Improved Bug Prediction Through Conceptual Metrics and Machine Learning

DSc dissertation

Rudolf FERENC

Szeged, 2023

Contents

List of Tables

List of Figures

1 Introduction

Software quality has a long history full of shocking moments [83, 119, 142, 217], after which confessions like the following are not uncommon: "If we had paid more attention to quality, this could have never happened...". Fortunately, software companies have been increasingly realizing the importance of quality assurance, and nowadays, they apply various monitoring techniques to regularly get a clear picture of the quality of their systems. This realization also pushed researchers toward exploring and understanding the complex connections between different software quality characteristics.

Two of the most basic yet hardest questions about software quality are 1) what it exactly is, and 2) how we can measure it. To answer these questions, we have to break quality down into smaller components that describe its different aspects. Various standards – such as the ISO/IEC 9126 [28] or its successor, the ISO/IEC 25010 [29] – have been proposed to address these challenges. However, these standards suggest aspects of a higher abstraction level only (e.g. maintainability and reliability) without specifying the exact low-level attributes. Consequently, several implementations have been presented to solve this problem, including software quality models [1]. Quality models rely on exact source code metrics for measuring different aspects of a system, such as size, complexity, documentation, coupling, and cohesion. These low-level attributes can be measured directly from the source code and can be propagated to higher-levels in order to measure quality subcharacteristics like stability and testability (which are in turn subcharacteristics of maintainability – an important component of quality).

One of the earliest realizations is that source code metrics are key components in describing quality. Cohesion and coupling metrics have shown their usefulness in different scenarios like fault prediction [16] and impact analysis [55, 232], which have direct influence on the stability and testability of software. Besides traditional coupling and cohesion metrics that capture the structure of the source code, their conceptual counterparts were also defined to capture a more abstract measurement of the system. In this thesis, we present the conceptual cohesion and coupling metrics that we defined and show their usefulness in fault prediction and in impact analysis.

Another quality indicator of software is the number of bugs it contains. The cost of maintaining existing software often exceeds the cost of the initial development [103].

Therefore, finding software bugs as soon as possible is extremely important because even if we follow all best practices, use up-to-date components, and use the latest language version idiomatically, we are still human, so a bug may find its way into the system even with the most rigorous review process. Bugs can cause damage (both financial and reputational) to both software development companies and software users. Finding software bugs can be extremely time-consuming, even if we use static or dynamic analyzer tools designed for this exact purpose. These tools often miss real bugs and have a high false positive rate. As a result, developers often do not take the tools' warnings seriously.

Nowadays, artificial intelligence-based solutions can also be employed to predict software bugs. But a significant amount of data is needed to train artificial intelligence algorithms effectively. Fortunately, with the proliferation of open source software (and version control systems such as GitHub or GitLab), researchers have access to a rich set of historical data of software development. However, the information is available in different formats and systems (even within the same project), and gathering them can be cumbersome. For this reason, there are bug datasets that contain a collection of software bugs from distinct systems; however, they are available in different formats, which makes it challenging to use them collectively in the field of artificial intelligence. In this work, we show how we combined and extended existing bug datasets to run machine learning algorithms and have the best possible outcome.

We know that for artificial intelligence, the more data, the better. One of the most outstanding is the increasingly popular deep artificial neural networks, which can be trained successfully with vast amounts of data. An important result we show in this thesis is a unified bug database, which we have created and used to evaluate several traditional machine learning algorithms. We also show in detail how software metrics can be used to predict bugs using deep neural networks. We describe the process step-by-step, including all the fine-tuning needed to achieve a good result.

In Chapter 2, we first introduce some general background information containing the absolute necessities to understand the rest of this work. Having provided the necessary foundation, we describe the thesis points of this work as chapters in their corresponding parts.

Part One deals with conceptual coupling and cohesion metrics.

T1 **Conceptual Cohesion in Fault Prediction.** I describe the conceptual cohesion metric $C3$, and then show by principal component analysis that it forms an independent dimension with respect to the structural cohesion metrics. Furthermore, I show that the $C3$ metric improves the error prediction capabilities of structural cohesion metrics (Chapter 3) [22].

T2 **Conceptual Coupling in Impact Analysis.** I describe the $CCBC$ and $CCBC_m$ as well as the $CoCC$ and $CoCC_m$ conceptual coupling metrics, and then show that the $CCBC_m$ metric ranks the related classes the best in impact analysis, far ahead of the structural coupling metrics from this point of view (Chapter 4) [23].

T3 **New Conceptual Coupling and Cohesion Metrics.** I define the parameterized new conceptual metrics $CCBO$ and $CLOM5$, which are easier to compute than their structural counterparts, while their fault prediction abilities are very similar. I also show that these conceptual metrics improve the error prediction capabilities of structural metrics (Chapter 5) [27].

Part Two deals with machine learning for bug prediction.

T4 **A Public Unified Bug Dataset for Bug Prediction.** I summarize how we explored all available, widely used Java bug datasets, compared and combined them to create a unified bug dataset, supplemented by a number of source code metrics we calculated. I show that the unified bug dataset can be used well for creating bug prediction machine learning models (within-project, merged, and cross-project) and I evaluate the results (Chapter 6) [12].

T5 **Deep Learning for Bug Prediction.** I provide a detailed methodology for using deep neural networks in bug prediction and finding optimal hyperparameters. I build a deep neural network on the unified bug dataset and compare it with the results of conventional machine learning algorithms. I show that only the random forest can outperform it. The combination of these two gives ultimately the best results. We also show that more learning data is expected to further improve the performance of the deep neural network (Chapter 7) [4].

Lastly, Chapter 8 concludes our discussion and outlines some possible directions for future work.

ferenc.rudolf_87_23

2 Background

Before diving deep into the details and the main results of the thesis, we briefly introduce some concepts and fundamental topics since they occur from time to time in this work and form the basis of all future discussions. Static source code analysis, the tools we used, and the static source code metrics we can obtain are key building blocks in this thesis. After introducing these topics, we will describe the LSI, a widely-used information retrieval technique, and also provide a quick overview of the statistical methods and machine learning concepts we utilize throughout the thesis.

2.1 Static Source Code Analysis

Static analysis is widely used to examine programs without having to run/execute them (in contrast with dynamic analysis). The input of the analysis is often the raw source code itself, however, precompiled binaries could also be used (e.g. in the case of Java, bytecode is also a common input for static analysis tools).

The main goals of a static analysis include obtaining static source code metrics, detecting coding rule violations, and revealing code duplications – all of which can be key to understanding the behavior of the subject software system.

Source code metrics are especially important since they form a vital input for further higher-level investigations, such as refactoring and bug prediction. In this work, we mainly focus on software *product metrics*, which characterize the software itself (while process metrics describe the development and maintenance during the project). Product metrics can be categorized into groups like size (logical lines of code, number of methods), complexity (McCabe's Cyclomatic Complexity), coupling (number of incoming/outgoing method calls), and cohesion (LCOM) metrics.

2.2 OpenStaticAnalyzer

OpenStaticAnalyzer (OSA)[1] is a source code analyzer tool, which can perform deep static analysis of the source code of complex systems. The tool's earlier name is *Columbus* [6].

The source code of a program is usually its only up-to-date documentation. At the same time, the source code is the exquisite bearer of knowledge, business processes and methodology, accumulated over a long period of time. Source code quality decrease, which happens due to many quick fixes and time pressure, results in the increase of development and testing costs, and operational risks. In spite of this, the source code usually receives hostile treatment and is merely considered as a tool.

OSA provides deep static analysis of source code. Using the results of the analysis, the quality of the analyzed source code can be improved and developed both in the short- and long term in a directed way.

It calculates more than 50 different (size, complexity, coupling, cohesion, inheritance, and documentation) source code metrics for packages and class-level elements, about 30 metrics for methods, and a few for files. OSA can also detect code duplications (Type-1 and Type-2 clones) and calculates code duplication metrics for packages, classes, and methods.

The tool was used in many different publications (e.g. [19, 14, 2, 20, 9, 24]), which shows the strength of the core idea behind it.

2.3 SourceMeter

SourceMeter[2] [9] is a commercial product based on OpenStaticAnalyzer (see Section 2.2). It is an innovative tool built for the precise static source code analysis of C, C++, Java, C#, Python, JavaScript, and RPG projects. This tool makes it possible to find the weak spots of a system under development from the source code only, without the need of simulating live conditions. The tool integrates also the best of available free static checker tools (Cppcheck, PMD, FindBugs, FxCop, Pylint) and presents their results in a unified way. Using the results of the analysis, the quality of the analyzed source code can be improved and developed both in the short– and long term in a directed way. Free version with limited functionality is available for all programming languages.

2.4 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is a corpus-based statistical method for inducing and representing aspects of the meanings of words and passages (of natural language) reflective in their usage in large bodies of text [80, 85]. LSI is based on a vector space model (VSM) [213], as it generates a real valued vector description for documents of text. Results have shown [44, 145] that LSI captures significant portions of the meaning not only of individual words but also of whole passages such as sentences, paragraphs, and short essays. The central concept of LSI is that the information about the contexts in which a particular word appears or does not appear provides a set of mutual

[1]https://openstaticanalyzer.github.io/
[2]http://sourcemeter.com/

constraints that determines the similarity of the meaning of sets of words compared to each other.

Since its first introduction, LSI has been used to support various code analysis tasks such as concept location [198], identification of abstract data types [162], clone detection [224], traceability link recovery among software artifacts [36, 78, 167], software clustering [141], quality assessment [146] and software measurement [82, 171, 22, 199, 23].

LSI was originally developed in the context of Information Retrieval (IR) as a way of overcoming problems with polysemy and synonymy that occurred in other VSM approaches. Some words appear in the same contexts, and an important part of word usage patterns is blurred by accidental and inessential information. The method used by LSI to capture the essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix (words by context) decomposed using singular value decomposition (SVD) [213]. As a result, LSI offers a way of assessing semantic similarity between any two samples of text in an automatic, unsupervised way.

LSI relies on a singular value decomposition of a matrix derived from a corpus of natural text that pertains to knowledge in the particular domain of interest. According to the mathematical formulation of LSI, the term combinations which occur less frequently in the given document collection tend to be precluded from the LSI subspace. LSI does "noise reduction" as less frequently co-occurring terms are less mutually related and, therefore, less sensible. Similarly, the most frequent terms are also eliminated from the analysis. The formalism behind SVD is rather complex and too lengthy to be presented here. The interested reader may refer to the work of Salton and McGill [213] for details.

Once the documents are represented in the LSI subspace, the user can compute similarity measures between documents using either the cosine between their corresponding vectors or their length. These measures can be used for clustering similar documents together to identify "concepts" and "topics" in the corpus. This type of usage is typical for text analysis tasks. Uses of LSI in software engineering are presented and discussed in [168].

2.5 Statistical Methods

Throughout the thesis, we will use a variety of statistical methods to corroborate our hypotheses in an objective manner. We list the most important ones here as a precursor.

2.5.1 Correlation Analysis

To try and find a direct or inverse linear relationship between two variables, we frequently utilize Pearson's correlation, leading to a p-value from within the [-1,1] interval. Values closer to 1 indicate direct dependence, while values closer to -1 indicate an inverse dependence. Values close to 0 means that we were not able to draw any meaningful conclusions – which is not the absence of a relationship, only a linear one.

We also acknowledge that correlation is *not* causation, therefore we are intentionally careful in the interpretation of such results.

2.5.2 Principal Component Analysis

Principal Component Analysis (PCA) is widely used in many domains to accomplish dimensionality reduction and uncover patterns in the data. PCA determines which dimensions are the most important and which ones represent the most variation in the data. PCA takes a dataset (a set of metrics in the cases we will use it in) as input and outputs principal components (uncorrelated dimensions) that span the direction of the 1st, 2nd, 3rd, ... largest variations. The overall purpose of PCA is to identify factors that explain as much of the variation with as few factors as possible.

2.5.3 Statistical tests

We frequently utilize different statistical tests to accept or reject certain hypotheses.

One is the Wilcoxon's signed-rank test [231] (a.k.a. Mann-Whitney U test [166]), which is a non-parametric statistical test to analyze whether the distribution of the values differs significantly between two groups. Moreover, the mean rank values produced by the test can be used to decide the direction of the differences.

Another test we reference is Kruskal-Wallis's test [140, 219], which is a non-parametric alternative to the one-way analysis of variance in those cases when more than three independent samples are present. It can be thought of as the generalization of the Mann-Whitney U test.

Lastly, we use *Cohen's d* as well, which indicates the standardized difference between two means. If this difference, namely Cohen's d value, is less than 0.2 we say that the effect size is small and more than 90% of the two groups overlap. If Cohen's d value is between 0.2 and 0.5, the effect size is medium, and if the value is larger than 0.8 the effect size is large.

2.6 Machine Learning Techniques

To make various predictions based on the available data, we also use multiple machine learning techniques. Here, we summarize the two main groups of algorithms – regression and classification – and then discuss how we evaluate the resulting models.

2.6.1 Regression

Regression analysis is a statistical process for estimating the relationships among variables. It tries to predict how the typical value of the dependent variable changes when any one of the independent variables changes. It accomplishes this by giving an estimation for the dependent variable from a continuous interval. There are different kinds of regressions, of which we focused mostly on *logistic regression*.

In **logistic regression**, the unknown variable (*i.e.*, the *dependent variable*) can take one of only two different values (usually labeled "0" and "1", but this can change depending on the context). Also, the known variables (also called the explanatory, or *independent* variables) can change over different ranges, which is often addressed through standardization, *i.e.*, each metric is transformed to have a zero mean and unit variance.

$$\pi(X_1, X_2) = \frac{e^{C_0 + C_1 \times X_{i1} + C_2 \times X_{i2}}}{1 + e^{C_0 + C_1 \times X_{i1} + C_2 \times X_{i2}}} \tag{2.1}$$

The **multivariate logistic regression** model is based on the relationship equation (shown in equation 2.1) where X_is are the explanatory variables and π is the probability that the dependent variable would be found as "1" during the validation procedure. The coefficients C_1 and C_2, are the estimated regression coefficients. The larger the absolute value of the coefficient, the stronger the impact (positive or negative, according to the sign of the coefficient) of the explanatory variable on the probability of the dependent variable. **Univariate logistic regression** is a special case of multivariate regression where only one exploratory variable X_1 is used.

The typical performance measures of these estimations are Pearson's correlation coefficient (showing how well the predicted values follow the tendency of the real value of the dependent variable) and Mean Absolute Error (which is a quantity used to measure how close forecasts or predictions are to the eventual outcomes). Another performance measure is the R^2 coefficient, which is defined as the proportion of the total variation in the dependent variable that is explained by the regression model. The bigger the value of R^2, the larger the portion of the total variance in the dependent variable that is explained by the regression model and the better the dependent variable is explained by the explanatory variables.

Since logistic regression is a commonly used statistical method, it will not be explained here in more detail. For a deeper discussion on regression analyses (and their applications on using metrics as we will use them in later chapters), the reader is referred to previous work [41, 56, 16, 222].

2.6.2 Classification

As opposed to a continuous interval where regression analyses take their results from, classification is a discrete process of choosing the most likely class an unclassified instance belongs to. Pre-classified observations are used to build such models through the use of various algorithms, the two most relevant of which we will discuss here.

Decision Tree. C4.5 is an enhanced implementation of the ID3 algorithm that was proposed by Quinlan in 1993 [204]. The C4.5 algorithm makes use of a variant of the rule post-pruning method to find high-precision hypotheses for the target concept of the learning problem. It generates a classification-decision tree for the given data set by recursively partitioning the data. Training examples are described by attributes (predictor values) whose choice for a given tree node depends on their information gain at each step during the growth of the tree.

Some of the features of the algorithm are that it results in smaller decision trees, it uses a depth-first strategy, and over-fitting is allowed – meaning that the growth of the tree continues until it best fits the training data. After the tree has been built up, it will be converted into an equivalent set of rules, all of which incorporate tests of the predictor values and that will provide the output decision.

Lastly, pruning is applied to the rules to reduce the size of the tree by rearranging and removing similar branches. C4.5 can handle both discrete valued predictors and continuous ones as well, and also training examples with missing predictor values.

Neural Network. The Backpropagation algorithm [46] works with neural networks that are the means for machine learning, whose reasoning concept was borrowed from the workings of the human brain.

This algorithm uses more layers of neurons; it gets the input patterns and gives them to the input layers. Then it computes the output layer (the output decision) from the input layer and the hidden (inner) layers. In addition, an error value is also calculated from the difference between the output layer and the target output pattern (the learning data).

The error value is propagated backward through the network, and the values of the connections between the layers are adjusted in such a way that the next time the output layer is computed, the result will be closer to the target output pattern. This method is repeated until the output layer, and target output pattern are almost equal or up to some iteration limit.

Others. In addition to the previous ones, we also utilize a suite of other machine learning techniques implemented in Weka [109] and scikit-learn [196] – open-source collections of machine learning algorithms for data mining tasks.

2.6.3 Evaluation

To evaluate our models, we commonly use N-fold cross-validation. In an N-fold cross-validation process, the original dataset is randomly partitioned into N subsamples, desirably equal in size, if possible. Out of the N subsamples, 1 subsample is retained as the validation data for testing the model, and the other N-1 subsamples are used as training data. The cross-validation process is then repeated N times (the number of folds), with each of the N subsamples used exactly once as the validation data. The results from the folds are then averaged to produce a single estimation. The most common value for N is 10, but it can be more or less depending on the amount of data available.

The results of a binary classification (which is what we focus on) can be summarized in a confusion matrix, separating the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) cases – also, $P = TP + FP$, and $N = TN + FN$. The performance of the classification is then usually described through various measures dependent on the cardinality of these sets.

- **Accuracy** = $(TP + TN)/(P + N)$ – measures how often the prediction is correct overall.
- **Precision** = $(TP)/(TP + FP)$ – measures how often the positive predictions are correct.
- **Recall** = $(TP)/(TP + FN)$ – measures the ratio of the positive instances we managed to find (TP rate).
- **F-Measure** = $(2 \times \mathit{Precision} \times \mathit{Recall})/(\mathit{Precision} + \mathit{Recall})$ – a weighted harmonic mean of precision and recall, frequently used as a comprehensive indicator of combined precision and recall values.
- **ROC curve** = Receiver Operating Characteristics, mapping the relationship between TP rate and FP rate at different classification thresholds.
- **AUC** = Area under the ROC curve.

Part I

Conceptual Coupling and Cohesion Metrics

ferenc.rudolf_87_23

3 Conceptual Cohesion in Fault Prediction

3.1 Introduction

Software modularization, Object-Oriented (OO) decomposition in particular, is an approach to improve the organization and comprehension of source code. In order to understand OO software, software engineers need to create a well connected representation of the classes that make up the system. Each class must be understood individually, and then the relationships among the classes must be understood as well. One of the goals of OO analysis and design is to create a system where classes have high cohesion and there is low coupling among them. These class properties facilitate comprehension, testing, reusability, maintainability, *etc.*

Software cohesion can be defined as a measure of the degree to which elements of a module belong together [45]. Cohesion can also be regarded from a conceptual point of view; in this view, a cohesive module is a crisp abstraction of a concept or feature from the problem domain, usually described in the requirements or specifications. Such definitions, while very intuitive, are quite vague and make actually measuring cohesion a difficult task, leaving too much room for interpretation. In OO software systems, cohesion is usually measured at class level and many different OO cohesion metrics have been proposed (see Section 3.2 for details), which try to capture different aspects of cohesion, or which reflect a particular interpretation of cohesion.

Proposals of measures and metrics for cohesion abound in the literature, as software cohesion metrics proved useful in different tasks [77] including assessment of design quality [40, 56], productivity, design and reuse effort [65], prediction of software quality, fault prediction [90, 16, 202], modularization of software [31, 162], identification of reusable components [92, 147], *etc.*

Most approaches to cohesion measurement have automation as one of their goals, as it is impractical to manually measure the cohesion of classes in large systems. The trade off is that such measures deal only with information that can be automatically extracted from software and analyzed by automated tools; thereby ignoring less structured, but rich information from the software (*e.g.*, textual information). Cohesion is usually measured using *structural* information extracted solely from the source code (*e.g.*, attribute references in methods, method calls, *etc.*) that captures the degree to which the elements of a class belong together from a structural point of view. These measures give information about the way a class is built and how its instances work together to address the goals of their design. The principle behind this class of metrics is to measure the coupling between the methods of a class. Thus, they give no clues whether the class is cohesive from a *conceptual* point of view (*e.g.*, whether a class implements one or more domain concepts), nor do they give any indication about the readability and comprehensibility of the source code. While other types of metrics were proposed by researchers (see Section 3.2 for details) to capture different aspects of cohesion, only few such metrics address the conceptual and textual aspects of cohesion [93, 171].

We propose a new measure for class cohesion, named the Conceptual Cohesion of Classes (C3), which captures the conceptual aspects of class cohesion, as it measures how strongly the methods of a class relate to each other conceptually. The conceptual relation between methods is based on the principle of textual coherence [156]. We interpret the implementation of methods as elements of discourse. There are many aspects of a discourse that contribute to coherence, including co-reference, causal relationships, connectives, and signals. Source code is far from a natural language and many aspects of natural language discourse do not exist in source code or need to be redefined. The rules of discourse are also different from natural languages.

C3 is based on the analysis of textual information in source code, expressed in comments and identifiers. Once again, this part of the source code, while closer to natural languages, is still different from them, thus using classic natural language processing methods such as propositional analysis is impractical or outright unfeasible. Hence, we use an Information Retrieval (IR) technique, namely Latent Semantic Indexing (LSI), to extract, represent, and analyze the textual information from source code (see Section 2.4). Our measure of cohesion can be interpreted as a measure of the textual coherence of a class within the context of the entire system.

Cohesion ultimately affects the comprehensibility of source code. For the source code to be easy to understand, it has to have a clear implementation logic (*i.e.*, design) and it has to be easy to read (*i.e.*, good language use). These two properties are captured by the structural and conceptual cohesion metrics, respectively.

The main contributions of our work are:

- Formal definition and calculation of the C3 metric.
- Two case studies aimed at comparing C3 with an extensive set of existing cohesion measures and assessing its ability to predict faults in source code, in combination with the existing metrics.
- Investigation of the differences between the structural and conceptual cohesion metrics.

3.2 Related Work

There are several different approaches to measure cohesion in OO systems. Many of the existing metrics are adapted from similar cohesion measures for non-OO systems (we are not discussing those here), while some of the metrics are specific for OO software.

Based on the underlying information used to measure the cohesion of a class, one can distinguish: structural metrics [45, 52, 67, 117, 122, 148, 245]; semantic metrics [93, 171]; information entropy-based metrics [34]; slice-based metrics [175, 192]; metrics based on data mining [181]; and metrics for specific types of applications like knowledge-based [138], aspect-oriented [242], and distributed systems [68].

The class of structural metrics is the most investigated category of cohesion metrics, which includes: LCOM[1], LCOM3 [122], LCOM4 [122], C_o (connectivity) [122], LCOM5 [117], Coh [52], TCC (tight class cohesion) [45], LCC (loose class cohesion) [45], ICH (information-flow-based cohesion) [148], NHD (normalized Hamming distance) [72], *etc.*

The dominating philosophy behind this category of metrics considers class variable referencing and data sharing between methods as contributing to the degree to which the methods of a class belong together. Most structural metrics define and measure relationships among the methods of a class based on this principle. Cohesion is seen dependent on the number of pairs of methods that share instance or class variables, one way or another. The differences among the structural metrics are based on the definition of the relationships among methods, the system representation, and the counting mechanism. A comprehensive overview of graph theory-based cohesion metrics is given by Zhou *et al.* [244]. Somewhat different in this class of metrics are LCOM5 and Coh, which consider cohesion directly proportional to the number of instance variables in a class that are referenced by the methods in that class. Briand *et al.* defined a unified framework for cohesion measurement in OO systems [52], which classifies and discusses all these metrics.

Other cohesion metrics exploit relationships that underline slicing [175, 192]. A large-scale empirical investigation of slice-based metrics [175] indicated that the slice-based cohesion metrics provide complementary views of cohesion to the structural metrics. While the information used by these metrics is also structural in nature, the mechanism used and the underlying interpretation of cohesion set these metrics apart from the structural metrics group.

From the perspective of measurement methodology, two other cohesion metrics are of interest here because – although used differently – they are also based on an IR approach. Patel *et al.* [195] proposed a composite cohesion metric that measures the information strength of a module. This measure is based on a vector representation of the frequencies of occurrences of data types in a module. The approach measures the cohesion of individual subprograms of a system based on the relationships to each other in this vector space. Maletic and Marcus [162] defined a file level cohesion metric based on the same type of information we are using for our proposed metrics here. Even though these metrics were not specifically designed for the measurement of cohesion in OO software, they could be extended in that direction.

The designers and the programmers of a software system often think about a class as a set of responsibilities that approximate the concept from the problem domain implemented by the class as opposed to a set of method-attribute interactions. Information that gives clues about domain concepts is encoded in the source code as comments and identifiers. Among the existing cohesion metrics for OO software, the Logical Relatedness of Methods (LORM) [91] and the Lack of Conceptual Cohesion in Methods (LCSM) [171] are the only ones that use this type of information to measure the conceptual similarity of the methods in a class. The philosophy behind this class of metrics – encompassing our work as well – is that a cohesive class is a crisp

[1]LCOM - lack of cohesion in methods, which was originally introduced [66] and subsequently extended [67] by Chidamber and Kemerer; we will refer to the first version of LCOM [66] as LCOM1 and to the extended one [67] as LCOM2.

implementation of a problem or solution domain concept. Hence, if the methods of a class are conceptually related to each other, the class is cohesive. The difficult problem here is defining and measuring conceptual relationships. LORM uses natural language processing techniques for the analysis needed to measure the conceptual similarity of methods and represents a class as a semantic network. LCSM uses the same information, indexed with LSI, and represents classes as graphs that have methods as nodes. It uses a counting mechanism similar to LCOM.

3.3 An Information Retrieval Approach to Class Cohesion Measurement

OO analysis and design methods decompose the problem addressed by the software system development into classes, in an attempt to control complexity. High cohesion for classes and low coupling among classes are design principles aimed at reducing the system's complexity. The most desirable type of cohesion for a class is model cohesion [88], such that the class implements a single, semantically meaningful concept. This is the type of cohesion we are trying to measure in our approach.

The source code of a software system contains *unstructured* and (semi-) *structured* data. The structured data is destined primarily for the parsers, while the unstructured information (*i.e.*, the comments and identifiers) is destined primarily to the human reader. Our approach is based on the premise that the unstructured information embedded in the source code reflects, to a reasonable degree, the concepts of the problem and solution domains of the software as well as the computational logic of the source code. This information captures the domain semantics of the software and adds a new layer of semantic information to the source code, in addition to the programming language semantics. Existing work on concept and feature location [172, 198], on traceability link recovery between source code and documentation [36, 170], on impact analysis [35], and other such tasks showed that our premise stands and this type of information extracted from source code is very useful.

In order to extract and analyze the unstructured information from source code, we use Latent Semantic Indexing [80], an advanced IR method (see Section 2.4). While the general approach (see Section 3.3.2) would work with other IR methods or with more complex natural language processing techniques, we decided to use LSI here as a proof of concept. LSI has been used in various software engineering problems like concept and feature location [172, 198, 200], traceability link recovery between source code and documentation [170], identification of abstract data types in legacy source code [162], clone detection [169], *etc.* Furthermore, LSI has been successfully used in cognitive psychology for the measurement of textual coherence [98], which is the principle we base our approach on.

The remainder of this section explains how LSI can be used to measure textual coherence. The extension of this concept to cohesion measurement is then discussed and the formalism behind the definition of C3 is presented together with examples.

3.3.1 Measuring Text Coherence with LSI

In a language such as English, there are many aspects of a discourse that contribute to coherence, including co-reference, causal relationships, connectives, and signals [112].

Existing approaches in cognitive psychology and computational linguistics for automatically measuring text coherence are based on propositional modeling. Foltz *et al.* [98] showed that LSI can be applied as an automated method that produces coherence predictions similar to propositional modeling.

The primary method for using LSI to make coherence predictions is to compare some unit of text to an adjoining unit of text in order to determine the degree to which the two are semantically related. These units could be sentences, paragraphs, individual words, or even whole books. This analysis can then be performed for all pairs of adjoining text units in order to characterize the overall coherence of the text. Coherence predictions have typically been performed at a propositional level, in which a set of propositions all contained within working memory are compared or connected to each other [136]. For LSI-based coherence analysis, using sentences as the basic unit of text appears to be an appropriate corresponding level that can be easily parsed by automated methods. Sentences serve as a good level in that they represent a small set of textual information (*e.g.*, typically 3-7 propositions) and thus would be approximately consistent with the amount of information that is held in short term memory.

To measure the coherence of a text, LSI is used to compute similarities between consecutive sentences in the text. High similarity between two consecutive sentences indicates the two sentences are related, whereas low similarity indicates a break in the topic. A well written article or book may provide coherence even at these break points, thus topic changes are not always marked by a lack of coherence. For example, an author may deliberately make a series of disconnected points, such as in a summary, which may not be a break in the discourse structure. The idea is that if the similarity between adjacent sentences is maintained high, the reader can follow the logic of and understand the text easier. As the similarity measure – as defined by LSI – is not transitive, it is possible to have non-adjacent sentences with very low similarity measure, yet maintain a high coherence. The overall coherence of a text is measured as the average of all similarity measures between consecutive sentences.

3.3.2 From Textual Coherence to Software Cohesion

We adapt the LSI-based coherence measurement mechanism to measure cohesion in OO software. One issue is the definition of documents in the corpus. For natural languages, sentences, paragraphs, and even sections are used as units of text to be indexed (*i.e.*, documents). Based on [171, 172, 199], we consider methods as elements of the source code that can be units for indexing. Thus the implementation of each method is converted to a document in the corpus to be indexed by LSI.

Another issue of interest lies in the extraction of relevant information from the source code. We extract all identifiers and comments from the source code. As mentioned before, we assume that developers used meaningful naming and commenting rules. One can argue that this information does not fully describe a piece of software. While this is true, significant information about the source code is embedded in this data, as [167] suggests. More than that, analogous approaches are used in other fields such as image retrieval. For example, when searching for images on the web with Google or other search engines, one really searches in the text surrounding these images in the web pages [134] (and while true image search is also possible nowadays, text based search is still the dominant technique).

Finally, Foltz's method for coherence measurement is based on measuring the similarity between adjacent elements of text (*i.e.*, sentences). OO source code does not follow the same discourse rules as natural languages, thus the concept of adjacent elements of text (*i.e.*, methods) is not present here. To overcome this issue, we compute similarities between every pair of methods in a class. There is an additional argument for this change. A coherent discourse allows for changes in topic, as long as these changes are rather smooth. In software, we interpret a cohesive class as implementing one concept (or a very small group of related concepts) from the software domain. With that in mind, each method of the class will refer to some aspect related to the implemented concept. Hence, methods should be related to each other conceptually.

We developed a tool, **I**nformation **R**etrieval based **C**onceptual **C**ohesion **C**lass **M**easurement (IRC^{3}M), which supports this methodology and automatically computes C3 for any class in a given software system. The following steps are necessary to compute the C3 metric (the tool is also used to measure the LCSM metric [171]):

- **Corpus creation.** The source code is preprocessed and parsed to produce a text corpus. Comments and identifiers from each method are extracted and processed. A document in the corpus is created for each method in every class.
- **Corpus indexing.** LSI is used to index the corpus and create an equivalent semantic space.
- **Computing conceptual similarities.** Conceptual similarities are computed between each pair of methods.
- **Computing C3.** Based on the conceptual similarity measures, C3 is computed for each class (definitions are presented in the next section).

IRC^{3}M is implemented as a MS Visual Studio .NET add-in and computes the C3 metric for C++ software projects in Visual Studio, based on the above methodology. Our source code parser component is based on the Visual C++ Object Extensibility Model. Using project information retrieved from Visual Studio .NET, the tool retrieves parts of the source code that are used to produce a corpus. For software projects developed outside .NET environment, *i.e.* Mozilla from our case study, we use external parsers (*e.g.*, Columbus [6] and srcML [161]) and a set of our own utilities to construct the corpus. The extracted comments and identifiers are processed – similarly to [172] – by elimination of stop words and splitting identifiers that follow predefined coding standards. We use the cosine similarity between vectors in the LSI space to compute conceptual relations.

3.3.3 The Conceptual Cohesion of Classes

In order to define and compute the C3 metric, we introduce a graph based system representation, similar to those used to compute other cohesion metrics.

Definition 3.1: System, Classes — *We consider an OO system as a set of classes* $C = \{c_1, c_2 \ldots c_n\}$. *The number of classes in the system* C *is* $n = |C|$.

Definition 3.2: Methods of a Class — *A class has a set of methods. For each class* $c \in C, M(c) = \{m_1, m_2 \ldots m_z\}$ *represents its set of methods, where* $z = |M(c)|$ *is the number of methods in a class* c. *The set of all methods in the system is defined as* $M(C)$.

Definition 3.3: Graph representation of an OO system — *An OO system C is represented as a set of connected graphs $G_C = \{G_1, \ldots, G_n\}$ with G_i representing class c_i. Each class $c_i \in C$ is represented by a graph $G_i \in G_C$ such that $G_i = (V_i, E_i)$, where $V_i = M(c_i)$ is a set of vertices corresponding to the methods in class c_i and $E_i \subset V_i \times V_i$ is a set of weighted edges that connect pairs of methods from the class.*

Definition 3.4: Conceptual similarity between methods (CSM) — *For every class $c_i \in C$, all the edges in E_i are weighted. For each edge $(m_k, m_j) \in E_i$, we define the weight of that edge $CSM(m_k, m_j)$ as the conceptual similarity between the methods m_k and m_j.*

The *conceptual similarity between two methods* m_k and m_j, $CSM(m_k, m_j)$ is computed as the cosine between the vectors corresponding to m_k and m_j in the semantic space constructed by the IR method (in this case LSI):

$$CSM(m_k, m_j) = \frac{vm_k^T vm_j}{|vm_k|_2 \times |vm_j|_2} \tag{3.1}$$

where vm_k and vm_j are the vectors corresponding to the $m_k, m_j \in M(c_i)$ methods, T denotes the transpose, and $|vm_k|_2$ is the length of the vector.

For each class $c \in C$ we have a maximum of $N = C_z^2$ distinct edges between different nodes, where $z = |M(c)|$.

With this system representation, we define a set of measures that approximate the cohesion of a class in an OO software system by measuring the degree to which the methods in a class are conceptually related.

Definition 3.5: Average conceptual similarity of methods in a class (ACSM) — *The average conceptual similarity of the methods in a class $c \in C$ is:*

$$ACSM(c) = \frac{1}{N} \times \sum_{i=1}^{N} CSM(m_i, m_j) \tag{3.2}$$

where $(m_i, m_j) \in E, i \neq j, m_i, m_j \in M(c)$, and N is the number of distinct edges in G, as defined previously.

In our view, $ACSM(c)$ defines the degree to which methods of a class belong together conceptually and thus it can be used as a basis for computing the conceptual cohesion of classes.

Definition 3.6: Conceptual cohesion of a class (C3) — *For a class $c \in C$, the conceptual cohesion of c, C3(c) is defined as follows:*

$$C3(c) = \begin{cases} ACSM(c) & if \quad ACSM(c) > 0 \\ else \quad 0 \end{cases} \tag{3.3}$$

Based on the above definitions, C3(c) $\in [0, 1]$ $\forall$ $c \in C$. If a class $c \in C$ is cohesive then C3(c) should be closer to one meaning that all methods in the class are strongly related conceptually with each other (*i.e.*, the CSM for each pair of methods is close to one). In this case, the class most likely implements a single concept or a very small group of related concepts (related in the context of the software system).

If the methods inside the class have low conceptual similarity values among each other (CSM close to or less than zero), then the methods most likely participate in the implementation of different concepts and C3(c) will be close to zero.

3.4 Assessment of the New Cohesion Measure

Newly proposed metrics require empirical evaluations [52, 54]. We present the results of two case studies aimed at comparing and combining C3 with a set of existing cohesion measures. Subsections 3.4.1 and 3.4.2 describe the objectives and the design of the case studies. In subsequent subsections, quantitative results are presented and explained for each case study separately.

3.4.1 Objectives and Methodology

In order to evaluate our measure, we conducted two case studies. The goal of the first case study was to determine whether the C3 measure captures *additional dimensions* of cohesion measurement when compared to existing structural cohesion measures. Our hypothesis is that given the nature of the information and counting mechanism employed by C3, it *should* capture different aspects of class cohesion than existing structural measures.

Existing research showed that cohesion measures can be used as good indicators for the fault-proneness of classes in OO systems [90, 16, 202]. So in the second case study, C3 is compared with existing metrics and combinations of C3 with existing cohesion metrics are also compared with combinations of structural metrics (with each other) to assess whether they provide *better results in predicting faults in classes* or not. Our assumption is that combining C3 with other structural cohesion metrics *should* be a more complete indicator of cohesion (given they capture different aspects of it), hence a better indicator of fault-proneness than combinations of structural metrics alone.

In summary, the case studies are addressing the following research questions:

- **$\mathbf{RQ_{3.1}}$**: Does C3 capture aspects of class cohesion that are not captured by other structural cohesion metrics?
- **$\mathbf{RQ_{3.2}}$**: Does the combination of structural cohesion metrics with C3 provide better results in predicting faults in classes than the combinations of structural metrics?

3.4.2 Design of the Case Studies

We followed recommendations in state-of-the-art work on case studies [97, 239] to design our two studies. We used several open source systems of different sizes.

Software Systems and Metrics

We chose three open-source software systems from different domains, developed mostly in C++: TortoiseCVS v.1.8.21, WinMerge v.2.0.2, and Mozilla v.1.6. TortoiseCVS is an extension for Microsoft Windows Explorer that makes using concurrent versioning system (CVS) convenient and easy. WinMerge is a tool for visual differencing and merging for both files and directories. Mozilla is an open-source web browser ported on almost all known software and hardware platforms. It is large enough to represent a real-world software system. The source code for TortoiseCVS and WinMerge were downloaded from `http://sourceforge.net`, whereas the source code of Mozilla is obtained from `http://www.mozilla.org`.

We selected the following structural cohesion metrics to compare against C3: LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, Coh, ICH, TCC, and LCC. Our choice of metrics is not random, since these structural metrics were extensively studied and compared to each other and to other metrics in previously published studies [40, 52, 56, 63, 94, 223]. The guiding criterion that we used to choose the metrics for our case study is the availability of results reported for these metrics in the literature, in order to facilitate comparison and evaluation with our results. For the definitions, explanations, and further references on these measures, please refer to Section 3.2. We computed all these metrics for 2,151 classes from the three open-source systems.

Settings of the Case Studies

All the structural metrics are collected using Columbus [6] and the conceptual cohesion metrics are computed with our tool, IRC^{3}M. IRC^{3}M can be used with several settings for the underlying LSI-based analysis. In these case studies, we used the following ones.

For constructing the corpora, we extracted all types of methods from classes in the source code, including constructors, destructors, and accessors. Comments and identifiers were extracted from each method. The resulting text was processed as follows: some tokens are eliminated (*e.g.*, operators, special symbols, some numbers, keywords of the programming language, standard library function names, *etc.*); the identifier names in the source code were split into parts based on known coding standards. For example all the following identifiers were broken into the words “split” and “identifiers” : “split_identifiers” , “Split_identifiers” , “SplitIdentifiers” , *etc.* The original form of each identifier is preserved in the documents. Since we do not consider n-grams, the order of the words is not important. It is essential to note, though, that LSI in this process does not use a predefined vocabulary, or a predefined grammar, therefore no morphological analysis or transformations are required. Some researchers use word stemming, however this is an optional step.

Based on previous experience with LSI on similarly sized corpora [170, 198], we used a 300 factor reduction. We defined a corpus with 637 documents and 1,915 terms for TortoiseCVS, one with 522 documents and 1,738 terms for WinMerge, and one with 48,823 documents and 64,979 terms for Mozilla (which we used in both case studies).

3.4.3 First Case Study – Principal Component Analysis of the Metric Data

Briand *et al.* [56] proposed a methodology to analyze software engineering data in order to make an experiment repeatable and the results comparable. The methodology consists of the following three steps: collecting the data, identifying outliers, and performing PCA (see Section 2.5.2). As the results of our analysis can be impacted by the outliers, they were removed [56, 63]. To identify outliers in the data, we utilized the T^{2}max procedure based on the Mahalanobis distance [127].

After outliers were eliminated, we performed PCA, which was used in our case to identify groups of variables (*i.e.*, metrics), which are likely to measure the same underlying dimension (*i.e.*, mechanism that defines cohesion) of the object to be measured (*i.e.*, cohesion of a class).

Table 3.1: Loading of the PCs

	PC$_1$	**PC$_2$**	**PC$_3$**	**PC$_4$**	**PC$_5$**	**PC$_6$**
Proportion	29.6	20.91	10.12	10.04	17.0	8.56
Cumulative	29.6	50.51	60.63	70.67	87.67	96.24
C3	-0.061	-0.037	-0.017	**0.996**	-0.043	0.008
LCOM1	**0.922**	-0.001	0.052	-0.032	0.317	-0.012
LCOM2	**0.914**	-0.018	0.044	-0.029	0.331	0.004
LCOM3	**0.609**	-0.129	0.052	-0.048	**0.736**	-0.138
LCOM4	0.206	-0.196	-0.001	-0.036	**0.937**	-0.102
LCOM5	0.084	0.032	**0.995**	-0.017	0.018	-0.040
ICH	**0.914**	0.056	0.066	-0.057	-0.065	-0.144
TCC	-0.023	**0.933**	-0.033	-0.002	-0.116	0.283
LCC	0.045	**0.966**	0.079	-0.050	-0.136	0.095
Coh	-0.118	0.476	-0.061	0.012	-0.176	**0.846**

Results

PCA revealed six principal components (PC) that describe 96% of the variance in our dataset. The loadings of every PC are presented in Table 3.1, where we marked important coefficients for each PC in bold. In addition, for every PC we provide the proportion for that PC in terms of the variance of the data set, which is explained by that PC and also the cumulative variance. We interpret the loadings determined for every PC as follows:

PC$_1$ (29.6%): LCOM1, LCOM2, LCOM3, ICH. These metrics count the number of pairs of methods which share instance variables. Another commonality among LCOM1-LCOM3 is that these measures are not normalized and they do not have upper bounds.

PC$_2$ (20.91%): TCC, LCC are among the measures that are computed as the ratio of method pairs with shared instance variables, also considering indirect sharing of instance variables by method invocations. Noticeably, the measures are also normalized.

PC$_3$ (10.12%): LCOM5 is a normalized cohesion measure with upper and lower bounds. However, it is also an inverse cohesion measure that ranges between 0 (max cohesion) and 2 (min cohesion). The metric is dependent upon instance variable usage, counting the number of interactions between instance variables and methods.

PC$_4$ (10.04%): C3 is our newly proposed conceptual cohesion metric that measures cohesion of a class in the context of the complete software system based on the usage of the terms shared between pairs of methods in the class, assuming there is an underlying or latent structure in word usage for the software system for which a document set (*i.e.*, corpus) is constructed.

PC$_5$(17.0%): LCOM3, LCOM4. These metrics count common attribute usages within a class. LCOM4 additionally accounts for method invocations, which do not seriously affect the distribution of the measure.

PC$_6$(8.56%): Coh is a normalized measure that counts individual references to attributes by methods.

Answer to RQ$_{3.1}$: *The PCA results shows that C3 defines a dimension of its own; C3 is the only major factor in PC4. These results statistically support our hypothesis that the C3 cohesion measure captures different aspects of what is considered to be a* *cohesion measurement of the class, as defined by all the metrics measured in the case study.*

The PCA results reinforce the work previously done by other researchers. Chae *et al.* [63], Briand et. al. [51], and Etzkorn et. al. [94] also used PCA over different data sets using similar collections of structural cohesion metrics to those presented in this chapter. Our results are closer to those of Briand *et al.* [51] and Etzkorn *et al.* [94], as in each case, the first two principal components are the same: both studies have LCOM1-LCOM3 measures in PC_1 and TCC-LCC in PC_2.

3.4.4 Second Case Study – Predicting Faults in Classes

The first case study showed that C3 captures different aspects of cohesion compared to the structural metrics we analyzed it against. Given our interpretation of cohesion, we believe that a combination of structural and conceptual metrics is a more complete cohesion indicator than any combination of structural metrics by themselves, since the combination of structural metrics still captures only the structural properties of cohesion, whereas the combination of structural and conceptual metrics might capture orthogonal, yet complementary properties of class cohesion. In other words, structural cohesion indicates whether a class is *built* cohesively, while conceptual cohesion indicates whether a class is *written* coherently (as a function of the identifier names and comments). One use of cohesion metrics in software engineering is to predict faults in classes [90, 16, 202]. The focus here is to analyze the extent to which each of the cohesion measures selected for the case study can be used to predict faults, as well as to see whether the combination of any structural cohesion measure with C3 outperforms the combinations of structural cohesion measures in identifying fault-prone classes.

This case study is performed similarly to Gyimóthy *et al.* [16]. We used Bugzilla (`http://bugzilla.mozilla.org/`), collected the bugs between two versions of Mozilla (*i.e.*, 1.6 and 1.7), and correlated each bug with specific classes. Details on how we mined the bugs can be found in our previous work [16].

Analyses

We employed regression analysis methods to discover possible relationships between values of collected metrics and fault-proneness of those classes (see Section 2.6.1). These methods have been widely used to study the relationships between the metrics and fault- or change-proneness of classes [39, 41, 50, 56, 16, 187, 222]. In order to analyze our data, we chose univariate and multivariate logistic regression analysis methods, which predict if a class is faulty or not.

In our case, the univariate regression analysis is used to analyze the effect of each metric separately, whereas multivariate regression is used to analyze the effect of the combination of metrics on the final results to see whether combinations of C3 with structural cohesion metrics can improve detecting fault-prone classes as compared to combinations of only structural metrics alone. All analyses in this case study are applied with the same settings as those described by Gyimóthy *et al.* [16].

For logistic multivariate analysis, we build models for predicting faults in classes based on all possible combinations of pairs of cohesion metrics used in this case study (*i.e.*, 45 different pairs of metrics comprised of 10 unique cohesion metrics). We study all the resulting models based on pairs of cohesion metrics to obtain more insight into whether the models where one of the exploratory variables is C3 are superior or not to those models where both exploratory variables are structural cohesion measures. The parameters (constant C_0 and coefficients C_1 and C_2) of all instantiated models are provided in Table 3.2, Table 3.3, and Table 3.4, respectively.

Results

First, we performed the univariate logistic regression (see the results in Table 3.2). In order to evaluate logistic regression models based on the metrics we studied (as well as combinations of them), we utilize the R^2 coefficient (see Section 2.6.1), as well as *accuracy*, *precision*, and *recall* (see Section 2.6.3).

The results allow us to draw the following conclusions: if we use every one of the 10 metrics as a separate indicator of fault-proneness, LCC is the least significant, while C3 has the 2nd largest *accuracy*, 3rd largest *recall*, 5th largest R^2 value, and 6th largest *precision*. These results are not surprising, as C3 captures only certain aspects of cohesion, whereas faults may be caused by other issues affecting cohesion, which are not captured by C3 alone. Nonetheless, C3 ranks better than many of the cohesion metrics we analyzed.

Table 3.2: Results of the univariate logistic regression (sorted by the R^2 values)

Model	Accuracy	Acc. Rank	Precision	Prec. Rank	Recall	Rec. Rank	R^2 Values	C_0	C_1
LCOM1	61.90	4	74.39	2	60.95	5	0.109	-0.41	0.0012
LCOM3	62.59	1	70.55	4	64.15	4	0.107	-0.71	0.061
LCOM2	62.05	3	75.93	1	59.16	7	0.106	-0.38	0.0013
LCOM4	59.75	7	66.36	5	54.85	8	0.079	-0.60	0.0725
C3	**62.05**	**2**	**61.35**	**6**	**73.13**	**3**	**0.073**	**2.22**	**-4.11**
ICH	60.92	6	73.52	3	53.80	9	0.069	-0.30	0.008
Coh	61.21	5	59.33	7	80.18	1	0.032	0.35	-1.96
LCOM5	56.56	8	54.48	8	77.83	2	0.006	-0.38	0.481
TCC	51.81	9	50.60	9	59.61	6	0.010	0.14	-0.799
LCC	50.73	10	49.47	10	42.64	10	0.002	0.04	-0.292

Table 3.3: Results of the multivariate logistic regression for the top ten pairs of cohesion metrics with largest R^2 values (sorted by R^2 values)

Model	Accuracy	Acc. Rank	Precision	Prec. Rank	Recall	Rec. Rank	R^2 Values	C_0	C_1	C_2
C3+LCOM3	66.20	1	63.47	26	76.26	6	0.160	1.384	-3.783	0.060
C3+LCOM1	65.23	2	68.23	27	74.27	10	0.154	1.536	-3.471	0.001
C3+LCOM2	64.88	5	67.54	29	73.49	11	0.151	1.572	-3.486	0.001
C3+LCOM4	64.98	4	66.20	30	75.61	7	0.141	1.594	-4.054	0.078
C3+ICH	63.71	6	64.74	34	74.60	9	0.119	1.710	-3.597	0.006
LCOM4+ICH	63.32	9	72.87	16	65.39	15	0.119	-0.717	0.058	0.006
LCOM3+ICH	63.46	7	72.61	17	65.32	16	0.118	-0.703	0.048	0.003
LCOM1+LCOM3	63.27	10	74.16	12	64.12	21	0.116	-0.611	0.001	0.034
LCOM1+LCOM4	61.90	28	73.05	15	61.41	32	0.114	-0.553	0.001	0.030
LCOM1+Coh	62.34	21	72.44	18	64.28	18	0.113	-0.208	0.001	-0.816

Our assumption is that C3 complements existing structural metrics, so in order to investigate whether combining C3 with structural cohesion measures can improve

Table 3.4: Results of the multivariate logistic regression for the remaining pairs of metrics, which contain C3 (sorted by R^2 values)

Model	Accuracy	Acc. Rank	Precision	Prec. Rank	Recall	Rec. Rank	R^2 Values	R^2 Rank	C_0	C_1	C_2
C3+Coh	65.03	3	64.22	35	79.43	4	0.105	28	2.709	-4.220	-2.086
C3+TCC	63.41	8	62.72	38	75.38	8	0.085	29	2.507	-4.225	-0.956
C3+LCOM5	63.22	11	62.47	39	76.59	5	0.079	34	1.878	-4.129	0.503
C3+LCC	63.17	12	62.80	37	72.64	12	0.078	37	2.440	-4.230	-0.495

the detection of fault-prone classes we applied multivariate logistic regression analysis (see Table 3.3). We built 45 models based on all combinations of pairs of cohesion metrics. Table 3.3 presents the top ten models based on the largest R^2 values. As it can be seen from the results, C3 appears in the first five combined models. In order to compare the R^2 values of these combinations, we performed Wilcoxon's signed-rank test (see Section 2.5.3). The results revealed that out of those five pairs, C3 appears in three statistically significant combinations, as the R^2 values for C3+LCOM3 and C3+LCOM4 and the R^2 values for C3+LCOM1 and C3+LCOM2 respectively, are not different.

In addition, all the models that contain C3 (see Table 3.3 and Table 3.4) are among the top 12 models in terms of accuracy and recall values. This result supports the idea that C3 is indeed *complementary* to the structural metrics, at least from the perspective of fault prediction.

Also note that the three models with the best accuracy are C3 + LCOM3, C3 + LCOM1, and C3 + Coh. Any of these models has better accuracy values than the best model based on a single metric from Table 3.2. Similarly, the models with best values for precision and recall outperform the relevant models based on a single metric.

Answer to $\mathbf{RQ_{3.2}}$: *Overall, the results indicate that C3 is a useful indicator of an external property of classes in object-oriented systems – fault-proneness of classes. Based on the results of the regression analyses we can conclude that C3 is a valuable complement in a number of combinations with other structural cohesion metrics. More importantly, the results support our assumption that the combination of C3 with other cohesion metrics allows us to build superior models for detecting fault-prone classes.*

3.5 Conceptual vs. Structural Cohesion

Henderson-Sellers [117] noted that: "It is after all possible to have a class with high internal, syntactic cohesion but little semantic cohesion." To gain more insight into how our metric differs from some of the structural ones, we manually analyzed classes from Mozilla and WinMerge for which the structural and conceptual metrics disagree (*i.e.*, high structural cohesion with low conceptual cohesion and vice versa). We selected classes based on high LCOM2 values (*i.e.*, indicating low cohesion) and high C3 values, and vice versa. We considered a value for a LCOM2 and C3 high if it is in the top 15% and low if it is in the bottom 15%, respectively. Based on this criteria, we identified 25 classes in Mozilla that have low structural cohesion (based on LCOM2) and high conceptual cohesion (based on C3) and 61 classes in the opposite category. In WinMerge, we identified 4 classes that have low structural cohesion and high conceptual cohesion and 9 classes in the opposite category. We provide a few examples of such classes (see Table 3.5), although we do not claim that all such cases in the two open source software systems we analyzed follow these patterns.

Table 3.5: Classes analyzed from WinMerge and Mozilla

WinMerge class	**C3**	**LCOM2**
IVSSItem	0.64	528
IVSSDatabase	0.635	136
IVSSItemOld	0.632	465
BCMenuData	0.434	0
CDirDoc	0.294	0
RescanSuppress	0.392	1
Mozilla class	**C3**	**LCOM2**
nsXlContext	0.314	1
txFormatNumberFunctionCall	0.306	1
nsAbCardProperty	0.810	8907
nsProfile	0.650	1768
nsPrintSettings	0.656	5402

3.5.1 Analyzing classes from WinMerge

The analysis of the classes in Table 3.5 yields very interesting results. For example, the IVSSItem class is a wrapper class that does not have data members, only methods that wrap the implementation for the OLE automation on the client side. High values for LCOM2 in methods are easily explained in this case. The intersection of any pair of methods in this class is empty, because the class does not contain any attributes. For the IVSSItem class, which has 33 methods, LCOM2 = 528. The high C3 value is also understandable, as the implementation of every method contains invocations of the InvokeHelper method of the derived class COleDispatchDriver and a similar subset of identifier names for local variables. Wrappers tend to group together methods that are conceptually similar. The IVSSDatabase and IVSSItemOld classes follow the same pattern since they also implement wrappers for the COleDispatchDriver interface. In conclusion, in these situations (*i.e.*, wrappers) it seems that C3 can give more clues on cohesion than LCOM2.

From the other group of investigated classes, BCMenuData is a class that implements a "property container" for menu items that are drawn using an "Office XP"-like style. It is a small class with a set of accessor functions. LCOM2 is 0, meaning that the number of intersecting sets is more than the number of non-intersecting ones. Close examination of the class supports the fact that the class represents a single meaningful abstraction. However, values of C3 do not capture this fact due to the large number of unique identifier names used in these accessors. As mentioned above, accessor methods, just like constructors, may significantly influence the measurement of C3 and such situations warrant the use of structural metrics in support of C3.

The CDirDoc class is an example of a class with concealed cohesion, which means that the class includes some attributes and methods that might create another class. Close analysis revealed that the class handles the following activities: creating and closing of a new document, representing "right-left" panel abstraction in the "view-merge" application, keeping track of updating time, status and content, as well as choosing different view modes. It has only several attributes like a pointer to the CDirView class and a container of CMergeDoc classes. Those attributes are referenced in most methods of the class. Thus LCOM2 for the CDirDoc is 0. On the other hand, the value of C3 for CDirDoc is 0.294 which shows low conceptual similarity of methods inside the class. Detailed analysis shows that the class implements a set of concepts that could be refactored into separate classes, each implementing one concept only. The low LCOM2 value would indicate a difficult refactoring since it may create high coupling. However, when considering the low number of attributes of the class, this may not be a major issue. While it is hard to generalize, in situations when a class has few attributes and many methods by comparison, a low LCOM2 value and a low C3 value may indicate the lack of cohesion and a need for refactoring.

The RescanSuppress class implements an abstraction of a simple lock that prevents objects of type CMergeDocs from rescanning within its lifetime unless the `clear()` method is called. It is a small class with three attributes and three methods – a constructor, a destructor, and the `clear()` method. Although the class represents a crisp abstraction from a user point of view, the small value for C3 can be explained by the small number of identifiers and their intersections within method implementations of the class. This is a situation where C3 showed its limits and structural metrics are needed.

3.5.2 Analyzing classes from Mozilla

We applied the same strategy to analyze several more classes from Mozilla (see Table 3.5). In this case, the nsXIContext class aggregates dialogs like "license" , "welcome" and widgets like "next" and "previous" buttons, which appear during the installation process. Therefore, the main purpose of this class is to store the user interface components during the installation process. The class has a set of methods to load, get, and release resources. It also contains a proprietary implementation of the *itoa* function, which in fact decreases the conceptual cohesion of the class, since it has low conceptual similarities with other methods in this class (*itoa* performs a very specific operation – conversion from integer to string, which does not relate conceptually to loading and releasing resource operations implemented in the class). Another class from the group of classes with low conceptual and high structural cohesion is txFormatNumberFunctionCall. This is a class that derives from an abstract class FunctionCall, which in turn also derives from an abstract class, namely Expr. This is a base class of XSL (e**X**tensible **S**tylesheet **L**anguage) expressions and its `evaluate()` method is responsible for evaluating and formatting numbers in XSL transformations. After analyzing the `evaluate()` method, we concluded that it has two parts: parsing the format of the string, and the actual formatting of the number. Thus, we could refactor the first part into another class (*e.g.*, txFormatParseState), which would be responsible for parsing the string, whereas txFormatNumberFunctionCall would perform only the actual transformation of numbers in XSL expressions. As with the class CDirDoc from WinMerge, this class is another example of a class with concealed cohesion.

On the other hand, we analyzed three classes with high conceptual and low structural cohesion. The first class in this group is nsAbCardProperty, which implements a well-defined concept – "Address Book Person Card Entry". It has over fifty different attributes that can occur in the address book (*e.g.*, m_PhoneticFirstName, m_DisplayName, m_DefaultEmail, *etc.*) and it has many accessor methods. Thus, the low structural cohesion can be easily explained because of the presence of these accessor methods, which usually reference one attribute at a time. Since all these methods are small and share many similar terms (*e.g.*, *card, property, set, PRUnichar, attribute, name, etc.*), on average, all pairs of methods have high conceptual similarities. We concluded that this class is one implementing a cohesive concept – storing and accessing the information about the user in the address book entry.

Another class, nsProfile, implements the concept of a profile in the Mozilla web browser and it is derived from two interfaces: nsIProfileInternal and nsIProfileChangeStatus. It is rather large, consisting of over a dozen methods and attributes, implemented in over 2 KLOC. Some of the methods with self-descriptive names are: LoadDefaultProfileDir, LoadNewProfilePrefs, MigrateProfileInternals, Update4xProfileInfo, *etc.* After inspecting all the methods we identified that they can be classified into two groups: operations on Profile and on Registry. This classification explains the high values of LCOM2, since these two groups of methods reference non-overlapping attributes. Overall, we conclude that the nsProfile class implements a single concept even though its operations may be categorized into two related groups (the second group of Registry operations relates to Profile operations in the sense that those methods are tailored towards reading and writing profile-related keys in the system registry).

The class nsPrintSettings describes the print settings for a document: print range, colors, paper size, orientation, *etc.* This class aggregates a lot of attributes which describe these properties. In addition, all these attributes have accessor methods. This class looks like a property container, since it does not have any operations on these attributes, which can be broadly classified into two groups: printer attributes (mPrintBGColors, mPrintBGImages, mPrintPreview) and paper attributes (mPaperName, mPaperSizeUnit). Low structural cohesion is explained by the number of accessor methods referencing unique attributes. However, conceptually the class implements a single concept – a property holder of print settings, which is supported by the high C3 values.

3.6 Threats to Validity

Several issues affect the results of the case studies and limit our interpretations and generalizations of the results. The first case study showed that our metric captures new dimensions in cohesion measurement; however, we obtained these results by analyzing classes from only three open-source applications written primarily in C++, even though one of the systems (*i.e.*, Mozilla) represents a real-life application. In order to generalize the results, large-scale evaluation, similar to the one presented by Succi *et al.* [223] is required, which should take into account software systems from different domains, written in different programming languages, and of varying class sizes [89]. In the case study, we compare our measures with existing structural cohesion measures, which could be computed with available tools. The results could be somewhat different if we considered semantic metrics or those based on information-theory approaches.

One issue may affect the internal validity of the second case study: cohesion is not the only factor affecting the fault-proneness of classes. To build complete models for fault prediction, other factors would have to be considered. However, this is out of the scope of this chapter as the purpose of analyzing fault prediction was to see if the combination of C3 with structural metrics brings any improvements (which it did).

The C3 metric depends upon reasonable naming conventions for identifiers and relevant comments contained in the source code. When these are missing, the only hope for measuring any aspects of cohesion rests on the structural metrics. In addition, methods such as constructors, destructors, and accessors may artificially increase or decrease the cohesion of a class [52]. While we did not exclude them in the results presented here, our method may be extended to exclude them from the computation of the cohesion using approaches for identifying types of method stereotypes [84]. In its current form, C3 does not take either polymorphism or inheritance into account, only considering methods of a class that are implemented or overloaded within the class.

3.7 Conclusions

Classes in object-oriented systems, written in different programming languages, contain identifiers and comments, which reflect concepts from the domain of the software system. This information can be used to measure the cohesion of software. To extract this information for cohesion measurement, Latent Semantic Indexing can be used in a manner similar to measuring the coherence of natural language texts.

This chapter defines the conceptual cohesion of classes, which captures new and complementary dimensions of cohesion compared to a host of existing structural metrics. Principal component analysis of measurement results on three open-source software systems statistically supports this fact. In addition, the combination of structural and conceptual cohesion metrics defines better models for prediction of faults in classes than combinations of structural metrics alone. Highly cohesive classes need to have a design that ensures a strong coupling among its methods as well as a coherent internal description.

Contribution

This chapter is based on the publication:

• Andrian Marcus, Denys Poshyvanyk, and **Rudolf Ferenc**. *Using the conceptual cohesion of classes for fault prediction in object oriented systems.* IEEE Transactions on Software Engineering, 34(2):287–300, March 2008. [22]

Defining and calculating the C3 metric was done by my co-authors, while the design, management, and evaluation of both case studies (including whether C3 can improve upon existing structural metrics in fault prediction) was my responsibility. Moreover, the source code analysis and computation of structural cohesion metrics for TortoiseCVS, WinMerge, and Mozilla was my work as well.

Some of my other notable papers that have contributed to this result:

• **Rudolf Ferenc**, István Siket, and Tibor Gyimóthy. *Extracting facts from open source software.* In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pages 60–69, Chicago, USA, September 2004. [10]

• Tibor Gyimóthy, **Rudolf Ferenc**, and István Siket. *Empirical validation of object-oriented metrics on open source software for fault prediction.* IEEE Transactions on Software Engineering, 31(10):897–910, November 2005. [16]

ferenc.rudolf_87_23

4 Conceptual Coupling in Impact Analysis

4.1 Introduction

During program comprehension, developers need to understand how software modules relate to each other. It is especially important when changes are being made to the software and developers need to assess the impact of their changes. One way to understand such relationships is to measure the coupling between parts of the software. Coupling, along with cohesion, is one of the fundamental properties of software with a strong influence on comprehension and maintenance of large software systems. Proposed coupling measures are used in software engineering tasks, such as change impact analysis [55, 232], assessing the fault-proneness of classes [90, 240, 16, 187], software re-modularization [32, 236], identifying software components [147] and design patterns [37], assessing software quality [56], *etc.*

Depending on the programming paradigm used, the choice of programming language for the implementation, and the design of a software system, coupling is influenced by several factors – such as control and data flow – and hence it may be measured differently. Researchers proposed a variety of coupling measures, but some studies [56] suggest that some of these metrics tend to compute the same form of coupling, only through different measuring mechanisms.

In Chapter 3, we defined a set of *cohesion* measures; in this chapter, we similarly define a set of *coupling* measures that capture new dimensions of coupling, based on the textual information shared between modules of the source code. While elements of the source code written in a programming language help identify control or data flow between software modules, the comments and identifiers express the intent of the software. Two parts of the software with similar intent will most likely refer to the same (or related) concepts in the problem or solution domains of the system. Hence, they are conceptually related. This is the same idea we used in Chapter 3, however now applied to coupling instead of cohesion. Conceptual relations have been also confirmed by earlier work of other researchers, who examined overlap of semantic information in comments and identifiers among different software modules [91, 220]. This relationship is the foundation for the new coupling measures, named *conceptual coupling.* The set of conceptual coupling metrics can be defined and used for any type of programming paradigm, but we define and use them here in the context of OO software systems.

Existing coupling measures have been previously used to support the impact analysis process, where the task is to identify all classes that would change when a given class is being changed. However, existing models [55] do not capture all the ripple effects of changes in existing software. Given that the conceptual coupling metrics reflect different relationships than structural coupling metrics, we assume that they also propagate changes in software. This chapter focuses on the use of the conceptual coupling metrics to predict classes that will change during impact analysis. We conducted a case study on a large open-source software system (Mozilla) to see how the conceptual coupling metrics compare with nine existing structural coupling metrics, when used during impact analysis. The case study indicates that one of our conceptual coupling metrics provides the best results for predicting classes that need to be changed.

Our contributions can be listed as follows:

- Formal definition and calculation of the conceptual coupling metrics.
- Examining the conceptual metrics in impact analysis through a case study and comparing them to the traditional structural metrics.

4.2 Related Work

We briefly discuss the major approaches to coupling measurement, in order to contrast between existing approaches and our proposed metrics. The conceptual coupling metrics are based on the use of IR methods and constitute a novel application, compared to previous uses of IR in program comprehension, which we also present here. Coupling measures have been used to support impact analysis and we present those approaches here as well.

4.2.1 Coupling Measurement

Coupling measurement is a rich and interesting body of research, resulting in different measuring approaches for structural coupling metrics [66, 67, 148, 49], dynamic coupling measures [39, 114], evolutionary and logical coupling [100], coupling measures based on information entropy [34], coupling metrics for specific types of software applications such as procedural systems [186], knowledge-based systems [138], ontology-based systems [189] and systems developed using an aspect-oriented approach [242].

The structural coupling metrics have received significant attention in the literature. These metrics are comprehensively described and classified within the unified framework for coupling measurement [53]. The best known among these metrics are CBO (coupling between objects) and CBO' [66, 67], RFC (response for class) [66] and RFC_∞ [67], MPC (message passing coupling) [150], DAC (data abstraction coupling) and DAC^1 [150], ICP (information-flow-based coupling) [148], and the suite of coupling measures by Briand *et al.* [49] (IFCAIC, ACAIC, OCAIC, FCAEC, *etc.*). Other structural metrics such as C_e (efferent coupling), C_a (afferent coupling), and COF (coupling factor) are also overviewed by Briand *et al.* [53].

Many of the coupling measures listed above are based on method invocations and attribute references. For example, the RFC, MPC, and ICP measures are based on method invocations only. CBO and COF measures count method invocations and references to both methods and attributes. The suite of measures defined by Briand *et al.* [49] captures several types of interactions between classes such as class-attribute, class-method, and method-method interactions. The measures from the suite also differentiate between import and export coupling as well as other types of relationships including friends, ancestors, descendants, *etc.*

Dynamic coupling measures [39, 114] were introduced as the refinement to existing coupling measures due to some gaps in addressing polymorphism, dynamic binding, and the presence of unused code by static structural coupling measures.

Another important family of coupling measures derives from the evolution of the software system in contrast to structural coupling which is determined by program analysis of a single version of software, or dynamic coupling which is obtained by executing the program. These are called evolutionary couplings among parts of the systems which are determined by the past common changes or co-changes [100].

Another form of coupling, namely interaction coupling, captures relations among software artifacts which are relevant to a particular software engineering task [248]. Interaction coupling uses information gleaned using an Integrated Development Environment on when artifacts are being used or modified in the same development task.

Furthermore, several specialized coupling metrics were proposed for different types of software systems. They are coupling metrics for knowledge-based systems [138] as well as coupling metrics for aspect-oriented programs [242].

Existing work on clustering software [162, 141], retrieving similar components in software libraries [176] and measuring semantic overlap of information in comments and identifiers among software modules [91] uses the concept of semantic similarity between elements of the source code [22], which stands at the foundation of the conceptual coupling, as defined in this chapter.

4.2.2 The Use of IR Methods in Program Comprehension

IR methods were proposed, and used successfully, to address tasks of extracting and analyzing textual information existing in software artifacts. Early models were used to construct software libraries [159, 95] and support reuse tasks [116, 92, 176, 193, 238], while more recent work focused on specific software maintenance and development tasks such as recovery of traceability links. Several approaches have been proposed to recover traceability links between source code and external documentation using probabilistic IR, vector space models [36] and LSI [170]. Other work proposed a set of approaches to recover traceability links among requirements [70, 155], requirements and source code [115], requirements and test cases [157], *etc.* A set of tools that integrates facilities to manage traceability links among different types of software artifacts was developed and evaluated as well [78].

IR methods have also been successfully used for concept and feature location [172, 243, 200, 198, 87] in the source code. Other approaches use IR methods to classify software systems based on their source code in open-source repositories [133] as well as to cluster source code to obtain high-level views of software systems [162, 141].

IR techniques were also used to identify the starting impact set of a maintenance request [35], and to link change request descriptions to the set of historical file revisions impacted by similar past change requests [57]. An approach to automatically classify the type of maintenance activity based on a textual description of changes was also proposed [179]. IR approaches have been used in the context of software measurement to assess the quality of identifiers and comments [146], measure the complexity of the underlying software [93], or compute the conceptual cohesion [195, 171] and coupling [199] of classes.

In addition, IR techniques have been applied to several other tasks, such as identification of duplicate bug reports [211, 228], classification of software maintenance requests [78], recommendation rendering for novice programmers [73] and identifying developer contributions [154].

4.2.3 Impact Analysis Approaches

During software change, programmers need to modify the source code of existing software systems. The first step during software change is to identify a part of the source code that needs to be changed. Once the starting point of the change is identified, developers need to (recursively) identify all the dependent components that need to be changed as well. Bohner *et al.* [47] recognized impact analysis as an activity that estimates all components to be changed. One of the techniques of impact analysis was proposed in the work of Queille *et al.* [203], where an interactive process was suggested, in which the programmer, guided by dependencies among program components (*i.e.*, classes, functions), inspects components one-by-one and identifies the ones that are going to change – this process involves both searching and browsing activities. This interactive process was supported via a formal model, based on graph rewriting rules [64].

More work appears in [48, 208, 120], where the tools they propose can help navigate and prioritize system dependencies during various software maintenance tasks. The work in [120] relates to our approach in that it also uses lexical (textual) clues from the source code to identify related methods. Several papers presented algorithms that estimate the impact of a change on tests [210, 137]. A comparison of different impact analysis algorithms was provided in [190].

Coupling measures have been used to support impact analysis in OO systems [55, 232]. Wilkie and Kitchenham [232] investigated if classes with high CBO coupling metric values are more likely to be affected by change ripple effects. Although CBO was found to be an indicator of change-proneness in general, it was not sufficient to account for all possible changes. The work of Briand *et al.* [55] investigated the use of coupling measures and derived decision models for identifying classes likely to be changed during impact analysis. The results of empirical investigation of the structural coupling measures and their combinations showed that the coupling measures can be used to focus underlying dependency analysis and reduce impact analysis effort. On the other hand, the study revealed a substantial number of ripple effects, which are not accounted for by the highly (structurally) coupled classes. This work motivated our quest for novel coupling measures, which use alternative sources of information (*i.e.*, text in identifiers and comments) to capture dependencies that are not captured by the existing structural coupling measures.

4.3 Using IR Methods for Coupling Measurement

Our approach to coupling measurement is based on the hypothesis that modules (or classes) in (OO) software systems are related in more than one way. The evident and most explored set of relationships is based on data and control dependencies. In addition to such relationships, classes are also related conceptually, as they may contribute to the implementation of the same domain concept. In this chapter, we propose a mechanism, based on IR techniques, to capture and measure this form of coupling, called *conceptual coupling*. Our choice of IR technique in this type of application is LSI, as it was in Chapter 3 as well.

In order to compute the conceptual coupling of classes, the source code of the software system is converted into a text corpus, where each document contains elements of the implementations of a method. Comments and identifiers are extracted from the source code, as well as structural information. The user has an option to choose the desired granularity (*e.g.*, class or method level) for documents (see more details in Section 4.5.1). LSI uses this corpus to create a term-by-document matrix, which captures the distribution of words in the methods. From here, the main idea is very similar to what we described in Chapter 3.

The definition of – and the methodology for measuring – the conceptual coupling would not change radically if another IR method was used. The only significant change would be in the definition of the conceptual coupling between methods, which we will discuss in the next section).

4.3.1 System Representation and Coupling Measures

In order to define and compute the conceptual coupling measures, we will use the graph based representation of a software system we presented in Section 3.3.3. The classes and the methods of a system are defined in definitions 3.1 and 3.2, respectively.

Conceptual coupling between methods can be interpreted as the conceptual similarity between the methods, thus Definition 3.4 can be (re)used to express the conceptual coupling between two methods.

As defined, the value of $CSM(m_k, m_j) \in [-1, 1]$, as CSM is a cosine in the LSI space. In order to comply with the non-negativity property of coupling metrics [53], we refine CSM as:

$$CSM^1(m_k, m_j) = \begin{cases} CSM(m_k, m_j) \quad if \quad CSM(m_k, m_j) \geq 0 \\ else \quad 0 \end{cases} \tag{4.1}$$

<u>Definition 4.1</u>: Conceptual Coupling between a Method and a Class (CCMC) — *Let $c_k \in C$ and $c_j \in C$ be two distinct ($c_k \neq c_j$) classes in the system. Each class has a set of methods $M(c_k) = \{m_{k1}, m_{k2} \dots m_{kr}\}$, where $r = |M(c_k)|$ and $M(c_j) = \{m_{j1}, m_{j2} \dots m_{jt}\}$, where $t = |M(c_j)|$. Between every pair of methods (m_k, m_j) there is a conceptual coupling measure – $CSM(m_k, m_j)$. We define the conceptual coupling between a method m_k and a class c_j as follows:*

$$CCMC(m_k, c_j) = \frac{\sum_{q=1}^{t} CSM^1(m_k, m_{jq})}{t}, \tag{4.2}$$

which is the average of the conceptual couplings between method m_k and all the methods from class c_j.

<u>Definition 4.2</u>: Conceptual Coupling between two Classes (CCBC) — *We define the conceptual coupling between two classes $c_k \in C$ and $c_j \in C$ as:*

$$CCBC(c_k, c_j) = \frac{\sum\limits_{l=1}^{r} CCMC(m_{kl}, c_j)}{r}, \tag{4.3}$$

which is the average of the couplings between all unordered pairs of methods from class c_k and class c_j ($c_k \neq c_j$). The definition ensures that the conceptual coupling between two classes is symmetrical, as $CCBC(c_k, c_j) = CCBC(c_j, c_k)$.

4.3.2 The Conceptual Coupling of a Class

With this system representation, we define a measure that approximates the coupling of a class in an OO software system by measuring the degree to which the methods of the class are conceptually related to the methods of the other classes.

Definition 4.3: Conceptual Coupling of a Class (CoCC) — *For a class $c \in C$, conceptual coupling is defined as:*

$$CoCC(c) = \frac{\sum\limits_{i=1}^{n} CCBC(c, d_i)}{n-1}, \tag{4.4}$$

where $n = |C|, d_i \in C$, and $c \neq d_i$.

If a class $c \in C$ is strongly coupled to the rest of the classes in the system, then $CoCC(c)$ should be closer to one meaning that the methods in the class are strongly related conceptually with the methods of the other classes. In this case, the class most likely implements concepts that overlap with concepts implemented in other classes (which are related in the context of the software system).

If the methods of the class have low conceptual coupling values with methods of other classes, then the class implements one or more concepts with limited interaction with the rest of the system. The value of CoCC(c) in this case will be close to zero.

In this form, CoCC does not make any distinction between method types. If needed, CoCC can be altered to account for overloaded, friend, and other method stereotypes, as discussed in [49].

4.3.3 The maximum conceptual coupling of a class

If a class $c \in C$ has a high CoCC value, one can easily infer that it is strongly related to most other classes in the system. The opposite conclusion can be inferred if the CoCC value is low. Little can be said, however, if the CoCC value is neither high nor low. It is a general drawback of average based metrics. In these cases, we can still have classes strongly related to c, which are important from a program comprehension point of view. These strong relationships can also propagate changes between classes.

An analogous logic can be applied to the coupling between two classes (*e.g.*, if two methods from different classes are conceptually similar, they might need to be changed in concert).

With that in mind, we refine CoCC to capture only the strongest couplings among methods. The goal here is to make sure that our measuring mechanism does not miss classes that are highly coupled even to a part of the system, as developers need to be aware of such classes. Thus, we define:

$$CCMC_m(m_k, c_j) = max\left\{CSM^1(m_k, m_{jt}), \forall t = 1..|M(c_j)|\right\} \tag{4.5}$$

The *maximum conceptual coupling* between method m_{kj} and class c_j is denoted by the highest conceptual coupling among all possible pairs of methods between method m_k and all the methods in class c_j.

The maximum conceptual coupling between two classes based on $CCMC_m$ is defined as the following:

$$CCBC_m(c_k, c_j) = \frac{\sum_{l=1}^{r} CCMC_m(m_{kl}, c_j)}{r} \tag{4.6}$$

The *maximum conceptual coupling metric* $CoCC_m$ for a class c, is then defined as:

$$CoCC_m(c) = \frac{\sum_{i=1}^{n} CCBC_m(c, d_i)}{n-1}, \tag{4.7}$$

where $n = |C|, d_i \in C, c \neq d_i$.

4.4 Using Coupling Measures for Impact Analysis

The coupling measures can help order (rank) classes in software systems, based on different types of dependencies among classes, captured by the coupling measures [55]. Such coupling measures and derived ranks of classes can be computed automatically. The next section describes probabilistic decision models based on coupling measurement to support impact analysis.

4.4.1 Ranking Classes Using Coupling Measures

For a given class $c \in C$ (which may be the starting point of a change, identified by the programmer based on his experience, or automatically with some feature location technique), the other classes in a software system are ranked according to their strength of coupling to the class c, based on a coupling measure or a combination of such measures [55]. The list of ranked classes is provided to the developer for further inspection. Since software systems may be large, sometimes containing thousands of classes, focusing impact analysis on strongly coupled classes may significantly reduce the burden on the developer.

In Section 4.2.1 we summarized the best known structural coupling measures. In the literature, these coupling measures are defined and used at the system level (classic definitions of coupling measures), meaning that they count, for a given class c, all dependencies (connections) from c to ***all*** other classes in the system. In order to use the coupling measures for impact analysis, they need to be modified to account for coupling between pairs of classes only. Table 4.1 shows how we redefined some of the structural coupling measures. More details on how other structural coupling measures are redefined on a class pair-wise basis are provided by Briend *et al.* [55]. Section 4.3.1 provides details on how we defined conceptual coupling measures on a pair-wise basis.

Table 4.1: Examples of redefined structural coupling measures used to rank classes during impact analysis

Name of the measure	Definition
CBO (coupling between object classes)	Two classes $c_i \in C$ and $d_i \in C$ are coupled to one another, if methods of one class use methods or attributes of the other, or vice versa. CBO is computed as a binary indicator, yielding 1 if c_i and d_i are coupled, else 0.
ICP (information-flow based coupling)	The number of method invocations in a class $c_i \in C$, of methods in a class $d_i \in C$, weighted by the number of parameters of the invoked methods. The measure also takes polymorphism into account.
DAC (data abstraction coupling)	The number of attributes in a class $c_i \in C$ that has class $d_i \in C$ as their type.

4.4.2 An Example of Using Coupling Measures for Impact Analysis in Mozilla

The following example illustrates how conceptual and structural coupling metrics are used to rank classes to focus impact analysis. The bug #232570[1] reports some problems associated with "ldap2.server.position values for ab pane and search order" in Mozilla. In order to fix the bug, the developer needs to find and change classes in the source code containing this bug. Assume that the starting point of this change, the class *nsAbDirectoryQuery*, is identified via some available feature location technique. Given the starting point, the developer needs to perform impact analysis to identify the remaining classes in order to complete the change. In our approach, we compute the set of pair-wise coupling measures for all possible pairs between *nsAbDirectoryQuery* and other classes. Using these coupling measures, all the classes in Mozilla are ranked based on the strength of coupling (different type of couplings are captured by different measures) to the *nsAbDirectoryQuery* class. The idea is that the strongly coupled classes to the given class are more likely to change [55]. In our example, Table 4.2 provides the list of top classes ranked by the values of two coupling metrics, $\mathrm{CCBC_m}$ and ICP. These measures provide the quantitative estimation of the strength of coupling between the class *nsAbDirectoryQuery* and the classes in Table 4.2. In order to determine the number of candidate classes suggested for inspection during impact analysis, different strategies can be used. The most common approaches are to use a cut point *cp* (*i.e.*, select the top n classes from the list or the top n%) or a threshold t (*i.e.*, select all classes that have a coupling value higher/lower than some metric value t). Combinations of the two approaches are also used. For example the top n% classes will be retrieved if they have a coupling value higher or lower than t.

In this example, for each metric, a cut point strategy is used (*e.g.*, the top five classes from each rank list are retrieved).

While using $\mathrm{CCBC_m}$ for ranking conceptually similar classes to the *nsAbDirectoryQuery* class, we retrieve five out of 4,853 (see Table 4.2). Two of these classes, *nsAbMDBDirectory* and *nsAbLDAPDirectory*, are among those ten classes in the official patch that were changed to fix this bug (*nsAbAutoCompleteSession, nsAbBSDi-*

[1]The bug can be accessed in Bugzilla at `https://bugzilla.mozilla.org/show_bug.cgi?id=232570` (verified at 06/04/2023)

Table 4.2: Classes strongly coupled with nsAbDirectoryQuery and ranked according to CCBC_m and ICP coupling measures

Rank	CCBC_m		ICP	
	Classes	Values	Classes	Values
1	nsAbQueryLDAPMessageListener	0.86	nsDebug	123
2	*nsAbMDBDirectory*	0.81	nsAFlatString	121
3	nsAbDirectoryQuerySimpleBoolExpression	0.79		
4	*nsAbLDAPDirectory*	0.76		
5	nsAbView	0.72		

rectory, nsAbCardProperty, nsAbDirProperty, nsAbDirectoryDataSource, nsAbDirectoryProperties, nsAbDirectoryQuery, nsAbLDAPDirectory, nsAbMDBDirectory, nsMsgCompose). However, when the ICP metric is used with this cut point, only two classes are suggested and none of them is among the classes that changed. The precision and recall for these two metrics is computed as the following. Precision for CCBC_m is $2/5 * 100 = 40\%$, while recall is $2/9 * 100 = 22\%$ (we use nine classes instead of ten in the denominator, since one of the classes that changed – *nsAbDirectoryQuery* – is already identified and used as a starting point for the impact analysis). None of the classes that changed have any structural dependencies on *nsAbDirectoryQuery* that could have been captured by the ICP measure, though, so the precision and recall for ICP is zero.

4.5 A Case Study on Using Coupling Measures to Support Impact Analysis

In this section, we present a case study where we empirically investigated how conceptual coupling metrics can be used during impact analysis, as well as compared them to a set of existing structural coupling measures used for the same task.

4.5.1 Design of the Case Study

The case study is designed in a similar fashion to the one presented in the work by Briand *et al.* [55], where a set of structural coupling metrics was used to rank classes during impact analysis in an OO system. While designing and conducting the case study, we followed the guidelines from two papers written by Yin and by Flyvbjerg, respectively [239, 97].

Objectives and Methodology

In this case study, the CCBC and CCBC_m measures are compared to nine existing structural coupling measures (*i.e.*, PIM, ICP, CBO, MPC, OCMIC, DAC, OCAIC, ACMIC, and ACAIC) to evaluate whether they provide better support for impact analysis or not. The premise is that given the nature of the information captured (*e.g.*, textual information in identifiers and comments) and counting mechanism employed by CCBC and CCBC_m, these measures should capture different aspects of coupling among classes as compared to the nine existing coupling metrics, which utilize structural information only.

In the case study, we used the source code of Mozilla v1.6 – as we did in Chapter 3 – since it is large enough (4,853 classes, approximately four million lines of source code including 738,180 lines of comments) to represent a real-world software system, and it also comes with an available history of changes.
Our case study addresses the following question:

$\mathbf{RQ_{4.1}}$: *Do CCBC and $CCBC_m$ provide better support for ranking classes during impact analysis than any of the following structural coupling measures: PIM, ICP, CBO, MPC, OCMIC, DAC, OCAIC, ACMIC and ACAIC?*

Settings of the Case Study

The setting of the case study is very similar to the one we used in Section 3.4.2. Here, we only list the differences.

All the structural coupling measures, including pair-wise versions of coupling measures, were computed using *Columbus* [6]. The conceptual coupling measures were computed with the IRC^{2}M tool [199], which can be used with several settings for the underlying LSI-based analysis. We used a reduction factor of 500 for the Mozilla software system corpus.

Collecting Change Data in Mozilla for Evaluation

In order to compare conceptual and structural coupling measures for identifying classes that change together (*i.e.*, changes related to the same bug report and having the same identification number in the configuration management system) during impact analysis, we utilized the history of changes in Mozilla. We used Bugzilla[2], a bug-tracking system used in the development of Mozilla, to collect the bugs between two specific versions (namely, 1.6 and 1.7), and correlated each bug with specific classes.

Although the Bugzilla database contained around 256,613 different bug entries for all the versions of the system at the time of this experiment, we restricted the scope of mining only to the bugs that appear between versions 1.6 and 1.7, and that were fixed (meaning that the bug was officially closed and contained an official patch file with modifications). In our analysis, we did not consider bug reports for accessory software systems such as Bonsai, Tinderbox, *etc.* We extracted 1,021 different bug entries that satisfied all the aforementioned requirements. By analyzing the patch files, associated with bug reports, we assigned bugs to particular intervals in the source code. This could be completed automatically, since each patch file contained the name of the file that changed, and it described how many lines were deleted from a given line number and how many lines were inserted at a given line number. Using this line-level change information, we determined the intervals of actual changes in the file and localized the bugs to implementations of specific classes in the source code. To ensure that the files in the patch were changed at the same time, we searched and checked log messages in the configuration management system to ensure that check-in messages for those files contained the same identification number, assigned by the Bugzilla system.

After collecting a set of bug reports and sets of changed classes respectively, we filtered the data to eliminate the bugs that contained only one modified class. After

[2]`http://bugzilla.mozilla.org/`

the filtering, we ended up with 391 bug reports, containing on average 7.3 modified classes (standard deviation: 14.6). In addition to this, we also removed some outliers in the data. For example, one such outlier was the patch in bug report #226439[3], which contained a record number of 149 modified classes to fix a single bug.

Evaluation Methodology

Our evaluation strategy was to utilize the history of changes observed in Mozilla to determine whether existing structural and conceptual coupling measures can be used during impact analysis to identify classes with common changes (*i.e.*, changes in classes related to the same bug report and having the same bug identification number in the configuration management system). The history of changes can be used to evaluate rankings of classes produced with different coupling measures against actual changes in the software system. We *expected* that the conceptual coupling measures, namely CCBC and CCBC_m, would be at least as effective as the nine existing coupling measures in ranking classes during impact analysis.

The evaluation methodology is summarized in the following steps:

- For a given software system, a set of bug reports $B = \{b_1, b_2 \ldots b_n\}$ is mined from the bug tracking system. The set of classes, which has been changed to fix each bug (*e.g.*, $c(b_1) = \{c_1, c_2 \ldots c_n\}$) are mined from the configuration management system.
- For each class in $c(b_i)$, pair-wise structural and conceptual coupling metrics are computed. The values of each metric are used to compute ranks of the remaining classes in the software system.
- Using a specific cut point criteria (which ranges anywhere from 10 to 500 classes), defined as *cp*, select top n classes in each ranked list of results generated by every metric. For every class in $c(b_i)$, which is used in the evaluation, we assess the effectiveness of identifying relevant classes (*i.e.*, the other classes in $c(b_i)$) via rankings of specific coupling metric.
- In order to evaluate each coupling measure and compare all the coupling measures used in the case study, the suggested ranked lists of classes are compared against classes that were actually changed. Average precision (P), recall (R), and F-measure (F) for each class in $c(b_i)$ for each $i = 1..|B|$ is computed for every metric (see Section 2.6.3).

4.5.2 Comparing Conceptual and Structural Coupling Metrics for Impact Analysis

In order to compare the coupling measures, we followed the evaluation methodology presented in Section 4.5.1. We computed precision and recall values for each coupling metric for every class in each of the 391 bug reports. In total, we computed 1,490 precision and recall values for eleven coupling measures. In addition to this, we studied the impact of different cut points on precision, recall, and F-measure values for particular coupling measures.

Our results are presented in tables 4.3, 4.4, and 4.5, containing precision and recall values for CCBC_m, ICP, PIM, CCBC, CBO, MPC, OCMIC, OCAIC, DAC, ACMIC,

[3]The bug can be accessed in Bugzilla at `https://bugzilla.mozilla.org/show_bug.cgi?id=226439` (verified at 06/04/2023)

and ACAIC – with different cut points ranging from 10 classes to 500 classes. The values of pair-wise conceptual and structural coupling measures taken at each cut point are provided in the Threshold (Thr) column.

Table 4.3: Impact analysis results – 10 to 50 classes

	Cut point=10			Cut point=20			Cut point=30			Cut point=40			Cut point=50		
	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr
$CCBC_m$	27.8	14.6	0.64	24.7	22.1	0.61	22.4	27.7	0.6	20.3	31.7	0.59	18.36	34.5	0.59
ICP	11.9	6.9	268	10.1	9.7	268	9.3	12	268	8.8	14.2	268	8.6	16.5	268
PIM	11.3	6.6	138	9.84	9.56	138	9.13	11.7	138	8.66	13.8	138	8.52	16.3	138
CCBC	10.8	5.6	0.47	9.5	8.9	0.45	8.1	11.1	0.45	7.2	12.8	0.44	6.7	14.1	0.44
CBO	7.2	6.2	10	5.4	9.4	20	4.1	10.4	30	3.3	10.9	40	2.8	11.3	50
MPC	6.6	5.7	3	3.9	6.7	0	2.8	7.0	0	2.1	7.0	0	1.7	7.0	0
OCMIC	2.0	2.1	2	1.1	2.2	0	0.8	2.3	0	0.6	2.3	0	0.5	2.3	0
OCAIC	1.7	2.0	0	1.0	2.1	0	0.6	2.1	0	0.5	2.1	0	0.4	2.1	0
DAC	1.8	2.0	0	1.0	2.1	0	0.6	2.1	0	0.5	2.1	0	0.4	2.1	0
ACMIC	0.9	0.4	0	0.5	0.4	0	0.3	0.4	0	0.2	0.4	0	0.2	0.4	0
ACAIC	0.8	0.3	0	0.4	0.3	0	0.3	0.3	0	0.2	0.3	0	0.2	0.3	0

Table 4.4: Impact analysis results – 60 to 100 classes

	Cut point=60			Cut point=70			Cut point=80			Cut point=90			Cut point=100		
	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr
$CCBC_m$	16.6	36.5	0.58	15.3	38.6	0.58	14.2	40.2	0.57	13.4	41.8	0.57	12.62	43.1	0.57
ICP	8.5	18.8	268	7.9	20.1	157	7.3	20.9	69	6.8	21.8	29	6.5	22.8	22
PIM	8.43	18.7	138	7.82	19.9	69	7.23	20.6	30	6.81	21.6	13	6.52	22.6	11
CCBC	6.3	15.5	0.43	5.9	16.5	0.43	5.6	17.7	0.43	5.4	18.8	0.43	5.2	19.8	0.42
CBO	2.4	11.6	60	2.1	11.7	70	1.9	11.8	80	1.7	11.9	90	1.6	12	100
MPC	1.4	7.1	0	1.3	7.2	0	1.1	7.2	0	1.0	7.2	0	0.9	7.2	0
OCMIC	0.4	2.4	0	0.4	2.5	0	0.3	2.5	0	0.3	2.5	0	0.3	2.5	0
OCAIC	0.3	2.2	0	0.3	2.3	0	0.3	2.3	0	0.3	2.3	0	0.2	2.3	0
DAC	0.3	2.2	0	0.3	2.3	0	0.3	2.3	0	0.3	2.3	0	0.2	2.3	0
ACMIC	0.2	0.5	0	0.2	0.6	0	0.2	0.6	0	0.2	0.6	0	0.1	0.6	0
ACAIC	0.1	0.4	0	0.2	0.5	0	0.2	0.5	0	0.1	0.5	0	0.1	0.5	0

Only two of the coupling metrics, CCBC and $CCBC_m$, are normalized (see Section 4.3.2), thus we could compute precision, recall, and F-measure values for various thresholds within the complete spectrum of metric values (see Figure 4.1). The other coupling metrics are not normalized as they count the total number of coupling connections of a class with other classes in the system (the larger the metric value, the stronger the coupling between two classes). The only exception is CBO, which has a binary value indicating if two classes have a coupling connection or not. In the case of CBO, we based our evaluation on choosing n coupled classes to a given class instead of using actual metric values (as it is done in the case of other structural coupling measures).

For each coupling measure, we varied the cut point from 10 to 500 classes. For instance, in the case of using $CCBC_m$ (see Table 4.3) with a cut point of 10 classes, we obtained a precision of 27.8%, recall was 14.6%, and F-measure was 19.1%. Increasing a cut point to 20 classes provides more candidate classes, thus decreasing precision to 24.7%, but significantly increasing recall values to 22.1% and increasing F-measure to 23.3%. Also notice that while using a cut point of 10 classes, the $CCBC_m$ value for a

Table 4.5: Impact analysis results – 200 to 500 classes

	Cut point=200			Cut point=300			Cut point=400			Cut point=500		
	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr
CCBC$_m$	8.39	52.4	0.55	6.47	58.2	0.54	5.35	62.1	0.53	4.61	65.1	0.52
ICP	4.13	27.3	22	3.26	31.1	22	2.89	34.9	22	2.63	39	22
PIM	4.12	27.1	11	3.25	30.8	11	2.88	34.8	11	2.62	38.9	11
CCBC	3.99	27	0.41	3.44	33.3	0.39	3.16	39.1	0.38	2.98	44.8	0.36
CBO	1.05	13.2	200	1.01	16.1	300	1.0	20.4	400	0.99	26.5	500
MPC	0.72	8.56	0	0.81	11.7	0	1.07	16.3	0	1.22	22.7	0
OCMIC	0.47	4.25	0	0.67	7.98	0	0.97	12.8	0	1.19	20.2	0
OCAIC	0.45	4.15	0	0.66	7.89	0	0.96	12.5	0	1.18	19.9	0
DAC	0.19	2.4	0	0.18	2.41	0	0.18	2.41	0	0.17	2.42	0
ACMIC	0.4	2.41	0	0.64	6.36	0	0.94	11.1	0	1.17	18.5	0
ACAIC	0.4	2.38	0	0.64	6.32	0	0.94	11.1	0	1.17	18.5	0

class is 0.64, which decreases to 0.61 while using a cut point of 20 classes. This means that the conceptual similarities for ten of the candidate classes are within the [0.61, 0.64] interval.

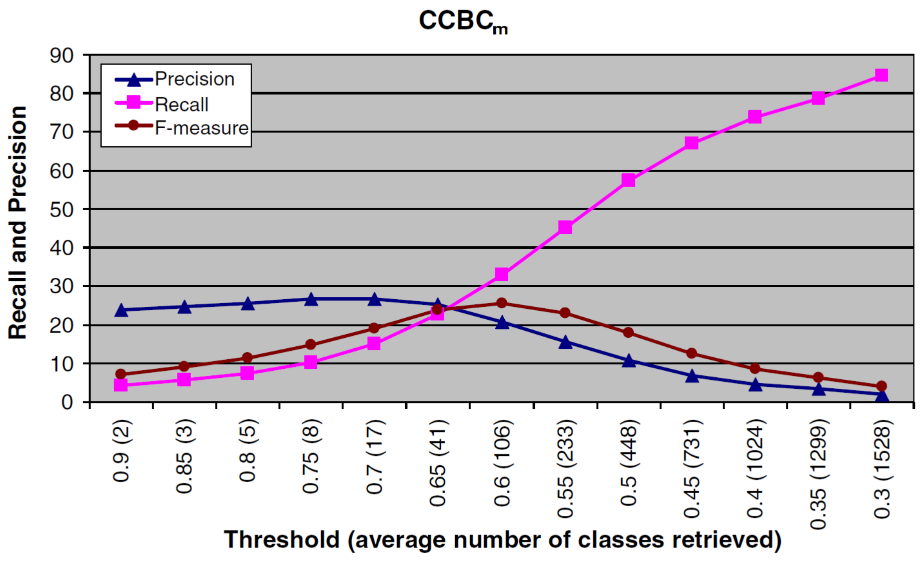
Analysis of the results presented in tables 4.3, 4.4, 4.5, and 4.6 reveals that CCBC$_m$ is the best coupling measure for ranking classes during impact analysis (in terms of precision, recall, and F-measure). None of the other coupling metrics achieve the same magnitude of F-measures (*i.e.*, a maximum of 24.8% for a cut point of 30/40 classes, and 19.0% on average across all the cut points) for any given cut point. For example, in the case of using a cut point of 30 classes, using CCBC$_m$ recovers around 28% of the classes that actually changed (*recall*), and one in five suggested classes is correct (*precision*). These are encouraging results, as the source code of Mozilla consist of 4,853 classes and focusing developers on a set of relevant classes can significantly reduce the amount of time developers spend on impact analysis.

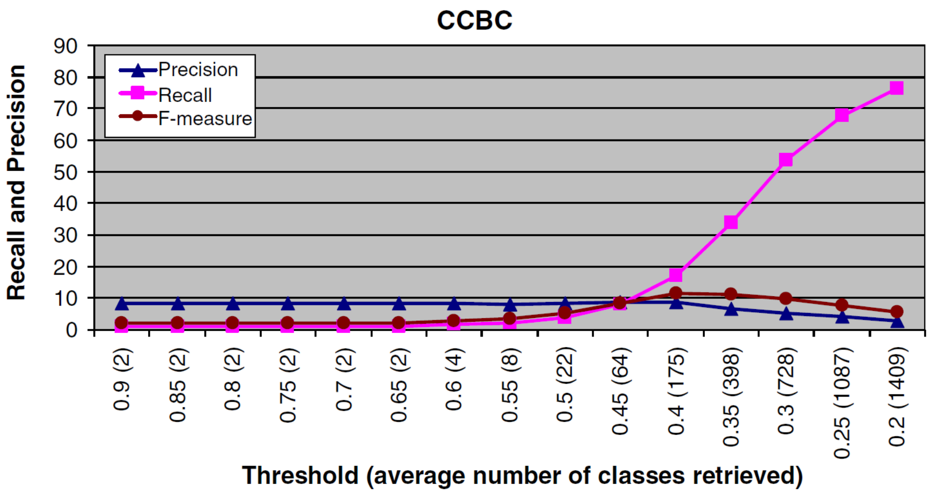
Table 4.6: Impact analysis results – F-measure values (in descending order).

	10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**	**200**	**300**	**400**	**500**	**Avg**
CCBC$_m$	19.1	23.3	24.8	24.8	23.9	22.9	21.9	21.1	20.3	19.5	14.5	11.6	9.86	8.62	**19**
ICP	8.77	9.89	10.5	10.8	11.3	11.7	11.3	10.8	10.4	10.1	7.19	5.91	5.34	4.93	**9.22**
PIM	8.35	9.7	10.3	10.7	11.2	11.6	11.2	10.7	10.4	10.1	7.15	5.89	5.33	4.93	**9.12**
CCBC	7.34	9.18	9.38	9.24	9.05	8.94	8.71	8.51	8.39	8.27	6.95	6.24	5.85	5.59	**7.97**
CBO	6.7	6.85	5.92	5.05	4.45	3.95	3.61	3.28	3.01	2.77	1.96	1.9	2.28	2.54	**3.88**
MPC	6.16	4.97	3.99	3.24	2.73	2.36	2.15	1.93	1.74	1.59	1.34	1.52	2.01	2.33	**2.72**
OCMIC	2.06	1.48	1.15	0.92	0.77	0.67	0.68	0.61	0.55	0.5	0.85	1.23	1.8	2.25	**1.11**
OCAIC	1.85	1.32	0.98	0.78	0.65	0.56	0.59	0.53	0.48	0.43	0.82	1.23	1.79	2.24	**1.02**
DAC	1.88	1.34	0.93	0.81	0.67	0.53	0.53	0.53	0.53	0.37	0.35	0.33	0.33	0.32	**0.68**
ACMIC	0.56	0.43	0.35	0.29	0.25	0.23	0.29	0.26	0.24	0.22	0.69	1.16	1.73	2.2	**0.64**
ACAIC	0.43	0.33	0.27	0.23	0.2	0.19	0.26	0.23	0.21	0.2	0.69	1.15	1.73	2.2	**0.6**

The results for using only structural coupling measures for the same task are less encouraging. The second best metric after CCBC$_m$ is the structural ICP (based on the average F-measure, see Table 4.6), which captures information flow based coupling. This coupling measure captures the number of invocations in a class $c_i \in C$, of methods in a class $d_i \in C$, weighted by the number of parameters of the invoked methods, and also taking polymorphism into account. While using the cut point of 20 classes, the precision of identifying relevant classes using ICP is 10.1%, recall is 9.7%, and F-

Figure 4.1: Results of using CCBC_m and CCBC to rank classes during impact analysis, based on different thresholds. (The number of classes we actually retrieved for every threshold is given in parenthesis)





measure is 9.89%. The best value of the F-measure for ICP (11.7%) is obtained while using a cut point of 60 classes (see Table 4.6).

The next metric after ICP is PIM, which captures the number of method invocations in class $c_i \in C$ of methods in class $d_i \in C$. The measure also takes polymorphism into account. For example, when using the first twenty classes with the highest PIM values as a cut point, the precision of identifying relevant classes is 9.84%, recall is 9.56%, and the F-measure is only 9.7%. The best value of the F-measure for PIM (11.6%) is obtained – again – while using a cut point of 60 classes. PIM has been shown to be a relatively effective coupling measure (as compared to other structural measures) to rank classes during impact analysis in other case studies [55].

The MPC coupling measure shows higher precision values compared to the others in some cases (more than 7%), however it has low recall (around 2% on average) for all of the studied cut points.

The other coupling measures, namely CBO, DAC, ACAIC, ACMIC, OCAIC and OCMIC have low precision and recall values (less than 10%) for all of the computed cut points.

While CCBC uses the same type of information as CCBC_m, it uses a different counting mechanism – it is based on average similarities as opposed to the strongest coupling link between classes. According to the results we saw so far, CCBC_m significantly outperforms CCBC. Moreover, CCBC is outperformed by some of the structural coupling measures such as ICP and PIM as well.

Answer to $\mathbf{RQ_{4.1}}$: *The results show that $CCBC_m$ is a useful indicator (the best among the coupling measures we studied) of an external property of classes in OO systems – change proneness. This coupling measure can be effectively used to rank relevant classes during impact analysis in OO systems.*

CCBC_m performed better on average than any of the structural metrics we compared it to. While we do not investigate the extent to which structural and conceptual coupling measures complement each other in this case study, there is a noticeable potential in combining these coupling measures for ranking classes during impact analysis.

4.5.3 Testing Statistical Significance of Differences Among Precision and Recall Values

In order to compare values of precision and recall for the coupling measures for each of the cut points and conclude whether or not the difference is statistically significant, we executed the Kruskal-Wallis test (see Section 2.5.3) for all of the coupling measures (results in Table 4.7).

Table 4.7: The results of running two Kruskal-Wallis tests for precision (Test 1) and recall (Test 2) values of eleven coupling metrics across the different cut points

	Test 1 Precision	Test 2 Recall
H (observed value)	126.55	110.905
H (critical value)	18.31	18.307
DF (degrees of freedom)	10	10
One-tailed p-value	< 0.0001	<0.0001
Alpha	0.05	0.05

In both tests, at the level of significance for alpha=0.05, the decision was to reject the null hypothesis of absence of differences between even metric values. In other words, both tests have shown that the differences between precision (first test) and recall (second test) values for eleven coupling metrics were statistically significant.

4.6 Threats to Validity

We identify several issues that affects the results of our case study and might limit the generalizability of our interpretations.

In the case study, we considered only structural metrics that were based on the static information obtained from the source code. The results can differ to some extent if dynamic coupling measures are used [39, 177].

The conceptual coupling measures depend on rational naming conventions for identifiers and comments in source code. When these are missing, – as we mentioned in the previous chapter as well – the only hope for measuring any aspects of coupling rests on the structural coupling measures.

Additionally, CCBC and CCBC_m, as they are currently defined, do not take either polymorphism or inheritance into account. The measures – similarly to C3 in the previous chapter – only consider methods for a class that are implemented or overloaded within the class. One of the solutions, which accounts for inheritance, consists of extending the measures to include the source code of inherited methods into the documents of derived classes, as it is done by Kuhn *et al.* [141].

In our case study, we used one large software system, while to allow for generalization of results, large-scale evaluation is necessary. Several releases of software systems from different domains, developed using different programming languages should also be taken into account.

Also, our evaluation is based on the class changes extracted from patches in related bug reports. This could have impacted the evaluation procedure as these patches may contain incomplete information about which classes actually changed, or the changes could have introduced other bugs. We alleviate this issue by considering only patches that are officially approved by module owners in Mozilla.

4.7 Conclusions

This chapter defines a novel set of operational measures for the conceptual coupling of classes, based on IR, which are theoretically valid and empirically studied. Moreover, one of the conceptual coupling measures – CCBC_m – appears to be a superior indicator of change ripple effects as compared to existing structural coupling measures, and it can be effectively used to rank classes of a large OO system during impact analysis.

Contribution

This chapter is based on the publication:

• Denys Poshyvanyk, Andrian Marcus, **Rudolf Ferenc**, and Tibor Gyimóthy. *Using information retrieval based coupling measures for impact analysis.* Empirical Software Engineering, 14(1):5–32, February 2009. Springer Nature. [23]

Similarly to the research referenced in Chapter 3, defining and calculating the conceptual metrics was done by my co-authors, while the design, management, and evaluation of the case study was my responsibility. Moreover, the source code analysis and computation of structural coupling metrics for Mozilla was my work as well.

Some of my other notable papers that were inspired by this result:

• Gábor Szőke, Gábor Antal, Csaba Nagy, **Rudolf Ferenc**, and Tibor Gyimóthy. *Empirical study on refactoring large-scale industrial systems and its effects on maintainability.* Journal of Systems and Software, 129(C):107–126, July 2017. Elsevier. [25]

• Péter Hegedűs, István Kádár, **Rudolf Ferenc**, and Tibor Gyimóthy. *Empirical evaluation of software maintainability based on a manually validated refactoring dataset.* Information and Software Technology, March 2018. Elsevier. [17]

5
New Conceptual Coupling and Cohesion Metrics

5.1 Introduction

In Chapter 3 and Chapter 4, we introduced coupling and cohesion measures that capture the degree of interaction and relationships among source code elements, such as classes, methods, and attributes in object-oriented (OO) software systems. One of the main goals behind OO analysis and design is to implement a software system where classes have high cohesion and low coupling among them. These class properties facilitate comprehension activities, testing efforts, reuse, and maintenance tasks.

The vast majority of coupling and cohesion metrics abound in the literature relies on structural information, which captures relations, such as method calls or attributes usages. These metrics have been proved useful in different tasks, such as assessment of design quality [40, 56], impact analysis [55, 23, 232], prediction of software quality [146], and faults [90, 16, 202], identification of design patterns [37, 5] *etc.* As we have already seen, however, these structural metrics lack the ability to identify conceptual links, which, for example, specify implicit relationships encoded in identifiers and comments in the source code.

In this chapter, we further investigate the usefulness of conceptual metrics and we propose two new ones, namely Conceptual Coupling between Object Classes (CCBO) and Conceptual Lack of Cohesion of Methods (CLCOM5). The proposed metrics are different from the conceptual cohesion and coupling metrics presented in chapters 3 and 4 as they utilize different counting mechanisms inspired by peer structural cohesion and coupling metrics (LCOM5 and CBO, respectively).

In order to evaluate the proposed metrics, we compare CCBO and CLCOM5 against a large host of existing structural and conceptual coupling metrics for predicting faults in a large open-source software system. Furthermore, we perform a comprehensive empirical evaluation of other parameters, including the impact of pre-processing techniques. These parameters also impact the performance of other existing conceptual metrics, such as C3 (see Chapter 3) and CoCC (see Chapter 4). The results of our empirical study indicate that not only can CCBO and CLCOM5 be used to build operational models for predicting the fault-proneness of classes, but they can also be effectively used in conjunction with other structural metrics to improve overall accuracy of bug prediction models.

This chapter offers the following contributions:

- We defined two new conceptual cohesion and coupling metrics, which are easier to compute than their structural counterparts.
- We carried out an extensive empirical study of 61 software metrics, including the newly proposed measures, to build models for fault prediction using machine learning and logistic regression analyses.
- We empirically studied a range of parameters that can impact the performance of CCBO and CLCOM5, such as corpus stemming and parameterized thresholds.
- We developed an online appendix summarizing the results of our empirical study to facilitate the development and comparison of conceptual metrics, as well as the reproducibility of our results.

5.2 Related Work

Our related work can be broadly classified into two areas – conceptual cohesion and coupling metrics, and predicting the fault-proneness of classes.

Conceptual cohesion and coupling metrics were introduced in chapters 3 and 4, respectively, thus the corresponding related work for these topics can be found in sections 3.2 and 4.2. However, some additional related work is also listed here.

WME is a conceptual cohesion metric based on Latent Dirichlet Allocation and information theory approaches [21]. This cohesion metric has been shown to capture different aspects of class cohesion and improve fault prediction for most existing cohesion metrics. While building comprehensive models for fault prediction was not at the focus of papers presenting conceptual metrics, this chapter not only introduces new metrics, but also explores their role in building complete models for fault prediction.

Existing research showed that software metrics can be used as good indicators for the fault-proneness of classes in OO systems [39, 41, 50, 56, 90, 16, 187, 202, 222]. More specifically, some of the existing approaches also utilized machine learning [16] and logistic regression analyses [39, 41, 50, 56, 16, 187, 222] to build metric-based models for fault prediction. We have also investigated the usefulness of the C3 metric in fault prediction in Chapter 3. This chapter, however, defines new conceptual metrics for class cohesion and coupling, which appear to be an improvement over the state-of-the-art.

The chapter also explores a set of machine learning techniques and regression analyses to test a number of models based on the combinations of structural and conceptual metrics along with the detailed investigation into principal factors impacting the performance of the conceptual metrics. Finally, prediction of fault-prone classes – or simply, bug prediction – is an active area of research that produced quite a few research papers in the last decade. Besides conference and journal publications on the topic, specialized conferences were organized such as PROMISE[1] and MSR[2] with their specialized data sets for predicting fault-prone classes in software.

[1]`http://promisedata.org`
[2]`https://www.msrconf.org`

5.3 Conceptual Metrics

Our approach to measuring coupling and cohesion relies on the assumption that the methods and classes of OO systems are connected in more than one way. While the most explored and evaluated set of relations among methods and classes are based on data and control dependencies, we rely on an orthogonal type of relationships, known as conceptual dependencies, to capture the conceptual cohesion and coupling of classes.

Conceptual coupling and cohesion metrics extract, encode, and analyze the semantic information embedded in the comments and identifiers in software (as it was shown in the previous chapters). Software developers utilize the comments and identifiers to represent elements of the problem or solution domain [59, 81]. While the C3 and CoCC metrics also capture this information, we augment the family of conceptual metrics with two new members, namely CCBO and CLOM5. These rely on the same underlying mechanism of LSI to extract and analyze the conceptual information (see Section 2.4).

5.3.1 Definitions

The definition of an OO system with its classes and the methods contained by the classes can be found in definitions 3.1 and 3.2, respectively. The graph representation of an OO system was described in Definition 3.3.

The *conceptual similarity between methods (CSM)* was expressed in Definition 3.4. As defined, the value of $CSM(m_k, m_j) \in [-1, 1]$, as CSM is a cosine similarity in the LSI space. In order to fulfill non-negativity property of software metrics [53], we refined CSM and introduced the definition of CSM^1 in Section 4.3.1. We defined the CCBC metric in Definition 4.2 as the conceptual coupling (similarity) between two classes.

In this chapter, we define conceptual cohesion and coupling metrics utilizing counting mechanisms stemming from existing structural metrics, which are sensitive to the input information such as nodes and edges (*e.g.*, methods and attribute references). Thus, in this section we introduce the notion of parameterized conceptual similarity, which distinguishes among significant and non-significant conceptual interactions among methods of classes.

Definition 5.1: Parameterized Conceptual Similarity — *We conjecture that it is possible to empirically derive a threshold for a given software system to distinguish between strong and weak conceptual similarities. More formally, we define parameterized CSM^P and $CCBC^P$ as:*

$$CSM^P(m_k, m_j, t) = \begin{cases} 1 \quad if \quad CSM^1(m_k, m_j) \geq t \\ else \quad 0 \end{cases} \tag{5.1}$$

$$CCBC^P(c_k, c_j, t) = \begin{cases} 1 \quad if \quad CCBC(c_k, c_j) \geq t \\ else \quad 0 \end{cases} \tag{5.2}$$

Of course, the particular threshold t depends on the specific software system. We previously experienced that the absolute value of the cosine similarity can not be used as a reliable indicator of either the presence or the absence of a conceptual relationship among pairs of methods; more comprehensive analysis of similarity distributions is required. One of the main research questions in our empirical evaluation is centered on empirically deriving such a threshold and analyzing the impact of that choice on the resulting metrics.

5.3.2 Conceptual Lack of Cohesion in Classes

In this section, we define our first metric, namely CLCOM5, using CSM^P (see Definition 5.1) as the foundation for computing conceptual similarities among methods of classes. However, in terms of the counting mechanism, we also heavily rely on one of the ideas from previously defined structural cohesion metrics, namely the graph-based LCOM5 [117]. The main difference between CLCOM5 (our new metric) and C3 (from Chapter 3), is that we define a *parameterized* metric using a different counting mechanism:

$$CLCOM5(c, x) = NoCC(G), \tag{5.3}$$

where $NoCC$ identifies the number of connected components in the graph $G_C = (M(c), E), c \in C, E \subseteq M(c) \times M(c)$, and $(m_k, m_j) \in E$ if $CSM^P(m_k, m_j, t) = 1$.

5.3.3 Conceptual Coupling between Object Classes

The definition of CCBO is inspired by the well-known CBO metric introduced by Chidamber and Kemerer [66, 67]. The definition relies on previous definitions for the CoCC metric, more concretely it uses the CCBC^P, the parameterized conceptual coupling between two classes according to Definition 5.1.

We define the Conceptual Coupling between Object Classes c as the following:

$$CCBO(c, t) = \sum_{c_k \in C, c \neq c_k} CCBC^P(c, c_k, t), \tag{5.4}$$

which is the sum of the parameterized conceptual couplings between a class c and all the other classes in the system.

5.4 Empirical Case Study

In this section, we present the design of the empirical case study aimed at comparing CLCOM5 and CCBO with other structural and conceptual coupling metrics for the task of predicting bugs in open-source software, as well as identifying and analyzing various factors impacting the performance of the proposed measures. The description of the study follows the *Goal-Question-Metrics* design presented in [42]. The data, which has been used to generate the results, was previously used in Chapter 4.

5.4.1 Definition and the Context

Our primary *goals* include comparing our new conceptual metrics against existing coupling and cohesion metrics and determining whether combining both categories could support the task of predicting bugs in large open-source software. In this empirical study, the *quality focus* was on establishing orthogonality among CCBO, CLCOM5, and existing coupling and cohesion metrics, and improving on the accuracy of bug prediction, while the *perspective* was of a software developer analyzing a release of a software system for possible faults. The context of this case study consists of a large open-source software system, that is, *Mozilla* (as it did in the previous chapters). As earlier, we analyzed only C++ classes from the source code and computed CCBO, CLCOM5, and other structural and conceptual metrics among object-oriented classes implemented in C++ only.

Cohesion and coupling metrics

In order to determine whether the newly proposed metrics capture new dimensions in coupling measurement, we selected 61 exiting structural and conceptual metrics for comparison, including coupling metrics (*e.g.*, CBO, RFC), cohesion metrics (*e.g.*, Coh, Coh, LCOM1, LCOM2, LCOM3), and the CK [66] metrics suite as well as other metrics implemented in Columbus [6]. In addition to these structural metrics, we also considered the C3 conceptual cohesion metric introduced in Chapter 3. Another guiding criterion we used to choose the metrics is the availability of results reported for these metrics elsewhere in the literature [56, 16] to facilitate systematic comparison and evaluation of the results.

Building and indexing text corpora

In order to compute CCBO and CLCOM5, we first needed to generate a corresponding corpus for the software system. To build such a corpus for *Mozilla*, we extracted the textual information (*e.g.*, identifiers and comments) from the source code using method level granularity, where each document in the corpus represents a method from the software system (*i.e.*, a sequence of identifiers and comments implementing the corresponding method). More specifically, we extracted the following textual information: (1) comments, (2) local and attribute variable names, (3) user defined types, (4) methods names, (5) parameter lists, and (6) names of the methods that were called. It should be noted that the comments preceding or proceeding the code have been extracted using similar heuristics to [96], which have been implemented in our Columbus reverse engineering framework. Finally, we opted for not including the names of the primitive types in the corpus and we considered those to be a part of our stop word list. Once a corpus is built this way, we index it through LSI using the term-by-document co-occurrence matrix corresponding to the corpus.

Settings of the Case Study

All the structural coupling measures were computed using *Columbus* [6]. In this case study, we used the compiler wrapper technology of Columbus to extract the facts from *Mozilla*'s source code [10]. The textual information needed to compute CCBO and CLCOM5 has been also extracted using the Columbus framework. We used a cross-platform numerical analysis and data processing library ALGLIB[3] to compute the Singular Value Decomposition, which is needed for the LSI algorithm.

We used mostly the same case study settings as in Section 3.4.2. However, one major difference here is that we built various corpora with and without stemming to study its impact on the metrics.

Predicting Faults Using Machine Learning Algorithms and Software Metrics

In order to evaluate the usefulness of our metrics, we conducted a number of analyses to discover possible relationships between the values of the metrics and the number of bugs found in *Mozilla*'s classes through regression analysis methods and machine learning techniques. See Section 2.6 for more information on these topics.

[3]`http://www.alglib.net`

Note that while logistic regression predicts if a class is faulty or not, it does not infer a probable number of bugs in classes. We used *univariate* logistic regression analysis to examine each metric separately, and *multivariate* analysis to study the common effectiveness of the combinations of various metrics (as we also did in Chapter 3).

In addition to regression analyses, we utilized machine learning methods to predict the fault-proneness of classes. In particular, we used Naïve Bayes, Bayesian Logistical Regression, Bayes Net, Logistic Regression, RBF Network, Simple Logistic Regression, SMO, IB-k, Conjunctive Rule, Decision Table, ADTree, and REP Tree, all implemented in Weka [109].

All the models were trained to provide binary predictions, which implies that they predict if a class is prone to be faulty or not based on the values – or combination of values – of particular metrics. In order to estimate the performance of the predictive models we generated, we utilized ten-fold cross-validation. As for the training bug data, we repurposed the bug data that was collected and used in our previous work [16].

5.4.2 Research Questions

We address the following research questions within the context of this empirical study.

- **RQ$_{\mathbf{5.1}}$**: Are the new metrics, CCBO and CLCOM5, orthogonal to existing structural and conceptual coupling and cohesion metrics?
- **RQ$_{\mathbf{5.2}}$**: How does stemming impact accuracy of CCBO and CLCOM5 for predicting the fault-proneness of classes?
- **RQ$_{\mathbf{5.3}}$**: What is the optimal threshold for CCBO and CLCOM5 for predicting fault-prone classes?
- **RQ$_{\mathbf{5.4}}$**: Does combining CCBO and CLCOM5 with existing structural and conceptual cohesion and coupling metrics improve the accuracy of predicting fault-prone classes?

5.4.3 Case Study Results

RQ$_{\mathbf{5.1}}$ – Results of the principal component analysis

PCA was performed on 3,625 classes from *Mozilla* (*i.e.*, classes for which we could compute all the metrics) with 61 structural and conceptual metrics. While the complete results are too lengthy to be presented here, we summarize some of the results, and provide the link to the complete results in the online appendix[4], which also contains the brief explanations of the metrics mentioned in the following.

The PCA resulted in 11 Principal Components (PCs) that describe 87.6% of the variance in our data set. We provide the top four PCs with their interpretations:

PC$_1$ (27%): There are several metrics which were included in this component: cohesion metrics LCOM-LCOM5, NLMA, NLMAni and our **CLCOM5**, size metrics NML, NMLD, NAML, NAL, NMLDpub, NMLpub, LOC, lLOC, coupling metrics NFMA, NOI and RFC, and the WMC complexity metric. These clusters of the results are consistent with previous work with some changes in the rankings of the PCs [56].

PC$_2$ (21%): This component was comprised of several coupling metrics RFC$_1$, RFC$_2$ and RFC$_3$, inheritance-based metrics AID, DIT, NOA, NMI and various size metrics, such as, NM, NMpub, NMprot, NMD, NMDpub, NMDprot, NAM.

[4]`http://www.cs.wm.edu/semeru/scam10-conceptual-metrics`

PC_3 (7.2%): This component was described mostly by NMLDpriv, NMDpriv, NMLpriv and NMpriv metrics. As it can be seen in the results for the other research questions, these metrics' prediction performance was quite offset from the other variants of these metrics, *i.e.*, NMLDpub, NMLDprot.

PC_4 (6%): This component consisted of the structural cohesion Co_1, Co_2, and Coh metrics, as well as the **CCBO** and C3 conceptual metrics.

In addition to the PCA, we also analyzed correlations among the metrics. While we pinpoint a few interesting observations in this paper, we refer the interested reader to the online appendix for the complete analysis results.

CCBO correlated with CLCOM5 with a coefficient of 0.41 and a few other structural metrics, such as Coh, CBO, RFC with a coefficient between 0.4 and 0.5. On the other hand, CLCOM5 was highly correlated with many other structural metrics such as LOC, LLOC, NOI, CBO, RFC, and WMC with correlation coefficients above 0.7. These results indicate that the new conceptual cohesion and coupling metrics are closer to structural metrics as previously defined conceptual metrics, such as C3. This result can be interpreted as a positive result as conceptual metrics are less expensive to compute compared to many structural metrics, and they do not depend on the specific programming language either.

$RQ_{5.2}$ – Identifying the impact of stemming on CCBO and CLCOM5 metrics for predicting fault-prone classes

The conceptual metrics rely on the quality of the underlying comments and identifiers in the source code as well as specific pre-processing strategies used to transform the corpus before indexing. While previous research did not look closely into this important factor, we perform close investigation of the impact of stemming on the performance of conceptual metrics and their combinations with structural metrics to identify fault-prone classes. The goal of this investigation is to identify whether stemming helps in building better models for predicting faults, which utilize conceptual metrics.

Table 5.1: Ten-fold cross validation of conceptual & structural metrics with & w/o stemming for predicting faults

ML Algorithm	Conceptual-no-stem				All-metrics-no-stem				Conceptual-with-stem				All-metrics-with-stem			
	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F
Bayesian Log. Reg.	68.3	70.5	69.1	69.8	71.8	76.7	67.3	71.7	68.8	70.4	71.2	70.8	71.5	71.5	74.7	72.3
Bayes Net	67	64.7	83.4	72.8	70.5	72.5	71.5	72	67.3	65	83.1	73	70.5	72.4	71.8	72.1
Naïve Bayes	68.1	67.9	75.9	71.7	69.2	73.1	66.3	69.6	68.5	67.9	77.2	72.3	69.1	73.1	66.3	69.5
Logistic Regression	67.6	73.6	61	66.7	72.5	76.6	69.5	72.8	67.9	72.6	63.4	67.7	72.4	77	68.4	72.4
RBF Network	67.1	70.1	66.4	68.2	69.3	70.7	71.9	71.3	68.7	68.8	75.1	71.8	69.7	71.9	70.5	71.2
Simple Logistic	67.6	73.5	60.9	66.6	72.1	75.9	69.6	72.6	67.8	72.6	63.4	67.7	71.8	75.6	69.2	72.3
SMO	67.8	73.8	61.1	66.9	72.4	76.2	69.8	72.8	68	72.5	64.1	68	72.2	76	69.7	72.7
IB-k	66.6	67.9	70.3	69.1	71.2	73.1	72.5	72.8	68.8	70.5	70.9	70.7	72.7	74.5	73.8	74.2
Conjunctive Rule	65.8	79.9	47.6	59.6	69.6	81.8	55	65.8	64.5	73.1	52.6	61.2	69.5	82.1	54.3	65.4
Decision Table	67.8	65.8	81.9	73	70.3	73.6	68.5	71	68.1	66.5	80.7	72.9	70.5	74.3	67.9	71
AD Tree	68.4	65.9	84.2	73.9	70.9	72.8	72.3	72.5	68.3	65.7	84.6	74	71	74.4	69.3	71.7
REP Tree	67.3	67.8	73.2	70.4	71.2	72.6	73.6	73.1	67.6	68.5	72.3	70.3	70.6	72.2	72.6	72.4

The results of the ten-fold cross-validation of various configurations of the models – with & without stemming – are presented in Table 5.1. The first part of the table presents the results of applying several machine learning techniques for predicting bugs in *Mozilla* using three conceptual metrics (*i.e.*, CCBO, CLCOM5 and C3) without stemming. As it can be seen, the performance of these models in terms of accuracy (A), precision (P), recall (R), and F-measure (F) are quite high as compared, for instance, to a random classifier. While the performance of the metrics are rather consistent across various machine leaning algorithms, we identify that the AD Tree algorithm produces the highest accuracy, recall and F-measure values (*i.e.*, 68.4%, 84.2%, and 73.9%, respectively), while Conjunctive rule achieves the highest precision.

It should also be noted that the results of combining new conceptual metrics (without stemming) for predicting fault-proneness is comparable to the combination of structural metrics (see Table 5.2). Furthermore, the models based on conceptual metrics are able to outperform the models based on structural metrics in terms of recall and F-measure (*i.e.*, 84.2% *vs.* 73%, and 73.9% *vs.* 72.4%, respectively).

When we compare the results of combining all the metrics (*i.e.*, all-metrics-no-stem in Table 5.1) against conceptual metrics without stemming (*i.e.*, conceptual-no-stem in Table 5.1), we can observe slight improvement in the accuracy (*i.e.*, 72.5% *vs.* 68.4%) and precision (*i.e.*, 81.8% *vs.* 79.9%), while the best recall and F-measure values are obtained with conceptual metrics (*i.e.*, 84.2% and 73.9%, respectively).

Table 5.2: Ten-fold cross validation of the structural metrics for predicting fault-proneness

ML Algorithm	**A**	**P**	**R**	**F**
Bayesian Log. Reg.	70.5	71.8	73	72.4
Bayes Net	69.9	72.1	70.6	71.4
Naïve Bayes	69.1	73.2	66.1	69.5
Logistic Regression	72.1	76.6	68.5	72.3
RBF Network	68.9	75	62.1	67.9
Simple Logistic	71.7	75.2	69.9	72.3
SMO	71.4	74.7	69.9	72.2
IB-k	70.2	72.3	71	71.6
Conjunctive Rule	70.1	81.7	56.4	66.7
Decision Table	70.6	75.3	66.5	70.7
AD Tree	70.9	75.2	67.5	71.1
REP Tree	70.5	72.9	70.5	71.7

According to the results, applying stemming (see conceptual-with-stem in Table 5.1) leads to improvements in the case of accuracy, recall, and F-measure. Moreover, we can observe that this improvement is consistent for these parameters across the different machine learning algorithms we utilized. We can also observe a noticeable improvement in recall and F-measure for conceptual metrics with stemming over structural metrics.

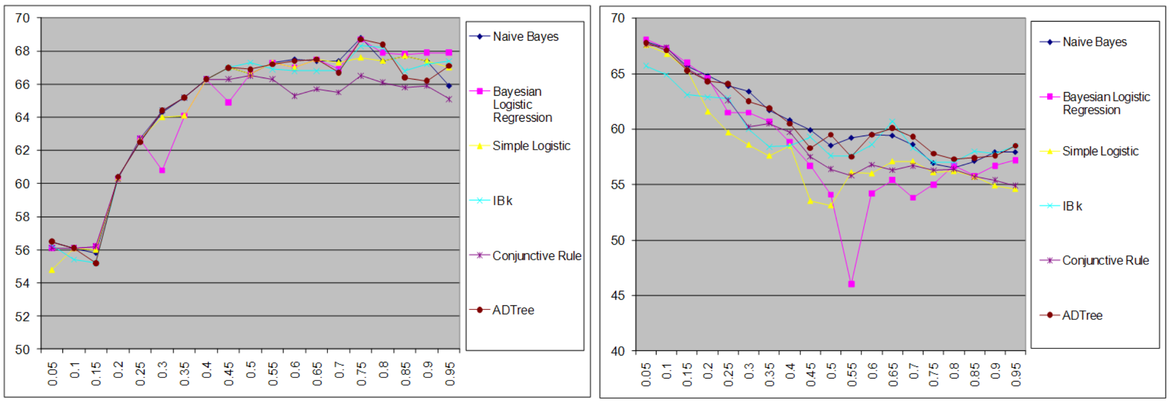
Finally, the results for combining conceptual metrics with stemming and all the structural metrics (all-metrics-with-stem in Table 5.1) leads to the conclusion that this combination produces the best values across all of accuracy, precision, recall, and F-measure (*i.e.*, 72.7%, 82.1%, 74.7%, and 74.2%, respectively). Likewise, the models with all the metrics and stemming outperforms the model based on a combination of pure structural metrics (see all-metrics-with-stem in tables 5.1 and 5.2).

Based on these results, we conclude that *stemming does improve the results for predicting fault-prone classes*. To the best of our knowledge, this is the first research result in the literature that empirically confirms the positive impact of stemming on conceptual metrics as they are applied to predict an external software quality attribute, such as the fault-proneness of classes. According to this, we apply stemming from this point forward to answer the remaining two research questions.

RQ$_{5.3}$ – Identifying the optimal thresholds for CCBO and CLCOM5 for predicting the fault-proneness of classes

CCBO and CLCOM5 are parameterized metrics that depend on the threshold t to identify conceptual similarities among methods. While we used a default threshold of 0.7 to answer RQ$_{5.2}$, it is necessary to identify acceptable values of this parameter for the given task. We acknowledge that the process of identifying an optimal threshold could be software system specific, thus, we present the results for *Mozilla* only.

Figure 5.1: CLCOM5 (left) and CCBO (right) accuracies across different thresholds



In order to search for the optimal thresholds for CCBO and CLCOM5 metrics on our dataset, we computed accuracy values of the metrics across various thresholds starting from 0.05 until 0.95 with a step of 0.05. It should be noted that we used a reduced set of machine learning algorithms in this case, which corresponded to the subset of algorithms indicating a superior performance in RQ$_{5.2}$. According to our results (see Figure 5.1), it can be seen that the thresholds for CLCOM5 resulting in the accuracy of at least 64% reside in the interval [0.3, 0.95], whereas the peak performance of 68.5% in accuracy is observed in the interval of [0.7, 0.8]. These results are consistent across all the machine learning algorithms used in this situation.

On the other hand, the accuracy of CCBO is more sensitive to threshold values as compared to CLCOM5. Here, we observe that the accuracies of the algorithms slowly decline from the 0.05 threshold onward. This finding is quite interesting, suggesting that we should assign higher thresholds for the CLCOM5 cohesion metric and lower thresholds for the CCBO coupling metrics to warrant a better prediction accuracy of fault-proneness.

While using the best thresholds (see Table 5.3) for CLCOM5 and CCBO, we observed some improvement in CLCOM5 over LCOM5 in terms or accuracy (*i.e.*, 68.8% *vs.* 64.6%), recall (*i.e.*, 72.2% *vs.* 71.3%) and F-measure (*i.e.*, 70.8% *vs.* 68.1%). CCBO appears to be better at recall (*i.e.*, 74.6% *vs.* 72.8%). Finally, both new conceptual measures outperform the C3 measure in terms of accuracy, precision, and F-measure.

Table 5.3: Ten-fold cross validation of CLCOM5, LCOM5, CCBO, and C3 for predicting faults in classes

Algorithm	**CLCOM5; t=0.75**				**LCOM5**				**CCBO; t=0.1**				**CBO**				**C3**			
	A	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**	**A**	**P**	**R**	**F**
Naïve Bayes	68.8	71	69.8	70.4	62.8	66.8	59.3	62.8	67.3	68.8	70.3	69.5	71.9	73.9	72.8	73.3	65.4	63.3	83.2	71.9
Bayesian Log. Reg.	68.7	70.2	71.4	70.8	61	69.7	46.9	56.1	67.3	68.4	71.5	69.9	71.9	74.1	72.4	73.2	55.3	54.4	97	69.8
Simple Logistic	67.6	73.5	61	66.6	64.6	65.2	71.3	68.1	66.8	66.7	74.6	70.5	71.9	74.1	72.4	73.2	55.4	54.5	97	69.8
IB-k	68.3	69.4	72.2	70.8	64.6	65.4	70.9	68	64.9	64.8	74.2	69.2	71.9	74.1	72.4	73.2	65.6	63	85.3	72.5
Conjunctive Rule	66.5	73.4	57.9	64.7	64.6	65.2	71.3	68.1	67.4	68.9	70.5	69.7	70.5	77.7	62.3	69.2	61.8	58.5	96.5	72.8
AD Tree	68.7	70.2	71.4	70.8	64.6	65.2	71.3	68.1	67.1	68.6	70.3	69.4	71.9	74.1	72.4	73.2	65.5	63.3	83.1	71.9

RQ$_{5.4}$ – Results of combining CCBO and CLCOM5 with structural and conceptual metrics for fault-proneness.

Lastly, we tested if combining CCBO, CLCOM5, and structural metrics improves the performance of models for fault prediction as compared to combinations of C3 and structural metrics (see Table 5.4).

Based on the results, we conclude that prediction models using a combination of CCBO and CLCOM5 with structural metrics are more robust than combinations of the existing conceptual metric C3 and structural metrics. We derive these conclusions based on the analysis of the average results of accuracy, precision, and recall measures.

Table 5.4: Combining CCBO and CLCOM5 with structural (left), and C3 & structural metrics (right)

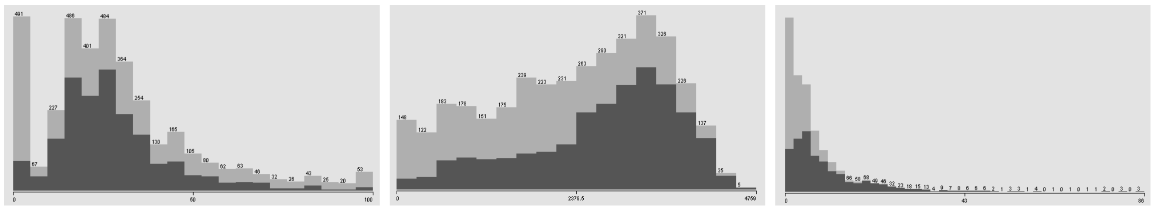
Algorithm	**CCBO, CLCOM5+struct**			**C3+struct**		
	A	**P**	**R**	**A**	**P**	**R**
Bayes. Log. Reg.	71.6	74.8	70.1	71.7	73.1	73.9
Bayes Net	70.7	72.6	72	70.3	72.3	71.6
Naïve Bayes	69.2	73	66.5	68.9	73	65.9
Logistic Reg.	72	75.8	69.6	71.8	76.3	68.1
RBF Network	69.8	72.4	69.8	69.8	72.4	69.5
Simple Logistic	71.6	75.5	69	71.9	75.6	69.5
SMO	72	74	72.7	72.1	75.6	70.1
IB-k	73	75.4	72.9	72.3	74.7	72.2
Conjunctive Rule	69.7	81.1	56	69.9	80.9	56.7
Decision Table	70.4	74.1	67.9	70.4	74.1	67.9
AD Tree	71	72.9	72.3	70.5	74.3	67.9
REP Tree	71.1	73.4	71.3	70.6	72.6	71.6

Analyzing metric intervals

In addition to answering our research questions, we examined the proposed metrics more closely. In particular, we analyzed histograms of distributions of faulty classes across metric intervals, where the x-axis represents metric intervals and the y-axis shows faulty (dark grey) and non-faulty (light gray) classes (see Figure 5.2).

Interestingly enough, all three conceptual metrics – C3, CCBO and CLCOM5 – reinforce our underlying hypotheses. In other words, C3 captures more faulty classes

Figure 5.2: Distribution of (non) faulty classes across C3 (left), CCBO (center) and CLCOM5 (right) metric intervals



while the metric values are low, similarly to the CLCOM5 metric; whereas CCBO captures more faulty classes when the metric values are getting higher.

Results for the logistic regression analysis.

We also decided to examine individual performance of the metrics using *univariate* logistic regression. The set-up of this study was similar to the one we presented in Chapter 3.

The results (shown in Table 5.5) present the top 12 performing metrics (out of a total of 61) in terms of accuracy. According to the numbers, CCBO and CLCOM5 were not the best measures overall. However, CLCOM5 appears to be the best measure within the family of cohesion metrics, and CCBO appears to be one of the best coupling metrics besides CBO, RFC, and RFC3. This result further supports the usefulness of our proposed metrics.

Table 5.5: Results of regression analysis

Metric	Acc.	Prec.	Rec.
CBO	71.9	74.1	72.4
NOI	71.4	76.8	66.1
WMC	70.3	77.7	61.8
RFC	69.8	75.9	63.2
NFMAni	69.8	75.5	63.7
NFMA	69.3	72.9	67.2
lLOC	68.9	76.4	59.8
RFC3	68.9	77.7	58.1
LOC	68.7	76.9	58.6
CLCOM5	67.5	73.5	60.9
CCBO	66.7	66.7	74.6
NAML	66.7	74.7	56.4

5.5 Threats to Validity

In recognition of some of the issues that could have affected the results of the case study (and our interpretation of it), we would like to refer back to sections 3.6 and 4.6. Our previous disclaimers about the fact that more (and diverse) software systems are needed for broader generalization, about naming conventions, and about handling polymorphism all apply here as well.

In addition to these, we also observed that the machine learning algorithms did not generate the best models in every case. In other words, we did not investigate collinearity among the metrics to identify similar groups of metrics to improve the predictive power for the models. Instead, we utilized all the software metrics generated by Columbus.

Also, our metrics rely on parameterized conceptual similarities among methods, which assume specifying a threshold for the operational measures. While we used near-optimal threshold values (as indicated via analysis of all other possible threshold values), these threshold values may vary for other software systems.

5.6 Conclusions

This chapter defines novel operational measures for conceptual class cohesion and coupling measurement, which have been empirically validated. An extensive case study using machine learning techniques on metrics data indicates that the measures we proposed have comparable accuracy to those defined using structural information. Moreover, combinations of the novel metrics with a host of existing measures attests statistically significant improvement in the results across multiple evaluation criteria.

Contribution

This chapter is based on the publication:

• Béla Újházi, **Rudolf Ferenc**, Denys Poshyvanyk, and Tibor Gyimóthy. *New conceptual coupling and cohesion metrics for object-oriented systems.* In Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010), pages 33–42, Timişoara, Romania, September 2010. Best paper of the conference. [27]

Defining the new conceptual metrics CLCOM5 and CCBO and comparing them to their existing counterparts is my contribution, along with the coordination and evaluation of the investigation studying their fault prediction capabilities.

Some of my other notable papers that have contributed to this result:

• Yixun Liu, Denys Poshyvanyk, **Rudolf Ferenc**, Tibor Gyimóthy, and Nikos Chrisochoides. *Modeling class cohesion as mixtures of latent topics.* In Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009), pages 233–242, Edmonton, Canada, September 2009. [21]

• **Rudolf Ferenc**, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. *Columbus – reverse engineering tool and schema for C++.* In Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM 2002), pages 172–181, Montréal, Canada, October 2002. [6]

Part II

Machine Learning for Bug Prediction

ferenc.rudolf_87_23

6 A Public Unified Bug Dataset for Bug Prediction

6.1 Introduction

Finding and eliminating bugs in software systems has always been one of the most critical issues in software engineering. Bug or defect prediction is a process by which we try to learn from mistakes committed in the past and build a prediction model to leverage the location and amount of future bugs. Many research papers on bug prediction introduced new approaches to achieve better precision values [246, 235, 110, 230]. Unfortunately, a reported bug is rarely associated with the source code lines that caused it or with the corresponding source code elements (e.g., classes and methods). Therefore, to carry out such experiments, bugs have to be associated with source code, which is a difficult task. It is necessary to properly use a version control system and a bug tracking system during development. Even in this case, it is still challenging to associate bugs with problematic source code locations.

Although several algorithms were published on associating a reported bug with the corresponding defective source code [74, 234, 62], only a few such bug association experiments were carried out. Furthermore, not all of these studies published the bug dataset, or even if they did, closed-source systems were used, limiting the bug dataset's verifiability and reusability. Despite these facts, several bug datasets (containing information about open-source software systems) were published and made publicly available for further investigations or to replicate previous approaches [229, 209]. These datasets are very popular; for instance, the NASA and the Eclipse Bug Dataset were used in numerous experiments [247, 107, 129, 216].

The main advantage of these bug datasets is that if someone wants to create a new bug prediction model or validate an existing one, it is enough to use a previously created bug dataset instead of building a new one, which would be very resource-consuming. It is common in these bug datasets that all of them store specific information about the bugs, such as the containing source code element(s) with their source code metrics or any additional bug-related information. Since different bug prediction approaches use various sources of information as predictors (independent variables), different bug datasets were constructed. Defect prediction approaches and, hereby, bug datasets can be categorized into larger groups based on the captured characteristics [76]:

- Datasets using process metrics [182, 184].
- Datasets using source code metrics [41, 53, 222].
- Datasets using previous defects [135, 191].

Although these datasets seem very similar, they are often very different in some aspects, which is also true within the categories mentioned above. Our research focused on datasets augmented with static source code metrics. Since this category itself has grown so immense, it is worth studying it as a separate unit. This category also has many dissimilarities between the existing datasets, including the granularity of the data (source code elements can be files, classes, or methods), the representation of element names (different tools may use different notations), and the metrics set can also be different. Even if the names or abbreviations of a metric calculated by different tools are the same, it can have different meanings because it can be defined or calculated slightly differently. The bug-related information given for a source code element can also be contrasting. An element can be labeled whether it contains a bug. Sometimes it shows how many bugs are related to that given source code element. From the information content perspective, it is less critical, but not negligible, that the format of the files containing the data can be CSV (Comma Separated Values), XML, or ARFF (which is the input format of Weka [109]), and these datasets can be found on different places on the Internet.

We collected five publicly available datasets, downloaded the corresponding source code for each system in the datasets, and analyzed the source code to obtain a standard set of source code metrics. As a result, we produced a unified bug dataset at the class and file-levels.

Table 6.1: Example bug dataset table (excerpt)

Type	**Name**	**Path**	**Line**	**Col.**	...	**WMC**	**CBO**	...	**LOC**	...	**bug**
Class	ASTParser	...	83	1	...	96	55	...	1077	...	1
Class	ASTRecoveryPropagator	...	28	1	...	131	57	...	422	...	0
Class	ASTRequestor	...	34	1	...	6	4	...	85	...	0
Class	ASTSyntaxErrorPropagator	...	20	1	...	44	13	...	129	...	0
Class	ASTVisitor	...	104	1	...	170	84	...	2470	...	0
Class	AbstractTypeDeclaration	...	27	1	...	20	11	...	230	...	0
Class	Annotation	...	25	1	...	16	12	...	157	...	0
Class	AnnotationBinding	...	27	1	...	63	31	...	217	...	2
Class	AnnotationTypeDeclaration	...	46	1	...	26	14	...	226	...	0
Class	AnonymousClassDeclaration	...	32	1	...	14	8	...	159	...	0
Class	ArrayAccess	...	28	1	...	27	7	...	243	...	0
Class	ArrayCreation	...	49	1	...	27	11	...	271	...	0
Class	ArrayInitializer	...	28	1	...	13	7	...	133	...	0
Class	ArrayType	...	30	1	...	23	7	...	211	...	0
Class	AssertStatement	...	28	1	...	24	8	...	234	...	0
Class	Assignment	...	30	1	...	33	9	...	312	...	0
Class	BindingComparator	...	33	1	...	91	15	...	275	...	0
Class	BindingResolver	...	31	1	...	53	45	...	971	...	0
...	...	...	...	...	...	...	...	...	...	...	...

To make it easier to imagine how a dataset looks like, Table 6.1 shows an excerpt of an example table where each row contains a Java class with its basic properties like

Name or Path, which are followed by the source code metrics (e.g., WMC, CBO), and the essential property, the number of bugs.

After constructing the unified bug dataset, we examined the diversity of the metric suites. We calculated Pearson correlation and Cohen's d effect size and applied the Wilcoxon signed-rank test (see Section 2.5.3) to reveal these possible differences. We then used a decision tree algorithm (see Section 2.6.2) to show the usefulness of the dataset in bug prediction. We found statistically significant differences in the values of the original and the newly calculated metrics. Furthermore, notations and definitions can severely differ. We compared the bug prediction capabilities of the original and the extended metric suites (within-project learning). Afterward, we merged all classes (and files) into one large dataset consisting of 47,618 elements (43,744 for files), and we evaluated the bug prediction model built on this large dataset. Finally, we also investigated the cross-project capabilities of the bug prediction models and datasets. We made the unified dataset publicly available for everyone. By using a public unified dataset as an input for different bug prediction-related investigations, researchers can make their studies reproducible, thus able to be validated and verified.

Our contributions can be listed as follows:

- Collection of the public bug datasets and source code.
- Unification of the contents of the collected bug datasets.
- Calculation of a common set of source code metrics.
- Comparison of the metrics suites.
- Assessment of the metadata of the datasets.
- Assessment of bug prediction capabilities of the datasets.
- Making the results publicly available.

6.2 Data Collection

This section gives a detailed overview of how we collected and analyzed the datasets. We applied a snowballing-like technique [233] as our data collection process. The following will describe how our start set was defined, the inclusion criteria, and how we iterated over the relevant papers.

6.2.1 Start Set

Starting from the early 70s [207, 123], many studies were introduced concerning software faults. According to Yu et al. [241], 729 studies were published until 2005 and 1,564 until 2015 on bug prediction (the number of studies has doubled in 10 years). From time to time, the enormous number of new publications on software faults made it unavoidable to collect the most critical advances in literature review papers [124, 118, 225]. Since these survey or literature review papers could serve as strong start-set candidates, we used Scopus and Google Scholar to look for these papers. We used these two search sites to fulfill the diversity rule and cover as many different publishers, years, and authors as possible. We considered only peer-reviewed papers. Our search string was the following: '`(defect OR fault OR bug) AND prediction AND (literature OR review OR survey)`'. Based on the title and the abstract, we ended up with 32 candidates. We examined these papers, and based on their content, we narrowed the start set to 12 [61, 110, 205, 118, 221, 60, 164, 130, 163, 225, 33, 151].

Other papers were excluded since they were out of scope, lacked peer review, or were not literature reviews. The included literature papers cover a time interval from 1990 to 2017.

6.2.2 Collecting Bug Datasets

Now we have the starting set of literature review papers; next, we applied backward snowballing to gather all the possible candidates which refer to a bug dataset. In other words, we considered all the references of the review papers to form the final set of candidates. Only one iteration of backward snowballing was used since the survey papers have already included the most relevant studies in the field, and sometimes they have also included reviews about the used datasets.

After having the final candidates (687), we filtered irrelevant papers based on keywords, titles, and abstracts. We also searched for the string 'dataset' or 'data set', or 'used projects'. Investigating the remaining set of papers, we took into consideration the following properties:

- Basic information (authors, title, date, publisher).
- Accessibility of the bug dataset (public, non-public, partially public).
- Availability of the source code.

The latter two are extremely important when investigating the datasets since we need to obtain the appropriate underlying data to construct a unified dataset.

From the final set of papers, we extracted all relevant datasets. We considered the following list to check whether a dataset meets our requirements:

- the dataset is publicly available,
- source code is accessible for the included systems,
- bug information is provided,
- bugs are associated with the relevant source code elements,
- included projects were written in Java,
- the dataset provides bug information at file/class-level, and
- the source code element names are provided and unambiguous (the referenced source code is identifiable).

If any condition was missing, we excluded the subject system or the whole dataset from the study because they could not be included in the unified bug dataset.

Initially, we did not insist on examining Java systems; however, relevant research papers mainly focus on Java language projects [215, 60, 205]. Consequently, we narrowed our research topic to datasets capturing information about systems written in Java. This way, we could use one static analysis tool to extract the characteristics from all the systems; furthermore, including heterogeneous systems would have added a bias to the unified dataset since the interpretation of the metrics, even more, the interpretable set of metrics themselves, can differ from language to language.

The list of found public datasets we could use for our purposes is the following (references are pointing to the original studies in which the datasets were first presented):

- PROMISE – Jureczko [129]
- Eclipse Bug Dataset [247]
- Bug Prediction Dataset [75]

- Bugcatchers Bug Dataset [111]
- GitHub Bug Dataset [26]

It is important to note that we will refer to the Jureczko dataset as the PROMISE dataset throughout the study; however, the repository contains more datasets, such as the NASA MDP [215] (had to be excluded since the source code is not accessible).

6.2.3 Public Datasets

In the following subsections, we will describe the chosen datasets in more detail, investigate each dataset's peculiarities, and look for common characteristics. Before introducing each dataset, we show some basic size statistics about the chosen datasets, presented in Table 6.2. We used the cloc[1] program to measure the Lines of Code. We only considered Java source files, and we neglected blank lines.

Table 6.2: Basic properties of the public bug datasets

Dataset	**Systems**	**Versions**	**Lines of Code**
PROMISE	14	45	2,805,253
Eclipse Bug Dataset	1	3	3,087,826
Bug Prediction Dataset	5	5	1,171,220
Bugcatchers Bug Dataset	3	3	1,833,876
GitHub Bug Dataset	15	105	1,707,446

PROMISE

PROMISE [215] is one of the largest research data repositories in software engineering. It is a collection of many different datasets, including the NASA MDP (Metric Data Program) dataset, used by numerous past studies. However, one should always mistrust the data from an external source [197, 107, 216, 106]. The repository is created to encourage repeatable, verifiable, refutable, and improvable predictive models of software engineering, which is essential for the maturation of any research discipline. The repository is community-based; thus, anybody can donate a new dataset or public tools to help other researchers build state-of-the-art predictive models. PROMISE provides the datasets under categories like code analysis, testing, and software maintenance, and it also has a category for defects. One of the prominent datasets in the repository is from Jureczko et al. [129], which we use in our study. The dataset uses the classic Chidamber & Kemerer (C&K) metrics [66] to characterize the bugs in the systems.

Eclipse Bug Dataset

Zimmerman et al. [247] mapped defects from the bug database of Eclipse 2.0, 2.1, and 3.0. The resulting dataset lists the number of pre- and post-release defects on the granularity of files and packages collected from the BUGZILLA bug tracking system. They collected static code features using the built-in Java parser of Eclipse. They calculated some features at a finer granularity; these were aggregated by taking the

[1]`https://www.npmjs.com/package/cloc`

average, total, and maximum values of the metrics. Data is publicly available and has been used in many studies since then.

Bug Prediction Dataset

The *Bug prediction dataset* [75] contains data extracted from 5 Java projects using inFusion and Moose to calculate the traditional C&K metrics for class-level. The source of information was mainly CVS, SVN, Bugzilla, and Jira, from which the number of pre- and post-release defects were calculated. D'Ambros et al. [75, 76] also extended the source code metrics with change metrics, which, according to their findings, could improve the performance of the fault prediction methods.

Bugcatchers Bug Dataset

Hall et al. presented the *Bugcatchers* Bug Dataset [111], which solely operates with bad smells, and found that coding rule violations have a small but significant effect on the occurrence of faults at the file-level. The Bugcatchers Bug Dataset contains bad smell information about Eclipse, ArgoUML, and some Apache software systems for which the authors used Bugzilla and Jira as the data sources.

GitHub Bug Dataset

Our preliminary work on bug datasets was the GitHub Bug Dataset. In this database, we selected 15 Java systems from GitHub and constructed a bug dataset at class and file-level [26]. This dataset was employed as an input for 13 different machine learning algorithms to investigate which algorithm family performs the best in bug prediction. We included many static source code metrics in the dataset and used these measurements as independent variables in the machine learning process.

6.2.4 Additional Bug Datasets

In this section, we show additional datasets which could not be included in the chosen set. Since this study focuses on datasets that fulfilled our selection criteria and could be used in the unification, we only briefly describe the most important but excluded datasets.

Defects4J – Defects4J is a bug dataset first presented at the ISSTA conference in 2014 [131]. It focuses on bugs from the software testing perspective. Defects4J encapsulates reproducible real-world software bugs. Its repository[2] includes software bugs with their manually cleaned patch files (irrelevant code parts were removed manually), and most importantly, it includes a test suite from which at least one test case fails before the patch was applied and none fails after the patch was applied. Initially, the repository contained 357 software bugs from 5 software systems, but it reached 436 bugs from 6 systems owing to active maintenance.

IntroClassJava – IntroClassJava [86] dataset is a collection of programs, each with several revisions[3]. The revisions were submitted by students and each revision is a maven project. This benchmark is interesting since it contains C programs transformed into Java. Test cases are also transformed into standard JUnit test cases. The

[2]`https://github.com/rjust/defects4j`

[3]`https://github.com/Spirals-Team/IntroClassJava`

benchmark consists of 297 Java programs, each having at least one failing test case. The IntroClassJava dataset is similar to Defects4J but does not provide the manually cleaned fixing patches.

QuixBugs – QuixBugs is a benchmark for supporting automatic program repair research studies [152]. QuixBugs consists of 40 programs written in both Python and Java[4]. It also contains the failing test cases for the one-line bugs located in each program. Defects are categorized, and each defect falls into exactly one category. The benchmark also includes the corrected versions of the programs.

Bugs.jar – Bugs.jar [212] is a large-scale, diverse dataset for automatic bug repair[5]. Bugs.jar falls into the same dataset category as the previously mentioned ones. It consists of 1,158 bugs with their fixing patches from 8 large open-source software systems. This dataset also includes the bug reports and the test suite to support reproducibility.

Bears – Bears dataset [160] is also present to support automatic program repair studies[6]. It uses the continuous integration tool named Travis to generate new entries in the dataset. It includes the buggy state of the source code, the test suite, and the fixing patch.

All the datasets described above focus on bugs from the software testing perspective and also support future automatic program repair studies. These datasets can be good candidates to be used in fault localization research studies as well. These datasets capture buggy states of programs and provide the test suite and the patch. Our dataset is fundamentally different from these datasets. The datasets we collected gathered information from a wider time interval and provided information for each source code element by characterizing them with static source code metrics.

6.3 Data Processing

Although the found public datasets have similarities (e.g. containing source code metrics and bug information), they are inhomogeneous. For example, they contain different metrics calculated with different tools and for different kinds of code elements. The file formats are also different; therefore, using these datasets together is challenging. Consequently, we aimed to transform them into a unified format and extend them with source code metrics calculated with the same tool for each system. This section will describe the steps we performed to produce the unified bug dataset.

First, we transformed the existing datasets into a common format. This means that if a bug dataset for a system consisted of separate files, we conflated them into one file. Next, we changed the *CSV* separator in each file to *comma (,)*, renamed the column called the '*number of bug*' in each dataset to '*bug*' and changed the column name '*source code element*' to '*filepath*' or '*classname*' depending on the granularity of the dataset. Finally, we transformed the source code element identifier into the standard form (e.g., *org.apache.tools.ant.AntClassLoader*).

[4]`https://github.com/jkoppel/QuixBugs`
[5]`https://github.com/bugs-dot-jar/bugs-dot-jar`
[6]`https://github.com/bears-bugs/bears-benchmark`

6.3.1 Metrics Calculation

The bug datasets contain different metric sets, which were calculated with different tools; therefore, even if the same metric name appears in two or more different datasets, we must be sure they mean the same metric. We analyzed all the systems with the same tool to eliminate this deficiency. For this purpose, we used the free and open-source *OpenStaticAnalyzer* 1.0 (OSA) tool (see Section 2.2) that can analyze Java systems (among other languages).

The metrics in the original bug datasets were calculated with five different tools (inFusion Moose, ckjm, Visitors written for Java parser of Eclipse, Bad Smell Detector, SourceMeter – which is a commercial product based on OSA; see Table 6.7). From these tools, only ckjm and SourceMeter are still available on the internet, but the last version of ckjm is from 2012, and the Java language has evolved a lot since then. Additionally, ckjm works on the bytecode representation of the code, which makes it necessary to compile the source code before analysis. Consequently, we selected OSA because it is a state-of-the-art analyzer that works on the source code and enables more precise analysis besides being easier to use. Additional tools are also available, but this work did not aim to find the best available tool.

For calculating the new metric values, we needed the source code itself. Since all datasets belonged to a release version of a given software, if the software was open-source and the given release version was still available, we could manage to download and analyze it. This way, we obtained two results for each system: one from the downloaded bug datasets and one from the OSA analysis.

6.3.2 Dataset Unification

We merged the original datasets with the results of OSA by using the "unique identifiers" of the elements (Java standard names at the class-level and paths at the file-level). More precisely, the basis of the unified dataset was our source code analysis result, and it was extended with the data of the given bug dataset. This means that we went through all elements of the bug dataset, and if the "unique identifier" of an element was found in our analysis result, then these two elements were conjugated (paired the original dataset entry with the one found in the result of *OSA*). Otherwise, it was left out of the unified dataset. Table A.1 and Table A.2 show the results of this merging process at class and file-level, respectively: column *OSA* shows how many elements OSA found in the analyzed systems, column *Orig.* presents the number of elements originally in the datasets, and column *Dropped* tells us how many elements of the bug datasets could not be paired, and so they were left out from the unified dataset. The numbers in parentheses show the number of dropped elements where the drop was caused by the sources not being real Java sources, such as *package-info.java* and *Scala files* (which are also compiled to byte code and hence included in the original dataset). Although these numbers are auspicious, we had to "modify" a few systems to achieve this, but there were cases where we could not solve the inconsistencies. The details of the source code modifications and the main reasons for the dropped elements are the followings:

Camel 1.2: In the *org.apache.commons.logging*, there were 13 classes in the original dataset that we did not find in the source code. There were five *package-info.java*[7] files

[7]Scala (see also Camel 1.4 and 1.6) and *package-info.java* files are "not real" Java source files

in the system, but these files did not contain any Java classes since they are used for package-level Javadoc purposes; therefore, OSA did not find such classes.

Camel 1.4 Besides the 7 *package-info.java* files, the original dataset contained information about 24 Scala files (they are also compiled to byte code); therefore, OSA did not analyze them.

Camel 1.6: There were 8 *package-info.java* and 30 Scala files.

Ckjm 1.8: A class in the original dataset did not exist in version 1.8.

Forrest-0.8: Two different classes appeared twice in the source code; therefore, we deleted the copies from the *etc/test-whitespace* subdirectory.

Log4j: There was a *contribs* directory that contained the source code of different contributors. These files were put into the appropriate sub-directories (where they belonged according to their packages), meaning they occurred twice in the analysis, preventing their merging. Therefore, we analyzed only those files in their appropriate subdirectories in these cases and excluded the files found in the *contribs* directory.

Lucene: All three versions had an *org.apache.lucene.search.Remote-Searchable_Stub* class in the original dataset that did not exist in the source code.

Velocity: In versions 1.5 and 1.6 there were two *org.apache.velocity.app.event. implement.EscapeReference* classes in the source code; therefore, it was impossible to conjugate them by using their "unique identifiers" only.

Xerces 1.4.4: Although the name of the original dataset and the corresponding publication state that this is the result of Xerces 1.4.4 analysis, we found that 256 out of the 588 elements did not exist in that version. We examined a few previous and following versions, and it turned out that the dataset is much closer to 2.0.0 than 1.4.4 because only 42 elements could not be conjugated with the analysis result of 2.0.0. Although version 2.0.0 was still not matched perfectly, we did not find a "closer version"; therefore, we used Xerces 2.0.0 in this case.

Eclipse JDT Core 3.4: Many classes appeared twice in the source code: once in the "code" and once in the "test" directory; therefore, we deleted the test directory.

Eclipse PDE UI 3.4.1: The missing 6 classes were not found in its source code.

Equinox 3.4: Three classes could not be conjugated because they did not have a unique name (there are more classes with the same name), while two classes were not found in the system.

Lucene 2.4 (BPD): Twenty-one classes from the original dataset were not present in the source code of the analyzed system.

Mylyn 3.1: 457 classes were missing from our analysis that were in the original dataset; therefore, we downloaded different versions of Mylyn, but still needed help finding the matching source code. We could only achieve a better result by knowing the correct version.

ArgoUML 0.26 Beta: Three classes in the original dataset did not exist in the source code.

Eclipse JDT Core 3.1: Twenty-five classes did not exist in the analyzed system.

GitHub Bug Dataset: Since OSA is the open-source version of SourceMeter, the tool used to construct the GitHub Bug Dataset, we could easily merge the results. However,

and should not be included in the original results. Their quantities are presented in parenthesis in Table A.1.

the class-level bug datasets contained elements with the same "unique identifier" (since class names are not the standard Java names in that case), so this information was insufficient to conjugate them. Luckily, the paths of the elements were also available, and we used them; therefore, all elements could be conjugated. Since they performed a machine learning step on the versions that contained the most bugs, we decided to select these release versions and present the characteristics of these release versions. We also used these versions of the systems to include in the unified bug dataset.

As a result, we obtained a unified bug dataset containing all the public datasets in a unified format. Furthermore, they were extended with the same metrics provided by the OSA tool. The last lines of Table A.1 and Table A.2 show that only 1.29% (624 out of 48,242) of the classes and 0.06% (28 out of 43,772) of the files could not be conjugated, which means that only 0.71% (652 out of 92,014) of the elements were left out from the unified dataset in total.

In many cases, the analysis results of OSA contained more elements than the original datasets. Since we did not know how the bug datasets were produced, we could not give an exact explanation for the differences, but we list the two main possible causes:

- In some cases, we could not find the proper source code for the given system (e.g., Xerces 1.4.4 or Mylyn), so two different but close versions of the same system might be conjugated.
- OSA considers nested, local, and anonymous classes while some datasets associated Java classes with files.

6.4 Original and Extended Metrics Suites

In this section, we present the metrics proposed by each dataset. Additionally, we will show a metrics suite that is used by the newly constructed unified dataset.

6.4.1 Original Datasets

PROMISE – The authors calculated the metrics of the PROMISE dataset with the tool called *ckjm* [173]. All metrics, except McCabe's Cyclomatic Complexity (CC), are *class* level metrics. Besides the C&K metrics, they also calculated some additional metrics shown in Table A.3.

Eclipse Bug Dataset – In the Eclipse Bug Dataset, there are two types of predictors. By parsing the structure of the obtained abstract syntax tree, they calculated the number of nodes for each type in a package and in a *file* (*e.g.* the number of return statements in a file) [247]. By implementing visitors to the Java parser of Eclipse, they also calculated various complexity metrics at method, class, file, and package levels. Then they used *avg, max, total avg, total max aggregation* techniques to accumulate to file and package level. The complexity metrics used in the Eclipse dataset are listed in Table A.4.

Bug Prediction Dataset – The Bug Prediction Dataset collects product and change (process) metrics. The authors produced the corresponding product and process metrics at *class* level [75]. Besides the classic CK metrics, they calculated some additional object-oriented metrics that are listed in Table A.5.

Bugcatchers Bug Dataset – The Bugcatchers Bug Dataset differs from the previous datasets since it does not contain traditional software metrics but the number of bad smells for files. They used five bad smells presented in Table A.6. Besides, in the CSV file, there are four source code metrics (blank, comment, code, codeLines), which are not explained in the corresponding publication [111].

GitHub Bug Dataset – The GitHub Bug Dataset [26] used the free version of the SourceMeter tool to calculate the static source code metrics, including software product metrics, code clone metrics, and rule violation metrics. The rule violation metrics were not used in our research, therefore, Table A.7 shows only the list of the software product and code clone metrics at the class-level. At the file-level, only a narrowed set of metrics is calculated, but there are four additional process metrics included, as Table A.8 shows.

6.4.2 Unified Bug Dataset

The unified dataset contains all the datasets with their original metrics and with further metrics that we calculated with OSA. The set of metrics calculated by OSA concurs with the metric set of the GitHub Bug Dataset because SourceMeter is a product based on the free and open-source OSA tool. Therefore, all datasets in the Unified Bug Dataset are extended with the metrics listed in Table A.7 except the GitHub Bug Dataset because it contains the same metrics originally.

In spite of the fact that several of the original metrics can be matched with the metrics calculated by OSA, we decided to keep all the original metrics for every system included in the unified dataset because they can differ in their definitions or in the ways of their calculation. One can simply use the unified dataset and discard the metrics that were calculated by OSA if they want to work only with the original metrics. Furthermore, this provides an opportunity to confront the original and the OSA metrics.

Instead of presenting all the definitions of metrics here, we give an external resource to show metric definitions because of the lack of space. All the metrics and their definitions can be found in the Unified Bug Dataset file, reachable as an online appendix (see Section A.2).

6.4.3 Comparison of the Metrics

In the unified dataset, each element has numerous metrics, but these values were calculated by different tools, therefore, we assessed them in more detail to get answers to questions like the following ones:

- Do two metrics with the same name have the same meaning?
- Do metrics with different names have the same definition?
- Can two metrics with the same definition be different?
- What are the root causes of the differences if the metrics share the definition?

Three out of the five datasets contain class level elements, but unfortunately, for each dataset, a different analyzer tool was used to calculate the metrics (see Table 6.7). To be able to compare class level metrics calculated by all the tools used, we needed at least one dataset for which all metrics of all three tools are available. We were already familiar with the usage of the ckjm tool, so we chose to calculate the ckjm metrics for the Bug Prediction dataset. This way, we could assess all metrics of all tools because the Bug Prediction dataset was originally created with Moose, so we have extended it with the OSA metrics, and also – for the sake of this comparison – with ckjm metrics.

In the case of the three file-level datasets, the used analyzer tools were unavailable, therefore, we could only compare the file-level metrics of OSA with the results of the other two tools separately on Eclipse and Bugcatchers Bug datasets.

In each comparison, we merged the different result files of each dataset into one, which contained the results of all systems in the given dataset, and deleted those elements that did not contain all metric values. For example, in the case of the Bug Prediction Dataset, we calculated the OSA and ckjm metrics, then we removed the entries which were not identified by all three tools. Because we could not find the analyzers used in the file-level datasets, we used the merging results seen in Section 6.3.2. For instance, in the case of the Bugcatchers Bug Dataset, the new merged (unified) dataset has 14,543 entries (491 + 1,752 + 12,300), out of which 2,305 were in the original dataset and not dropped (191 + 1,582 + 560 - 28).

The resulting spreadsheet files can be found in the online appendix. Table 6.3 shows how many classes or files were in the given dataset and how many of them remained. We calculated the basic statistics (minimum, maximum, average, median, and standard deviation) of the examined metrics and compared them (see Table 6.4). Besides, we calculated the pairwise differences of the metrics for each element and examined its basic statistics as well. In addition, the *Equal* column shows the percentage of the classes for which the two examined tools gave the same result (for example, at class level OSA and Moose calculated the same WMC value for 2,635 out of the 4,167 elements, which is 63.2%, see Table 6.4).

Table 6.3: Number of elements in the merged systems

Name	**Merged**	**Remained elements**
Bug Prediction Dataset	11,370	4,167
Eclipse Bug Dataset	25,295	25,210
Bugcatchers bug Dataset	14,543	2,305

Since the basic statistic values gave only some impression about the similarity of the metric sets, we performed a Wilcoxon signed-rank test (see Section 2.5.3), which determines whether two dependent samples were selected from populations having the same distribution. In our test, the H_0 hypothesis is that the difference between the pairs follows a symmetric distribution around zero, while the alternative H_1 hypothesis is that the difference between the pairs does not follow a symmetric distribution around zero. We used 95% confidence level in the tests to calculate the *p-values.* This means that if a *p-value* is higher than *0.05*, we accept the H_0 hypothesis, otherwise, we reject it and accept the H_1 alternative hypothesis instead. In all cases, the p-values were less than 0.001, therefore, we had to reject the H_0 and accept that the difference between the pairs does not follow a symmetric distribution around zero.

Although from the statistical point of view, the metric sets are different, we see that in many cases, there are a lot of equal metric values. For example, in the case of the file-level dataset of Eclipse (see Table 6.6), only 11 out of 25,199 metrics are different, but 10 out of these 11 are larger only by 1 than their pairs, so the test recognizes well that it is not symmetric. On the other hand, in this case, we can say that the two groups can be considered identical because less than 0.1% of the elements differ, and the difference is neglectable. Therefore we calculated *Cohen's d* as well, which indicates the standardized difference between two means. Besides, to see how strong or weak the correlations between the metric values are, we calculated the *Pearson correlation coefficient* (see Section 2.5.3). We used 0.8 for the threshold above which the correlation is considered strong.

Class Level Metrics

The unified bug dataset contains the class level metrics of OSA and Moose on the Bug Prediction dataset. We downloaded the Java binaries of the systems in this dataset and used ckjm version 2.2 to calculate the metrics. The first difference is that while OSA and Moose calculate metrics on source code, ckjm uses Java bytecode and takes "external dependencies" into account, therefore, we expected differences, for instance, in the coupling metric values.

We compared the metric sets of the three tools and found that, for example, CBO and WMC have different definitions. On the other hand, the efferent coupling metric is a good example of a metric that is calculated by all three tools but with different names (see Table 6.4, CBO row). In the following paragraphs, we only examine those metrics whose definitions coincide in all three tools, even if their names differ. Table 6.4 shows these metrics where the *Metric* column contains the abbreviation of the most widely used name of the metric. The *Tool* column presents the analyzer tools, and in the *Metric name* column, the metric names are given using the notations of the different datasets. The "$tool_1 - tool_2$" means the pairwise difference where, for each element, we extracted the value of tool_2 from the value of tool_1 and the name of this "new metric" is diff. The following columns present the basic statistics of the metrics. The Equal column denotes the percentage of the elements having the same metric value (i.e. the difference is 0), and the last two columns are the Cohen's d value and the Pearson correlation coefficient (where it is appropriate). We highlighted with boldface those values that suggest that the two metric set values are close to each other from a given aspect:

- if more than half of the element pairs are equal (i.e. Equal is above 50%),
- if the effect size is small (i.e. Cohen's d is less than 0.2),
- if there is strong linear correlation between the elements (i.e. the absolute value of the Pearson correlation coefficient is larger than 0.8).

Next, we will analyze the metrics one at a time.

WMC: This metric expresses the complexity of a class as the sum of the complexity of its methods. In its original definition, the method complexity is deliberately not defined exactly; and usually, the uniform weight of 1 is used. In this case, this variant of WMC is calculated by all three tools. Its basic statistics are more or less the same, and the pairwise values of OSA and ckjm seem to be closer to each other (see OSA−ckjm row) than to Moose, and the extremely high Pearson correlation value (0.995) supports this. Among the Moose results, there are several very low values where the other tools found a great number of methods, and that caused the extreme difference (e.g. the max. value of OSA−Moose is 420). In spite of this, the Pearson correlations between the results of Moose and the other tools are high. On the other hand, OSA and Moose gave the same result for almost two third of the classes, which means that the difference probably comes from outliers. And finally, Cohen's d values are small, so we can say that the three tools gave very similar results but with notable outliers.

Table 6.4: Comparison of the common class level metrics in the Bug Prediction dataset

Metric	Tool	Metric name	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
WMC	OSA	NLM	0	426	11.04	7	18.12	-	-	-
	Moose	Methods	0	403	9.96	6	14.38	-	-	-
	ckjm	WMC	1	426	11.96	7	18.49	-	-	-
	OSA−Moose	diff	-4	**420**	1.08	0	9.40	**63.2%**	**0.066**	**0.857**
	OSA−ckjm	diff	-48	0	-0.91	0	1.94	**51.9%**	**0.050**	**0.995**
	Moose−ckjm	diff	**-421**	4	-1.99	-1	9.57	41.1%	**0.120**	**0.859**
CBO	OSA	CBO	0	214	8.86	5	12.25	-	-	-
	Moose	fanOut	0	93	6.22	4	7.79	-	-	-
	ckjm	Ce	0	213	13.78	8	16.88	-	-	-
	OSA−Moose	diff	-32	161	2.65	2	7.61	16.0%	0.258	**0.801**
	OSA−ckjm	diff	-120	83	-4.91	-1	9.72	26.2%	0.333	**0.823**
	Moose−ckjm	diff	-160	32	-7.56	-4	11.84	7.4%	0.575	0.780
CBOI	OSA	CBOI	0	607	9.38	3	26.14	-	-	-
	Moose	fanIn	0	355	4.69	1	14.30	-	-	-
	ckjm	Ca	0	611	7.64	2	22.13	-	-	-
	OSA−Moose	diff	-18	**607**	4.69	1	16.55	43.4%	0.222	**0.821**
	OSA−ckjm	diff	-100	189	1.74	0	11.02	**59.6%**	**0.072**	**0.909**
	Moose−ckjm	diff	**-611**	146	-2.95	-1	15.30	39.6%	**0.158**	0.727
RFC	OSA	RFC	0	600	22.82	12	34.53	-	-	-
	Moose	rfc	0	2,603	50.62	23	108.06	-	-	-
	ckjm	RFC	2	684	38.93	23	49.72	-	-	-
	OSA−Moose	diff	**-2,095**	600	-27.80	-8	83.70	8.4%	0.347	0.786
	OSA−ckjm	diff	-327	12	-16.11	-9	22.72	4.9%	0.376	**0.917**
	Moose−ckjm	diff	-673	**2,049**	11.69	-1	75.42	5.7%	**0.139**	0.787
DIT	OSA	DIT	0	8	1.31	1	1.63	-	-	-
	Moose	dit	**1**	9	2.08	2	1.44	-	-	-
	ckjm	DIT	0	5	0.38	0	0.60	-	-	-
	OSA−Moose	diff	-3	0	-0.76	-1	0.43	23.9%	0.496	**0.969**
	OSA−ckjm	diff	-5	8	0.94	1	1.96	22.1%	0.761	0.418
	Moose−ckjm	diff	-4	9	1.70	2	1.79	30.2%	1.540	0.453
NOC	OSA	NOC	0	107	0.73	0	3.27	-	-	-
	Moose	noc	0	49	0.64	0	2.55	-	-	-
	ckjm	NOC	0	107	0.64	0	2.95	-	-	-
	OSA−Moose	diff	-3	97	0.08	0	1.68	**96.8%**	**0.028**	**0.863**
	OSA−ckjm	diff	0	42	0.09	0	1.15	**97.1%**	**0.029**	**0.937**
	Moose−ckjm	diff	-97	34	0.01	0	1.81	**95.8%**	**0.003**	0.794
LOC	OSA	LLOC	2	8,746	131.99	56	357.39	-	-	-
	Moose	LinesOfCode	0	7,341	124.01	51	306.54	-	-	-
	ckjm	LOC	4	26,576	399.42	147	1142.60	-	-	-
	OSA−Moose	diff	-1,068	7,824	7.98	3	157.69	3.1%	**0.024**	**0.898**
	OSA−ckjm	diff	-19,150	112	**-267.43**	-91	791.30	0.6%	0.316	**0.988**
	Moose−ckjm	diff	-26,541	198	**-275.41**	-93	879.89	0.1%	0.329	**0.893**
NPM	OSA	NLPM	0	404	7.23	4	13.67	-	-	-
	Moose	PublicMethods	0	387	6.42	4	11.28	-	-	-
	ckjm	NPM	0	404	7.48	5	13.64	-	-	-
	OSA−Moose	diff	-4	236	0.81	0	6.55	**68.0%**	**0.065**	**0.879**
	OSA−ckjm	diff	-8	0	-0.25	0	0.45	**75.8%**	**0.018**	**0.999**
	Moose−ckjm	diff	-237	3	-1.06	0	6.55	**62.2%**	**0.085**	**0.879**

CBO: In this definition, CBO counts the number of classes the given class depends on. Although it is a coupling metric, it counts efferent (outer) couplings, therefore, the metric values should have been similar. On the other hand, based on the statistical values and the pairwise comparison, including Equal and Cohen values, we can say that these metrics differ significantly. We can observe a strong linear correlation among them (Pearson values are close to or above 0.8), but since there are few equal values, we can suspect that they differ by a constant in most cases. The reasons can be, for example, that ckjm takes into account "external" dependencies (e.g. classes from *java.util*) or it counts coupling based on generated elements too (e.g. generated default constructor), but further investigation would be required to determine all causes.

CBOI: It counts those classes that use the given class. Although the basic statistics of OSA and ckjm are close to each other, their pairwise comparison suggests that they are different because there are large outliers and the averages of the diffs are commensurable with the averages of the tools. Based on Equal, Cohen, and Pearson values, it seems that the metric values of OSA and ckjm are close to each other, but the metric values of Moose are different. The main reason can be, for example, that OSA found two times more classes, therefore, it is explicable that more classes use the given class or ckjm takes into account the generated classes and connections as well that exist in the bytecode, but not in the source code.

RFC: All three tools defined this metric in the same way, but the comparison shows that the metric values are very different. The tools are able to calculate the same metric value for less than 10% of the classes (Equal value), but the high or almost high Pearson correlation values indicate that there is some connection between the values. The reasons for this are mainly the same as in the case of the CBO metric.

DIT: Although the statistical values "hardly" differ compared to the previous ones, these values are usually small (as the max. values show), therefore, these differences are quite large. From the minimal values, we can see that Moose probably counts *Object* too as the base class of all Java classes, while the other two tools neglect this. The only significant connection among them is the very large Pearson correlation (0.969) between OSA and Moose, but its proper explanation would require a deeper investigation of the tools.

NOC: Regarding this metric, the three tools calculate very similar values. More than 95% of the metric values are the same for each pairing which is very good. On the other hand, the remaining almost 5% of the values may differ significantly because the minimum and maximum values of the differences are large compared to the absolute maximum metric values and, at the same time, the Pearson correlation values are not extremely large, which means that the 5% impairs it a lot.

LOC: Lines of code should be the most unambiguous metric, but it also differs a lot. Although this metric has several variants and it is not defined exactly how Moose and ckjm count it, we used the closest one from OSA based on the metric values. The very large value of ckjm is surprising, but it counts this value from the bytecode; therefore, it is not easy to validate it. Besides, OSA and Moose have different values, in spite of the fact that both of them calculate LOC from source code. The 0 minimal value of Moose is also interesting and suggests that either Moose used a different definition or the algorithm was not good enough. We found the fewest equal metric pairs for LOC metric (Equal values are 3.1%, 0.6%, 0.1%), but the large Pearson correlation values (close or above 0.9%) suggest that the metrics differ mainly by a constant value only.

NPM: Based on the statistical results, the number of public methods metrics seems to be the most unambiguous metric. Both the basic statistics and the Equal, Cohen, and Pearson triplet imply that the metrics are close to each other. OSA and ckjm are really close to each other (75% of the values are the same, and the rest is also close to each other because the Pearson correlation coefficient is 0.999), while Moose has slightly different results. However, the average difference is around 1, which can be caused by counting implicit methods (constructors, static init blocks) or not.

The comparison of the three tools revealed that, even though they calculate the same metrics, in some cases, the results are very divergent. A few of the reasons can be that *ckjm* calculates metrics from bytecode while the other two tools work on source code, or *ckjm* takes into account external code as well while *OSA* does not. Besides, we could not compare the detailed and precise definitions of the metrics to be sure that they are really calculated in the same way, therefore, it is possible that they differ slightly, which causes the differences.

File-Level Metrics

Bugcatchers, Eclipse, and GitHub Bug Dataset are the ones that operate at the file-level (GitHub Bug Dataset contains class level too). Unfortunately, we could make only pairwise comparisons between file-level metrics since we could not replicate the measurements used in the Eclipse Bug Dataset (custom Eclipse JDT visitors were used) and in the Bugcatchers Bug Dataset (unknown bad smell detector was used).

In the case of Bugcatchers Bug Dataset, we compared the results of OSA and the original metrics, which were produced by a code smell detector. Since OSA only calculates a narrow set of file-level metrics, Logical Lines of Code (LLOC) is the only metric we could use in this comparison. Table 6.5 presents the result of this comparison. Min, max, and median values are likely to be the same. Moreover, the average difference between LLOC values is less than 1 with a standard deviation of 6.05 which could be considered insignificant in the case of LLOC at the file-level. Besides, more than 90% of the metric values are the same, and the remaining values are also close because Cohen's d is almost 0, and the Pearson correlation coefficient is very close to 1. This means that the two tools calculate almost the same LLOC values.

There is an additional common metric (CLOC) which is not listed in Table 6.5 since OSA returned 0 values for all the files. This possible error in OSA makes it superfluous to examine CLOC in further detail.

Table 6.5: Comparison of file-level metrics in the Bugcatchers dataset

Metric	Tool	Met. name	Min	Max	Avg	Med	Dev	Equal	Cohen	Pearson
	OSA	LLOC	3	5,774	93.33	41	221.16	-	-	-
LLOC	Smell Detector	code	3	5,774	92.34	40	219.06	-	-	-
	OSA−Smell Detector	diff	-11	130	**0.98**	0	**6.05**	**90.8%**	**0.004**	**1.000**

In the case of the Eclipse Bug Dataset, LLOC values are the same in most of the cases (see Table 6.6). OSA counted one extra line in 10 cases out of 25,210, and once it missed 7 lines which is a negligible difference. This is the cause of the "perfect" statistical values. Unfortunately, there is a serious sway in the case of McCabe's Cyclomatic Complexity. There is a case where the difference is 299 in the calculated values, which is extremely high for this metric. We investigated these cases and found that OSA does not include the number of methods in the final value. There are many cases when OSA gives 1 as a result, while the Eclipse Visitor calculates 0 as complexity. This is because OSA counts class definitions but not method definitions. There are cases where OSA provides higher complexity values. It turned out that OSA took the ternary operator (?:) into consideration, which is correct since these statements also form conditions. Both calculation techniques seem to have some minor issues, or at least we have to say that the metric definitions of cyclomatic complexity differ. This is why there are only so few matching values (4%), but the Cohen's d and the Pearson correlation coefficient suggest that these values are still related to each other.

Table 6.6: Comparison of file-level metrics in the Eclipse dataset

Metric	**Tool**	**Met. name**	**Min**	**Max**	**Avg**	**Med**	**Dev**	**Equal**	**Cohen**	**Pearson**
	OSA	LLOC	3	5,228	122.59	52	230.02	-	-	-
LLOC	Visitor	TLOC	3	5,228	122.59	52	230.02	-	-	-
	OSA−Visitor	diff	-7	1	**0.0001**	0	**0.048**	**100.0%**	**0.000**	**1.000**
	OSA	McCC	1	1,198	19.55	5	48.27	-	-	-
McCC	Visitor	VG_sum	0	1,479	28.06	10	**60.35**	-	-	-
	OSA−Visitor	diff	**-299**	123	-8.50	-4	15.83	4.0%	**0.156**	**0.982**

The significant differences both at the class and file-level show that tools interpret and implement the definitions of metrics differently. Our results further strengthen the conclusion reported by Lincke et al. [153], who described similar findings.

6.5 Evaluation

In this section, we first evaluate the unified bug dataset's basic properties, like the number of source code elements and the number of bugs to gain a rough overview of its contents. Next, we show the dataset's metadata, like the used code analyzer or the calculated metric set. Finally, we present an experiment in which we used the unified bug dataset for its main purpose, namely for bug prediction. Our aim was not to create the best possible bug prediction model but to show that the dataset is a usable source of learning data for researchers who want to experiment with bug prediction models.

6.5.1 Datasets and Bug Distribution

Table A.9 shows the basic properties of each dataset. In the *SCE* column, the number of source code elements is presented. Based on granularity, it means the number of classes or files in the system. There are systems in the datasets with a wide variety of sizes from 2,636 *Logical Lines of Code (LLOC)*[8] up to 1,594,471.

SCEwBug means the number of source code elements that have at least one bug, *SCEwBug%* is the percentage of the source code elements with bugs in the dataset. The *SCEwBug%* as the percentage of buggy classes or files describes how well-balanced the datasets are. Since it is difficult to overview the numbers, Figure 6.1 and Figure 6.2 show the distribution of the percentages of faulty source code elements (SCEwBug%) for classes and files, respectively. The percentages are shown horizontally, and, for example, the first column means the number of systems (part of a dataset) that have between 0 and 10 percentages of their source code elements buggy (0 included, 10 excluded). In the case of the systems that give bug information at the class-level, this number is 11 (5 at the file-level). We can see that there are systems where the percentages of the buggy classes are very high; for example, 98.79% of the classes are buggy for Xalan 2.7 or 92.20% for Log4J 1.2. Although the upper limit of SCEwBug% is 100%, the reader may feel that these values are extremely high, and it is very difficult to believe that a release version of a system can contain so many bugs. The other end

[8]Lines of code not counting comments and whitespace.

is when a system hardly contains any bug; for example, in the case of MCT and Neo4j, less than 1% of the classes is buggy. From the project point of view, it is very good, but on the other hand, these systems are probably less usable when we want to build bug prediction models. This phenomenon further strengthens the motivation to have a common unified bug dataset that can blur these extreme outliers. Although there are many systems with extremely high or low SCEwBug% values, we used them "as is" later in this research because their validation was not the aim of this work.

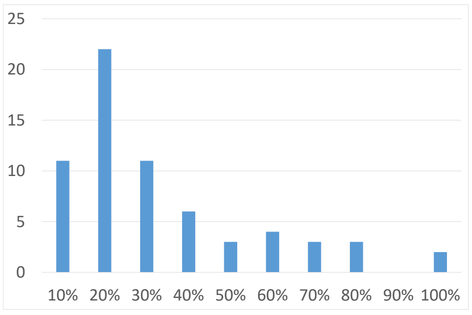


Figure 6.1: Fault distribution (classes)

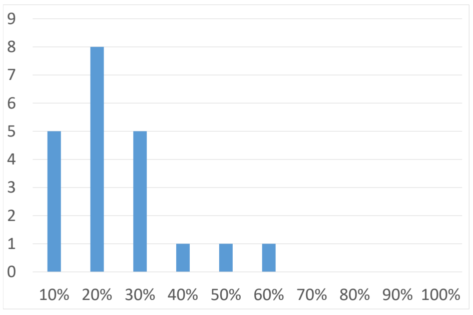


Figure 6.2: Fault distribution (files)

6.5.2 Metadata of the Datasets

Table 6.7 lists some properties of the datasets, which show the circumstances of the dataset rather than the data content. Our focus is on how the datasets were created and how reliable the tools used, and the applied methods were. Since most of the information in the table was already described in previous sections (Analyzer, Granularity, Metrics, and Release), we will describe only the Bug information row in this section.

The Bug Prediction Dataset used the commit logs of SVN and the modification time of each file in CVS to collect co-changed files, authors, and comments. Then, they linked the files with bugs from Bugzilla and Jira using the bug id from the commit messages [75]. Finally, they verified the consistency of timestamps. They filtered out inner classes and test classes.

The PROMISE dataset used Buginfo to collect whether an SVN or CVS commit is a bugfix or not. Buginfo uses regular expressions to detect the commit messages which contain bug information.

The bug information of the Eclipse Bug Dataset was extracted from the CVS repository and Bugzilla. In the first step, they identified the corrections or fixes in the version history by looking for patterns that are possible references to bug entries in Bugzilla. In the second step, they mapped the bug reports to versions using the version field of the bug report. Since the version of a bug report can change during the life cycle of a bug, they used the first version.

The Bugcatchers Bug Dataset followed the methodology of Zimmermann et al. (Eclipse Bug Dataset). They developed an Ant script using the SVN and CVS plugins of Ant to checkout the source code and associate each fault with a file.

In our previous work – in the case of the GitHub bug dataset – we gathered the relevant versions to be analyzed from GitHub. Since GitHub can handle references between commits and issues, it was quite handy to use this information to match

Table 6.7: Metadata of the datasets

	Bug Prediction Dataset	**PROMISE**	**Eclipse Bug Dataset**	**Bugcatchers Bug Dataset**	**GitHub Bug Dataset**
Analyzer	inFusion Moose	ckjm	Visitors written for Java parser of Eclipse	Bad Smell Detector	SourceMeter
Granularity	Class	Class	File	File	Class, File
Bug information	CVS, SVN, Bugzilla, Jira	SVN, CVS	CVS, Bugzilla	CVS, SVN, Bugzilla, Jira	GitHub
Metrics	C&K, process metrics	C&K	Complexity, Structure of abstract syntax tree	Bad Smell	C&K, Complexity, Clone, Rule violation
Release	post	pre	pre & post	pre	post

commits with bugs. In that work, we collected the number of bugs located in each file/class for the selected release versions (about 6-month long time intervals).

6.5.3 Bug Prediction

We evaluated the strength of bug prediction models built with the Weka [109] machine learning software. For each subject software system in the Unified Bug Dataset, we created 3 versions of ARFF files (which is the default input format of Weka) for the experiments (containing only the original, only OSA, and both sets of metrics as predictors). In these files, we transformed the original bug occurrence values into two classes as follows: 0 bug → non buggy class, and at least 1 bug occurrence → buggy class. Using these ARFF files, we could run several tests about the effectiveness of fault prediction models built based on the dataset.

Within-project Bug Prediction

As described in Section 6.4, we extended the original datasets with the source code metrics of the OSA tool in order to create a unified bug dataset. We compared the bug prediction capabilities of the original metrics, the OSA metrics, and the extended metric suite (both metric suites together). First, we handled each system individually, so we trained and tested on the same system data using tenfold cross-validation. To build the bug prediction models, we used the J48 (C4.5 decision tree – see Section 2.6.2) algorithm with default parameters. We used only J48 since we did not focus on finding the best machine learning method, but we rather wanted to show a comparison of the different predictors' capabilities with this one algorithm. We chose J48 because it has shown its power on the GitHub Bug Dataset [26] and because it is relatively easy to interpret the resulting decision trees and to identify the subset of metrics that are the most important predictors of bugs. Different machine learning algorithms (*e.g.* neural networks) might provide different results. It is also important to note that we did not use any sampling technique to balance the datasets; we used the datasets 'as is'. We will outline some connections between the machine learning results and the characteristics of the datasets (*e.g.* SCEwBug%).

The F-measure results can be seen in Table A.10 for classes and in Table A.11 for files. We also included the *SCEwBug%* characteristic of each dataset to gain a more detailed view of the results. The tables contain two average values since the GitHub Bug Dataset used SourceMeter, which is based on OSA, to calculate the metrics. The results of OSA and the Merged metrics would be the same. This could distort the averages, so we decided to detach this dataset from others when calculating the averages.

Results for classes need some deeper explanation for clear understanding. There are a few missing values since there were less than 10 data entries, not enough to do the ten-fold cross-validation on (Ckjm and Forrest-0.6).

There are some outstanding numbers in the tables. Considering Xalan 2.7, for example, we can see F-measure values of 0.992 and above. This is the consequence of the distribution of the bugs in that dataset which is extremely high as well (98.79%). The reader could argue that resampling the dataset would help to overcome this deficiency, however, the corresponding AUC value for the Original dataset is not outstanding (only 0.654). Besides, later in this section, we will investigate the bug prediction capabilities of the datasets in a cross-project manner, and it turns out that these extreme outliers generally performed poorly in cross-project learning, as we will describe later in detail. Let us consider Forrest 0.8 as an example where the *SCEwBug%* is low (6.25), however, the F-measure values are still high (0.891-0.907). In this case, the high values came from the fact that there are 32 classes, out of which only 2 are buggy, so the decision tree tends to mark all classes as non-buggy. On the other hand, the AUC values range from very poor (0.100) to very high (0.900), which suggests that for small examples and a small number of bugs, it is difficult to build “reliable” models.

The averages of F-measure and AUC slightly change in the case of class-level datasets, and the average of the Merged dataset is only slightly better than the Original or the OSA. If we consider F-measure, the GitHub Bug Dataset performed 10% better generally, while the averages of AUC decreased by only 1%, which is neglectable. One possible explanation can be that the PROMISE dataset includes all the smaller projects, while the GitHub Bug Dataset and the Bug Prediction Dataset rather contain larger projects.

Small differences in the case of the GitHub Bug Dataset come from the difference in the versions of the static analyzers. The SourceMeter version used in creating the GitHub Bug Dataset is older than the OSA version used here, in which some metric calculation enhancements took place. (SourceMeter is based on OSA.)

Results at the file-level are quite similar to class-level results in terms of F-measure and AUC. The only main difference is that the F-measures average of OSA for Bugcatchers and Eclipse bug datasets is notably smaller than the other values. The reason for this might be that there are only a few file-level metrics provided by OSA, and a possible contradiction in metrics can decrease the training capabilities (we saw that even LLOC values are very different). On the other hand, the corresponding AUC value is close to the others, so a deeper investigation is required to find out the causes. Besides, the average F-measure of the Merged model is slightly worse, while the average AUC is slightly better than the Original, meaning that in this case, the OSA metrics were not able to improve the bug prediction capability of the model. The small difference between the Original and the OSA results in the case of the GitHub Bug Dataset comes from the slightly different metric suites since OSA calculates the Public Documented API (PDA) and Public Undocumented API (PUA) metrics as well. Furthermore, the aforementioned static analyzer version differences also caused this small change in the average F-measure and AUC values.

Merged Dataset Bug Prediction

So far, we compared the bug prediction capabilities of the old and new metric suites on the systems separately, but we have not used its main advantage, namely, there are metrics that were calculated in the same way for all systems (OSA metric suite). Seeing the results of the within-project bug prediction, we can state that creating a larger dataset, which includes projects varying in size and domain, could lead to a more general dataset with increased usability and reliability. Therefore, we merged all classes (and files) into one large dataset, which consists of 47,618 elements (43,744 for files), and by using tenfold cross-validation, we evaluated the bug prediction model built by J48 on this large dataset as well. The F-measure is 0.818 for the classes and 0.755 for the files, while the AUC is 0.721 for classes and 0.699 for files. Although the F-measure values are, for example, a little worse than the average of GitHub results (0.892 for classes and 0.820 for files), the AUC values are better than the best averages (0.656 for classes and 0.676 for files) even if they were measured on a much larger and very heterogeneous dataset.

After training the models at class and file-levels, we dug deeper to see which predictors are the most dominant. Weka gives us pruned trees as a result of both at the class and file-levels. A pruned tree is a transformed one obtained by removing nodes and branches without affecting the performance too much. The main goal of pruning is to reduce the risk of overfitting. We not only constructed a Unified Bug Dataset for all of the systems together with the unified set of metrics, but we also built one for each collected dataset, one for the Bug Prediction Dataset, one for the PROMISE dataset, etc., in which we could use a broader set of metrics (including the original ones).

Table 6.8 presents the most dominant metrics for the datasets extracted from the constructed decision trees. We marked those metrics with boldfaces that occur in multiple datasets. At the class-level, WMC (Weighted Methods per Class) and TNOS (Total Number of Statements) are the most important ones, however, they happened not to be the most dominant ones in the Unified Bug Dataset.

Table 6.8: Most dominant predictors per dataset

Dataset	**Level**	**Dominant predictors**
Bug Prediction Dataset	class	**wmc**, **TNOS**, cvsEntropy
GitHub Bug Dataset	class	**WMC**, NOA, **TNOS**
PROMISE	class	LOC, DIT, TNM
Unified Bug Dataset	class	CLOC, TCLOC, CBO, NOI, DIT
Bugcatchers Bug Dataset	file	**code**, PDA, SpeculativeGenerality
GitHub Bug Dataset	file	**McCC**, NumOfPrevMods, NumOfDevCommits
Eclipse Bug Dataset	file	TypeLiteral, NSF_max, **MLOC_sum**
Unified Bug Dataset	file	**LOC**, **McCC**

Regarding the Unified Bug Dataset, we found CLOC, TCLOC (comment lines of code metrics), DIT (Depth of Inheritance), CBO (Coupling Between Objects), and NOI (Number of Outgoing Invocations) as the most dominant metrics at the class-level. In other words, these metrics have the largest entropy. This set of metrics being the most dominant is reasonable since it is easier to modify and extend classes that have better documentation. Furthermore, coupling metrics such as CBO and NOI have already demonstrated their power in bug prediction [16, 53]. DIT is also an expressive metric for fault prediction [90].

At the file-level, the most dominant predictors in the different datasets show overlapping with the ones being dominant in the Unified Bug Dataset. LOC (Lines of Code) and McCC (McCabe's Cyclomatic Complexity) were the upmost variables to branch on in the decision tree. These two metrics are reasonable as well since the larger the file, the more it tends to be faulty. Complexity metrics are also important factors to include in fault prediction [247].

The diversity of the most dominant metrics shows the diversity of the different datasets themselves.

Cross-project Bug Prediction

As a third experiment, we trained a model using only one system from a dataset and tested it on all systems in that dataset. This experiment could not have been done without the unification of the datasets since a common metric suite is needed to perform such machine learning tasks. The result of the cross-training is an NxN matrix where the rows and the columns of the matrix are the systems of the dataset and the value in the i^{th} row, and j^{th} column shows how well the prediction model performed, which was trained on the i^{th} system and was evaluated on the j^{th} one.

We used the OSA metrics to test this criterion, but the bug occurrences are derived from the original datasets, which are transformed into the buggy and non-buggy labels. The matrix for the whole Unified Bug Dataset would be too large to show here; thus we will only present submatrices (the full matrices for file and class-levels are available in the online appendix). A submatrix for the PROMISE dataset can be seen in Table A.12 and A.13. Only the first version is presented for each project except for the ones where another version has a significantly different prediction capability (for example, Xalan 2.7 is much worse than 2.4, or Pbeans 2 is much better than 1). The values of the matrix are F-measure and AUC values provided by the J48 algorithm. Higher F-measure and AUC values are indicated with darker gray, however, it is important to note that the coloring is done for each table individually (we will introduce cross-training for other datasets as well in the rest of this section). Absolute white is the color for the lowest value (not necessarily 0.0), and the deepest gray encodes the highest value (not necessarily 1.0). Avg. F-m. and Avg. AUC columns[9] present the average of the F-measures and AUC values the given model achieved on the other systems. For a better overview, we repeat *SCEwBug%* value here as well (see Table A.9).

We can observe (see Table A.12) that models built on systems (highlighted bold in the table) having lots of buggy classes (more than 70%) performed very poor in the cross-validation if we consider the average F-measure (the values are under 0.4). Even more, these models do not work well on other very buggy systems either. For example, the model trained on Xalan 2.7 (the buggiest system) achieved only 0.078 F-measure on Log4J 1.2 (the second most buggy system). Besides, in the case of Ckjm 1.8, Lucene 2.2, and Poi 1.5, the rate of buggy classes is between 50 and 70 percent, and their models are still weak; the average F-measure is between 0.4 and 0.6 only. And finally, the other systems can be used to build such models that achieve better

[9]We present the average values only for PROMISE dataset because it contains extremely low values, namely almost white lines.

results, namely higher F-measures, on other systems, even the ones having lots of bugs. On the other hand, this trend cannot be observed on the AUC values (see Table A.13) because they range between 0.483 and 0.629, but there is no “white line” which means that there is no model whose performance is very poor for all other systems. However, there are 0.000 and 0.900 AUC values in the table that suggest that the bug prediction capabilities heavily depend on the training and testing dataset. For example, models evaluated on Forrest 0.6 have extreme values, and it is probably only a matter of luck whether the model takes into account the appropriate metrics or not.

Table A.14 shows the cross-training F-measure values for the GitHub Bug Dataset. Testing on Android Universal Image Loader is the weakest point in the matrix, as it is clearly visible. However, the values are not critical, and the lowest value is still 0.611. Based on the F-measure, Elasticsearch performed slightly better than the others in the role of a training set. This might be because of the size of the system, the average amount of bugs, and the adequate number of entries in the dataset. On the other hand, the AUC values in the column of Android Universal Image Loader are not significantly worse than any other value (see Table A.15), but rather the values in its row seem a little bit lower. From the AUC point of view, Mission Control T. is the most critical test system because it has the lowest (0.181) and the highest (0.821) AUC values. The reason can be the same as for Forrest 0.6. In general, the AUC values are very diverse, ranging from 0.181 to 0.821, and the average is 0.548.

Table 6.9 and 6.10 show the F-measure and AUC values of cross-training for the Bug Prediction Dataset. Based on the F-measure, Eclipse JDT Core passed the other systems in terms of training (which is unequivocally shown by the deep gray color in the table), but its AUC values, or at least the first two, seem worse than the others. Equinox performed the worst in the role of being a training set, i.e., having the lowest F-values, but from the AUC point of view, Equinox is the most challenging test set because 3 out of 4 of its AUC values are much worse than the average. From testing, training, and F-measure and AUC value point of view, Mylyn is the best system.

Table 6.9: Cross training (Bug prediction dataset - class level) - F-Measure values

Train/Test	Eclipse JDT Core 3.4	Equinox 3.4	Lucene 2.4 BPD	Mylyn 3.1	PDE UI 3.4.1
Eclipse JDT Core 3.4		0.878	0.874	0.848	0.839
Equinox 3.4	0.451		0.614	0.664	0.641
Lucene 2.4 BPD	0.754	0.706		0.808	0.809
Mylyn 3.1	0.730	0.702	0.753		0.815
PDE UI 3.4.1	0.733	0.695	0.748	0.769	

Table 6.10: Cross training (Bug prediction dataset - class level) - AUC values

Train/Test	Eclipse JDT Core 3.4	Equinox 3.4	Lucene 2.4 BPD	Mylyn 3.1	PDE UI 3.4.1
Eclipse JDT Core 3.4		0.489	0.467	0.555	0.601
Equinox 3.4	0.593		0.638	0.590	0.535
Lucene 2.4 BPD	0.561	0.487		0.560	0.506
Mylyn 3.1	0.685	0.642	0.628		0.569
PDE UI 3.4.1	0.383	0.435	0.548	0.626	

The above-described results were calculated for class-level datasets. Let us now consider the file-level results. The three file-level datasets are the GitHub, the Bug-catchers, and the Eclipse bug datasets.

The GitHub Bug Dataset results can be seen in Table A.16 and A.17. As in the case of class-level, the Android Universal Image Loader project performed the worst in being the test set based on F-measure and the worst system for building models (low AUC values). It would be difficult to select the best system, but the Mission Control T. system is the most critical test system again because it has the lowest and highest AUC values.

The cross-training results of the Bugcatchers Bug Dataset are listed in Table 6.11 and 6.12. The table contains 3 high and 3 low F-measure values, and they range on a wide scale from 0.234 to 0.800, while the AUC values are much closer to each other (0.500−0.606). Since only three systems are used, and there is no significant best or worst system, we cannot state any conclusion based on these results.

Table 6.11: Cross training (Bugcatchers - file-level) - F-Measure values

Train/Test	Apache	ArgoUML	Eclipse JDT Core 3.1
Apache		0.800	0.676
ArgoUML	0.436		0.665
Eclipse JDT Core 3.1	0.388	0.234	

Table 6.12: Cross training (Bugcatchers - file-level) - AUC values

Train/Test	Apache	ArgoUML	Eclipse JDT Core 3.1
Apache		0.541	0.519
ArgoUML	0.563		0.606
Eclipse JDT Core 3.1	0.500	0.588	

In the Eclipse Bug Dataset, there are only three systems as well, but they are three different versions of the same system, namely Eclipse, therefore, we expected better results. As we can see in Table 6.13 and 6.14, in this case, the F-measure and AUC values coincide, which means that either both of them are high or both of them are low. The model built on version 2.0 performed better (0.705 and 0.700 F-measures and 0.723 and 0.708 AUC values) on the other two systems than the models built on the other two systems and evaluated on version 2.0 (0.638 and 0.604 F-measures and 0.639 and 0.633 AUC values). This is perhaps caused by the fact that this version contains the least number of bug entries. The other two systems are "symmetrical"; the results are almost the same when one is the train, and the other one is the test system.

Table 6.13: Cross training (Eclipse - file-level) - F-Measure values

Train/Test	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0		0.705	0.700
Eclipse 2.1	0.638		0.725
Eclipse 3.0	0.604	0.676	

We also performed a full cross-system experiment involving all systems from all datasets. This matrix is, however, too large to present here, consequently, it can be found in the online appendix and Table 6.15 and 6.16 show only the average of the F-measures and AUC values of the models performed on other datasets. More precisely, we trained a model using each system separately and tested this model on the other

Table 6.14: Cross training (Eclipse - file-level) - AUC values

Train/Test	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0		0.723	0.708
Eclipse 2.1	0.639		0.700
Eclipse 3.0	0.633	0.702	

systems, and we calculated the averages of the F-measures and AUC values to see how they performed on other datasets. For example, we can see that the average F-measures of the models trained and tested on PROMISE is only 0.607, while if these models are validated on the GitHub dataset, the average is 0.729. Examining F-measures, we can see that, in general, models trained on the PROMISE dataset perform the worst, and what is surprising is that they gave the worst result (0.607) on themselves. On the other hand, if we consider AUC values, the model trained on PROMISE achieved the best result on PROMISE (0.554 vs. 0.544). But in this case, the coloring of the table might be misleading because the AUC values range from 0.541 to 0.562, meaning that the difference is only 0.021, which is small, and the two values are close to each other, but one of them is white while the other is dark gray. This means that, in this case, AUC values do not help us to select the best or worst predictor/test dataset or even compare the results. Another interesting observation is that the models perform better on the GitHub dataset no matter which dataset was used for training (GitHub column). On the other hand, GitHub models achieve only slightly worse F-measures on the other datasets than the best one on the given dataset (0.678 vs. 0.680 on PROMISE and 0.727 vs. 0.742 on Bug Prediction Dataset), which suggests that the good testing results are not a consequence of some unique feature of the dataset because it can also be used to train “portable” bug prediction models.

Table 6.15: Average F-measures of the full cross-system experiment (class level)

Train/Test	GitHub	PROMISE	Bug prediction
GitHub	0.842	0.678	0.727
PROMISE	0.729	0.607	0.617
Bug prediction	0.815	0.680	0.742

Table 6.16: Average AUC values of the full cross-system experiment (class level)

Train/Test	GitHub	PROMISE	Bug prediction
GitHub	0.548	0.544	0.562
PROMISE	0.552	0.554	0.557
Bug Prediction	0.555	0.541	0.555

Table 6.17 shows the average F-measures for the file-level datasets. We can observe a similar trend, namely, all models performed the best on the GitHub dataset. Besides, the results of Bugcatchers are poor, and the other two bug prediction model sets performed better on it. On the other hand, the AUC values do not support this observation (see Table 6.18). Compared to class level, the AUC values range on a broader scale, from 0.525 to 0.684, and suggest that Eclipse is the best dataset for model building, and this observation is supported by the F-measures as well.

Table 6.17: Average F-measures of the full cross-system experiment (file-level)

Train/Test	GitHub	Bugcatchers	Eclipse
GitHub	0.786	0.613	0.583
Bugcatchers	0.654	0.533	0.529
Eclipse	0.769	0.660	0.675

Table 6.18: Average AUC values of the full cross-system experiment (file-level)

Train/Test	GitHub	Bugcatchers	Eclipse
GitHub	0.587	0.525	0.585
Bugcatchers	0.579	0.553	0.612
Eclipse	0.661	0.564	0.684

To sum up the previous findings based on the full cross-system experiment, it may seem like file-level models performed slightly better than class level predictors because the average F-measure is 0.717 and the average AUC value is 0.594 for files while they are only 0.661 and 0.552 for classes. On the other hand, this might be the result of the fact that we saw that several systems of PROMISE contain too many bugs, therefore, they cannot be used alone to build sophisticated prediction models. Another remarkable result is that all models performed notably better on the GitHub dataset, which requires further investigation in the future.

6.6 Threats to Validity

First of all, we accepted the collected datasets "as is", which means that we did not validate the data; we just used them to create the unified dataset and to examine the bug prediction capabilities of the different bug datasets. Since the bug datasets did not contain the source code or step-by-step instructions on how to reproduce the bug datasets, we had to accept them, even if there were a few notable anomalies in them. For example, Camel 1.4 contains classes with LOC metrics of 0; in the Bugcatchers dataset, there are two MessageChains metrics, and in several cases, the two metric values are different; or there are more datasets with extreme *SCEwBug%* (buggy classes percentage) values (more than 90% high or less than 1% low).

Although the version information was available for each system, in some cases, there were notable differences between the result of OSA and the original result in the corresponding bug dataset. Even if the analyzers would parse the classes in different ways, the number of files should have been equal. If the analysis result of OSA contains the same number of elements or more, and (almost) all elements from the corresponding bug dataset could be paired, we can say that the unification is acceptable because all elements of the bug dataset were put into the unified dataset. On the other hand, for a few systems, we could not find the proper source code version, and we had to leave out a negligible number of elements from the unified dataset.

Many systems were more than 10 years old when the actual Java version was 1.4, and these systems were analyzed according to that standard. The Java language has evolved a lot since then, and we analyzed all systems according to the latest standard, which might have caused minor but negligible mistakes in the analysis results.

In Section 6.4.3, we used ckjm 2.2 to analyze the projects included in the Bug

Prediction Dataset. We chose version 2.2 since the original paper did not mark the exact version of ckjm [75], consequently, we experimented with different ckjm versions (1.9, 2.0, 2.1, 2.2), and we experienced version 2.2 to be the best candidate since it produced the smallest differences in metric values compared to the original metric values in the Bug Prediction Dataset.

We used a heuristic method based on name matching to conjugate the elements of the datasets. Although there were cases where the conjugation was unsuccessful, we examined these cases manually, and it turned out that the heuristics worked well and the cause of the problem originated from the differences between the two datasets (all cases are listed in Section 6.3). We examined the successful conjugations as well, and all of them were correct. Even though the heuristics could not handle elements having the same name during the conjugation, only a negligible amount of such cases happened.

Even when the matching heuristics worked well, the same class name could have different meanings in different datasets. For example, OSA handles nested, local, and anonymous classes as different elements, while other datasets did not take into account such elements. Even more, the whole file was associated with its public class. This way, a bug found in a nested or inner class is associated with the public class in the bug datasets, but during the matching, this bug will be associated with the wrong element of the more detailed analysis result of OSA.

6.7 Conclusions

There are several public bug datasets available in the literature which characterize the bugs with static source code metrics. Our aim was to create a public unified bug dataset, which contains all the publicly available ones in a unified format. This dataset can provide researchers with real value by offering a rich bug dataset for their new bug prediction experiments.

We considered five different public bug datasets: the PROMISE dataset, the Eclipse Bug Dataset, the Bug Prediction Dataset, the Bugcatchers Bug Dataset, and the GitHub Bug Dataset. We gave detailed information about each dataset, which contains, among others, their size, enumeration of the included software systems, used version control, and bug tracking systems.

We developed and executed a method on how to create a unified set of bug data, which encapsulates all the information that is available in the datasets. Different datasets use different metric suites; hence we collected the Java source code for all software systems of each dataset and analyzed them with one particular static source code analyzer (OpenStaticAnalyzer) to have a common and uniform set of code metrics (bug predictors) for every system. We constructed the unified bug dataset from the gathered public datasets at the file and class-level and made this unified bug dataset publicly available to anyone for future use.

We investigated the possible differences between the calculated metric values. For this purpose, we ran the ckjm analyzer tool on the Bug Prediction Dataset to have all the metrics from different metric suites. Then, we selected the source code elements which were identified by all three static analyzers. For this subset of source code elements, we calculated the pairwise differences and provided basic statistics for these differences as well. We performed the same steps for the Bugcatchers and the Eclipse bug datasets at the file-level. To see if there is any statistically significant difference, we applied a pairwise *Wilcoxon signed-rank test.* We experienced statistically significant differences in all cases except in the case of NOC (at the class-level between Moose and ckjm) and LLOC (at the file-level between OSA and Eclipse).

We evaluated the datasets according to their metadata and functional criteria. Metadata analysis includes the investigation of the used static analyzer, granularity, bug tracking, and version control system, the set of used metrics, etc. As functional criteria, we compared the bug prediction capabilities of the original metrics, the unified ones, and both together. We used the J48 decision tree algorithm from Weka to build and evaluate bug prediction models per project (within-project learning) in the unified bug dataset. Next, we united the contents of the datasets both at the class (47,618 elements) and file-level (43,744 elements) and evaluated the bug prediction capabilities of this united dataset. This large dataset blurs the deficiencies of the smaller datasets (for example, datasets with more than 90% of buggy source elements). As an additional functional criterion, we used different software systems for training and testing the models, also known as cross-project training. We performed this step on all the systems of the various datasets. Our experiments showed that the unified bug dataset can be used effectively in bug prediction.

We encourage researchers to use this large and public unified bug dataset in their experiments, and we also welcome new public bug datasets.

Contribution

This chapter is based on the publication:

• **Rudolf Ferenc**, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. *A public unified bug dataset for java and its assessment regarding metrics and bug prediction.* Software Quality Journal, 28:1447–1506, 2020. Springer Nature. [12]

I was responsible for the design of the research study and the assessment of bug prediction capabilities of the datasets. The latter includes the implementation of the machine learning experiment and the evaluation of the results.

Some of my other notable papers that were inspired by this result.

• Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, **Rudolf Ferenc**, and Ali Mesbah. *Bugsjs: a benchmark and taxonomy of javascript bugs.* Software Testing, Verification and Reliability, October 2020. Wiley. [15]

• **Rudolf Ferenc**, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. *An automatically created novel bug dataset and its validation in bug prediction.* Journal of Systems and Software, 169:110691, November 2020. Elsevier. [7]

• Péter Hegedűs and **Rudolf Ferenc**. *Static code analysis alarms filtering reloaded: A new real-world dataset and its ml-based utilization.* IEEE Access, 10:55090–55101, 2022. [18]

7 Deep Learning for Bug Prediction

7.1 Introduction

There are many ways to detect software bugs, some of which have already been described. We can use rule-based static or dynamic analysis tools that flag potentially faulty code fragments; we can also use natural language processing, model analysis, and many other techniques. Lu et al. [158] have previously classified bug detection tools in two ways: according to their analysis method, they can be static, dynamic, and model checking; while the other classification is based on how the tools find the bugs, they can be programming-rule-based, statistic-rule-based, or annotation-based tools. This study was carried out in 2005, but a lot has happened since then: the use of deep learning (and artificial intelligence, in general) has grown significantly in recent years.

Deep learning is a new and very successful area in machine learning; the name stems from the fact that it applies deep neural networks (DNNs). These deep networks differ from previously used artificial neural networks in one key aspect, namely that they contain many hidden layers. Unfortunately, with these deep structures, we have to face the fact that the traditional training algorithm encounters difficulties ("vanishing gradient effect") and fails to train good models. As a solution to this problem, several new algorithms and modifications have been proposed over the years. Of these, we opted for one of the simplest ones, the so-called deep rectifier network [104]. With a simple modification to the activation function, the DNN can be trained without any further changes using the standard stochastic gradient descent (SGD) algorithm.

Therefore, we took the Unified Bug Dataset and used it to compare the performance of DNNs to other, more traditional machine learning techniques within the domain of bug prediction – specifically, bug prediction from static source code metrics. We emphasize the interdisciplinary aspect of this experiment by thoroughly detailing every step we took on our way to training our optimal model, including the possible data preprocessing, parameter fine-tuning, and further examinations regarding current or future expectations. Consequently, the coming sections could be a useful tool for static analysts not familiar with deep learning, while the nature and quantity of the data –

along with the conclusions we can draw from them – could provide new insights for machine learning practitioners as well.

Our best deep learning model achieved an F-measure of 53.59% using a dynamically updated learning rate on the quite imbalanced bug dataset, which contains 8,780 (18%) bugged and 38,838 (82%) not bugged Java classes. The only single approach capable of outperforming it was a random forest classifier with an improvement of 0.12%, while an ensemble model combining these two reached an F-measure of 55.27%. Additionally, a separate experiment suggests that these deep learning results could increase even further with more data points, as data quantity seems to be more beneficial for neural networks than it is for other algorithms.

The contributions of our work include:

- A detailed methodology, that serves as an interdisciplinary guideline for merging software quality and machine learning best practices;
- A large-scale case study, that demonstrates the applicability of both deep learning and static source code metrics in bug prediction; and
- An adaptable implementation, that provides replicability, a lower barrier to entry, and facilitates the wider use of deep learning.

7.2 Related Work

Defect prediction has been the focus of numerous research efforts for a long time. In this section, we give a high-level overview of the trends we observed in this field and highlight the differences in our approach.

Bug prediction features. Earlier work concentrated on static source code metrics as the main predictors of software faults, including size, complexity, and object-orientation measures [90, 222, 16, 183, 185]. The common denominator in these approaches is the ability to look at a certain version of the subject system in isolation and the relative ease with which these metrics are computable.

Later research shifted its attention to process-based metrics like added or deleted lines, developer and history-related information, and various aspects of the changes between versions [184, 113, 182, 43, 132]. These features aim to capture bugs as they enter the source code, thereby having to consider only a fraction of the full codebase. In exchange, however, a more complicated data collection process is required.

In this work, we utilize static source code metrics, only combined with deep learning, a pairing that has not been sufficiently explored in our opinion. We also note that more exhaustive surveys of defect detection approaches are published by Menzies et al. [174] and D'Ambros et al. [75].

Bug prediction methods. Once feature selection is decided, the next customization opportunity is the machine learning algorithm used to build the prediction model. There have been previous efforts to adapt Support Vector Machines [218], Decision Trees [226], or Linear Regression [75] to bug prediction. Comparative experiments [101, 206] also incorporate Bayesian models, K Nearest Neighbors, clustering, and ensemble methods. In contrast, we rely on Deep Neural Networks – discussed below – and compare their performance to these more traditional approaches.

Another aspect is the granularity of the collected data and, by extension, the predictions of the model. Many techniques stop at the file level, we – among others – use class-level features, and there are method-level studies as well.

Deep learning and bug prediction. With the advent of more computing performance, deep learning [121] became practically applicable to a wide spectrum of problems. We have seen it succeed in image classification [69, 139], speech recognition [104, 180], natural language processing [178, 214], etc. It is reasonable, then, to try and apply it to the problem of bug prediction as well.

From the previously mentioned features, however, only the change-based ones seem to have "survived" the deep learning paradigm shift [237]. On the other hand, there are multiple recent studies focusing on source code-based tokenization with vector embeddings, approaching the problem from a language processing perspective [227, 201]. Another use for these vector embeddings is bug localization, where the existence of the bug is known beforehand, and the task is automatically pairing it to its corresponding report [143, 144, 125].

Although there are studies where static source code metrics and neural networks appear together, we feel that the relationship is not sufficiently explored. Therefore, our work aims to revitalize the use of static source code metrics for bug prediction by combining it with modern deep learning methodologies and a larger-scale empirical experiment.

A taxonomy of static bug prediction. To focus more exclusively on the closest "neighbors" of our approach, we examined a number of publications in order to build a local taxonomy of differences. The three inclusion criteria were 1) static metric-based methods that are 2) concerned with bugs, and 3) utilizing some type of machine learning. A systematic review led to five aspects of potential variations:

- **Deep Learning**: whether the approach employed deep learning
- **Other Sources**: whether it collected data from sources other than static source code metrics
- **Quantity**: the number of training instances that were available (represented in powers of 10)
- **Granularity**: the level of the source code elements that were considered instances (**M**ethod, **C**lass, or **F**ile)
- **Prediction**: whether there were any actual predictions or only statistical evaluation

The results are presented in Table 7.1.

Table 7.1: A taxonomy of static bug prediction

Aspect	[165]	[194]	[71]	[102]	[126]	[149]	[58]	[38]	[16]	[108]	Our
Deep Learning	✓		✓		✓	✓	✓	✓			✓
Other Sources		✓		✓		✓					
Quantity (10^x)	3	6	2	5	3	3	3	2	3	3	4
Granularity	M	M	F	M	M	F	M	C	C	C	C
Prediction	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓

As the taxonomy shows, the novelty of our work lies in its specific combination of aspects. While there are other studies using class-level granularity, the evaluation is usually on a much smaller scale and does not involve a deep learning-based inference. On the other hand, when there is more data or neural networks are used, the granularity is different. So as far as class-level bug prediction is concerned, this is the largest scale experiment yet, and, to the best of our knowledge, the first ever to investigate actual deep learning prediction. Additionally, none of the works from the table try ensemble models, nor do they consider the possible effects of data quantity.

Since not only our classifier but also our evaluation dataset is new, exact comparisons to other state-of-the-art results are meaningless – even if there were works that would conform to ours in all their taxonomy aspects, which we are not aware of. We would like to note, however, that a matching granularity usually leads to accuracies and F-measures in the same ballpark, while significantly better performances seem to depend on the method-level dataset in question. In the case of [165], for example, a (losing) stock Bayesian network produced better results than our winners, thereby showcasing the meaningful impact of the raw input. From our perspective, the relative performance differences of the various approaches – which can only be measured within an identical context – are much more relevant.

7.3 Methodology

7.3.1 Overview

To complete the experiment outlined in Section 7.1, we first selected an appropriate dataset and applied optional preprocessing techniques (detailed in Section 7.3.2). This was followed by a stratified 10-fold train/dev/test split where the original dataset was split into 10 approximately equal bins in a way that each bin had roughly the same bugged/not bugged distribution as the whole. This allowed us to repeat every potential learning algorithm 10 times, separating a different bin pair for "dev" – a so-called development set reserved for gauging the effect of later hyperparameter tweaks – and "test", respectively. The remaining 8 bins were then merged together to form the training dataset.

In an additional parametric resampling phase, we could even choose to alter the ratio of bugged and not bugged instances – only in the current training set – in the hopes of enhancing the learning procedure. In this case, upsampling meant repeating certain bugged instances to increase their ratio, downsampling meant randomly discarding certain not bugged instances to decrease their ratio, and the amount of resampling meant how much of the gap between the two classes should be closed. Note that while a complete resampling (including even the dev and test sets) is not unheard of in isolated empirical experiments, it does not correctly indicate real-world predictive power, as we have no influence over the distribution of the instances we might see in the future. This distinction should be taken into account when comparing the magnitude of our results to other studies.

After all, these preparations came the actual machine learning through deep neural networks and several other well-known algorithms, which we will discuss in Section 7.3.3. These algorithms have many parameters, and multiple "constellations" were tried for each to find the best-performing models. This arbitrary limiting and potential discretization of parameter values and the evaluation of some (or all) tuples from their Cartesian product is commonly referred to as a *grid search*. Finally, we aggregated, evaluated, and compared the various results based on the principles explained in Section 7.3.4.

7.3.2 Bug Dataset

The basis for any machine learning endeavor is a large and representative dataset. Our choice is the class-level part of the Unified Bug Dataset [11], which contains 47,618 classes. The details of the Unified Bug Dataset were discussed in Chapter 6. To our knowledge, this is the most complete bug database for Java projects, so it was an obvious choice. Furthermore, since we created the database, we are familiar with its capabilities and its structure.

As there are instances where multiple versions of the same project appear, using the dataset as-is could face the issue of "the future predicting the past", where training instances from the more recent state help predict older bugs. We did not treat this as a threat, though, because a) the whole metric-based approach to bug prediction relies on the assumption that the metrics are representative of the underlying faults, so it shouldn't matter where they came from, and b) there can be legitimate causes for trying to use the insight gained in later versions and extrapolate it back to past snapshots of the codebase.

As for preprocessing, the main step preceding every execution was the "binarization" of the labels, i.e., converting the number of bugs found in a class to a boolean false or true (represented by 0 and 1), depending on whether the number was 0 or not, respectively. This can be thought of as making a "bugged" and a "not bugged" class for prediction.

Additional preprocessing options for the features included normalization – where metrics were linearly transformed into the [0,1] interval – and standardization – where each metric was decreased by its mean and divided by its standard deviation, leading to a Gaussian distribution. These transformations can defend against predictors unjustly influencing model decisions just because their range or scale is drastically different. For example, predictor A being a few orders of magnitude larger than predictor B does not automatically mean that A's changes should affect predictions more than B's.

7.3.3 Algorithms and Infrastructure

Once the training dataset is given, machine learning can begin using multiple approaches. These approaches are implemented following the Strategy design pattern to be easily exchangeable and independently parameterizable. Our obvious main goal was to prove the usefulness of deep neural networks – which we attempted with the help of TensorFlow – but we also utilized numerous "traditional" algorithms from the scikit-learn Python package. To be able to experiment quickly, we relied on an NVIDIA Titan Xp graphics card to perform the actual low-level computations. We note, however, that not having access to a dedicated graphics card should not be considered a barrier to entry because a CPU-based execution only makes the experiments slower, not infeasible.

TensorFlow. TensorFlow [30] is an open, computation graph-based machine learning framework that is especially suited for neural networks. Our dependency is on at least the 1.8.0 version, but training can also be run with anything more recent. We followed the setup steps of the `DNNClassifier` class which we later fine-tuned using the Estimator API and custom model functions. One other important requirement was repeatability, so the Estimator's `RunConfig` object always contains an explicitly set random seed.

The structure of the networks we train is always rectangular and dense (fully connected). Initial parameters can set the number of layers, the number of neurons per layer (which is the same for every hidden layer, hence the "rectangular" attribute), the batching (how many instances are processed at a time), and the number of epochs learning should run for. The defaults for these values are 3, 100, 100, and 5, respectively. This algorithm will be referred to as sdnnc, for "**s**imple **d**eep **n**eural **n**etwork **c**lassifier". More complex parameters and approaches are explained as our experiment unfolds step by step in Section 7.4.

scikit-learn To make sure that going through the trouble of configuring and training deep neural networks is actually worth it, we have to compare their results to "easier" – i.e., simpler, more quickly trainable – models. We did so using the excellent scikit-learn 0.19.2 module [196]. The 8 algorithms we included in our study (and the names we will use to refer to them from now on) are: KNeighborsClassifier (knn), GaussianNB (bayes), DecisionTreeClassifier (tree), RandomForestClassifier (forest), LinearRegression (linear), LogisticRegression (logistic), and SVC (svm).

Note that from the above-listed algorithms, LinearRegression is not really a classifier so we did an external binning on the output and determined the prediction bugged if the result was above 0.5. This threshold was not considered a parameter hereafter. Also note that LogisticRegression, despite its name, is indeed a classifier. Finally, each of these models started out with scikit-learn-provided defaults but were later fairly fine-tuned to make their competition with deep neural networks unbiased.

DeepBugHunter. DeepBugHunter is our experimental Python framework collecting the above-mentioned libraries and algorithms into a high abstraction level, parametric tool that makes it easy to either replicate our results or to adapt the approach to other, possibly unrelated fields as well. We provide it as an accompanying, open-source contribution through GitHub [79]. Our experiments were performed using Python 3.6, and dependencies (apart from TensorFlow and scikit-learn) included numpy v1.14.3, scipy v1.0.1, and pandas v0.22.0.

7.3.4 Model Evaluation

As mentioned at the beginning of this section, our main model evaluation strategy is a 10-fold cross-validation. We do not, however, compute accuracy, precision, or recall values independently for any fold but collect and aggregate the raw confusion matrices (the true positive, true negative, false positive, and false negative values). This enables us to calculate the higher-level measures once, at the end. Our primary measure and basis of comparison is the F-measure, but in the case of the best models per algorithm, we calculated additional ROC curves (Receiver Operating Characteristics), AUC values (area under the ROC curve), as well as training and evaluation runtimes (see Section 2.6.3).

We also note that due to the nature of cross-validation, each fold gets a chance to be part of both the development and the test set. This, however, does not mean that information from the test data "leaks" into the hyperparameter tuning phase, as each fold leads to a different model with a separate set of training data.

7.4 Results

This section details the results we achieved, step by step, as we refined our approach.

7.4.1 Preprocessing

The first phase, even before a single machine learning pass, involved examining the available preprocessing strategies. Note that, as mentioned in Section 7.3, the "binarization" of labels is already a given.

Normalization vs. Standardization As a preprocessing step for the 60 features – or, predictors – we compared the results of the default algorithms on the original data (none) vs. normalization and standardization, introduced in Section 7.3.2. A comparison of the techniques is presented in Table 7.2.

Table 7.2: Preprocess method comparison

	none	normalize	**standardize**
knn	44.38%	42.63%	**46.47%**
bayes	34.35%	34.35%	**34.35%**
forest	24.15%	24.15%	**24.13%**
tree	25.95%	25.95%	**25.95%**
linear	21.47%	21.58%	**21.40%**
logistic	23.45%	24.44%	**28.02%**
svm	N/A	9.23%	**9.88%**
sdnnc	19.56%	25.04%	**34.07%**

The results suggest that standardization almost always performs well – as expected from previous empirical experiments. Even when it does not, it is negligibly close, and it is also responsible for the largest improvement in our deep neural network strategy. As there are already many dimensions to cover in our search for the optimal bug prediction model, with much more still to come – and even more we could have added – we decided to finalize this standardization preprocessing step for all further experimentation.

Note that bold font is used to denote our chosen configuration for the given step while italic font (if any) denotes the previous state. Also note that the "N/A" cell for the un-preprocessed svm means that execution had to be shut down after even a single round of the 10-fold cross-validation failed to complete in the allotted timeframe of 12 hours (while in the other 2 cases, an svm fold took mere minutes).

Resampling Similarly to preprocessing, we compared a few resampling amounts in both directions. The results in Table 7.3 show the effect of altering the ratio of bugged and not bugged instances in training set on predicting bugs in an *unaltered* test set. The numbers in the header column represent the percentage of resampling in the given direction, as described in Section 7.3.1.

Table 7.3: Resample method and amount comparison

	down				none		**up**		
	100	75	50	25	*0*	25	**50**	75	100
knn	49.21%	51.10%	49.93%	48.46%	*46.47%*	50.08%	**51.11%**	51.17%	51.04%
bayes	34.70%	34.52%	34.39%	34.38%	*34.35%*	34.39%	**34.62%**	34.65%	34.78%
forest	47.91%	47.67%	41.17%	32.61%	*24.13%*	44.22%	**48.43%**	49.39%	48.15%
tree	46.83%	46.84%	44.28%	34.28%	*25.95%*	45.19%	**47.42%**	48.18%	47.37%
linear	46.34%	43.96%	36.02%	27.60%	*21.40%*	38.89%	**45.57%**	46.70%	46.50%
logistic	46.95%	45.10%	39.22%	33.44%	*28.02%*	41.35%	**46.60%**	47.64%	47.12%
svm	46.49%	41.13%	25.94%	15.69%	*9.88%*	31.00%	**43.85%**	47.12%	46.53%
sdnnc	48.25%	49.32%	46.04%	36.73%	*34.07%*	50.67%	**52.03%**	51.66%	50.59%

We ended up choosing the 50% upsampling because it was the best performing option for our sdnnc strategy and produced comparably good results for the other algorithms as well. Similarly to the above, it is also considered a fixed dimension from here on out so we can concentrate on the actual algorithm-specific hyperparameters. We do note, however, that while it was out of scope for this particular study, replicating the experiments with different resampling amounts definitely merits further research.

7.4.2 Hyperparameter Tuning

Simple Grid Search In our first pass at improving the effectiveness of deep learning, we tried fine-tuning the hyperparameters that were already present in the default implementation, namely the number of layers in the network, the number of neurons per layer (in the hidden layers), and the number of epochs – i.e., the number of times we traverse the whole training set. Note that the activation function of the neurons (rectified linear) and the optimization method (Adagrad) were constant throughout this study, while the batching number could have been varied – and it will be in later stages – but were kept at a fixed 100 at this point. The performances of the different configurations are summarized in Table 7.4, where a better F-measure can help us select the most well-suited hyperparameters.

Table 7.4: Basic hyperparameter search

Layers	Neurons	Epochs	Result
2	100	5	52.01%
3	*100*	*5*	*52.03%*
4	100	5	51.84%
5	100	5	51.83%
5	150	5	52.46%
5	200	5	52.04%
5	200	2	51.26%
5	**200**	**10**	**52.47%**
5	200	20	52.18%

As the F-measures show, the best setup so far is 5 layers of 200 neurons each, learning for 10 epochs. It is important to note, however, that these F-measures are evaluated on the *dev* set, as the performance information they provide can factor into what path we choose in further optimization. Were we to use the test set for this, we would lose the objectivity of our estimations about the model's predictive power, so test evaluations should only happen at the very end.

Initial Learning Rate The next step was to consider the effects of changing the learning rate – i.e., the amount of a new batch of information influences and changes the model's previous opinions. These learning rates are set only once at the beginning of the training process and are fixed until the set number of epochs pass. Their effects on the resulting model's quality are shown in Table 7.5.

Table 7.5: The effect of the initial learning rate

Learning Rate	Result
0.025	52.69%
0.05	**52.70%**
0.1	*52.47%*
0.2	52.36%
0.3	51.76%
0.4	52.37%
0.5	51.87%

As we can see, lowering the learning rate to 0.05 – thereby making the model take "smaller steps" toward its optimum – helped it find a better overall configuration.

Early Stopping and Dynamic Learning Rates Our most dramatic improvement was reached when we introduced validation during training, and instead of learning for a set number of epochs, we implemented early stopping. This meant that after every completed epoch, we evaluated the F-measure of the in-progress model on the development set and checked whether it was an improvement or a deterioration. In the case of deterioration, we reverted the model back to the previous – and, so far, the best – state, halved the learning rate, and tried again; a strategy called "new bob" in the QuickNet framework [128]. We repeated this loop until there were 4 consecutive "misses", signaling that the model seemed unable to learn any further. The rationale behind this approach is that a) we start from a set learning rate and let the model learn while it can, and b) if there is a "misstep", we assume that it happened because the learning rate is now too big and we overshot our target so we should retry the previous step with a lower rate.

The performance impact of this change is meaningful, as shown in Table 7.6. Note that both the above limit of 4 for the consecutive misses and the halving of the learning rates come from previous experience and are considered constant. We will refer to this approach as cdnnc, for "**c**ustomized **d**eep **n**eural **n**etwork **c**lassifier".

Regularization At this point, to decrease the gap between the training and dev F-measures and hopefully increase the model's generalization capabilities, we tried L2 regularization [105]. It is a technique that adds an extra penalty term to the model's loss function in order to discourage large weights and avoid over-fitting.

Table 7.6: The effect of dynamic learning rates

Learning Rate	Result
0.025	53.98%
0.05	54.18%
0.1	**54.48%**
0.2	53.93%
0.3	54.14%
0.4	54.29%
0.5	54.31%

In our case, however, setting the coefficient of the L2 penalty term (denoted by β) to non-zero caused only F-measure degradation (as shown in Table 7.7), so we decided against its use. Note that we also tried β values above 0.05, but those also led to complete model failure.

Table 7.7: The effect of L2 regularization

β	Result
0.0005	54.07%
0.001	53.34%
0.002	52.60%
0.005	51.05%
0.01	49.32%
0.02	43.35%
0.05+	0.00%

Another Round of Hyperparameter Tuning Considering the meaningful jump in quality that cdnnc brought, we found it pertinent to repeat the hyperparameter grid search paired with the early stopping as well, netting us another +0.45% improvement. The tweaked parameters were, again, the number of layers, the number of neurons per layer, the batching amount, and the initial learning rate (that was still halved after every miss). The results, which are also our final results for deep learning in this domain, are summarized in Table 7.8.

The best model we were able to build, then, has 5 layers, each with 250 neurons, gets its input in batches of 100, starts with a learning rate of 0.1, and halves its learning rate after every misstep with backtracking until 4 consecutive misses, thereby producing a 54.93% F-measure on the development set. Having decided to stop refining the model, we could also evaluate it on the test set, resulting in an F-measure of **53.59%**.

Algorithm Comparison To get some perspective on how good the performance of deep learning is, we needed to compare it to similarly fine-tuned versions of the other, more "traditional" algorithms listed in Section 7.3.3. Their possible parameters are listed in the official scikit-learn documentation [196], the method we used to tweak them is the same grid search we utilized for deep learning previously, and the best

Table 7.8: Further hyperparameter tuning

Layers	Neurons	Batch	Learning Rate	Result
4	200	100	0.1	54.77%
6	200	100	0.1	54.33%
5	150	100	0.1	54.67%
5	**250**	**100**	**0.1**	**54.93%**
5	200	50	0.1	54.65%
5	200	150	0.1	54.58%
5	300	100	0.1	54.68%
5	300	100	0.2	54.29%
5	300	100	0.3	54.48%
6	300	100	0.1	54.08%
6	300	100	0.2	54.49%
6	300	100	0.3	54.29%
6	350	100	0.1	54.58%
6	350	100	0.2	54.29%
6	350	100	0.3	54.29%
7	350	100	0.1	54.51%
7	350	100	0.2	53.89%
7	350	100	0.3	53.95%

configurations we found are summarized in Table 7.9 in descending order of their test F-measures. Note that although we used F-measures to guide the optimization procedure, we list additional AUC values belonging to these final models for a more complete evaluation. We also measured model training and test set evaluation times, which are given in the last two columns, respectively.

Table 7.9: The best version of each algorithm

Alg.	Parameters	Train		Dev		Test		Time	
		F-mes.	AUC	F-mes.	AUC	F-mes.	AUC	Train	Eval.
forest	--max-depth 10 --criterion entropy --n-estimators 100	74.38%	89.19%	53.55%	**83.23%**	**53.71%**	**82.98%**	87.7s	0.5s
cdnnc	--layers 5 --neurons 250 --batch 100 --lr 0.1	**79.10%**	**91.16%**	**54.93%**	81.92%	53.59%	81.79%	2132.5s	12.7s
knn	--n_neighbors 18	73.75%	89.17%	52.47%	81.36%	52.40%	81.14%	124.3s	273.2s
svm	--kernel rbf --C 2.6 --gamma 0.02	69.30%	75.87%	52.62%	70.96%	52.25%	70.75%	3142.0s	106.2s
tree	--max-depth 10	72.33%	87.04%	50.26%	77.85%	49.77%	77.34%	11.1s	0.1s
logistic	--penalty l2 --solver liblinear --C 2.0 --tol 0.0001	58.23%	78.28%	46.66%	78.38%	46.43%	78.06%	58.4s	0.1s
linear		57.34%	77.64%	45.57%	77.74%	45.61%	77.47%	3.9s	0.1s
bayes		39.78%	74.36%	34.62%	74.62%	34.84%	74.40%	0.5s	0.1s

The highest generalization on the independent test set goes to the random forest algorithm, although the highest train and dev results belong to our deep learning approach according to both F-measure and AUC figures. The numbers also show a fairly relevant gap between the performance of the two best models (forest and cdnnc) and the rest of the competitors. Additionally, while their evaluation times are at least comparable – with others meaningfully behind – training a neural network is two orders of magnitude slower.

Despite the close second place, the reader might justifiably discard deep learning as a viable option for bug prediction at this point. Why bother with the complex training procedure when a random forest can yield comparable results in a small fraction of the time? In the following, however, we will attempt to show that deep learning can still be useful (in its current form) with the potential to become even better over time.

7.4.3 Ensemble Model

One interesting aspect we noticed when comparing our cdnnc approach to random forest was that although they perform nearly identically in terms of F-score, they arrive there in notably different ways. Taking a look at the separate confusion matrices of the two algorithms in Tables 7.10 and 7.11 shows a non-negligible amount of disagreement between the models. Computing their precision and recall values (shown in the first two columns of Table 7.13) confirm their differences: cdnnc has higher recall (which is arguably more important in bug prediction anyway) at the price of lower precision, while the forest is the exact opposite.

Table 7.10: CDNNC confusion matrix

		Predicted	
		Bugged	Not Bugged
Measured	Bugged	5435	3345
	Not Bugged	6069	32769

Table 7.11: Forest confusion matrix

		Predicted	
		Bugged	Not Bugged
Measured	Bugged	5098	3682
	Not Bugged	5105	33733

This prompted us to try and combine their predictions to see how well they could complement each other as an “ensemble” [188]. The method of the combination was averaging the probabilities of each model assigned to the bugged class and seeing if that average itself was over or under 0.5 – instead of a simple logical *or* on the class outputs. The thinking behind this experiment was that if the two models did learn the same “lessons” from their training, then disregarding deep learning and simply using forest is indeed a reasonable decision. If, on the other hand, they learned different things, their combined knowledge might even surpass those of the individual models. Tables

Table 7.12: Ensemble confusion matrix

		Predicted	
		Bugged	Not Bugged
Measured	Bugged	5360	3420
	Not Bugged	5255	33583

Table 7.13: Comparison of individual and ensemble results

	CDNNC	Forest	Ensemble
Precision	47.24%	49.97%	**50.49%**
Recall	61.90%	58.06%	**61.05%**
F-Measure	53.59%	53.71%	**55.27%**
AUC	81.79%	82.98%	**83.99%**

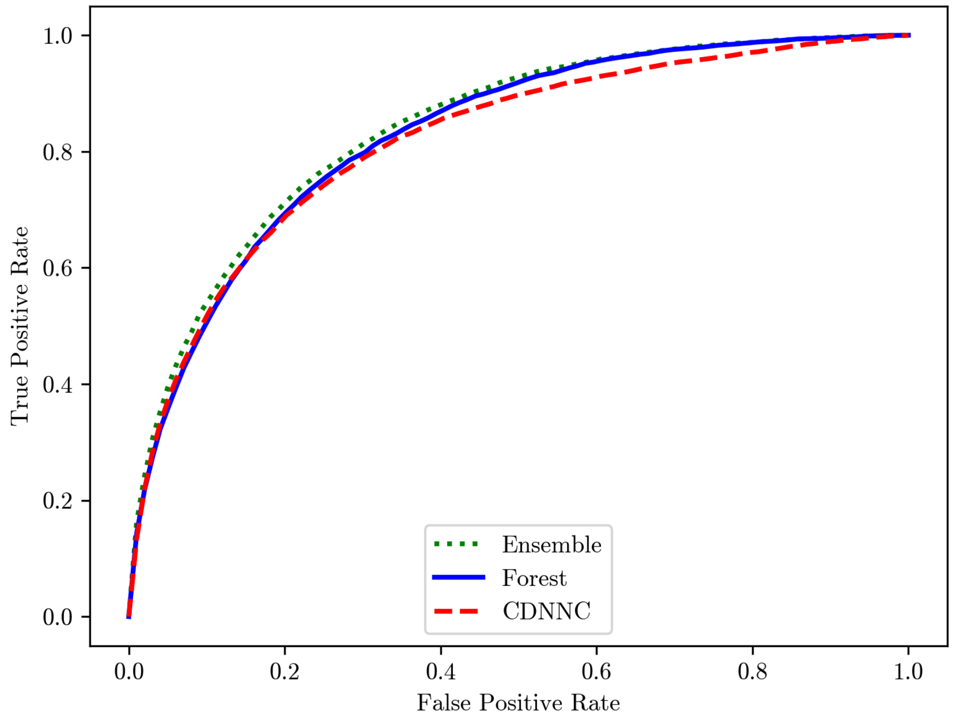


Figure 7.1: ROC comparison for CDNNC, forest, and their ensemble

7.12 and 7.13 attest to the second theory, as the ensemble F-measure reached 55.27% (a 1.56% overall improvement) while the AUC reached 83.99% (a 1.01% improvement).

Moreover, the corresponding ROC curves provide subtle (yet useful) visual support for this theory. As we can see in Figure 7.1, CDNNC and Forest learned differently, hence the differences in their curves. CDNNC slightly outperforms Forest at lower false positive rates, but the relationship is reversed at higher rates. Combining their judgments leads to the dotted Ensemble curve, which outperforms both.

This leads us to believe that deep neural networks might already be useful for bug prediction – even if not by themselves but as parts of a higher-level ensemble model.

7.4.4 The Effect of Data Quantity

Another auxiliary experiment we tried was based on the assumption that "deep learning performs best with large datasets". And by "large", we mean data points in at least the millions. While our dataset cannot be considered small by any measure, – it is the most comprehensive unified bug dataset we are aware of – it is still not on the "large dataset" scale.

The question then became the following: how could we empirically show that deep learning would perform better on more data without actually having more data? The answer we came up with inverts the problem: we theorize that if data quantity is proportional to the "dominance" of a deep learning strategy, then it would also manifest as a faster deterioration than the other algorithms when even fewer data is available. So we artificially shrank – i.e., did a uniform stratified downsampling on – the full dataset three times to produce a 25%, a 50%, and a 75% subset to replicate our whole previous process. The results are summarized in Table 7.14.

Table 7.14: F-Measures across different data quantities

Algorithm	Test results				Relative difference				Normalized relative difference			
	25	50	75	100	25	50	75	100	25	50	75	100
sdnnc	47.66%	50.73%	51.27%	53.37%								
cdnnc	48.19%	51.01%	52.84%	53.59%								
forest	50.02%	51.96%	53.31%	53.71%	-1.83%	-0.95%	-0.47%	-0.12%	0.00%	51.88%	79.92%	100.00%
knn	48.09%	49.55%	50.88%	52.40%	0.10%	1.46%	1.96%	1.19%	0.00%	73.04%	100.00%	58.27%
linear	43.91%	45.55%	45.16%	45.61%	4.28%	5.46%	7.68%	7.98%	0.00%	31.81%	91.93%	100.00%
logistic	44.75%	45.77%	46.02%	46.43%	3.44%	5.24%	6.82%	7.16%	0.00%	48.50%	91.02%	100.00%
svm	47.49%	49.77%	51.80%	52.25%	0.70%	1.24%	1.04%	1.34%	0.00%	83.59%	52.35%	100.00%
tree	45.96%	47.52%	48.82%	49.77%	2.23%	3.49%	4.02%	3.82%	0.00%	70.22%	100.00%	89.02%
bayes	35.48%	35.96%	35.25%	34.84%	12.71%	15.05%	17.59%	18.75%	0.00%	38.80%	80.79%	100.00%

The table consists of three regions, namely the various F-measures evaluated on their test sets (left), the difference between the best deep learning strategy and the current algorithm (middle), and the same difference, only normalized into the [0,1] interval (right). The normalized relative differences are also illustrated in Figure 7.2, where the slope of the lines represents the change in the respective differences. So we track these relative differences over changing dataset sizes, and the *steeper* the incline of the lines, the *less* influence dataset sizes have over their corresponding algorithms compared to neural networks.

An imaginary $y=x$ diagonal line would mean that deep learning is linearly more sensitive to more data, which would lead us to believe that if there were any more data, we could linearly increase our performance. And what we see in Figure 7.2 is not far off from this theoretical indicator. In the case of logistic vs. cdnnc, for example, growth in the differences means that cdnnc is leaving logistic farther and farther behind as more data becomes available. While in the case of forest vs. cdnnc, it means that cdnnc is "catching up" – since the figures are negative, but their absolute values are decreasing.

As most tendencies of the changing differences empirically corroborate, more data is good for every algorithm, but it has a bigger impact on deep learning. Naturally, there are occasional swings like SVM's decrease at 75% – possibly due to the more

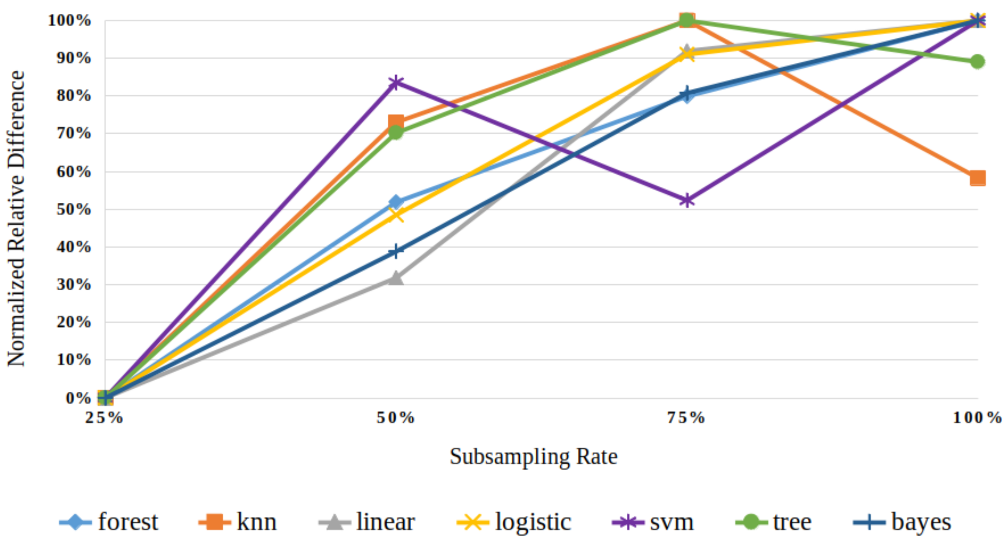


Figure 7.2: The tendencies of the normalized relative differences

"hectic" nature of the technique – or KNN's "hanging tail" at 100%. If we assume a linear kind of relationship, however, even these cases show overall growth. This leads us to speculate that deep neural networks could dominate their opponents – individually, even without resorting to the previously described model combination – when used in conjunction with larger datasets. We also note that scalability should not be an issue, as larger input datasets would affect only the *training* times of the models – which is usually an acceptable up-front sacrifice – while leaving prediction speeds unchanged.

7.5 Threats to Validity

Throughout this study, we aimed to remain as objective as possible by disclosing all our presuppositions and publishing only concrete, replicable results. However, there are still factors that could have skewed the conclusions we drew.

One is the reliability of the bug dataset we used as our input. Building on faulty data will lead to faulty results – also known as the "garbage in, garbage out" principle – but we are confident that this is not the case here. The dataset is independently peer-reviewed, accepted, and compiled using standard data mining techniques.

Another factor might be – ironically – bugs in our bug prediction framework. We tried to combat this through rigorous manual inspections, tests, and replications. Additionally, we are also making the source code openly available on GitHub and inviting community verification or comments.

Yet another factor could be the study dimensions we decided to fix – namely, the preprocessing technique, the preliminary resampling, the number of consecutive misses before stopping early, the 0.5 multiplier for the learning rate "halving", and even the random seed, which was the same for every execution. Analyzing how changes to these parameters would impact the results – if at all – was out of the scope of this study.

Finally, the connections and implications we discovered from the objective figures might just be coincidences. Although there are perfectly logical and reasonable explanations for the unveiled behavior – which we discussed – there is still much to be examined and confirmed in this domain.

7.6 Conclusions

In this Chapter, we presented a detailed approach on how to apply deep neural networks to predict the presence of bugs in classes from static source code metrics alone. While neither deep learning nor bug prediction are new topics in themselves, we aim to benefit their intersection by combining ideas and best practices from both.

Our greatest contribution is the thorough, step-by-step description of our process which – apart from the underexplored coupling of concepts – leads to a deep neural network that is on par with random forests and dominates everything else. Additionally, we unveiled that an ensemble model made from our best deep neural network and forest classifiers is actually better than either of its components individually, – suggesting that deep learning is applicable right now – and that more data is likely to make our approach even better. These are two further convincing arguments supporting the assumption that the increased time and resource requirements of training a deep learning model are worth it. Moreover, we open-sourced the experimental tool we used to reach these conclusions and invite the community to build on our findings.

Contribution

This chapter is based on the publication:

• **Rudolf Ferenc**, Dénes Bán, Tamás Grósz, and Tibor Gyimóthy. *Deep learning in static, metric-based bug prediction.* Array, 6:100021, July 2020. Elsevier. [4]

My main contributions include the design of the research study and the detailed methodology of how we adapt deep neural networks to bug prediction, and their comparison to multiple traditional algorithms.

Some of my other notable papers that contributed to or were inspired by this result:

• **Rudolf Ferenc**. *Bug Forecast: A method for automatic bug prediction.* In Proceedings of the 2010 International Conference on Advanced Software Engineering & Its Applications (ASEA 2010), volume 117 of Communications in Computer and Information Science (CCIS), pages 283–295, Jeju Island, Korea, December 2010. Springer Nature. [3]

• **Rudolf Ferenc**, Tamás Viszkok, Tamás Aladics, Judit Jász, and Péter Hegedűs. *Deep-water framework: The swiss army knife of humans working with machine learning models.* SoftwareX, 12:100551, December 2020. Elsevier. [13]

• **Rudolf Ferenc**, Péter Hegedűs, Péter Gyimesi, Gábor Antal, Bán Dénes, and Tibor Gyimóthy. *Challenging machine learning algorithms in predicting vulnerable javascript functions.* In Proceedings of the 7th IEEE/ACM International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), pages 8–14. IEEE/ACM, May 2019. [8]

8 Summary

In this thesis, we discussed two main topics: applying conceptual cohesion and coupling metrics in fault prediction and in impact analysis; and creating and utilizing a unified bug database using various machine learning and deep learning algorithms. All the presented results have practical relevance and a potential to advance the state of practice.

In the field of conceptual cohesion and coupling metrics, we used identifiers and comments of the source code to extract concepts by feeding them into a latent semantic indexing component. By using principal component analysis, we were able to show that conceptual metrics capture complementary dimensions compared to their structural counterparts. As a consequence, we found that combining structural and conceptual cohesion metrics can improve the accuracy of fault prediction models. Among conceptual coupling metrics, $\mathrm{CCBC_m}$ appeared to be a strong candidate to be applied in impact analysis since it can help obtain a better ranking of classes. Finally, we defined and assessed the new conceptual metrics CLCOM5 and CCBO. We found that combining these two novel metrics with structural ones improved the capabilities of fault prediction models.

In the field of bug datasets, we developed and executed a method on how to combine several sets of bug data, which encapsulates all the information that is available in the previously existing bug datasets. As they used different metric suites, we collected the Java source code for all software systems of each dataset and analyzed them with one particular static source code analyzer to get consistent results. To demonstrate the potential of such a dataset, we analyzed the bug prediction capabilities of the unified dataset. We also presented a detailed, step-by-step description of how to apply deep neural networks to predict the presence of bugs in classes using only static source code metrics. The result of the fine-tuned deep neural network is on par with random forests and dominates any other machine learning algorithm we have used. We unveiled that an ensemble model made from our best deep neural network and forest classifiers is better than either of its components individually and that more data is likely to make our approach even better.

Acknowledgments

This research was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory; and by the national Project no. TKP2021-NVA-09 which has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Appendices

ferenc.rudolf_87_23

A
A Public Unified Bug Dataset for Bug Prediction

A.1 Tables

Table A.1: Merging results (number of elements) – class-level datasets

Dataset	**Name**	**OSA**	**Orig.**	**Dropped**
PROMISE	Ant 1.3	530	125	0
	Ant 1.4	602	178	0
	Ant 1.5	945	293	0
	Ant 1.6	1,262	351	0
	Ant 1.7	1,576	745	0
	Camel 1.0	734	339	0
	Camel 1.2	1,348	608	13 (+5)
	Camel 1.4	2,339	872	0 (+31)
	Camel 1.6	3,174	965	0 (+38)
	Ckjm 1.8	9	10	1
	Forrest 0.6	159	6	0
	Forrest 0.7	76	29	0
	Forrest 0.8	53	32	0
	Ivy 1.4	421	241	0
	Ivy 2.0	637	352	0
	JEdit 3.2	552	272	0
	JEdit 4.0	647	306	0
	JEdit 4.1	722	312	0
	JEdit 4.2	888	367	0
	JEdit 4.3	1,181	492	0
	Log4J 1.0	180	135	0
	Log4J 1.1	217	109	0
	Log4J 1.2	410	205	0
	Lucene 2.0	758	195	1
	Lucene 2.2	1,394	247	1
	Lucene 2.4	1,522	340	1
	Pbeans 1	38	26	0
	Pbeans 2	77	51	0
	Poi 1.5	472	237	0
	Poi 2.0	667	314	0
	Poi 2.5	780	385	0
	Poi 3.0	1,508	442	0
	Synapse 1.0	319	157	0
	Synapse 1.1	491	222	0
	Synapse 1.2	618	256	0
	Velocity 1.4	275	196	0
	Velocity 1.5	377	214	1
	Velocity 1.6	458	229	1
	Xalan 2.4	906	723	0
	Xalan 2.5	992	803	0
	Xalan 2.6	1,217	885	0
	Xalan 2.7	1,249	909	0
	Xerces 1.2	564	440	0
	Xerces 1.3	596	453	0
	Xerces 1.4	782	588	42
Bug Prediction Dataset	Eclipse JDT Core 3.4	2,486	997	0
	Eclipse PDE UI 3.4.1	3,382	1,497	6
	Equinox 3.4	742	324	5
	Lucene 2.4	1,522	691	21
	Mylyn 3.1	3,238	1,862	457
GitHub Bug Dataset	Android U. I. L. 1.7.0	84	73	0
	ANTLR v4 4.2	525	479	0
	Elasticsearch 0.90.11	6,480	5,908	0
	jUnit 4.9	770	731	0
	MapDB 0.9.6	348	331	0
	mcMMO 1.4.06	329	301	0
	MCT 1.7b1	2,050	1,887	0
	Neo4j 1.9.7	6,705	5,899	0
	Netty 3.6.3	1,300	1,143	0
	OrientDB 1.6.2	2,098	1,847	0
	Oryx	562	533	0
	Titan 0.5.1	1,770	1,468	0
	Eclipse p. for Ceylon 1.1.0	1,651	1,610	0
	Hazelcast 3.3	3,765	3,412	0
	Broadleaf C. 3.0.10	2,094	1,593	0
Sum	**All**	**76,623**	**48,242**	**624**

Table A.2: Merging results (number of elements) – file-level datasets

Dataset	**Name**	**OSA**	**Orig.**	**Dropped**
Eclipse Bug Dataset	Eclipse 2.0	6,751	6,729	0
	Eclipse 2.1	7,909	7,888	0
	Eclipse 3.0	10,635	10,593	0
Bugcatchers Bug Dataset	Apache Commons	491	191	0
	ArgoUML 0.26 Beta	1,752	1,582	3
	Eclipse JDT Core 3.1	12,300	560	25
GitHub Bug Dataset	Android U. I. L. 1.7.0	63	63	0
	ANTLR v4 4.2	411	411	0
	Elasticsearch 0.90.11	3,540	3,035	0
	jUnit 4.9	308	308	0
	MapDB 0.9.6	137	137	0
	mcMMO 1.4.06	267	267	0
	MCT 1.7b1	1,064	413	0
	Neo4j 1.9.7	3,291	3,278	0
	Netty 3.6.3	914	913	0
	OrientDB 1.6.2	1,503	1,503	0
	Oryx	443	280	0
	Titan 0.5.1	981	975	0
	Ceylon for Eclipse 1.1.0	699	699	0
	Hazelcast 3.3	2,228	2,228	0
	Broadleaf C. 3.0.10	1,843	1,719	0
Sum	**All**	**57,530**	**43,772**	**28**

Table A.3: Metrics used in the PROMISE dataset

Name	**Abbr.**
Weighted methods per class	WMC
Depth of Inheritance Tree	DIT
Number of Children	NOC
Coupling between object classes	CBO
Response for a Class	RFC
Lack of cohesion in methods	LCOM
Afferent couplings	Ca
Efferent couplings	Ce
Number of Public Methods	NPM
Lack of cohesion in methods (by Henderson-Sellers)	LCOM3
Lines of Code	LOC
Data Access Metric	DAM
Measure of Aggregation	MOA
Measure of Functional Abstraction	MFA
Cohesion Among Methods of Class	CAM
Inheritance Coupling	IC
Coupling Between Methods	CBM
Average Method Complexity	AMC
McCabe's cyclomatic complexity	CC
Maximum McCabe's cyclomatic complexity	MAX_CC
Average McCabe's cyclomatic complexity	AVG_CC
Number of files (compilation units)	NOCU

Table A.4: Metrics used in the Eclipse Bug Dataset

Name	Abbr.
Number of method calls	FOUT
Method lines of code	MLOC
Nested block depth	NBD
Number of parameters	PAR
McCabe cyclomatic complexity	VG
Number of field	NOF
Number of method	NOM
Number of static fields	NSF
Number of static methods	NSM
Number of anonymous type declarations	ACD
Number of interfaces	NOI
Number of classes	NOT
Total lines of code	TLOC
Number of files (compilation units)	NOCU

Table A.5: Metrics used in the Bug Prediction Dataset

Name	Abbr.
Number of other classes that reference the class	FanIn
Number of other classes referenced by the class	FanOut
Number of attributes	NOA
Number of public attributes	NOPA
Number of private attributes	NOPRA
Number of attributes inherited	NOAI
Number of lines of code	LOC
Number of methods	NOM
Number of public methods	NOPM
Number of private methods	NOPRM
Number of methods inherited	NOMI

Table A.6: Bad smells used in the Bugcatchers Bug Dataset

Name	Fowler et al. definitions [99]
Data Clumps	Some data items occur together in lots of places: fields in a couple of classes, parameters in many method signatures.
Message Chains	You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. Navigating this way means the client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to change.
Middle Man	You look at an interface of a class and find that the half of the methods are delegating to this other class. It may mean problems.
Speculative Generality	If the machinery were being used, it would be worth it. But if it isn't, it isn't. The machinery just gets in the way, so get rid of it.
Switch Statements	Switch statements often lead to duplication. Most times you see a switch statement you should consider polymorphism.

Table A.7: Class-level metrics produced by SourceMeter/OpenStaticAnalyzer

Name	**Abbr.**	**Name**	**Abbr.**
API Documentation	AD	Number of Local Public Methods	NLPM
Clone Classes	CCL	Number of Local Setters	NLS
Clone Complexity	CCO	Number of Methods	NM
Clone Coverage	CC	Number of Outgoing Invocations	NOI
Clone Instances	CI	Number of Parents	NOP
Clone Line Coverage	CLC	Number of Public Attributes	NPA
Clone Logical Line Coverage	CLLC	Number of Public Methods	NPM
Comment Density	CD	Number of Setters	NS
Comment Lines of Code	CLOC	Number of Statements	NOS
Coupling Between Object classes	CBO	Public Documented API	PDA
Coupling Between Obj. classes Inv.	CBOI	Public Undocumented API	PUA
Depth of Inheritance Tree	DIT	Response set For Class	RFC
Documentation Lines of Code	DLOC	Total Comment Density	TCD
Lack of Cohesion in Methods 5	LCOM5	Total Comment Lines of Code	TCLOC
Lines of Code	LOC	Total Lines of Code	TLOC
Lines of Duplicated Code	LDC	Total Logical Lines of Code	TLLOC
Logical Lines of Code	LLOC	Total Number of Attributes	TNA
Logical Lines of Duplicated Code	LLDC	Total Number of Getters	TNG
Nesting Level	NL	Total Number of Local Attributes	TNLA
Nesting Level Else-If	NLE	Total Number of Local Getters	TNLG
Number of Ancestors	NOA	Total Number of Local Methods	TNLM
Number of Attributes	NA	Total Number of Local Public Attr.	TNLPA
Number of Children	NOC	Total Number of Local Public Meth.	TNLPM
Number of Descendants	NOD	Total Number of Local Setters	TNLS
Number of Getters	NG	Total Number of Methods	TNM
Number of Incoming Invocations	NII	Total Number of Public Attributes	TNPA
Number of Local Attributes	NLA	Total Number of Public Methods	TNPM
Number of Local Getters	NLG	Total Number of Setters	TNS
Number of Local Methods	NLM	Total Number of Statements	TNOS
Number of Local Public Attributes	NLPA	Weighted Methods per Class	WMC

Table A.8: File-level metrics produced by SourceMeter/OpenStaticAnalyzer

Name	**Abbr.**
McCabe's Cyclomatic Complexity	McCC
Comment Lines of Code	CLOC
Logical Lines of Code	LLOC
Number of Committers	-
Number of developer commits	-
Number of previous modifications	-
Number of previous fixes	-

Table A.9: Basic properties of each dataset

Name	Granularity	SCE	SCEwBug	SCEwBug%	#Bug	kLLOC	Bug/kLine
Ant 1.3	class	125	20	16.00%	33	33	1.00
Ant 1.4	class	178	40	22.47%	47	43	1.09
Ant 1.5	class	293	32	10.92%	35	72	0.49
Ant 1.6	class	351	92	26.21%	184	98	1.88
Ant 1.7	class	745	166	22.28%	338	116	2.91
Camel 1.0	class	339	13	3.83%	14	26	0.54
Camel 1.2	class	590	216	36.61%	522	47	11.11
Camel 1.4	class	841	145	17.24%	335	76	4.41
Camel 1.6	class	928	188	20.26%	500	99	5.05
Ckjm 1.8	class	9	5	55.56%	23	8	2.88
Forrest 0.6	class	6	1	16.67%	1	19	0.05
Forrest 0.7	class	29	5	17.24%	15	4	3.75
Forrest 0.8	class	32	2	6.25%	6	3	2.00
Ivy 1.4	class	241	16	6.64%	18	31	0.58
Ivy 2.0	class	352	40	11.36%	56	54	1.04
JEdit 3.2	class	272	90	33.09%	382	55	6.95
JEdit 4.0	class	306	75	24.51%	226	63	3.59
JEdit 4.1	class	312	79	25.32%	217	72	3.01
JEdit 4.2	class	367	48	13.08%	106	88	1.20
JEdit 4.3	class	492	11	2.24%	12	109	0.11
Log4J 1.0	class	135	34	25.19%	61	10	6.10
Log4J 1.1	class	109	37	33.94%	86	14	6.14
Log4J 1.2	class	205	189	92.20%	498	23	21.65
Lucene 2.0	class	194	91	46.91%	268	68	3.94
Lucene 2.2	class	246	144	58.54%	414	111	3.73
Lucene 2.4	class	340	203	59.71%	632	126	5.02
Pbeans 1	class	26	20	76.92%	36	3	12.00
Pbeans 2	class	51	10	19.61%	19	6	3.17
Poi 1.5	class	237	141	59.49%	342	63	5.43
Poi 2.0	class	314	37	11.78%	39	82	0.48
Poi 2.5	class	385	248	64.42%	496	94	5.28
Poi 3.0	class	442	281	63.57%	500	140	3.57
Synapse 1.0	class	157	16	10.19%	21	20	1.05
Synapse 1.1	class	222	60	27.03%	99	33	3.00
Synapse 1.2	class	256	86	33.59%	145	46	3.15
Velocity 1.4	class	196	147	75.00%	210	26	8.08
Velocity 1.5	class	213	141	66.20%	330	33	10.00
Velocity 1.6	class	228	78	34.21%	190	37	5.14
Xalan 2.4	class	723	110	15.21%	156	104	1.50
Xalan 2.5	class	803	387	48.19%	531	126	4.21
Xalan 2.6	class	885	411	46.44%	625	154	4.06
Xalan 2.7	class	909	898	98.79%	1,213	160	7.58
Xerces 1.2	class	440	71	16.14%	115	65	1.77
Xerces 1.3	class	453	69	15.23%	193	69	2.80
Xerces 1.4	class	547	396	72.39%	1,554	74	21.00
Lucene 2.4	class	670	63	9.40%	96	126	0.76
Mylyn 3.1	class	1,405	209	14.88%	296	166	1.78
PDE UI 3.4.1	class	1,492	208	13.94%	340	186	1.83
Equinox 3.4	class	319	126	39.50%	239	64	3.73
Eclipse JDT Core 3.4	class	997	206	20.66%	374	630	0.59
Apache Commons	file	191	84	43.98%	223	53	4.21
ArgoUML 0.26 Beta	file	1,579	192	12.16%	285	186	1.53
Eclipse JDT Core 3.1	file	535	310	57.94%	567	1,594	0.36
Eclipse 2.0	file	6,729	2,610	38.79%	7,635	792	9.64
Eclipse 2.1	file	7,888	2,139	27.12%	4,975	988	5.04
Eclipse 3.0	file	10,593	2,913	27.50%	7,422	1,307	5.68
Android U. I. L. 1.7.0	class	73	20	27.40%	35	4	8.75
ANTLR v4 4.2	class	479	21	4.38%	27	40	0.68
Broadleaf C. 3.0.10	class	1,593	292	18.33%	353	125	2.82
Eclipse p. for Ceylon 1.1.0	class	1,610	68	4.22%	104	112	0.93
Elasticsearch 0.90.1 1	class	5,908	678	11.48%	1,182	362	3.27
Hazelcast 3.3	class	3,412	380	11.14%	1,053	179	5.88
JUnit 4.9	class	731	35	4.79%	41	16	2.56
MapDB 0.9.6	class	331	40	12.08%	96	47	2.04
mcMMO 1.4.06	class	301	57	18.94%	114	23	4.96
MCT 1.7b1	class	1,887	9	0.48%	9	104	0.09
Neo4j 1.9.7	class	5,899	58	0.98%	60	328	0.18
Netty 3.6.3	class	1,143	271	23.71%	423	76	5.57
OrientDB 1.6.2	class	1,847	280	15.16%	494	184	2.68
Oryx	class	533	74	13.88%	80	29	2.76
Titan 0.5.1	class	1,468	96	6.54%	106	80	1.33
Android U. I. L. 1.7.0	file	63	18	28.57%	33	4	8.25
ANTLR v4 4.2	file	411	41	9.98%	59	40	1.48
Broadleaf C. 3.0.10	file	1,719	286	16.64%	350	125	2.80
Eclipse p. for Ceylon 1.1.0	file	699	57	8.15%	94	112	0.84
Elasticsearch 0.90.11	file	3,035	487	16.05%	899	362	2.48
Hazelcast 3.3	file	2,228	317	14.23%	911	179	5.09
JUnit 4.9	file	308	42	13.64%	50	16	3.13
MapDB 0.9.6	file	137	22	16.06%	59	47	1.26
mcMMO 1.4.06	file	267	57	21.35%	114	23	4.96
MCT 1.7b1	file	413	6	1.45%	6	104	0.06
Neo4j 1.9.7	file	3,278	32	0.98%	33	328	0.10
Netty 3.6.3	file	913	243	26.62%	381	76	5.01
OrientDB 1.6.2	file	1,503	270	17.96%	493	184	2.68
Oryx	file	280	44	15.71%	48	29	1.66
Titan 0.5.1	file	975	70	7.18%	80	80	1.00

Table A.10: F-Measure and AUC values in independent training at class-level

	Original		**OSA**		**Merged**		
Dataset	**F-Measure**	**AUC**	**F-Measure**	**AUC**	**F-Measure**	**AUC**	**SCEwBug%**
Eclipse JDT Core 3.4	0.665	0.687	0.819	0.699	0.817	0.637	9.40
Equinox 3.4	0.727	0.711	0.764	0.793	0.790	0.798	14.88
Lucene 2.4 BPD	0.891	0.635	0.878	0.611	0.879	0.443	13.94
Mylyn 3.1	0.932	0.635	0.806	0.600	0.813	0.610	39.50
PDE UI 3.4.1	0.856	0.638	0.820	0.608	0.816	0.592	20.66
Ant 1.3	0.799	0.603	0.846	0.747	0.827	0.707	16.00
Ant 1.4	0.724	0.681	0.717	0.603	0.726	0.616	22.47
Ant 1.5	0.887	0.579	0.854	0.562	0.871	0.571	10.92
Ant 1.6	0.775	0.665	0.752	0.704	0.745	0.663	26.21
Ant 1.7	0.794	0.727	0.788	0.652	0.788	0.654	22.28
Camel 1.0	0.943	0.433	0.946	0.494	0.932	0.391	3.83
Camel 1.2	0.626	0.624	0.673	0.627	0.686	0.647	36.61
Camel 1.4	0.805	0.637	0.806	0.594	0.814	0.612	17.24
Camel 1.6	0.744	0.597	0.767	0.628	0.776	0.661	20.26
Ckjm	-	-	-	-	-	-	55.56
Forrest 0.6	-	-	-	-	-	-	16.67
Forrest 0.7	0.793	0.737	0.676	0.279	0.676	0.320	17.24
Forrest 0.8	0.891	0.900	0.907	0.100	0.907	0.700	6.25
Ivy 1.4	0.897	0.493	0.915	0.547	0.899	0.511	6.64
Ivy 2.0	0.855	0.624	0.835	0.517	0.847	0.585	11.36
Jedit 3.2	0.727	0.695	0.703	0.710	0.724	0.689	33.09
Jedit 4.0	0.774	0.662	0.766	0.655	0.780	0.654	24.51
Jedit 4.1	0.747	0.640	0.778	0.698	0.781	0.753	25.32
Jedit 4.2	0.866	0.634	0.847	0.650	0.849	0.631	13.08
Jedit 4.3	0.967	0.469	0.965	0.620	0.780	0.608	2.24
Log4J 1.0	0.776	0.637	0.835	0.759	0.781	0.687	25.19
Log4J 1.1	0.739	0.681	0.743	0.718	0.849	0.696	33.94
Log4J 1.2	0.892	0.621	0.885	0.421	0.963	0.495	92.20
Lucene 2.0	0.610	0.571	0.659	0.635	0.634	0.626	46.91
Lucene 2.2	0.584	0.631	0.655	0.657	0.661	0.638	58.54
Lucene 2.4	0.698	0.689	0.656	0.688	0.663	0.676	59.71
Pbeans 1	0.813	0.633	0.769	0.741	0.813	0.779	76.92
Pbeans 2	0.727	0.565	0.686	0.503	0.740	0.553	19.61
Poi 1.5	0.743	0.744	0.753	0.747	0.782	0.778	59.49
Poi 2.0	0.851	0.465	0.841	0.729	0.831	0.576	11.78
Poi 2.5	0.787	0.792	0.809	0.788	0.789	0.791	64.42
Poi 3.0	0.768	0.763	0.768	0.768	0.764	0.759	63.57
Synapse 1.0	0.830	0.524	0.843	0.531	0.839	0.576	10.19
Synapse 1.1	0.758	0.683	0.767	0.717	0.749	0.672	27.03
Synapse 1.2	0.738	0.670	0.734	0.677	0.733	0.663	33.59
Velocity 1.4	0.827	0.744	0.824	0.733	0.856	0.796	75.00
Velocity 1.5	0.706	0.660	0.728	0.696	0.778	0.742	66.20
Velocity 1.6	0.703	0.649	0.774	0.783	0.748	0.730	34.21
Xalan 2.4	0.802	0.621	0.807	0.645	0.794	0.562	15.21
Xalan 2.5	0.654	0.648	0.693	0.686	0.699	0.718	48.19
Xalan 2.6	0.751	0.747	0.730	0.707	0.747	0.750	46.44
Xalan 2.7	0.994	0.654	0.992	0.820	0.992	0.745	98.79
Xerces 1.2	0.815	0.716	0.824	0.683	0.823	0.716	16.14
Xerces 1.3	0.860	0.629	0.837	0.625	0.836	0.743	15.23
Xerces 1.4	0.938	0.904	0.804	0.815	0.932	0.935	72.39
Average	**0.793**	**0.653**	**0.793**	**0.646**	**0.798**	**0.656**	-
Android U. I. L. 1.7.0	0.749	0.657	0.742	0.677	-	-	27.40
ANTLR v4 4.2	0.944	0.618	0.922	0.555	-	-	4.38
Broadleaf C. 3.0.10	0.898	0.826	0.881	0.796	-	-	18.33
Eclipse p. for Ceylon 1.1.0	0.942	0.611	0.939	0.586	-	-	4.22
Elasticsearch 0.90.11	0.871	0.645	0.874	0.683	-	-	11.48
Hazelcast 3.3	0.876	0.693	0.878	0.658	-	-	11.14
jUnit 4.9	0.927	0.579	0.928	0.624	-	-	4.79
MapDB 0.9.6	0.911	0.659	0.886	0.673	-	-	12.08
mcMMO 1.4.06	0.791	0.593	0.776	0.568	-	-	18.94
MCT 1.7b1	0.993	0.480	0.993	0.478	-	-	0.48
Neo4j 1.9.7	0.985	0.494	0.985	0.486	-	-	0.98
Netty 3.6.3	0.809	0.699	0.802	0.699	-	-	23.71
OrientDB 1.6.2	0.868	0.687	0.862	0.715	-	-	15.16
Oryx	0.889	0.764	0.898	0.725	-	-	13.88
Titan 0.5.1	0.922	0.696	0.926	0.706	-	-	6.54
Average	**0.892**	**0.647**	**0.886**	**0.642**	**-**		-

Table A.11: F-Measure and AUC values in independent training at file-level

	Original		**OSA**		**Merged**		
Dataset	**F-Measure**	**AUC**	**F-Measure**	**AUC**	**F-Measure**	**AUC**	**SCEwBug%**
Apache Commons	0.779	0.733	0.454	0.536	0.712	0.778	43.98
Argo UML 0.26 Beta	0.899	0.730	0.832	0.677	0.880	0.719	12.16
Eclipse JDT Core 3.1	0.827	0.657	0.528	0.567	0.638	0.658	57.94
Eclipse 2.0	0.682	0.654	0.670	0.713	0.694	0.660	38.79
Eclipse 2.1	0.749	0.623	0.750	0.711	0.745	0.639	27.12
Eclipse 3.0	0.728	0.597	0.735	0.694	0.727	0.603	27.50
Average	**0.777**	**0.666**	**0.662**	**0.650**	**0.733**	**0.676**	**-**
Android U. I. L. 1.7.0	0.611	0.484	0.624	0.654	-	-	28.57
ANTLR v4 4.2	0.900	0.795	0.849	0.753	-	-	9.98
Broadleaf C. 3.0.10	0.862	0.759	0.823	0.795	-	-	16.64
Eclipse p. for Ceylon 1.1.0	0.879	0.639	0.874	0.574	-	-	8.15
Elasticsearch 0.90.11	0.775	0.669	0.778	0.678	-	-	16.05
Hazelcast 3.3	0.829	0.688	0.796	0.626	-	-	14.23
jUnit 4.9	0.800	0.585	0.801	0.592	-	-	13.64
MapDB 0.9.6	0.784	0.686	0.766	0.654	-	-	16.06
mcMMO 1.4.06	0.731	0.669	0.794	0.614	-	-	21.35
MCT 1.7b	0.977	0.291	0.978	0.551	-	-	1.45
Neo4j 1.9.7	0.985	0.474	0.985	0.474	-	-	0.98
Netty 3.6.3	0.677	0.700	0.732	0.762	-	-	26.62
OrientDB 1.6.2	0.739	0.759	0.751	0.722	-	-	17.96
Oryx	0.771	0.467	0.861	0.735	-	-	15.71
Titan 0.5.1	0.894	0.498	0.894	0.498	-	-	7.18
Average	**0.814**	**0.611**	**0.820**	**0.646**	**-**	**-**	**-**

Table A.12: Cross training (PROMISE - class-level) - F-Measure values

Project	SCEwBug%	Avg. F-m.	Ant 1.3	Camel 1.0	Ckjm 1.8	Forrest 0.6	Ivy 1.4	Jedit 3.2	Log4J 1.0	Log4J 1.2	Lucene 2.0	Lucene 2.2	Pbeans 1	Pbeans 2	Poi 1.5	Poi 2.0	Synapse 1.0	Velocity 1.4	Velocity 1.6	Xalan 2.4	Xalan 2.7	Xerces 1.2	Xerces 1.4
Ant 1.3	16.0	0.71		0.830	0.776	0.776	0.785	0.784	0.794	0.762	0.754	0.739	0.718	0.718	0.706	0.711	0.674	0.660	0.648	0.659	0.590	0.596	0.588
Camel 1.0	3.8	0.66	0.755		0.719	0.719	0.732	0.731	0.747	0.711	0.701	0.684	0.660	0.661	0.648	0.655	0.618	0.604	0.593	0.608	0.524	0.532	0.524
Ckjm 1.8	*55.6*	*0.49*	0.307	0.442		0.488	0.481	0.482	0.496	0.495	0.496	0.497	0.500	0.500	0.503	0.493	0.507	0.514	0.513	0.511	0.510	0.505	0.501
Forrest 0.6	16.7	0.63	0.767	0.741	0.697		0.712	0.710	0.728	0.691	0.681	0.663	0.637	0.638	0.623	0.631	0.587	0.573	0.562	0.576	0.487	0.496	0.487
Ivy 1.4	6.6	0.67	0.807	0.756	0.714	0.714		0.735	0.748	0.715	0.707	0.692	0.671	0.672	0.658	0.665	0.622	0.610	0.601	0.613	0.533	0.540	0.532
Jedit 3.2	33.2	0.69	0.733	0.746	0.719	0.718	0.728		0.750	0.725	0.721	0.711	0.696	0.697	0.691	0.691	0.678	0.665	0.655	0.664	0.584	0.589	0.579
Log4J 1.0	25.2	0.70	0.753	0.731	0.722	0.722	0.733	0.734		0.723	0.718	0.713	0.703	0.704	0.696	0.697	0.677	0.672	0.668	0.671	0.624	0.627	0.621
Log4J 1.2	**92.2**	**0.20**	0.060	0.118	0.180	0.179	0.177	0.179	0.163		0.183	0.189	0.200	0.199	0.205	0.198	0.220	0.230	0.236	0.224	0.271	0.268	0.273
Lucene 2.0	46.9	0.67	0.750	0.718	0.696	0.696	0.697	0.698	0.700	0.680		0.680	0.672	0.673	0.671	0.664	0.665	0.655	0.651	0.656	0.619	0.620	0.615
Lucene 2.2	*58.5*	*0.47*	0.311	0.369	0.428	0.428	0.446	0.463	0.476	0.477	0.480		0.496	0.496	0.497	0.496	0.495	0.495	0.494	0.483	0.497	0.488	0.485
Pbeans 1	**76.9**	**0.36**	0.320	0.319	0.352	0.353	0.350	0.345	0.329	0.336	0.341	0.346		0.358	0.363	0.358	0.375	0.378	0.386	0.378	0.401	0.393	0.395
Pbeans 2	19.6	0.71	0.737	0.769	0.747	0.747	0.759	0.755	0.767	0.743	0.737	0.728	0.714		0.709	0.709	0.687	0.676	0.668	0.674	0.620	0.623	0.614
Poi 1.5	*59.5*	*0.55*	0.394	0.538	0.555	0.555	0.550	0.554	0.578	0.571	0.569	0.567	0.564	0.564		0.569	0.575	0.576	0.574	0.572	0.562	0.557	0.554
Poi 2.0	11.8	0.66	0.734	0.742	0.705	0.705	0.714	0.712	0.726	0.699	0.691	0.678	0.663	0.664	0.652		0.630	0.621	0.614	0.622	0.560	0.565	0.557
Synapse 1.0	10.2	0.65	0.772	0.745	0.706	0.706	0.720	0.720	0.734	0.699	0.691	0.675	0.652	0.652	0.638	0.645		0.594	0.582	0.596	0.509	0.516	0.509
Velocity 1.4	**75.0**	**0.27**	0.223	0.209	0.232	0.233	0.229	0.229	0.252	0.265	0.267	0.271	0.277	0.277	0.280	0.283	0.298		0.323	0.316	0.347	0.343	0.343
Velocity 1.6	34.2	0.67	0.700	0.693	0.697	0.698	0.707	0.707	0.697	0.681	0.677	0.669	0.660	0.661	0.660	0.661	0.645	0.641		0.642	0.624	0.620	0.614
Xalan 2.4	15.2	0.69	0.792	0.767	0.718	0.718	0.732	0.741	0.760	0.728	0.721	0.705	0.684	0.685	0.677	0.683	0.655	0.642	0.632		0.599	0.604	0.594
Xalan 2.7	**98.8**	**0.10**	0.044	0.054	0.073	0.073	0.066	0.068	0.062	0.078	0.083	0.092	0.105	0.105	0.113	0.109	0.134	0.142	0.149	0.142		0.199	0.205
Xerces 1.2	16.1	0.64	0.734	0.723	0.677	0.677	0.692	0.693	0.711	0.681	0.671	0.655	0.632	0.632	0.618	0.626	0.587	0.576	0.567	0.579	0.513		0.519
Xerces 1.4	**72.4**	**0.29**	0.353	0.280	0.278	0.279	0.277	0.276	0.261	0.271	0.273	0.278	0.286	0.287	0.292	0.288	0.302	0.306	0.312	0.314	0.343	0.343	

Table A.13: Cross training (PROMISE - class-level) - AUC values

Project	SCEwBug%	Avg. AUC	Ant 1.3	Camel 1.0	Ckjm 1.8	Forrest 0.6	Ivy 1.4	Jedit 3.2	Log4J 1.0	Log4J 1.2	Lucene 2.0	Lucene 2.2	Pbeans 1	Pbeans 2	Poi 1.5	Poi 2.0	Synapse 1.0	Velocity 1.4	Velocity 1.6	Xalan 2.4	Xalan 2.7	Xerces 1.2	Xerces 1.4
ant-1.3	16.0	0.524		0.395	0.500	0.500	0.812	0.578	0.456	0.476	0.510	0.468	0.554	0.444	0.540	0.535	0.513	0.538	0.509	0.552	0.532	0.505	0.560
camel-1.0	3.8	0.537	0.516		0.500	0.500	0.658	0.506	0.657	0.556	0.544	0.565	0.421	0.638	0.476	0.497	0.511	0.504	0.515	0.580	0.550	0.538	0.512
ckjm-1.8	*55.6*	0.585	0.520	0.656		0.200	0.558	0.606	0.699	0.703	0.610	0.583	0.575	0.633	0.642	0.543	0.620	0.561	0.573	0.620	0.783	0.495	0.522
forrest-0.6	16.7	0.500	0.500	0.500	0.500		0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
ivy-1.4	6.6	0.629	0.767	0.576	0.600	0.500		0.690	0.692	0.587	0.661	0.607	0.575	0.676	0.626	0.648	0.690	0.523	0.666	0.728	0.676	0.523	0.571
jedit-3.2	33.2	0.580	0.707	0.593	0.650	0.400	0.719		0.762	0.584	0.634	0.545	0.525	0.621	0.606	0.549	0.650	0.499	0.489	0.550	0.455	0.578	0.489
log4j-1.0	25.2	0.574	0.695	0.475	0.400	0.400	0.523	0.568		0.575	0.623	0.636	0.550	0.674	0.510	0.668	0.516	0.563	0.604	0.694	0.594	0.641	0.574
log4j-1.2	**92.2**	0.483	0.531	0.446	0.475	0.600	0.510	0.493	0.533		0.552	0.562	0.467	0.344	0.499	0.470	0.504	0.398	0.454	0.490	0.408	0.495	0.422
lucene-2.0	46.9	0.592	0.781	0.594	0.500	0.500	0.814	0.684	0.546	0.390		0.609	0.575	0.534	0.641	0.578	0.564	0.451	0.569	0.699	0.689	0.571	0.556
lucene-2.2	*58.5*	0.614	0.538	0.659	0.900	0.600	0.637	0.532	0.680	0.527	0.648		0.850	0.463	0.626	0.584	0.622	0.630	0.539	0.597	0.712	0.398	0.541
pbeans-1	**76.9**	0.558	0.595	0.564	0.625	0.300	0.589	0.582	0.585	0.469	0.598	0.580		0.705	0.600	0.572	0.526	0.514	0.619	0.571	0.559	0.465	0.550
pbeans-2	19.6	0.543	0.511	0.454	0.500	0.600	0.561	0.565	0.533	0.443	0.578	0.580	0.625		0.579	0.565	0.521	0.488	0.589	0.587	0.565	0.527	0.485
poi-1.5	*59.5*	0.551	0.551	0.598	0.875	0.400	0.426	0.553	0.646	0.583	0.517	0.528	0.288	0.570		0.447	0.620	0.488	0.574	0.566	0.722	0.430	0.635
poi-2.0	11.8	0.532	0.546	0.665	0.500	0.400	0.722	0.444	0.487	0.388	0.495	0.542	0.538	0.761	0.544		0.459	0.683	0.592	0.454	0.492	0.486	0.436
synapse-1.0	10.2	0.509	0.526	0.556	0.700	0.400	0.326	0.489	0.624	0.537	0.487	0.567	0.475	0.513	0.534	0.540		0.525	0.474	0.509	0.491	0.426	0.488
velocity-1.4	**75.0**	0.485	0.444	0.332	0.600	0.900	0.317	0.442	0.642	0.596	0.453	0.474	0.625	0.393	0.431	0.442	0.473		0.533	0.333	0.427	0.411	0.429
velocity-1.6	34.2	0.592	0.499	0.630	0.825	0.000	0.670	0.573	0.681	0.559	0.628	0.617	0.775	0.533	0.652	0.540	0.709	0.470		0.619	0.820	0.461	0.580
xalan-2.4	15.2	0.605	0.728	0.541	0.600	0.500	0.501	0.697	0.743	0.571	0.649	0.560	0.600	0.677	0.644	0.646	0.687	0.468	0.618		0.619	0.527	0.516
xalan-2.7	**98.8**	0.511	0.514	0.522	0.500	0.500	0.507	0.544	0.535	0.487	0.469	0.516	0.475	0.561	0.535	0.518	0.514	0.524	0.494	0.511		0.515	0.484
xerces-1.2	16.1	0.493	0.471	0.570	0.250	0.500	0.683	0.518	0.543	0.476	0.451	0.479	0.521	0.570	0.410	0.562	0.480	0.512	0.541	0.508	0.348		0.464
xerces-1.3	**72.4**	0.529	0.595	0.511	0.575	0.000	0.629	0.594	0.518	0.436	0.550	0.506	0.721	0.644	0.581	0.527	0.582	0.498	0.525	0.533	0.399	0.547	0.636

Table A.14: Cross training (GitHub – class-level) - F-Measure values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elasticsearch	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.885	0.772	0.822	0.813	0.821	0.827	0.827	0.825	0.836	0.868	0.859	0.852	0.850	0.853
ANTLR v4	0.611		0.785	0.852	0.843	0.842	0.847	0.847	0.845	0.862	0.893	0.882	0.876	0.874	0.876
Broadleaf Commerce	0.635	0.877		0.928	0.870	0.859	0.863	0.862	0.860	0.873	0.894	0.886	0.880	0.879	0.879
Eclipse p. for Ceylon	0.611	0.894	0.781		0.847	0.847	0.851	0.851	0.848	0.864	0.895	0.884	0.879	0.877	0.879
Elasticsearch	0.642	0.868	0.835	0.875		0.900	0.902	0.900	0.897	0.904	0.914	0.904	0.897	0.895	0.895
Hazelcast	0.635	0.876	0.792	0.831	0.828		0.864	0.863	0.861	0.871	0.888	0.879	0.872	0.871	0.872
jUnit	0.644	0.849	0.774	0.837	0.819	0.813		0.822	0.821	0.835	0.867	0.858	0.854	0.853	0.854
MapDB	0.670	0.852	0.799	0.855	0.852	0.853	0.857		0.858	0.871	0.894	0.884	0.879	0.877	0.878
mcMMO	0.642	0.879	0.793	0.855	0.845	0.849	0.853	0.853		0.866	0.888	0.880	0.874	0.873	0.875
Mission Control T.	0.611	0.890	0.773	0.843	0.836	0.836	0.840	0.840	0.838		0.890	0.879	0.872	0.870	0.872
Neo4j	0.611	0.890	0.774	0.843	0.836	0.836	0.841	0.840	0.838	0.856		0.878	0.871	0.869	0.871
Netty	0.644	0.873	0.787	0.848	0.834	0.831	0.837	0.836	0.834	0.847	0.874		0.869	0.867	0.868
OrientDB	0.611	0.845	0.800	0.857	0.835	0.841	0.846	0.846	0.845	0.859	0.888	0.878		0.882	0.882
Oryx	0.712	0.874	0.787	0.845	0.837	0.840	0.845	0.845	0.843	0.857	0.879	0.870	0.865		0.868
Titan	0.611	0.895	0.787	0.849	0.840	0.843	0.847	0.847	0.845	0.859	0.890	0.880	0.874	0.872	

Table A.15: Cross training (GitHub – class-level) - AUC values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elasticsearch	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.257	0.282	0.335	0.415	0.390	0.559	0.361	0.428	0.706	0.464	0.504	0.382	0.664	0.515
ANTLR v4	0.578		0.548	0.699	0.641	0.624	0.548	0.713	0.672	0.792	0.541	0.610	0.664	0.548	0.641
Broadleaf Commerce	0.541	0.748		0.635	0.633	0.649	0.572	0.667	0.604	0.753	0.564	0.624	0.622	0.589	0.644
Eclipse p. for Ceylon	0.537	0.532	0.439		0.503	0.553	0.538	0.558	0.518	0.486	0.520	0.480	0.584	0.502	0.490
Elasticsearch	0.637	0.504	0.556	0.484		0.523	0.505	0.509	0.511	0.656	0.488	0.563	0.571	0.525	0.531
Hazelcast	0.341	0.662	0.412	0.615	0.547		0.486	0.531	0.493	0.494	0.479	0.463	0.564	0.671	0.349
jUnit	0.571	0.730	0.466	0.443	0.526	0.514		0.645	0.668	0.821	0.502	0.577	0.494	0.428	0.538
MapDB	0.542	0.685	0.619	0.654	0.589	0.597	0.713		0.626	0.694	0.608	0.541	0.592	0.577	0.529
mcMMO	0.576	0.523	0.532	0.584	0.637	0.673	0.694	0.611		0.814	0.608	0.657	0.589	0.666	0.584
Mission Control T.	0.491	0.536	0.530	0.554	0.515	0.527	0.513	0.541	0.524		0.514	0.524	0.537	0.506	0.517
Neo4j	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500		0.500	0.500	0.500	0.500
Netty	0.539	0.609	0.451	0.522	0.512	0.523	0.506	0.487	0.537	0.603	0.553		0.535	0.522	0.501
OrientDB	0.485	0.650	0.555	0.577	0.485	0.508	0.511	0.495	0.552	0.643	0.511	0.494		0.441	0.490
Oryx	0.637	0.675	0.534	0.668	0.607	0.573	0.596	0.628	0.558	0.420	0.612	0.528	0.506		0.581
Titan	0.617	0.673	0.510	0.440	0.492	0.547	0.572	0.367	0.404	0.181	0.530	0.519	0.464	0.495	

Table A.16: Cross training (GitHub – file-level) - F-Measure values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elasticsearch	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.791	0.736	0.724	0.707	0.724	0.727	0.726	0.724	0.725	0.753	0.746	0.743	0.743	0.749
ANTLR v4	0.595		0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Broadleaf Commerce	0.631	0.832		0.842	0.803	0.803	0.803	0.803	0.800	0.806	0.848	0.835	0.826	0.825	0.829
Eclipse p. for Ceylon	0.595	0.841	0.797		0.804	0.805	0.805	0.806	0.803	0.809	0.850	0.835	0.829	0.828	0.832
Elasticsearch	0.595	0.817	0.771	0.797		0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Hazelcast	0.595	0.819	0.775	0.799	0.783		0.791	0.790	0.788	0.796	0.843	0.828	0.819	0.818	0.822
jUnit	0.722	0.813	0.759	0.767	0.759	0.766		0.768	0.766	0.771	0.802	0.793	0.788	0.788	0.793
MapDB	0.622	0.847	0.808	0.822	0.813	0.814	0.815		0.814	0.819	0.855	0.843	0.837	0.836	0.839
mcMMO	0.710	0.806	0.800	0.786	0.770	0.780	0.782	0.781		0.780	0.791	0.784	0.779	0.778	0.782
Mission Control T.	0.595	0.821	0.773	0.797	0.782	0.785	0.785	0.785	0.782		0.841	0.826	0.818	0.817	0.821
Neo4j	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790		0.825	0.816	0.815	0.820
Netty	0.682	0.818	0.773	0.795	0.774	0.783	0.784	0.784	0.784	0.789	0.827		0.818	0.817	0.819
OrientDB	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825		0.815	0.820
Oryx	0.637	0.830	0.768	0.795	0.787	0.789	0.791	0.790	0.787	0.794	0.839	0.825	0.816		0.822
Titan	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	

Table A.17: Cross training (GitHub – file-level) - AUC values

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elasticsearch	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.		0.468	0.271	0.355	0.426	0.433	0.637	0.341	0.465	0.271	0.391	0.534	0.374	0.665	0.339
ANTLR v4	0.649		0.682	0.710	0.667	0.631	0.634	0.732	0.727	0.777	0.687	0.621	0.627	0.538	0.648
Broadleaf Commerce	0.537	0.598		0.729	0.675	0.641	0.549	0.684	0.752	0.667	0.570	0.604	0.644	0.598	0.632
Eclipse p. for Ceylon	0.517	0.593	0.528		0.565	0.549	0.500	0.680	0.592	0.725	0.541	0.538	0.574	0.512	0.543
Elasticsearch	0.598	0.582	0.679	0.626		0.637	0.559	0.752	0.699	0.295	0.616	0.607	0.622	0.553	0.623
Hazelcast	0.481	0.460	0.579	0.652	0.545		0.500	0.597	0.534	0.209	0.523	0.496	0.564	0.537	0.498
jUnit	0.707	0.723	0.732	0.658	0.665	0.647		0.598	0.681	0.653	0.721	0.669	0.642	0.634	0.630
MapDB	0.327	0.462	0.493	0.643	0.576	0.547	0.368		0.574	0.633	0.463	0.511	0.529	0.486	0.481
mcMMO	0.608	0.799	0.655	0.685	0.684	0.640	0.563	0.806		0.814	0.728	0.611	0.622	0.519	0.600
Mission Control T.	0.450	0.534	0.492	0.506	0.485	0.497	0.463	0.472	0.535		0.496	0.474	0.499	0.535	0.511
Neo4j	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500		0.500	0.500	0.500	0.500
Netty	0.656	0.686	0.733	0.678	0.673	0.652	0.698	0.746	0.698	0.856	0.753		0.693	0.784	0.733
OrientDB	0.573	0.686	0.718	0.559	0.661	0.603	0.600	0.794	0.640	0.786	0.737	0.736		0.611	0.750
Oryx	0.682	0.724	0.725	0.663	0.644	0.610	0.658	0.595	0.664	0.578	0.689	0.703	0.613		0.648
Titan	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	

A.2 Online Appendix

The Unified Bug Dataset 1.2 is available as an online appendix at the following links:

- `https://doi.org/10.5281/zenodo.3693685`
- `http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet`

The *UnifiedBugDataset-1.2.zip* file contains

- the original bug datasets in their original form,
- the list of projects contained in each dataset,
- the source code of the systems that were used to develop the datasets,
- the unified dataset in CSV and ARFF format at file/class-level,
- the dataset containing only the results of OpenStaticAnalyzer in ARFF format at file/class-level,
- description of the OpenStaticAnalyzer metrics,
- metrics comparisons in spreadsheet format of the PROMISE [173], Eclipse [247], Bug Prediction [75], Bugcatchers [111], and GitHub [26] bug datasets,
- the results of the within-project training at file/class-level in spreadsheet format,
- the results of the cross-training at file/class-level in spreadsheet format.

For a more exhaustive description of the exact contents of the files and usage information, one should refer to the '`README.txt`' file, which is located in the root folder of the Unified Bug Dataset package.

Bibliography

Referenced Publications of the Author

[1] Tibor Bakota, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A probabilistic software quality model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*, pages 243–252, Williamsburg, VA, USA, September 2011. IEEE Computer Society.

[2] Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc. Code ownership: Impact on maintainability. In *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA 2015)*, volume 9159 of *Lecture Notes in Computer Science (LNCS)*, pages 3–19, Banff, Alberta, Canada, June 2015. Springer-Verlag.

[3] Rudolf Ferenc. Bug Forecast: A method for automatic bug prediction. In *Proceedings of the 2010 International Conference on Advanced Software Engineering & Its Applications (ASEA 2010)*, volume 117 of *Communications in Computer and Information Science (CCIS)*, pages 283–295, Jeju Island, Korea, December 2010. Springer-Verlag.

[4] Rudolf Ferenc, Dénes Bán, Tamás Grósz, and Tibor Gyimóthy. Deep learning in static, metric-based bug prediction. *Array*, 6:100021, July 2020. Open Access.

[5] Rudolf Ferenc, Árpád Beszédes, Lajos Jenő Fülöp, and János Lele. Design pattern mining enhanced by machine learning. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 295–304, Budapest, Hungary, September 2005. IEEE Computer Society.

[6] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181, Montréal, Canada, October 2002. IEEE Computer Society.

[7] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software*, 169:110691, November 2020. Open Access.

[8] Rudolf Ferenc, Péter Hegedűs, Péter Gyimesi, Gábor Antal, Bán Dénes, and Tibor Gyimóthy. Challenging machine learning algorithms in predicting vulnerable javascript functions. In *Proceedings of the 7th IEEE/ACM International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 8–14. IEEE/ACM, May 2019.

[9] Rudolf Ferenc, László Langó, István Siket, Tibor Gyimóthy, and Tibor Bakota. SourceMeter SonarQube plug-in. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 77–82, Victoria, British Columbia, Canada, September 2014. IEEE Computer Society.

[10] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from open source software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69, Chicago Illinois, USA, September 2004. IEEE Computer Society.

[11] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 12–21. ACM, 2018.

[12] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, 28:1447–1506, 2020. Open Access.

[13] Rudolf Ferenc, Tamás Viszkok, Tamás Aladics, Judit Jász, and Péter Hegedűs. Deep-water framework: The swiss army knife of humans working with machine learning models. *SoftwareX*, 12:100551, December 2020. Open Access.

[14] Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of source code defects by data mining conducted on GitHub. In *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA 2015)*, volume 9159 of *Lecture Notes in Computer Science (LNCS)*, pages 47–62, Banff, Alberta, Canada, June 2015. Springer-Verlag.

[15] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark and taxonomy of javascript bugs. *Software Testing, Verification and Reliability*, October 2020. Open Access.

[16] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, November 2005.

[17] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, March 2018.

[18] Péter Hegedűs and Rudolf Ferenc. Static code analysis alarms filtering reloaded: A new real-world dataset and its ml-based utilization. *IEEE Access*, 10:55090–55101, 2022. Open Access.

[19] István Kádár, Péter Hegedűs, and Rudolf Ferenc. Adding constraint building mechanisms to a symbolic execution engine developed for detecting runtime errors. In *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA 2015)*, volume 9159 of *Lecture Notes in*

Computer Science (LNCS), pages 20–35, Banff, Alberta, Canada, June 2015. Springer-Verlag.

[20] Gergely Ladányi, Zoltán Tóth, Rudolf Ferenc, and Tibor Keresztesi. A software quality model for RPG. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, pages 91–100, Montréal, Canada, March 2015. IEEE Computer Society.

[21] Yixun Liu, Denys Poshyvanyk, Rudolf Ferenc, Tibor Gyimóthy, and Nikos Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 233–242, Edmonton, Canada, September 2009. IEEE Computer Society.

[22] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, March 2008.

[23] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, February 2009.

[24] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 95–104, Victoria, British Columbia, Canada, September 2014. IEEE Computer Society.

[25] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software*, 129(C):107–126, July 2017.

[26] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. 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)*, pages 625–638, Beijing, China, July 2016. Springer International Publishing.

[27] Béla Újházi, Rudolf Ferenc, Denys Poshyvanyk, and Tibor Gyimóthy. New conceptual coupling and cohesion metrics for object-oriented systems. In *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, pages 33–42, Timişoara, Romania, September 2010. IEEE Computer Society. Best paper of the conference.

Other References

[28] ISO/IEC 9126-1:2001 Software engineering — Product quality — Part 1: Quality model . 2001.

[29] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. 2013.

[30] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from `tensorflow.org`, Accessed: 2023-03-20.

[31] F. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57, March 2001.

[32] F. Abreu, G. Pereira, and P. Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 13–22, March 2000.

[33] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto. A systematic literature review of open source software quality assessment models. *SpringerPlus*, 5(1):1936, 2016.

[34] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen. Measuring coupling and cohesion of software modules: an information-theory approach. In *Proceedings Seventh International Software Metrics Symposium*, pages 124–134, April 2001.

[35] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance request: a case study. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 227–230, March 2000.

[36] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links Between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.

[37] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *Proceedings of the Fifth International Symposium on Software Metrics (METRICS98)*, pages 23–34. IEEE Computer Society, November 1998.

[38] Ö. F. Arar and K. Ayan. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing*, 33:263–277, 2015.

[39] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, Augustus 2004.

[40] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[41] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[42] V. R. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm.* John W & S, 1994.

[43] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[44] M. W. Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputing Applications*, 6(1):13–49, April 1992.

[45] J. M. Bieman and B-K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the 1995 Symposium on Software Reusability*, SSR '95, pages 259–262. ACM, 1995.

[46] C. M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford., 1995.

[47] S. Bohner. Impact analysis in the software change process: a year 2000 perspective. In *1996 Proceedings of International Conference on Software Maintenance*, pages 42–51, Nov 1996.

[48] S. A. Bohner and D. Gracanin. Software impact analysis in a virtual environment. In *28th Annual NASA Goddard Software Engineering Workshop, 2003. Proceedings.*, pages 143–151, Dec 2003.

[49] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *Proceedings of the (19th) International Conference on Software Engineering*, pages 412–421, May 1997.

[50] L. Briand, W. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7):706–720, July 2002.

[51] L. C. Briand, J. Daly, V. Porter, and J. Wust. A comprehensive empirical validation of design measures for object-oriented systems. In *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB100262)*, pages 246–257, Nov 1998.

[52] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3:65–117, 1998.

[53] L. C. Briand, J. W. Daly, and J. K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan 1999.

[54] L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[55] L. C. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 475–482, Aug 1999.

[56] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, may 2000.

[57] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 9 pp.–29, Sep. 2005.

[58] L. F. Capretz and J. Xu. An empirical validation of object-oriented design metrics for fault prediction. *Journal of Computer Science*, 4(7):571, 2008.

[59] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*, pages 97–107, Oct 2000.

[60] C. Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[61] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[62] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243, 2011.

[63] H. S. Chae, Y. R. Kwon, and D. H. Bae. A cohesion measure for object-oriented classes. *Software: Practice and Experience*, 30(12):1405–1431, 2000.

[64] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 241–247, June 2000.

[65] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, 1998.

[66] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 197–211, New York, NY, USA, 1991. ACM.

[67] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, jun 1994.

[68] E. S. Cho, C. J. Kim, D. D. Kim, and S. Y. Rhew. Static and dynamic metrics for effective object clustering. In *Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No.98EX240)*, pages 78–85, Dec 1998.

[69] D. Cireşan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012.

[70] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 135–144, Aug 2005.

[71] C. J. Clemente, F. Jaafar, and Y. Malik. Is predicting software security bugs using deep learning better than the traditional machine learning algorithms? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 95–102. IEEE, 2018.

[72] S. Counsell, S. Swift, and J. Crampton. The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology*, 15(2):123–149, apr 2006.

[73] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[74] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 433–436, New York, NY, USA, 2007. Association for Computing Machinery.

[75] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th Working Conference on Mining Software Repositories (MSR)*, pages 31–41. IEEE, 2010.

[76] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[77] D. P. Darcy and C. F. Kemerer. OO metrics in practice. *IEEE Software*, 22(6):17–19, nov 2005.

[78] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4), September 2007.

[79] DeepBugHunter – experimental python framework for deep learning. `https://github.com/sed-inf-u-szeged/DeepBugHunter`, 2019. Accessed: 2023-03-20.

[80] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, sep 1990.

[81] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[82] B. Dit, D. Poshyvanyk, and A. Marcus. Measuring the semantic similarity of comments in bug reports. In *in Proc. of 1 st STSM'08*, 2008.

[83] M. Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[84] N. Dragan, M. Collard, and J. Maletic. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, sep 2006.

[85] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, jun 1991.

[86] T. Durieux and M. Monperrus. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Technical report, Universite Lille 1, 2016.

[87] M. Eaddy, A. V. Aho, G. Antoniol, and Y. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *2008 16th IEEE International Conference on Program Comprehension*, pages 53–62, June 2008.

[88] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt, 1994.

[89] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, jul 2001.

[90] K. El Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, feb 2001.

[91] L. Etzkorn and H. Delugach. Towards a semantic metrics suite for object-oriented design. In *Proceedings. 34th International Conference on Technology of Object-Oriented Languages and Systems - TOOLS 34*, pages 71–80, Aug 2000.

[92] L. H. Etzkorn and C. G. Davis. Automatically identifying reusable OO legacy code. *Computer*, 30(10):66–71, 1997.

[93] L. H. Etzkorn, S. Gholston, and W. E. Hughes. A semantic entropy metric. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(4):293–310, 2002.

[94] L. H. Etzkorn, S. E. Gholston, J. L. Fortune, C. E. Stein, D. Utley, P. A. Farrington, and G. W. Cox. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, 46(10):677–687, aug 2004.

[95] B. Fischer. Specification-based browsing of software component libraries. In *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*, pages 74–83, Oct 1998.

[96] B. Fluri, M. Würsch, E. Giger, and H. C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, December 2009.

[97] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, 12(2):219–245, apr 2006.

[98] P. W. Foltz, W. Kintsch, and T. K. Landauer. The measurement of textual coherence with latent semantic analysis. *Discourse Processes*, 25(2-3):285–307, jan 1998.

[99] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[100] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 13–23, Sep. 2003.

[101] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[102] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

[103] R. L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE software*, 18(3):112–111, 2001.

[104] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[105] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[106] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 96–103. IET, 2011.

[107] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Reflections on the nasa mdp data sets. *IET software*, 6(6):549–558, 2012.

[108] D. L. Gupta and K. Saxena. Software bug prediction using object-oriented metrics. *Sādhanā*, 42(5):655–669, 2017.

[109] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[110] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[111] T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol.*, 23(4):33:1–33:39, September 2014.

[112] M. A. K. Halliday. *Cohesion in English.* Longman, London, 1976.

[113] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[114] Y. Hassoun, R. Johnson, and S. Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 339–346, March 2004.

[115] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, Jan 2006.

[116] R. Helm and Y. S. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 47–61, New York, NY, USA, 1991. ACM.

[117] B. Henderson-Sellers. *Software Metrics.* U.K., Prentice Hall, 1996.

[118] S. Herbold, A. Trautsch, and J. Grabowski. A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, 44(9):811–833, 2017.

[119] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch. The race to the vulnerable: Measuring the log4j shell incident. *arXiv preprint arXiv:2205.02544*, 2022.

[120] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 14–23, New York, NY, USA, 2007. ACM.

[121] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[122] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

[123] J. Horning, H. Lauer, P. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. *Operating Systems*, pages 171–187, 1974.

[124] S. Hosseini, B. Turhan, and D. Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, Feb 2019.

[125] X. Huo, M. Li, and Z-H. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, pages 1606–1612, 2016.

[126] R. Jayanthi and L. Florence. Software defect prediction techniques using metrics based on neural network classifier. *Cluster Computing*, 22(1):77–88, 2019.

[127] J. D. Jobson. *Applied Multivariate Data Analysis*. Springer New York, 1992.

[128] D. Johnson. Quicknet, 2019. Accessed: 2023-03-20.

[129] 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, pages 9:1–9:10, New York, NY, USA, 2010. ACM.

[130] M. Jureczko and L. Madeyski. A review of process metrics in defect prediction studies. *Metody Informatyki Stosowanej*, 5:133–145, 2011.

[131] 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, pages 437–440, New York, NY, USA, 2014. ACM.

[132] 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*, 39(6):757–773, 2013.

[133] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: an automatic categorization system for open source repositories. In *11th Asia-Pacific Software Engineering Conference*, pages 184–193, Nov 2004.

[134] M. L. Kherfi, D. Ziou, and A. Bernardi. Image retrieval from the world wide web. *ACM Computing Surveys*, 36(1):35–67, mar 2004.

[135] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[136] W. Kintsch. *Comprehension: A paradigm for cognition.* Cambridge University Press, New York, 1998.

[137] R. Kosara, C. G. Healey, V. Interrante, D. H. Laidlaw, and C. Ware. Visualization viewpoints. *IEEE Computer Graphics and Applications*, 23(4):20–25, July 2003.

[138] S. Kramer and H. Kaindl. Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Transactions on Software Engineering and Methodology*, 13(3):332–358, jul 2004.

[139] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[140] W. H. Kruskal and W. A. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[141] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230 – 243, 2007. 12th Working Conference on Reverse Engineering.

[142] S. Kyatam, A. Alhayajneh, and T. Hayajneh. Heartbleed attacks implementation and vulnerability. In *2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–6. IEEE, 2017.

[143] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[144] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*, pages 218–229. IEEE, 2017.

[145] T. K. Landauer and S. T. Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997.

[146] D. J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 149–158, June 2006.

[147] J. K. Lee, S. J. Jung, S. D. Kim, W. H. Jang, and D. H. Ham. Component identification method with coupling and cohesion. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pages 79–86, Dec 2001.

[148] Y. S. Lee, B. S. Liang, S. F. Wu, and F. J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of International Conference on Software Quality*, Maribor, Slovenia, 1995.

[149] J. Li, P. He, J. Zhu, and M. R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[150] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993. Object-Oriented Software.

[151] Z. Li, X. Jing, and X. Zhu. Progress on approaches to software defect prediction. *IET Software*, 12(3):161–175, 2018.

[152] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, pages 55–56, New York, NY, USA, 2017. ACM.

[153] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.

[154] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining eclipse developer contributions via author-topic models. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 30–30, May 2007.

[155] K. K. Lo, C. K. Man, and E. Baniassad. Isolating and relating concerns in requirements using latent semantic analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 383–396, New York, NY, USA, 2006. ACM.

[156] R. F. E. Lorch Jr and E. J. O'Brien. *Sources of coherence in reading.* Lawrence Erlbaum Associates, Inc, 1995.

[157] M. Lormans and A. van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–56, March 2006.

[158] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5. Chicago, Illinois, 2005.

[159] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, Aug 1991.

[160] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. *CoRR*, abs/1901.06024, 2019.

[161] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proceedings 10th International Workshop on Program Comprehension*, pages 289–292, June 2002.

[162] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112, May 2001.

[163] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[164] R. Malhotra and A. Jain. Software fault prediction for object oriented systems: A literature review. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–6, 2011.

[165] C. Manjula and L. Florence. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*, 22(4):9847–9863, 2019.

[166] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[167] A. Marcus. *A Semantic Driven Program Analysis.* PhD thesis, Kent State University, Kent, OH, USA, 2003.

[168] A. Marcus, A. De Lucia, J. H. Hayes, and D. Poshyvanyk. Working session: Information retrieval based approaches in software evolution. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, sep 2006.

[169] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov 2001.

[170] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(05):811–836, oct 2005.

[171] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 133–142, Sep. 2005.

[172] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering*, pages 214–223, Nov 2004.

[173] T. Menzies, R. Krishna, and D. Pryor. The Promise Repository of Empirical Software Engineering Data, 2015. Accessed: 2023-03-20.

[174] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[175] T. M. Meyers and D. Binkley. Slice-based cohesion metrics and software intervention. In *11th Working Conference on Reverse Engineering*, pages 256–265, Nov 2004.

[176] A. Michail and D. Notkin. Assessing software libraries by browsing similar classes, functions and relationships. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 463–472, May 1999.

[177] Á. Mitchell and J. F. Power. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. *Science of Computer Programming*, 59(1):4 – 25, 2006. Special Issue on Principles and Practices of Programming in Java (PPPJ 2004).

[178] A. Mnih and G. E. Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

[179] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pages 120–, Washington, DC, USA, 2000. IEEE Computer Society.

[180] A. Mohamed, G. E. Dahl, G. Hinton, et al. Acoustic modeling using deep belief networks. *IEEE Trans. Audio, Speech & Language Processing*, 20(1):14–22, 2012.

[181] C. Montes de Oca and D. L. Carver. Identification of data cohesive subsystems using data mining techniques. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 16–23, Nov 1998.

[182] 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*, pages 181–190. ACM, 2008.

[183] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[184] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[185] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[186] A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *Journal of Systems and Software*, 20(3):295 – 308, 1993. Oregon Metric Workshop on Software Metrics, 1992.

[187] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, 33(6):402–419, jun 2007.

[188] D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198, 1999.

[189] A. M. Orme, H. Tao, and L. H. Etzkorn. Coupling metrics for ontology-based system. *IEEE Software*, 23(2):102–108, March 2006.

[190] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings. 26th International Conference on Software Engineering*, pages 491–500, May 2004.

[191] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[192] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *[1993] Proceedings First International Software Metrics Symposium*, pages 71–81, May 1993.

[193] Y. Pan, L. Wang, L. Zhang, B. Xie, and F. Yang. Relevancy based semantic interoperation of reuse repositories. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 211–220, New York, NY, USA, 2004. ACM.

[194] L. Pascarella, F. Palomba, and A. Bacchelli. Re-evaluating method-level bug prediction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 592–601. IEEE, 2018.

[195] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *International Conference on Software Engineering*. IEEE, 1992.

[196] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[197] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo. The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 13. ACM, 2016.

[198] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[199] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, sep 2006.

[200] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 37–48, June 2007.

[201] Michael Pradel and Koushik Sen. Deep learning to find bugs. *Technical Report*, 2017.

[202] T. Quah and M. M. T. Thwin. Application of neural networks for software quality prediction using object-oriented metrics. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 116–125, Sep. 2003.

[203] J-P. Queille, J-F. Voidrot, N. Wilde, and M. Munro. The impact analysis task in software maintenance: a model and a case study. In *Proceedings 1994 International Conference on Software Maintenance*, pages 234–242, Sep. 1994.

[204] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[205] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[206] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan. The impact of using regression models to build defect classifiers. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 135–145. IEEE Press, 2017.

[207] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 10(2):220–232, 1975.

[208] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 11–20, New York, NY, USA, 2005. ACM.

[209] G. Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 171–180. IEEE, 2010.

[210] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 43–55, New York, NY, USA, 2001. ACM.

[211] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 499–510, May 2007.

[212] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 10–13, May 2018.

[213] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1993.

[214] R. Sarikaya, G. E. Hinton, and A. Deoras. Application of deep belief networks for natural language understanding. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 22(4):778–784, 2014.

[215] J. Sayyad Shirabad and T. J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[216] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215, 2013.

[217] R. Shetty, K-K. R. Choo, and R. Kaufman. Shellshock vulnerability exploitation and mitigation: a demonstration. In *International Conference on Applications and Techniques in Cyber Security and Intelligence: Applications and Techniques in Cyber Security and Intelligence*, pages 338–350. Springer, 2018.

[218] B. Shuai, H. Li, M. Li, Q. Zhang, and C. Tang. Software defect prediction using dynamic support vector machine. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*, pages 260–263. IEEE, 2013.

[219] S. Siegel and N. J. Castellan. *Nonparametric Statistics for the Behavioral Sciences.* McGraw-Hill, New York, 1988.

[220] C. Stein, L. H. Etzkorn, G. W. Cox, P. A. Farrington, S. Gholston, D. R. Utley, and J. Fortune. A new suite of metrics for object-oriented software. In *Software Audit and Metrics*, pages 49–58, 2004.

[221] J. D. Strate and P. A. Laplante. A literature review of research in software defect reporting. *IEEE Transactions on Reliability*, 62(2):444–454, June 2013.

[222] R. Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, apr 2003.

[223] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, jan 2005.

[224] R. Tairas and J. Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, 19 September 2008.

[225] R. S. Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.

[226] J. Wang, B. Shen, and Y. Chen. Compressed c4. 5 models for software defect prediction. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 13–16. IEEE, 2012.

[227] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 297–308. IEEE, 2016.

[228] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 461–470, May 2008.

[229] E. J. Weyuker, R. M. Bell, and T. J. Ostrand. Replicate, replicate, replicate. In *Replication in Empirical Software Engineering Research (RESER), 2011 Second International Workshop on*, pages 71–77. IEEE, 2011.

[230] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[231] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[232] F. G. Wilkie and B. A. Kitchenham. Coupling Measures and Change Ripples in C++ Application Software. *J. Syst. Softw.*, 52(2):157–164, June 2000.

[233] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. Citeseer, 2014.

[234] W. E. Wong, V. Debroy, and D. Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, 2012.

[235] Z. Xu, T. M. Khoshgoftaar, and E. B. Allen. Prediction of software faults using fuzzy nonlinear regression modeling. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 281–290. IEEE, 2000.

[236] H. Y. Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *2005 Australian Software Engineering Conference*, pages 212–221, March 2005.

[237] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *QRS*, pages 17–26, 2015.

[238] Y. Ye and G. Fischer. Reuse-conducive development environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[239] R. K. Yin. *Applications of Case Study Research.* Sage Publications, Inc, CA, USA, 2003.

[240] P. Yu, T. Systa, and H. Muller. Predicting fault-proneness using oo metrics. an industrial case study. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 99–107, March 2002.

[241] Z. Yu, N. A. Kraft, and T. Menzies. How to read less: Better machine assisted reading methods for systematic literature reviews. *arXiv preprint arXiv:1612.03224*, 2016.

[242] J. Zhao and B. Xu. Measuring aspect cohesion. In *Fundamental Approaches to Software Engineering*, pages 54–68. Springer Berlin Heidelberg, 2004.

[243] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, April 2006.

[244] Y. Zhou, J. Lu, H. Lu, and B. Xu. A comparative study of graph theory-based class cohesion measures. *ACM SIGSOFT Software Engineering Notes*, 29(2):13, mar 2004.

[245] Y. Zhou, B. Xu, J. Zhao, and H. Yang. ICBMC: an improved cohesion measure for classes. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 44–53, Oct 2002.

[246] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[247] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*, page 9. IEEE Computer Society, 2007.

[248] L. Zou, M. W. Godfrey, and A. E. Hassan. Detecting interaction coupling from task interaction histories. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 135–144, June 2007.