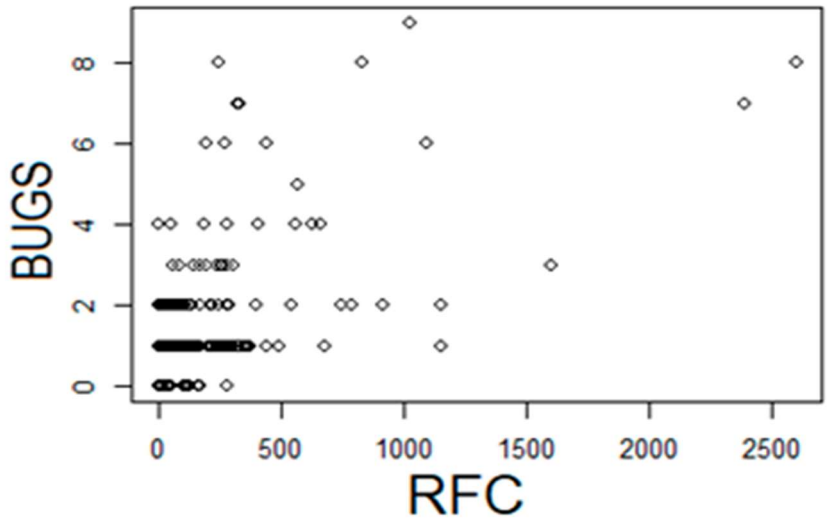


Figure 4.3: Eclipse Core bugs to RFC (62.58% correlation).

Not all the metrics correlate enough with defects and some of them might even interfere successful predictions. Because of this they should be filtered. There are three main correlation coefficients: Pearson, Spearman, Kendall. Using built in R language “cor” function weights of -1, 0 and 1 are applied to all metrics in all possible combinations. 0 weight means metric is excluded from current combination, -1 means reverse correlation calculation, 1 normal correlation.

Current amount of various sets is (3^11)/2, that is equal to 88574 algorithm operations. (3^11)/2 formula means 11 metrics having 3 possible states (-1, 0, 1) and divided by 2 to exclude mirrored results, i.e. all metrics have 1 or -1 coefficient, which will result in same correlation coefficient. Pearson algorithm usage takes approximately 38 minutes for a 249-class project on Intel i5-4670. Best results from

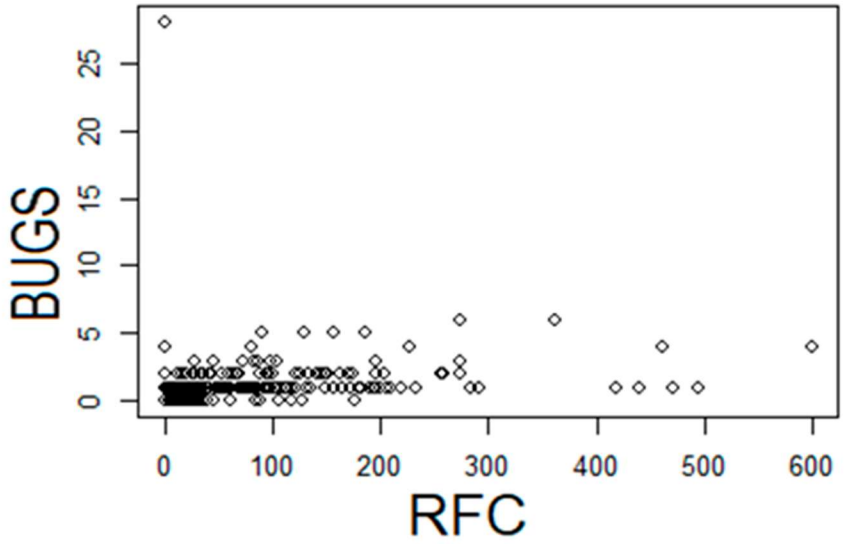


Figure 4.4: Eclipse PDE bugs to RFC (16.06% correlation).

these calculations are shown in Figure 4.5. In this figure first column has sorted correlation coefficient for each metrics combination (combination is set accordingly to other metrics' factors). This means that best correlation percentage (strongest

	CBO	DIT	FANin	FANout	LCOM	NOC	NOA	LOC	NOM	RFC	WMC
0.788343	1	1	1	1	0	1	1	0	-1	1	1
0.788266	1	0	1	1	0	1	1	0	-1	1	1
0.788181	1	-1	1	1	0	1	1	0	-1	1	1
0.78813	1	1	1	1	0	0	1	0	-1	1	1
0.788052	1	0	1	1	0	0	1	0	-1	1	1
0.788036	1	1	1	1	0	1	0	0	-1	1	1
0.787999	1	0	1	1	0	1	0	0	-1	1	1
0.787967	1	-1	1	1	0	0	1	0	-1	1	1
0.787955	1	-1	1	1	0	1	0	0	-1	1	1
0.787908	1	1	1	1	0	-1	1	0	-1	1	1
0.78783	1	0	1	1	0	-1	1	0	-1	1	1
0.78782	1	1	1	1	0	0	0	0	-1	1	1
0.787783	1	0	1	1	0	0	0	0	-1	1	1
0.787744	1	-1	1	1	0	-1	1	0	-1	1	1
0.787738	1	-1	1	1	0	0	0	0	-1	1	1
0.787595	1	1	1	1	0	-1	0	0	-1	1	1
0.787557	1	0	1	1	0	-1	0	0	-1	1	1
0.787511	1	-1	1	1	0	-1	0	0	-1	1	1
0.786402	1	1	1	0	0	1	1	0	-1	1	1
0.786323	1	0	1	0	0	1	1	0	-1	1	1
0.786237	1	-1	1	0	0	1	1	0	-1	1	1
0.786186	1	1	1	0	0	0	1	0	-1	1	1

Figure 4.5: The highest outcome from Pearson coefficient calculation.

relation between metrics and bug amount) is 78.83% for combination of CBO, DIT, FANin, FANout, NOC, NOA, -NOM, RFC, WMC. On the topic of other algorithms, they show significantly less relations with bug data, as can be seen in

Figure 4.6. This figure shows side-to-side comparison of different correlation algorithms calculated from same metrics combination (referenced to first column of Figure 4.5).

After calculating needed weights for the metrics refinement of these coefficients is calculated. This step is done to find if metric correlate more or less in comparison to others. For this task a listing (brute-force approach) through weights combination is used from 0.1 weight to 1 and -1 to -0.1 accordingly. This is resulting in final correlation coefficient of 0.80882334 or 80.88%, that is enough for building decision tree. The final weights set is: 0.1*CBO, 0.8*DIT, FANin, FANout, NOC, 0.1*NOA, -NOM, 0.3*RFC, 0.1*WMC. Next step is creating a machine learning structure to predict defect occurrence. "rpart" library is used to build Breiman's regression tree from computed data. To setup the function it is important to specify several parameters.

Pearson	Kendall	Spearman
0.7883427	0.455999	0.5735192
0.7882657	0.4554492	0.573
0.7881813	0.4553468	0.5726711
0.7881295	0.455243	0.572554
0.7880521	0.4550904	0.5720763
0.7880364	0.4550699	0.5719708
0.7879995	0.4549708	0.5719035
0.7879673	0.4549127	0.5719029
0.7879547	0.4546109	0.5718427
0.7879075	0.4544162	0.5718184
0.7878297	0.4544035	0.5717672
0.7878202	0.454294	0.5716155

Figure 4.6: Correlation algorithms comparison.

The first is "minsplit". This parameter is responsible for creating a new leaf in the tree when minimum number of observations has happened. If it is equal "1" that means new leaf would be created every time and successful prediction is possible only using almost equal import data. A total of 249 classes was split to 29 testing and 220 learning after a common advice to have 80% of learning and 20% of testing data. For the learning set of 220 elements value of 5 shows the best result. Another parameter is a tree-building method called "method". For this particular research "anova" [25] has the biggest use. It calculates average amount of bugs that were classified for given leaf. For example, if leaf was accessed by entries with 3, 4, 5 and 6 bugs count, it would have the value of 4,5 and predict this number of bugs during testing phase. Other 29 entries are saved for testing prediction chance. The resulting tree is shown in Figure 4.7. As explained earlier, to predict number of defects in a class, it follows a tree from top to a particular leaf. Decision to follow left branch or right one is made from value of according metric. After reaching leaf, the average number of predicted bugs is the top number and "n=" shows amount of entries, that reached this leaf during training stage.

The final step to answer third research question is to travel by the predicting tree model using metrics of remained 29 entries and compare predicted value to real. For this particular project prediction chance, calculated by comparing amount of predicted number of bugs to actual ones, is 89%. For calculating this value, it is first required to divide data in 4 categories as shown in Table 4.1.

	Real positive	Real negative
Predicted positive	TP: 8	FP: 1
Predicted negative	FN: 2	TN: 18

Table 4.1: F measure table calculated from testing data.

This is the result of calculations: precision = 0.89, recall = 0.8, F measure = 0.84.

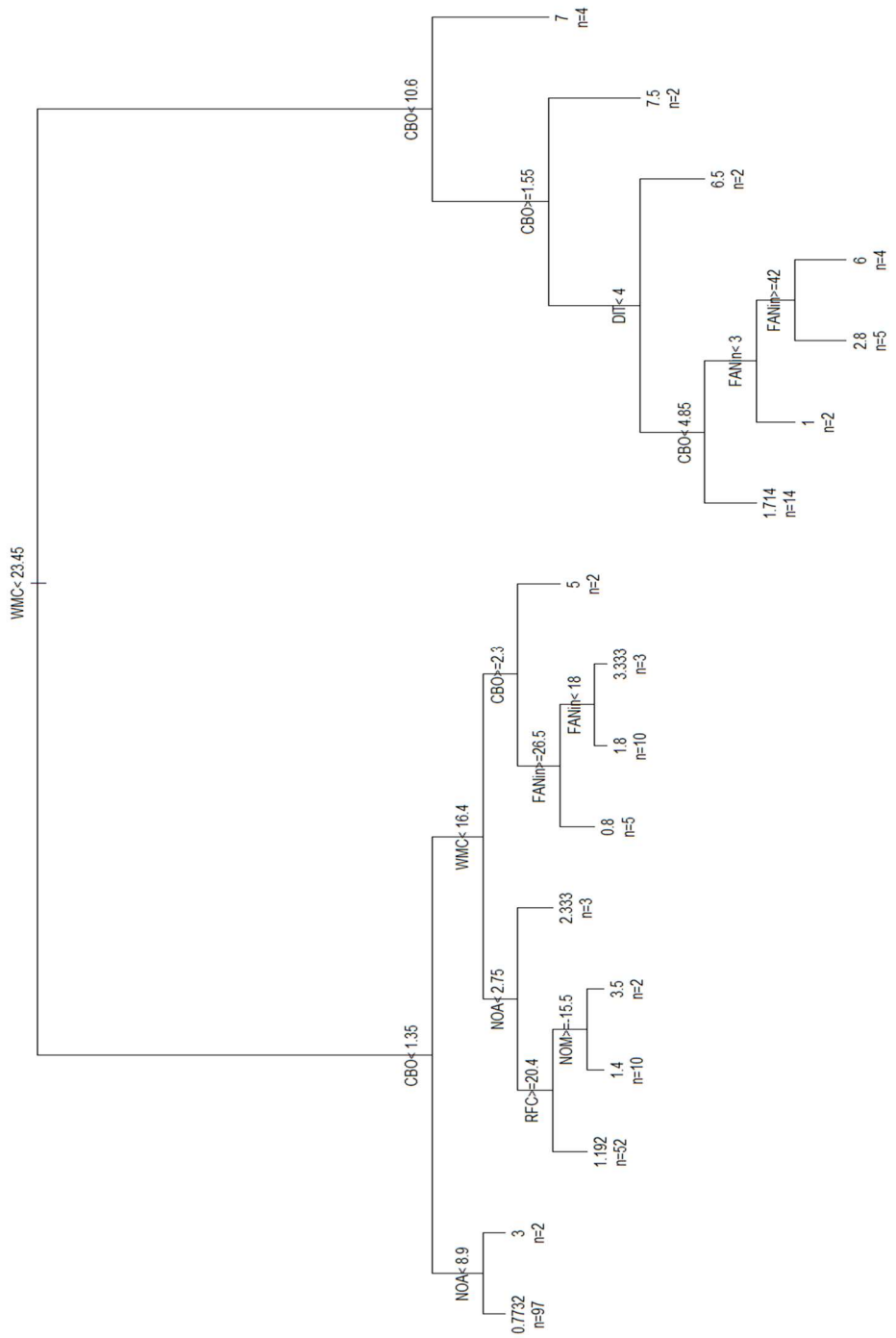


Figure 4.7: Resulting decision tree.

Defect detection state DetState = 41 (Defects) / 46 (EstDefects) = 0.89 or 89% of total real defects in a testing set of classes were found. For this whole project estimated expected quality EQ = 15.37 (KLoC) / 46 (EstDefects) = 0.34. That means this project estimated to have less than 3 bugs per 1000 lines of code which is a high-quality product [37].

As a result of the conducted experiment it can be said, that found knowledge can help with increasing quality of delivered products. All the research questions were answered. Questions 1 and 3 were answered by showing existence of metrics and defects correlation as well as possibility to predict future defect occurrences. Answering question 2, it was shown that metrics do not correlate universally across different projects. The research was based on metrics and prediction assumptions that lead to proving hypothesis of prediction possibility and creating a way of improving software developing and managing. This research's findings are related to other studies by proving possibility of predicting software defectiveness from source code and showing metrics correlation. The findings also show the possibility to predict defects, though also make successful prediction model with acceptable chance of bugs forecasting by using different approach (filtering, weights, Breiman's tree etc.).

The main goal of this research was to build prediction model. Despite the possibility of making a model purely theoretical and hand-written as well as answering first research question, the fact of working with software, open source software repositories and large amounts of data, consisting of thousands of data table entries required to build a software during the research. In this work 4 software units were built. The data that is passing between all modules was handled manually due to lack of automated endpoints in used concomitant software.

The first one is defect data parser. It is a Java program that connects to GitHub public API, retrieves bug information and stores it into csv file. This software solves a trivial task and does not require detailed description and is self-explanatory. Such actions could be also performed manually but would require more time to process.

Second piece of software is Pearson filtration algorithm. It is written in R language and used to apply weights (-1, 0, 1) to all metrics. The program checks all possible combinations of metrics to find which set of values has the highest correlation or reverse correlation coefficient. In the Figure 4.8 method for cycling through weights is shown.

```
incrementWeights = function (mass, inc, shift) {
  tmp <- mass[length(mass) - shift];
  if (isTRUE(all.equal(tmp, 1))) {
    mass[length(mass) - shift] = -1;
    mass <- incrementWeights(mass, inc, shift + 1);
  } else {
    mass[length(mass) - shift] = mass[length(mass) - shift] + inc;
  }
  return (mass);
}
```

Figure 4.8: Method for changing metrics weights.

Another software is used to list through received filtered metrics and breaks up coefficients into smaller ones by a default brute-force approach. It is performed

to specify weight of every metric and find combination with detailed correlation coefficient. The last part of this work's implementation is construction and using a decision tree. The code for this stage is shown in Figure 4.9.

```
library(rpart);
data <- read_delim(datasource, ";",
                   escape_double = FALSE, trim_ws = TRUE);
mass = weightedMass;
datatmp = sweep (inputData, 2, mass, "*");
datatmp$BUGS <- data$BUGS;
dataLearn <- datatmp[2:220,];
dataCheck <- datatmp[221:249,];
file.create("outro.csv");
fit <- rpart(BUGS~., data = dataLearn, method = "anova",
             control=rpart.control(minsplit=5, minbucket=2, cp=0.007));
par(xpd=TRUE);
plot(fit);
predictResult = predict(fit, dataCheck, type = "anova");
setwd(workingDir);
file.create("outro.csv");
write.csv(prepre, "outro.csv");
dataOut <- read_delim(workingDir + "/outro.csv",
                      ",", escape_double = FALSE,
                      trim_ws = TRUE);
paste0(round(100*with(dataCheck, mean(dataCheck$BUGS==dataOut$x))), "%");
```

Figure 4.9: Constructing and using decision tree.

Using "rpart" library for creating and training a Breiman's algorithm made the code implementation relatively small and simple to use and to draw visual presentation of the tree. The tree is using classifier approach and capable of predicting any number of defects in software unit if this amount has already appeared during learning phase.

5 Discussions

The goal of this research is to give the developers and managers during their development process certain approaches, that allow project team to use available means proactively, instead of doing it reactively to make resulting product better in terms of software quality and measurement before the release of the project. The work is mainly focused on exploring the application of the process of static source code mining on a multiple program data, alternating received information, measuring defects as well as testing the possibility of using gathered data to predict future bug occurrences. This research is focused mainly on gaining knowledge and developing a new approach that leads to successful defects prediction.

To gain that observation results this work explained a number of hypotheses about metrics and bugs correlation, defectiveness estimation, that were focused on in a performed experiment, that held data from open source software "Eclipse" including its defects data and its natural language augmentation. All the conducted experiments resulted in the fact, that a machine learning application, based on a decision tree approach is capable of producing meaningful results using static data gathered from open source repository.

The total of 11 metrics was extracted from "Eclipse" project for further manipulations. Also, a list of classes with number of bugs in each class was received from the software. Because of analyzed project being a large software of high quality, number of reported bugs in every class is not very high, which resulted in reduced prediction accuracy. Low amount of data received from natural language processing has the same reason, as well as the fact that studied product was not uploaded to GitHub from the very beginning of its development.

Found metrics were filtered by applying weights and correlation algorithm. R language library allows using three different correlation algorithms. Since this research has no goal to use a particular algorithm, they all were calculated and compared to find the most suitable one. Pearson's had a highest correlation value of 78.83% and was chosen for the method of this research. After this decision tree was built to predict defects from part of data. "Eclipse" project had only several versions, which is not enough to build prediction model within a single class. Because of that, several classes among all data were left for predicting purposes. If the project would be developed further, it will be possible to apply this research method for multiple versions as a future work. Satisfactory results were achieved after following the developed research method, with prediction chance of 89%. This percentage is more than stated minimal 80%, which makes this research successful. As stated in previous chapters, many other studies had more than 90% of predicting chance. Since their approach is not completely different from this research, the major difference is initial data. The lower percentage was received due to having data with less bug amount and versions. Talking about the relation of software unit's size to bug amount, this research showed that they are not related. It was discovered that a most obvious and widespread metric "Lines of code" has little to no correlation and cannot be used for defectiveness estimation. In general, size metrics are of little value for estimating defectiveness.

These are answers to stated research questions:

- Some metrics correlate with defects data
- Correlations of same metrics and defect amount are not equal between different projects

- The possibility to predict bugs was proved, including their location, as a program class, considering stated accuracy that included metrics received from same software version

Because some software metrics could be calculated as complex representation of basic metrics combination, this research was only able to show partly analyzed causes that lead to increase of bugs. This fact rejected the possibility of fully identifying all factors and making precise calculations in this research. For increasing accuracy of calculations and prediction this work could include more detailed metrics of the source code (for example, lines of code changed between versions) as well as bug data, like extracted additional defect's metadata, in conducted experiments. It is also important to include more complex software metrics, for example Halstead complexity measures and McCabe's cyclomatic complexity. Using this extra software metrics this research could analyze source code from different perspective and in more detail. Things like fixed bugs and made code modifications which were the reason of changing in defect density in files and program classes could be also taken into account.

The reliability of this research in terms of being able to construct the same model and predict same defect values in software can be questioned. There are two major problems of the study inconsistency. Fist one is human factor. Natural language processing (NLP) algorithms are working with live data of software repository, meaning that slight change in discussions (for example new comments, editing old comments etc.) may result in having different data from the one used in experiment. To make this change as unimportant as possible, using human language for gathering data is not the main goal. It is only used for enhancing quality of resulting prediction model and sometimes can be totally excluded if the outcome of the decision tree has already high prediction percentage (the exact acceptable percentage might change for different experiments' scopes).

The second reliability issue is using a set of third party software for calculating different steps in this research. They might change with new version. Also, several computing time calculations in this research were done on specific machine running particular set of software in background. Trying to get exact same results would be impossible even using same algorithm.

On the topic of results construct validity, this research contains small amount of developed software, most of which is a trivial task (e.g. retrieving data from public API) and is not the main goal of this work. Other software pieces are not the most optimal or fast, but solve given problem, that is processing data correctly accordingly to used algorithms. Internal validity should be out of question since collected and processed data is not influenced by opinions, apart from issues similar to reliability ones. External validity question is covering the application of experiment to other programming languages, version control systems and projects (second research question) and is broadly discussed in other parts of the thesis. Main problem is the scope of conducted experiments: the programing language is Java; open source software repository is GitHub and changing one of these may result in totally different outcome.

6 Conclusions and Future Work

This research has shown existence of highly related software metrics to corresponding defects. It also described possibility of predicting future bugs by using data extracted from static code of open source repositories. The results are relevant for software development and maintaining fields where they can be used for optimizing workload and decreasing the cost of failure. This research method can be applied to other big projects written in Java programing language. Since it was not tested on other languages it might be possible to apply the method there with certain changes. As it was stated in previous chapter, this research could be done more precise having better initial data, especially more versions of software units. There is another thing that could be changed during research to achieve better results is natural language processing. It is required more complex linguistic setting to improve outcome.

The future work can be done in two different ways: expanding the scope of created method and improving method itself. In the first case research should be done on other programing languages. Experiment should be conducted on other languages besides object-oriented ones, and on other repositories. For the second case it is possible to analyze same metrics in another way, for example combining them using not only simple weights, calculating delta metrics, that are changed in time. It is also important to find the way of analyzing only given software units even with small history of changes and not including other units to predict only within a single software piece change history.

References

[1] The Standish Group, "Chaos," *The Standish Group Report Chaos*, 2014. [Online] Available: https://www.projectsmart.co.uk/white-papers/chaos-report.pdf. [Accessed: Nov. 07, 2018].

[2] J. Vener, J. Sampson, N. Cerpa, and S. Bleistein, "What factors lead to software project failure and whose fault was it," *Ingénierie des systèmes d'information*, vol. 14, no. 4, pp. 55-75, Aug. 2009.

[3] L. D. Dias Junior and E. Favero, "Integrating software repository mining: a decision support centered approach," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 3, no. 6, pp 57-75, Nov. 2012.

[4] S. O. Olatunji, S. U. Idrees, Y. S. Al-Ghamdi, and J. S. Al-Ghamdi, "Mining Software Repositories – A Comparative Analysis," *IJCSNS International Journal of Computer Science and Network Security*, vol. 10, no. 8, pp. 161-174, Aug. 2010.

[5] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300-336, Apr. 2009.

[6] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, Jan. 1997.

[7] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and Validity in Comparative Studies of Software Prediction Models," *IEEE Transactions on Software Engineering*, vol. 31, no. 5, pp. 380–391, May 2005.

[8] J. E. Gaffney, "Estimating the Number of Faults in Code," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 459 –464, Aug. 1984.

[9] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, Jan. 1984.

[10] L. Hatton, "Reexamining the Fault Density-Component Size Connection," *IEEE Software*, vol. 14, no. 2, pp. 89–97, Apr. 1997.

[11] K.-H. Moller and D. Paulish, "An empirical investigation of software fault distribution," in *Proceedings of the First International Software Metrics Symposium, Baltimore, MD, USA, May, 1993.* IEEE, 1993, pp. 82-90.

[12] M. Neil, "Multivariate Assessment of Software Products," *Software Testing, Verification and Reliability*, vol. 1, no. 4, pp. 17–37, Jan. 1992.

[13] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Sep. 2000.

[14] M. Suffian and M. Abdullah, “Establishing a defect prediction model using a combination of product metrics as predictors via Six Sigma methodology,” in *2010 International Symposium on Information Technology, June, 2010, Kuala Lumpur.* IEEE, 2010, pp. 1087 –1092.

[15] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Predicting the location and number of faults in large software systems,” *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340-355, Apr. 2005.

[16] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, July 2000.

[17] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429-445, June 2005.

[18] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees.* Monterey, CA, USA: Wadsworth & Brooks/Cole Advanced Books & Software, 1984.

[19] L. Breiman, “Bagging predictors,” *Machine Learning*, vol. 24, no. 2, pp.123-140, Aug. 1996.

[20] J. H. Friedman, “Stochastic gradient boosting,” *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367-378, Feb. 2002.

[21] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: Data mining, inference, and prediction.* New York, NY, USA: Springer, 2001.

[22] M. D'Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” *Proceedings of the International Conference on Software Engineering*, pp. 31-41, May 2010.

[23] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting Faults from Cached History,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07), June, 2007, Minneapolis, MN, USA.* IEEE, 2007, pp. 489-498.

[24] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting Defects for Eclipse,” in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007), June, 2007, Minneapolis, MN, USA.* IEEE, 2007, pp. 9-9.

[25] T. Therneau, B. Atkinson, and B. Ripley, “Package ‘rpart’,” *rpart*, Feb. 23, 2018. [Online] Available: https://cran.r-project.org/web/packages/rpart/rpart.pdf. [Accessed Nov. 07, 2018].

[26] A. Radhi and C. Bach “Data Mining and Warehousing,” in *ASEE 2014 Zone I Conference, April 3-5, 2014, University of Bridgeport, Bridgpeort, CT, USA.* IEEE, 2014.

[27] S. Chaudhuri and U. Dayal, “An overview of data warehousing and OLAP technology,” *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65-74, Mar. 1997.

[28] Y. Chen, H. Ip, S. Li, and G. Wang, “Discovering hidden knowledge in data classification via multivariate analysis,” *Expert Systems*, vol. 27, no. 2, pp. 90-100, May 2010.

[29] E. Métais, F. Meziane, M. Saraee, V. Sugumaran, and S. Vadera, *Natural language processing and information systems*. Springer, 2016.

[30] D. Graves, J. Noppen, and W. Pedrycz, “Clustering with proximity knowledge and relational knowledge,” *Pattern Recognition*, vol. 45, no. 7, pp. 2633-2644, July 2012.

[31] A. Vesra and R. Rahul, “A Study of Various Static and Dynamic Metrics for Open Source Software,” *International Journal of Computer Applications*, vol. 122, no. 10, pp. 17-21, July 2015.

[32] E. Takimoto and A. Maruoka, “Top-down decision tree learning as information based boosting,” *Theoretical Computer Science*, vol. 292, no. 2, pp. 447-464, Jan. 2003.

[33] C. J van Rijsbergen, *Information Retrieval*. Newton, MA, USA: Butterworth-Heinemann, 1979.

[34] W. L. Neuman, *Social research methods: qualitative and quantitative approaches*. Boston: Pearson, 2014.

[35] JetBrains, “MetricsReloaded,” *MetricsReloaded – Plugins | JetBrains,* Sep. 05, 2016. [Online] Available: https://plugins.jetbrains.com/plugin/93-metricsreloaded. [Accessed Nov. 07, 2017].

[36] H. Barkmann, R. Lincke, and W. Löwe, “Quantitative Evaluation of Software Quality Metrics in Open-Source Projects,” in *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops, May, 2009, Bradford, UK.* IEEE, 2009, pp. 1067-1072.

[37] S. MacConnell, *Software estimation: demystifying the black art*. Redmond, WA, USA: Microsoft, 2006.

[38] G. Carrozza, R. Pietrantuono, and S. Russo, “Defect analysis in mission-critical software systems: a detailed investigation,” *Journal of Software: Evolution and Process*, vol. 27, no. 1, pp. 22–49, Jan. 2015.

A Eclipse JDT Core combined GitHub data (part)

classname	CBO	DIT	FANin	FANout	LCOM	NOC	NOA	LOC	NOM	RFC	WMC	BUGS
org/eclipse/jdt/core,	4	1	4	0	28	0	2	24	8	8	8	1
org/eclipse/jdt/core,	3	2	1	2	120	0	5	123	16	43	31	1
org/eclipse/jdt/core,	8	2	4	4	780	0	49	347	40	105	100	1
org/eclipse/jdt/core,	7	1	0	7	15	0	13	128	6	31	24	1
org/eclipse/jdt/core,	24	2	1	23	120	0	4	190	16	100	63	2
org/eclipse/jdt/core,	55	1	2	53	2080	0	12	1947	65	680	470	1
org/eclipse/jdt/core,	41	1	9	33	210	0	19	426	21	170	89	1
org/eclipse/jdt/core,	1	1	1	1	6	1	1	20	4	5	6	0
org/eclipse/jdt/core,	7	3	1	6	105	0	3	220	15	85	57	1
org/eclipse/jdt/core,	4	2	3	1	3	0	2	14	3	4	4	0
org/eclipse/jdt/core,	35	2	4	34	741	0	7	1106	39	369	331	1
org/eclipse/jdt/core,	16	2	1	15	153	0	2	638	18	220	113	1
org/eclipse/jdt/core,	6	2	3	4	105	0	5	95	15	51	25	0
org/eclipse/jdt/core,	24	1	6	19	325	0	4	281	26	150	74	1
org/eclipse/jdt/core,	6	4	2	4	78	0	4	87	13	46	23	0
org/eclipse/jdt/core,	6	3	1	5	105	0	3	166	15	76	41	1
org/eclipse/jdt/core,	5	3	1	5	105	0	5	96	15	51	25	0
org/eclipse/jdt/core,	22	2	1	22	1653	0	16	952	58	361	271	1
org/eclipse/jdt/core,	4	1	3	1	45	0	276	177	10	26	41	2
org/eclipse/jdt/core,	19	1	11	8	91	0	3	219	14	109	69	1
org/eclipse/jdt/core,	58	1	25	40	2080	0	182	1058	65	413	217	4
org/eclipse/jdt/core,	4	1	1	3	45	0	1	52	10	34	17	0
org/eclipse/jdt/core,	3	1	0	3	91	0	3	142	14	59	38	2
org/eclipse/jdt/core,	1	4	1	0	3	0	2	11	3	3	3	1
org/eclipse/jdt/core,	1	4	1	0	6	0	3	13	4	4	4	1
org/eclipse/jdt/core,	1	4	1	0	0	0	0	3	1	1	1	1
org/eclipse/jdt/core,	2	2	1	1	3	1	0	10	3	5	3	1
org/eclipse/jdt/core,	10	2	9	1	28	1	2	16	8	10	5	1
org/eclipse/jdt/core,	36	2	18	23	190	2	12	1126	20	244	276	1
org/eclipse/jdt/core,	4	4	2	2	6	0	1	16	4	7	6	2
org/eclipse/jdt/core,	36	1	30	7	861	0	54	841	42	294	225	1
org/eclipse/jdt/core,	0	1	0	0	20706	0	0	612	204	204	204	1
org/eclipse/jdt/core,	0	1	0	0	0	0	1	0	0	0	0	1
org/eclipse/jdt/inter	8	8	3	5	21	0	3	115	7	24	34	1
org/eclipse/jdt/inter	1	7	1	0	15	0	1	21	6	10	6	0
org/eclipse/jdt/inter	5	6	1	4	3	0	0	32	3	17	11	0
org/eclipse/jdt/inter	78	3	2	76	21115	0	94	4390	206	###	1148	1

B Eclipse PDE UI combined GitHub data (part)

classname	CBO	DIT	FANin	FANout	LCOM	NOC	NOA	LOC	NOM	RFC	WMC	BUGS
org/eclipse/pde/core/M	4	1	4	0	21	1	6	26	7	7	8	1
org/eclipse/pde/core/p	50	1	46	5	78	0	5	66	13	40	23	1
org/eclipse/pde/interna	2	1	2	2	45	0	3	51	10	20	15	2
org/eclipse/pde/interna	12	2	4	8	28	0	1	36	8	36	10	0
org/eclipse/pde/interna	8	5	0	8	21	0	2	37	7	28	10	1
org/eclipse/pde/interna	26	5	17	9	78	0	4	62	13	35	14	1
org/eclipse/pde/interna	14	1	0	14	45	0	0	134	10	80	30	1
org/eclipse/pde/interna	7	1	1	7	6	0	0	40	4	23	13	1
org/eclipse/pde/interna	14	3	1	13	171	0	3	355	19	273	137	2
org/eclipse/pde/interna	18	1	1	18	55	0	8	230	11	101	56	2
org/eclipse/pde/interna	2	1	2	0	1	0	42	6	2	3	1	1
org/eclipse/pde/interna	11	2	1	10	55	0	9	172	11	106	60	0
org/eclipse/pde/interna	20	1	4	16	253	0	10	347	23	172	93	1
org/eclipse/pde/interna	4	1	0	4	1	0	0	29	2	14	7	1
org/eclipse/pde/interna	8	1	2	6	36	0	1	127	9	73	34	1
org/eclipse/pde/interna	5	4	1	4	78	0	2	51	13	24	16	0
org/eclipse/pde/interna	23	1	3	20	465	0	33	452	31	291	118	1
org/eclipse/pde/interna	7	1	1	6	325	0	19	376	26	191	100	1
org/eclipse/pde/interna	12	1	3	9	10	0	0	79	5	47	29	1
org/eclipse/pde/interna	2	1	1	1	1	0	1	11	2	6	2	1
org/eclipse/pde/interna	7	2	0	7	15	2	1	53	6	37	10	2
org/eclipse/pde/interna	6	1	6	0	0	0	11	0	1	1	0	1
org/eclipse/pde/interna	46	1	9	37	528	2	9	582	33	273	139	6
org/eclipse/pde/interna	39	2	1	38	351	0	3	613	27	362	143	6
org/eclipse/pde/interna	9	3	1	8	3	0	1	29	3	17	8	2
org/eclipse/pde/interna	16	1	3	13	136	0	4	132	17	69	38	2
org/eclipse/pde/interna	10	2	2	8	10	0	1	53	5	31	14	1
org/eclipse/pde/interna	12	6	4	8	120	0	4	106	16	49	25	1
org/eclipse/pde/interna	25	1	0	25	120	0	4	281	16	182	60	1
org/eclipse/pde/interna	24	1	16	9	190	1	11	185	20	96	66	2
org/eclipse/pde/interna	21	1	14	7	10	0	3	68	5	26	16	1
org/eclipse/pde/interna	11	1	2	9	6	0	5	170	4	90	36	5
org/eclipse/pde/interna	153	1	146	17	231	0	24	168	22	80	49	4
org/eclipse/pde/interna	0	1	0	0	0	0	185	0	0	0	0	4
org/eclipse/pde/interna	13	1	4	13	136	1	5	155	17	92	51	1
org/eclipse/pde/interna	30	2	6	26	435	0	6	343	30	220	88	1
org/eclipse/pde/interna	10	5	3	7	78	0	5	144	13	84	38	0