Mining Software Repositories: Using Change History and Process Metrics to Predict Software Bugs

Jesper Holbæk Mark

Kongens Lyngby 2019

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of this thesis is to mine software repositories in order to learn more about the software development process for a given project. By using techniques from machine learning and data mining, the goal is to get a better understanding of the underlying structures in a development process that leads to the introduction of software bugs. The rationale is that changes lead to the introduction of new bugs. To achieve this, a set of process metrics are used to describe the development process. Combinations of these metrics are used to train a mathematical model with historic data. The trained model is then used to predict the defect parts of a software system based on knowledge about the ongoing development process.

The outcome of this thesis is, among others, a software library comprised of a set of functions that is used to classify defective software. The library has been realized with python and can be used across all GitHub-hosted repositories. Moreover, this thesis has studied, how combinations of process metrics perform in terms of prediction power across different repositories.

Summary (Danish)

Målet for dette speciale er at trække data ud af en række software-repositories for at lære mere om softwareudviklingsprocessen i et givent projekt. Ved hjælp af teknikker indenfor maskinlæring og data mining er målet at forstå de strukturer, der i en udviklingsprocess fører til fejl i software. Rationalet er, at tilføjelse af ny kode og rettelser i den eksisterende kildekode introducerer nye fejl i et softwaresystem. Til dette formål benyttes en række veldefinerede procesmetrikker fra literaturen til at beskrive udviklingshistorikken. Kombinationer af disse metrikker kan bruges til at træne en matematisk model med historisk data. Denne model kan herefter bruges til at forudsige hvilke dele af et softwaresystem der er fejlbehæftet baseret på viden om den igangværende udviklingsproces.

Arbejdet med dette speciale har ført til udvklingen af et programbibiliotek, der samler en række funktioner til klassificering af fejlbehftætet software. Biblioteket er udviklet i python og kan anvendes på samtlige GitHub-repositories. I specialet er der ydermere undersøgt, hvordan kombinationer af processmetrikker kan bruges til forudsigelse af fejlbehæftet software på tværs af softwareprojekter og repositories.

Preface

This thesis was carried out at the section for Software and Process Engineering at Technical University of Denmark, DTU Compute in fulfilment of the requirements for acquiring a M.Sc. in Engineering (Computer Science and Engineering). The thesis accounts for 35 ECTS points and has been carried out under the supervision of Ekkart Kindler and Andrea Burattin. The work period lasted from August 2018 to January 2019.

Lyngby, 27 January 2019

Jesper Holbæk Mark

Acknowledgements

I would like to thank my supervisors Ekkart Kindler and Andrea Burattin for the great guidance over the course of the project. Our discussions and your competent feedback have been very valuable for my work. Thank you for being available and for sharing your insights and good advice. I have learned a lot under your supervision.

I would also like express my sincere gratitude to my family. To my mom, sister and family-in-law for their great care and support. To my children for their unstoppable courage and love and most of all to my wife, Trine, for her unconditional encouragement and support.

In memory of my beloved dad, Jan Mark.

Contents

Chapter 1

Introduction

Software engineering in an ever changing world is a challenging task. New requirements arise and existing requirements change while new technologies open doors for unprecedented opportunities. New features are constantly implemented across software projects, and consequently, defects are introduced. Human beings make errors, and even the most experienced craftsmen in any field make mistakes. In software engineering, this phenomenon typically manifests itself in the introduction of *software defects*. These defects present themselves either immediately to the developer at development time or remain hidden in the source code waiting for a specific set of conditions to happen before they will present themselves as business critical software bugs or inconveniences at best. Despite significant effort in the past, it is often a costly affair to test the correctness of software.

Since changes give rise to new bugs, what if we could make use of that knowledge to actually *predict* bugs in source code. This could allow the management of a software development team to make more informed decisions about the resource allocation. The source code that has been predicted as *bug prone* would draw more attention in the forms of testing and code reviewing.

By building on top of state-of-the-art theories and knowledge within the bug prediction field, the work of this thesis has resulted in a realization of a two-component software system that is able to predict bugs across open source repos-

itories hosted on GitHub. In addition to the bug prediction capabilities, this thesis has studied how combinations of specific metrics in the software development process can be used as bug predictors. An experimental setup consisting of: 10 different GitHub-hosted repositories, four different combinations of process metrics, and the capabilities of the implemented software has been established. The results indicate, that while there does not exist one best set of process metrics across all projects, there is not a significant difference between the performance of the combinations.

The thesis is structured as follows.

Chapter 2 - *Problem statement*

Establishes the problem definition and research questions that this thesis will answer. Related work to the thesis is presented as well as the contributions of this work. Lastly the methods and a description of the methods and the experimental setup used to collect the results will follow.

Chapter 3 - *Terms and concepts*

Introduces the different theories and concepts that this thesis is built upon.

Chapter 4 - *Design*

Gives a detailed explanation of how the realization is designed.

Chapter 5 - *Implementation*

Presents the implementation of the tool and gives an in-depth analysis of how the different key phases of the prediction are accounted for.

Chapter 6 - *Results and evaluation*

Presents the evaluation of the prediction performance. Results are shown in tables across releases as well as across repositories and are compared with similar existing approaches.

Chapter 7 - *Perspective*

Puts the thesis in perspective by discussing the road ahead of bug prediction across repositories using process metrics as well as describe unanswered areas of the thesis.

Chapter 8 - *Conclusion*

Concludes by summarizing the problem, method and the key results.

Chapter 2

Problem statement

The overall goal of this project is to use the information available about the development process to predict which parts of a system under development are defective. By combining valuable information that exists in software repositories with proven data mining and machine learning techniques, the aim is to assist development teams when allocating resources to quality assurance. This allows development teams to focus their work on these parts and, thereby, correct more bugs before the software is deployed to production. Different metrics will be extracted and used as variables in a classification process, and the results will eventually be visualized in a clear and understandable manner.

In software engineering, quality[1] is a much desired and essential concept. It is a cornerstone in every software system, and the desire to achieve a high degree of quality is indisputable since, the system is ultimately evaluated by this measure. In software engineering, a common citation is that it costs significantly more to correct a bug later in the software development life cycle compared to earlier stages [3]. A software system, that is not able to *"satisfy the stated or implied needs"*, is tantamount to low quality. The opposite also holds true, that is, a system with a high degree of quality must not deviate from the requirements.

In order to ensure high-quality software, there has been a vast amount of re-

[1] ISO defines quality as *"the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs"* [11]

search in the past, that investigates which factors affect the software quality. This includes, among others, investigating the software development process, the software quality properties[2], and system testing. Moreover, a lot of work has been put in standardization allowing businesses to adopt to international standards e.g. ISO 25010 relating to product quality and ISO 15504 relating to the software process.

Quality management is a process adopted in most industries, and in software engineering, this often refers to the practice of testing a system and reviewing the changes of fellow developers. However, testing is expensive and error-prone itself, and a high *code coverage* is not necessarily equivalent to less bugs [28]. Additionally, code reviewing is a costly process in terms of wages and, due to limited resources in every businesses, covering all modules and parts of a complex software system with test cases might not be feasible. Therefore, it could be helpful to inform developers about which parts of a software system that would need more quality assurance.

One of the fields, that has been given substantial attention in research, is *bug prediction* [20], where the overall goal is to guide the developers to parts of a system which are subject to more bugs. This facilitates better management of resources, so that test and review efforts can be directed to more error prone areas of a system. Many prediction models have been proposed, and while many of them report promising results on the test data, it seems that the prediction power decreases to a high extent when the classification models are transferred into other contexts [20]. In addition to the transferability difficulties, it has also been reported that, a single set of metrics, which yields the highest prediction power for all software, exists [35].

Over the last decade, researchers have increasingly looked into software repositories as a source of valuable information about the software development process and its artifacts [25]. As the field of *data mining* has grown into a mature and proven discipline, subfields, like Mining Software Repositories (MSR), have emerged. The goal of MSR is to *"analyze the rich data available in software repositories to uncover interesting and actionable information on software systems and projects"* [37]. By using the rich data about the software development process extracted from a software repository within domain of machine learning, namely data classification, it is possible to build bug prediction software that have shown promising research results, especially on the Eclipse JDT project[3] [33] [34].

This thesis will investigate how knowledge about the development process can

[2] or the so-called "ilities" [46]

[3] `https://www.eclipse.org/jdt/`

be extracted from a software repository and used to predict defective parts of a software system. By exploiting data about the software development process, this thesis will use techniques and theories from data mining and the MSR field to implement bug prediction software, which is able to predict which files that are defective. Furthermore, this thesis will investigate how well a model trained with using data from one repository is performing when transferred to another repository. By using data from open source systems (OSS) available on GitHub, the objective is also to realize a general prediction tool that performs satisfyingly well across multiple independent repositories.

The thesis will answer the following research questions:

RQ1 *How powerful are combinations of process metrics when transferred across different open source repositories?*

RQ2 *How do combinations of process metrics that have proven powerful on the Eclipse project transfer to different open source repositories?*

The motivation behind *RQ1* is, to get a better understanding of the development process across software projects. How do process metrics, that are strong predictors of bugs in one project predict the bugs in other software projects with possibly a quite different development process? The next research question, *RQ2* seeks to investigate, how the combinations of process metrics from previous studies, that also have yielded good results in the Eclipse project do, when they are used to predict bugs in open source projects. Do the current process metrics discovered in the literature work as good predictors for multiple projects?

2.1 Related work

MSR has received increasing attention over the last decade and a yearly MSR conference has been held since 2004 [37]. The MSR field seeks to achieve multiple different goals, with some of them overlapping others. These goals include but are not limited to [25]:

- Supporting software maintenance
- Improving the software development process
- Validating new ideas in software engineering

- Predicting defects or inconsistencies

Bug prediction is arguably one of the most investigated areas of MSR, and it has been studied extensively with the rising attention given to MSR [20]. By utilizing supervised learning techniques from data mining, multiple bug prediction approaches have been proposed [9] [27] [38] [5] [49].

Ohlsson et al. reported a study where metrics that was derived from design documents could be used to predict fault-prone modules [36].

In an early study, Basili et. al concluded that object oriented (OO) metrics work as predictors for software bugs [2]. This was later confirmed by [44]. Zimmerman et al. studied the Eclipse project and found a considerable correlation between complexity metrics and post-release defects. In addition, they contributed with a publicly available data set of the Eclipse project with files mapped with defects and complexity metrics [49]. Naggapan et. al also found, that complexity metrics correlate with post-release defects, and that there exist a set of metrics which can predict post-release bugs *within* a project. Likewise, they partially confirmed that the combined metrics found within one project can predict bugs in other projects, but this is only limited to similar projects. Furthermore, they rejected the hypothesis that one single set of metrics which predict bugs in all software projects exists [35]. Menzies et al. also found that there exist no best set of metrics for complexity metrics. [31]

Hassan [23] found, that *"the more complex changes to a file, the higher the chance the file will contain faults."* He also showed, that prior faults are better predictors compared to prior changes. Yu et al. showed, that previous bugs work well as predictors for future bugs [47].

In 2000, Graves et al. predicted defects using process metrics, and argued that process metrics posses better prediction power compared to product metrics [18]. Moser et. al [33] showed that process metrics outperform product related metrics (including complexity metrics) and a combination of product and process metrics in the Eclipse project. They used naïve Bayes, logistic regression and decision trees to conduct their experiments. Zimmerman et al. used a combination of process and product metrics to investigate the power of cross-project bug prediction by building models from one project and transferring them to another project. The concluded, that classification models cannot be used across software projects [48]. Muthukumaran et al. proposed new process metrics related to the distribution of change over a time period and experimented with bug prediction on the Eclipse project using data from GitHub and reported promising results. In their experiment, they used naïve Bayes, logistic regression, decision trees, and the hybrid näive bayes trees [34].

2.2 Methodology

This thesis is based on empirical software engineering, where a software system is implemented to set up an experiment. The experiment will be conducted to collect empirical data about the performance of process metrics, and this data will be used to answer the research questions of the thesis. To carry out this experiment, a software system that is able to extract, preprocess and analyze repository data must be built. The overall functionality of the system will be based on the work of Muthukumaran et al. [34], who also built a tool to investigate process metrics extracted from Eclipse project hosted on GitHub. Their methods are partly reproduced in this thesis, but also extended in the sense, that the implementation will be able to mine data from all GitHub repositories.

The experimental part of the thesis will be carried out by selecting a number of repositories that satisfy a defined set of inclusion and exclusion criteria. Each of the included repositories are mined, and the performance of multiple combinations of process metrics is evaluated across the repositories. The evaluation is based on assessing the classification errors which are reported with a set of widely used evaluation metrics.

2.3 Contribution

The main contributions of this thesis are threefold:

- extending the work of Muthukumaranet et al. [34] by proposing a bug prediction library supporting all GitHub-hosted repositories.
- studying how the combination of process metrics are transferred across open source repositories.
- studying how combinations of process metrics that have proven considerable results on the Eclipse project perform when they are used to predict bugs in open source projects.

Chapter 3

Concepts and terms

This chapter introduces the key concepts and terms which are used throughout the thesis. First, a general explanation of the software development process is conducted followed by an introduction to the MSR field. Subsequently, a general explanation of the data mining process is provided, and concepts related to MSR are elaborated on. In MSR, and data mining in general, several terms are used interchangeably to describe the same thing. In data mining literature, one single standard terminology has not been adopted, and in order to avoid a mix-up and confusion of terms, this chapter establishes a terminology which is consistently used throughout this thesis. It is noteworthy, that the introduced terms do not stem from one specific text book. Along with this chapter, a glossary is found on page 101.

3.1 Software development process

Software engineers have over the history adopted different development processes to increase productivity and efficiency. A software development process is divided into a number of phases that covers different aspects of the realization of a software system. Historically, two very distinct processes have been used; namely, the sequential approach, which dominated the 1970's through the

1990's, referred to as the waterfall model, and the iterative approach, referred to as agile, which is more or less the de facto standard today. In sequential software development, a project is divided into phases which are performed sequentially with only slightly overlapping phases, hence, it can be perceived as a waterfall. A typical realization of the waterfall model would start with the analysis of a problem followed by the definition of a formal requirements specification. Subsequently, a complete design which would satisfy every defined requirement would be made and, lastly, the design would be completely implemented and only then ultimately tested by the end user. It is outside the scope scope of this thesis to cover all advantages and disadvantages of each approach, but the waterfall model is obviously flawed, when considering, that most things in this world are anything but constant. The lack of flexibility upon the change of requirements that the waterfall model imposes, has led to severe delays and over-budget software projects over time. During the 1990's, the iterative approach saw increasing usage through the frameworks of Scrum, extreme programming (XP), and rational unified process (RUP); ultimately collected as principles in the *Agile manifesto*. Iterative software development allows for a high level of agility through continuous delivery and a close customer interaction. The iterative approach has been highly adopted by most commercial software companies, and while many elements from the agile method are recurring, the overall development process can be seen in many variations across software projects.

When applying the principles of software engineering to fulfill a software related task, that being analysis, design, implementation, or maintenance, it is typically related to some kind of a project. Most software projects are carried out in one or more teams of people that collaborate to reach a shared goal. An example of a typical agile-based software development team working on a specific product or module would consist of a number of junior and senior developers, an architect and a team lead. This team would try to conform with a defined set of development principles. To give an example, this could among other things state that the team should *release* a new version of their product approximately every other week as well as commit their code changes to the version control system as early as possible.

The latter in the example is a generally adopted rule across projects, as development teams with multiple contributors often work on the same parts of the system and thus each developer often rely on the changes of their fellow developers. As in any other software project, this team also introduces new bugs in their source code as a result of their changes, and to this end, the team have a bug tracking system in which new bugs are reported. To maintain traceability in the revision history, every change and commit that is related to the task of fixing a bug is tagged with the identifier of the respective bug. The concept of releases and the reporting of new bugs will be introduced in more details in Section 3.7.

The general practices mentioned in this example are also widely adopted across many software projects of varying size in proprietary closed-source projects as well as in open-source software (OSS) projects. However, as software engineering is typically performed in teams, the characteristics of the team: e.g. the size and the group dynamics greatly influence how the development process in reality is carried out. Additionally, OSS projects tend to be even more diverse in their development processes due to their very different organizational structure compared to more traditional corporate development teams. While some OSS projects have been founded by a single developer, who have implemented most of the product and then scaled it up with the assistance of multiple contributors[1], other projects are much more distributed in terms of contributions[2].

3.2 Data mining terminology

"Data mining is the study of collecting, cleaning, processing, analyzing, and gaining useful insights from data" [1]. In this thesis, data mining techniques will be used to recognize patterns in the development process that lead to defective software. To lay out a theoretical foundation for the remainder of the thesis, multiple concepts and terms from the data mining field are introduced. To put the terms into the setting of bug prediction, a simple running example will be provided.

3.2.1 Data set

In data mining, a data set is a collection of data that is typically represented as a table or a matrix with columns and rows. A record corresponds to one single row in the data set, where each record contains a set of fields. Throughout this chapter, a running example from the software development domain is established. Table 3.1 shows an example of a multidimensional data set containing data about files from a software repository that have been changed over time.

Filename	Commits	Developers	Additions	Deletions
HelloWorld.js	2	1	8	3
Foo.cs	7	1	15	13
Bar.js	12	4	168	57

Table 3.1: Data set A. Example of a simple multidimensional data set containing 5 columns and 3 records

[1] Like vue.js (`https://github.com/vuejs/vue`)
[2] Like the Wordpress Calypso project (`https://github.com/Automattic/wp-calypso`)

Each record in the data set corresponds to a file, and is identified by the column *filename*. Each record does also contain additional fields that describe the characteristics of the record. It is seen that in this data set, the file *HelloWorld.js* has had two commits by one developer. In total, there has been a total of 8 added lines of code (LOC) and 3 deleted LOC to the file.

3.2.2 Feature

A feature is a specific field in a record which is used to describe the particular record. A feature is in other words a characteristic of the record. In the provided example, the columns *commits*, *developers*, *additions*, and *deletions* are features in the data set.

3.2.3 Label

A class label describes the class to which a specific record belongs. The class is the *ground truth* of the record. Reusing the running example, the data set can be extended to also contain a column which is considered the class label of each record. In the context of MSR and bug prediction, the column *defective* is introduced, which tells if a file is defective e.g. has one or more software bugs in it. This is seen in Table 3.2

Filename	Commits	Developers	Additions	Deletions	Defective
HelloWorld.js	2	1	8	3	No
Foo.cs	7	1	15	13	No
Bar.js	12	4	168	57	Yes

Table 3.2: Data set A. The column *defective* does in fact tell whether a file contains one or more bugs or not

The data set in this example has a label that is binary nominal. The label is binary, as it belongs to a set of two classes namely *No* or *Yes*. The label is nominal since it does not have a quantitative value and it cannot be ordered. On the other hand, the example could also include a multi-class ordinal label which, as the name suggests, would belong to a set of multiple classes. An example in this context could be the labels: *Not defective*, *Somewhat defective* and *Very defective*, indicating that *Somewhat defective* files would hold only a few bugs, while *Very defective* files would hold many bugs. These classes are ordinal, as the order of the classes is significant, that is, *Very defective* is perceived worse than *Somewhat defective*, which again is worse than *Not defective*.

3.2.4 Data classification

Data classification is a technique used to categorize data into classes. The categorization is conducted by learning from a data set, where the records are already organized in existing classes identified by the class labels. By learning from existing examples, a model can be trained and then be used to estimate class labels of unlabelled data. The data that is already categorized by class labels are referred to as the training data, while the unseen data is referred to as the test data. To actually build the model, a classification algorithm is used [1]. When a model is built and trained, test data is used as input to the model, and the output of the model is an estimation of the class labels of the test data i.e. a prediction. Using the data set from the example as training data, a model can be trained and built to classify the label of an unlabelled data set i.e. test data. An example of an unlabelled data set is seen in Table 3.3. Using the model trained with the training data, the prediction of the classification model is seen in Table 3.4

Filename	Commits	Developers	Additions	Deletions
Baz.ts	1	1	3	1
Qux.cs	16	3	381	182

Table 3.3: Data set B. Example of an unlabelled data set used as test data

Filename	Prediction (defective)
Baz.ts	No
Qux.cs	Yes

Table 3.4: The predicted label, *defective* of each record identified by the name of the file

3.2.5 Feature selection

In general, data mining algorithms tend to be ineffective when working with too many features. Some features may be irrelevant for the classification problem, and, thus, add noise to the model. Some features may be highly or completely correlated with other features i.e. redundant, and will, therefore, not bring any prediction power to the model itself. To this end, it is crucial to select a relevant subset of features to be used in the training of a model. This is referred to as *feature selection*, and is the process of evaluating each feature, followed by choosing only the most relevant features and, thereby, reduce the dimensionality of the problem. It is important, however, to mention that, although a single

feature may not be relevant by itself, it can be transformed or used to construct new features that bring more relevant information.

3.3 The data mining process

Due to the rising amount of data available and the increasing computing power, data mining has proven to be a very useful process to extract knowledge. The domains, in which data mining can be applied are manifold, and finance, biology, health care et cetera are areas and industries where data mining is commonly used. Thus, the data sources can be quite heterogeneous with distinct data types, formats and missing data. To this end, a general data processing pipeline is used among data scientists and analysts. The overall process is the same across all problem domains. The pipeline and its varieties include different phases which are divided into: a collection phase, a preprocessing phase, and an analytical phase. An outline of such process can be seen in Figure 3.1.

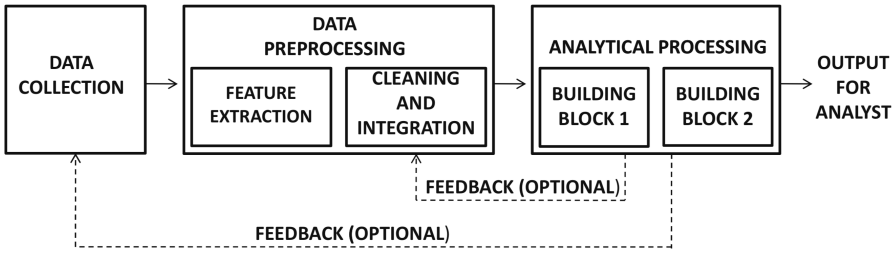


Figure 3.1: The data processing pipeline. Reprinted by permission from Springer Nature [1].

The data collection phase is about extracting and collecting the data to be analyzed. The data can be collected from multiple sources like specific types of hardware, physical documents or external systems, and the goal of the data collection phase is to store the data into a single source e.g. a database, a data warehouse or similar. A common idiom in machine learning and data mining is *"garbage in, garbage out"*, which means that input of low quality or flawed input yields a similarly bad output. To this end, it is crucial to ensure that the quality of the extracted data is as high as possible.

The next phase is the data preprocessing phase. The ultimate goal of this phase is to generate a structured data set that can be used for further analysis. The most crucial step in the preprocessing phase is the feature extraction. When

having a data set, feature extraction is used to find the most important features that can describe the data set. Feature extraction is highly dependent on the domain and application of use.

The collected raw data, which was a result of the data collection phase, can have multiple flaws, which may result in poor or simply misleading results in the analytical phase. The raw data can be perceived as dirty when it is incomplete, inaccurate, or inconsistent [6]. Data cleaning is used in the preprocessing phase to handle the erroneous data by removing or correcting some specific entries. When the above steps have been completed, the result should be a structured set of features, that describes the complete data set.

The analytical phase is where the actual analytical algorithms are applied to the preprocessed data set. *Aggarwal* [1] lists four problems, that are considered fundamental and repeatedly seen in the context of data mining. These are: data clustering, classification, association pattern mining, and outlier detection. Bug prediction is by its nature a very suitable problem for the classification technique, as classification does indeed use historical data for estimating the class of a record.

Since the primary technique of this thesis is classification, it is outside the scope of this thesis to introduce the three other techniques mentioned.

Figure 3.2 outlines the internal relationship between the introduced terms. In this thesis, two types of classifiers will be used, namely: probabilistic classifiers and decision trees.

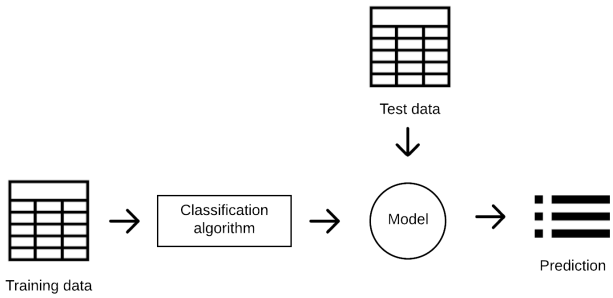


Figure 3.2: Training a classification model

3.4 Classification algorithms

As previously described in Subsection 3.2.4, in order for a classification algorithm to classify records, it has to learn from training data i.e. a data set that has already been labeled. By using the training data to build the underlying mathematical model, a classifier will use the structure of the features of the training data to determine the label of, hitherto, unseen test data [1]. A more detailed description of how a classifier is validated follows in Section 3.5. In order to build a classification model, a classification algorithm has to be used. The most commonly used algorithm types are

- Probabilistic classifiers (Naïve Bayes, logistic regression)
- Decision trees
- Random forests
- Neural networks
- k-nearest neighbor
- Support vector machines

The bug prediction problem can both be defined as a binary and a multi-class classification problem, but to keep the problem domain as simple as possible, while still being able to answer the research questions, the bug prediction problem in this thesis is defined as binary i.e. whether a file is defective or not. To this end, all classification algorithms, that are able to solve binary class classification problems can be used. However, in order to be consistent with similar researches and to be able to compare the results, the realization of the bug prediction tool will only make use of a subset of these algorithms, namely: Naïve Bayes, logistic regression, decision trees and random forests [33] [34]. As this work does not go into the implementation details of the different algorithms, nor propose any novel modifications to the algorithms, existing implementations of the algorithms are merely used as black boxes configured by a number of parameters. The following sections describe each of the algorithms used in this thesis at a high level.

3.4.1 Probabilistic classifiers

Probabilistic classifiers are a group of classifiers that use the feature variables to model the probability of a label [1]. The two most commonly used probabilistic classifiers are naïve Bayes and logistic regression.

Naïve Bayes:

The Naïve Bayes classifier is a Bayesian classifier that is based on Bayes' theorem [21] which is defined as *"the ratio between the value at which anexpectation depending on the happening of the event ought to be computed, and the value of the thing expected upon its happening"* [22]. Simply put, the theorem is *"a simple mathematical formula used for calculating conditional probabilities"* [24]. In data mining, the theorem is used to model the probability of the class label using the features of a record [1]. Using Bayes' theorem for a classification problem on data sets of a substantial size would be extremely computationally expensive, and to this end, the theorem is used together with the naïve assumption; that all features are independent. This is referred to as naïve due to the fact that second to no features are in reality completely independent. However, the simple Naïve Bayes classifier has proven very strong and its performance has been comparable with even more complex classification algorithms [21].

Logistic regression:

Logistic regression is another probabilistic classifier that, instead of using Bayes' theorem to map the features with the label probability, builds upon linear regression. Linear regression uses a linear function to estimate continuous output. If the bug prediction problem was changed to a regression problem the prediction of a file would not be binary, but rather continuous e.g. an estimate of how many bugs a file would have. Using linear regression, a relationship between the number of commits and the number of bugs could be modelled. By using a linear function, linear regression would output the number of bugs for any given input number of commits. Logistic regression does instead use a logistic function to map the features with the label. Instead of outputting a continuous value, logistic regression instead uses the probability to estimate the binary label of a record.

3.4.2 Decision trees

A decision tree is another classification model, that does not model the probability of a label, but rather model a *"set of hierarchical decisions on the feature variables, arranged in a tree-like structure"* [1]. The nodes in the trees, referred to as split criteria, indicate a decision on a specific feature variable. Reusing the example data sets introduced in Subsection 3.2.1 and Subsection 3.2.4, and for simplicity, considering only the features *Commits* and *Additions*, a decision tree can be trained with training set A containing the ground truth. Predicting the label of data set B, a decision tree is constructed, as seen in Figure 3.3

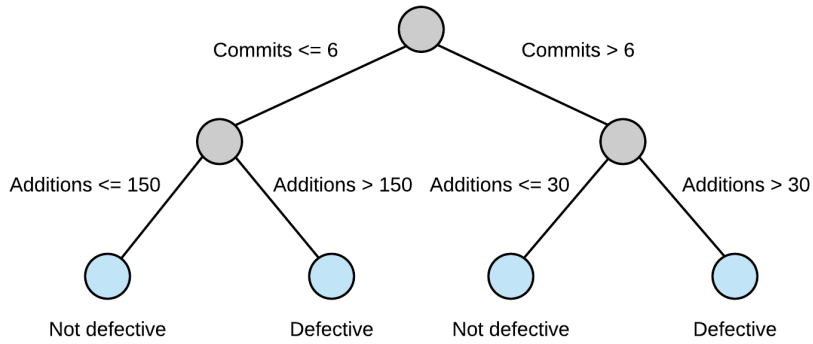


Figure 3.3: Example decision tree trained with data set A used for predicting the labels of data set B

The random forest algorithm builds upon the ensemble method which is *"an approach to increase the prediction accuracy by combining the results from multiple classifiers."* [1]. The concept of the random forest is to build and combine multiple decision tree models and create one composite model. The output of each decision tree classifier is considered as a vote, and the class label with the majority of the votes is the output of the random forest algorithm.

3.5 Classifier evaluation

To validate if a classification model actually goes beyond blind guessing and to also evaluate how it performs compared to other classification models, different validation techniques and evaluation metrics are used in data mining. Multiple validation techniques exist; including hold-out, cross-validation, and bootstrapping.

3.5.1 Evaluation metrics

In binary classification, a classifier can either classify a record as negative (0) or as positive (1). In bug prediction this corresponds to classifying a file as not defective (0) and defective (1). In reality, the resulting classification of a classifier

belongs to one of four possible categories. If the classifier correctly classified a specific record as positive this is referred to as *true positive*, on the other hand, if a classifier classified a record as positive, while it in fact are negative, this is referred to *false negative*. The four possible outcomes of a classification are as follows:

- True positive (TP): Records that are correctly classified as positive
- False positive (FP): Records that are incorrectly classified as positive
- True negative (TN): Records that are correctly classified as negative
- False negative (FN): Records that are incorrectly classified as negative

A basic evaluation metric is *accuracy*, which is a measure of the degree to which the predictions match the reality [41]. However, this metric does not necessarily provide a fair and complete picture of the classification performance on data sets that are considerably class-imbalanced [4]. Consider a software project, where a fictive number of 10 % of all files are defective, a data set of 500 files would consist of 450 negatives (not defective) and only 50 positives (defective). If a very pessimistic classifier classified all of the files in data set as negatives (not defective), the accuracy of the classifier would still be quite high, although all the defective files would stay under the radar, which cannot be considered a success. Since the data sets used in the domain of bug prediction are most likely not class balanced, accuracy is not powerful enough to evaluate the performance. To this end, a confusion matrix comprised of TP, FP, TN and FN can be used to calculate more complete evaluation metrics. The structure of a confusion matrix is seen in Table 3.5

		Predicted	
		0	1
Actual	0	TN	FP
	1	FN	TP

Table 3.5: Confusion matrix in binary classification

Instead of measuring how accurate the classifier is on a class-imbalanced data set, it would be more useful to understand the classifier's ability to classify the positive records within a data set. This evaluation metric is referred to as recall

and is defined in Equation 3.1.

$$Recall = \frac{TP}{TP + FN} \tag{3.1}$$

However, recall cannot be used to describe the classifier performance alone, since if the classifier were to classify every record as positive, the recall would be 1.0. A recall of 1.0 would, for instance, mean that all files in a software repository would be classified as defective. To this end, the evaluation metric, precision is used to describe the classifier's ability to classify only the true positives as positive [29]. Precision is defined in Equation 3.2.

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

Reusing the example where a classifier would classify every file as defective, the recall would be perfect, while the precision would be extremely low. On the other hand, if the classifier were to only classify one file as defective, which in fact is truly defective, the precision would be perfect as no false positives would exist. In that case, the recall would be extremely low.

This pair of evaluation metrics is useful for measuring the performance of a classifier on a class-imbalanced data set as they are complementary to each others weaknesses in their expressive power. These metrics can also be combined into one metric referred to as F1 defined in Equation 3.3.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3.3}$$

It is possible to assign different weights to precision and recall, depending on which metric is considered more valuable in a specific context. F1 is balanced and is, therefore, the harmonic mean of recall and precision. When evaluating the performance of the process metrics used with the different classifiers in this thesis, F1 is the score that determines which set of metrics performs better.

3.5.2 Validation techniques

In the hold-out technique, a data set is split into two sets: one with training data and one with test data respectively. The training data is used for training

the model, while the test data is unknown to the model [42]. To conduct the evaluation, the model predicts the labels of the test data and then the predictions are compared with the ground truth. A major drawback of the hold-out method is, however, the fact that the evaluation is heavily affected by the way the data set is split. On highly imbalanced data sets, a test set with the size of 25 % of the complete data set, may contain no or only a few records belonging to one class. This clearly affects the evaluation and the result is that the performance estimate generally has a high variance [1]. Redoing the method with a different split may yield quite different results.

k-fold cross-validation builds upon the hold-out method, but instead of splitting the data set into two disjoint sets, the data set is split into k different subsets. The hold-out technique is then repeated k times using one of the k subsets as the test set and the other k-1 subsets joint together as the training set. The average result of all tests are then reported as the result of the k-fold cross validation. The overall concept of this technique is depicted in Figure 3.4

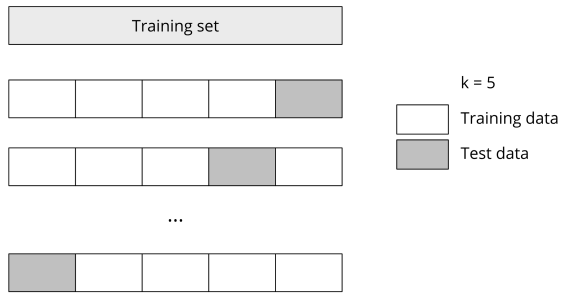


Figure 3.4: k-fold cross validation

The main advantage of using k-fold cross validation is the lower variance of the performance estimate which is achieved by repeating the hold-method on multiple splits. A disadvantage on large data sets is, that the computation is significantly more expensive, given that the model has to be trained again on each k run.

To answer the research questions of this thesis, it has been considered satisfactory to use k-fold cross-validation to obtain the recall, precision and F1 scores, and specifically use the F1 to compare the performance across the process metrics combinations and classification algorithms. As the absolute scores of the classifiers are not overly important in this case, but rather comparing the relative performance, it has been deemed out of scope to go into detail about the recall/precision trade-off and for instance consider ROC/AUC curves on every

data set.

3.6 Metrics

In the context of bug prediction, two classes of metrics have traditionally been used as features, namely *product metrics* and *process metrics*.

Name	Description
Commits	Sum of commits to the file
Deletions	Sum of deleted lines of code
Additions	Sum of added lines of code
Authors	Number of different contributors to the file
Commits pre release	Number of commits just before a release. Can be defined by a fixed number n days before the release date.
Last commit	Days since the last commit to the file
Refactorings	The number of times a file has been refactored (checking if commit message contains refactor [33]
In development bugs	Number of bugs reported to the file in the development phase [33]
Entropy	The distribution of changes in the timeline [34]
Mean period of change	Estimation of center of concentration of changes [34]
Maximum burst	Sequence of consecutive changes using parameters gap size and burst size [35]
Maxmimum change set	Maximum number of other files changed together with the file [33]
Average change set	Average number of files changed together with the file [33]
Age	Total age in days of the file [33]
Weighted age	Weighted age [33]
Maximum additions	Max number of added lines to the file
Average additions	Average number of added lines to the file
Max deletions	Maximum number of deletions to the file
Average deletions	Average number of deletions to the file
Code churn	Sum of added lines minus deleted lines to the file
Maximum code churn	Maximum code churn to the file
Average code churn	Average code churn to the file
Average time between changes	The average number of days that have passed between the changes of the file
No commit messages	Number of commits without a commit message

Table 3.6: Process metrics

Process metrics do not describe the actual software, but rather the underlying development process, i.e. the process that leads to the product. Based on historical data, process metrics describe the *changes* that have been made to a software system. This include the additions and deletions of code [12], how many different developers that have made the changes, and *when* they were made. It does also include the number of previous bug fixes [33].

Table 3.6 outlines different process metrics that has been used in previous studies to predict bugs. These metrics are thereby the foundation for the research questions in this this work.

Product metrics are, as the name indicates, about the state of the product, i.e. the software system. Metrics in this class describe properties of the actual software, which include, but are not limited to: LOC, OO metrics [19], usage of design patterns (and antipatterns [45]), complexity, and the structure of the code. There are several other product metrics used in literature, but as this thesis focuses on process metrics, it is outside the scope of this thesis to present them in details.

3.7 Releases and bug reports

Now that the technical foundation for bug prediction has been introduced, it is time to investigate, how releases and bug reports of a software project can be used to construct a data set suitable for bug prediction.

In software engineering, a release is a specific version of a software system which is published to its end users. A release is therefore comprised of a state of the software system as well as a release date. As software naturally matures, the development process that leads to bugs in the early stage of a project's life cycle may be significantly different from the process of a system which has been refined for years. To this end, releases fit very well into the construction of a data set in the sense that developers make changes to software for a planned release in order to jointly publish these changes at the same time. By reusing these release dates as delimiters, multiple bug prediction data sets within the life cycle of a software project can be created.

Most software is tagged by a version number, that is, a *version number* is assigned to a unique state of a software system. One of the most widespread versioning schemes adopted are *The Semantic Versioning specification* which defines a version number as:

"A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers, and MUST NOT contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: 1.9.0 → 1.10.0 → 1.11.0" [39].

Different projects do however, vary in the scope of their major, minor and, patch versions. Some projects tend to release a lot of changes in a minor version or patch version, while other projects will release the same of amount of changes in a major version. It is, therefore, important to note that across repositories, the releases of both major and minor and patch versions can be picked as the delimiter of a data set. The reason is, that both the number of changes and the absolute time between major and minor and patch versions vary greatly due to project policies and requirements.

Recall the labelled data set A in Table 3.2 on page 12 which consists of both features and a label for each file. In this thesis, a data set is created from the changes that have occurred in the time span *between two consecutive releases*. In order to construct such a data set, it is simple to extract the process metrics of a file by using the change history between the two releases. It is, however, not that simple to extract if the file indeed does contain bugs or not. To this end, the bug reports, that have been reported *after* a release and, most importantly, the *bug fixes* that have been made after a release, are used to decide whether a file or not is defective.

The bug prediction data set will, therefore, be constructed by extracting features *pre-release* while the labelling of each file is done by extracting bug reports and bug fixes *post-release*. Given the release dates R_1 and R_2 such that $R_1 < R_2$, the *pre-release* period will in this thesis be defined as $T_{pre} = \Delta R$. Given a *post-release bug fix period*, $Y > 0$, the *post-release* period will be defined as $T_{post} = Y$. The *post-release bug fix period* is the maximum number of days that will have passed after the release, before bugs reported post-release will have been fixed.

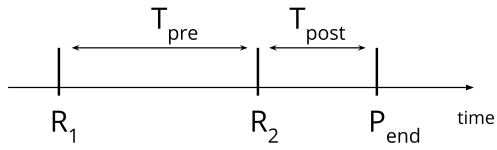


Figure 3.5: A prediction timeline constituted by two release dates R_1 and R_2 and a post release bug fix period which ends at P_{end}.

Figure 3.5 shows how a pre-release period T_{pre} and a post-release period T_{post}

constitutes the timeline used in the bug prediction classification problem. The details on how this extraction is carried out are decribed in Chapter 5.

3.8 Git and GitHub

Software consists of source code, which is text files containing human-readable sequences of executable code. When software developers work together on the same source code i.e. the same text files, it is essential to have proper a version control system (VCS) to contain the history of the changes but also allow for collaborative changes without the risk of losing some contributions. A VCS usually consists of a central repository, to which developers submit their modifications. Git is a distributed VCS, where each developer has their own local version of the repository. While it is not strictly necessary to have a central repository (hence it is distributed), most of today's workflows have exactly that, where the developers *merge* the changes from the local repository to a central, remote repository. This is seen in Figure 3.6

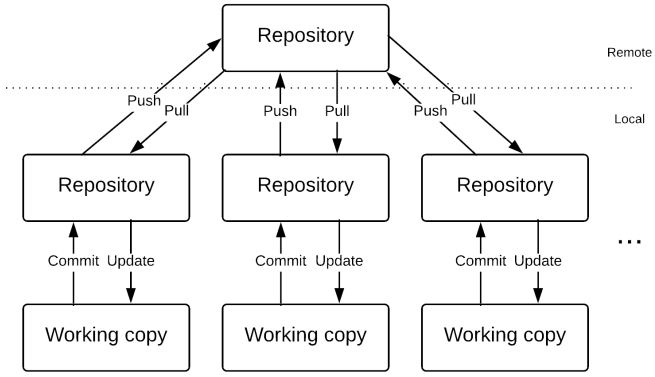


Figure 3.6: A distributed VCS with a central remote repository. Source: [10]

With more than 85 million repositories[3] [13], GitHub is the world's largest online version control hosting service using Git. Beside the obvious hosting of source code, GitHub also provides a large number of important features, including: issue tracking, pull request, graphs, and social-network like features. The wide adoption of GitHub among companies of sizes varying from small startups to

[3]Reported by April 2018.

Facebook, open source communities and academia, bring a lot of possibilities of extracting valuable information about the software development process.

The most central element in git and GitHub is a commit. For this project, it is less relevant *how* a commit is created in git, but rather *what* it tells about the change. Simply put; a commit contains information about the modified files on the local repository which can be merged and pushed to the remote repository.

Additionally, GitHub does through each repository provide a built-in issue tracker to manage bugs and any other issues in a software project. The structure of a GitHub issue is very simple as seen in Figure 3.7. GitHub issues have a very limited structure with only text field where the description of the issue is provided. Issues can be labelled using repository-defined labels, as well as, being attached to repository-specific projects and milestones.

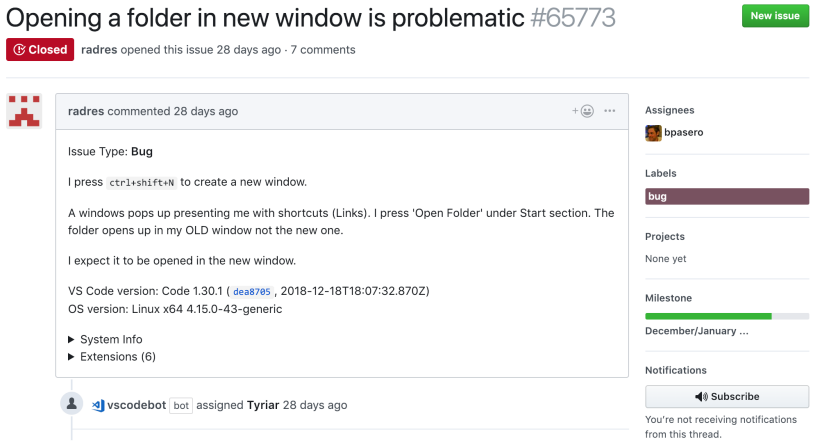


Figure 3.7: A typical GitHub issue with a title, a textual description and a label

Another central element of git is *branching*. A software project will at all time have a *master* branch, which should always contain the latest *working version* of software system. To allow for the development of new features, without the necessity of commiting directly to the master branch, git offers a lightweight branching model, in which developers are encouraged to diverge from the master branch, and work on their own branch dedicated to the work at hand e.g. the realization of a new feature or a bug fix.

To leverage this branching feature, GitHub recommends the usage of their GitHub Flow depicted in Figure 3.8.

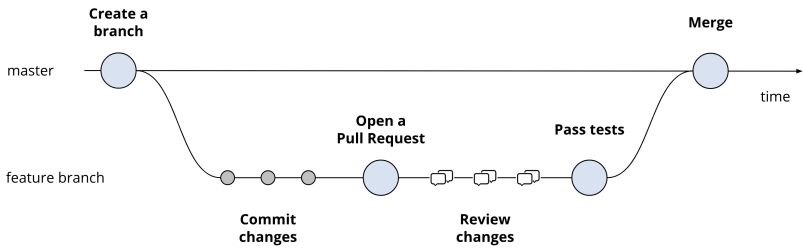


Figure 3.8: GitHub Flow. Source: [17] - modified

A pull request is a specialized type of issues, which also contains references to commits in a specific branch. Pull requests have been widely adopted in OSS environment, allowing developers to create their own branch which they can work from and open a new pull request containing their changes. This gives other contributors the opportunity to discuss and review the code changes. When the pull request has passed the test cases and is accepted by the owners of the project, it can be merged into the master branch. This enables developers, who do not have the privileges to commit to the master branch, to contribute to the project. An example of how a pull request can look like can be seen in Figure 3.9

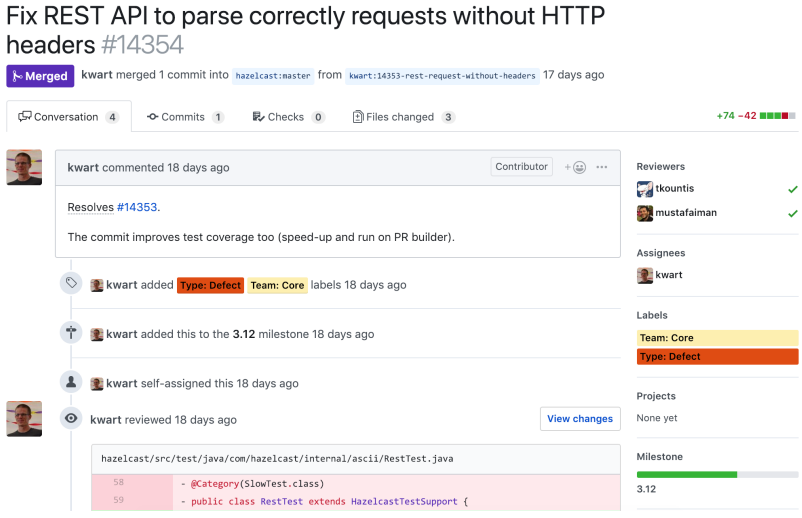


Figure 3.9: A labelled pull request that states to resolve a specific bug. The PR has been labelled, reviewed and merged into the master branch

This strong relationship between commits, issues, and pull requests, can be utilized by cross-referencing issues and pull requests in commits and vice-versa.

GitHub does also provide an API to query the information about publicly available projects. Table 3.7 shows a highlight of some valuable GitHub objects and attributes that contain useful data for studying the code changes, as well as, the cross-references to issues and pull requests.

Object	Attribute	Description
Commit	Author	The name of the author of the commit
	Date	The date and time when the commit was authored
	Files	List of the files changed in the commit
	Stats	Number of additions and deletions in the commit
	Message	The commit message (can be multiline)
File	Additions	Number of additions to the file
	Deletions	Number of deletions to the file
	Filename	The filename (including relative path) of the file
	Status	The status of the file (e.g. *modified*)
Issue	Assignees	List of users assigned to the issue
	Body	The body text of the issue
	Closed at	The date, if any, the issue was closed
	Closed by	The user, if any, the issue was closed by
	Created at	The date the issue was created
	Labels	List of the current labels labelled to the issue
	Milestone	The milestone, if any, the issue refers to
	Pull Request	The pull request, if it was created as such[4]
	State	The current state of the issue (e.g. *open*)

Table 3.7: Highlight of available data in GitHub

In a study from 2014, Kalliamvakou et al. identified multiple risks in using the data from GitHub for research and drawing conclusions in general [26]:

- A repository is not necessarily a project.
- Most projects have very few commits.
- Most projects are inactive.
- A large portion of repositories are not for software development.
- Two thirds of projects (71.6% of repositories) are personal.

[4] *"GitHub's REST API v3 considers every pull request an issue, but not every issue is a pull request. For this reason, "Issues" endpoints may return both issues and pull requests in the response"* [15].

- Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.
- If the commits in a pull-request are reworked (in response to comments) GitHub records only the commits that are the result of the peer-review, not the original commits.
- Most pull requests appear as non-merged even if they are actually merged.
- Many active projects do not conduct all their software development in GitHub

Kalliamvakou et al. also found, that not all active projects make use of GitHub's builtin issue tracker, and therefore manage issues in other systems. Additionally, some projects do also have contribution outliers, namely; committers with a very large nmber of commits, which suggests the committer to be a bot.

These perils are important to take into consideration in the context of MSR and bug prediction and therefore, a number of measures have to be taken during the development of the software system and the evaluation process. To evaluate the performance of combinations of process metrics across repositories, a systematic approach has to be used to select which repositories should be investigated. Consequently, a set of inclusion and exclusion criteria for the experiment will be defined in Chapter 7.

3.9 Domain

Following the introduction of the essential concepts and terms required for realizing the bug prediction tool, modelling, and describing the domain is a significant task towards understanding the requirements and proposing a design for the system. Modelling the domain allows for establishing the physical and abstract objects of the domain and their internal associations. The purpose for a domain model is manifold as it can, at a very high level, convey the key problem, scope and essentials of a software system. Additionally, the domain model helps defining the requirements of the system, as well as, to accelerate the design of the data model and the system architecture. The domain model in Figure 3.10 depicts the different key entities in the domain.

It is clearly seen, how *Github repository*, *File*, *Commit* and *Bug* are central entities in the domain with multiple and important associations. The entity *Project* can be perceived as the overarching concept, which gives rise to the

other entities in the domain. The *Project* and a corresponding *Github repository* relate to the overall software development process described in Section 3.1, while the *GitHub repository* and its derived entities stem from concepts and terms introduced in Section 3.8. Lastly, the entities *Feature*, *Prediction*, *Model*, and *Evaluation* originate from the terms introduced in Section 3.2.

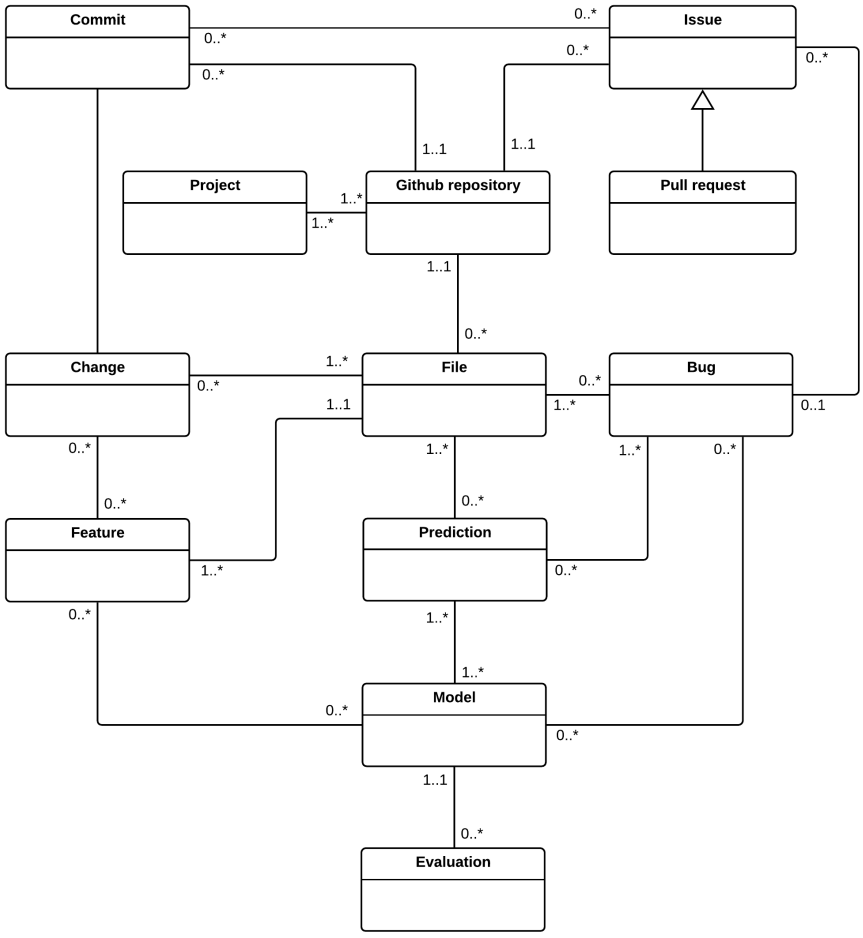


Figure 3.10: Domain model

Chapter 4

Requirements

With the terms and concepts necessary and relevant for the problem statement introduced, a set of formal requirements for the prediction system has been established. This chapter will cover the scope of the realization and present a requirements specification. Since the overall development method is based on the agile methodology, and since the prediction system to be developed will mainly be perceived as a prototype, the requirements have been prioritized to allow for a flexible work process.

4.1 Scope

The overall vision of this project is to create a prototype of a prediction system where the purpose is twofold. The main objective of the prediction system, and ultimately of the thesis, is to investigate and answer *RQ1* and *RQ2*. However, a derived objective is to also create a flexible and extensible library, for further use by either academia or industry. To this end, it is deemed important that the final product of the thesis is designed with sound software engineering practices and principles.

4.2 Requirements specification

Requirements are identified with a unique number in order to keep traceability in the project. Requirements prefixed with 1. refer to functional requirements whereas requirements prefixed with 2. refer to non-functional requirements. All requirements are labeled M, S, C or W using the rather informal priority technique, MoSCoW [7]. A mapping for the labels is seen in Table 4.1.

Label	Priority	Description
M	Must have	The requirement is all-important for the software.
S	Should have	The requirement is important, and has been estimated to be feasible within the project period.
C	Could have	The requirement is desirable, but not crucial for the product within the project period
W	Won't have	The requirement is identified and recognized as desirable, but has been estimated to be infeasible within the project period

Table 4.1: Labels and priorities with the MoSCoW technique

All requirements considered a "must have", do constitute the minimum viable product (MVP) of the realization. If one requirement prioritized as a must have is not achieved, the project can be considered a failure.

4.2.1 Functional requirements

The following functional requirements for the software system are as follows:

1.1: The system must be able to extract metadata from an OSS system hosted on GitHub **(M)**

1.2: The system must be able to extract metadata between two releases defined by datetimes **(M)**

1.3: The system must only extract necessary metadata **(S)**

1.4: The system must be able to extract post release metadata

1.5: The system must be able to identify all files changed in a time span **(M)**

1.6: The system must be able to extract metadata from two defined release numbers **(S)**

1.7: The system must be able to determine if a post release change is a bug fix **(M)**

1.8: The system must store the extracted data in an easily parsable format **(S)**

1.9: The system must be able to identify the number of commits made to a file between two releases **(M)**

1.10: The system must be able to identify the sum, average and maximum number of additions made to a file between two releases **(M)**

1.11: The system must be able to identify the sum, average and maximum number of deletions made to a file between two releases **(M)**

1.12: The system must be able to identify the number of different developers who made changes to a file between two releases **(S)**

1.13: The system must be able to identify the number of commits made to a file within a defined number of days before the later release **(C)**

1.14: The system must be able to identify the number of days from the later release since the last commit to a file **(C)**

1.15: The system must be able to identify if a commit includes refactoring **(C)**

1.16: The system must be able to identify the number of refactoring commits made to a file between two releases **(C)**

1.17: The system must be able to identify the number of bugs fixed in a file between two releases. **(S)**

1.18: The system must be able to identify the entropy of commits made to a file between two releases **(C)**

1.19: The system must be able to identify the center of concentration of the commits to a file **(C)**

1.20: The system must be able to identify the maximum sequence of commits to a file within a defined maximum gap between the commits **(C)**

1.21: The system must be able to identify the maximum and average number of other files changed together with file **(C)**

1.22: The system must be able to identify the age of a file **(C)**

1.23: The system must be able to identify the average number of days between commits to a file **(C)**

1.24: The system must be able to identify the number of commits to a file with no commit message **(C)**

1.25: The system must be able to identify the age of a file **(C)**

1.26: The system must be able to predict post release bugs by using a classification algorithm **(M)**

1.27: The system must support bug prediction by using multiple classification algorithms **(S)**

1.28: The system must support k-fold cross validation to evaluate the performance of a classifier **(S)**

4.2.2 Non-functional requirements

2.1: It must be possible to execute each data mining phase either alone or in combination with other phases of the data mining processing pipeline

Chapter 5

Design

This chapter describes how the bug prediction tool is designed. By applying and combining the concepts presented in Chapter 3, this chapter presents a detailed design that satisfies the requirements set for the system. The high level architecture of the system is presented, followed by a description of both the structural or static aspects of the system and the behavioural or dynamic aspects of the components. This chapter covers what is to be considered as "important" design decisions, and argues for the justification of each design decision.

Since the vision and ultimate goal of this realization is to answer the research questions, but also to create an extensible product, the architecture of the system has to be created in such a way, that both of these objectives are taken into consideration. This calls for an architecture that allows the system to execute the full data mining process pipeline presented in Figure 3.1, but also having the system logic behind each phase sufficiently decoupled in order to, for instance, execute only a subset of the phases on a particular data set[1] at a time. As the scope of this realization is to mine software repositories hosted on GitHub, there is only need for one integration to the outside world of the system, namely; to collect the data exposed on GitHub. At a high level, Figure 5.1 outlines what the realization should cover. Data from a software repository hosted at GitHub will feed the prediction tool with raw data, from which process metrics are extracted.

[1] Which can be either raw or preprocessed at execution time

A well structured data set using these metrics as features is preprocessed, and will go into a classification algorithm which is able to produce results, that is, predictions and evaluations of the classifier performance. Ultimately, these results will be visualized in a clear and understandable matter.

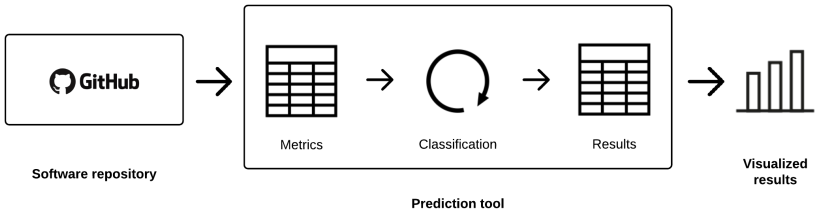


Figure 5.1: High level overview of the realization

To satisfy requirement *2.1*,

2.1: *It must be possible to execute each data mining phase either alone or in combination with other phases of the data mining processing pipeline*

the system must consist of a number of architectural elements or *components*, each having a clearly defined set of boundaries and responsibilities. The data mining process already has such boundaries defined in form of the different phases. For this purpose, it seems logical from an architectural point of view to propose a structural design that imitates these phases. In other words, the structural design will consist of components responsible for the tasks in each of the data mining phases.

Additionally, to accommodate the different needs for the system, a software system consisting of two main components is proposed as shown in Figure 5.2. The component `msrlib` is a library exposing the functionality necessary to perform each of the phases in the data mining processing pipeline. The library can be used as it is, but to produce the results necessary to answer RQ1, the component `RaV` is added to the system. `RaV` (Research and Visualization) consists of a research component, handling the execution of the full sequential data mining process by using the exposed functionality of `msrlib`. Additionally, `RaV` also contains a visualization component, which is able to present the gathered results in an understandable manner.

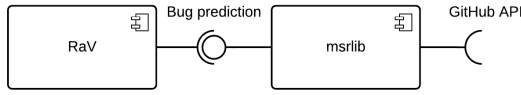


Figure 5.2: The two components constituting the realization of the software system

5.1 msrlib

Given that the purpose of `msrlib` is to provide access to the tasks contained in each data mining phase, it would not make sense to organize the component by a multilayered architecture. Rather, the library is structured "by feature" or sliced "vertically" thereby grouping related domain concepts [30]. The advantage of choosing a package-by-feature is in this case, that it allows for a higher modularity and minimal coupling between the other components. Figure 5.3 shows the internal structure of `msrlib`, where each subcomponent has the responsibility of each phase in the data mining processing pipeline.

The main responsibilities of each subcomponent are described as follows

`Extraction`: is responsible for extracting raw data for further use in the data mining processing pipeline. The primary input to this component is a data source providing access to a software repository (which in this implementation is GitHub), while the primary output is structured set of relevant change history data. Moreover, this component exposes functionality to search for GitHub repositories with a set of search properties.

`Preprocessing`: is responsible for the feature extraction, data cleaning and transformation (if necessary) and labelling of the files. The primary input to this component is the raw data extracted from the `Extraction` component. The output is a well structured data set as defined in Subsection 3.2.1. Additionally, the component can be extended to also provide an analysis of the features, as well as, feature selection.

`Prediction`: is responsible for conducting the prediction of bug prone files based on yet unseen data i.e. the class label of each file. The input is a trained model as well as a preprocessed data set with an unknown class label. The output is the predicted class label of each file.

`Validation`: is responsible for evaluating the performance of multiple classification algorithms. The input is a well structured data set containing

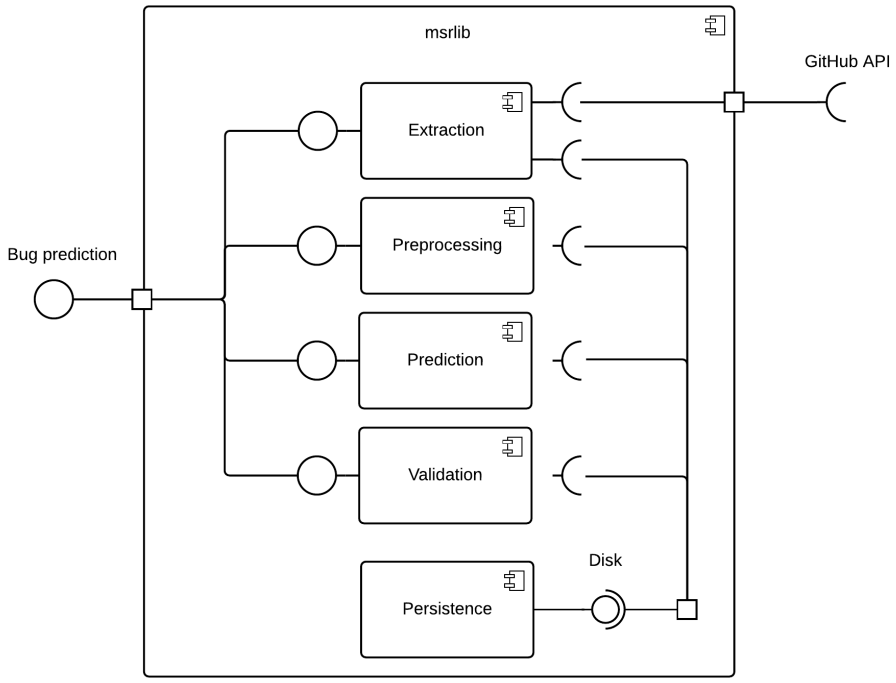


Figure 5.3: High level overview of the components and their dependencies

features and labels for each file. The output is a report on the results from the evaluation

`Persistence`: is responsible serializing the produced data to the disk. The component provides the interface *Disk*, which enables the callers to serialize the data structures to documents on the disk.

In general, the subcomponents of `msrlib` do not have any dependencies to each other, but instead, each provide an interface allowing a caller to use the service of each component individually. Since the subcomponents are not interdependent, but rather dependent on the results serialized by the use of `Persistence`, a data model has to be established.

5.1.1 Data model

The data model will be defined at the three traditional levels, namely with a conceptual model, a logical model and a physical model. The conceptual data model is closely associated with the domain model discovered in Figure 3.10. The conceptual data model reuses some of the entities discovered in the domain model, while it is necessary to introduce two new entities to evaluate the classification performance through the `Validation` component. The conceptual model is depicted as an entity-relationship (ER) model in Figure 5.4[2].

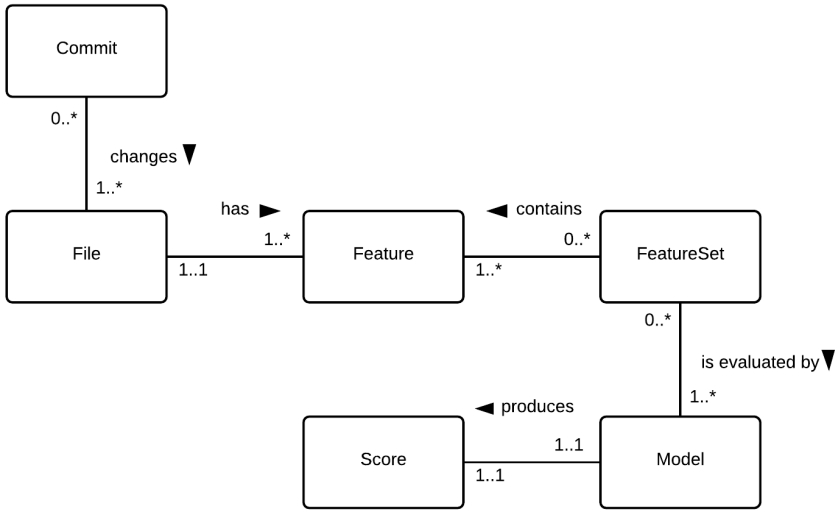


Figure 5.4: Conceptual entity-relationship model

Two new entities, *FeatureSet* and *Score* are introduced in the conceptual model. Score is a conceptual entity derived from the *Evaluation* entity, while *FeatureSet* is simply constituted by a number of features. The logical data model describes the information structure, reusing the entities and relationships identified in the conceptual model. The logical model is, however, technology independent, and does not describe the implementation of the data model. The logical data model is depicted in Figure 5.5

[2]Multiple ER notations exist and have been used throughout the software engineering history. In this thesis, UML is consistently used.

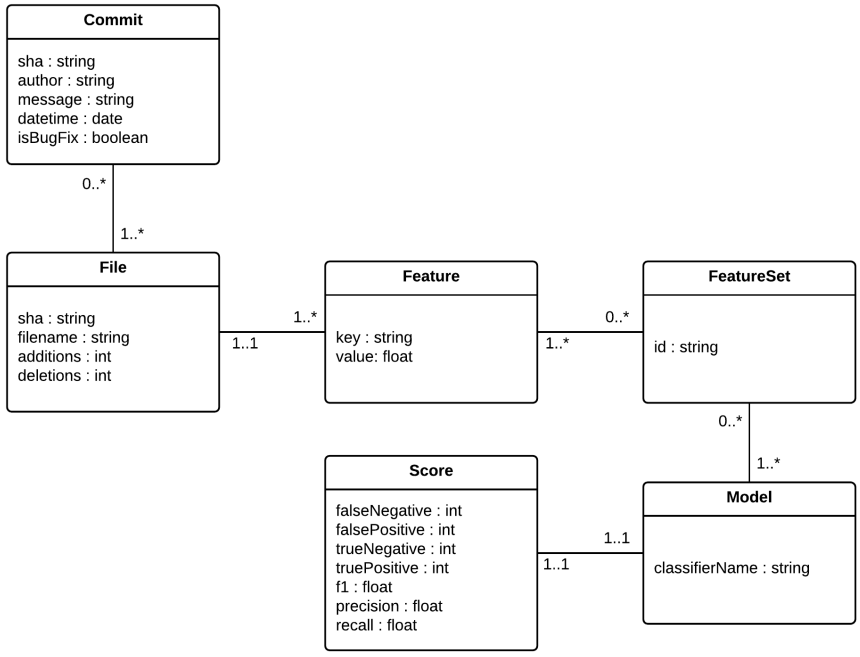


Figure 5.5: Logical data model

Ultimately, the physical data model describes the implementation and the *physical schema*. The ability to use each subcomponent of `msrlib` independently from each other, is a significant requirement of the software system. The physical data model must accommodate this need as well. To this end, the design avoids the use of a relational database schema, but rather use a non-relational approach by storing *documents* associated with each subcomponent. This allows callers of the library to only use a subset of the provided functionality, while still being able to get useful results, which can be used outside the `msrlib` environment. Moreover, as multiple machine learning libraries mostly work with flat files e.g. stored as comma-separated values (CSV) or JavaScript Object Notation (JSON), it has been deemed much more convenient for the callers to get the exact same file format instead of data stored in a less flexible relational database. For this purpose, all data is stored as JSON, and where possible also as CSV files. To keep the initial version of `msrlib` simple and flexible, a simple file-based approach has been chosen. A viable and more robust alternative, which could be an extension of the system, would be to set up a NoSQL database to store the documents. Working with non-relational documents instead of relational

tables, an embedded data model allows for storing the references as embedded data in the same document [32]. Relationships that, in a traditional relational schema, would be implemented as related tables with foreign keys are instead embedded in the same record.

When `msrlib` is used, the caller selects an output directory of the library. Each component will through the `Persistence` component store their output in a related directory. The directory structure below depicts what an output directory would contain following a full run:

```
/
├── extraction
│   ├── pre-release-commits
│   │   ├── 0a70cc6d46f75971b1bdbd465b1e633b175b5557.json
│   │   ├── bb7a60ffc5b3d19b592059fc0d98b771ebe45d2a.json
│   │   └── ..
│   ├── pre-release-files.json
│   ├── post-release-commits
│   │   ├── 6b2c0d902b003425a495c9236aad56adffd252d5.json
│   │   └── ..
│   └── post-release-files.json
├── preprocessing
│   ├── data.json
│   └── data.csv
├── validation
│   └── results.json
└── prediction
    └── predictions.json
```

`Extraction` stores information about each extracted commit in either the directory *pre-release commits* or *post-release-commits*. Likewise, a list of all the files modified in the period (either T_{pre} or T_{post} that is) are stored in a separate file. Listing 1 shows an example of the physical schema of an extracted pre-release commit. Please note how the *File* entity is embedded in the *Commit* entity.

A post-release commit contain most of the same information, but the additions and deletions of each file are replaced with a boolean value indicating whether the file is defective or not. An example of the physical schema is provided in Listing 3 in Appendix A.

The preprocessed data set that `Preprocessing` produces, is a simple multidimensional data set of records. These records stored as a combination of the *File* entitiy and the *Feature* entity both as CSV and JSON. An example of the physical data schema of the JSON document can be seen in Appendix A.

```
{
    "author": "jrieken",
    "datetime": "2018-10-02 19:04:47",
    "files": [
        {
            "additions": 3,
            "deletions": 3,
            "filename": "src/vs/editor/contrib/suggest/completionModel.ts",
            "sha": "a87441ed3911ccb10257f872dd619c1d6c5bcb69"
        }
    ],
    "message": "make sure `ensureLowerCaseVariants` has been called",
    "sha": "0fb324496a962f8fd0f1b8734e9d2daa4eaf3059"
}
```

Listing 1: Example of an extracted pre-release commit. Source of data: `https://github.com/Microsoft/vscode`

`Prediction` stores prediction results as a very simple list of files and a boolean value indicating whether each file is defective or not. This is stored both as CSV and JSON. An example of the physical data schema of the JSON document can be seen in Appendix A.

Lastly, `Validation` stores the results as a combination of the *FeatureSet*, *Model* and *Score* entity in a single document. An example of the physical data schema of the JSON document can be seen in Appendix A.

5.1.2 Extraction

To establish a data set that is suitable for the classification of buggy files, the outcome of the extraction phase should be the preliminary step towards creating the data set. The `Extraction` component must therefore, extract the raw data about all the changes including the files affected *before* a release. In other words, all commits in T_{pre} referred to as *pre release commits* must be extracted. The calculation of the different process metrics introduced in Section 3.6 will be explained in the following section, but in terms of the raw data needed, the following attributes introduced in Table 3.7 will be extracted: **author**, **date**, **message**, **file additions**, **file deletions**, **filenames** as well as the **commit sha** and the **file shas**.

Besides extracting the details of the commits to calculate the process metrics, it is essential to know the ground truth if a file that has been changed in T_{pre} is defective or not. To achieve this, it is necessary to consult the commits in T_{post} in order to infer whether a post release commit did fix a bug that was introduced in T_{pre}. If a commit is considered a bug fix, and since the changeset of a commit may include multiple files, all files in the changeset will be labeled defective. Thus, to construct the data set, all pre-release commits will be used in the preprocessing phase to extract the features of the files, while all post-release commits will be used to label the files.

In order to infer the ground truth from the post release commits, a better understanding of the relationship between files changed in pre- and post-release commits and the reported post-release issues is needed. Figure 5.6 shows an example of a excerpt of the lifecycle of the files f and g and the issue i. Consider the two files, f and g, that each have had three changes in T_{pre}. Whether or not these changes stem from the same commits is irrelevant in this context. Just after the transition into T_{post}, the issue i is created. Later, both files f and g are changed in the same commit whose commit message holds a reference to i. As it turns out that issue i is marked as a bug, and both files f and g was part of the changeset in the bug fix, both f and g can be considered defective.

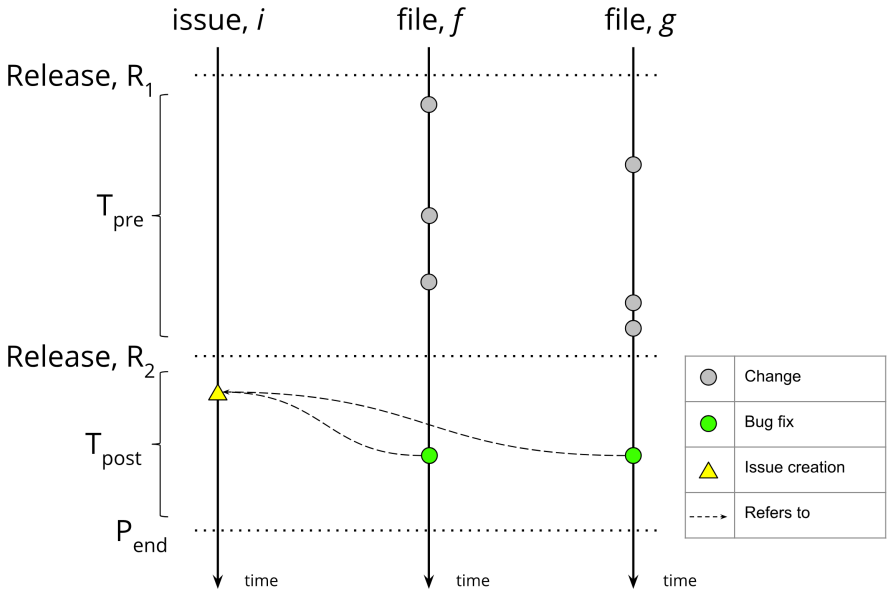


Figure 5.6: Example of changes to files f and g and their relationship with issue i. Issue i can also be a pull request

The introduction of the concepts of issues and pull requests in Chapter 3 revealed, that issues and pull requests can contain chains of references. A simple example is a commit that through its commit message refers to pull request, A. The body text of PR A contains a reference to issue B, and issue B contains a reference to issue C in its title. This leads to a huge network of references, which would require more sophisticated measures to evaluate whether a commit is a bug fix through a chain of references.

Therefore, it has been decided that `msrlib` will only evaluate the first reference from a commit message and thus not parse issue texts to find and lookup additional references. This introduces an important limitation to `msrlib` in the sense that the directly referenced issue/PR from a commit message *must* indicate whether it is a bug or not. Since many GitHub issues are mostly a mix-up of reported bugs, new features requests, questions, etc. the issue labels will be used to determine whether an issue is a bug or not.

With this in mind, a general approach to label the files as defective or not is proposed as pseudocode in algorithm 1.

Algorithm 1: IsBugFix

Input: A GitHub commit with a commit message
Output: Whether the commit is a bug fix or not

```
if commit message contains issue reference then
    get issue from issue tracker
    if issue was created in T_post then
        if issue is marked as a bug then
            return true
        end
    end
end
return false
```

This simple approach will find the issue reference (if any) in the commit message, and find the issue in GitHub's builtin issue tracker. If the issue was created in T_{post} and is also marked as a bug, the changeset in the commit is labelled defective.

Multiple approaches are available to actually extract data from GitHub. The two most commonly used are: (1) directly requesting the GitHub REST API, (2) using a curated dataset mirroring GitHub's data. The use of these approaches are approximately distributed evenly among other studies [8]. The main disadvantage of the GitHub API is the request rate limit, which is currently 5,000

requests/hour for an authenticated user. The alternative to the GitHub API is primarily *GHTorrent* which is an offline mirrored data set. For the scope of this project, it has been deemed acceptable to use the GitHub REST API despite the rate limit constraint. Since data from only a single repository will be extracted, the rate limit is not considered problematic. As the extraction component is not subject to execution time requirements, measures can be taken to get around the limit if need should be. Additionally, wrapper libraries for the GitHub REST API are avaliable for multiple programming languages [16] with a substantial amount of documentation and examples.

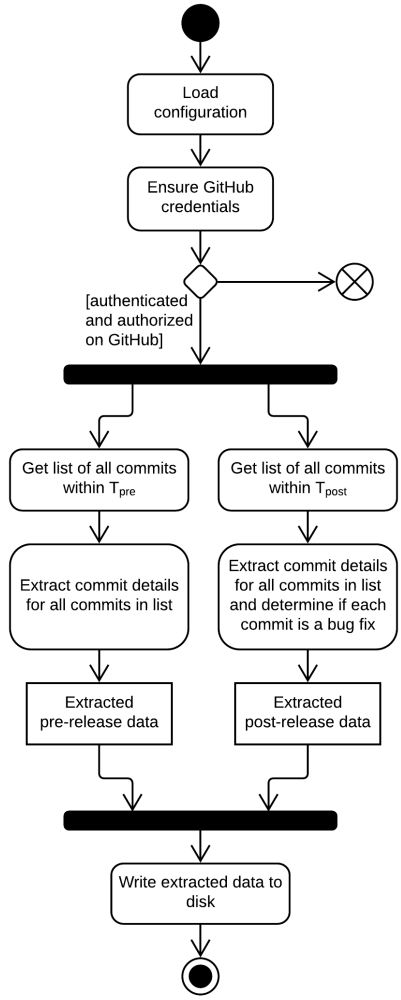


Figure 5.7: Overview of the overall extraction process

Combining these decisions, the overall process in the extraction phase is outlined in Figure 5.7. Initially, the environment configuration is loaded, providing the GitHub credentials for the extraction, the repository to extract data from the definition of T_{pre} and T_{post} as well as additional extraction settings.

5.1.3 Preprocessing

The `Preprocessing` component has multiple responsibilities, but the majority of the work in this design, is to extract and construct new features from the raw data. Recall the process metrics that was introduced in Table 3.6 on page 22. A mapping has been created between the descriptive names of the features and an internal code name as seen in Table 5.1. It must be noted that, `msrlib` in its current form does not support the extraction of the metrics *max burst*, *age* and *weighted age*. This limitation will be addressed in Chapter 7.

Code name	Descriptive name
COMMITS	Commits
DELE	Deletions
ADD	Additions
AUTHORS	Authors
CMPRERE	Commits pre release
LASTCM	Last commit
REFACTOR	Refactorings
INDEVBUGS	In development bugs
ENTROPY	Entropy
MPOCH	Mean period of change
MAXCHSET	Maxmimum change set
AVGCHSET	Average change set
MAXADD	Maximum additions
AVGADD	Average additions
MAXDEL	Max deletions
AVGDEL	Average deletions
AVGTBCH	Average time between changes
NOCMMSG	No commit messages

Table 5.1: Mapping between the descriptive names and the code names of the process metrics

The following section describes how these metrics are calculated from the raw data.

COMMITS:
The sum of commits in T_{pre} that have had the file as part of their changeset.

ADD:
The sum of LOC added to the file by all commits in T_{pre}.

DELE:
The sum of LOC deleted from the file by all commits in T_{pre}.

AUTHORS:
The sum of unique authors that have commited changes to the file in T_{pre}. The authors are identified from the file commits.

CMPRERE:
The sum of commits in T_{pre} which had the file as part of their changeset "just before" the release date, R_2. As release policies vary across projects, the term "just before" can be configured by the number, n, such that all commits in $[R_2 - n; R_2]$ are considered "just before".

LASTCM:
The number in days between the latest commit to the file in T_{pre} and R_2.

REFACTOR:
The sum of commits changing the file in T_{pre} which contains refactoring changes in their changeset. As contribution policies across projects vary, commits are identified as refactorings, if their commit message contains a refactoring related keyword e.g. 'refactor'. Accordingly, these keywords can be defined as part of the configuration.

INDEVBUGS [33]:
The sum of commits changing the file in T_{pre} that fixes bugs in their changeset. As contribution policies across projects vary, commits are identified as bug fixes if their commit message contains a bug fix related keyword e.g. 'bug' or 'fix'. Consequently, these keywords can be defined as part of the configuration.

ENTROPY [34]:
T_{pre} can be divided into N equal periods, and provided that all changes to a file in T_{pre} are uniformly distributed, an equal amount of commits would be present in each of the N periods. However, as this is almost guaranteed to not be true in reality, Shannon's entropy can be used to calculate the change distribution. The distribution of changes can be defined by following Shannon's definition of information entropy [43]:

$$H = -\sum_i p_i \log p_i \tag{5.1}$$

Given C_i as the number of commits to the file in the ith period of N and C_{total} as the total number of commits to the file in T_{pre} the entropy for a file is calculated as:

$$ENTROPY = -\sum_{i=1}^{N} \left(\frac{C_i}{C_{total}} \right) \log \left(\frac{C_i}{C_{total}} \right) \tag{5.2}$$

MPOCH [34]:
The mean period of change is constructed in a similar way as the entropy, by dividing the timeline into N periods. The mean period of change is an estimate the center of the concentration of changes in the timeline. This is done as follows:

$$MPOCH = \sum_{i=1}^{N} i * \left(\frac{C_i}{C_{total}} \right) \tag{5.3}$$

MAXCHSET [33]:
The maximum number of files in the same changeset as the specific file of all commits in T_{pre}.

AVGCHSET [33]:
The average number of files in the same changeset as the specific file of all commits in T_{pre}.

MAXADD:
The maximum number of LOC added to the file by one of all commits in T_{pre}.

AVGADD:
The average number of LOC added to the file by all commits in T_{pre}.

MAXDEL:
The maximum number of LOC deleted from the file by one of all commits in T_{pre}.

AVGDEL:
The average number of LOC deleted from the file by all commits in T_{pre}.

AVGTBCH [40]:
The average time in days between all commits changing the file in T_{pre}.

NOCMMSG:
The sum of commits changing the file in T_{pre} with an empty commit message

In addition to the calculation of the process metrics, `msrlib.preprocessing` will also label each file with the binary class 0;1 indicating if the file is defective (1) or not (0). This is simply done by transforming the total number of bug fixes to a file to the binary equivalent. If a file has had at least one bug fix, it will be labelled defective (1). The overall process of the feature extraction in `msrlib.preprocessing` is seen in Figure 5.8.

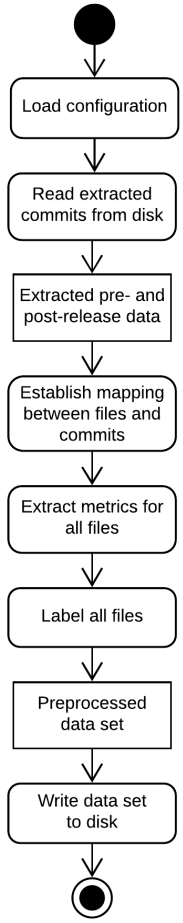


Figure 5.8: Overview of the overall process of the calculation of process metrics

In addition the calculation of process metrics, `msrlib.preprocessing` will also expose limited functionality to explore and gain knowledge about the extracted features. In the scope of this project, `msrlib.preprocessing` will provide calculations of the feature correlations, as well as, the generation of a correlation heatmap. Moreover, `msrlib.preprocessing` provides a basic data-based approach of feature selection namely through recursive feature elimination with cross validation. The details will also be covered in Chapter 6.

5.1.4 Prediction

`msrlib.prediction` is used to predict which files of a yet unseen data set are defective. Using this component is done by providing two data sets; (1) a preprocessed data set containing the relevant features *and* the ground truth which will be used to train the model and (2), a preprocessed data set without class labels that the classifier will use to predict on. The caller of this component can use the other components `msrlib.extraction` and `msrlib.preprocessing` to generate such data sets. Both of these components support the possibility to work with unlabelled data sets i.e. `msrlib.extraction` will not try to infer the ground truth and `msrlib.preprocessing` will not label the files.

5.1.5 Validation

The overall design of `msrlib.validation` is quite simple, since the sole purpose of this component is to build multiple classification models from the preprocessed data set and evaluate the performance of each classifier built with different metric sets by using k-fold cross validation.

An important design decision is however, to handle the generally high class imbalance in the bug prediction domain and the quite small data sets overall. `msrlib` incoorporates two measures to deal with this matter. A classification model is built in such a way, that it penalizes the prediction errors of the minority class higher than errors related to the majority class. The details of the implementation of this penalization is provided in Chapter 6.

Moreover, cross validation with a high number of folds could potentially lead to training and test folds without any records with the positive class for highly imbalanced data sets. For instance, with a data set containing 500 files with only 10 % of the files being defective and using cross validation with 10 folds could possibly lead to the case, where only a few or no positive labels would figure in a fold. This scenario is generally not desirable, as a fold would thereby not be

representative of the whole data set. Therefore, the technique of *stratification* is used along with cross validation. *"Stratified cross-validation uses proportional representation of each class in the different folds and usually provides less pessimistic results."* [1].

`msrlib.preprocessing` can be configured both in terms of which classifiers and which metrics that should be used to build a model. The overall validation process in `msrlib.validation` is seen in Figure 5.9.

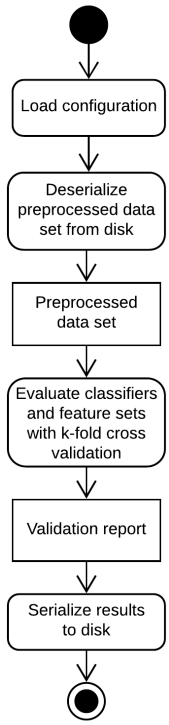


Figure 5.9: Overview of the overall validation process

5.1.6 Configuration

In order to make `msrlib` flexible and applicable across repositories with different structures and policies, `mrslib` has to be made highly configurable in all parts of the data mining process that the library supports. As a result, a global configuration file will be shipped with the library, which allows the caller to configure the different phases as needed. As the general use of `msrlib` will

include significant configuration, it has been deemed more convenient to gather the configuration in one file, rather than passing parameters to single functions.

5.2 RaV

The purpose of the `RaV` component is twofold. The main objective of the component is to answer the research questions of the thesis, by using the functionality exposed by the `msrlib` library. Moreover, the component contains a prototype of the visualization of the results generated by the use of `msrlib`. The motivation behind this, is to also facilitate the bug prediction capabilities to other activities than research i.e. real usage in current software projects. The visualization component is intended as a first iteration of a web-based product that includes the functionality of `msrlib`.

5.2.1 Research

The `Research` subcomponent will be able to execute the full data mining process by using all of the before-mentioned subcomponents of `msrlib`. Furthermore, the component will handle the selection of repositories used in the evaluation in Chapter 7 by systematically inspecting relevant repositories.

5.2.2 Visualization

The `Visualization` subcomponent will be a prototype that, in simple manner, will visualize the results generated by `msrlib`. The prototype is the first step towards a fully-fledged open-source web application that can be used for bug prediction in practice. The overall vision of this component is a web-based bug prediction application allowing fellow developers to make bug predictions on their GitHub-hosted projects. This could be done by typing in a target repository, use GitHub with OAuth and request the GitHub API with their own user, and then use `msrlib` to conduct the actual predictions. The results could ultimately be presented in the user interface. `msrlib.rav` will, as a result, contain a single page application realized with a reactive JavaScript framework together with a proven user interface design. The details of the realization of this component will be presented in Chapter 6

Chapter 6

Implementation

This chapter introduces the technologies, frameworks and libraries used for the realization of the system, as well as, describing the most important parts of the implementation. The system is generally implemented to conform with the SOLID design principles [30].

6.1 Technologies

Python has been used to realize most of the components, since it is a general purpose programming language that has been widely adopted in the data science community. With a high flexibility and an easy syntax, it is easy to create both small scripts at hand as well as full-fledged applications. Due to a substantial community, a wide variety of libraries are available, many of them centered around data science and data manipulation. Most notably - and relevant for this realization - a variety of sophisticated machine learning libraries are avaliable e.g. *scikit-learn* and *TensorFlow*.

`msrlib` constitutes the core of the system and is implemented as a python package. The vision of the library is to make it publicly available through the python package manager, *pip*. In its current form, `RaV` is also structured as a

python package containing both the python scripts used for the research and a JavaScript web application used for the visualization. These subcomponents should eventually be divided into self-contained packages. The web application is realized as a JavaScript application with a vue.js frontend running on a webpack development server.

The following list highlights the technologies and libraries used for the realization:

- **Python 2**
  - **scikit-learn**: used for the all machine learning tasks, including classification and performance evaluation
  - **pandas**: used for general data manipulation in combination with scikit
  - **numpy**: used for manipulation of arrays
  - **PyGitHub**: used to request the GitHub REST API. The library wraps GitHub entities to python objects
  - **python-bugzilla**: used to request the BugZilla REST API. The library also wraps BugZilla entities to python objects
  - **seaborn**: used to visualize feature correlation with a heatmap
  - **matplotlib**: used for various plotting tasks during development
  - **pickle**: used to serialize classification models
- **HTML5, JavaScript (ES6), CSS**
  - **vue.js**: used as a reactive JavaScript framework to build a component-based frontend.
  - **vue-material**: used for the design of the UI. The library provides vue components designed to conform with the Google Material Design specifications
  - **webpack**: used during development to serve the vue.js project on localhost
- **JetBrains PyCharm** (with several plugins)
- **Atom** (with several plugins)
- **Git and GitHub**

6.2 msrlib

Only the most important and non-trivial aspects of the implementation of `msrlib` will be covered in this section since both the structural and behavioral design have been covered in Chapter 5.

6.2.1 Extraction

The most important object in `Extraction` is the class, `GHExtraction` which exposes one function, namely `extract(is_post_release, label)`. The function enables the caller to extract commit data from the repository specified in the run commands file. The whole extraction is handled internally by the class and the output is a list of `ExtractedCommit` classes. The `ExtractedCommit` class is a simple data structure that contains primitive data about a commit e.g. the author, commit time, message, files etc.

The GitHub API has a request limit in its current version at 5,000 requests/hour for any OAuth authenticated users. Many extraction tasks will not exceed the hourly limit, but `GHExtraction` can be configured to add a delay between each request to keep below 5,000 requests/hour. The limit is reset an hour from the first request, which allows for continuous extraction, as long as the limit is not exceeded. The GitHub API is requested via the PyGitHub library which is a wrapper library for the REST API.

The Eclipse projects in GitHub do not use the builtin issue tracker, but has bug reports managed in BugZilla. To accommodate the evaluation of the eclipse projects, BugZilla is used similarly to the GitHub issue tracker, to extract whether a commit is a bug fix or not. This is achieved with the use of the python-bugzilla library. BugZilla does not have any request limit.

6.2.2 Preprocessing

The calculation of the process metrics is done by the `FeatureExtraction` class with the `extract_features(pre_rel_commits, post_rel_commits, label)` function. The input is a list of extracted pre-release and post-release commits and a boolean value, indicating if the files shall be labeled. For each of the pre-release commits, the changed files are identified and stored as `File` classes. The `File` class has a filename and it contains a list of commit ids i.e. the SHA1 of a commit and a dictionary of the process metrics, where the calculated values will be

stored with the metric key. When all files are identified, all process metrics are calculated and saved to each file. The output is a list of `File` classes containing process metrics.

Moreover, `Preprocessing` provides functionality to do data based feature selection by means of recursive feature elimination using stratified k-fold cross validation. The class `FeatureSelection` exposes `select_features(dataset)` which takes in a preprocessed data set, builds a classification model using a specified algorithm and runs recursive feature elimination on the k folds. The output is the set of features that yielded the highest score on the data set. The implementation of this functionality is based on the scikit-learn library.

6.2.3 Validation

The evaluation of the classifiers can be done by the use of the `Validation` class and its `eval_performance(dataset)` function. The function builds one or more classification models defined in the run commands, and conducts stratified k-fold cross validation for all combinations of metrics on the data set. The output is a *ValidationReport* object, containing the scores of all metric sets and different classifiers. The different classifiers, naïve Bayes, logistic regression, decision trees and random forest are built with the scikit-learn library. To overcome the class-imbalanced problem of the data sets, logistic regression, decision trees and random forest are built with a class weight, which penalizes classification errors with a defined class weight. Scikit-learn provides a *balanced* class weight, so that classification errors made to the minority class is penalized harder than erros to the majority class.

6.2.4 Configuration

The configuration of `msrlib` is conducted by adjusting the library's run parameters. The file `msrlibrc.py` is stored at the root of the `msrlib` package, and contains a list of all global run parameters. The default run parameters can be seen in Appendix B. Listing 2 shows an example of how to override the default run parameters e.g. by setting up the library to target the Eclipse JDT Core project.

```
from msrlib import msrlibrc

msrlibrc.REPOSITORY['USERNAME'] = 'eclipse'
msrlibrc.REPOSITORY['REPOSITORY_NAME'] = 'eclipse.jdt.core'
msrlibrc.RUN_TITLE = 'eclipse-4.9-4.10'
msrlibrc.OUTPUT_DIR = '~/bug-prediction/runs'
msrlibrc.START_DATE = '2018-09-19'
msrlibrc.END_DATE = '2018-12-19'
msrlibrc.MAX_FIX_TIME = 45
```

Listing 2: Example of overriding the default run parameters of `msrlib`

6.3 RaV

This section covers the non-trivial details of the implementation of the `RaV` component.

6.3.1 Research

`rav.research` ties together the different functionalities of `msrlib` to conduct the experiment of the thesis. This cover all the phases in the data mining process pipeline including extraction, preprocessing and analysis. The analysis covers both the evaluation of the classifier performance, as well as, a live prediction of defective files.

6.3.2 Visualization

`rav.visualization` is a *Vue.js* project used together with the *vue-material* library. The implementation is a prototype of how a potential realization of a full-fledged web application could look like. The motivation is to allow the adoption of MSR and bug prediction in industry for real life decision making. A webpack development server is used to serve the web application locally. The application is a single page application comprised of six different *vue components*. *TabRouter.vue* uses routing to provide access to five different tabs. The content of each tab; *Intro*, *Metrics*, *Correlation*, *Validation*, and *Prediction* is created by respective vue components. An example of the validation page can been seen in Figure 6.1

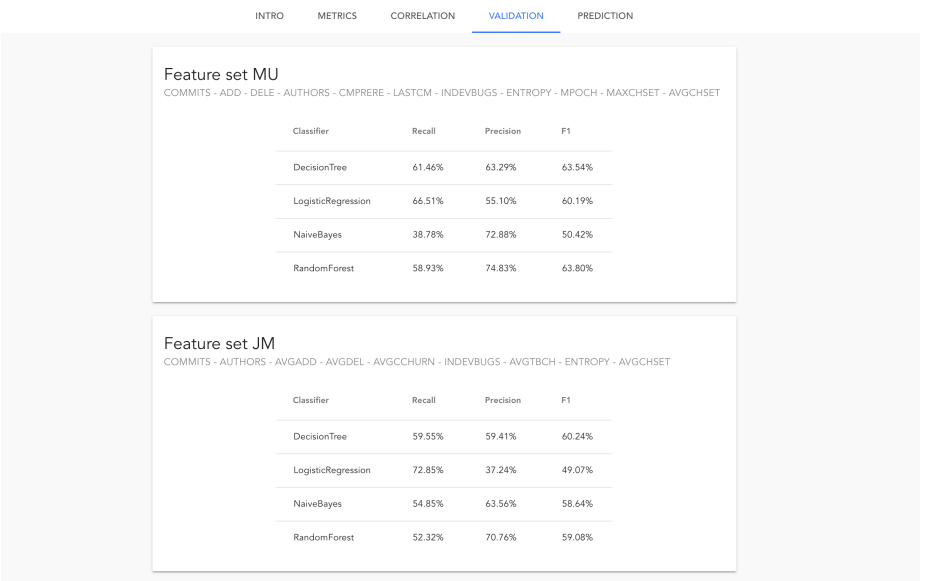


Figure 6.1: Screenshot of the *Validation* component vue.js project showing the performance of each metric set on the different classifiers

Given that the visualization component only plays a minor role in the complete system implementation, the web application should merely be perceived as a high fidelity prototype of the user interface in a full-fledged solution. The numbers used in the example stem from sample data that is shipped with the application. Hence, it is currently not possible to provide a new data set through the user interface.

Chapter 7

Results and evaluation

In this chapter, the performance of the defined process metrics in Subsection 5.1.3 is evaluated. As for the experimental setup, the implemented `RaV` component will be used in combination with the bug prediction library `msrlib` to answer the research questions that motivated the work. For the experiments, a set of inclusion and exclusion criteria is established in order to select relevant repositories to analyze. When the results are presented, each repository is briefly introduced by some basic properties e.g. the number of commits, contributors, issues and pull requests, followed by an informal description of the main structure and contribution policy of the repository.

7.1 Data selection

To scope the project and to provide the most valuable results, a set of inclusion and exclusion criteria for the repositories to be studied are defined. The criteria are created in such a way, that the extracted data sets are sufficiently large both in terms of commits and reported bugs. The inclusion and exclusion criteria are defined in Table 7.1

Inclusion criteria (a repository must)	Exclusion criteria (a repository must not)
(I1) be publicly available on GitHub (I2) have more than 9,000 stars (I3) contain more than 10,000 commits (I4) have at least 5 contributors (I5) use GitHub's issue tracker (I6) contain commits which reference GitHub issues/pull requests (I7) contain a software project that is divided with releases	(E1) have bugs reported in third party issue trackers (besides BugZilla) (E2) mix together bugs with other types of issues unless they are clearly marked as such (e.g. new features)

Table 7.1: Inclusion and exclusion criteria

As GitHub hosts millions of projects, a number of criteria are defined to narrow the target group. To this end, repository *stars* are used to initially filter out a vast majority of all publicly available repositories. A repository star indicates that a GitHub user has actively chosen to follow that repository, thus, GitHub uses stars to rank repositories' popularity and suggest similar repositories to interested followers [14]. The rationale behind using a high number of stars i.e. the most popular repositories on GitHub, is to exclude repositories that may suffer from the perils identified in Section 3.8. Given that most of the GitHub repositories have very few commits, are inactive or simply not used for software development, the repository popularity works as a good initial indiciator in the data selection process. The numbers used in criteria (I2)-(I4) are consequently used as an evidence for a substantial data set.

The initial filtering of the available repositories are conducted automatically by implementing inclusion criteria (I1)-(I5) as a python script. The `RaV` component has been extended to also contain data selection functionality by using the PyGitHub library. (I1)-(I5) are all criteria which can easily be automated by simple checks when inspecting each repository. Criteria (I6)-(I7) and (E1)-(E2) are, however, more difficult to apply computationally, as the structure and policies of the repositories vary greatly. To this end, these criteria have been applied by inspecting and assessing the structure of the repositories manually. All closed issues and pull requests are filtered by the keyword *bug* through the GitHub user interface, which searches for the keyword in the body of the issues or pull requests. All 25 results from the first page are assessed. If at least 2/25 issues or pull requests clearly indicate that they are bugs without being properly

labeled as bugs, the repository is rejected. The indication of an issue/PR being a bug is assessed by inspecting the title. An example is a PR with the title *"Fixes #66573 - search hidden search results"*. While the referenced issue *#66573* is indeed an issue labelled as a bug, the pull request itself is not labelled as a bug nor a bug fix. As `msrlib` does not support inferring if a commit is a bug fix by looking a network of issues or pull requests, but only look up the directly referenced issue from the commit message, the repository is rejected.

The data selection process is outlined in Table 7.2

Step	**# of repositories**
(I1) - (I2): Identify public repositories with at least 9,000 stars by using PyGitHub and the search endpoint of the GitHub API	1,169
(I3) - (I5): Traverse each repository and check for number of commits, contributors and the use of the issue tracker using PyGitHub and the repository endpoint of GitHub API	108
(I6) - (I7) and (E1) - (E2): Manually inspect the structure of the repositories assessing the use of labels in issues/PRs.	8

Table 7.2: Inclusion and exclusion criteria

The 108 repositories included after the application of criteria (I3) - (I5) are listed in Appendix C.

Project name	**Location**
Eclipse JDT Core	`https://github.com/eclipse/eclipse.jdt.core`
Eclipse JDT UI	`https://github.com/eclipse/eclipse.jdt.ui`
NumPy	`https://github.com/numpy/numpy`
Plotly	`https://github.com/plotly/plotly.js`
Yii	`https://github.com/yiisoft/yii2`
Terraform	`https://github.com/hashicorp/terraform`
Godot	`https://github.com/godotengine/godot`
Symfony	`https://github.com/symfony/symfony`
Ansible	`https://github.com/ansible/ansible`
Elastisearch	`https://github.com/elastic/elasticsearch`

Table 7.3: Selected repositories

In addition to the 10 accepted repositories, the Eclipse project is also included to compare the results with previous studies. The experimental setup can easily be expanded to cover more repositories, as the implemented library `msrlib` supports

bug prediction on any GitHub-hosted repositories. The selected repositories for this experiment are listed in Table 7.3.

7.2 Metric sets

To answer the research questions, *RQ1* and *RQ2*, multiple sets of metrics will be used in the evaluation. Two metric sets proposed in other studies will included in the evaluation, as well as two new sets proposed in this thesis. The metrics used by Moser et al. [33] on the Eclipse project performed well with high recall scores. These metrics are used as a set in this work referred to as *MO*. The metrics used by Muthukumaran et al. [34] constitute another set of metrics referred to as *MU*. It is important to note, that a current limitation in `msrlib` is, that it does not support the calculation of the metrics *AGE*, *WAGE* and *MAXBURST*. As a result, the metric sets are not completely identical to the ones used in the other studies. In addition to these metric sets, two new sets are proposed, containing a subset of the available metrics in `msrlib`.

Identifier	Metrics			
MO	COMMITS	ADD	MAXDEL	AVGCCHURN
	REFACTOR	MAXADD	AVGDEL	MAXCHSET
	INDEVBUGS	AVGADD	CCHURN	AVGCHSET
	AUTHORS	DELE	MAXCCHURN	
MU	COMMITS	AUTHORS	INDEVBUGS	MAXCHSET
	ADD	CMPRERE	ENTROPY	AVGCHSET
	DELE	LASTCM	MPOCH	
JM	COMMITS	AVGDEL	INDEVBUGS	ENTROPY
	AUTHORS	AVGCCHURN	AVGTBCH	AVGCHSET
	AVGADD			
JM2	COMMITS	AUTHORS	MAXADD	MAXCHSET
ALL	COMMITS	REFACTOR	AVGCHSET	AVGTBCH
	ADD	INDEVBUGS	MAXADD	NOCMMSG
	DELE	ENTROPY	AVGADD	CCHURN
	AUTHORS	MPOCH	MAXDEL	MAXCCHURN
	CMPRERE	MAXCHSET	AVGDEL	AVGCCHURN
	LASTCM			

Table 7.4: Set of metrics used in the experiments

The first, *JM* has been created by (1) analyzing the correlation between the

metrics and (2) trying to include metrics that describe different aspects of the development process. The correlation between the metrics have been analyzed by merging the five generated Eclipse JDT Core data sets into one joint data set. A correlation heatmap has the been generated using the Spearman correlation and only the metrics with small correlation coefficients are considered. The correlation heatmap can be seen in Appendix D. The metrics included in JM is an attempt to describe the changes by content, time and people. The other set, *JM2* has been created by considering only the commits made to the files. Lastly, a set with all available metrics is included referred to as *ALL*.

The different combinations of metrics are seen in Table 7.4.

7.3 Results

The empirical data have been obtained by using the implemented `RaV` in combination with `msrlib`. Lists of the classification performances for all repositories, measured by the precision (P), recall (R) and combined F1 score can be seen in Appendix E. The numbers are presented as percentages. Table 7.5 maps the classifiers to identifiers in the generated results tables. The configuration of the classifiers are the same across all runs, and all features have been extracted with the parameters seen in Appendix B. The bug fix period has for all releases been set to the number of days from the release date until the next consecutive release of the same version type i.e. major, minor or patch.

Identifier	Classifier
NB	Gaussian naïve Bayes
LR	Logistic regression
DT	CART decision tree
RF	Random forest using CART decision trees

Table 7.5: Classifiers

In the study of Moser et al. [33], they evaluated the performances with the recall and the false positive rate (FPR). They justified this, by putting a strong emphasis on the recall score, as it expresses the classifiers ability to deal with false negatives i.e. how many of the actual positives were predicted. False negatives are in the domain of bug prediction quite undesirable, since an actual defective file that is not predicted will lead to costly software bugs. On the other hand, precision expresses how the classifier deals with false positives i.e. how many of the positive predictions were actually correct.

The results in this are ultimately presented using the F1 score. This is justified by the fact, that precision expresses classification errors that undeniably also can be costly for development teams in the sense, that they will waste resources on non-defective files. Moreover, the F1 is calculated by the harmonic mean of the precision and recall, and thus, if the recall score would very low, e.g. 10 % and the precision high e.g. 75 %, the F1 score would still be very low (17.5 % in this example).

The following pages contain a brief introduction to each repository along with tables showing the best performing metric set for each release and for each classifier. Most importantly, the last column of the tables, *Best* shows which metric set scored the highest F1 score in the release.

7.3.1 Eclipse JDT Core

Eclipse JDT Core `https://github.com/eclipse/eclipse.jdt.core`	
# **of commits:** 23755	# **of issues:** N/A
# **of contributors:** N/A	# **of pull requests:** N/A
Summary: Eclipse JDT Core constitutes the infrastructure of the popular Eclipse IDE. It is developed by The Eclipse Foundation and was initially released in 2001. The project is written in Java. Eclipse has a thorough introduction to contributing to the project with certain policies on how to proper use the version control system and structure the code.	

Release	NB	LR	DT	RF	Best
3.0 - 3.1	JM	MU	JM2	MU	**MU**
3.1 - 3.2	JM	MO	JM2	ALL	**MO**
3.2 - 3.3	JM	MU	JM2	JM2	**MU**
3.3 - 3.4	JM	JM2	ALL	ALL	**JM2**
3.4 - 3.5	JM	MO	JM	JM	**JM**

7.3.2 Eclipse JDT UI

Eclipse JDT UI https://github.com/eclipse/eclipse.jdt.ui	
# **of commits:** 27908	# **of issues:** N/A
# **of contributors:** N/A	# **of pull requests:** N/A
Summary: Eclipse JDT UI constitutes the user interface of the Eclipse IDE. Contribution guidelines similar to the ones in JDT Core are available. The project is written in Java.	

Release	NB	LR	DT	RF	Best
3.0 - 3.1	JM	MO	ALL	MU	**JM**
3.1 - 3.2	MU	JM	MO	JM	**JM**
3.2 - 3.3	JM	MO	MU	JM	**MO**
3.3 - 3.4	MU	MO	ALL	MU	**MO**
3.4 - 3.4	MU	MO	MO	JM2	**MO**

7.3.3 NumPy

NumPy https://github.com/numpy/numpy	
# **of commits:** 19625	# **of issues:** 6883
# **of contributors:** 723	# **of pull requests:** 5980
Summary: NumPy is a very popular python package for scientific computing. Half of the content in the repository is C files and the other half is python files. NumPy has an extensive contribution guide, that introduces a requierd development workflow, the release policy, project governance etc.	

Release	NB	LR	DT	RF	Best
1.12 - 1.13	JM2	JM2	MO	JM2	**JM2**
1.13 - 1.14	MU	MU	MU	JM	**MU**
1.14 - 1.15	JM	MU	MO	JM2	**MU**

7.3.4 Plotly

Plotly https://github.com/plotly/plotly.js	
# **of commits:** 17647	# **of issues:** 2022
# **of contributors:** 108	# **of pull requests:** 1463
Summary: Plotly.js is an open source JavaScript charting library maintained by the company, Plotly. The project has a set of contribution guidelines that states how new issues should be structured, as well as how to get started with developing on the project. The guidelines do not enforce any specific development workflow.	

Release	NB	LR	DT	RF	Best
1.38 - 1.39	JM2	ALL	MU	JM2	**ALL**
1.39 - 1.40	MU	MU	ALL	MU	**MU**
1.40 - 1.41	MU	ALL	JM	JM2	**ALL**

7.3.5 Yii

Yii https://github.com/yiisoft/yii2	
# **of commits:** 18796	# **of issues:** 10279
# **of contributors:** 948	# **of pull requests:** 6628
Summary: Yii is a PHP web framework used to ease the development of MVC web applications.	

Release	NB	LR	DT	RF	Best
2.0.9 - 2.0.10	MO	MO	MO	MU	**MO**
2.0.10 - 2.0.11	MU	MU	MU	JM2	**MU**
2.0.11 - 2.0.12	MU	MU	MU	MO	**MU**

7.3.6 Terraform

Terraform https://github.com/hashicorp/terraform	
# **of commits:** 23302	# **of issues:** 11414
# **of contributors:** 1236	# **of pull requests:** 8573
Summary: Terraform is a tool used to plan and build data center infrastructure as code. The project is created by HashiCorp and is written in Go.	

Release	NB	LR	DT	RF	Best
0.8.0 - 0.9.0	JM2	MO	ALL	MU	**ALL**
0.9.0 - 0.10.0	MU	JM2	MU	MU	**MU**

7.3.7 Ansible

Ansible https://github.com/ansible/ansible	
# **of commits:** 42268	# **of issues:** 21340
# **of contributors:** 4123	# **of pull requests:** 30025
Summary: Ansible is a configuration management and deployment tool used for various automation tasks. It is written in Python.	

Release	NB	LR	DT	RF	Best
2.4.0 - 2.5.0	MO	MO	MU	MU	**MO**
2.5.0 - 2.6.0	JM2	MU	JM	MU	**JM**

7.3.8 Symfony

Symfony https://github.com/symfony/symfony	
# **of commits:** 40020	# **of issues:** 11510
# **of contributors:** 1767	# **of pull requests:** 18474
Summary: Symnfony is like Yii, a PHP web application framework. Is is written in PHP.	

Release	NB	LR	DT	RF	Best
3.0.0 - 3.1.0	MO	JM2	MU	JM	**JM2**
3.1.0 - 3.2.0	MO	MO	MU	MU	**MO**
3.2.0 - 3.3.0	ALL	ALL	JM	MU	**ALL**

7.3.9 Godot

Godot https://github.com/godotengine/godot	
# **of commits:** 18856	# **of issues:** 15155
# **of contributors:** 788	# **of pull requests:** 9383
Summary: Godot is a 2D and 3D game engine mainly written in C++. The release policy and versioning scheme used by Godot result in data sets that are poorly defined in terms of bug prediction. As a result, only release 2.0.0 - 2.1.0 have been analyzed.	

Release	NB	LR	DT	RF	Best
2.0.0 - 2.1.0	MU	JM2	ALL	ALL	**MU**

7.3.10 Elasticsearch

Elasticsearch https://github.com/elastic/elasticsearch	
# **of commits:** 44060	# **of issues:** 18701
# **of contributors:** 1145	# **of pull requests:** 19102
Summary: Elasticsearch is a disbributed RESTful search engine based on the Apache Lucene library. It is written in Java.	

Release	NB	LR	DT	RF	Best
6.0.0 - 6.1.0	ALL	ALL	JM2	JM2	**ALL**
6.1.0 - 6.2.0	JM	MU	MU	MU	**MU**
6.2.0 - 6.3.0	JM	MU	MU	MU	**MU**
6.3.0 - 6.4.0	MU	JM	ALL	MU	**ALL**

7.4 Discussion

This section provides a discussion of the collected data from the different repositories. First, an evaluation of how the metric sets perform *intra-repository* i.e. across the releases in the same repository is provided followed by an evaluation of the *inter-repository* performance, that is, how the metric sets perform across the repositories.

When assessing the results for each of the repositories, it is clear, that there is not one repository, where the same metric set has performed best across *all* releases. To synthesize the results from the previous pages, Table 7.6 shows the number of releases across all repositories, where each metric set has the highest F1 score.

Metric set	# of highest F1
MU	11
JM	4
MO	7
JM2	3
ALL	6

Table 7.6: Number of runs where each metric set scored the highest F1 score

This clearly shows that one metric set cannot be claimed to yield the best performance across all releases and repositories.

Different combinations of metrics prove to perform the best across different releases, while there are some indications that, for some repositories, a metric set is recurring across some of the releases. This is the case for Eclipse JDT UI, NumPy, Plotly, Yii and Elasticsearch. With regard to the repositories, Eclipse JDT Core, Terraform, Ansible and Symfony, the results do not indicate that one metric set is the best across *all* releases.

In order to evaluate, how the metric sets have performed inter-repository, the arithmetic mean of the F1 scores for all releases within a project has been calculated. In addition, the mean of the F1 scores for all releases with the best metric sets has been calculated. To give an example, the average performance of the JM2 metric set in Eclipse JDT Core has been calculated by averaging the best F1 score of the JM2 metric set across all releases in Eclipse JDT Core. The overall best average in Eclipse JDT Core has been calculated by averaging the best F1 score of all metric sets across all releases. The numbers are seen in Table 7.7

Project	**MU**	**JM**	**MO**	**JM2**	**ALL**	**Best**
Eclipse JDT Core	54.69	54.65	53.57	53.26	53.17	**55.76**
Eclipse JDT UI	35.81	35.70	35.87	35.07	34.94	**36.68**
NumPy	58.41	58.88	58.21	58.52	57.06	**59.76**
Plotly	46.51	42.93	44.11	43.37	46.11	**46.52**
Yii	41.54	40.72	42.52	38.07	39.12	**43.52**
Terraform	24.72	20.33	20.17	22.31	25.34	**25.81**
Ansible	53.68	53.37	54.66	53.82	53.16	**55.32**
Symfony	46.59	47.32	48.10	47.64	48.01	**48.82**
Godot	49.72	43.51	46.37	43.92	46.98	**49.72**
Elasticsearch	25.59	21.50	21.79	20.64	25.94	**26.66**

Table 7.7: The arithmetic mean of the F1 scores for all metric sets across repositories. The column, *Best* is calculated using the best F1 score from each release

These numbers reveal that there is generally not a big difference in the performance when using one specific metric set for all releases compared to the best metric sets. The highest observed difference is on the Godot project using the *JM* set with a difference 6.21 percentage point. This confirms, that metric sets that generally perform well in one repository also have a good prediction power

when compared to the best achieved performance of the other sets. There is only a small number of percentage points in difference.

Table 7.7 also show, that the metric sets, *MU* and *MO* which have proven to perform well on the Eclipse project, generally perform equally when transferred to the open source projects analyzed in this thesis. Looking at the actual values, however, show that the overall prediction power is not impressive. The highest F1 score achieved for a single repository including all releases is on the NumPy project with an average F1 score at 59.76 %

The highest observed F1 score achieved across all releases and repositories in this experiment is 64.74 % with *MU* and the random forest classifier on the Eclipse JDT Core 3.0 - 3.1 data set. The highest observed recall is 98.75 % with *JM2* and the naïve Bayes classifier on the Numpy 1.12 - 1.13 data set.

The results can be compared with the original studies of the included metric sets. Moser et al. did not report precision or F1 scores, but instead, they mainly focused on the recall score. They analyzed the Eclipse core project on the releases 2.0, 2.1 and 3.0. Without the use of cost sensitive analysis, they scored a recall of 30%, 47%, and 65% for the NB, LR, and DT classifiers respectively. With the cost-sensitive analysis, which penalized the false negatives errors, they scored a recall of 83% with DT. The recall on Eclipse JDT Core 3.0 in this project was 42%, 64%, 64%, and 58% for NB, LR, DT, and RF respectively. The recall of NB and LR is higher in this thesis, while the recall of DT is lower.

Muthukumaran reported precision, recall and F1, however, they calculated these evaluation metrics differently, as they calculated the average recall using both the positive and the negative labels. This differs from the broadly used definition of precision, recall and F1 in binary classification, where only the positive label is considered. To this end, it does not make sense to compare the results between the studies.

7.5 Limitations and threats to validity

One of the main threats to the validity of these results, is the reliability of the extracted data. There is a great source of an erroneous extraction of the ground truth i.e. whether a commit is a bug fix or not, due to (1) the labelling of issue and pull requests in GitHub are evidently not consistent across all repositories and (2) that `msrlib` only supports inferring the ground truth from the first direct issue reference. With a more sophisticated extraction of the ground truth, the results might show different. In addition, it cannot be guaranteed that the

extraction of the process metrics that are based on other developers ability to be consistent are not flawed. For instance, not all commits that mainly involved refactoring may have been identified as the *REFACTOR* metric relies solely on the *refactor* keyword in the commit message. Not all in development bugs may have been extracted either due the before-mentioned labelling inconsistencies. With regard to the extraction of the other process metrics, it cannot be completely guaranteed that it is flawless. The general extraction of the process metrics have been assessed extensively, but as with any software implementation, there are be edge cases that might not have been covered.

In general, the generalization on other types of projects cannot be claimed. As development processes vary to a great extent and the process metrics used so far only describe a few aspects of the process, the results might be very different on other repositories with different characteristics than the ones used in this thesis. Closed source software, software in other industries etc. might yield other results.

The classification models used in this thesis have been built with simple algorithms from the machine learning theory. The reason for this was, that these algorithms have yielded promising results in other studies, however the results obtained using these classifiers cannot be claimed to be the same for more complex learners e.g. neural networks. The combinations of metrics might perform differently using other mathematical models.

The metrics have been evaluated in different combinations. These combinations did either originate from other studies or have been proposed in this work by analyzing the correlations of the process metrics combined with domain knowledge and intuition. For this reason, the results and conclusions obtained cannot be generalized for all possible sets of known process metrics. It cannot be concluded based on this thesis, that there does not exist a single set of metrics that perform satisfyingly well across open source projects. Future work could include a data based feature selection approach e.g. by using recursive feature elimination across open source projects to conclude, which process metrics would be the most powerful predictors.

Chapter 8

Conclusion

The software development process is a complex structure of many variables that vary greatly across software projects. Predicting defective parts of a software system by extracting and gaining knowledge from the change process have in earlier studies proven to be possible. One of the current approaches is to extract process metrics from software repositories, and it has also been shown, that for some projects, process metrics are stronger indicators of defective software than product related metrics for instance expressing code complexity. However, there is little knowledge about, how these process metrics perform when they are transferred into open source projects with software processes that are anything but similar.

Over the course of this thesis, a software system has been realized comprised of a bug prediction library and a research and visualization component. The bug prediction library, `msrlib` has been implemented so that it can be used to extract and gain knowledge about the change history from any publicly available GitHub-hosted repository. By using proven machine learners, the library is able to learn from historic data, to predict the presence of defective files in a software system.

Empirical software engineering research have been conducted, by creating an experiment that extracted data from 10 very popular and different software repositories hosted on GitHub. The data sets from the different releases across

the repositories were used to build classification models to predict bugs within a repository. Different sets of process metrics were used for the predictions. The results showed, that one set of process metrics did not prove to be the best across the all repositories. Moreover, it was found, that while there may not be a *best* set of metrics, the difference in performance between the different sets was considerably low, and they did generally perform fairly equal. Lastly, the experiment showed, that the metric sets that have proven powerful on the Eclipse project did not yield noteworthy results across the open source projects used in this experiment. However, there is still plenty of research to do in the field of bug prediction with process metrics. This thesis did not use a data-based feature selection approach, and there might therefore exist better combinations of process metrics that describe the relationship between the development process and the introduction of bugs in a better way within each project. Moreover, the approach used in this thesis was to create a generalized bug prediction system, that could predict bugs across many GitHub-based repositories. The classifiers for each release and each repository was not tuned in any way, and as a result, it is very likely, that tuning of hyperparameters and the inclusion of more sophisticated learners will yield higher scores.

Appendices

Appendix A

Physical data model

A.1 Post-release commits (JSON schema)

```
{
    "author": "Martin Aeschlimann",
    "datetime": "2018-10-18 09:42:43",
    "files": [
        {
            "filename": "src/vs/workbench/services/textfile/common/textFileService.ts",
            "is_bugfix": true,
            "sha": "766d4e482b38b66dcbc675a275bdccd0a5e6764d"
        }
    ],
    "message": "Save file dialog initialized with the filesystem root. Fixes #60939",
    "sha": "0ca196a5cd4af9529334348f44873b7868b53b90"
}
```

Listing 3: Example of an extracted post-release commit

A.2 Preprocessed data set (JSON schema)

```
[
    {
        "ADD": 1008,
        "AUTHORS": 4,
        "AVGADD": 201.6,
        "AVGCHSET": 299.0,
        "AVGDEL": 175.6,
        "AVGTBCH": 39118.5,
        "CMPRERE": 1,
        "COMMITS": 5,
        "DELE": 878,
        "ENTROPY": 0.7219280948873623,
        "INDEVBUGS": 0,
        "LASTCM": 36141,
        "MAXADD": 219,
        "MAXCHSET": 299,
        "MAXDEL": 178,
        "MPOCH": 0.4,
        "NOCMMSG": 0,
        "REFACTOR": 0,
        "bug_prone": 1,
        "filename": "build/lib/standalone.ts"
    },
    {
        "ADD": 1304,
        "AUTHORS": 10,
        (..)
        "bug_prone": 0,
        "filename": "build/lib/stats.js"
    },
    ..
]
```

Listing 4: Example of a preprocessed data set

A.3 Validation report (JSON schema)

```
{
[
    {
        "runs": {
            "DecisionTree": {
                "confusionMatrix": {
                    "fn": 97,
                    "fp": 101,
                    "tn": 1053,
                    "tp": 76
                },
                "scores": {
                    "f1": 0.6791870001795396,
                    "precision": 0.6625779192114636,
                    "recall": 0.6722180576378478
                }
            },
            "LogisticRegression": {
                "confusionMatrix": {
                    "fn": 130,
                    "fp": 19,
                    "tn": 1135,
                    "tp": 43
                },
                "scores": {
                    "f1": 0.639644007031624,
                    "precision": 0.78785134015486,
                    "recall": 0.606422426041881
                }
            }
            // ..
        }
    },
}
```

Listing 5: Example of a validation report

Appendix B

Configuration

```
1  ##################################
2  #       General settings         #
3  ##################################
4  # Required
5  MODE = 'full'
6
7  # Required
8  RUN_TITLE = 'eclipse_jdt_core_3.0-3.1'
9
10 # Required
11 OUTPUT_DIR = '~/dev/master-thesis'
12
13 ##################################
14 #       Extraction settings      #
15 ##################################
16
17 # Required
18 REPOSITORY = {
19     "USERNAME": "eclipse",
20     "REPOSITORY_NAME": "eclipse.jdt.core",
21     "CREDENTIALS": {
22         "USERNAME": "foo",
23         "PASSWORD": "bar"
```

```
    }
}

# Required
START_DATE  = "2004-06-25"
END_DATE    = "2005-06-27"

# Required
DATETIME_PATTERN = '\%Y-\%m-\%d'

# Required
MAX_FIX_TIME = 'auto'

# Required
BUG_REPOSITORY = 'bugzilla'

# Required
BUGZILLA_URL = "https://bugs.eclipse.org/bugs/xmlrpc.cgi"

# Required
BUGZILLA_DATETIME_PATTERN = '\%Y\%m\%dT\%H:\%M:\%S'

# Required
BUGZILLA_RESOLUTION = 'FIXED'

#
INCLUDED_FILES = [
    "java",
    "ts",
    "cs",
    "js",
    "c",
    "py",
    "cpp",
    "php",
    "go",
]

# Required
# ISSUE_REGEX = '[#](\d+)'  # Github "#362"
ISSUE_REGEX = r'(\d{5,6}\b)'  # BugZilla 5-6 digits

# Required
BUGFIX_KEYWORDS = [
    "bug",
    "fix"
```

```
]

# Required
IGNORE_PULL_REQUESTS = False

# Required
FILTER_ISSUES_BY_LABEL = True

# Required
BUG_LABELS = [
    "bug"
]

API_REQUEST_DELAY = 0

#################################
#       Preprocessing settings  #
#################################

# Required
REFACTOR_KEYWORDS = [
    "refactor"
]

# Required
CMPRERE_THRESHOLD = 10

# Required
ENTROPY_PERIODS = 10

# Required
MPOCH_PERIODS = 10

#################################
#       Validation settings     #
#################################

# Required
CLASSIFIERS = [
    "NaiveBayes",
    "LogisticRegression",
    "DecisionTree",
    "RandomForest"
]

# Required
```

```
116 CROSS_VALIDATION_FOLDS = 10
117
118 # Required
119 FEATURE_SETS = {
120     "MO": [
121         "COMMITS",
122         "REFACTOR",
123         "INDEVBUGS",
124         "AUTHORS",
125         "ADD",
126         "MAXADD",
127         "AVGADD",
128         "DELE",
129         "MAXDEL",
130         "AVGDEL",
131         "CCHURN",
132         "MAXCCHURN",
133         "AVGCCHURN",
134         "MAXCHSET",
135         "AVGCHSET"
136     ],
137     "MU": [
138         "COMMITS",
139         "ADD",
140         "DELE",
141         "AUTHORS",
142         "CMPRERE",
143         "LASTCM",
144         "INDEVBUGS",
145         "ENTROPY",
146         "MPOCH",
147         "MAXCHSET",
148         "AVGCHSET"
149     ],
150     "JM": [
151         "COMMITS",
152         "AUTHORS",
153         "AVGADD",
154         "AVGDEL",
155         "AVGCCHURN",
156         "INDEVBUGS",
157         "AVGTBCH",
158         "ENTROPY",
159         "AVGCHSET"
160     ],
161     "JM2": [
```

```
        "COMMITS",
        "AUTHORS",
        "MAXADD",
        "MAXCHSET"
    ],
    "ALL": [
        "COMMITS",
        "ADD",
        "DELE",
        "AUTHORS",
        "CMPRERE",
        "LASTCM",
        "REFACTOR",
        "INDEVBUGS",
        "ENTROPY",
        "MPOCH",
        "MAXCHSET",
        "AVGCHSET",
        "MAXADD",
        "AVGADD",
        "MAXDEL",
        "AVGDEL",
        "AVGTBCH",
        "NOCMMSG",
        "CCHURN",
        "MAXCCHURN",
        "AVGCCHURN"
    ]
}

#################################
#       Prediction settings     #
#################################

# Required
PRED_START_DATE = '2005-06-25'

# Required
PRED_END_DATE = '2006-06-29'

# Required
PRED_CLASSIFIER = 'RandomForest'

# Required
PRED_FEATURES = [
    "COMMITS",
```

```
    "AUTHORS",
    "MAXADD",
    "MAXCHSET"
]

#################################
#       Miscellaneous           #
#################################

# Required
ALL_FEATURES = [
        "COMMITS",
        "ADD",
        "DELE",
        "AUTHORS",
        "CMPRERE",
        "LASTCM",
        "REFACTOR",
        "INDEVBUGS",
        "ENTROPY",
        "MPOCH",
        "MAXCHSET",
        "AVGCHSET",
        "MAXADD",
        "AVGADD",
        "MAXDEL",
        "AVGDEL",
        "AVGTBCH",
        "NOCMMSG",
        "CCHURN",
        "MAXCCHURN",
        "AVGCCHURN"
    ]
```

Appendix C

Data selection

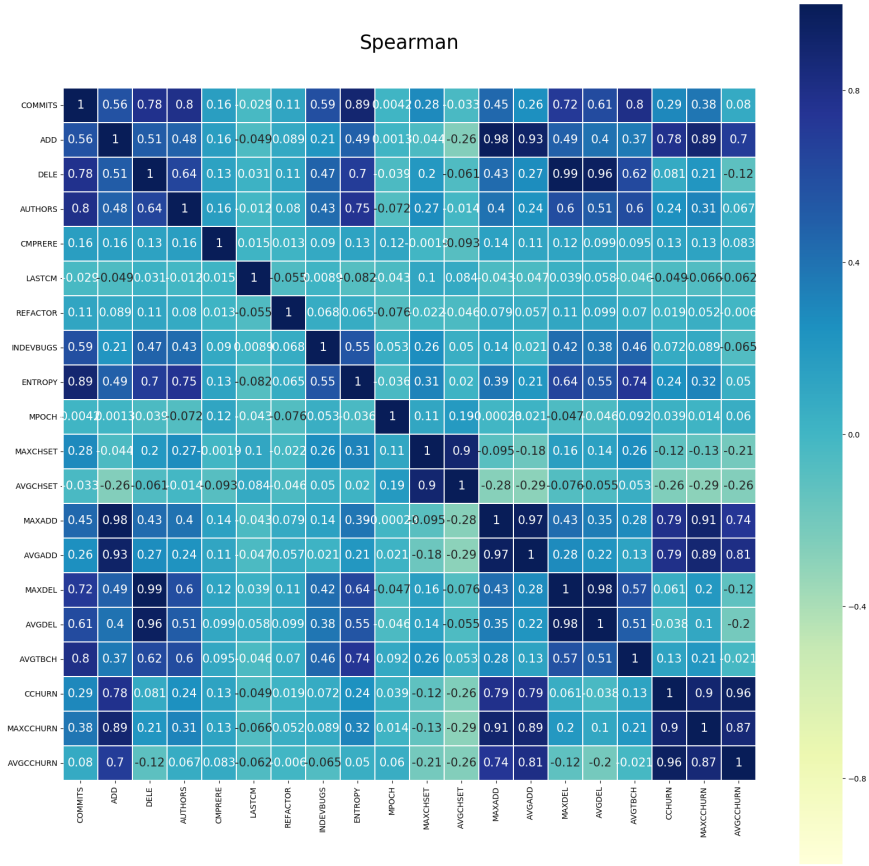
Name	URL	Commits
AspNetCore	https://github.com/aspnet/AspNetCore	37073
mpv	https://github.com/mpv-player/mpv	46701
openssl	https://github.com/openssl/openssl	23286
radare2	https://github.com/radare/radare2	20573
numpy	https://github.com/numpy/numpy	19614
cphalcon	https://github.com/phalcon/cphalcon	10484
akka	https://github.com/akka/akka	22965
plotly.js	https://github.com/plotly/plotly.js	17632
salt	https://github.com/saltstack/salt	102504
spree	https://github.com/spree/spree	19223
NodeBB	https://github.com/NodeBB/NodeBB	17903
cypress	https://github.com/cypress-io/cypress	11506
ccxt	https://github.com/ccxt/ccxt	20197
jquery-mobile	https://github.com/jquery/jquery-mobile	13109
eos	https://github.com/EOSIO/eos	10013
deeplearning4j	https://github.com/deeplearning4j/deeplearning4j	23769
matomo	https://github.com/matomo-org/matomo	25529
roslyn	https://github.com/dotnet/roslyn	41706
wp-calypso	https://github.com/Automattic/wp-calypso	32686
react-native-macos	https://github.com/ptmt/react-native-macos	12983
coreclr	https://github.com/dotnet/coreclr	19609
tutorials	https://github.com/eugenp/tutorials	11476
celery	https://github.com/celery/celery	10730
incubator-weex	https://github.com/apache/incubator-weex	11152
curl	https://github.com/curl/curl	23904
apollo	https://github.com/ApolloAuto/apollo	10369

amphtml	https://github.com/ampproject/amphtml	11926
odoo	https://github.com/odoo/odoo	122755
crystal	https://github.com/crystal-lang/crystal	11146
yii2	https://github.com/yiisoft/yii2	18793
generator-jhipster	https://github.com/jhipster/generator-jhipster	16916
cocos2d-x	https://github.com/cocos2d/cocos2d-x	36873
phpunit	https://github.com/sebastianbergmann/phpunit	11369
realm-cocoa	https://github.com/realm/realm-cocoa	10785
metabase	https://github.com/metabase/metabase	17352
selenium	https://github.com/SeleniumHQ/selenium	23367
servo	https://github.com/servo/servo	36000
ipython	https://github.com/ipython/ipython	23697
mattermost-server	https://github.com/mattermost/mattermost-server	10610
pandoc	https://github.com/jgm/pandoc	11970
homebrew-cask	https://github.com/Homebrew/homebrew-cask	82408
elixir	https://github.com/elixir-lang/elixir	15279
libgdx	https://github.com/libgdx/libgdx	13509
metasploit-framework	https://github.com/rapid7/metasploit-framework	50254
terraform	https://github.com/hashicorp/terraform	23285
cockroach	https://github.com/cockroachdb/cockroach	36033
influxdb	https://github.com/influxdata/influxdb	28275
corefx	https://github.com/dotnet/corefx	31390
CNTK	https://github.com/Microsoft/CNTK	16029
hhvm	https://github.com/facebook/hhvm	30162
framework	https://github.com/laravel/framework	22937
brew	https://github.com/Homebrew/brew	17769
emscripten	https://github.com/emscripten-core/emscripten	19040
mongoose	https://github.com/Automattic/mongoose	10235
pandas	https://github.com/pandas-dev/pandas	18769
vagrant	https://github.com/hashicorp/vagrant	11821
godot	https://github.com/godotengine/godot	18744
grpc	https://github.com/grpc/grpc	36590
sentry	https://github.com/getsentry/sentry	24086
symfony	https://github.com/symfony/symfony	39917
julia	https://github.com/JuliaLang/julia	43979
DefinitelyTyped	https://github.com/DefinitelyTyped/DefinitelyTyped	56716
ember.js	https://github.com/emberjs/ember.js	17908
home-assistant	https://github.com/home-assistant/home-assistant	17000
Rocket.Chat	https://github.com/RocketChat/Rocket.Chat	16083
rethinkdb-legacy	https://github.com/rethinkdb/rethinkdb-legacy	33387
go-ethereum	https://github.com/ethereum/go-ethereum	10468
etcd	https://github.com/etcd-io/etcd	14895
phaser	https://github.com/photonstorm/phaser	12058
pytorch	https://github.com/pytorch/pytorch	15855
fastlane	https://github.com/fastlane/fastlane	14834
pdf.js	https://github.com/mozilla/pdf.js	11378
spring-framework	https://github.com/spring-projects/spring-framework	17732
grafana	https://github.com/grafana/grafana	19407
foundation-sites	https://github.com/zurb/foundation-sites	16309

legacy-homebrew	https://github.com/Homebrew/legacy-homebrew	63881
brackets	https://github.com/adobe/brackets	17702
neovim	https://github.com/neovim/neovim	12491
opencv	https://github.com/opencv/opencv	25865
babel	https://github.com/babel/babel	12169
scikit-learn	https://github.com/scikit-learn/scikit-learn	23599
storybook	https://github.com/storybooks/storybook	15728
rust	https://github.com/rust-lang/rust	88758
spring-boot	https://github.com/spring-projects/spring-boot	20123
ansible	https://github.com/ansible/ansible	42187
jekyll	https://github.com/jekyll/jekyll	10297
bitcoin	https://github.com/bitcoin/bitcoin	19350
elasticsearch	https://github.com/elastic/elasticsearch	43956
meteor	https://github.com/meteor/meteor	21792
ant-design	https://github.com/ant-design/ant-design	13604
rails	https://github.com/rails/rails	71764
TypeScript	https://github.com/Microsoft/TypeScript	26335
angular	https://github.com/angular/angular	12712
youtube-dl	https://github.com/rg3/youtube-dl	16758
kubernetes	https://github.com/kubernetes/kubernetes	73729
atom	https://github.com/atom/atom	35947
three.js	https://github.com/mrdoob/three.js	26147
flutter	https://github.com/flutter/flutter	12978
moby	https://github.com/moby/moby	36290
go	https://github.com/golang/go	39068
node	https://github.com/nodejs/node	25463
vscode	https://github.com/Microsoft/vscode	44912
electron	https://github.com/electron/electron	21070
react-native	https://github.com/facebook/react-native	15546
tensorflow	https://github.com/tensorflow/tensorflow	47597
react	https://github.com/facebook/react	10613
bootstrap	https://github.com/twbs/bootstrap	18357
freeCodeCamp	https://github.com/freeCodeCamp/freeCodeCamp	20854

Appendix D

Correlation heat map


Spearman
COMMITS 1 0.56 0.78 0.8 0.16 -0.029 0.11 0.59 0.89 0.0042 0.28 -0.033 0.45 0.26 0.72 0.61 0.8 0.29 0.38 0.08
ADD 0.56 1 0.51 0.48 0.16 -0.049 0.089 0.21 0.49 0.0013 0.044 -0.26 0.98 0.93 0.49 0.4 0.37 0.78 0.89 0.7
DELE 0.78 0.51 1 0.64 0.13 0.031 0.11 0.47 0.7 -0.039 0.2 -0.061 0.43 0.27 0.99 0.96 0.62 0.081 0.21 -0.12
AUTHORS 0.8 0.48 0.64 1 0.16 -0.012 0.08 0.43 0.75 -0.072 0.27 -0.014 0.4 0.24 0.6 0.51 0.6 0.24 0.31 0.067
CMPRERE 0.16 0.16 0.13 0.16 1 0.015 0.013 0.09 0.13 0.12 -0.001 -0.093 0.14 0.11 0.12 0.099 0.095 0.13 0.13 0.083
LASTCM -0.029 -0.049 0.031 -0.012 0.015 1 -0.053 0.0089 0.082 0.043 0.1 0.084 -0.043 -0.047 0.039 0.058 -0.046 -0.049 -0.066 -0.062
REFACTOR 0.11 0.089 0.11 0.08 0.013 -0.055 1 0.068 0.065 -0.076 0.022 0.046 0.079 0.057 0.11 0.099 0.07 0.019 0.052 -0.006
INDEVBUGS 0.59 0.21 0.47 0.43 0.09 0.0089 0.068 1 0.55 0.053 0.26 0.05 0.14 0.021 0.42 0.38 0.46 0.072 0.089 -0.065
ENTROPY 0.89 0.49 0.7 0.75 0.13 -0.082 0.065 0.55 1 -0.036 0.31 0.02 0.39 0.21 0.64 0.55 0.74 0.24 0.32 0.05
MPOCH 0.0042 0.0013 -0.039 0.072 0.12 -0.043 -0.076 0.053 -0.036 1 0.11 0.19 0.00028 -0.021 -0.047 -0.046 0.092 0.039 0.014 0.06
MAXCHSET 0.28 -0.044 0.2 0.27 -0.0019 0.1 -0.022 0.26 0.31 0.11 1 0.9 -0.095 -0.18 0.16 0.14 0.26 -0.12 -0.13 -0.21
AVGCHSET -0.033 -0.26 -0.061 -0.014 -0.093 0.084 -0.046 0.05 0.02 0.19 0.9 1 -0.28 -0.29 -0.076 -0.055 0.053 -0.26 -0.29 -0.26
MAXADD 0.45 0.98 0.43 0.4 0.14 -0.043 0.079 0.14 0.39 0.00028 -0.095 -0.28 1 0.97 0.43 0.35 0.28 0.79 0.91 0.74
AVGADD 0.26 0.93 0.27 0.24 0.11 -0.047 0.057 0.021 0.21 0.021 -0.18 -0.29 0.97 1 0.28 0.22 0.13 0.79 0.89 0.81
MAXDEL 0.72 0.49 0.99 0.6 0.12 0.039 0.11 0.42 0.64 -0.047 0.16 -0.076 0.43 0.28 1 0.98 0.57 0.061 0.2 -0.12
AVGDEL 0.61 0.4 0.96 0.51 0.099 0.058 0.099 0.38 0.55 -0.046 0.14 -0.055 0.35 0.22 0.98 1 0.51 -0.038 0.1 -0.2
AVGTBCH 0.8 0.37 0.62 0.6 0.095 -0.046 0.07 0.46 0.74 0.092 0.26 0.053 0.28 0.13 0.57 0.51 1 0.13 0.21 -0.021
CCHURN 0.29 0.78 0.081 0.24 0.13 -0.049 0.019 0.072 0.24 0.039 -0.12 -0.26 0.79 0.79 0.061 -0.038 0.13 1 0.9 0.96
MAXCCHURN 0.38 0.89 0.21 0.31 0.13 -0.066 0.052 0.089 0.32 0.014 -0.13 -0.29 0.91 0.89 0.2 0.1 0.21 0.9 1 0.87
AVGCCHURN 0.08 0.7 -0.12 0.067 0.083 -0.062 -0.006 0.065 0.05 0.06 -0.21 -0.26 0.74 0.81 -0.12 -0.2 -0.021 0.96 0.87 1
COMMITS ADD DELE AUTHORS CMPRERE LASTCM REFACTOR INDEVBUGS ENTROPY MPOCH MAXCHSET AVGCHSET MAXADD AVGADD MAXDEL AVGDEL AVGTBCH CCHURN MAXCCHURN AVGCCHURN
0.8
0.4
0.0
−0.4
−0.8

Appendix E

Classifier performance

This appendix synthesizes the complete results generated by the classifier performance evaluation across all repositories.

E.1 Eclipse JDT Core

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
3.0 - 3.1	MU	70.74	40.68	51.43	53.96	69.47	60.38	61.81	57.43	60.12	73.72	57.23	64.74
	JM	61.73	56.12	58.57	38.75	73.27	49.9	61.33	58.08	59.91	73.0	50.63	61.18
	MO	72.81	38.15	49.71	53.74	65.03	58.89	61.78	65.27	62.68	72.62	51.05	60.09
	JM2	68.26	42.15	51.95	55.03	63.74	58.96	62.18	64.01	62.84	70.59	58.06	63.65
	ALL	76.8	37.52	50.13	47.69	67.56	55.35	62.01	57.23	58.6	72.9	54.25	62.45
3.1 - 3.2	MU	67.68	34.62	45.64	50.46	72.31	58.86	50.42	51.54	50.02	70.34	41.03	50.69
	JM	64.8	44.36	52.34	49.34	72.56	58.8	48.05	44.1	44.6	57.37	37.18	43.78
	MO	66.98	27.69	38.76	51.3	70.26	59.26	52.1	49.49	50.55	62.07	43.59	52.5
	JM2	70.79	40.0	50.75	50.69	67.18	57.58	53.35	50.77	52.34	58.54	46.15	52.58
	ALL	67.03	27.95	39.02	51.7	71.28	59.2	53.67	52.31	51.29	68.32	42.31	52.68
3.2 - 3.3	MU	52.44	23.8	32.17	42.17	68.27	52.16	37.82	39.35	37.06	49.92	24.78	35.46
	JM	48.86	57.13	51.48	38.66	72.42	50.95	35.45	38.6	35.97	46.66	27.62	34.02
	MO	47.53	34.43	38.4	39.61	63.75	48.48	36.64	37.82	36.62	41.76	26.68	31.52
	JM2	59.47	29.53	39.0	43.04	56.28	48.39	38.85	38.57	37.99	43.56	31.25	39.62
	ALL	47.67	34.85	38.31	38.86	68.28	50.01	34.46	34.93	35.59	52.34	25.95	33.59
3.3 - 3.4	MU	62.32	37.69	46.91	43.84	74.62	54.8	48.03	45.77	46.37	56.93	35.77	43.44
	JM	56.46	42.31	48.19	41.01	80.0	54.25	42.35	43.08	42.41	54.78	36.92	39.97
	MO	58.34	32.31	41.34	43.53	71.15	53.77	41.64	39.62	39.45	50.2	32.31	37.89
	JM2	59.93	39.62	47.57	45.46	70.77	55.1	40.7	39.62	40.38	49.9	34.62	42.57
	ALL	57.06	30.38	39.39	41.78	76.54	53.77	48.31	47.69	46.72	60.08	35.38	45.04
3.4 - 3.5	MU	62.02	31.87	41.37	31.0	68.05	42.87	39.22	39.74	38.38	67.34	24.41	34.32
	JM	47.41	49.71	47.56	32.46	68.01	43.0	39.64	37.35	39.31	62.9	27.5	37.37
	MO	59.13	31.87	40.87	34.36	60.7	43.67	34.26	37.21	37.19	62.49	24.34	36.1
	JM2	57.46	31.91	40.25	33.03	57.61	41.6	35.01	38.64	34.74	43.46	35.51	33.45
	ALL	60.08	31.21	40.41	29.76	66.91	40.34	42.04	36.25	38.3	53.84	24.49	34.23

Table E.1: Eclipse JDT Core

E.2 Eclipse JDT UI

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
3.0 - 3.1	MU	38.01	43.1	40.24	23.77	70.8	35.6	28.2	31.18	29.35	40.19	18.44	26.4
	JM	39.16	44.89	41.59	22.3	72.91	33.75	26.97	30.37	27.95	43.1	16.35	20.86
	MO	37.16	39.82	38.25	25.63	68.12	37.15	29.14	32.47	29.38	39.15	12.76	19.94
	JM2	37.1	41.31	38.87	24.55	66.63	35.75	23.2	29.18	26.06	27.72	19.63	22.92
	ALL	37.64	37.43	37.38	23.75	71.41	35.72	31.08	33.32	30.73	39.11	18.17	24.38
3.1 - 3.2	MU	39.11	29.71	33.67	27.9	64.57	38.92	31.46	30.57	32.03	53.67	19.43	26.75
	JM	42.96	23.71	30.42	28.24	71.14	40.18	29.78	31.14	29.97	56.65	17.43	30.02
	MO	37.62	26.0	30.66	28.87	62.86	39.45	33.97	33.43	35.3	48.54	16.86	22.63
	JM2	37.88	26.29	30.84	26.42	58.29	36.27	24.7	28.29	25.22	35.19	21.14	24.68
	ALL	36.94	22.86	28.2	27.13	69.71	38.72	31.99	32.29	32.49	58.11	20.29	27.98
3.2 - 3.3	MU	51.32	29.05	36.75	35.67	53.61	42.29	41.35	41.97	41.64	64.3	25.59	32.81
	JM	53.45	31.26	38.81	31.67	58.33	40.46	39.48	37.25	38.61	51.58	22.73	33.57
	MO	49.63	31.26	38.12	35.12	54.86	42.54	37.23	36.96	36.72	50.76	22.1	31.67
	JM2	54.31	30.6	38.53	35.17	52.36	41.73	36.21	38.79	37.35	43.74	29.06	30.48
	ALL	51.41	29.35	37.0	32.51	56.73	41.98	39.86	39.45	38.38	56.6	23.97	32.5
3.3 - 3.4	MU	34.85	35.92	35.14	24.31	67.98	35.76	27.32	25.37	24.62	53.67	16.32	23.73
	JM	39.44	31.18	33.73	22.0	66.73	32.89	22.78	24.15	22.3	45.83	13.9	14.15
	MO	32.31	36.54	34.16	24.91	64.3	35.81	26.39	26.47	26.49	41.08	15.18	20.94
	JM2	34.92	33.53	33.55	24.7	63.71	35.57	22.53	27.98	23.83	22.8	15.7	21.37
	ALL	35.07	34.78	34.72	23.35	68.57	34.84	27.83	28.16	27.96	42.95	13.86	22.76
3.4 - 3.5	MU	25.63	20.76	21.86	11.88	53.26	19.07	11.82	16.59	13.62	10.1	7.05	5.23
	JM	24.59	20.08	21.24	14.21	54.32	22.55	15.64	17.35	14.97	14.48	4.39	9.72
	MO	17.92	22.58	19.49	14.79	55.91	23.28	18.06	18.56	17.25	11.92	7.88	7.65
	JM2	26.27	18.26	20.47	14.79	51.67	22.91	10.4	21.74	15.07	8.15	9.77	9.72
	ALL	20.12	19.24	18.87	13.48	56.82	21.79	12.11	15.68	13.47	10.32	2.58	3.96

Table E.2: Eclipse JDT UI

E.3 NumPy

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
1.12 - 1.13	MU	35.33	96.25	51.48	50.83	55.0	53.17	49.3	47.32	45.59	61.21	42.86	46.52
	JM	35.8	96.07	52.11	52.76	61.79	55.23	51.83	49.64	50.36	69.83	43.21	50.53
	MO	35.18	97.5	51.63	52.7	64.29	54.76	56.58	54.82	54.15	61.68	40.54	42.07
	JM2	35.86	98.75	52.55	58.74	59.11	57.23	52.14	51.43	51.54	58.33	39.64	50.85
	ALL	35.33	97.5	51.81	48.79	57.68	51.56	57.44	50.54	48.27	65.91	40.71	43.35
1.13 - 1.14	MU	73.12	48.0	57.08	61.88	69.89	64.34	64.89	62.78	61.56	65.21	51.22	52.54
	JM	66.99	39.67	48.74	61.95	65.44	64.25	60.95	56.56	59.1	64.05	50.11	61.79
	MO	72.55	35.33	45.79	66.34	62.56	63.75	56.82	60.56	56.33	69.2	53.44	48.96
	JM2	71.11	39.44	49.7	65.31	65.44	63.94	46.39	47.67	47.96	56.54	45.78	52.98
	ALL	72.76	36.44	47.08	62.18	62.67	62.11	55.36	57.56	55.07	69.43	44.67	59.91
1.14 - 1.15	MU	61.62	40.14	47.23	47.02	72.78	57.71	41.51	42.22	40.62	55.66	39.86	49.4
	JM	45.91	74.86	56.38	48.75	71.25	57.16	51.45	47.36	47.95	57.14	42.36	49.08
	MO	34.94	84.72	48.58	48.27	67.92	56.12	54.92	52.22	51.35	61.48	38.61	48.89
	JM2	65.12	40.97	48.79	54.34	56.39	54.38	50.03	44.72	45.48	64.38	47.22	53.63
	ALL	33.73	82.08	46.98	48.86	72.92	57.26	50.17	48.61	48.53	60.88	38.61	50.99

Table E.3: NumPy

E.4 Plotly

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
1.38 - 1.39	MU	30.83	29.17	29.64	33.81	55.0	39.99	46.36	49.17	46.1	53.33	20.0	23.71
	JM	27.65	59.17	35.22	27.52	69.17	38.69	21.67	22.5	20.97	15.0	17.5	24.0
	MO	16.01	91.67	27.22	31.12	63.33	41.13	24.0	21.67	15.43	20.0	10.0	8.0
	JM2	41.42	40.83	39.18	26.74	54.17	33.53	41.5	36.67	39.51	48.33	32.5	36.63
	ALL	16.6	91.67	28.05	35.56	70.0	46.11	35.67	36.67	45.6	50.67	22.5	26.33
1.39 - 1.40	MU	29.09	82.14	42.89	37.97	62.86	47.76	40.14	37.62	36.04	56.01	26.43	36.42
	JM	24.81	93.81	39.21	32.75	71.43	45.08	36.28	41.9	37.71	52.95	30.0	30.95
	MO	23.96	94.29	38.18	35.8	65.71	46.63	37.17	39.76	35.27	41.81	17.38	23.77
	JM2	25.26	91.43	39.43	34.99	68.57	46.24	38.25	32.14	34.67	49.76	27.86	32.73
	ALL	23.93	94.29	38.14	38.69	64.29	45.68	40.13	42.38	40.34	53.69	25.71	30.08
1.40 - 1.41	MU	29.29	92.86	44.41	32.12	78.57	45.68	38.74	34.29	35.16	38.79	30.0	26.97
	JM	27.34	94.29	42.3	30.5	88.57	45.01	39.98	31.43	35.46	42.42	21.43	25.96
	MO	12.76	95.71	22.52	30.23	77.14	44.57	30.67	31.43	27.59	36.17	24.29	25.83
	JM2	28.64	88.57	43.12	31.51	78.57	44.36	32.65	40.0	33.47	36.17	37.14	36.22
	ALL	12.98	97.14	22.9	32.57	80.0	45.83	40.8	38.57	34.31	39.17	25.71	24.72

Table E.4: Plotly

E.5 Yii

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
2.0.9 - 2.0.10	MU	20.55	20.0	18.28	25.17	62.5	36.02	33.14	39.17	36.74	72.67	36.67	48.38
	JM	20.0	76.67	31.57	29.65	77.5	42.65	57.62	57.5	51.99	70.95	37.5	48.13
	MO	19.89	87.5	32.35	32.21	70.83	43.94	49.46	65.0	54.33	63.33	45.83	46.59
	JM2	0.0	0.0	0.0	17.05	69.17	27.19	38.53	47.5	40.65	47.67	39.17	38.43
	ALL	19.58	87.5	31.93	28.07	65.83	39.0	43.45	50.83	45.74	64.33	31.67	46.71
2.0.10 - 2.0.11	MU	32.8	35.24	32.67	25.73	62.62	36.0	37.6	24.52	30.04	50.0	12.62	9.32
	JM	32.01	30.95	30.06	20.36	65.95	30.84	16.67	16.9	18.58	10.0	4.29	11.0
	MO	35.0	26.19	27.33	24.37	56.67	33.93	17.56	16.43	14.33	17.5	1.43	4.0
	JM2	30.83	24.76	26.82	25.76	57.86	35.37	19.94	19.52	17.21	19.5	9.05	13.74
	ALL	43.36	29.29	32.14	24.78	56.67	31.94	25.32	29.29	24.71	25.0	10.71	13.48
2.0.11 - 2.0.12	MU	39.65	44.0	40.23	33.35	54.0	40.01	30.44	34.0	32.08	14.83	12.33	8.5
	JM	30.82	42.33	34.83	32.34	54.0	39.33	32.0	28.67	29.73	13.17	16.67	15.16
	MO	34.65	47.67	39.3	30.4	55.67	38.33	15.33	22.33	17.24	28.17	8.67	24.61
	JM2	44.02	35.33	38.08	29.55	55.67	38.18	21.15	24.0	22.28	27.08	10.0	16.59
	ALL	30.36	45.67	35.46	29.38	55.67	38.52	28.83	34.0	25.38	24.76	14.0	6.04

Table E.5: Yii

E.6 Terraform

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
0.8.0 - 0.9.0	MU	44.02	25.3	31.75	20.68	77.66	32.61	37.64	37.19	36.56	51.67	27.98	35.05
	JM	39.8	24.7	30.14	20.43	76.17	31.7	34.95	37.43	36.21	44.81	24.4	28.78
	MO	37.41	23.21	28.42	20.85	79.46	32.91	33.82	33.64	34.95	37.09	17.54	24.6
	JM2	48.18	26.21	33.53	20.88	53.29	29.95	31.32	33.64	33.14	38.9	25.9	27.45
	ALL	36.35	20.83	26.26	20.89	77.38	32.77	39.37	39.86	38.73	49.03	21.72	30.43
0.9.0 - 0.10.0	MU	6.97	33.5	11.51	3.11	66.0	5.93	7.84	36.5	12.88	5.24	16.5	9.18
	JM	1.72	73.5	3.32	2.32	56.0	4.45	1.17	20.0	4.17	0.19	2.5	0.58
	MO	0.91	90.0	1.8	2.81	63.5	5.38	0.98	17.5	2.1	0.0	2.5	0.0
	JM2	6.85	29.0	11.08	3.18	56.0	6.01	0.75	15.0	2.06	0.15	7.5	6.86
	ALL	0.94	92.5	1.85	2.54	56.0	5.14	7.51	34.0	11.94	4.42	14.0	6.84

Table E.6: Terraform

E.7 Ansible

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
2.4.0 - 2.5.0	MU	54.95	33.05	41.17	43.28	59.34	49.55	46.49	50.02	48.74	62.78	40.36	48.79
	JM	53.7	33.05	40.77	37.88	70.04	47.7	43.17	44.5	44.48	56.98	33.66	42.95
	MO	51.84	37.8	43.56	46.67	57.97	51.61	42.6	44.21	44.13	58.33	33.2	42.69
	JM2	57.13	30.6	39.68	46.12	57.06	50.97	41.36	47.11	43.42	47.24	37.32	41.41
	ALL	52.65	35.34	42.13	39.04	63.61	48.88	45.31	50.03	47.23	61.23	37.63	47.84
2.5.0 - 2.6.0	MU	63.66	31.92	42.2	53.9	56.12	54.87	56.8	60.12	57.81	64.15	51.8	57.56
	JM	65.31	32.82	43.38	51.89	57.2	54.41	58.29	57.41	59.03	61.11	50.35	55.94
	MO	62.49	32.85	42.91	55.33	53.78	54.53	57.08	57.44	57.7	60.06	47.31	53.29
	JM2	63.62	34.27	44.4	54.87	51.97	53.26	56.52	58.14	56.66	62.6	53.07	55.75
	ALL	63.71	30.31	40.85	52.43	57.2	54.02	56.03	57.04	56.89	65.15	49.63	57.43

Table E.7: Ansible

E.8 Symfony

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
3.0.0 - 3.1.0	MU	60.77	28.12	37.93	43.32	54.07	47.66	43.97	40.26	42.23	58.19	32.34	40.84
	JM	63.7	31.94	42.3	46.56	52.15	48.99	39.36	41.11	38.7	53.54	29.72	41.26
	MO	61.41	33.06	42.57	46.91	50.23	48.28	41.23	43.72	41.24	54.92	30.78	35.53
	JM2	59.11	29.67	39.18	48.61	51.75	49.87	38.78	39.52	38.39	48.44	35.33	40.67
	ALL	60.28	33.05	42.16	45.35	55.93	48.76	42.42	41.01	41.44	57.75	32.36	39.48
3.1.0 - 3.2.0	MU	56.8	33.33	41.67	41.03	55.02	46.77	40.43	41.53	41.15	49.68	29.21	35.85
	JM	57.18	33.04	41.62	39.15	58.17	46.5	40.46	38.97	40.33	49.81	29.23	32.2
	MO	54.62	35.26	42.62	44.02	55.01	48.75	39.34	40.28	39.39	46.1	23.55	32.99
	JM2	58.89	29.26	38.6	42.36	53.76	47.28	33.8	36.77	35.82	45.29	27.97	33.3
	ALL	52.94	34.94	41.89	40.65	58.48	47.43	39.88	40.27	39.4	55.46	25.8	33.41
3.2.0 - 3.3.0	MU	54.76	29.32	37.88	38.6	55.0	45.35	40.8	42.05	41.13	54.92	33.41	38.76
	JM	59.53	29.55	39.14	39.62	56.82	46.48	41.64	44.09	43.42	54.7	30.91	37.32
	MO	50.67	32.27	39.03	42.98	53.18	47.27	44.39	43.41	43.24	52.32	30.45	35.25
	JM2	57.51	29.09	38.01	41.66	51.14	45.77	41.03	41.82	41.55	47.37	33.64	36.26
	ALL	51.86	32.95	39.79	41.75	55.91	47.84	42.02	45.23	43.17	48.69	27.27	37.44

Table E.8: Symfony

E.9 Godot

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
2.0.0 - 2.1.0	MU	47.93	55.69	49.72	22.65	81.53	35.68	25.41	35.0	28.75	39.72	28.33	22.91
	JM	35.93	56.94	43.51	20.72	82.36	32.97	25.53	37.22	26.57	48.6	27.5	33.25
	MO	43.16	53.33	46.37	21.24	83.89	33.67	28.13	30.0	26.5	42.68	26.94	32.29
	JM2	47.56	46.53	43.92	27.49	67.22	38.75	26.66	29.31	27.03	46.79	28.89	26.71
	ALL	40.88	58.06	46.98	20.87	80.28	33.01	25.79	28.61	29.39	49.37	24.44	33.64

Table E.9: Godot

E.10 Elasticsearch

Release	FS	NB			LR			DT			RF		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
6.0.0 - 6.1.0	MU	19.33	41.67	26.06	11.56	70.97	19.83	16.73	14.86	15.81	21.5	13.61	12.73
	JM	19.91	30.56	23.64	11.14	68.75	19.13	13.75	13.75	11.84	17.5	1.11	8.76
	MO	18.65	46.25	26.38	12.21	69.72	20.75	20.07	21.67	21.4	15.33	3.47	1.43
	JM2	19.93	26.11	22.04	10.83	63.06	18.45	19.38	31.53	25.77	13.75	19.17	22.24
	ALL	22.36	41.81	28.88	11.76	68.61	20.88	13.15	14.86	13.93	29.17	2.22	3.43
6.1.0 - 6.2.0	MU	18.81	15.38	16.66	10.73	69.43	18.65	23.83	50.42	32.39	23.79	41.92	29.35
	JM	16.66	21.26	18.45	10.58	59.05	17.92	16.11	48.14	23.84	15.21	36.05	19.99
	MO	7.64	64.96	13.5	10.53	59.51	17.88	15.9	46.42	23.5	13.73	33.81	18.59
	JM2	14.86	10.87	12.38	6.54	43.81	11.36	13.71	46.4	20.95	13.17	36.52	20.09
	ALL	7.69	67.59	13.75	11.01	71.26	18.54	23.6	49.96	31.51	24.31	38.32	28.01
6.2.0 - 6.3.0	MU	23.95	23.8	23.73	16.19	62.6	25.74	25.1	32.97	28.27	27.97	18.8	22.8
	JM	27.15	24.29	25.51	15.49	52.24	23.78	20.2	29.84	23.52	29.37	14.95	21.9
	MO	22.02	24.3	22.95	15.66	51.53	23.95	22.33	30.35	25.29	27.67	15.88	18.86
	JM2	25.88	22.36	23.83	14.97	53.68	23.38	16.93	35.14	22.94	17.77	22.38	19.95
	ALL	23.34	24.06	23.51	15.97	61.16	25.33	22.31	31.77	26.24	26.94	13.98	20.65
6.3.0 - 6.4.0	MU	10.58	15.22	12.39	6.43	56.1	11.12	14.85	17.25	15.62	29.33	10.11	11.42
	JM	9.88	14.51	11.66	6.9	56.87	12.15	12.26	12.2	13.01	25.55	7.86	9.07
	MO	8.46	13.74	10.43	6.62	54.67	11.78	12.33	12.97	11.97	23.86	4.29	6.28
	JM2	9.1	10.82	9.74	6.72	56.87	12.02	2.31	6.54	3.46	2.51	3.57	4.48
	ALL	7.74	13.02	9.66	6.54	56.04	11.37	13.66	15.05	17.11	18.69	6.48	10.62

Table E.10: Elasticsearch

Glossary

accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$.
class label	A discrete attribute assigned to a record in a data set. The value of the class label is what is predicted.
code coverage	A measure of how much of the source code is covered by test cases. 100 % code coverage tells, that all functions, statements, and branches in the source code will be executed when running the test suite.
confusion matrix	A matrix of actual binary value as well as the predicted binary values of a classification model. The matrix is, thus, comprised of TP, TN, FP, and FN.
CSV	comma-separated values.
data set	A number of records containing features.
decision tree	A simple classification algorithm.
ER	entity-relationship.
F1	$2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$.
FN	false negative.
FP	false positive.

issue	An entity in GitHub used to contain information about bugs, requests, new features etc. in a repository.
JSON	JavaScript Object Notation.
LOC	lines of code.
model	A mathematical model that has been built by used of a classification algorithm which has been trained with historical data.
MSR	Mining Software Repositories.
OSS	open-source software.
precision	$\frac{TP}{TP+FP}$.
probabilistic classifier	A classification algorithm that is based on probability theory.
pull request	A specialized issue in GitHub, used to announce the presence of a new change to other users of a repository. Can be used to discuss and review the change..
recall	$\frac{TP}{TP+FN}$.
record	A specific instance in a data set. A record is comprised of one to many features.
test data	The subset of an entire data set that is used to evaluate the model. The test data is unknown to the model.
TN	true negative.
TP	true positive.
training data	The subset of an entire data set that is used to train the model.
VCS	version control system.

Bibliography

[1] Charu C. Aggarwal. *Data Mining: The Textbook*. Springer Publishing Company, Incorporated, 2015.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, October 1996.

[3] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10):1462–1477, October 1988.

[4] Max Bramer. *Principles of Data Mining*. Springer Publishing Company, Incorporated, 2nd edition, 2013.

[5] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. *2013 Ieee Sixth International Conference on Software Testing, Verification and Validation (icst 2013)*, pages 252–261, 2013.

[6] M.Y. Chu. *Blissful Data: Wisdom and Strategies for Providing Meaningful, Useful, and Accessible Data for All Employees*. Amacom, 2004.

[7] Dai Clegg and Richard Barker. *Case Method Fast-Track: A Rad Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[8] Valerio Cosentino, Javier Luis Canovas Izquierdo, and Jordi Cabot. Findings from github: methods, datasets and limitations. *2016 Ieee/acm 13th Working Conference on Mining Software Repositories (msr). Proceedings*, pages 137–41, 137–141, 2016.

[9] Dario Di Nucci, Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. Dynamic selection of classifiers in bug prediction: An adaptive method. *Ieee*

Transactions on Emerging Topics in Computational Intelligence, 1(3):202–12, 202–212, 2017.

[10] Michael Ernst. Version control concepts and best practices. `https://homes.cs.washington.edu/~mernst/advice/version-control.html`, 2019. Accessed 03 Jan 2019.

[11] International Organization for Standardization. ISO 8402:1986 - Quality – Vocabulary. Standard, International Organization for Standardization, Geneva, CH, June 1986.

[12] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. Method-level bug prediction. *International Symposium on Empirical Software Engineering and Measurement*, pages 171–180, 2012.

[13] GitHub. Thank you for 10 years. `https://github.com/ten`, 2018. Accessed 02 Jan 2019.

[14] GitHub. About stars. `https://help.github.com/articles/about-stars/`, 2019. Accessed 14 Jan 2019.

[15] GitHub. Issues. `https://developer.github.com/v3/issues/`, 2019. Accessed 02 Jan 2019.

[16] GitHub. Libraries, github developer guide. `https://developer.github.com/v3/libraries/`, 2019. Accessed 04 Jan 2019.

[17] GitHub. Understanding github flow. `https://guides.github.com/introduction/flow/`, 2019. Accessed 21 Jan 2019.

[18] TL Graves, AF Karr, JS Marron, and H Siy. Predicting fault incidence using software change history. *Ieee Transactions on Software Engineering*, 26(7):653–661, 2000.

[19] Dharmendra Lal Gupta and Kavita Saxena. Software bug prediction using object-oriented metrics. *Sadhana-academy Proceedings in Engineering Sciences*, 42(5):655–669, 2017.

[20] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *Ieee Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[21] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[22] J.A. Hartigan. *Bayes theory*. Springer,, 1983.

[23] Ahmed E. Hassan. Predicting faults using the complexity of code changes. *Proceedings - International Conference on Software Engineering*, pages 5070510, 78–88, 2009.

[24] James Joyce. Bayes' theorem. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition, 2016.

[25] Woosung Jung, Eunjoo Lee, and Chisu Wu. A survey on mining software repositories. *Ieice Transactions on Information and Systems*, E95-D(5):1384–1406, 2012.

[26] Eirini Kalliamvakou, Leif Singer, Georgios Gousios, Daniel M. German, Kelly Blincoe, and Daniela Damian. The promises and perils of mining github. *11th Working Conference on Mining Software Repositories, Msr 2014 - Proceedings*, pages 92–101, 2014.

[27] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. *Proceedings - Conference on Software Maintenance*, 2010.

[28] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66(4):1213–1228, Dec 2017.

[29] Will Koehrsen. Beyond accuracy: Precision and recall. `https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c`, 2018. Accessed 15 Jan 2019.

[30] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017.

[31] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *Ieee Transactions on Software Engineering*, 33(1):2–13, 2007.

[32] MongoDB. Data model design. `https://docs.mongodb.com/manual/core/data-model-design/index.html`, 2019. Accessed 14 Jan 2019.

[33] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *IEEE International Conference on Software Engineering*, pages 181–190, 2008.

[34] K. Muthukumaran, Abhinav Choudhary, and N. L. Bhanu Murthy. Mining github for novel change metrics to predict buggy files in software systems. *2015 International Conference on Computational Intelligence and Networks (cine)*, pages 15–20, 2015.

[35] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. *Proceedings - International Conference on Software Engineering*, 2006:452–461, 2006.

[36] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, December 1996.

[37] International Conference on Mining Software Repositories. Mining software repositories. `http://www.msrconf.org/`, 2018. Accessed 16 Aug 2018.

[38] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. *Proceedings - Conference on Software Maintenance*, pages 245–256, 2016.

[39] Tom Preston-Werner. Semantic versioning 2.0.0. `https://semver.org/`, 2019. Accessed 02 Jan 2019.

[40] Jacek Ratzinger, Martin Pinzger, and Harald Gall. Eq-mine: Predicting short-term defects for software evolution. 2008.

[41] Claude Sammut and Geoffrey I. Webb, editors. *Accuracy*, pages 9–10. Springer US, Boston, MA, 2010.

[42] Claude Sammut and Geoffrey I. Webb, editors. *Holdout Evaluation*, pages 506–507. Springer US, Boston, MA, 2010.

[43] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001.

[44] R Subramanyam and MS Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Ieee Transactions on Software Engineering*, 29(4):297–310, 2003.

[45] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. Predicting bugs using antipattern. *Proceedings - Conference on Software Maintenance*, pages 270–279, 2013.

[46] J Voas. Software's secret sauce: The "-ilities". *IEEE Software*, 21(6):14–15+18, 14–18, 2004.

[47] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. Softw. Eng.*, 14(9):1261–1270, September 1988.

[48] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[49] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. *Proceedings - Icse 2007 Workshops: Third International Workshop on Predictor Models in Software Engineering, Promise'07*, page 4273265, 2007.