MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Software Metrics for Software Defects Prediction

Master's Thesis

BC. DOMINIK ARNE REBRO

Brno, Spring 2022

MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Software Metrics for Software Defects Prediction

Master's Thesis

BC. DOMINIK ARNE REBRO

Advisor: Bruno Rossi, PhD

Brno, Spring 2022

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Dominik Arne Rebro

Advisor: Bruno Rossi, PhD

Acknowledgements

I would like to express my deepest gratitude to my thesis advisor Bruno Rossi, PhD, for all of his valuable advice, support and positive attitude. Secondly, I would like to thank my girlfriend Diana, as well as my family and close friend for all the support throughout my studies.

Abstract

This diploma thesis deals with the usage of software metrics in software defect prediction models. The main goal of the thesis is to evaluate the overall usefulness of software metrics for defect prediction, as well as to analyse the impact of individual metrics on the prediction. Software product metrics can be computed relatively quickly during the entire software development process, therefore their usefulness in defect prediction can potentially lead to improvements in resource allocation, as well as reduced development or maintenance costs.

Firstly, a class-level dataset consisting of 275 release versions of 39 Java projects was created by mining GitHub repositories. The dataset consists of defect information and source code used to compute 12 software metrics. Secondly, the created dataset was used to train 3 defect classification models. Thirdly, the results of the prediction models were analysed based on scoring metrics such as F-measure and AUC-ROC. Lastly, the feature importance of each individual metric was analysed, providing a ranking of metrics to identify the best performers.

The results across all projects indicate that overall software metrics can be useful for defect prediction. The Decision Tree (DT) and Random Forest (RF) classifiers achieved results mostly between 50% and 75% F-measure and AUC-ROC. While the best performing individual metrics were NOC, NPA, DIT and LCOM5.

Keywords

defect prediction, software metrics, data mining, machine learning, software quality, data analysis

Contents

List of Tables

List of Figures

Abbreviations

ACAIC	Ancestor class-attribute import coupling	17
ACM	Antipattern complexity metric	17
ACMIC	Ancestor class-method import coupling	17
ACPD	Antipattern cumulative pairwise differences	17
AMC	Average method complexity	17
ANA	Average number of antipatterns	17
ARL	Antipattern recurrence length	17
ATFD	Access to foreign data	17
ATLD	Access to local data	17
AUC	Area Under the Curve	29, 32
BCC	Basic code change	17
Ca	Afferent coupling	17
CAM	Cohesion among methods of class	17
CBM	Coupling between methods	17
CBO	Coupling between objects	17, 26
CBOI	Coupling between objects inverse	25, 27
CC	McCabe's cyclomatic complexity	17, 26
CD	Comment density	25, 27
CDISP	Coupling dispersion	17
Ce	Efferent coupling	17
CINT	Access to local data	17
CK	Chidamber & Kemerer[1]	24
CM	Changing methods	17
CSV	Comma separated values	21
DAM	Data access metric	17
DCAEC	Descendant class-attribute export coupling	17
DCMEC	Descendant class-method export coupling	17
DIT	Depth of inheritance tree	17, 26
ECC	Extended code change	17
EDHCM	Exponentially decayed history complexity metric	17
FN	False Negative	31
FP	False Positive	31
HCM	History complexity metric	17
IC	Inheritance coupling	17
KLOC	Thousands lines of code	17, 22, 24

LCOM	Lack of cohesion in methods	17, 26
LDHCM	Linearly decayed history complexity metric	17
LGDHCM	Logarithmic-ally decayed history complexity metric	17
LOC	Lines of code	17, 25, 26
LOCNAMM	Lines of code without accessor or mutator methods	17
MaMCL	Maximum message chain length	17
MAXNESTING	Maximum nesting level	17
MeMCL	Mean message chain length	17
MFA	Measure of functional abstraction	17
MLOC	Executable lines of codes	17
MOA	Measure of aggregation	17
NBR	Number of bug reports	24
NC	Number of contributors	24
NF	Number of forks	24
NLE	Nesting level else-if	25, 27
NMCS	Number of message chain statements	17
NOA	Number of attributes	17
NOAI	Number of attributes inherited	17
NOAM	Number of accessor methods	17
NOC	Number of children	17, 26
NOF	Number of attributes	17
NOLV	Number of local variables	17
NOM	Number of methods	17
NOMI	Number of methods inherited	17
NOMNAMM	Number of not accessor or mutator methods	17
NOPRA	Number of private attributes	17
NOPRM	Number of private methods	17
NPA	Number of public attributes	17, 25, 26
NPM	Number of public methods	17, 25, 26
OCAEC	Other class-attribute export coupling	17
OCAIC	Other class-attribute import coupling	17
OCMEC	Other class-method export coupling	17
OCMIC	Other class-method import coupling	17
PAR	Number of parameters	17
PRE	Number of pre-released bugs	17
RFC	Response for a class	17, 26
ROC	Receiver Operating Characteristics curve	29, 32
TCC	Tight class cohesion	17

TN	True Negative	31
TNC	Total number of commits	24
TP	True Positive	31
VIF	Variation Inflation Factor	27
WMC	Weighted method per class	17, 26
WOC	Weight of class	17

1 Introduction

Detecting and correcting defects in software systems has always been a crucial part of software engineering. One of the ways to improve the defect fixing process is to utilize defect prediction, i.e. using past defect information to highlight source code elements/segments prone to defects [2].

1.1 Motivation

Defect prediction has been actively researched in the last decades. With time and manpower being limited resources, knowing which parts of the software are more prone to defects can be beneficial for efficient resource allocation.

Using statistical models for defect prediction requires some form of numerical or categorical input [3]. Since software metrics are used to assign mathematical values to various software system attributes, they are the perfect candidate to be used as input to defect prediction models.

Using software metrics as a representation of the system for defect prediction has been becoming an increasing researched topic. An array of bug prediction approaches has been proposed in research articles, varying in both the type of metrics used (e.g. source code metrics, process metrics or code smells) and the granularity (e.g. file-level, class-level or method-level) of the created dataset [4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

The tools used in these articles are mostly deprecated or publicly unavailable. Therefore the respective bug prediction databases created even if public, can not be recomputed or extended with new projects. Creating a publicly available bug prediction tool would allow for further research to be conducted by allowing new baseline bug prediction databases to be constructed.

1.2 Goals and Methodology

The main goal of the thesis is to evaluate the usefulness of software metrics as defect predictors. A secondary goal is to design and implement a software tool capable of creating a dataset consisting of metrics and bug information, later using it to train and evaluate bug prediction models. For each project, the prediction performance of different models will be scored using multiple metrics such as F-measure and AUC-ROC.

Another aim is to evaluate which out of the selected metrics is the best predictor. This can be achieved by calculating feature importance values for each metric and evaluating them.

In summary, the goals of this thesis are:

Thesis goals

- Creating a dataset consisting of source code metrics and defect information.
- Using the dataset to train and evaluate bug prediction algorithms.
- Analysing performance of used algorithms.
- Evaluating which metrics are the best predictors.

The thesis will follow the research question methodology, as is customary in related works.

Firstly, we want to look at the overall applicability of source code metrics in defect prediction and the performance of such models. Secondly, we want to measure how individual metrics impact the prediction and which metrics are the best predictors. Lastly, we want to compare the results across multiple projects to evaluate their consistency. Considering the goals of the thesis 3 research questions were selected:

Research questions

RQ1. What is the impact of code metrics on software defect prediction?

RQ2. Which metrics are the most suitable predictors?

RQ3. How stable are the results across projects?

1.3 Structure

The thesis is divided into the following chapters:

Chapter 2 introduces the topics of software quality and bug prediction.

Chapter 3 summarizes publications that use software metrics for bug prediction.

Chapter 4 presents the experimental design and implementation concerning the defect and metric data mining, dataset creation and bug prediction.

Chapter 5 presents and evaluates the experimental results. It also answers the three research questions.

Chapter 6 provides a conclusion, states threats to validity, and points at possible future improvements.

2 Background

This chapter will present topics that are crucial for understanding this thesis. First, it will outline the topic of software quality with a focus on software metrics, defects and code smells. Afterwards, it will introduce the topic of machine learning in the context of defect prediction (also called predictive modeling).

2.1 Software Quality

Software quality can be defined in multiple different ways. The two most common views of software quality are product-based and user-based [15, 16]. The product-based view defines software quality as the capability of a software product to meet to its specifications, while the user-based view focuses on meeting user needs and expectations.

Fundamentally, successful software development management relies on the ability to measure progress and quality of the software product. Measurement is the process of assigning numbers or symbols to attributes of real-world entities in a way where they can be described according to clearly defined rules [3]. In software development, a wide variety of standardized software measurements has been proposed: they are more commonly referred to as metrics.

2.1.1 Software metrics

Depending on what aspect of software development is measured, software metrics can be divided into three main categories [17, 18, 19]:

- *Product metrics* describe the attributes of the software product such as size, complexity, performance, reliability or level of quality. They can be computed at any stage of the software development process and can be a useful indication of whether user requirements are being met. They can focus on various levels of the system from source code (individual methods, classes or modules) to the software system as a whole.

- *Process metrics* focus on measuring the software development process in terms of maintenance, development costs, scheduling, etc. They are useful for assessing the efficiency and effectiveness of the software development process in order to provide a framework for the improvement of the process.

- *Project metrics* evaluate the characteristics of the software project as a whole, looking at progress, costs, staffing details or productivity. These metrics are mostly focused on project management and are therefore not relevant for this thesis.

Over the years, the field of software measurement has defined many metrics describing various attributes of software systems. We focus mostly on product metrics, as they seem the most related to the defect-proneness of software systems. Metrics computed from the source code mostly focus on either the size or the structure of the software. The most well known software metric describing size is the Lines of Code (LOC) metric. Structure can be described in terms of control-flow structure (e.g. using the cyclomatic complexity metric), information flow (e.g. using CK [1] metric) or data structure(e.g. looking at the number of variables and operands). With the popularity of object oriented design, the number of explored object oriented metric also increased. The most well-known and empirically validated [1, 14] set of object oriented metrics is the mentioned CK metrics suite, consisting of six metrics WMC, CBO, DIT, NOC, RFC, LCOM. The 6 CK metrics as well as 5 additional object oriented metric used in this thesis are closer described in **Table 4.2**.

Utilizing software metrics has many advantages [17]. They are useful for comparing software systems against specifications and requirements in a clearly defined way and they help with analysing the development process in order to increase understanding and guide process improvement. They provide feedback about progress and quality during various stages of the development process, while also being a useful early indicator of potential problems [15].

However, metrics are only useful if they are properly understood. Misinterpreted or misunderstood metrics can be misleading and bring about poor development or management decisions [15, 3]. Additionally, collecting metrics is associated with certain costs and should therefore be done under clearly defined goals and regularly assessed for effectiveness and efficiency.

2.1.2 Software defects and code smells

A software defect or bug can be defined as a fault or error in the software that causes it to compute incorrect or unexpected results, additionally, it can be defined as a deviation from specifications or expectations which can lead to failure in operation [20, 21]. Every software system is bound to have some defects occur during its life cycle, some are easy to find and others deeply concealed, some can be considered negligible and others might lead to significant costs.

Being related to defects, code smells are defined as quality issues in source code, that lead to difficulty with maintaining and evolving the software product [22]. Code smells might not necessarily indicate defect proneness, they rather hint that something might be wrong. It is possible to remove code smells by the use of refactoring [22]. Code smells are usually identified by exceeding certain software metric thresholds [23, 24, 25].

2.2 Defect prediction

It is not enough to wait for defects to appear and only then fix them. In order for a certain level of software quality to be assured, defects must be effectively detected and corrected at the earliest possible point in the software development process. This means it would be notably beneficial to be able to predict parts of the system that are defect-prone and use this information to improve resource allocation and software quality assurance.

Machine learning research focuses on identifying trends in large datasets and using the gathered information to train statistical models capable of predicting outcomes based on inputted data [18]. Software defect prediction model refers to statistical models that try to predict potential software defects, based on some test data provided [26]. Given the mathematical nature of metrics, they are suitable to be used as input for defect prediction models. Prediction models consist of independent variables (software metrics) and a dependent variable indicating defectiveness. There exist multiple machine learning techniques viable for defect prediction:

- *Regression* is a statistical process of evaluating the relationship between independent variables and a continuous dependent variable. The relationship is represented in the form of a function(e.g. linear) of the independent variables that can be used to predict the dependant variable [18, 6, 7, 10].

- *Classification* aims to predict a certain outcome based on a given input. It uses input data referred to as a training set, which consists of objects with known class labels. The goal of a classification algorithm is to analyse the training set in order to develop a model, which can be used to classify objects with unknown labels. In terms of defect prediction, the most commonly used is binary classification, where objects are labelled as defective or not defective [18, 2, 27, 8].

- *Clustering* is a typical unsupervised machine learning method, which tries to categorize a set of objects into clusters or groups with similar characteristics. It is based on grouping said objects in a way where objects share similarities only with other objects within the same cluster while being dissimilar to those in different clusters. In terms of defect prediction, the goal is to identify a cluster of defective objects [18, 28, 4, 20].

The most popular machine learning method for defect prediction is classification [29]. A variety of classification algorithms has been proposed in machine learning literature [30, 31]. The classification algorithms utilized by this thesis, as well as being three out of four most popular algorithms used for defect prediction according to [29], are the following:

- *Naive Bayes* (NB) is based on Bayes theorem and assumes independence between each pair of predictors. The NB classifier is relatively quick and requires less training data than more sophisticated approaches [30], however, the performance is affected by its strong assumption of independence.

- *Decision Tree* (DT) uses a tree structure to represent both classification and regression models. It refers to a hierarchical model of decisions and their consequences, where the leaf nodes represent classification [18].

- *Random Forest* (RF) can be viewed as an extension of the DT algorithm. It is based on running multiple decision trees in parallel where, in the end the results are averaged across all the trees to provide the final classification [30, 32]. It is suitable for both classification and regression techniques.

3 Related works

This chapter will summarize already published research regarding the usage of software metrics for the purpose of defect prediction. The literature is reviewed in terms of the rationale, used metrics and achieved results. At the end of the chapter is a summary of key findings.

3.1 Researched literature

Jureczko et al. [4] conducted a study regarding the clustering of software projects with regard to software defect prediction. In this study 19 class-level source code metrics were used to identify groups of software with similar characteristics making them viable to be analysed using the same bug prediction models. The metrics were obtained using the Ckjm[1] tool. Using 38 proprietary, open-source and academic projects, the study found 6 different clusters. However, only 2 clusters, proprietary and proprietary/open were statistically proven.

Palomba et al. [2] proposed using a new metric indicating the severity of detected code smells called code smell intensity. It was designed to improve traditional models that use source code metrics for software defect prediction by adding code smell related information in form of the intensity index, computed by JCodeOdor [33]. The results showed that the code smell intensity is a relevant predictor in models based on both process and product metrics. The achieved F-score was reported to be at least 13 % greater than state-of-the-art models.

Taba et al. [5] proposed the usage of metrics based on antipatterns for defect prediction. The rationale is that antipatterns indicate poor design decisions and therefore also defect prone software elements. They found that antipattern presence is related to a higher risk and density of defects. This study used DECOR [25] antipatterns, detected by the Ptidej [34] tool, measured over several releases of Eclipse and ArgoURL to calculate custom antipattern metrics. Using antipattern metrics to extend baseline source code metrics provided a roughly 12.5% increase in F-measure scores of bug prediction models.

1. `http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm`

Hassan et al. [6] proposed multiple metrics based on code change complexity, as a superior predictor to past faults or modifications. Code change complexity is computed from past modifications, with the exception of Fault Repairing and General Maintenance modifications, since they are not considered to be a part of the code change process. The rationale is that code elements undergoing complex changes are more prone to contain faults since such elements can be difficult for the entire development team to fully understand at all times. They used statistical paired tests to study the difference in prediction error between the use of change complexity metrics and prior faults. The results indicated a decrease in prediction error of 15-38% compared to previous faults. The study, however, lacks a general performance analysis of the defect prediction models built on change complexity metrics.

D'Ambros et al. [7] performed an extensive comparison of bug prediction approaches and created a dataset by combining the changelog and single-version approaches to construct bi-weekly snapshots of 5 open-source systems. This study considered classes at file-level, ignoring the fact that a Java file can contain multiple nested classes which was confirmed to be the case in many open-source projects [35]. The resulting dataset was created by first transforming source code into FAMIX [36] models using inFusion[2], calculating metrics from FAMIX models using Moose[3] and lastly linking metrics with bug tracking information using Churrasco[4]. In addition to source code metrics, also previous defects, change metrics, code churn and entropy of changes are available in the dataset making it suitable for benchmarking the performance of new prediction models. The reached conclusion was that a single technique performing well in all contexts does not exist.

Ferenc et al. [8] researched using deep learning in static, metric-based defect prediction. Machine learning using deep neural networks has been performing generally well [37], therefore it is reasonable to believe it can perform well also in the field of bug prediction. They discovered that using deep learning resulted in performance on par with or better than the best performing metric-based algorithm, Random

2. `http://www.intooitus.comJ`
3. `http://www.moosetechnology.org`
4. `http://churrasco.inf.usi.ch`

forest. Their model achieved F-measure of roughly 55% and AUC-ROC of almost 84%, noting that better performance is attainable by using a larger training dataset.

Emam et al. [9] performed a validation of object-oriented metrics regarding their use for fault prediction and software quality estimation. This study followed one of the hypotheses stated in [38], that structural class properties affect the cognitive complexity(i.e. the mental burden of individuals working with the component), which in turn impacts external attributes such as fault-proneness or maintainability. They considered 2 inheritance and 8 coupling metrics, however the combination of metrics DIT and OCMEC as predictors was studied, achieving AUC-ROC score of roughly 78%. The results indicate that inheritance (DIT) and export coupling (OCMEC) metrics are strongly associated with fault-proneness, making them suitable for bug prediction.

Bell et a. [10] validated several code change metrics proposed by [39], with the rationale being that frequently changed code elements are more likely to contain defects. They investigated to what degree faults can be predicted using code churn alone and also in combination with other characteristics, conducted on 18 versions of a provisioning system containing files from multiple programming languages including Java, C and C++. The conclusion was that the various tested metrics based on prior code changes performed equivalently well, demonstrating the usefulness of such metrics in bug prediction.

3.2 Summary

- Many different types of software metrics and prediction approaches have been already proposed, with most of them achieving solid prediction performance. For example, the smell intensity metric model [2] achieved around 90% f-measure and 80% AUC-ROC.

- In the researched studies, there is a variety of prediction results scoring metrics used. For classification models, the most common being precision, recall, F-measure and AUC-ROC, while for regression models metrics such as mean squared error, R^2

and Spearman's correlation coefficient are used. Some papers focused on the comparison between metrics, while others on pure prediction model performance. This inconsistency makes the comparison of results across all related works relatively problematic.

- There is some overlap in terms of software metrics used to build bug prediction models, these include variants of the LOC metric, CK metric suite [1]: WMC, DIT, NOC, RFC, LCOM, CBO. These metrics seem to be suitable for defining a baseline, against which novel metrics can be compared and have been frequently used for this reason in researched literature.

- Some studies make the created datasets publicly available, however, most reviewed studies only provide experiment results. The entire experimental package is publicly available only for Palomba et al. [2]. Thus most of the studies could be difficult to reproduce, even if the analysis was done on open-source projects.

- The granularity of the analysis is most commonly at file-level, class-level or method-level, since higher level, e.g. package-level, are less useful because of the need to pinpoint more precisely the location of defects for defect fixing efforts.

- The results indicate that models using the Random Forest algorithm tend to achieve the best classification performance as observed in [8, 27, 40], with the Decision Tree and Simple Logistic algorithms lagging slightly behind.

The used metric types and bug prediction models are briefly summarized in **Table 3.1**, while **Table 3.2** lists software metrics used in related works.

Table 3.1: Used metrics types and prediction models in related works

Src.	Metric type	Prediction model
[4]	Product metrics	Kohonen's neural network
[2]	Code smell intensity	Simple Logistic, Decision Tree Majority, Naive Bayes, Logistic Regression, etc.
[5]	Product, antipattern metrics	Various intra-system and cross-system
[6]	Change complexity metrics	Statistical linear regression
[7]	Product, process, other metrics	Generalized linear regression
[8]	Product metrics	Deep neural networks
[9]	Inheritance, coupling metrics	Calibrated logistic regression
[10]	Code change metrics	Binomial regression

Table 3.2: List of used metrics in related works

Src.	Used metrics
[4]	• Source code metrics: WMC, DIT, NOC, RFC, LCOM, CBO, LCOM3, NPM, DAM, MOA, MFA, CAM, IC, CBM, AMC, Ca, Ce, CC, LOC
[2]	• Source code metrics used for smell detection: ATFD, ATLD, CC, CDISP, CINT, CM, FanOut, LOC, LOCNAMM, MaMCL, MAXNESTING, MeMCL, NMCS, NOAM, NOLV, NOMNAMM, NPA, TCC, WOC • Code smell metric: Code smell intensity
[5]	• Source code metrics: LOC, MLOC, PAR, NOF, NOM, NOC, CC, DIT, LCOM, NOT, WMC • Process metrics: PRE, Churn • Antipattern metrics: ANA, ACM, ARL, and ACPD
[7]	• Source code metrics: WMC, DIT, NOC, RFC, LCOM, CBO,LOC, FanIn, FanOut, NOA, NPA, NOPRA, NOAI, NOM, NPM, NOPRM, NOMI • Change metrics: EDHCM, LDHCM, LGDHCM
[6]	• Change complexity metrics: BCC, ECC, HCM
[8]	• Source code metrics: 60 class-level complexity, size and object-oriented metrics
[9]	• Inheritance metrics: DIT, NOC • Coupling metrics: ACAIC, ACMIC, DCAEC, DCMEC, OCAIC, OCAEC, OCMIC, OCMEC
[10]	• Explored metrics: log(KLOC), Prior faults, Prior changes, Prior changed, Prior developers, Prior lines added/deleted/modified

4 Experimental design and implementation

As the aim of this thesis is to evaluate the impact of software metrics on defect prediction, this chapter will focus on the design and implementation of the data analysis process performed during the evaluation, explaining in detail how our defect dataset was defined and used by machine learning models in the defect prediction process. The experimental design is tailored to the needs of the three selected research questions:

Research questions

RQ1. What is the impact of code metrics on software defect prediction?

RQ2. Which metrics are the most suitable predictors?

RQ3. How stable are the results across projects?

4.1 Dataset creation

The first option considered was to use an existing database suitable for bug prediction, i.e. containing metrics and bug information. However, databases researched such as PROMISE [41], Bug prediction dataset [7] or Eclipse dataset [42] were consisting of a small number of projects or were created by analyzing proprietary software. Using open-source projects is a better option for defect prediction databases, seeing that the source code is publicly available, different types of metrics or prediction models can be tested on the same source code, allowing for accurate results comparison.

In the end, we decided to create a dataset from scratch by mining defect information, computing code metrics and linking them together. Apart from the reasons mentioned above, this allows for a freedom to choose projects and group them based on their domain for further analysis of the prediction results.

4.1.1 Data acquisition

The needed bug information can be obtained from external bug tracking systems such as Bugzilla[1] or Backlog[2]. However, some source code hosting services such as GitHub[3] include an internal issue tracking system, allowing for a more consistent and straightforward acquisition of source code and bug information by cloning project repositories and using the GitHub restful API[4]. For this reason, we decided to use GitHub as the sole source of data for our dataset. The GitHub API provides only 60 requests per hour for unauthenticated users. This number can be increased to 5000 requests per hour for authenticated users (e.g. using an OAuth token), which is enough for the need of our implementation.

Python was our first choice of programming language because it provides all the necessary libraries needed to implement our solution. The GitHub API access and local repository mining was done using PyGithub[5] and PyDriller[6], while the bug prediction was performed by utilizing the popular machine learning library scikit-learn[7]. An overview of the data acquisition process is presented in **Figure 4.1**.

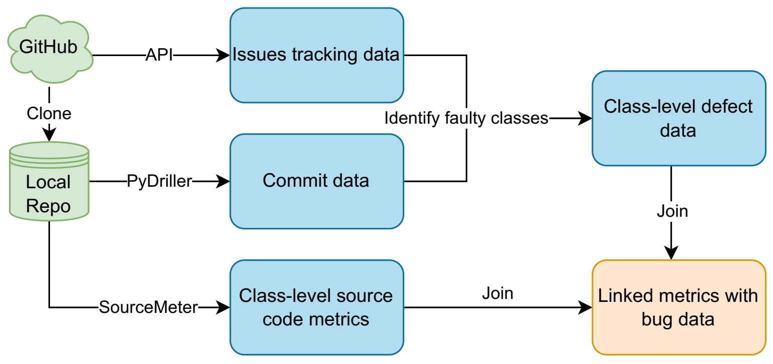

Figure 4.1: Data acquisition process

1. `https://www.bugzilla.org/`
2. `https://backlog.com/bug-tracking-software/`
3. `https://github.com/`
4. `https://docs.github.com/en/rest`
5. `https://github.com/PyGithub/PyGithub`
6. `https://github.com/ishepard/pydriller`
7. `https://scikit-learn.org/stable/`

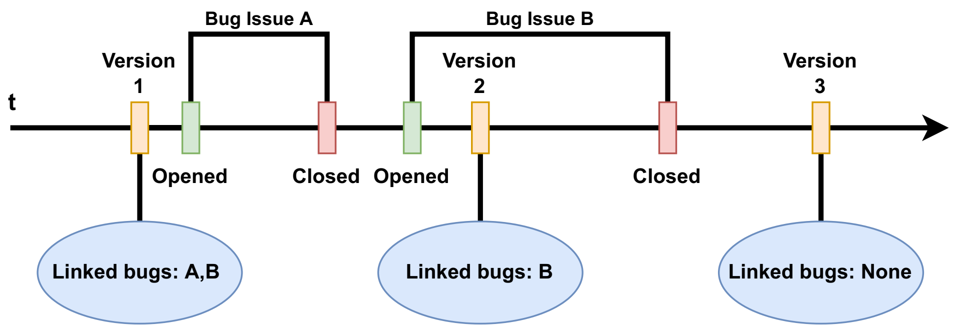

Figure 4.2: Link between bugs and versions

The traditional approach of linking fault information to system versions, which we employed, is based on selecting multiple system versions spread at roughly the same interval, 6 months in our case. This approach allows for the accumulation of multiple bugs into a single version, the source code of which can be analyzed to calculate software metrics. With the intention of including as many bugs as possible and to some extent creating a more balanced dataset, issues created, fixed and closed between two selected versions are also associated with the earlier version, similarly to [27, 40, 43]. The relationship between bugs and versions is illustrated in **Figure 4.2**.

Linking issues with the associated bug fixing commits has been traditionally done by parsing commit messages to find references to issues. This is done internally by GitHub and the information is available as a list of events, which can be obtained with ease using the GitHub restful API. The *issue_stats_miner.py* module is responsible for obtaining project statistics along with a list of all closed bug issues containing for each issue their id, opening and closing timestamps, as well as all bug fixing commits associated with them.

The next step is to obtain bug information for each of the bug fixing commits. Linking defect information at file-level is relatively straightforward, because of the fact that commits include a list of all modified files. This brings us to the issue of nested classes, where a single file can contain multiple classes. Due to the object-oriented nature of the projects considered, as well as class-level metrics being found more effective than file-level metrics, the dataset will be created considering only class-level metrics. It is, therefore, necessary to first identify the source code location of classes before it is possible to link them with defect information. First, we experimented with using regular expressions to look for class declarations, however, this is problematic because of the need to match opening and closing brackets to correctly identify the span of each class. A much more consistent method of identifying the classes is to use static source code analysis. The downside being a significantly higher computational complexity compared to regular expressions, because of the need to run static analysis on every modified file in every commit linked to a bug issue.

The source code positions of individual classes, as well as over 60 metrics, were computed using the free static analysis tool SourceMeter[8]. It was selected because it supports most of the popular programming languages (Java, C/C++, C#, Python, etc.) and is usable from the command line, making it possible to be called from inside a python script for automated analysis. The results are stored in CSV format, where each row represents a single class, making data readable and convenient to link with the mined defect information.

By parsing diff files available for all files modified in a commit, we are able to learn which specific lines of a file were modified. Now that we have already discovered the source code positions of all classes we can match the modified lines to specific classes and mark them as defective. Linking this defect information with the results of static analysis marks the final step in the dataset creation process.

4.1.2 Selected Projects

For the reason mentioned at the beginning of **Section 4.1**, we consider only projects on GitHub, that also use the GitHub issue tracker so that all the information necessary to build our defect prediction dataset can be accessed through the GitHub API and by processing Git commits.

Another requirement is for the considered projects to have a good amount of closed bug reports relative to the total time used for version selection. The minimal time chosen was 2 years i.e. 4 versions spread 6 months apart, however, most projects were analysed in a 3-4 year time window.

8. `https://sourcemeter.com/`

As for the size of the project, the goal was to focus mostly on large projects. We define a large project as having around 30+ KLOC and consisting of commits in the order of thousands. In the end, we ended up analysing some smaller projects as well, for the sake of creating a more extensive defect prediction dataset.

Part of the initial plan was to mine repositories with an even split between Java and C++ projects. However, only a couple of suitable C++ projects were found, therefore, in the end, only Java repositories were considered.

14 projects were selected because of being included in a related study [27], as well as passing all the requirements. In the end, 39 suitable Java projects, consisting together of 275 versions, were selected for our dataset. They are summarized in **Table 4.1**, including statistics regarding the repository size, development team size and bug tracking information.

Table 4.1: Project statistics

Name	Size (MB)	KLOC	NC	NF	TNC	NBR
junit4	24.3	26	144	3169	2486	90
junit5	621.2	79	169	1155	7122	273
powermock	5.9	37	46	556	1626	282
netty	79.3	71	363	14282	10774	2640
Broadleaf	282.6	142	76	1168	17594	1195
jetty.project	201.2	339	164	1780	24058	830
spring-boot	148.4	192	368	35995	37114	3068
framework	489.0	347	195	729	19155	5282
elasticsearch	970.8	508	349	21461	63555	11027
opengrok	292.0	70	97	669	6830	773
titan	91.2	142	32	1036	4434	127
orientdb	248.5	371	131	849	20924	1863
mapdb	10.2	46	38	861	2184	235
neo4j	543.9	680	218	2128	73042	1821
presto	168.0	589	347	4541	19721	562

Name	Size (MB)	KLOC	NC	NF	TNC	NBR
hugegraph	9.9	102	33	366	1373	98
antlr4	65.8	50	263	2626	8199	188
RxJava	133.6	270	277	7590	5969	732
guava	372.1	395	271	9831	5788	393
Discord4J	23.1	27	68	251	3705	193
A-U-I-L	27.1	13	36	6244	1038	75
find-sec-bugs	7.6	36	56	412	1302	92
checkstyle	105.0	209	305	8221	11270	560
pmd	386.6	144	243	1274	18836	559
jsoup	5.0	15	89	1999	1648	115
fastjson	15.3	172	170	6400	3946	228
jmonkeyengine	976.1	607	131	1052	7615	107
mcMMO	28.6	29	152	722	6528	727
Terasology	299.8	175	262	1307	11750	779
Mindustry	1764.6	87	399	1974	12941	1908
ExoPlayer	164.0	161	213	5607	12940	1039
kura	209.1	196	52	286	5192	660
dubbo	39.3	185	386	24819	5412	352
paho.mqtt.java	6.1	35	47	748	846	133
webcam-capture	23.9	17	29	1093	688	76
oryx	7.5	95	15	412	1113	68
ceylon [9]	86.3	143	3	31	10	819
hazelcast	355.0	771	272	1601	35981	6558
shardingsphere	43.5	26	85	3146	2204	149

9. Ceylon statistics are incomplete, because the project has been moved to a new repository

The statistics used and their rationale can be found in the list below:

- Size in megabytes - Size(MB), representing the size of the entire repository considering all file types;
- Number of forks - NF, indicating development activity;
- Total number of commits - TNC, showing repository size and development activity;
- Lines of code in thousands - KLOC, revealing source code size;
- Number of contributors - NC, indicating development team size;
- Number of bug reports - NBR, showing issue tracking activity.

For each project repository, the necessary information about the owner, name, domain, bug label and versions analysed was obtained mostly manually and stored inside a JSON configuration file constructed as a list of such project objects. To analyze new projects using this tool, the mentioned project information must be provided in the same format and added as an entry into the list of projects in the JSON configuration file.

4.1.3 Selected Metrics

As mentioned in **Section 3.2**, many related works employ the CK metric suite, however not enough analysis of which metrics from the suite are the best predictors has been conducted. The 6 CK metrics identify the size, complexity, cohesion and coupling of source code elements and were found to correlate with defects [1]. To have a meaningful analysis of individual metrics as predictors and have a chance to compare different metric suites we selected a metrics suite OTHER(consisting of 5 object-oriented metrics), intended mostly to extend the CK metric suite with new uncorrelated metrics. The simple LOC metric suggested by [43] to be a suitable defect predictor will be used as a baseline for our analysis. Using only source code metrics provides the advantage of allowing defects to be spotted very early in the development process, by relying only on prior and current source code and thus resulting in potentially lower correction costs.

The amount of different metrics necessary to achieve strong prediction results is quite low, in fact using only 3 metrics can be enough according to [44]. We want to compare the prediction impact of multiple metrics, however including too many metrics in the analysis would only aggravate the issue of multicollinearity, thus hindering our results by wrongly scoring in favour of uncorrelated metrics, as explained in **Section 4.2** in more detail.

From the over 60 source code metrics computed, the following 12 metrics (also summarized in **Table 4.2**) were selected:

- Baseline metric LOC indicates the size of source code for each class and therefore also the probability of defect occurrence, since larger classes are more likely to contain faults than smaller ones. Additionally, it has been suggested by [43] to be a suitable predictor of defects, but because of high correlation with many other product metrics, it has been used as a stand-alone baseline metric in many other related works[7].

- The CK [1] metric suite, consisting of 6 object-oriented metrics WMC, DIT, NOC, RFC, LCOM5 (a modified version of the original LCOM metric), CBO. These metrics were validated by [14] on several C++ projects, showing their usefulness as a quality indicator. The CK suite has been extensively used for the purposes of defect prediction, however, the comparison of the impact of its individual metrics on defect prediction has been mostly disregarded in larger dataset studies, making it suitable for our **RQ2** feature importance analysis.

- The OTHER metric suite, consisting of 5 additional object-oriented metrics NPA, NPM, NLE, CBOI, CD. These metrics were selected, because of their potential to include new information into the prediction model, due to some conceptual differences with the CK metrics. This also means that the correlation between the OTHER and CK metrics is low for the majority of projects.

Table 4.2: Metrics selected for defect prediction

Acronym	Full name	Description
LOC	Lines of code	Size metric, defined as the number of all lines of code, excluding nested, anonymous and local classes.
WMC	Weighted method per class	Complexity metric, defined as sum of McCabe's cyclomatic complexity of the classes local methods and init blocks.
DIT	Depth of inheritance tree	Inheritance metric, defined as the distance from the farthest ancestor in the inheritance tree.
NOC	Number of children	Inheritance metric, defined as the number of all directly derived classes, interfaces, enums and annotations.
CBO	Coupling between objects	Coupling metric, computed as the number of directly used classes, i.e. through inheritance, reference, method calls, etc.
RFC	Response for a class	Coupling metric, defined as the number of local methods and other methods directly invoked within local methods and attribute initialization.
LCOM5	Lack of cohesion in methods 5	Cohesion metric, defined as the number of functionalities of the class, with relation to the encapsulation, or single-responsibility principles.
NPA	Number of public attributes	Size metric, computed as the number of all public attributes, excluding attributes of nested, anonymous and local classes.
NPM	Number of public methods	Size metric, computed as the number of all public methods, excluding methods of nested, anonymous and local classes.

Acronym	Full name	Description
NLE	Nesting level else-if	Complexity metric, computed as the maximum depth of conditional, iteration and exception handling embeddedness, considering only the fist if instruction in the if-else-if construct.
CBOI	Coupling between objects inverse	Coupling metric, computed as the inverse of CBO, i.e. the number of other classes directly using the class.
CD	Comment density	Documentation metric, computed as the ration of comment lines to total lines(i.e. comment and logical)

4.2 Data preprocessing

As suggested earlier defect prediction models can suffer from multicollinearity, where two or more metrics of the model are highly correlated. Multicollinearity does not affect the predictive power of the model, however, it does distort feature importance analysis related to RQ2. In our case, feature importance values for each metric were computed using a 10-fold cross-validated permutation feature importance[45] model, defined as the decrease in the model score when a single feature value is randomly shuffled [45]. Multicollinearity can distort the results since when two features are correlated and one of them is permuted, the model still has access to the permuted feature through the correlated feature, which leads to low importance for metrics that might in reality be of high importance. Therefore for the feature importance analysis to show us an accurate picture of reality, the data must first be preprocessed to mitigate the issue of multicollinearity between metrics [46, 5].

To address this issue the Variation Inflation Factor is usually used [5, 2], it is a measure of multicollinearity defined as the ratio of overall model variance to the variance of a model consisting of a single independent variable(metric). By calculating the VIF value for each metric, it is possible to detect problematic suffering from multicollinearity. As suggested in [5] the threshold of VIF was set to 2.5 for further investigation, while VIF values above 10 would indicate serious multicollinearity.

While [14] found correlation between RFC and CBO, in [2] the pair of metrics RFC and WMC was found to be highly correlated, which lead to the removal of RFC from further analysis. Even after disregarding RFC, the WMC metric still consistently achieved VIF values above 5. Therefore we decided to remove both metrics RFC and WMC from the feature importance analysis relating **RQ2**.

For each project the VIF values of all metrics were computed, in the majority of projects, all metrics were able to fall within or close to the VIF $<$ 5 threshold for all metrics. In a couple of outlying cases some of the metrics achieved VIF close to 10, which indicates significant multicollinearity. We concluded that there will probably always be an outlier project for any metric combination, that suffers from high multicollinearity. Since in a vast majority of projects the metrics selected achieved reasonable VIF values, therefore instead of removing further metrics we decided to simply disregard these outlier projects from **RQ2** feature importance analysis.

Usually, defective parts of the system represent a small minority, meaning our dataset is skewed toward classes without bugs. This means that it might prove beneficial to first use under-sampling to get rid of the class imbalance as this can improve the accuracy of prediction models. However, too much under-sampling can lead to loss of information and thus the underfitting of the model, which can cause poor bug prediction results. For this reason, class imbalance was only addressed to the extent of deleting duplicate rows, which were almost exclusively of the majority$(not\ defective)$ class.

4.3 Defect prediction

Our dataset contains information about the presence or absence of defects at class-level for each specific version, making us treat the prediction problem as a binary classification problem, assigning each class into the $defective$ or $not\ defective$ category. The selection of classification algorithms was mostly guided by related works. As explained in Section 3.2, the best performing algorithms were the Random Forest and Decision Tree algorithms. In addition to the top performers, we also selected the Naive Bayes algorithm because of its frequent use as a naive baseline to help in comparing performance results.

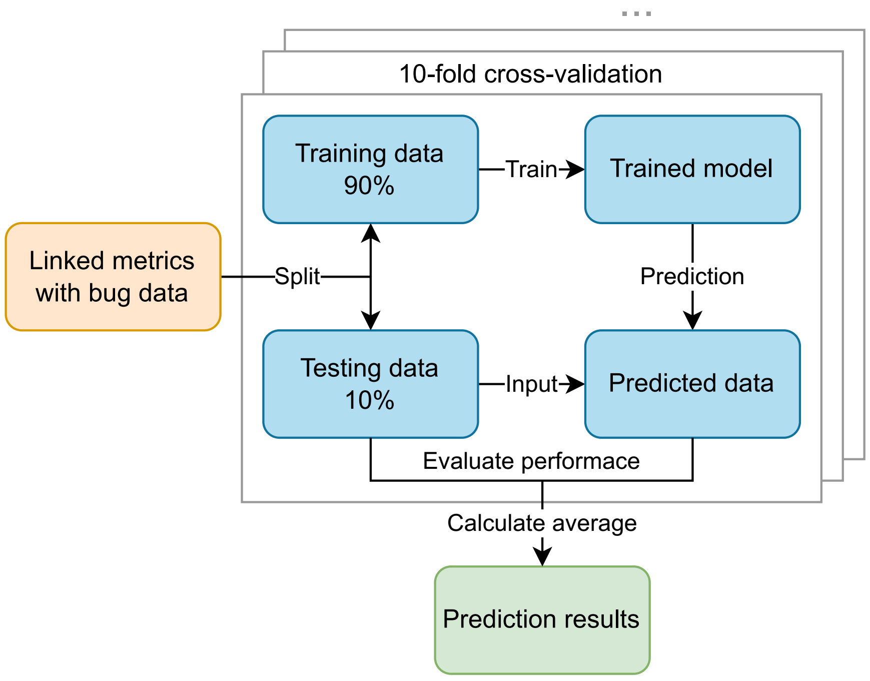

Figure 4.3: Defect prediction process

The preprocessed dataset is divided into a training and testing subset. The training set is used to fit(train) the prediction model, meaning the algorithm is learning how to predict the dependent variable. The testing set is used to predict dependent variable values and validate them, based on scoring metrics such as precision, recall, F-measure or AUC-ROC.

For a better estimate of our models' performance, 10-fold cross-validation was employed by randomly splitting the data into 10 stratified[10] folds, with 90% of the data used for training and 10% for testing. The folds are constructed in a way where over all folds, 100% of the

10. Stratified - Ensuring in all training and testing subsets the same proportion of *defective* to *not defective* observations as in the original dataset

data entries are used to validate the prediction. In the end, the mean of all individual folds' scores is computed. The prediction process implemented within each fold is illustrated in **Figure 4.3**.

Within the same folds, all metrics expect LOC, WMC and RFC are ranked according to their permutation importance values. All the rank data can be plotted using boxplots, providing more information than a computing simple average rank. Details on how the prediction results and importance ranks were evaluated, along with answers to our research questions are provided in **Chapter 5**.

5 Experimental results evaluation

The aim of this chapter is to analyse and evaluate the experimental results, with the main focus being on answering our three research questions:

Research questions

RQ1. What is the impact of code metrics on software defect prediction?

RQ2. Which metrics are the most suitable predictors?

RQ3. How stable are the results across projects?

5.1 Scoring metrics

The results of defect prediction models can be evaluated using various scoring metrics. For evaluation, we will use the standard measurement metrics for binary classifier scoring, i.e. precision, recall, F-measure and AUC-ROC, utilized in related works [2, 8, 9]. These metrics are computed with the help of the confusion matrix illustrated in **Table 5.1**.

Table 5.1: The Confusion Matrix

	Actual value	
Predicted value	Defective	Not Defective
Defective	True Positive (TP)	True Negative (TN)
Not Defective	False Positive (FP)	False Negative (FN)

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

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

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Figure 5.1: Scoring metrics formulae

Precision, also called positive predictive value, indicates the ratio of TP to all positive results, which shows us out of all the classes classified as defective, what percentage was correctly classified. Recall, also called the true positive rate, indicates the ratio of TP to all actual positives, which shows us out of all the actual defects, what percentage was correctly classified as defective. In this sense, we can think of recall as the bug coverage of the prediction model. F-measure is defined as the harmonic mean between precision and recall [47]. It combines both metrics into a single metric, making it suitable for a more compact and clear result comparison. The formulae defining the metrics are summarized in **Figure 5.1**

The Receiver Operating Characteristics curve (ROC) curve [48] is a graphical interpretation of classifier results, it plots false positive rate($\frac{FP}{FP+TN}$) against recall, at all possible classification thresholds. The Area Under the Curve (AUC) of the ROC curve reduces the dimensions to a single value between 0 and 1, essentially providing a summary of the ROC curve. This makes AUC-ROC suited for result comparison and has been widely used as a classifier scoring method. AUC-ROC can be viewed as the probability that a classifier ranks a random positive case higher than a random negative case.

5.2 Results analysis

In order to make the results easier to comprehend they will first be showcased on a single project. Since **RQ3** does not make sense in terms of a single project, only **RQ1** and **RQ2** will be considered in this example. After that, the overall analysis of results across all projects will be discussed in terms of all three research questions.

5.2.1 Single project example

The project chosen for this example is opengrok, because of being average in the project statistics mentioned in **Table 4.1**, while also containing a good ratio of defects and thus achieving favourable prediction results.

Table 5.2: Results of prediction for project opengrok split by metrics and classifiers

	Naive Bayes				Decision Tree				Random Forest			
	Prec.	Rec.	F-m.	A-R	Prec.	Rec.	F-m.	A-R	Prec.	Rec.	F-m.	A-R
LOC	0.438	0.5	0.467	0.5	0.684	0.602	0.64	0.602	0.694	0.623	0.656	0.623
CK	0.537	0.578	0.557	0.578	0.832	0.775	0.802	0.775	0.844	0.784	0.812	0.784
OTHER	0.474	0.441	0.456	0.441	0.849	0.791	0.819	0.791	**0.866**	0.788	0.825	0.788
CK + OTHER	0.566	0.627	0.595	0.627	0.844	0.795	0.819	0.795	0.861	**0.8**	**0.829**	**0.8**

RQ1 Looking at the models' scores in **Table 5.2**, right away we can see that overall the scores indicate meaningful defect prediction performance. The Random Forest model seems to be the best performer, with the Decision Tree model achieving only slightly worse scores, both attaining F-measures over 80% and AUC-ROC values around 79%. On the other hand, the Naive Bayes model seems to perform relatively poorly except when using the CK+OTHER metric suite, which achieved F-measure and AUC-ROC scores of around 60%.

By observing the results further, it becomes apparent that the combined metric suite CK+OTHER is superior in almost every score in all three classifiers. As expected the baseline metric suit consisting of only LOC produces the worst prediction results, although still maintaining above 60% F-measure and AUC-ROC. For DT and RF the OTHER metric suit outperforms the CK metric suit, while for NB it is the other way around.

Looking at the coverage of defects, which is indicated by the recall score, we can observe that in the better performing models DT and RF the recall is just under 80%, which is an impressive result when accompanied by precision values of around 85%.

RQ1 answer The aim of the single project case was to showcase the analysis of prediction results. The results achieved for the opengrok project indicate the potential of our defect prediction models, however, in order to reach a general conclusion we must first analyse the scoring metrics for each project in our dataset.

RQ2 Our aim is to find out which individual metrics (features) contribute in the largest extent to the quality of our prediction models. The metric with the best feature importance score can be considered the most suitable predictor since it influences the classification the most. It is important to note that multicollinearity can cause unreliable feature importance results, which is why the dataset has been preprocessed to reduce this problem as explained in Section 4.2. Because of this, metrics LOC, WMC and RFC will not be considered in our importance scoring. For each fold, the analysed metrics were ranked based on their feature importance. The rankings for each project were then grouped according to classifiers and plotted as box plots to describe the distribution of rankings for each metric separately. The rankings are defined in a way where the lower the rank the better the results, i.e. the best performing metric will achieve a ranking of 1.

From Figure 5.2 we can observe that the results vary slightly across the three classifiers, with DT and RF achieving very similar metrics rankings. In this project, the best overall ranking was achieved by the metrics NOC, NPA, LCOM5. On the other hand, metrics CBO, CD and NPM, were the least relevant for prediction as is indicated, by their poor feature importance ranking. It is important to note that the analysis is still to some degree influenced by the remaining multicollinearity. For example, the CBO metric ranked last in every single fold across classifiers, while achieving VIF value of 2.6, which was the highest value for this project and might have affected the ranking. On the other hand, the second most multicollinear feature NLE (2.2 VIF) achieved pretty reasonable results, on average being ranked around fifth, which shows that the metrics' importance rankings are not simply following the multicollinearity rankings.

RQ2 answer We can see that in the case of the project opengrok, the metrics most suitable for defect prediction were NOC, NPA and LCOM5 across all algorithms. The worst performing metric across all was found to be CBO, with CD and NPM being only sightly ahead,

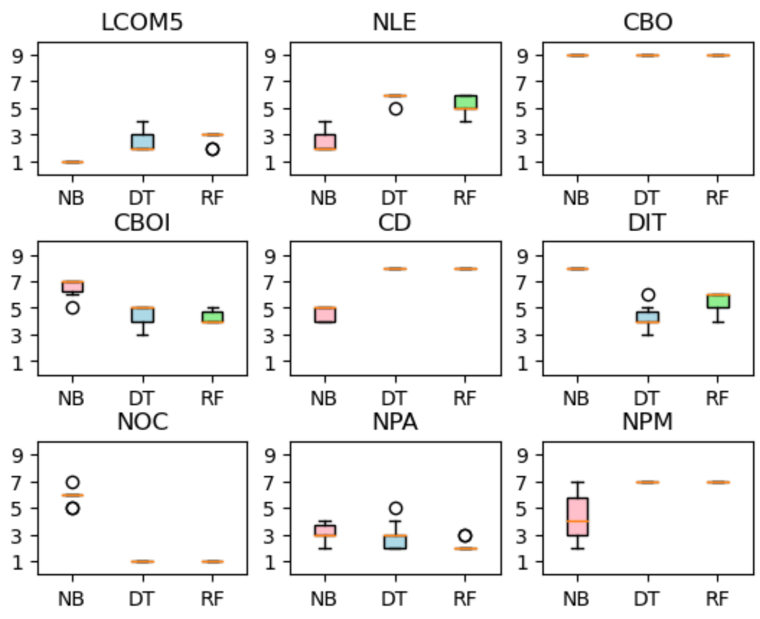

Figure 5.2: Rankings of metrics by importance in project opengrok

partly thanks to reasonable performance in the NB model. However, these are just exploratory results showcasing the feature importance analysis and need to be validated across all projects from our dataset.

5.2.2 Multi-project result analysis

Here we will discuss the prediction and importance ranking results in the scope of almost all projects[1].

RQ1 Regarding the performance of our defect prediction models, we decided to evaluate the results separately for each metric suite to be able to compare their effectiveness. For each scoring metric we provide a figure, containing box plots describing the distribution of results of each model(classifier). The F-measure, AUC-ROC, recall and precision are illustrated in **Figures 5.3, 5.4, 5.5, 5.6** respectively.

1. Because of prevalent multicollinearity (over 14 VIF) project Mindustry will not be considered during importance ranking analysis

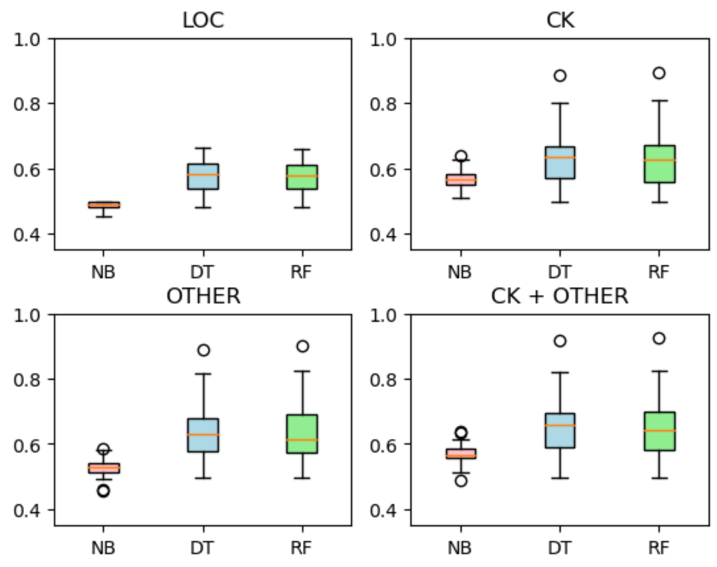

Figure 5.3: F-measure distribution grouped by metric suites (all projects)

Looking at the F-measure distribution in **Figure 5.3**, immediately we can observe that similarly to the single project case, in every metrics suite models DT and RF achieve very similar results, while NB is performing significantly worse overall. This confirms previous research suggesting the suitability of DT and RF models for defect prediction [8, 27]. By comparing the interquartile range (dispersion) and median (location) of boxplots of models DT and RF we can infer that using our defect dataset the DT is slightly superior, across all metric suites to RF in terms of F-measure.

Comparing the F-measure between metric suites, as expected LOC achieved by far the worst and least dispersed results. The differences between the distribution of results for suites CK, OTHER and CK+OTHER are much more subtle. Since the overall range of box plots is almost identical for all three, we focus on the location of the and skewness of the results. In the case of suites CK and OTHER the results are skewed to the lower end, while in the combined CK+OTHER metric the results are much more balanced. We conclude that the combined suite CK+OTHER performed the best in terms of F-measure.

Overall, the achieved F-measures were mostly between 50% and 80% which is indicative of the usefulness of software metrics for defect prediction, similar results were reached in [2].

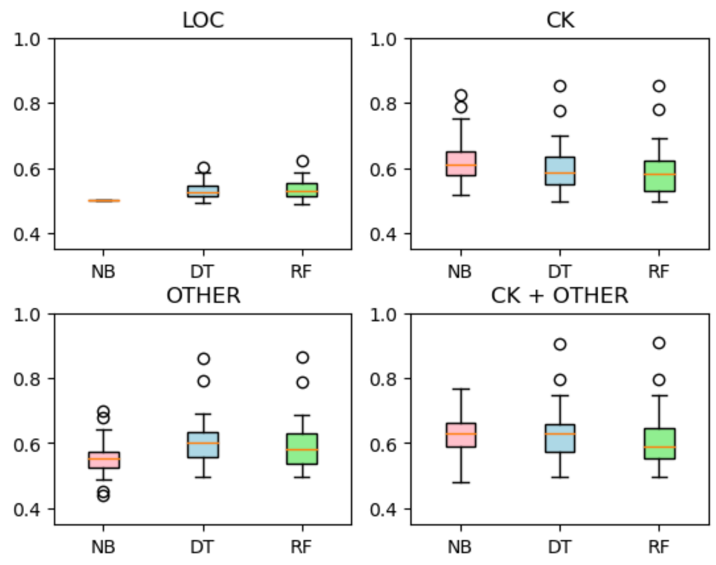

Figure 5.4: AUC-ROC distribution grouped by metric suites (all projects)

The AUC-ROC scores found in **Figure 5.4**, tell a slightly different story than the F-measure. In the case of LOC and OTHER suites the NB model is still underperforming, however, in suites CK and CK+OTHER NB model demonstrates slightly superior performance. However, it is important to note that the results for NB models are marginally more unstable, achieving also AUC-ROC marginally under 50% which brings concerns about the suitability of NB for defect prediction.

In terms of comparing AUC-ROC between metric suites, we can draw the same conclusions as in the case of F-measure, i.e. CK+OTHER performing the best, with CK close behind, OTHER slightly more behind and LOC performing the worst.

Overall, the achieved AUC-ROC scores were mostly between 50% and 75%, which is a promising result showing that the created dataset is suitable for defect prediction. However, this is marginally worse than the results achieved in some of the related research [8, 2], which achieved AUC-ROC scores ranging from around 60% to 80%.

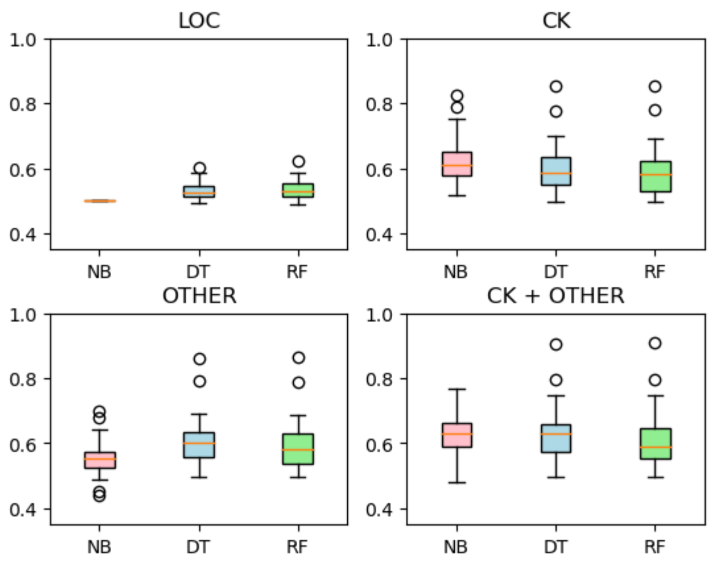

Figure 5.5: Recall distribution grouped by metric suites (all projects)

By inspecting the recall scores found in **Figure 5.5** we can get an idea of what bug coverage ratio our models achieved. In comparison with the previous box plots, the distributions are relatively similar and therefore the same observations apply. We will however note, that on average a recall between 50% and 70% was achieved, which is similar to results achieved in [8, 49].

Lastly, looking at the distributions of precision scores shown in **Figure 5.6**, we can observe that the results for DT and RF are more spread out than in previous scoring metrics, while on the other hand, the NB model achieved consistently poor precision. This explains why the F-measure scores indicated such poor results for the NB model, compared to the AUC-ROC scores. We can see that the best precision was achieved by RF in every metric suite except LOC, where DT performed better despite falling just short of RF in the three better performing metric suites. Once again CK+OTHER suite achieved the best overall precision.

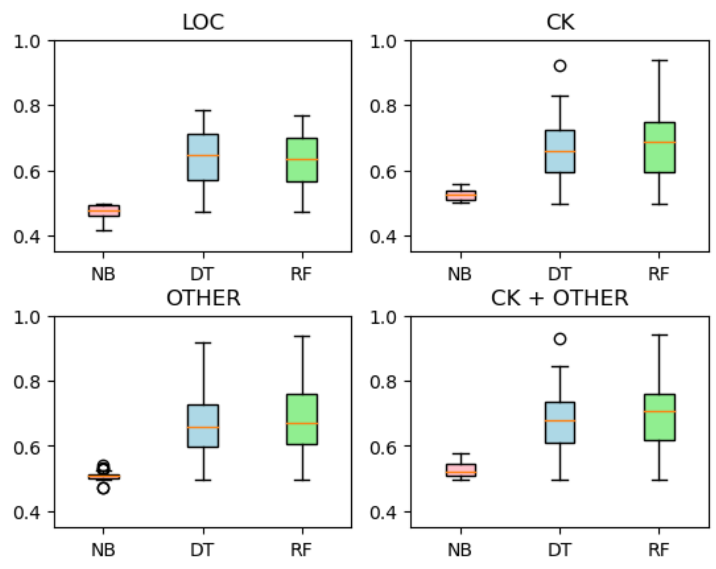

Figure 5.6: Precision distribution grouped by metric suites (all projects)

RQ1 answer As in the single project case, here we conclude that using code metrics in defect prediction models can produce meaningful results when a suitable combination of metrics and classifiers is found. In our case, the best results were achieved by DT and RF models using the CK+OTHER metric suite.

RQ2 To identify which metrics had the most positive influence on the defect prediction, just as in the single project case, we inspect the distributions of feature importance ranks of each metric for all suitable projects. The box plots of metric ranks for every project are available in **Appendix A.2**. Since we have found the models DT and RF to be superior to NB, we will consider mostly the rankings of these two models, which achieved similar results.

While carefully analysing the distributions of feature importance ranks for each metric across all projects we observed multiple trends. NOC is by far the best performing metric, being ranked first across the vast majority of instances. Although being less stable, metrics NPA, DIT and LCOM5 tend to rank between 2 and 4, which shows that these metrics together with NOC consistently have a significant positive influence on the defect prediction models.

Another stable trend observed is that CBO performed consistently worst, being ranked last across most projects. With slightly more variability metrics CD and NPM tended to consistently achieve ranks 6 to 8. This indicated that these metrics are lacking in predictive power and therefore contribute less to the overall defect prediction results. On the other hand, it could still be because of prevalent multicollinearity, which we addressed earlier in **Section 4.3**.

Metrics NLE and CBOI tended to rank in the middle, therefore no strong conclusion can be drawn in this case. However, these metrics seem to complement well with the best performers to form good prediction models.

RQ2 answer Similarly to the single project case of opengrok, we observe that the best performing metric across all projects is NOC, followed by NPA, DIT and LCOM5. Including these metrics in defect prediction models should lead to favourable results. On the other hand, the worst performing metric across all projects was CBO, followed by CD and NPM. These metrics seem to provide little information to defect prediction models and therefore should not be prioritized.

RQ3 This research question is heavily linked to the first two and has to a large degree been already discussed. As for the stability of defect prediction model's performance, we observed that the results were relatively stable with only a couple of positive outliers, which are acceptable.

The NB classifier tended to have inferior results to classifiers DT and RF, while also showing the most variability in feature importance rankings for individual metrics. While DT and RF models showcased some variability in achieved performance (especially in terms of precision) across projects, they were overall very stable in terms of feature importance rankings.

Concerning the stability of results in the scope of the metric suites, the achieved results are very stable. Since LOC was used as a baseline consisting of a single metric, it showcased the lowest variability. Overall the combined CK+OTHER metric suite consistently performed best, with CK and OTHER suites close behind.

The most stable results were probably achieved during the analysis of feature importance for individual metrics. Across projects, individual metrics displayed very little variability in their rankings.

RQ3 answer We found that prediction models DT and RF achieved relevant performance across all projects. Similarly, the differences in performance between metric suites were consistent across all projects. Overall, the ranking of individual metrics also produced very stable results with only marginal discrepancies.

5.3 Summary

Using the scoring metrics precision, recall, F-measure and AUC-ROC, we analysed the performance of our three defect prediction models. During the evaluation of prediction models built on software metrics as predictors, we observed, similarly to related works, that such defect prediction models are capable of achieving good prediction results(50% to 75% F-measure and AUC-ROC). In line with previous research [8, 40] we found that the models based on algorithm DT and RF tended to outperform the NB model. Regarding individual metrics we found that the consistently best predictors are metrics NOC, NPA, DIT and LCOM5, while in comparison metrics CBO, CD and NPM consistently under-performed.

6 Conclusion

The following chapter first discusses threats to the validity of our experiment. After that, we summarize the conclusions drawn from our results and provide possibilities of future improvement of our work.

6.1 Threats to validity

Threats to *construct validity* [50] concern the relationship between theory and observation. The first threat is concerning the way defects are linked to project versions and subsequently to classes. The problem is that we rely on commit messages to identify defect fixing commits, therefore any defect not mentioned in some commit messages cannot be identified using our approach. Additionally, there might be differences in commit message conventions across projects, which could add further inconsistency. However, this technique represents the state of the art in identifying defects in versioning system files [7, 51].

Another construct validity threat concerns the way defects are linked to release versions. Firstly, defects are assigned to versions based on the time of creation of the related issue. It would be desirable to be able to link defects with the version where they were introduced, this is information is, however, not available from issue tracking systems. Secondly, the way defects are aggregated to versions spread in a 6-month interval does not perfectly reflect reality. However, if defects were not aggregated in such a way, not enough defect data would be present in individual versions. Additionally, this approach has been traditionally used in defect prediction research [27, 2].

Threats to *internal validity* [52] concern the selection of tools and methods of analysis. Here we identify a threat regarding the analysis of feature importance of individual metrics. As discussed in Section 4.3, the results of this analysis are influenced by multicollinearity between metrics. To address this threat, with the help of the VIF measure, we removed the most problematic metrics from feature importance analysis, with the remaining metrics achieving around 2.5 VIF, which was deemed reasonable in related research [5].

Threats to *external validity* [52] concern the generalization of results. The dataset defined in this thesis consists of 275 release versions from 39 open-source Java projects. To minimize this threat, the selected projects come from a variety of different application domains, in addition to having different characteristics such as code size, number of classes or rate of defects. However, extending the dataset with projects in different programming languages or even proprietary projects would be desirable.

6.2 Final remarks

Here we will discuss the key points and results of this thesis, showing also how the thesis goals were achieved. At the beginning of the thesis we set the following goals:

Thesis goals

- Creating a dataset consisting of source code metrics and defect information.
- Using the dataset to train and evaluate bug prediction algorithms.
- Analysing performance of used algorithms.
- Evaluating which metrics are the best predictors.

A dataset consisting of 39 Java projects (275 release versions, spread 6-months apart) was created during this thesis. The necessary data was obtained by mining GitHub repositories of the selected projects. The creation process included the acquisition of defect information and project source codes, this data was used to calculate 12 class-level software metrics and identify which classes contained defects.

The defined dataset was used to train three bug prediction models based on algorithms Naive Bayes, Decision Tree and Random Forest. The performance of prediction models was evaluated using scoring metrics precision, recall, F-measure and AUC-ROC. Additionally, feature importance analysis was performed in order to identify software metrics with the most positive effect on defect prediction.

In Chapter 5, we discussed and answered the three research questions selected for this experiment. Overall, we found that using software metrics for defect prediction leads to favourable results. Prediction models based on algorithms Decision Tree and Random Forest achieved similar results, while the Naive Bayes prediction model proved to be less suitable for our dataset. Regarding the metrics with the most positive influence on the prediction, we found that metrics NOC, NPA, DIT and LCOM5 were consistently ranked as best performing across all projects. On the other hand, metrics CBO, CD, NPM were consitently ranked as the worst predictors.

6.3 Future work

There are multiple possibilities of future improvement of the quality and robustness of created dataset. For example, with slight changes the created tool can be extended to support projects in other programming languages in addition to Java, resulting in a more general sample of projects. Additionally, using a wider array of machine learning algorithms to validate the created dataset could be desirable.

Bibliography

1. CHIDAMBER, S.R.; KEMERER, C.F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 1994, vol. 20, no. 6, pp. 476–493. ISSN 1939-3520. Available from DOI: 10.1109/32.295895.
2. PALOMBA, Fabio; ZANONI, Marco; FONTANA, Francesca Arcelli; DE LUCIA, Andrea; OLIVETO, Rocco. Toward a Smell-Aware Bug Prediction Model. *IEEE Transactions on Software Engineering*. 2019, vol. 45, no. 2, pp. 194–218. Available from DOI: 10.1109/TSE.2017.2770122.
3. FENTON, Norman; BIEMAN, James. *Software metrics: a rigorous and practical approach*. CRC press, 2014. ISBN 9780429106224.
4. JURECZKO, Marian; MADEYSKI, Lech. Towards Identifying Software Project Clusters with Regard to Defect Prediction. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. Timişoara, Romania: Association for Computing Machinery, 2010. PROMISE '10. ISBN 9781450304047. Available from DOI: 10.1145/1868328.1868342.
5. TABA, Seyyed Ehsan Salamati; KHOMH, Foutse; ZOU, Ying; HASSAN, Ahmed E.; NAGAPPAN, Meiyappan. Predicting Bugs Using Antipatterns. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 270–279. ISSN 1063-6773. Available from DOI: 10.1109/ICSM.2013.38.
6. HASSAN, Ahmed E. Predicting faults using the complexity of code changes. In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 78–88. Available from DOI: 10.1109/ICSE.2009.5070510.
7. D'AMBROS, Marco; LANZA, Michele; ROBBES, Romain. An extensive comparison of bug prediction approaches. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 31–41. Available from DOI: 10.1109/MSR.2010.5463279.

8. FERENC, Rudolf; BÁN, Dénes; GRÓSZ, Tamás; GYIMÓTHY, Tibor. Deep learning in static, metric-based bug prediction. *Array*. 2020, vol. 6, p. 100021. ISSN 2590-0056. Available from DOI: `https://doi.org/10.1016/j.array.2020.100021`.

9. EMAM, Khaled El; MELO, Walcelio; MACHADO, Javam C. The Prediction of Faulty Classes Using Object-Oriented Design Metrics. *J. Syst. Softw.* 2001, vol. 56, no. 1, pp. 63–75. ISSN 0164-1212. Available from DOI: `10.1016/S0164-1212(00)00086-8`.

10. BELL, Robert M.; OSTRAND, Thomas J.; WEYUKER, Elaine J. Does Measuring Code Change Improve Fault Prediction? In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. Banff, Alberta, Canada: Association for Computing Machinery, 2011. Promise '11. ISBN 9781450307093. Available from DOI: `10.1145/2020390.2020392`.

11. GIGER, Emanuel; D'AMBROS, Marco; PINZGER, Martin; GALL, Harald C. Method-level bug prediction. In: *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 2012, pp. 171–180. ISSN 1949-3789. Available from DOI: `10.1145/2372251.2372285`.

12. MO, Ran; WEI, Shaozhi; FENG, Qiong; LI, Zengyang. An exploratory study of bug prediction at the method level. *Information and Software Technology*. 2022, vol. 144, p. 106794. ISSN 0950-5849. Available from DOI: `https://doi.org/10.1016/j.infsof.2021.106794`.

13. GRAVES, T.L.; KARR, A.F.; MARRON, J.S.; SIY, H. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*. 2000, vol. 26, no. 7, pp. 653–661. ISSN 1939-3520. Available from DOI: `10.1109/32.859533`.

14. BASILI, V.R.; BRIAND, L.C.; MELO, W.L. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*. 1996, vol. 22, no. 10, pp. 751–761. ISSN 1939-3520. Available from DOI: `10.1109/32.544352`.

15. WALKINSHAW, N. *Software Quality Assurance: Consistency in the Face of Complexity and Change*. Springer International Publishing, 2017. Undergraduate Topics in Computer Science. ISBN

9783319648224. Available also from: `https://books.google.sk/books?id=UxsuDwAAQBAJ`.

16. TRIPATHY, P.; NAIK, K. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2011. ISBN 9781118211632. Available also from: `https://books.google.sk/books?id=neWaoJKSkvgC`.
17. SINGH, Gurdev; SINGH, Dilbag; SINGH, Vikram. A Study of Software Metrics. 2011, vol. 37, pp. 2230–7893.
18. PRASAD, Manjula.C.M.; FLORENCE, Lilly; ARYA, Arti. A Study on Software Metrics based Software Defect Prediction using Data Mining and Machine Learning Techniques. *International Journal of Database Theory and Application*. 2015, vol. 7, pp. 179–190. Available from DOI: `10.14257/ijdta.2015.8.3.15`.
19. SHEETAL, Phani; KONGARA, Ravindranath. Software metric evaluation on cloud based applications. *International Journal of Engineering and Technology(UAE)*. 2018, vol. 7, pp. 13–18. Available from DOI: `10.14419/ijet.v7i1.5.9071`.
20. CHEN, Yuan; SHEN, Xiang-heng; DU, Peng; GE, Bing. Research on software defect prediction based on data mining. In: *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*. 2010, vol. 1, pp. 563–567. Available from DOI: `10.1109/ICCAE.2010.5451355`.
21. GROUP, ISDW et al. 1044-2009-IEEE Standard Classification for Software Anomalies. *IEEE, New York*. 2010.
22. FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 2018. ISBN 978-0-13-475759-9.
23. PALOMBA, Fabio; BAVOTA, Gabriele; DI PENTA, Massimiliano; OLIVETO, Rocco; DE LUCIA, Andrea; POSHYVANYK, Denys. Detecting bad smells in source code using change history information. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 268–278. Available from DOI: `10.1109/ASE.2013.6693086`.

24. FONTANA, Francesca Arcelli; ZANONI, Marco; MARINO, Alessandro; MÄNTYLÄ, Mika V. Code Smell Detection: Towards a Machine Learning-Based Approach. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 396–399. ISSN 1063-6773. Available from DOI: `10.1109/ICSM.2013.56`.

25. MOHA, Naouel; GUÉHÉNEUC, Yann-Gaël; DUCHIEN, Laurence; LE MEUR, Anne-Françoise. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*. 2010, vol. 36, no. 1, pp. 20–36. Available also from: `https://hal.inria.fr/inria-00538476`.

26. LI, Zhiqiang; JING, Xiao-Yuan; ZHU, Xiaoke. Progress on approaches to software defect prediction. *Iet Software*. 2018, vol. 12, no. 3, pp. 161–175.

27. TÓTH, Zoltán; GYIMESI, Péter; FERENC, Rudolf. A Public Bug Database of GitHub Projects and Its Application in Bug Prediction. In: *Proceedings of the 16th International Conference on Computational Science and Its Applications (ICCSA 2016)*. Beijing, China: Springer International Publishing, 2016, pp. 625–638. Available from DOI: `10.1007/978-3-319-42089-9_44`.

28. XU, Zhou; LI, Li; YAN, Meng; LIU, Jin; LUO, Xiapu; GRUNDY, John; ZHANG, Yifeng; ZHANG, Xiaohong. A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software*. 2021, vol. 172, p. 110862. ISSN 0164-1212. Available from DOI: `https://doi.org/10.1016/j.jss.2020.110862`.

29. WAHONO, Romi Satria. A systematic literature review of software defect prediction. *Journal of software engineering*. 2015, vol. 1, no. 1, pp. 1–16.

30. SARKER, Iqbal H. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Computer Science*. 2021, vol. 2, no. 3, p. 160. ISSN 2661-8907. Available from DOI: `10.1007/s42979-021-00592-x`.

31. WITTEN, Ian H; FRANK, Eibe. Data mining: practical machine learning tools and techniques with Java implementations. *Acm Sigmod Record*. 2002, vol. 31, no. 1, pp. 76–77.

32. GYIMESI, Péter. Automatic Calculation of Process Metrics and their Bug Prediction Capabilities. *Acta Cybernetica*. 2017, vol. 23, pp. 537–559. Available from DOI: `10.14232/actacyb.23.2.2017.7`.

33. FONTANA, Francesca Arcelli; FERME, Vincenzo; ZANONI, Marco; ROVEDA, Riccardo. Towards a prioritization of code debt: A code smell Intensity Index. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 2015, pp. 16–24. Available from DOI: `10.1109/MTD.2015.7332620`.

34. GUÉHÉNEUC, Yann-Gaël; ANTONIOL, Giuliano. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Transactions on Software Engineering*. 2008, vol. 34, no. 5, pp. 667–684. Available from DOI: `10.1109/TSE.2008.48`.

35. GRECHANIK, Mark; MCMILLAN, Collin; DEFERRARI, Luca; COMI, Marco; CRESPI REGHIZZI, Stefano; POSHYVANYK, Denys; FU, Chen; XIE, Qing; GHEZZI, Carlo. An empirical investigation into a large-scale Java open source code repository. In: 2010. Available from DOI: `10.1145/1852786.1852801`.

36. DEMEYER, Serge; TICHELAAR, S.; DUCASSE, Stéphane. FAMIX 2. 1-the FAMOOS information exchange model. 2001.

37. HINTON, G. E.; SALAKHUTDINOV, R. R. Reducing the Dimensionality of Data with Neural Networks. *Science*. 2006, vol. 313, no. 5786, pp. 504–507. Available from DOI: `10.1126/science.1127647`.

38. BRIAND, L.C.; WUST, J.; IKONOMOVSKI, S.V.; LOUNIS, H. Investigating quality factors in object-oriented designs: an industrial case study. In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 1999, pp. 345–354. Available from DOI: `10.1145/302405.302654`.

39. MOSER, Raimund; PEDRYCZ, Witold; SUCCI, Giancarlo. Analysis of the Reliability of a Subset of Change Metrics for Defect Prediction. In: *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. Kaiserslautern, Germany: Association for Computing Machinery, 2008, pp. 309–311. ESEM '08. ISBN 9781595939715. Available from DOI: `10.1145/1414004.1414063`.

40. FERENC, Rudolf; GYIMESI, Péter; GYIMESI, Gábor; TÓTH, Zoltán; GYIMÓTHY, Tibor. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software*. 2020, vol. 169, p. 110691. ISSN 0164-1212. Available from DOI: `https://doi.org/10.1016/j.jss.2020.110691`.

41. SAYYAD SHIRABAD, J.; MENZIES, T.J. *The PROMISE Repository of Software Engineering Databases.* [School of Information Technology and Engineering, University of Ottawa, Canada]. 2005. Available also from: `http://promise.site.uottawa.ca/SERepository`.

42. ZIMMERMANN, Thomas; PREMRAJ, Rahul; ZELLER, Andreas. Predicting Defects for Eclipse. In: *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. 2007, pp. 9–9. Available from DOI: `10.1109/PROMISE.2007.10`.

43. GYIMOTHY, T.; FERENC, R.; SIKET, I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*. 2005, vol. 31, no. 10, pp. 897–910. Available from DOI: `10.1109/TSE.2005.112`.

44. WANG, Huanjing; KHOSHGOFTAAR, Taghi; SELIYA, Naeem. How Many Software Metrics Should be Selected for Defect Prediction? In: 2011.

45. BREIMAN, Leo. Random Forests. Statistics Department. *University of California, Berkeley, CA*. 2001, vol. 4720.

46. TANTITHAMTHAVORN, Chakkrit; MCINTOSH, Shane; HASSAN, Ahmed E.; MATSUMOTO, Kenichi. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering*. 2019, vol. 45, no. 7, pp. 683–711. Available from DOI: `10.1109/TSE.2018.2794977`.

47. HAN, J.; PEI, J.; KAMBER, M. *Data Mining: Concepts and Techniques*. Elsevier Science, 2011. The Morgan Kaufmann Series in Data Management Systems. ISBN 9780123814807. Available also from: `https://books.google.sk/books?id=pQws07tdpjoC`.

48. FAWCETT, Tom. An introduction to ROC analysis. *Pattern Recognition Letters*. 2006, vol. 27, no. 8, pp. 861–874. ISSN 0167-8655. Available from DOI: `https://doi.org/10.1016/j.patrec.2005.10.010`. ROC Analysis in Pattern Recognition.

49. KORU, A.G.; LIU, H. Building effective defect-prediction models in practice. *IEEE Software*. 2005, vol. 22, no. 6, pp. 23–29. Available from DOI: `10.1109/MS.2005.149`.

50. RALPH, Paul; TEMPERO, Ewan. Construct Validity in Software Engineering Research and Software Metrics. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. Christchurch, New Zealand: Association for Computing Machinery, 2018, pp. 13–23. EASE'18. ISBN 9781450364034. Available from DOI: `10.1145/3210459.3210461`.

51. FISCHER, M.; PINZGER, M.; GALL, H. Populating a Release History Database from version control and bug tracking systems. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* 2003, pp. 23–32. Available from DOI: `10.1109/ICSM.2003.1235403`.

52. SIEGMUND, Janet; SIEGMUND, Norbert; APEL, Sven. Views on Internal and External Validity in Empirical Software Engineering. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Florence, Italy: IEEE Press, 2015, pp. 9–19. ICSE '15. ISBN 9781479919345.

A All results

A.1 All prediction model scores

Table A.1: All Naive Bayes results

Project	LOC				CK				OTHER				CK+OTHER			
	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R
A-U-I-L	0.467	0.5	0.483	0.5	0.535	0.621	0.574	0.621	0.534	0.634	0.579	0.634	0.579	0.718	0.64	0.718
antlr4	0.478	0.5	0.488	0.5	0.545	0.714	0.618	0.714	0.511	0.566	0.537	0.566	0.528	0.646	0.58	0.646
BroadleafCor	0.46	0.5	0.479	0.5	0.541	0.626	0.58	0.626	0.512	0.539	0.525	0.539	0.548	0.624	0.583	0.624
ceylon-ide-eclipse	0.484	0.5	0.492	0.5	0.524	0.617	0.567	0.617	0.513	0.548	0.529	0.548	0.529	0.607	0.565	0.607
checkstyle	0.473	0.5	0.486	0.5	0.545	0.653	0.594	0.653	0.529	0.644	0.581	0.644	0.55	0.687	0.611	0.687
Discord4J	0.486	0.5	0.493	0.5	0.537	0.79	0.638	0.79	0.51	0.586	0.537	0.586	0.547	0.766	0.635	0.766
dubbo	0.469	0.5	0.484	0.5	0.505	0.517	0.511	0.517	0.503	0.509	0.506	0.509	0.505	0.517	0.511	0.517
elasticsearch	0.449	0.5	0.473	0.5	0.543	0.597	0.569	0.597	0.508	0.52	0.514	0.52	0.524	0.565	0.544	0.565
ExoPlayer	0.457	0.5	0.478	0.5	0.54	0.6	0.568	0.6	0.523	0.568	0.544	0.568	0.562	0.658	0.606	0.658
fastjson	0.499	0.5	0.499	0.5	0.519	0.75	0.612	0.75	0.504	0.677	0.575	0.677	0.508	0.757	0.606	0.757
find-sec-bugs	0.493	0.5	0.497	0.5	0.508	0.605	0.55	0.605	0.5	0.503	0.499	0.503	0.507	0.601	0.547	0.601
framework	0.493	0.5	0.496	0.5	0.53	0.752	0.622	0.752	0.512	0.643	0.57	0.643	0.513	0.7	0.592	0.7
guava	0.497	0.5	0.499	0.5	0.504	0.646	0.566	0.646	0.502	0.553	0.526	0.553	0.504	0.652	0.568	0.652
hazelcast	0.483	0.5	0.492	0.5	0.53	0.647	0.583	0.647	0.509	0.557	0.532	0.557	0.521	0.638	0.574	0.638
incubator-hugegraph	0.47	0.5	0.484	0.5	0.533	0.615	0.571	0.615	0.51	0.542	0.525	0.542	0.528	0.611	0.566	0.611
jetty.project	0.458	0.5	0.478	0.5	0.524	0.565	0.544	0.565	0.529	0.571	0.549	0.571	0.528	0.588	0.556	0.588
jmonkeyengi	0.497	0.5	0.499	0.5	0.504	0.629	0.558	0.629	0.502	0.564	0.528	0.564	0.503	0.622	0.554	0.622
jsoup	0.496	0.5	0.498	0.5	0.514	0.825	0.626	0.825	0.506	0.699	0.587	0.699	0.506	0.705	0.589	0.705
junit4	0.489	0.5	0.495	0.5	0.509	0.563	0.534	0.563	0.506	0.574	0.536	0.574	0.511	0.629	0.564	0.629
junit5	0.484	0.5	0.492	0.5	0.514	0.597	0.552	0.597	0.498	0.489	0.493	0.489	0.514	0.603	0.555	0.603
kura	0.492	0.5	0.496	0.5	0.505	0.564	0.532	0.564	0.505	0.572	0.534	0.572	0.512	0.678	0.583	0.678
mapdb	0.468	0.5	0.483	0.5	0.53	0.611	0.568	0.611	0.502	0.508	0.505	0.508	0.521	0.585	0.551	0.585
mcMMO	0.415	0.5	0.454	0.5	0.535	0.558	0.546	0.558	0.471	0.451	0.46	0.451	0.544	0.57	0.557	0.57
Mindustry	0.458	0.5	0.478	0.5	0.534	0.6	0.565	0.6	0.519	0.561	0.539	0.561	0.517	0.554	0.534	0.554
neo4j	0.499	0.5	0.499	0.5	0.502	0.638	0.561	0.638	0.5	0.531	0.514	0.531	0.502	0.633	0.559	0.633
netty	0.458	0.5	0.478	0.5	0.52	0.549	0.534	0.549	0.524	0.577	0.549	0.577	0.53	0.591	0.559	0.591
opengrok	0.438	0.5	0.467	0.5	0.537	0.578	0.557	0.578	0.474	0.441	0.456	0.441	0.566	0.627	0.595	0.627
orientdb	0.473	0.5	0.486	0.5	0.531	0.576	0.553	0.576	0.51	0.532	0.52	0.532	0.553	0.598	0.575	0.598
oryx	0.456	0.5	0.477	0.5	0.528	0.578	0.552	0.578	0.542	0.63	0.583	0.63	0.556	0.673	0.609	0.673
paho.mqtt.jav	0.46	0.5	0.479	0.5	0.558	0.658	0.603	0.658	0.498	0.498	0.497	0.498	0.544	0.632	0.585	0.632
pmd	0.476	0.5	0.488	0.5	0.511	0.551	0.53	0.551	0.508	0.539	0.523	0.539	0.497	0.481	0.488	0.481
powermock	0.49	0.5	0.495	0.5	0.503	0.53	0.515	0.53	0.5	0.496	0.497	0.496	0.507	0.557	0.53	0.557
presto	0.498	0.5	0.499	0.5	0.502	0.602	0.546	0.602	0.501	0.531	0.513	0.531	0.503	0.637	0.56	0.637
RxJava	0.495	0.5	0.497	0.5	0.508	0.629	0.561	0.629	0.512	0.613	0.557	0.613	0.507	0.631	0.561	0.631
shardingsphe elasticjob	0.471	0.5	0.485	0.5	0.544	0.675	0.602	0.675	0.507	0.529	0.516	0.529	0.552	0.691	0.613	0.691
spring-boot	0.463	0.5	0.481	0.5	0.523	0.57	0.546	0.57	0.516	0.557	0.535	0.557	0.531	0.586	0.557	0.586
Terasology	0.496	0.5	0.498	0.5	0.506	0.662	0.572	0.662	0.502	0.571	0.533	0.571	0.505	0.655	0.57	0.655
titan	0.492	0.5	0.496	0.5	0.516	0.679	0.584	0.679	0.504	0.545	0.519	0.545	0.515	0.667	0.578	0.667

Project	LOC				CK				OTHER				CK+OTHER			
	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R
webcam-capture	0.457	0.5	0.478	0.5	0.539	0.58	0.558	0.58	0.497	0.494	0.495	0.494	0.532	0.552	0.542	0.552

Table A.2: All Decision Tree results

Project	LOC				CK				OTHER				CK+OTHER			
	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R
A-U-I-L	0.562	0.529	0.54	0.529	0.708	0.649	0.671	0.649	0.658	0.606	0.629	0.606	0.719	0.668	0.69	0.668
antlr4	0.691	0.586	0.629	0.586	0.69	0.636	0.661	0.636	0.697	0.656	0.674	0.656	0.679	0.638	0.657	0.638
Broadleaf Com-merce	0.713	0.54	0.613	0.54	0.658	0.567	0.609	0.567	0.642	0.564	0.6	0.564	0.64	0.568	0.602	0.568
ceylon-ide-eclipse	0.631	0.524	0.571	0.524	0.624	0.573	0.597	0.573	0.626	0.565	0.594	0.565	0.617	0.581	0.598	0.581
checkstyle	0.706	0.548	0.616	0.548	0.775	0.653	0.709	0.653	0.76	0.642	0.696	0.642	0.769	0.684	0.724	0.684
Discord4J	0.553	0.526	0.536	0.526	0.579	0.544	0.558	0.544	0.605	0.576	0.587	0.576	0.537	0.524	0.53	0.524
dubbo	0.642	0.516	0.571	0.516	0.594	0.552	0.572	0.552	0.59	0.552	0.571	0.552	0.604	0.573	0.588	0.573
elasticsearch	0.73	0.529	0.613	0.529	0.73	0.634	0.678	0.634	0.701	0.625	0.661	0.625	0.716	0.654	0.684	0.654
ExoPlayer	0.729	0.571	0.64	0.571	0.693	0.634	0.662	0.634	0.677	0.632	0.654	0.632	0.688	0.649	0.668	0.649
fastjson	0.694	0.577	0.626	0.577	0.766	0.667	0.711	0.667	0.783	0.685	0.729	0.685	0.775	0.68	0.721	0.68
find-sec-bugs	0.493	0.499	0.496	0.499	0.539	0.52	0.528	0.52	0.623	0.589	0.604	0.589	0.686	0.64	0.661	0.64
framework	0.779	0.577	0.662	0.577	0.922	0.852	0.885	0.852	0.916	0.861	0.888	0.861	0.928	0.905	0.917	0.905
guava	0.497	0.5	0.499	0.5	0.6	0.542	0.57	0.542	0.589	0.548	0.567	0.548	0.616	0.567	0.59	0.567
hazelcast	0.645	0.514	0.57	0.514	0.596	0.564	0.579	0.564	0.584	0.549	0.566	0.549	0.601	0.578	0.589	0.578
hugegraph	0.649	0.531	0.58	0.531	0.643	0.585	0.612	0.585	0.585	0.552	0.568	0.552	0.634	0.592	0.612	0.592
jetty.project	0.732	0.521	0.608	0.521	0.647	0.562	0.602	0.562	0.649	0.559	0.6	0.559	0.627	0.562	0.593	0.562
jmonkeyengi	0.577	0.516	0.539	0.516	0.775	0.603	0.676	0.603	0.685	0.591	0.634	0.591	0.734	0.63	0.677	0.63
jsoup	0.496	0.497	0.497	0.497	0.496	0.497	0.497	0.497	0.496	0.495	0.496	0.495	0.496	0.496	0.496	0.496
junit4	0.723	0.516	0.592	0.516	0.771	0.589	0.667	0.589	0.807	0.628	0.705	0.628	0.792	0.642	0.708	0.642
junit5	0.701	0.519	0.591	0.519	0.554	0.524	0.539	0.524	0.557	0.524	0.54	0.524	0.556	0.531	0.543	0.531
kura	0.571	0.509	0.536	0.509	0.719	0.629	0.67	0.629	0.719	0.613	0.661	0.613	0.753	0.654	0.699	0.654
mapdb	0.706	0.578	0.634	0.578	0.705	0.633	0.666	0.633	0.742	0.641	0.687	0.641	0.725	0.672	0.697	0.672
mcMMO	0.61	0.555	0.581	0.555	0.66	0.642	0.65	0.642	0.619	0.602	0.61	0.602	0.645	0.635	0.64	0.635
Mindustry	0.648	0.541	0.589	0.541	0.579	0.549	0.564	0.549	0.617	0.606	0.611	0.606	0.615	0.599	0.607	0.599
neo4j	0.617	0.518	0.559	0.518	0.736	0.682	0.708	0.682	0.772	0.665	0.713	0.665	0.794	0.748	0.77	0.748
netty	0.675	0.53	0.591	0.53	0.631	0.566	0.597	0.566	0.627	0.556	0.589	0.556	0.647	0.573	0.607	0.573
opengrok	0.684	0.602	0.64	0.602	0.827	0.775	0.8	0.775	0.846	0.79	0.817	0.79	0.845	0.795	0.819	0.795
orientdb	0.714	0.544	0.617	0.544	0.687	0.618	0.65	0.618	0.685	0.616	0.649	0.616	0.688	0.632	0.659	0.632
oryx	0.786	0.565	0.655	0.565	0.799	0.699	0.743	0.699	0.741	0.689	0.713	0.689	0.755	0.7	0.725	0.7
paho.mqtt	0.722	0.549	0.62	0.549	0.727	0.603	0.659	0.603	0.743	0.634	0.683	0.634	0.759	0.661	0.705	0.661
pmd	0.631	0.505	0.554	0.505	0.653	0.549	0.597	0.549	0.711	0.655	0.682	0.655	0.711	0.67	0.69	0.67
powermock	0.721	0.531	0.606	0.531	0.71	0.577	0.636	0.577	0.701	0.593	0.642	0.593	0.665	0.581	0.619	0.581
presto	0.554	0.512	0.531	0.512	0.605	0.578	0.591	0.578	0.584	0.556	0.569	0.556	0.602	0.594	0.598	0.594
RxJava	0.565	0.509	0.534	0.509	0.559	0.522	0.539	0.522	0.57	0.53	0.549	0.53	0.564	0.531	0.547	0.531
shardingsphe elasticjob	0.471	0.493	0.482	0.493	0.554	0.551	0.552	0.551	0.618	0.57	0.592	0.57	0.594	0.574	0.584	0.574
spring-boot	0.709	0.521	0.6	0.521	0.716	0.587	0.645	0.587	0.715	0.605	0.655	0.605	0.734	0.644	0.686	0.644
Terasology	0.496	0.5	0.498	0.5	0.584	0.543	0.562	0.543	0.571	0.543	0.556	0.543	0.553	0.529	0.54	0.529
titan	0.542	0.506	0.519	0.506	0.539	0.527	0.533	0.527	0.548	0.537	0.542	0.537	0.548	0.544	0.546	0.544

Project	LOC				CK				OTHER				CK+OTHER			
	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R
webcam-capture	0.611	0.53	0.559	0.53	0.705	0.612	0.651	0.612	0.733	0.634	0.679	0.634	0.727	0.642	0.681	0.642

Table A.3: All Random Forest results

Project	LOC				CK				OTHER				CK+OTHER			
	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R
A-U-I-L	0.562	0.529	0.54	0.529	0.585	0.583	0.582	0.583	0.668	0.581	0.615	0.581	0.619	0.584	0.596	0.584
antlr4	0.691	0.586	0.629	0.586	0.72	0.629	0.67	0.629	0.689	0.625	0.654	0.625	0.71	0.647	0.676	0.647
Broadleaf Commerce	0.695	0.549	0.612	0.549	0.676	0.575	0.621	0.575	0.642	0.565	0.601	0.565	0.645	0.567	0.603	0.567
ceylon-ide-eclipse	0.631	0.528	0.574	0.528	0.612	0.539	0.573	0.539	0.639	0.552	0.592	0.552	0.621	0.545	0.58	0.545
checkstyle	0.703	0.558	0.622	0.558	0.773	0.647	0.705	0.647	0.782	0.642	0.705	0.642	0.791	0.684	0.734	0.684
Discord4J	0.553	0.526	0.536	0.526	0.528	0.527	0.527	0.527	0.512	0.529	0.52	0.529	0.511	0.513	0.512	0.513
dubbo	0.644	0.518	0.573	0.518	0.595	0.541	0.566	0.541	0.609	0.545	0.575	0.545	0.627	0.563	0.593	0.563
elasticsearch	0.72	0.533	0.612	0.533	0.737	0.631	0.68	0.631	0.724	0.621	0.668	0.621	0.735	0.644	0.687	0.644
ExoPlayer	0.699	0.573	0.63	0.573	0.717	0.623	0.667	0.623	0.725	0.635	0.677	0.635	0.738	0.647	0.689	0.647
fastjson	0.691	0.577	0.625	0.577	0.829	0.693	0.752	0.693	0.792	0.676	0.726	0.676	0.77	0.688	0.726	0.688
find-sec-bugs	0.493	0.499	0.496	0.499	0.539	0.52	0.528	0.52	0.671	0.562	0.607	0.562	0.71	0.591	0.641	0.591
framework	0.77	0.579	0.66	0.579	0.939	0.854	0.894	0.854	0.939	0.864	0.9	0.864	0.943	0.911	0.927	0.911
guava	0.526	0.502	0.513	0.502	0.599	0.526	0.56	0.526	0.623	0.532	0.574	0.532	0.62	0.537	0.575	0.537
hazelcast	0.634	0.517	0.568	0.517	0.602	0.549	0.574	0.549	0.603	0.541	0.571	0.541	0.613	0.56	0.585	0.56
hugegraph	0.613	0.54	0.572	0.54	0.585	0.523	0.552	0.523	0.619	0.532	0.57	0.532	0.619	0.538	0.575	0.538
jetty.project	0.723	0.523	0.606	0.523	0.645	0.571	0.606	0.571	0.645	0.568	0.604	0.568	0.631	0.577	0.602	0.577
jmonkeyengi	0.622	0.532	0.568	0.532	0.769	0.598	0.668	0.598	0.704	0.581	0.635	0.581	0.75	0.63	0.684	0.63
jsoup	0.496	0.497	0.497	0.497	0.496	0.497	0.497	0.497	0.496	0.497	0.497	0.497	0.496	0.498	0.497	0.498
junit4	0.719	0.516	0.59	0.516	0.771	0.597	0.671	0.597	0.797	0.624	0.699	0.624	0.779	0.649	0.708	0.649
junit5	0.7	0.52	0.591	0.52	0.563	0.52	0.54	0.52	0.54	0.511	0.524	0.511	0.566	0.523	0.543	0.523
kura	0.57	0.513	0.539	0.513	0.756	0.622	0.681	0.622	0.768	0.609	0.678	0.609	0.787	0.638	0.703	0.638
mapdb	0.704	0.584	0.637	0.584	0.757	0.644	0.695	0.644	0.797	0.638	0.707	0.638	0.781	0.661	0.715	0.661
mcMMO	0.598	0.564	0.58	0.564	0.686	0.617	0.649	0.617	0.605	0.563	0.583	0.563	0.691	0.63	0.658	0.63
Mindustry	0.636	0.547	0.587	0.547	0.57	0.528	0.548	0.528	0.623	0.564	0.592	0.564	0.622	0.564	0.592	0.564
neo4j	0.617	0.518	0.559	0.518	0.832	0.683	0.749	0.683	0.866	0.657	0.745	0.657	0.894	0.748	0.814	0.748
netty	0.649	0.532	0.584	0.532	0.632	0.567	0.597	0.567	0.606	0.547	0.575	0.547	0.618	0.563	0.59	0.563
opengrok	0.694	0.622	0.656	0.622	0.841	0.779	0.808	0.779	0.864	0.788	0.824	0.788	0.858	0.796	0.826	0.796
orientdb	0.699	0.546	0.612	0.546	0.704	0.597	0.646	0.597	0.698	0.599	0.645	0.599	0.709	0.616	0.659	0.616
oryx	0.731	0.572	0.641	0.572	0.81	0.674	0.734	0.674	0.786	0.687	0.732	0.687	0.792	0.684	0.733	0.684
paho.mqtt	0.696	0.558	0.618	0.558	0.721	0.605	0.658	0.605	0.751	0.642	0.691	0.642	0.759	0.658	0.703	0.658
pmd	0.608	0.505	0.548	0.505	0.667	0.547	0.601	0.547	0.747	0.638	0.688	0.638	0.746	0.654	0.696	0.654
powermock	0.713	0.538	0.608	0.538	0.71	0.582	0.639	0.582	0.704	0.586	0.639	0.586	0.675	0.583	0.625	0.583
presto	0.554	0.512	0.531	0.512	0.722	0.557	0.626	0.557	0.556	0.516	0.534	0.516	0.724	0.573	0.635	0.573
RxJava	0.557	0.509	0.531	0.509	0.599	0.521	0.555	0.521	0.549	0.513	0.529	0.513	0.593	0.526	0.557	0.526
shardingsphe elasticjob	0.471	0.49	0.48	0.49	0.545	0.524	0.532	0.524	0.581	0.534	0.554	0.534	0.586	0.543	0.562	0.543
spring-boot	0.688	0.525	0.595	0.525	0.711	0.588	0.644	0.588	0.717	0.6	0.653	0.6	0.734	0.631	0.678	0.631
Terasology	0.496	0.5	0.498	0.5	0.597	0.524	0.554	0.524	0.612	0.53	0.566	0.53	0.583	0.52	0.547	0.52
titan	0.542	0.506	0.519	0.506	0.559	0.522	0.538	0.522	0.538	0.522	0.529	0.522	0.567	0.53	0.547	0.53

Project	LOC				CK				OTHER				CK+OTHER			
	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R	Prec	Rec	F-m	A-R
webcam-capture	0.609	0.53	0.558	0.53	0.731	0.598	0.65	0.598	0.794	0.628	0.698	0.628	0.758	0.618	0.678	0.618

A.2 All importance ranking boxplots

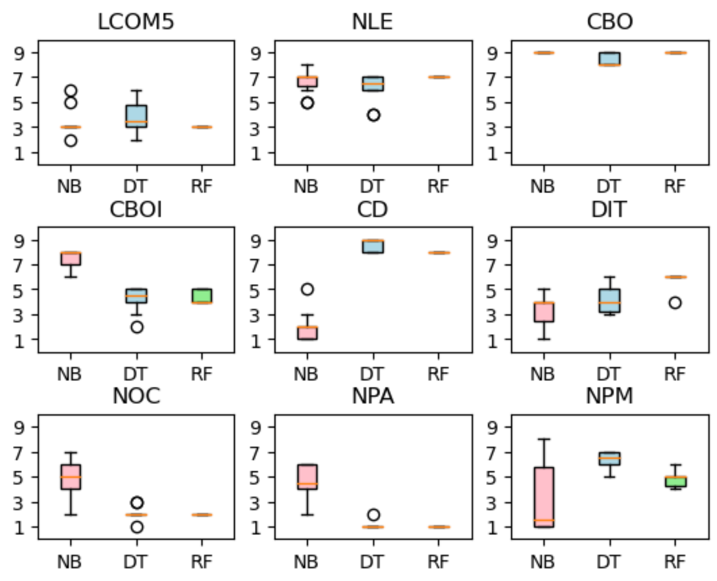

Figure A.1: Rankings of metrics by importance in project Android-Universal-Image-Loader

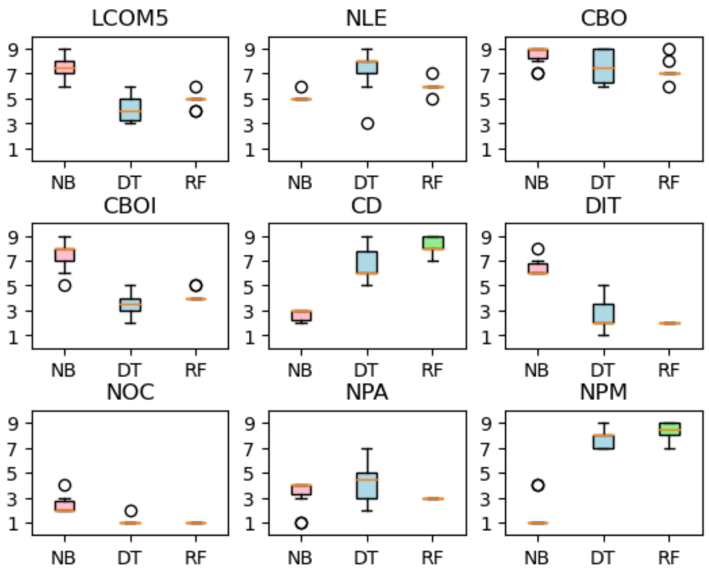

Figure A.2: Rankings of metrics by importance in project antlr4

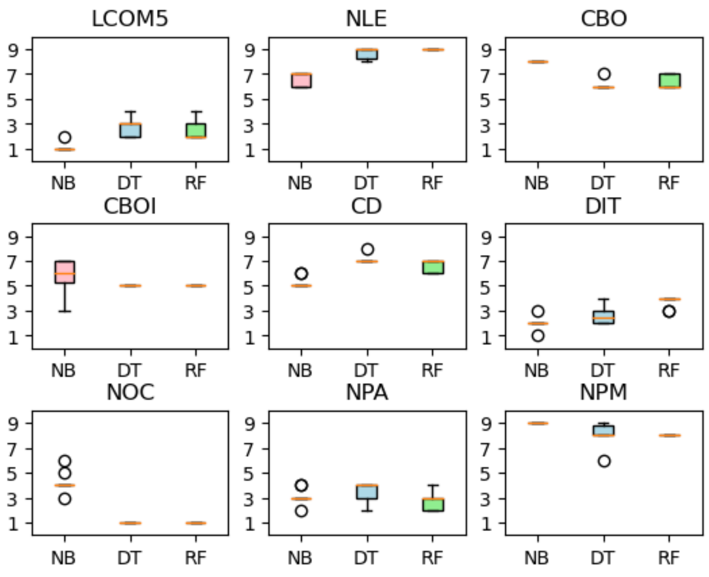

Figure A.3: Rankings of metrics by importance in project Broadleaf-Commerce

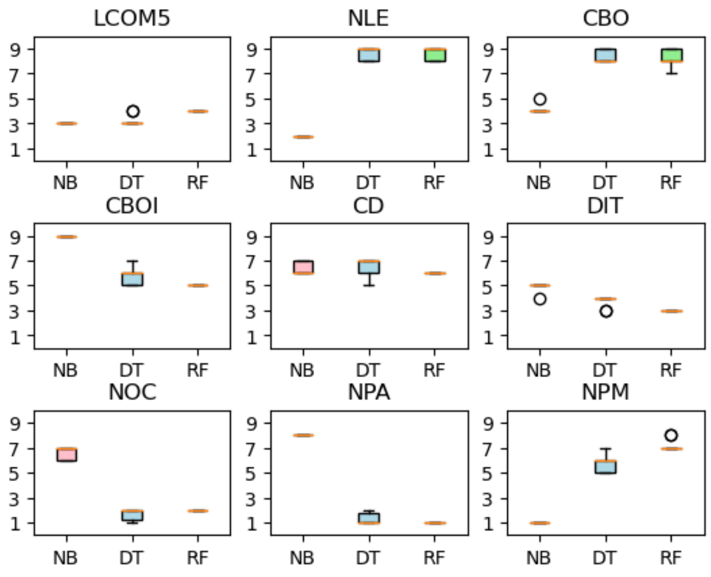

Figure A.4: Rankings of metrics by importance in project ceylon-ide-eclipse

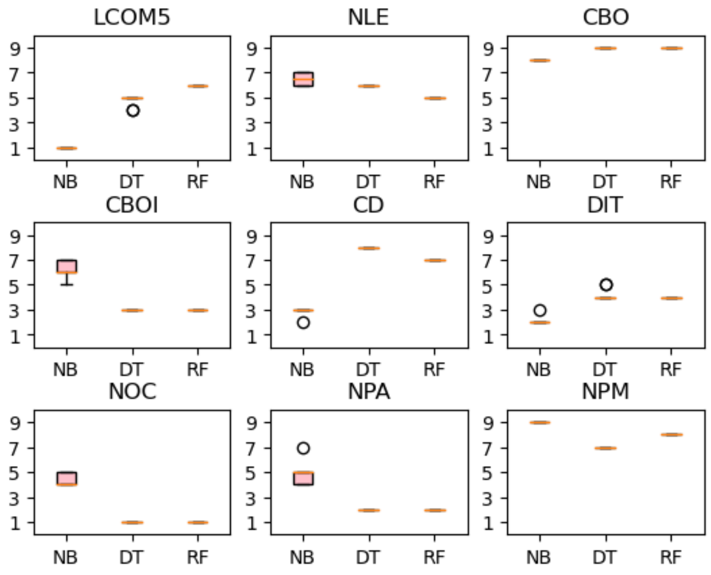

Figure A.5: Rankings of metrics by importance in project checkstyle

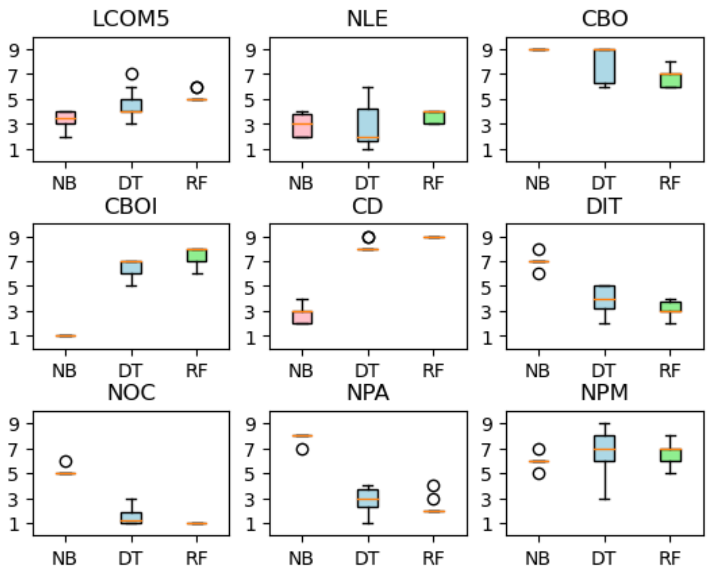

Figure A.6: Rankings of metrics by importance in project Discord4J

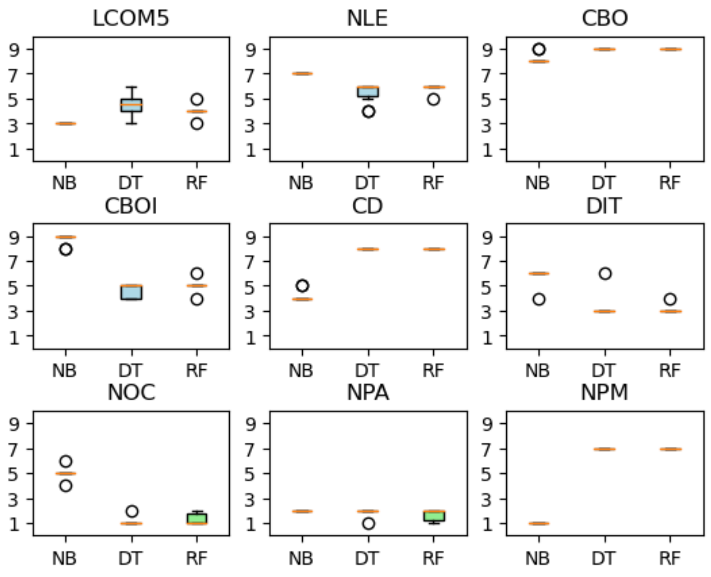

Figure A.7: Rankings of metrics by importance in project dubbo

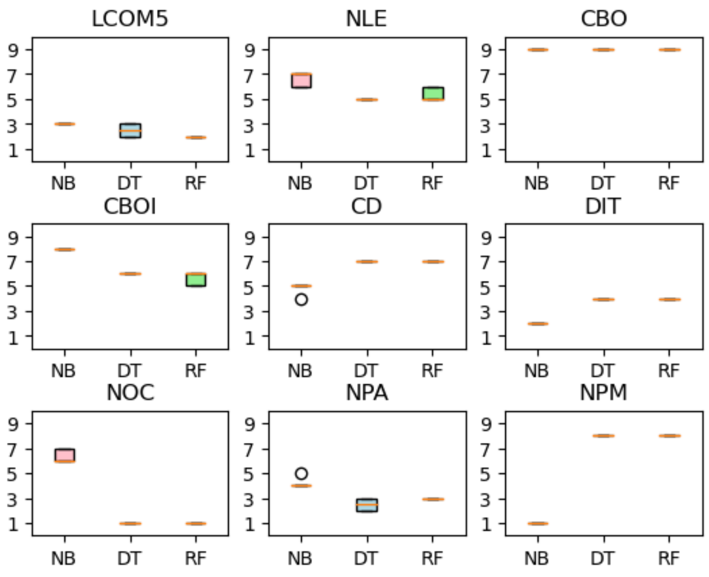

Figure A.8: Rankings of metrics by importance in project elasticsearch

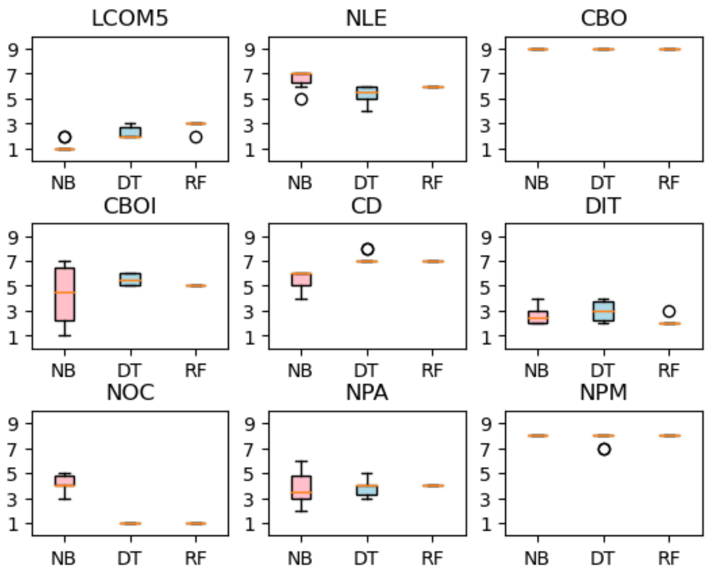

Figure A.9: Rankings of metrics by importance in project ExoPlayer

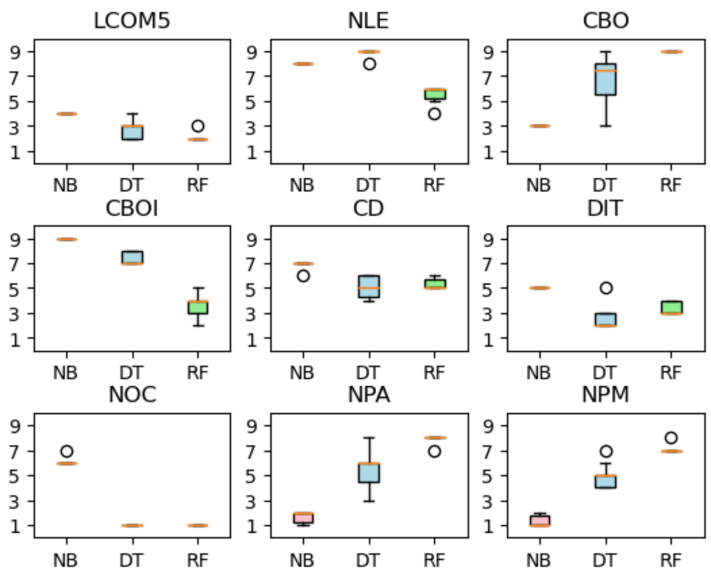

Figure A.10: Rankings of metrics by importance in project fastjson

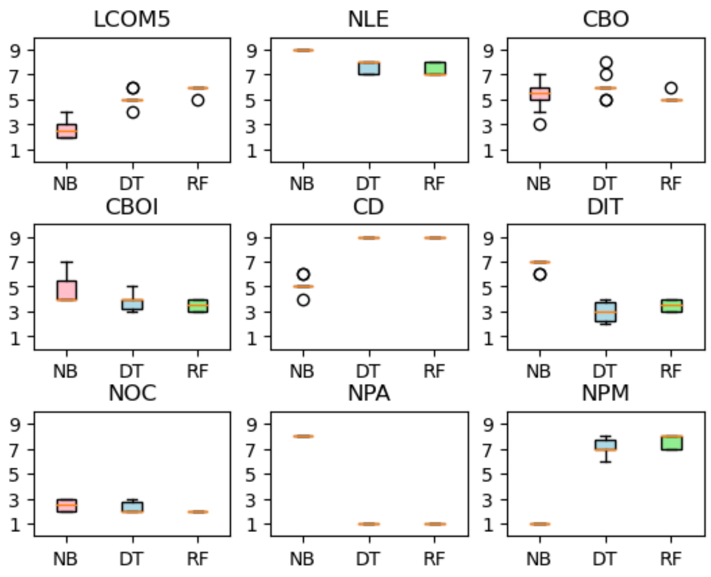

Figure A.11: Rankings of metrics by importance in project find-sec-bugs

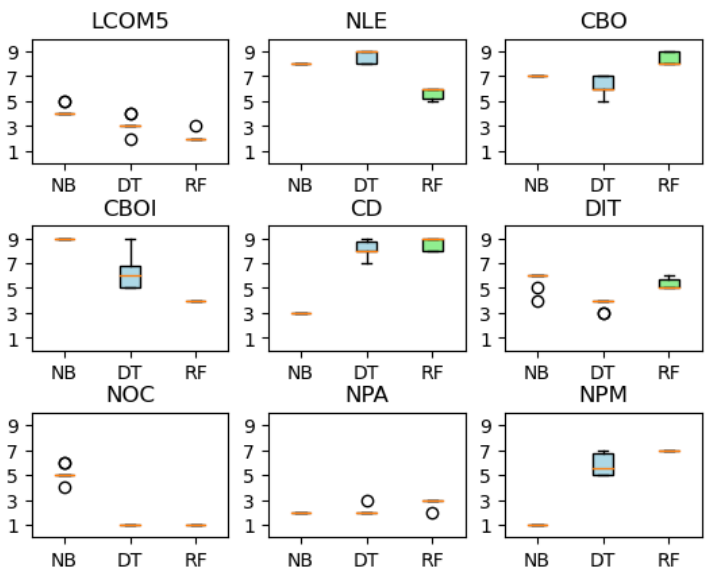

Figure A.12: Rankings of metrics by importance in project framework

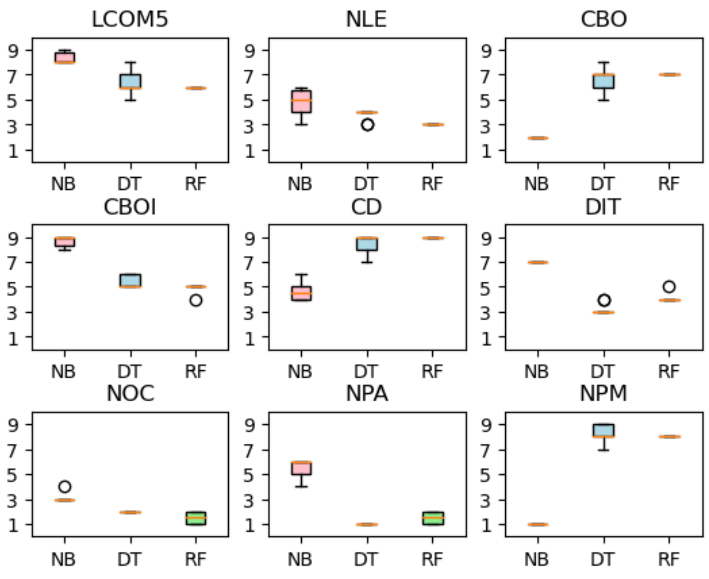

Figure A.13: Rankings of metrics by importance in project guava

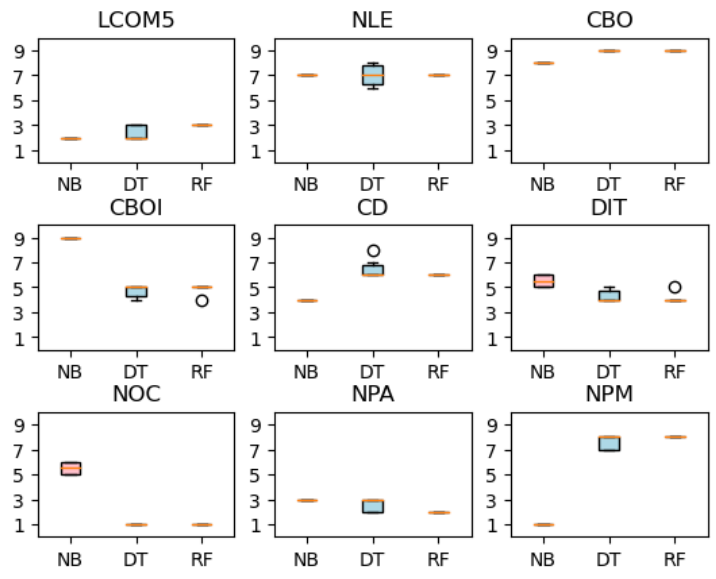

Figure A.14: Rankings of metrics by importance in project hazelcast

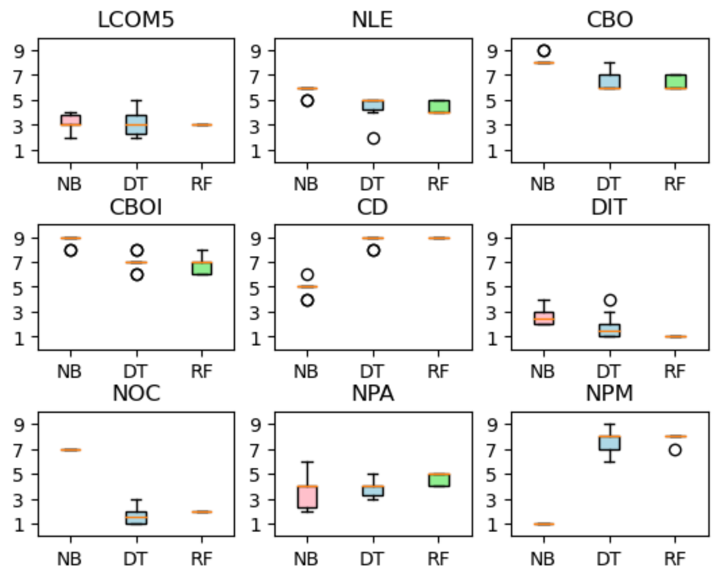

Figure A.15: Rankings of metrics by importance in project incubator-hugegraph

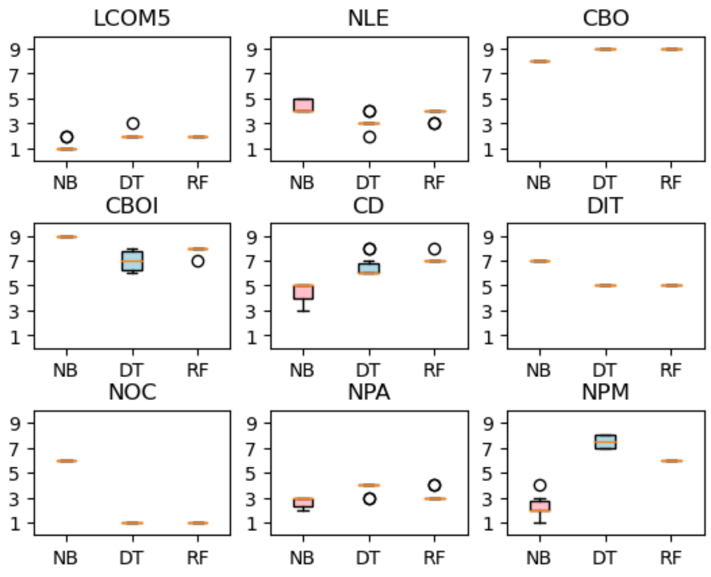

Figure A.16: Rankings of metrics by importance in project jetty.project

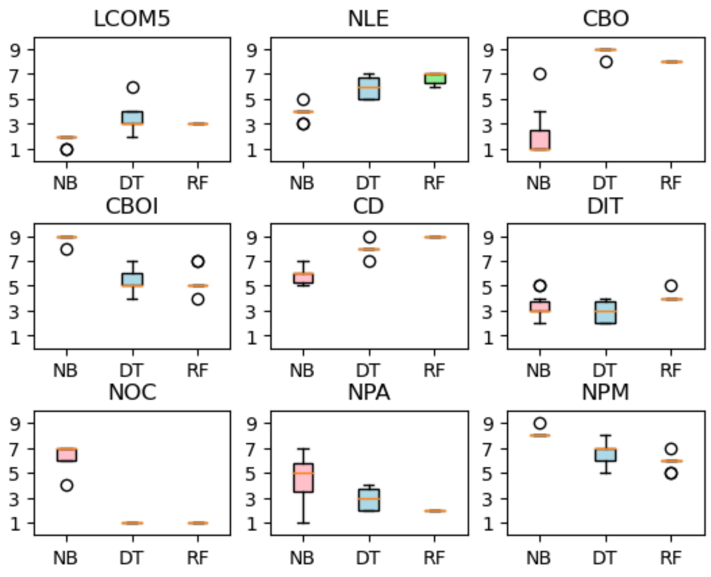

Figure A.17: Rankings of metrics by importance in project jmonkeyengine

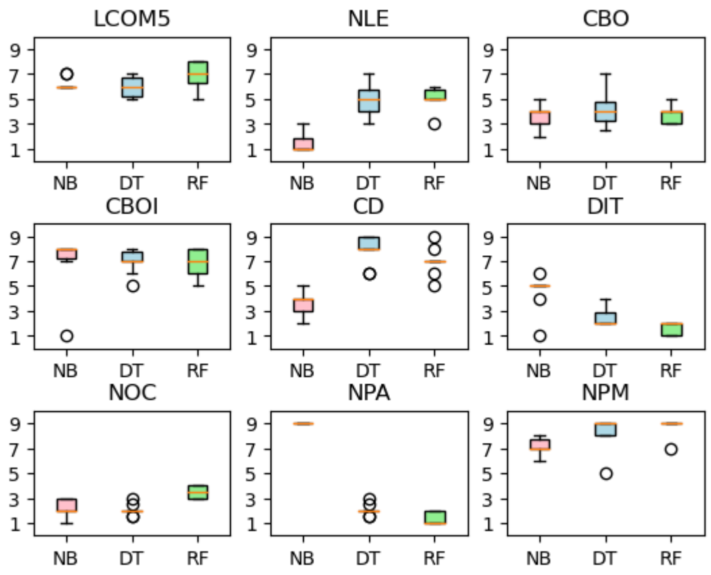

Figure A.18: Rankings of metrics by importance in project jsoup

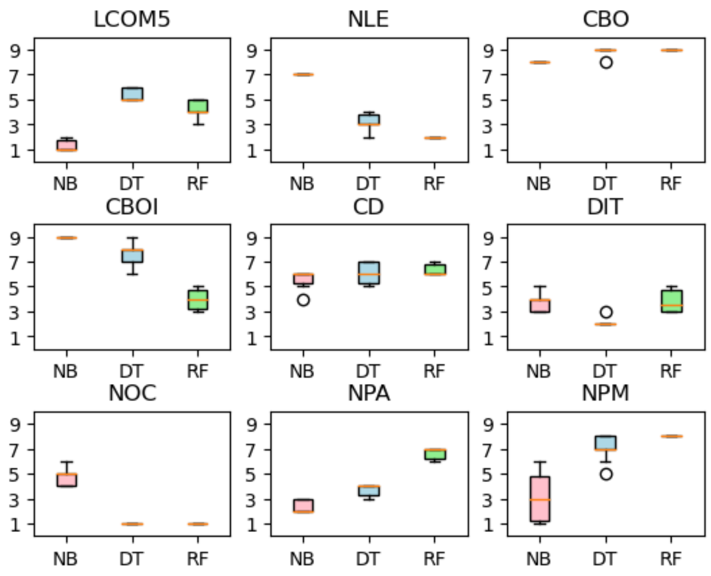

Figure A.19: Rankings of metrics by importance in project junit4

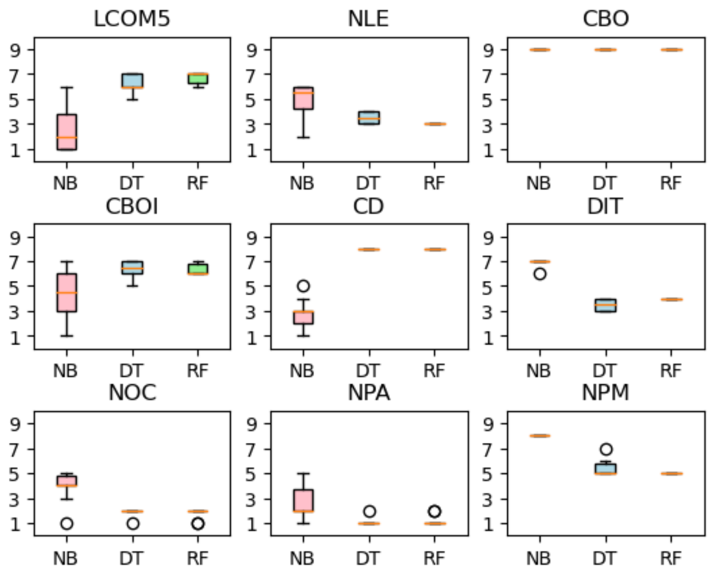

Figure A.20: Rankings of metrics by importance in project junit5

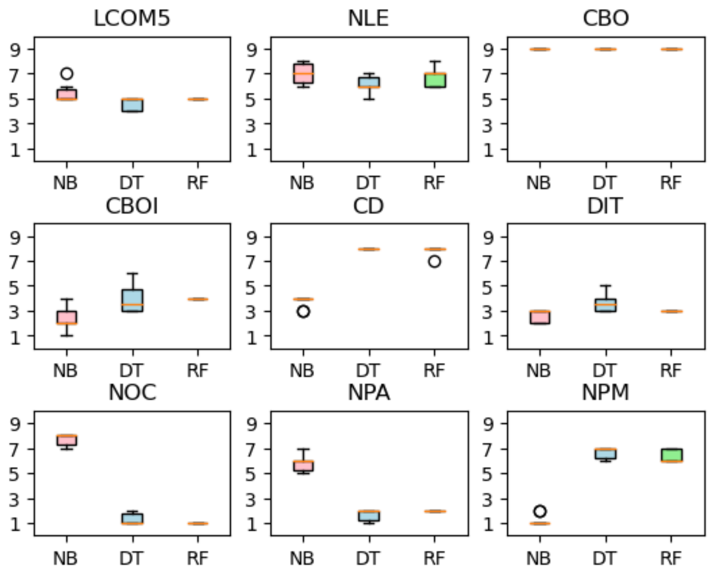

Figure A.21: Rankings of metrics by importance in project kura

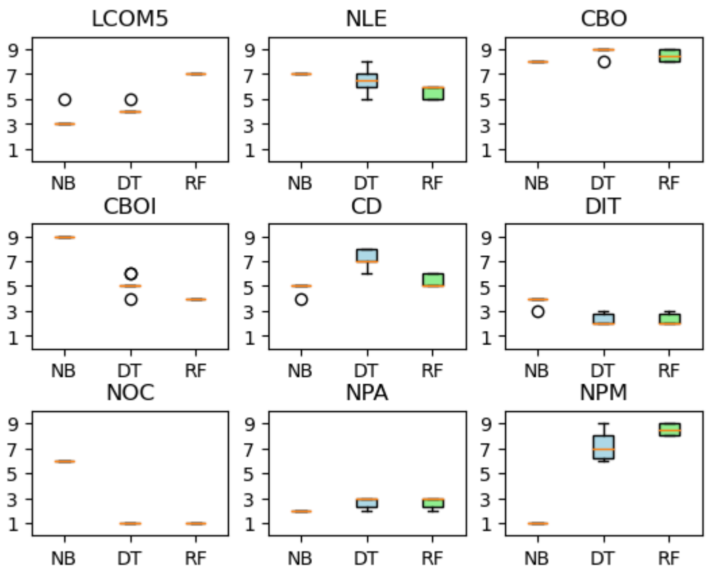

Figure A.22: Rankings of metrics by importance in project mapdb

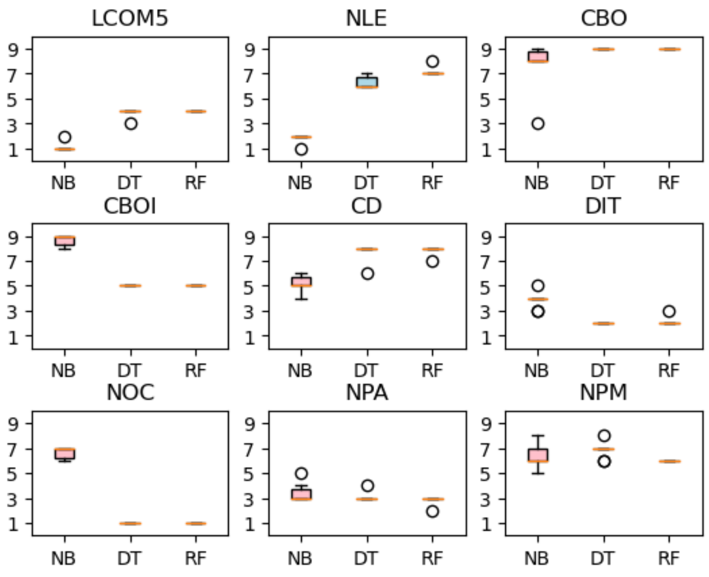

Figure A.23: Rankings of metrics by importance in project mcMMO

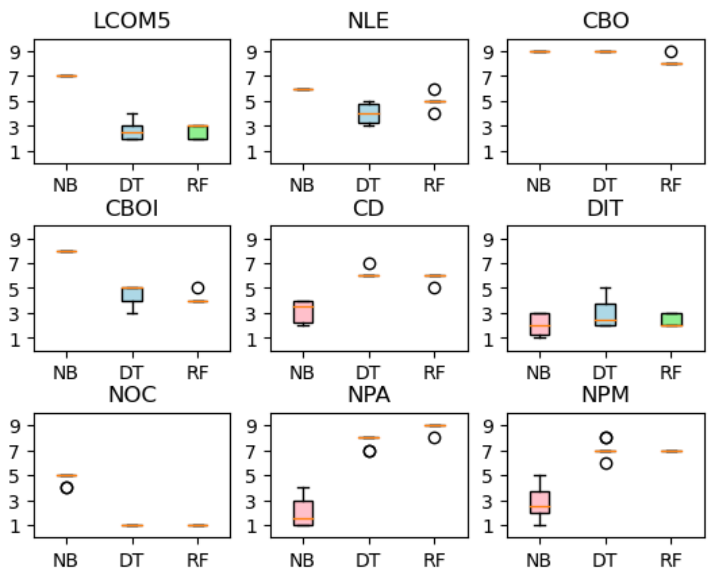

Figure A.24: Rankings of metrics by importance in project Mindustry

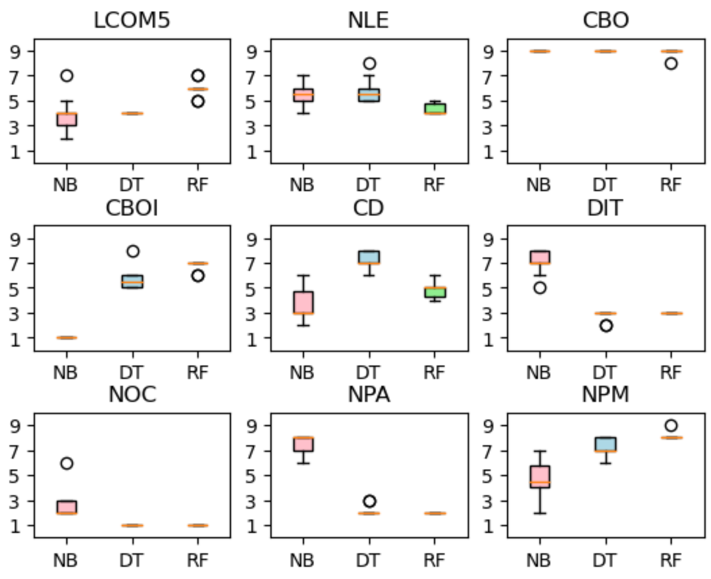

Figure A.25: Rankings of metrics by importance in project neo4j

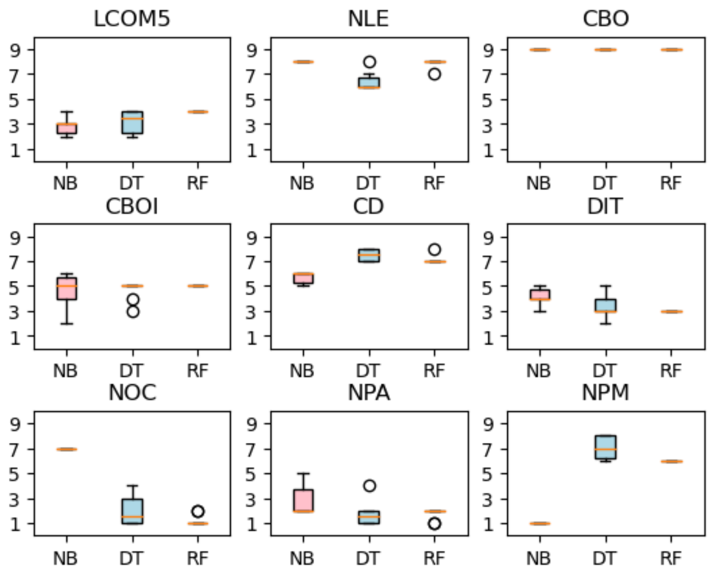

Figure A.26: Rankings of metrics by importance in project netty

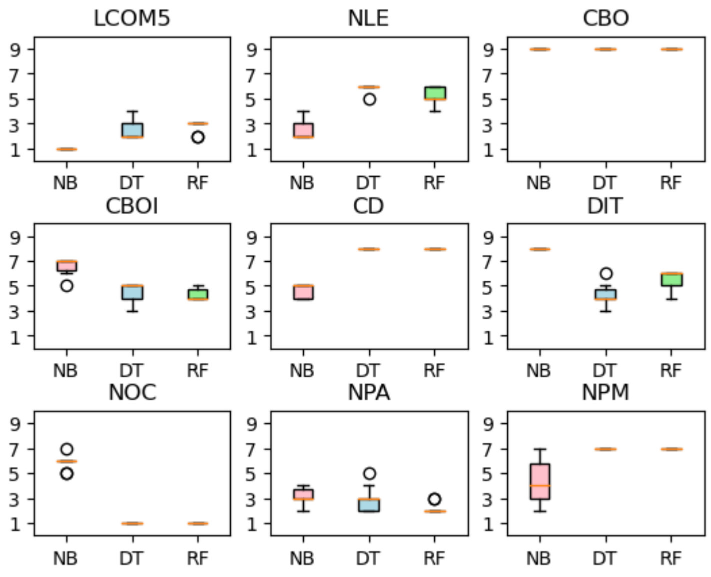

Figure A.27: Rankings of metrics by importance in project opengrok

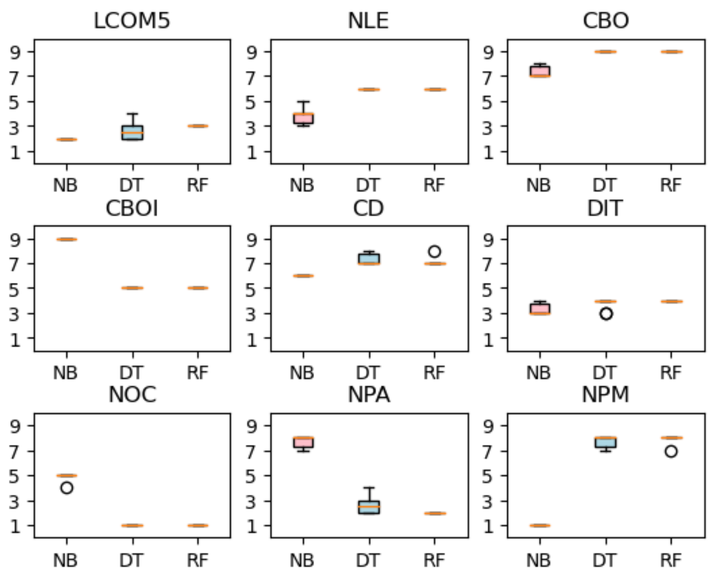

Figure A.28: Rankings of metrics by importance in project orientdb

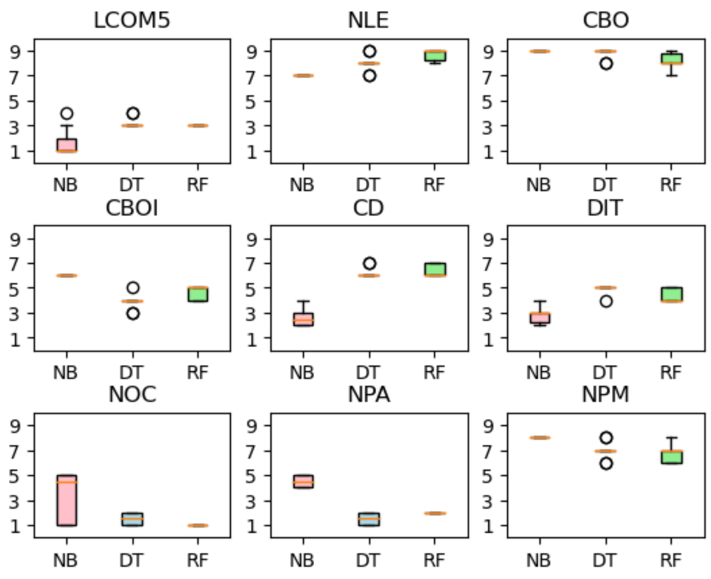

Figure A.29: Rankings of metrics by importance in project oryx

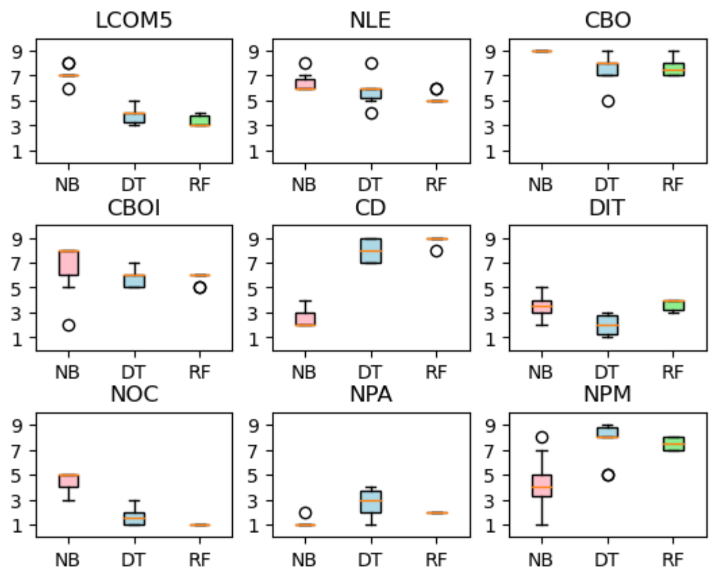

Figure A.30: Rankings of metrics by importance in project paho.mqtt.java

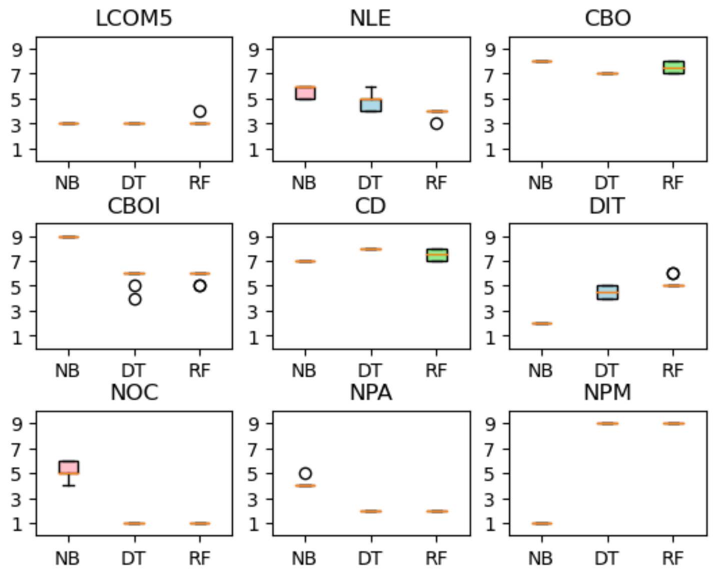

Figure A.31: Rankings of metrics by importance in project pmd

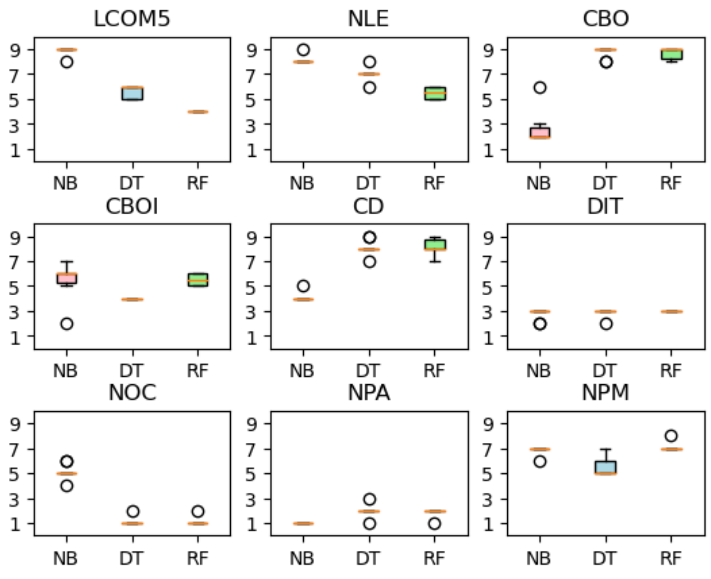

Figure A.32: Rankings of metrics by importance in project powermock

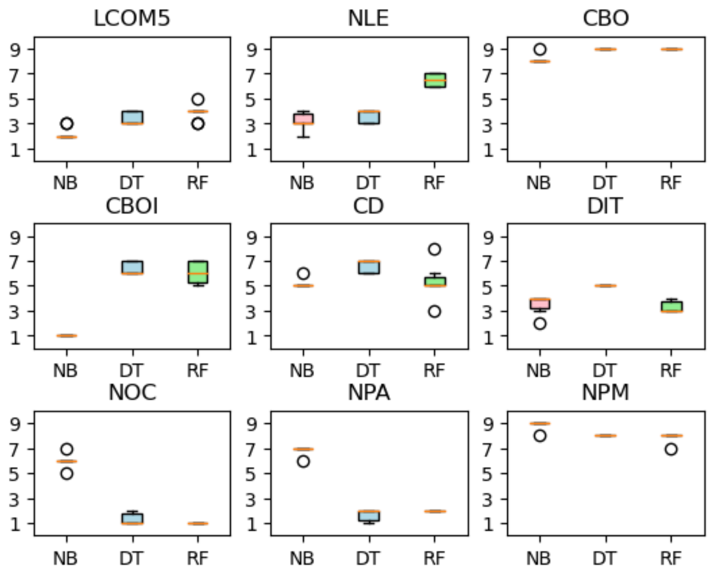

Figure A.33: Rankings of metrics by importance in project presto

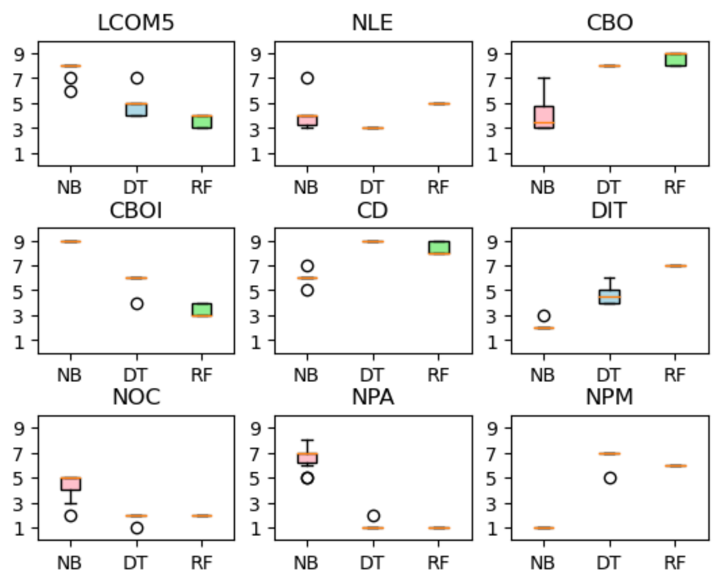

Figure A.34: Rankings of metrics by importance in project RxJava

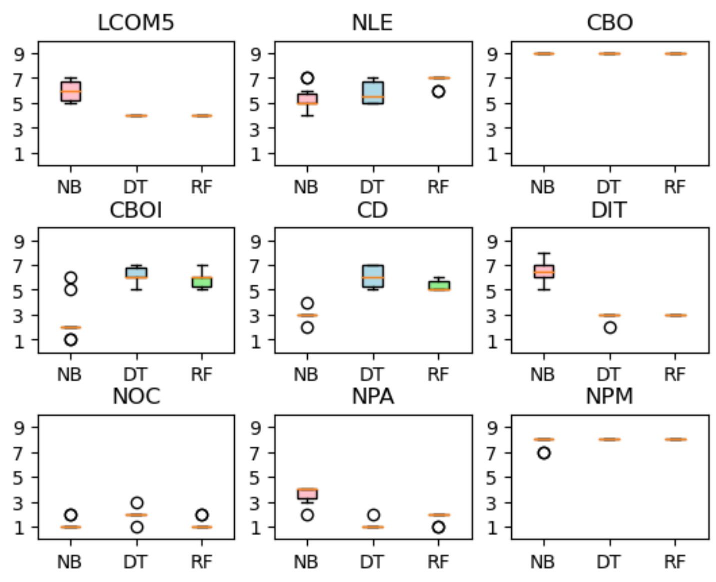

Figure A.35: Rankings of metrics by importance in project shardingsphere-elasticjob

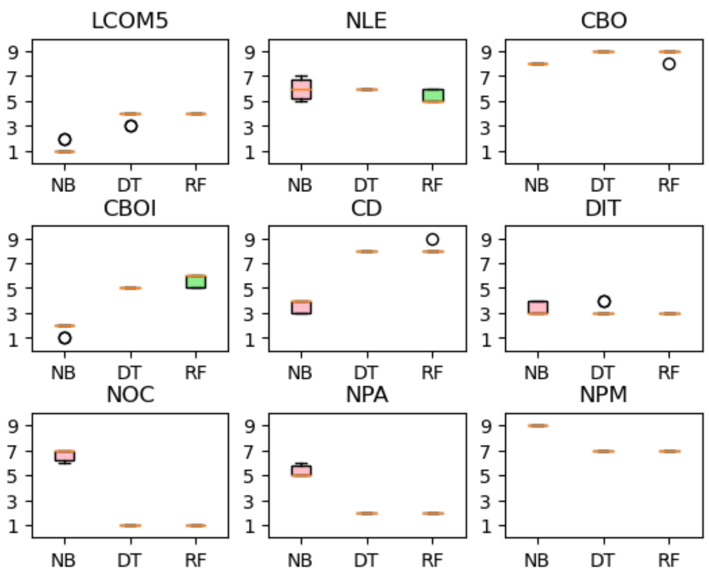

Figure A.36: Rankings of metrics by importance in project spring-boot

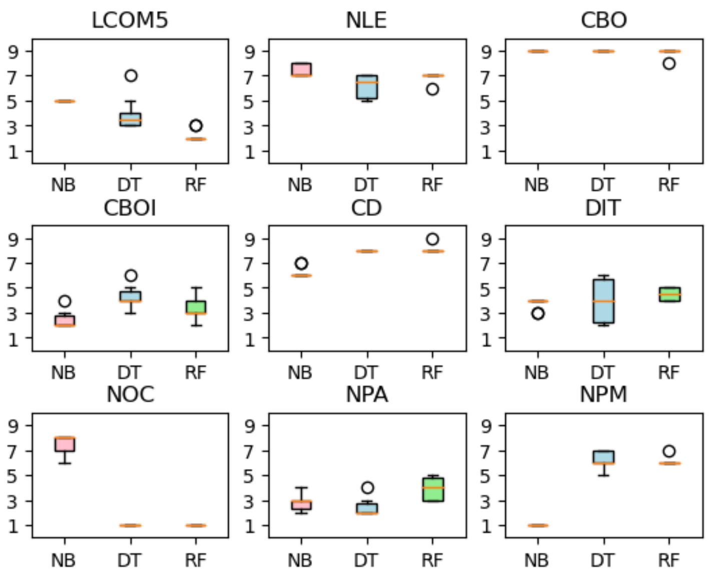

Figure A.37: Rankings of metrics by importance in project Terasology

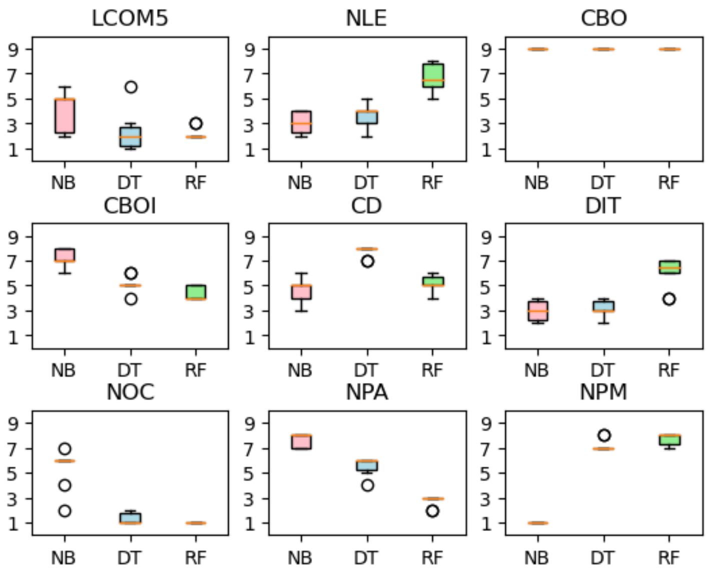

Figure A.38: Rankings of metrics by importance in project titan

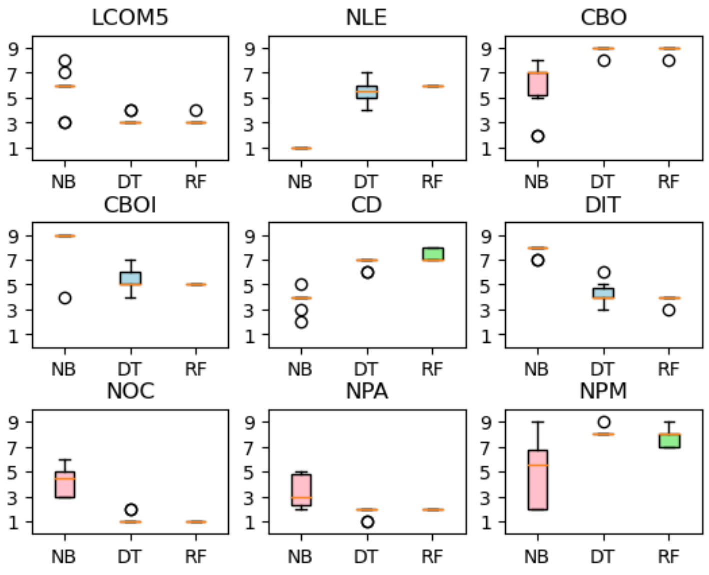

Figure A.39: Rankings of metrics by importance in project webcam-capture