BEIRUT: Repository Mining for Defect Prediction

Amir Elmishali*†, Bruno Sotto-Mayor*†, Inbal Roshanski*†, Amit Sultan*† and Meir Kalech*‡
**Software and Information Systems Engineering*
Ben-Gurion University of the Negev
Be'er-Sheva, Israel
†{amirel,machadob,inbalros,amitsul}@post.bgu.ac.il
‡kalech@bgu.ac.il

***Abstract*—Software Defect Prediction is an important activity used in the Testing Phase of the software development life cycle. Within the research of new defect prediction approaches and the selection of training sets for the classification task, different benchmarks have been analyzed in the literature. They provide several features and defective information over specific software archives. Therefore, they are commonly used in research to evaluate new approaches. However, the current benchmarks contain several limitations, such as lack of project variability, outdated benchmarks, single-version projects, a small number of projects and metrics, unavailable resources, poor usability, and non-extensible tools. Therefore, we introduce a novel tool *Bgu rEpository mIning foR bUg predicTion* (BEIRUT) for benchmark generation for defect prediction, composed of three main features: Given an open-source repository from GitHub, BEIRUT mines the software repository by (1) selecting the best k versions, based on the defective rate of each version, (2) generating training sets and a testing set for defect prediction, composed of a large number of metrics and defective information extracted from each of the selected versions and (3) creating defect prediction models from those extracted metrics. In the end, BEIRUT extracts a diversified catalog of 644 metrics and the defective information from each component of k versions, automatically selected based on the rate of defects in each version. They were collected from 512 different projects, starting from 2009. The tool is also supplemented with an easy-to-use web interface that provides a configurable selection of projects and metrics and an interface to manage the defect prediction tasks. Moreover, this tool is adapted to be extended with new projects and new extractors, introducing new metrics to the benchmark. The web service tool can be found at icc.ise.bgu.ac.il/njsw08.**

***Index Terms*—Defect Prediction, Software Quality Metrics, Repository Mining Tool, Open Source Metrics**

I. Introduction

Software systems take a critical role in all major areas of our society, in such a way that software defects in those systems lead to significant damages both to businesses and people's lives. As a result, considerable research has been proposed into novel predictive models and tools that assist software engineers and testers in narrowing down which components are more likely to contain defects in the software codebase. Therefore, leaning toward more precise testing, which includes tasks such as selecting or prioritizing test cases [1,2].

Defect Prediction approaches aim at predicting which software components are more likely to be defective based on particular metrics from the software repository. Common techniques apply classification algorithms on a training set, usually composed of multiple metrics extracted from software archives [3]. These metrics range from code metrics that measure properties from the source code to process metrics that reflect the changes over time on the software development process [4]. Overall, the metrics used for the classification have a significant weight on the performance of defect prediction. In addition, to being composed of metrics, each of the training set instances includes a label defining whether the respective component is defective [3].

One of the most important problems when applying defect prediction is the selection of the software defect benchmark. From the early stages of research, there has been an evolution in the benchmarks presented in the literature. It started with proprietary benchmarks, making it impossible to compare the results of prediction methods since they could not be obtained [5]. Hence, it led to the release of public benchmarks and, in particular, the aggregation of them into repositories that could be used as baselines for future work, for instance, the PROMISE benchmark [6]. From there, other benchmarks were introduced aiming to explore other metrics and different projects, providing different levels of interaction and attempting to go beyond limitations of previous available benchmarks [7]–[16].

Although there is an extensive collection of benchmarks commonly used for defect prediction, they contain certain limitations. For instance, some benchmarks provide a low variety of projects. For example, the NASA benchmark only includes features for NASA spacecraft specific software. Then, some benchmarks have outdated data from old projects published in a range between 5 to 15 years ago. Some projects include only a single version, thus not providing features that show an accurate representation of the software's evolution. Most benchmarks are limited to a small number of projects, thus having on average 7 projects, ranging from 1 to 20. Moreover, each project contains, on average, 23 versions, ranging from 1 to 96. Moreover, previous benchmarks contained an average of 75 features from software projects, ranging from 12 to 485 features. There are published benchmarks with missing resources; for instance, some papers are missing either the data itself or the source code to which they acquired the features. Furthermore, most benchmarks have poor usability since they are limited in regards to interacting with projects and their components and do not provide a high degree of flexibility to

This research has been funded by the Cyber Security Research Center at Ben-Gurion University of the Negev.

acquire specific features. Last, the benchmarks do not provide mechanisms to extend their data with new projects or features.

To overcome these limitations, we propose a novel tool – BEIRUT– to produce benchmarks from open source projects for defect prediction. It is composed of three main features. Given an open-source repository, BEIRUT selects the best k versions based on each version's rate of defects. Then, BEIRUT extracts up to 644 code metrics from each of the selected versions and creates the training and testing sets for the defect prediction. Moreover, it collects the defective information for each of the components in the versions. Lastly, BEIRUT includes a defect prediction module that allows creating classification models to predict defects using the extracted metrics and the defective labels.

At the time of writing, BEIRUT is capable of generating benchmarks for 512 open source projects, starting from 2009. We selected the projects from the Apache software foundation since they have a high reputation and include a considerable number of programs written in Java and developed using Jira. Moreover, we selected recently active projects, with at least three versions containing at least 5% of defects. The tool uses the version control functionality from Git to select the versions and access its source code. Then, it uses the issue tracker information from Jira Issues to obtain the defective labels for each component. Then, to extract the metrics, we created an integration environment. When given a set of features to extract, BEIRUT uses the respective publicly available metric extraction tools to collect them.

To interact with BEIRUT, we include a web interface that allows the user to configure the projects to extract, which versions and how many versions to select, either manually or automatically, and which metrics to extract. It also provides a means to apply defect prediction models based on the selected metrics.

The rest of this paper is organized as follows. In Section II, we introduce the background and related work on the different metrics that BEIRUT extracts, including the respective metrics extraction tools, and we review the existing literature benchmarks. In Section III, we describe the paradigm and design decisions behind BEIRUT, including the three features composing the tool. In Section IV, we present the tool demo, where we outline the process to use the different features of BEIRUT. In Section V, we discuss and evaluate BEIRUT against previous benchmarks and present its limitations. Last, in Section VI we present our conclusions.

II. Background and Related Work

In this section, we introduce and describe the metrics extracted by BEIRUT, including the publicly available metrics extraction tools that BEIRUT runs to extract them. Moreover, we describe and review the benchmarks available in the literature for defect prediction.

A. Metrics

Extracting metrics as features for defect prediction is one of the most important components for building powerful prediction models. Within the literature, defect prediction metrics have been classified into three categories, regarding the type of structures that are being evaluated [4]. The first is Traditional metrics, and they refer to metrics that measure the size and complexity of the source code. The second is Object-Oriented metrics, and they measure the coupling, cohesion, and inheritance attributes of source-code at the class-level. The third is Process metrics, which measure attributes regarding the development process, thus collecting historical information from the version control systems and issue trackers.

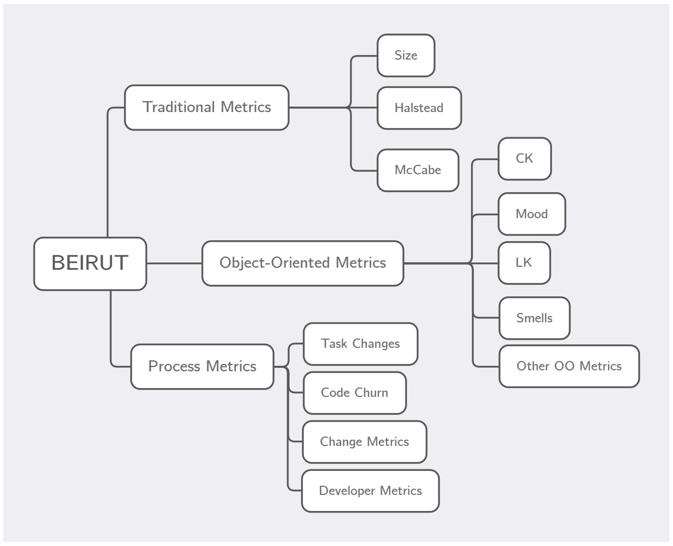


Fig. 1. Diagram of the metrics extracted by BEIRUT.

BEIRUT extracts several metrics from each category. In Figure 1, we display the different metrics extracted by BEIRUT, including the respective categories.

Traditional code metrics are metrics extracted directly from the source code that measure how large and complex the software is at an imperative level, i.e., statements and lines of code. BEIRUT considers three types of metrics within the Traditional metrics: The *Size metrics* measure size-related information from the code, for example, number of lines of code, number of statements, number of lines with comments, and so on [17]. The *Halstead metrics* measure several properties of the software, and the relations between them [18]. It follows a similar classification principle as measuring the properties of matter (e.g., volume and mass) and the relationship between them. Therefore, it includes, for example, the volume of the code, which is the length of the program, plus the logarithm of the vocabulary of the program. The *McCabe metrics* measure the program's complexity by counting the number of linearly independent paths through a program's source code [19].

Object-Oriented metrics measure object-oriented properties from the source code, such as coupling, cohesion, and inheritance. Among these metrics, BEIRUT extracts 6 known types of metrics presented in the literature. The *Chidamber and Kemerer metrics suite* is a set of measures that evaluate object-oriented principles [20,21]. They include, for example, the number of non-inherited classes that are coupled to a particular class (CBO), the depth for a class in the inheritance tree (DIT), or the number of direct subclasses (children) of a class (NOC). The *Mood metrics* were proposed by Brito and Abreu. They are extra measures that evaluate the use of main abstractions in object-oriented programming such as inheritance, encapsulation, and information hiding, or polymorphism [22]. For instance, the number of private, protected, and public attributes; the number of inherited and polymorphic methods; or the number of coupled classes. The *Lorenz and Kidd (LK) metrics* focus on method size and internals, and class inheritance and internals [23]. For example, for method size and internals, they measure the number of messages sent, the lines of code, and the method complexity (extension of McCabe's complexity for OOP), and strings of message sends. The *Code Smells* are patterns in the source code that possibly indicate a deeper issue in the system [24]–[26]. They are defined by specific structures that violate certain principles of good programming. Generally, they are composed of a threshold applied to a specific combination of metrics. For example, a Long Method is a smell that is detected when the number of lines of code in a method is superior to a threshold that makes it too long. *Other Object-Oriented metrics* are metrics that do not fit a specific suite of metrics published in the literature but are available for extraction and can provide more information about the source code base on object-oriented principles. For example, they include the length of anonymous inner classes and the coupling of class data abstraction.

Process metrics are extracted from the combination of the source code and repository. They measure the characteristics and complexities of the development process [27]. They target the historical information from version control systems and issue trackers. In particular, they measure the changes over time and quantify several characteristics from the source code, for example, the number of code changes and the number of developers. BEIRUT extracts 4 types of process metrics. The *Task Changes metrics* measures the changes from the issue tracking system to complete a task, i.e., a ticket in the issue tracking system (e.g., fix, new feature, improvement, and so on). One example is the average number of lines of code that were added to implement a new feature. The *Code Churn metrics* measure the changes in the code made in a repository over a period of time [28]–[30]. For instance, one example is the number of lines of code and files that were changed and deleted. The *Change metrics* refer to measures related to the changes, such as the number of commits, refactoring, bug fixes, authors, and so on [31]. The *Developer based metrics* measure information about each developer regarding their involvement in the code [32]. For example, they include the number of developers that changed a specific version of the file.

B. Benchmarks

An essential component of the defect prediction process is the benchmark used for the training of the classification models.

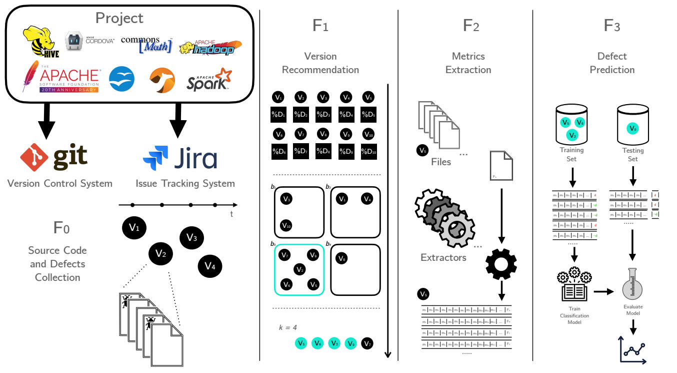


Fig. 2. Overview of BEIRUT's architecture.

Each benchmark comprises the software archive where it was extracted, the features collected from each component of that software, and a label that indicates whether a component has a bug or not. In the early stages of defect prediction research, academic researchers and companies used non-public Benchmarks to produce the defect prediction models. However, due to the inability to re-use those benchmarks, both for validating the proposed models and comparing new approaches, in 1990 it was published a public repository called University of California Irvine (UCI) Machine Learning Repository [3]. As a consequence of its success, in 2005, the PROMISE benchmark was published, which aggregates several publicly available benchmarks into a single public repository [6]. As of this paper's writing, the PROMISE repository contains 71 versions of 38 different open-source projects. However, most of the project versions have a small number of components, or they include only a single version, thus making them infeasible for defect prediction. Therefore, we only considered the projects with over 100 components and are extended towards more than a single version, resulting in 35 versions over 10 projects. Furthermore, PROMISE contains a total of 20 metrics, in particular traditional and object-oriented metrics. Despite its recurrent use in defect prediction, it contains several limitations: not providing direct access to the source code, a small number of evaluated metrics, and being old software with most projects containing metrics for a single version. BEIRUT covers all of these limitations since it extracts 512 relatively recent projects from a broad range of domains. It can extract over 644 metrics from components over several versions of each project.

In addition to the PROMISE Repository, independent benchmarks have been published on specific studies for empirical studies on defect prediction.

In 2007, NASA MDP benchmark and ECLIPSE1 defects benchmark were published to study defect prediction. For NASA MDP, Menzies et al. use 8 out of 13 benchmarks collected from the NASA Metrics Data Program (MDP) website [33]. They considered 38 metrics, 3 McCabe metrics, 12 Halstead metrics, and 23 Traditional Size metrics, where 6 are

LOC metrics and 17 are miscellaneous metrics, including code properties counts (e.g., condition count and decision count), call pairs, densities, complexities and so on. Note that the PROMISE repository includes 13 out of the 14 benchmarks provided by the NASA MDP benchmark. However, they were found to contain essential differences between them [14]. Moreover, Shepperd et al. analyzed the MDP benchmarks and verified that it contains 20 to 40 code metrics, including size, readability, complexity attributes, etc. ECLIPSE1 is a benchmark collected from Zimmermann et al. [16]. They extracted metrics from three releases of Eclipse 72 static code metrics: 31 on the file-granularity and 41 on the package-granularity. Within the 31 code metrics of the file-granularity, they considered 12 unique metrics: 1 CK(FOUT), 1 McCabe (VG), 1 Size (LOC), and 9 Other Object-oriented Metrics; they considered the maximum, average, and total count for each of 9 metrics and only the other 4 metrics' value. They considered an extra metric on the package level: the number of files, which we do not consider in our study since BEIRUT's scope focuses on the file-level. Their goal was to study the correlation between complexity metrics and defects in components.

SOFTLAB benchmark was published in 2009, and it was collected from 3 projects from an embedded controller software for a Turkish white-goods manufacturer [34]. Their paper studies the relevancy of cross-company data to build localized defect prediction models using static code features. Therefore, they collect 29 code features in the SOFTLAB benchmark, specifically 19 Size metrics, 2 McCabe metrics, and 8 Halstead metrics.

In 2010, Jureczko and Madeyski released a benchmark containing 20 code metrics, extracted from 92 versions of 38 software projects [10]. In particular, they extracted 6 CK metrics, 1 McCabe's (CC) metric, 5 Mood metrics, 1 Size metric (LOC) and 6 Other OO metrics [35,36] The study's goal was to group software projects based on similar characteristics within the context of defect prediction. They considered 48 versions of 15 Apache open-source projects, 27 versions of 6 proprietary projects within the domain of insurance companies, and 17 academic projects performed by students, each with a single release. Moreover, in 2010, D'Ambros et al. released the AEEEM benchmark [8]. The study applies an extensive comparison of defect prediction approaches. Thus they introduce a benchmark in the form of public data sets comprised of multiple software projects. It is composed of five projects, and for each, they extract 61 metrics: 17 Object-oriented metrics, 5 Code Delta, 5 Change metrics (i.p., Entropy of Change), 17 Entropy of Source Code, and 17 Code Churn metrics.

Kim et al. published the ECLIPSE2 in 2011 to study and propose new approaches to deal with noise in defect prediction [13]. ECLIPSE2 benchmark is composed of 13 metrics, extracted from two projects of Eclipse 3.4. – SWT and Debug. In particular, they extracted 3 traditional metrics, 6 object-oriented metrics, and 4 process metrics. Furthermore, ReLink was published in 2011 by Wu et al. [15]. Their goal is to improve the quality of the defective information by proposing an algorithm to recover the missing links between defects and changes. They then apply defect prediction using the generated benchmark to evaluate their novel linking approach compared with previous approaches from the literature. The respective benchmark consists of 41 to 60 complexity metrics and other object-oriented metrics extracted from 3 open-source projects – ZXing, OpenIntents, and Apache.

NETGENE benchmark was proposed to study the predictive power of change genealogies, related to the history of a file, for defect prediction [9]. The benchmark contains a total of 485 metrics extracted from 4 open-source projects. These metrics include complexity metrics, network metrics, and change genealogy metrics. As such, their current benchmark includes 38 complexity metrics: 2 McCabe metrics, 5 CC metrics, and 31 Object-Oriented metrics; 51 code dependency network metrics, considering the incoming, outgoing, and undirected dependencies; and 414 change genealogy metrics. Furthermore, Altinger et al. published the AEV benchmark in 2015 to predict defects in the domain of automotive projects [7]. The benchmark includes 29 code metrics extracted from 3 software projects developed by Audi Electronics Venture GmbH (AEV): 12 Size metrics, 7 Halstead metrics, 2 McCabe metrics, and 8 change metrics.

Although the benchmarks we reviewed have been proved to be useful for defect prediction, they show major limitations, which do not allow further and more in-depth exploration of defect prediction on a large scale. Therefore, we introduce BEIRUT, which complements several features that are not addressed in the current benchmarks. Table I summarizes the size and properties describing the range of projects, versions, files, features, and age addressed in BEIRUT and the other benchmarks from the literature. Our tool outperforms the other benchmarks in terms of the number of projects and features. In Table II, we further explore the types of features included in each benchmark, as we count the number of metrics extracted for each benchmark, by category of metrics, as displayed in Figure 1. Moreover, the benchmarks are outdated, with the most recent being at least 5 years old. Consequently, several benchmarks are not available to be downloaded. In Table III we summarize the outlined limitations.

III. Paradigm Overview

In this section, we describe BEIRUT's paradigm and overall architecture. It is composed of three features: a versions recommendation system, which is capable of automatically select k versions based on specific defect criterion; an extensible metrics' extraction module that collects a set of metrics as desired by the user, including the defective information of each software component; and a defect prediction generation feature that automatically builds classification models from the generated metrics to predict defective components. In Figure 2, we display an overview of BEIRUT's architecture, including the main features.

In the diagram, there is an extra initial step, which represents the extraction of the source code from the version control

TABLE I
SIZE AND PROPERTIES OF EACH BENCHMARK PROPOSED IN THE LITERATURE AND BEIRUT.

	# Projects †	# Versions ‡	# Files ‡	# Features	Age
BEIRUT	**512**	**49**	**1258**	**644**	**2021**
AEV [7]	3	93	40	29	2015
NETGENE [9]	4	NA	4436	485	2013
JIT [12]	6	NA	NA	14	2013
PROMISE [6]	20	4	1115	20	2012
ECLIPSE2 [13]	2	1	659	17	2011
RELINK [15]	3	1	217	60	2011
AEEEM [8]	5	96	1177	61	2010
JUREKZO [10]	15	4	412	20	2010
NASA/MDP [14,33]	13	1	6217	38	2007
ECLIPSE1 [16]	1	3	7085	12	2007
SOFTLAB [34]	3	1	69	29	2005

† Number of publicly available projects.
‡ Average number of versions and files per project.

system (*VCS*) and the matching of the source code to the issues in the issue tracker (*IT*). Although it is not a main feature of the tool, it is essential for each feature. In a simplified view, a project P is composed of w versions $V_P = \{v_1, v_2, ..., v_w\}$, ordered chronologically from oldest to newest. Each version contains r files $v_i = \{f_1, f_2, ..., f_r\}$. The algorithm automatically extracts all the files in each version using the VCS. It, then, assigns whether each file is defective by mapping each commit to a particular issue in the IT. In the end, it creates for each version two lists: one containing the files that are defective and the other the files that are not defective.

$$D(v_i) = \begin{cases} [f_{1d}, f_{2d}, ..., f_{nd}], & \text{defective files} \\ [f_{1\bar{d}}, f_{2\bar{d}}, ..., f_{m\bar{d}}], & \text{non-defective files} \end{cases} \tag{1}$$

The methodology for the collection of the defects is a variant to the approach implemented in the *SZZ* algorithm [37] which accounts for its vulnerabilities [38]. BEIRUT starts by extracting closed issue reports from Jira for the project under extraction. In other words, it only considers issues of defects that are closed and were fixed. Then, BEIRUT links each collected issue to the respective bug-fixing commit on Git. First, for every issue, it clusters all possible commits based on the version and on the date it was submitted. It, then, uses the id from the issue (e.g., MATH-432 on the commons-math project) to look for the correspondent bug-fixing commit by applying pattern matching and searching for a corresponding issue id on all of the commits titles and commit messages. After matching each issue to the respective bug fixing commit, BEIRUT labels the files as defective if they were changed in that particular fixing commit. Consequently, for each version, the files labeled as defective are the ones that were previously changed at least once in a fixing commit.

Considering the problems reported by [38] regarding SZZ approach, our approach for defect labeling only consists of the linkage between issues and bug fixing commits; thus excludes the identification of bug-inducing commits. Therefore, from the issues BEIRUT needs to address, we excluded those targeting the task of identifying the bug-inducing commits. Moreover, SZZ was originally created for Bugzilla. Each issue id in this tracker corresponds to a number, which by attempting to match to the commit messages, leads to higher probability of incorrectly matching a number associated to the commit description and not the issue. Henceforth, in our approach, we matched the issue identifier of the JIRA tracker, which follows the format <PROJECT>–<NUMBER>. After identifying the bug-fixing commit, another reported issue was the selection of the files involved in the change that were actually responsible for the defect. We followed the solution proposed by the authors ([38]). We selected only Java files and excluded files that were test cases, refactorings, and where changes targeted comments alone. Furthermore, [39] found that within the issues that are reported as bugs, some of them are actually requests for new features, bad documentations, or refactorings. This originates from bad issue classification on the part of the developers and maintainers. Therefore, [38] proposed a manual validation solution; however it does not fit our requirements since we are proposing an automated tool, and to the extend of our knowledge there is no automated solution up to this point. Therefore, we do not address this issue and consider it as noise in the defect prediction.

A. Versions Recommendation

After gathering information on each version's defects, the first main feature in BEIRUT's paradigm is the selection of k versions from a given project P, which will be used for the training set of the prediction model. It is challenging to decide which versions are the best to build a good prediction model. We believe that a good training set should contain versions with a similar percentage of bugs. Since the number of files, commits, and relevant issues differ between versions, these differences may lead to poor learning due to the class imbalance. Therefore, we choose the versions based on their defect percentage (i.e., the number of defect files per number of files). Our rationale is that the chosen versions in the training set should represent a consistent development process. Therefore, we want to avoid releases with extremely high (such as hotfixes) and extremely low (such as releases focused on refactoring or documentation) rates of defects. From the authors' experience, this range was set to be between $10 - 30\%$; however, this assumption can be modified by the user. Moreover, from the view of machine learning, we want to collect quality samples that reflect the common distribution of bugs in versions during the development process because the use case of defect prediction does not apply for contingency cases like hotfixes. To this end, the algorithm calculates the defective rate (D_r) for each version (v_i), by calculating the percentage of files $F_d = \{f_{1d}, f_{2d}, ..., f_{nd}\}$ that are defective over all files $F = \{f_1, f_2, ..., f_{n+m}\}$:

$$\%D_r(v_i) = \frac{\#F_d}{\#F} \times 100 \tag{2}$$

The versions recommendation system algorithm receives as input the defective rates over each of the versions of a given

TABLE II
NUMBER OF METRICS EXTRACTED FROM OUR TOOL AND EACH BENCHMARK FROM THE LITERATURE, WITHIN THE CATEGORIES CONSIDERED BY BEIRUT.

	Traditional Metrics			Object-oriented Metrics					Process Metrics					Total
	Size	Halstead	McCabe	CK	Mood	LK	Smells	Other	Task Changes	Code Churn	Change Metrics	Developer Metrics	Other	
BEIRUT	**58**	**9**	**3**	**6**	**18**	**20**	**47**	**37**	**234**	**0**	**17**	**195**	**0**	**644**
AEV	12	7	2	-	-	-	-	-	-	-	8	-	-	29
NETGENE	-	-	2	5	-	-	-	13	-	-	-	-	465	485
JIT	-	-	-	-	-	-	-	-	-	1	10	3	-	14
PROMISE	1	-	2	6	-	-	-	11	-	-	-	-	-	20
ECLIPSE2	1	-	2	6	-	-	-	-	-	-	3	1	-	13
RELINK	-	-	-	6	-	-	-	54	-	-	-	-	-	60
AEEEM	1	-	-	6	-	-	-	10	-	17	5	-	22	61
JUREKZO	1	-	2	6	5	-	-	6	-	-	-	-	-	20
NASA/MDP	23	12	3	-	-	-	-	-	-	-	-	-	-	38
ECLIPSE1	1	-	1	1	-	-	-	9	-	-	-	-	-	12
SOFTLAB	19	2	8	-	-	-	-	-	-	-	-	-	-	29

TABLE III
LIMITATIONS OF THE BENCHMARKS FOUND IN THE LITERATURE.

	High #Projects (> 50)	High #Features (> 50)	New (> 10y)	Available	Extensible
BEIRUT	✓	✓	✓	✓	✓
AEV	-	-	✓	✓	-
NETGENE	-	✓	✓	✓	-
JIT	-	-	✓	-	-
PROMISE	-	-	✓	-	-
ECLIPSE2	-	-	✓	-	-
RELINK	-	✓	✓	-	-
AEEEM	-	✓	-	✓	-
JUREKZO	-	-	-	✓	-
NASA/MDP	-	-	-	✓	-
ECLIPSE1	-	-	-	✓	-
SOFTLAB	-	-	-	-	-

project, $\%D_r(V_P)$, and a parameter k defining the number of versions to be selected. Recall that since the difference between the versions calculates the process metrics, we need at least 3 versions to be selected $(k \geq 3)$ to create a training and a testing set.

We start by organizing the files into bins with ranges of 5%, i.e. $[1\%, 5\%[$, $[5\%, 10\%[$, ..., $[95\%, 100\%[$. Note that we start with 1% because a version with $< 1\%$ does contains too little information about defects and will not be useful for defect prediction.

Then, we assign each version into the respective bin based on its defective rate. For example, if version v_1 has defective rate of $\%D_r(v_1) = 7\%$, then v_1 is going to be assigned to the second bin $([5\%, 10\%[)$.

After assigning the versions into each bin, we select the k versions based on the following criterion:

- Apply the selection in ascending order, i.e., start from the bins with the least number of defects to the most number of defects.
- Select versions from a single bin, since we only want to recommend versions adjacent to each other.
- The selected bin must be the first bin with $l \geq k$ versions.
- From the chosen bin, it returns the k most recent versions.

The justification for the criterion is as follows: (1) we apply the selection in ascending order since the versions with the least amount of defects are the most common; thus, they are easier to select; (2) we only choose from a single bin since we want to recommend versions that have similar defective rates; therefore we avoid odd versions (e.g., bug fixes version, release candidates, and hotfixes); (3) we select the first bin with the number of versions that is higher than k since we need at least k versions from a single bin.

We present the pseudo-code for the versions recommendation system in Algorithm 1.

Algorithm 1: Versions Recommendation System

```
Input:
    $\%D_r(V_P)$ - Defective rates of all versions
    $k \geq 3$ - Number of versions to be selected
Output:
    $\{v_1, v_2, v_3, ..., v_k\}$ - Selected Versions
$bins \longleftarrow initialize\_bins(interval = 5\%)$ for
 $\%D_r(v_i) \in \%D_r(V_P)$ do
    $Add\ v_i\ to\ bins[\%D_r(v_i)]$
end
for $bin \in bins\ (ascending\ order)$ do
    if $\#bin.versions \geq k$ then
        return $most\_recent(bin.versions,\ k)$
    end
end
```

As an example, consider the following scenario. Given a project A, with 9 versions:

$$V_A = \{v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, v_{A_5}, v_{A_6}, v_{A_7}, v_{A_8}, v_{A_9}\}$$

And the respective defective rates:

$$\%D_r(A) = \{4, 38, 10, 14, 17.3, 11, 7, 12.4, 14.9\}$$

. The algorithm maps each version to the respective bin:

$$[1\%, 5\%[\longleftarrow \{v_{A_1}\}$$
$$[5\%, 10\%[\longleftarrow \{v_{A_7}\}$$
$$[10\%, 15\%[\longleftarrow \{v_{A_3}, v_{A_4}, v_{A_5}, v_{A_6}, v_{A_8}, v_{A_9}\}$$
$$[35\%, 40\%[\longleftarrow \{v_{A_2}\}$$

Assuming that $k = 5$, the first bin with the number of elements greater than k is $[10\%, 15\%[$. Therefore, the recommended versions are the k most recent versions of the selected bin: $\{v_{A_4}, v_{A_5}, v_{A_6}, v_{A_8}, v_{A_9}\}$.

B. Extractors and Metrics

Given that we have k versions for a project P, selected either from the versions recommendation system or manually, the next feature applies a set of extractors to extract the desired metrics.

The extractor feature's main idea is to collect any subset of metrics provided by BEIRUT's metric collection from a set of versions of a specific project. It runs the extractors for each file of the selected version and obtains all the extractors' metrics. Then, BEIRUT maps the obtained metrics to the specific version and aggregates them into a data set.

An extractor is any tool that follows a specific convention to extract and collect a particular set of metrics:

- It must receive a data source regarding a particular version, such as source code, change information, and so on.
- It must use the given sources to extract metrics.
- It must collect all the metrics and map them into the respective entity (e.g., file, class, method).

This abstraction allows BEIRUT to be extended easily, thus increasing the available extractable set of metrics. BEIRUT knows how to provide the required data sources for the extractor to extract the metrics from each entity and knows how to manage and collect the extracted metrics. Currently, BEIRUT uses 9 extractors. Table IV presents the extractors considered in our tool and the category of metrics that each one extracts.

Formally, BEIRUT is given a set V_S, representing the selected versions, composed of k versions: $V_s = \{v_1, v_2, ..., v_k\}$. Moreover, it is given a set of p extractors $E = \{e_1, e_2, ..., e_p\}$ able to extract a specific number of metrics from the source code, and a set $M_S \subseteq M$, that contains the metrics the user wants to extract for each version. BEIRUT is able to map the respective extractors that need to be executed to the selected metrics M_S and, later, collect the selected metrics:

Therefore, since each version is composed of a set of files $v_i = \{f_1, f_2, ..., f_{n+m}\}$, BEIRUT applies each extractor e_j to all files and collects the respective set of metrics, which are, in the end, aggregated into a data set:

$$\{\bigcup E_m(f)), f \in v_i \wedge m \in M_S\}$$

Algorithm 2 illustrates the algorithm to extract and collect the desired metrics from each version.

Algorithm 2: Metrics Extractor System

```
Input:
    $version \in V_S$
    $M_S$ - Selected Metrics
    $E$ - Extractors
Output:
    $\{f, m_{1f}, ..., m_{if} \mid f \in version \wedge m_i \in M_S\}$
$extracted\_metrics \longleftarrow \emptyset$ for $metric \in M_S$ do
    $extractor \longleftarrow get\_extractor\_for(metric, E)$
    $metrics \longleftarrow extractor.extract(version)$
    $extracted\_metrics.collect(version, metrics)$
end
return extracted_metrics
```

C. Generating Defect Prediction Model

Defect prediction is the task of predicting which components in the software are more likely to be defective. It is a relevant task since it can help developers pinpoint possible faulty locations in the source code, which need more careful analysis. For instance, one application is to consider the components that may be defective and only run (or run first) the test suites that cover them, thus promoting a fast delivery of software while maintaining its quality. Commonly, defect prediction is applied as a classification task, where the model predicts whether a specific component is defective or not.

Classification is a supervised learning strategy that builds predictive models by training a classification algorithm to predict specific labels. A classification algorithm's primary goal is to analyze a set of N training examples and produce an inferred function $h : X \longrightarrow Y$ that should map new examples. Therefore, since it is a generalization problem, optimally, the function would correctly determine the class labels of all unseen instances. Moreover, the set used to create the model has the form $\{(x_1, y_1), ..., (x_N,\ \ y_N)\}\{(x_1, y_1), ..., (x_N,\ \ y_N)\}$, where the $x_i \in X$ is a row representing a feature vector and $y_i \in Y$ is the label that function h should output when given x_i. After building a classification model, a testing set comprised of feature vectors is used to evaluate the performance of the new classifier by comparing the actual labels y_i and predicted labels $\hat{y_i}$. In classification with binary labels, the most common methods to evaluate the models use the confusion matrix, which considers true and false positives, and true and false negatives to evaluate the prediction quality. The most common evaluation methods are Area Under the Curve (AUC), the precision, the recall, and the F1-score.

Defect prediction approaches mostly use binary classification to predict defects. The main process uses several metrics extracted from each component of a software archive and trains a classification model with the information of whether each component is defective or not. In the end, one of the most important elements to develop good defect prediction models is based on which metrics are used to predict the defects. BEIRUT analyses 512 projects and extracts 644 metrics, and labels each component to whether it is defective or not. Therefore, there is an excellent motivation for defect prediction applications to use BEIRUT, since it is able to extract a large number of metrics over a large number of projects, and it can also select and extract the best versions or each project to apply defect prediction.

TABLE IV
EXTRACTORS IN BEIRUT AND RESPECTIVE EXTRACTED METRICS.

	Traditional Metrics			Object-oriented Metrics					Process Metrics		
	Size	Halstead	McCabe	CK	Mood	LK	Smells	Other	Task Changes	Change Metrics	Developer Metrics
Designite [40]	-	-	✓	✓	-	-	✓	✓	-	-	-
Checkstyle [41]	✓	-	✓	-	-	-	-	✓	-	-	-
SourceMonitor [42]	✓	-	-	-	-	-	-	-	-	-	-
CK [43]	✓	-	-	✓	-	-	-	✓	-	-	-
Mood [44]	-	-	-	-	✓	-	-	-	-	-	-
Halstead†	-	✓	-	-	-	-	-	-	-	-	-
Jasome [45]	✓	-	✓	✓	✓	✓	-	✓	-	-	-
Process†	-	-	-	-	-	-	-	-	-	✓	-
Issues†	-	-	-	-	-	-	-	-	✓	✓	✓

† Extractor developed by authors for BEIRUT.

IV. Tool Demo

In this section, we describe BEIRUT's web service interface, as we demonstrate a simple use case [1].

The tool's interface is composed of two main components: a form to run the extractors on a **new project**; and a menu to view and interact with the **already extracted projects**, to select and download the desired features.

Initially, as a required step, the tool requests the user to register himself, followed by a request to login into the system. Therefore, the user must fill in a username, email, and password. The tool has a notification system that will send the user an email once the project and extraction are completed.

A. Request for a New Project

The first available component is a form that allows the user to request a new project to extract and collect all features. The form requires a project id from the version control system, particularly, for now, BEIRUT requests a GIT id; and a project id for the issue tracking system, for now, from either Jira or Bugzilla. The system will retrieve the Git and issue tracker's data and run all the extractors for every project in the background. Hence, it will extract all the metrics available in BEIRUT's catalog.

After the user requests the project and the extraction is complete, BEIRUT sends a notification to the user's email. From this point on, it will include the extracted project in the menu to view, interact, and download the desired versions and features.

B. Download Extracted Projects

The second component of BEIRUT's interface consists of a project and version selection menu to select the desired versions and a feature selection menu to decide which metrics are to be downloaded. This menu provides an interaction to download already extracted benchmarks (with the projects and metrics included in this paper) and to obtain the benchmarks extracted using the menu from the previous section.

[1]Samples of the data sets with the extracted metrics and defect predictions can be accessed from [46].

Figure 3 shows the interface to select and download the desired versions. On the left sidebar, selecting the desired project outputs a table on the right, with necessary information regarding each version of the project. It includes the version name, the number of committed files in the version, the number of defective files in the version, the ratio of defects, the number of commits, the number of defective commits, the percentage of defective commits, the version data, the URL to the version commit tree, and the type of the version (e.g., minor and major). The tool allows the user to manually select the desired versions and projects for download or using the version recommendation system to choose the best k versions for training the prediction model as described in Section 1. The version recommendation system also gives the option to apply a flexible selection of the versions, to which it selects all the versions within the selected bin. In addition, this menu also contains a predict functionality that allows the user to obtain the result of the prediction (defective or not) and the probability classes based on the selected features. BEIRUT predicts using a random forest with 1000 estimators and with the $gini$ criterion. Note that the project previously requested for extraction, after completion, will be available in the project list for selection, download, and prediction.

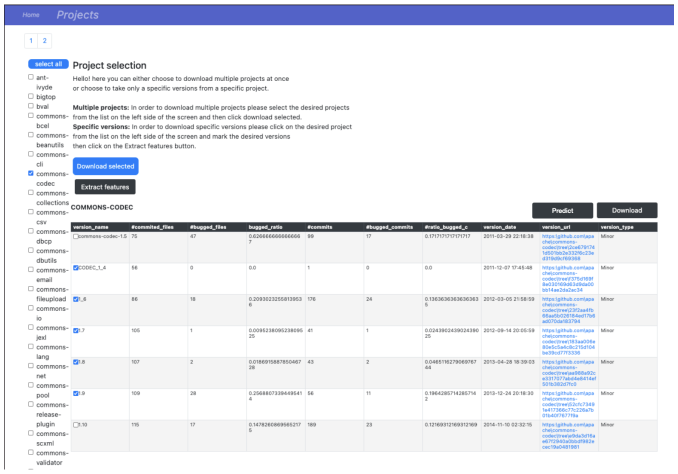
Furthermore, in addition to allowing the selection of projects and versions, it also includes a feature selection interface that enables the user to choose which feature groups to download from the selected projects and versions.

Joining all capabilities of BEIRUT, with just one click, it is possible to download all available projects with the recommended versions and features.

V. Discussion and Limitations

In this section, we present the limitations of BEIRUT and the threats to validity in this tool.

Although BEIRUT outperforms several other benchmarks regarding the number of projects and features, it has some limitations that can be further extended in future work. First, BEIRUT is only able to extract projects that are written in Java. This is a characteristic of the extractors currently used by



Project selection

Hello! here you can either choose to download multiple projects at once
or choose to take only a specific versions from a specific project.

Multiple projects: In order to download multiple projects please select the desired projects
from the list on the left side of the screen and then click download selected.
Specific versions: In order to download specific versions please click on the desired project
from the list on the left side of the screen and mark the desired versions
then click on the Extract features button.

COMMONS-CODEC

version_name	#commited_files	#bugged_files	bugged_ratio	#commits	#bugged_commits	#ratio_bugged_c	version_date	version_url	version_type
[ ] commons-codec-1.5	75	47	0.6266666666666667	99	17	0.1717171717171717	2011-03-29 22:18:38	https:\github.com\apache\commons-codec\tree\2ce6791741d501bb2e332f6c23ed319d9cf69368	Minor
[✓] CODEC_1_4	56	0	0.0	1	0	0.0	2011-12-07 17:45:48	https:\github.com\apache\commons-codec\tree\f375d169f8e030169d63d9da00bb14ae2da2ac34	Minor
[✓] 1_6	86	18	0.20930232558139536	176	24	0.13636363636363635	2012-03-05 21:58:59	https:\github.com\apache\commons-codec\tree\23f2aa4fb66aa5b026184ed17b6ad070da183794	Minor
[✓] 1.7	105	1	0.009523809523809525	41	1	0.024390243902439025	2012-09-14 20:05:59	https:\github.com\apache\commons-codec\tree\183aa006e80e5c5a4c8c215d104be39cd77f3336	Minor
[✓] 1.8	107	2	0.018691588785046728	43	2	0.046511627906976744	2013-04-28 18:39:03	https:\github.com\apache\commons-codec\tree\aa988a92ce3317077abd4e8414ef501b382d7fc0	Minor
[✓] 1.9	109	28	0.25688073394495414	56	11	0.19642857142857142	2013-12-24 20:18:30	https:\github.com\apache\commons-codec\tree\52cfc73491e417366c77c226a7b01b40f7677f9a	Minor
[ ] 1.10	115	17	0.14782608695652175	189	23	0.12169312169312169	2014-11-10 02:32:15	https:\github.com\apache\commons-codec\tree\e9da3d16ae67f2940a0bbdf982ecec19a0481981	Minor

Fig. 3. Menu to select the projects and versions to download.

the tool and can easily be extended by adding new extractors that support other programming languages. Second, BEIRUT can only extract projects that use Git as the version control system and Jira or Bugzilla as the issue tracking system. This limitation can be extended by adapting the process of obtaining the source code and defects in the initial step. Third, the version recommendation system (first feature) is limited to only versions from a single project. Ideally, the tool should be able to select versions from any project in BEIRUT and create a benchmark from it, thus allowing the option to perform cross-project defect prediction. Last, there are metrics from benchmarks in the literature that BEIRUT does not implement, for example, the change genealogy metrics from NETGENE [9]. This could be expanded in future work by creating new extractors that implement such metrics.

A threat to validity in our tool is the choice of the algorithm used to extract the defects, to which we used a variant of the SZZ. In addition, BEIRUT is biased towards open source projects.

VI. Conclusion

Defect prediction is an important task to improve the time of software development while maintaining its quality. The general approach uses supervised learning algorithms to classify software components as defective. An essential factor in defect prediction is the features, and the software archives used to train the classification algorithms. Several benchmarks have been proposed for defect prediction over the literature, and they have been the go-to benchmarks for novel defect prediction approaches. However, these benchmarks are very limited in terms of number of projects, number of versions, size of each project, number of features, age, availability, extensibility, and usability. Therefore, we introduced BEIRUT, a novel tool able to generate benchmarks targeted for defect prediction. It currently extracts 644 features from 512 projects, and it is capable of being easily extended for new features and projects. Moreover, it includes a web service with an interface that allows an easy-to-use interaction with the tool and the benchmarks. BEIRUT is composed of three features that can be interacted with by using the interface: (1) a version recommendation system, (2) a feature extraction system composed of multiple tools that extract metrics from each of the selected versions, and (3) a defect prediction feature that applies defect prediction using the feature extractor's data set. This paper evaluated our tool by comparing it with several other benchmarks proposed in the literature. We showed that BEIRUT provides better capabilities on the several limitations presented on the other benchmarks. As future work, we see BEIRUT being extended for more features and metrics. Furthermore, it could be extended for vulnerabilities and be extended to categorize different types of defects. It would also be interesting to expand the granularity of the feature extraction components, for example, to package granularity. We also want to explore more applications for BEIRUT.

References

[1] Y. Kamei and E. Shihab, "Defect Prediction: Accomplishments and Future Challenges," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita, Osaka, Japan: IEEE, Mar. 2016, pp. 33–45. [Online]. Available: http://ieeexplore.ieee.org/document/7476771/

[2] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An Empirical Study on the Use of Defect Prediction for Test Case Prioritization," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Xi'an, China: IEEE, Apr. 2019, pp. 346–357. [Online]. Available: https://ieeexplore.ieee.org/document/8730206/

[3] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, no. 3, pp. 161–175, Jun. 2018. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2017.0148

[4] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000426

[5] E. Shihab, "An Exploration of Challenges Limiting Pragmatic Software Defect Prediction," PhD, Queen's University, Kingston, Ontario, Canada, Aug. 2012. [Online]. Available: https://qspace.library.queensu.ca/bitstream/handle/1974/7354/Emad_Shihab_201208_PhD.pdf?sequence=3

[6] J. Sayyad Shirabad and T. Menzies, "The PROMISE Repository of Software Engineering Databases." 2005, published: School of Information Technology and Engineering, University of Ottawa, Canada. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[7] H. Altinger, S. Siegl, Y. Dajsuren, and F. Wotawa, "A Novel Industry Grade Dataset for Fault Prediction Based on Model-Driven Developed Automotive Embedded Software," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, Italy: IEEE, May 2015, pp. 494–497. [Online]. Available: http://ieeexplore.ieee.org/document/7180126/

[8] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. Cape Town, South Africa: IEEE, May 2010, pp. 31–41. [Online]. Available: http://ieeexplore.ieee.org/document/5463279/

[9] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Pasadena, CA, USA: IEEE, Nov. 2013, pp. 118–127. [Online]. Available: http://ieeexplore.ieee.org/document/6698911/

[10] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*. Timi&#351;oara, Romania: ACM Press, 2010, p. 1. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1868328.1868342

[11] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: a database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. San Jose, CA, USA: ACM Press, 2014, pp. 437–440. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2610384.2628055

[12] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6341763/

[13] S. Kim, H. Zhang, R. Wu, and L. Gong, “Dealing with noise in defect prediction,” in *Proceeding of the 33rd international conference on Software engineering - ICSE ’11*. Waikiki, Honolulu, HI, USA: ACM Press, 2011, p. 481. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1985793.1985859

[14] M. Shepperd, Q. Song, Z. Sun, and C. Mair, “Data Quality: Some Comments on the NASA Software Defect Datasets,” *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, Sep. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6464273/

[15] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, “ReLink: recovering links between bugs and changes,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE ’11*. Szeged, Hungary: ACM Press, 2011, p. 15. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2025113.2025120

[16] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting Defects for Eclipse,” in *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*. Minneapolis, MN, USA: IEEE, May 2007, pp. 9–9. [Online]. Available: http://ieeexplore.ieee.org/document/4273265/

[17] A. G. Koru and H. Liu, “An investigation of the effect of module size on defect prediction using static measures,” in *Proceedings of the 2005 workshop on Predictor models in software engineering - PROMISE ’05*. St. Louis, Missouri: ACM Press, 2005, pp. 1–5. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1083165.1083172

[18] M. H. Halstead, *Elements of software science*, ser. Operating and programming systems series. New York: Elsevier, 1977, no. 2.

[19] T. J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, conference Name: IEEE Transactions on Software Engineering.

[20] S. R. Chidamber and C. F. Kemerer, “Towards a metrics suite for object oriented design,” *ACM SIGPLAN Notices*, vol. 26, no. 11, pp. 197–211, Nov. 1991. [Online]. Available: https://dl.acm.org/doi/10.1145/118014.117970

[21] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://ieeexplore.ieee.org/document/295895/

[22] F. Brito e Abreu and R. Carapuça, “Object-Oriented Software Engineering: Measuring And Controlling The Development Process,” in *4th International*. McLean, VA, USA: Zenodo, Oct. 1994, publisher: Zenodo. [Online]. Available: https://zenodo.org/record/1217609

[23] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*, ser. Prentice Hall object-oriented series. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[24] W. J. Brown, Ed., *AntiPatterns: refactoring software, architectures, and projects in crisis*. New York: Wiley, 1998.

[25] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*, ser. The Addison-Wesley object technology series. Reading, MA: Addison-Wesley, 1999.

[26] S. Ganesh, T. Sharma, and G. Suryanarayana, “Towards a Principle-based Classification of Structural Design Smells.” *The Journal of Object Technology*, vol. 12, no. 2, p. 1:1, 2013. [Online]. Available: http://www.jot.fm/contents/issue_2013_06/article1.html

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

[28] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, May 2005, pp. 284–292, iSSN: 1558-1225.

[29] M. D’Ambros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: a benchmark and an extensive comparison,” *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, Aug. 2012. [Online]. Available: https://doi.org/10.1007/s10664-011-9173-9

[30] J. C. Munson and S. G. Elbaum, “Code churn: a measure for estimating the impact of code change,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, Nov. 1998, pp. 24–31, iSSN: 1063-6773.

[31] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE ’08. New York, NY, USA: Association for Computing Machinery, May 2008, pp. 181–190. [Online]. Available: https://doi.org/10.1145/1368088.1368114

[32] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, “Using Developer Information as a Factor for Fault Prediction,” in *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*, May 2007, pp. 8–8.

[33] T. Menzies, J. Greenwald, and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors,” *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan. 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4027145/

[34] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, “On the relative value of cross-company and within-company data for defect prediction,” *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, Oct. 2009. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9103-7

[35] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. USA: Prentice-Hall, Inc., 1995.

[36] J. Bansiya and C. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, Jan. 2002. [Online]. Available: http://ieeexplore.ieee.org/document/979986/

[37] M. Borg, O. Svensson, K. Berg, and D. Hansson, “SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project,” in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*. Tallinn, Estonia: ACM Press, 2019, pp. 7–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3340482.3342742

[38] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, “Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection,” *arXiv:1911.08938 [cs]*, Feb. 2020, arXiv: 1911.08938. [Online]. Available: http://arxiv.org/abs/1911.08938

[39] K. Herzig, S. Just, and A. Zeller, “It’s not a bug, it’s a feature: How misclassification impacts bug prediction,” in *Proceedings of the 2013 international conference on software engineering*, ser. ICSE ’13. IEEE Press, 2013, pp. 392–401, place: San Francisco, CA, USA Number of pages: 10.

[40] T. Sharma, “Designite - a software design quality assessment tool,” May 2016. [Online]. Available: https://doi.org/10.5281/zenodo.2566832

[41] “Checkstyle.” [Online]. Available: https://checkstyle.sourceforge.io/

[42] “Source Monitor.” [Online]. Available: http://www.campwoodsw.com/sourcemonitor.html

[43] M. Aniche, “Java code metrics calculator (CK),” 2015.

[44] Thainamariani, “Mood.” [Online]. Available: https://github.com/thainamariani/gorgeous_metrics

[45] Rodhilton, “rodhilton/jasome.” [Online]. Available: https://github.com/rodhilton/jasome

[46] Anonymous, “Beirut: Repository mining for defect prediction,” Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5082700