Formal Definition and Automatic Generation of Semantic Metrics: An Empirical Study on Bug Prediction

Ting Hu[1], Ran Mo*[1], Pu Xiong[1], Zengyang Li[1], Qiong Feng[2]
[1]*School of Computer, Central China Normal University*
[2]*School of Computer Science and Engineering, Nanjing University of Science and Technology*
Email: 826473959@qq.com, moran@mail.ccnu.edu.cn, mos365@hotmail.com, zengyangli@mail.ccnu.edu.cn,qiongfeng@njust.edu.cn

***Abstract*—Bug prediction is helpful for facilitating bug fixes and improving the efficiency in software development and maintenance. In the past decades, researchers have proposed numerous studies on bug prediction by using code metrics. However, most of the existing studies use syntax-based metrics, there exists little work building bug prediction models with semantic metrics from source code. In this paper, we propose a new model, semantic dependency graph *(SDG)*, to represent semantic relationships among source files. Based on the *SDG*, we formally define a suite of semantic metrics reflecting semantic characteristics of a project's source files. Moreover, we create a tool to automate the generation of our proposed SDG-based metrics. Through our experimental studies, we have demonstrated that the SDG-based semantic metrics are effective for building bug prediction models, and the SDG-based metrics outperform traditional syntactic metrics on bug prediction. In addition, models using the SDG-based metrics could achieve a better prediction performance than two state-of-the-art models that learn semantic features automatically. Finally, we have also presented that our approach is applicable in practice in terms of execution time and space.**

***Index Terms*—Semantic Code Metrics, Semantic Dependency, Bug Prediction**

I. Introduction

Bug prediction is an active and important research area. Accurate predictions help developers effectively identify bug-prone files at an early stage and prioritize their fixing and testing, which in turn could improve the efficiency and productivity of development teams. Over the past decades, researchers have proposed numerous bug prediction approaches and tools to predict bug-prone files.

Using code metrics to build prediction models is one of the most prevalent directions in the field of bug prediction. Numerous studies [1], [2], [3], [4], [5], [6], [7] have leveraged various code metrics or the combinations of them to predict bug-prone files. However, most of the existing studies only use syntactic code metrics for bug prediction, there is little work building prediction models based on semantic code metrics.

Generally, a program contains syntactical information that reflects how a program is represented. Based on the syntactical aspects of programs, various syntactic metrics have been proposed and become the most widely used code metrics over the past decades. For example, the well-known LOC (lines of code) measures a program's size; McCabe's *cyclomatic complexity* [8] reflects the number of linearly independent paths through a program's source code; The suite of C&K metrics [9] consists of six metrics which focus on measuring the characteristics of Object-oriented programs, such as dependency counts, inheritance counts, and method counts, etc.; Fan-in calculates the number of calling programs, Fan-out calculates the number of called programs. All of these syntactic metrics are formally defined and have been widely used for bug prediction [2], [4], [5], [10], [6].

A program doesn't only embody syntactical information, but semantic information too, which reflects textual relations among programs. Compared with a program's syntax, the semantic information is often more implicit and hidden more deeply in a program's source code. Therefore, unlike the syntactic metrics, which have been well defined and there exist numerous open-source or commercial tools supporting their extractions and calculations, so far, we still lack a suite of well-defined semantic metrics, not to mention the tools that support the automatic detection of such metrics. Meanwhile, although semantic information has been used for code suggestion [11], [12] and code completion [13], [14], there is little work well constructing semantic code metrics for developing bug prediction models.

To bridge the gap between semantic metrics and bug prediction, we propose a suite of semantic metrics with explicit definitions and descriptions, and create a tool for automating the generation of these proposed metrics. In this paper, we first extract semantic information from source files by using latent semantic analysis (LSA) [15] with the measure of TD-IDF [16]. According to the extracted semantic information, we propose to generate a semantic similarity matrix and a novel model, called *Semantic Dependency Graph (SDG)*, which represents the semantic dependencies among files. The nodes of a *SDG* are a project's source files, and each edge, $e(f_i, f_j)$, indicates there exists a semantic dependency from file i to j. Based on the generated similarity matrix and semantic dependency graph, we formally define eleven SDG-based semantic metrics. Using the proposed SDG-based semantic metrics as

*Corresponding Author

attributes, we further apply machine learning classifiers to build bug prediction models.

To investigate the effectiveness of our SDG-based semantic metrics on bug prediction, we seek to answer the following research questions:

RQ1: Is it possible to build effective bug prediction models based on our proposed SDG-based semantic metrics?
RQ2: Do the SDG-based semantic metrics outperform traditional syntactic metrics on bug prediction?
RQ3: Do the SDG-based semantic metrics have a better performance on bug prediction when compared to the state of the art?
RQ4: What is the performance of our approach in terms of execution time and space?

Through our analyses on seven sets of experimental studies, we have found that: 1) by using our proposed SDG-based semantic metrics, we can build promising bug prediction models, where 90.5% (19/21) of all the derived models have a F-measure greater than 0.5; 2) models using our SDG-based semantic metrics outperform the ones using transitional syntactic metrics, the average increases of performance range from 16.7% to 47.1%; 3) Compared with the state-of-the-art approaches that automatically learn semantic features for bug prediction, the models using the semantic metrics generated by our SDG-based approach could achieve a better performance; 4) At last, we track the whole process of SDG-based metrics generation, and have demonstrated that the execution time and space of our approach are affordable. Over all experimental studies, the process of semantic metrics generation costs 2.9-111.6 minutes and consumes 106-146 MB memory, suggesting that our approach is applicable in practice.

In conclusion, we believe our work contributes to the state of the art as follows:

- Our work proposes a novel model, SDG, which could explicitly represent the semantic relations among source files.
- Our work presents a systematic methodology to automatically analyze program semantics, generate the semantic similarity metrics and semantic dependency graph, and extract the semantic metrics.
- To the best of our knowledge, our work is the first study of proposing and formally defining a suite of semantic metrics, and presenting how to use these metrics to develop bug prediction models.

The rest of this paper is organized as follows: Section II presents the background of our work. Section III shows related work. Section IV describes the details of our approach for SDG-based semantic metrics generation and bug prediction. Section V presents the design of our empirical study, our evaluation methods. Section VI shows the evaluation analysis and results. Section VII discusses the limitations and threats to validity in our work, and Section VIII concludes.

II. Background

In this section, we introduce the basic concepts and techniques behind our work, including the description of code semantics, and the concepts related to LSA and TF-IDF.

A. Code Semantics

Semantics of a program reflects the meaning of this program, that is, what the program does. In general, semantic aspects of a program could often be represented by its lexical tokens, such as keywords, comments, operators and identifiers, etc.. Thus, for each program, we can generate its corresponding textual document which represents the program's meaning. Such a textual document will consist of a set of textual terms derived from the analysis of lexical information.

B. Latent Semantic Analysis

Latent semantic analysis (LSA), also known as latent semantic indexing (LSI), is a technique to examine semantic relationships between a set of documents and their incorporated terms [15]. The process of LSA first constructs a term-document matrix to describe the occurrence of terms with respect to different documents. For such a term-document matrix, the rows represent terms and columns represent documents. In this paper, we leverage the model of TD-IDF (Term Frequency-Inverse Document Frequency Method) [16] to populate each element within the term-document matrix.

The value of TD-IDF measures how relevant a term is to a document in a collection of documents. After all TD-IDF values are populated into the term-document matrix, LSA then applies *singular value decomposition (SVD)* [17] to perform a matrix decomposition for filtering out noises. In this way, the row dimension of the original term-document matrix would be reduced but the similarity structure among columns would be preserved. Thus, each column of the decomposition matrix can still represent a particular document.

For any two document, d_1 and d_2, using their corresponding columns to construct vectors, V_1 and V_2, the semantic similarity of documents d_1 and d_2 could be directly calculated by taking the cosine value between V_1 and V_2. Each similarity value could be normalized into the range of [0, 1], and a greater value indicates there exists higher semantic similarity between two documents.

C. Term Frequency-Inverse Document Frequency

The Term Frequency-Inverse Document Frequency (TF-IDF) [16] measures how relevant a term or a word is to a document in a corpus of documents. It has been widely used for information retrieval and text mining. In this paper, we leveraged TF-IDF to populate the elements of a term-document matrix. Generally, TF-IDF consists of two important measures: Term Frequency (TF) and Inverse Document Frequency (IDF). TF indicates how many times that a term has been found in a document: $tf_{t,d} = f_{t,d}$, where $f_{t,d}$ is the frequency of term t that occurs in document d. The study of Croft et al. [18] has presented that the logarithm variant of TF can effectively improve TF-IDF's performance in practice. In addition, Lan et al.'s study [19] has also demonstrated that, to avoid distortion of the results caused by the high original frequency of a term in a specific document, the logarithm variant of TF should be used as the term frequency. Therefore, TF is often calculated as follows:

$$tf_{t,d} = 1 + log(f_{t,d}) \tag{1}$$

IDF means the inverse of the number of documents where the term is found in the whole corpus. IDF is calculated as follows:

$$idf_{t,d} = log(\frac{N}{n_t}) \tag{2}$$

where N is the total number of documents in the document corpus, and n_t is the number of documents containing term t.

At last, TD-IDF is calculated as the product of TD and IDF: $\textit{tf-idf}_{t,d} = tf_{t,d} \times idf_{t,d}$. According to this formula, we can observe that TF-IDF value will logarithmically increase to the frequency of a term in a document, but will decrease by the total number of documents containing a term. By combining both TF and IDF measures, TD-IDF can filter some common but unrepresentative terms in documents.

III. Related work

Bug prediction has been one of the most active research fields for decades. Researchers have proposed numerous predictive models by using various code metrics.

A. Syntactic metrics in bug prediction.

Nagappan et al. [2] investigated different complexity metrics and demonstrated that these metrics are useful and successful for predicting the defective files. In this paper, the authors collected a suite of code metrics and proposed a model which uses a set of them as predictors. They applied the model to five large-scale software systems and demonstrated it can accurately predict post-release defects. Although a given set of metrics might be a good predictor in a given project, there is not a single set of metrics that can serve the best for all projects. That is, one would need to select a different set of metrics to accurately predict defective files in different kinds of projects.

Menzies et al. [3] demonstrated the static code metrics could be used as defect predictors. When applying Naive Bayes learner, they presented the defect predictors are useful for defect detection with an average detection probability rate of 71%, and an average false alarms rate of 25%. In addition, they presented that these predictors should be regarded as probabilistic indicators instead of absolute indicators.

Gyimothy et al. [1] calculated the syntactic code metrics from the source code of Mozilla, then based on these metrics, they leveraged two statistical methods and two machine learning techniques to predict the failure-proneness of each class. They also investigated how the predicted fault-proneness and code metrics of Mozzilla changed over seven versions.

Giger et al.'s study [5] investigated to build bug prediction models at method level. The authors presented that their models based on method-level syntactic code metrics and change metrics could be used to accurately predict bug-prone methods. They also showed that change metrics outperformed source code metrics on bug prediction, and presented that their models are robust with respect to different distributions of samples.

He et al. [10] collected a list of twenty syntactic code metrics as the bug prediction features, then applied the feature selection to build bug prediction models. Jing et al. [6] selected twenty code metrics mined from software projects, and learned multiple dictionaries and sparse representation coefficients for predicting software bugs.

Selby and Basili [20] have investigated the relation between dependency structure and software defects. Ostrand et al. [21] developed a negative binomial regression model using file size and file change information, and demonstrated the model could effectively predict the expected number of faults in each file in the next release of a software project. Lessamann et. al. [22] used 39 code metrics and applied 22 classifiers to improve the defect prediction.

As shown in the above studies, numerous bug prediction models have been built by using syntactic metrics. Our study complementarily contributes to this field by proposing a new suite of semantic metrics, and presenting how to use these metrics for bug prediction.

B. Semantics in bug prediction

Based on the analysis of textual similarity between bug reports and source files, researchers have also proposed various approaches to predict bug-prone files. For example, Zhou et al. [23] proposed a revised Vector Space Model (rVSM) to analyze the textural similarity between bug reports and source files. In their experiments, the authors used the weighted sum of the two rankings to locate the related source files for a bug.

Li et al. [24] introduced the deep learning-based model, DeepFL. It used the textual similarity between source code methods and failed test information, code metrics and other history measures as attributes to predict bugs.

Ye et al. [25] proposed LR+WE model to predict bugs. This model represented bug reports and source files as word embeddings and extracted textual similarity between bug reports and files, then used it as a feature for bug prediction.

Xiao et al. [26] introduced the model of DeepLoc, which used an enhanced convolutional neural network (CNN) to build a prediction model where the bug-fixing recency, bug frequency, and textual similarity between bug reports and files were considered as features.

Unlike the above studies which mainly leveraged the textual semantics between bug reports and source files to predict bugs, our work focused on conducting semantic analysis on source files, that is, the semantic similarity among source files.

There are two previous studies [7], [27] similar to our work. Both of them automatically learned semantic metrics from source code to predict software bugs. Wang et al. [7] proposed a DBN-based (Deep Belief Network) approach to automatically learn semantic features. Based on these features, the authors applied ADTree, NB and LR classifiers to predict bug-prone files. Huo et al. [27] proposed a CAP-CNN (Convolutional Neural Network for Comments Augmented Programs) model, which also automatically learned semantic features from the source code, and then directly used such features to predict bug-prone files.

However, our work proposes a suite of semantic metrics. Meanwhile, for each semantic metric, we provide a formal definition and detailed description. In addition, we present how to use such metrics to build bug prediction models. In the following sections, we conduct extensive comparisons between our approaches and these two existing models.

IV. Methodology

Figure 1 depicts the overview of our approach for extracting semantic metrics and conducting bug prediction. 1) Given a project's source files, we first extract lexicon information for constructing a textual document for each file; 2) According to these created documents, we then calculate the semantic similarities among the documents by implementing LSA with the measure of TD-IDF; 3) Based on the semantic similarities, we further generate a semantic similarity matrix and a semantic dependency graph (SDG), from which we calculate the SDG-based semantic metrics for each source file; 4) Finally, we apply machine learning algorithms to build bug prediction models. We have shared the tool chain for the generation of SDG-based semantic metrics in our GitHub repository[1]. Next, we introduce the details of each main procedure within our approach.

A. Source code Parse

Given a project's source code, we leverage a reverse-engineering tool, Understand[2], to parse the source files. Each source file will be first represented as a Abstract Syntax Tree (AST), from which, the lexicon parser of Understand further generates the lexical information (identifiers, comments, etc.). For example, if we have a code fragment like this:

int len=10;//Length

the lexicon parser will extract a set of lexical tokens as shown in Table I.

TABLE I: An Example of Extracted Lexical Tokens

Text	Token	Text	Token	Text	Token
int	Keyword		Whitespace	a	Identifier
=	Operator	10	Literal	;	Punctuation
Length	Comment				

B. LSA Implementation

According to the derived lexical information, we could construct a collection of documents, $D = D_1, D_2, \ldots D_n$, where n is the number of files, and each document is derived for a particular file. Unlike a source file, which contains programs, its corresponding document consists of a set of textual terms derived from the related lexical information, including keywords, identifiers, comments and operators. Given a collection of documents, we then implement LSA to capture the semantic similarity among files. As we introduced in Section II, a term-document matrix will be generated by using TD-IDF. Based on the decomposition matrix after applying SVD, each document would be represented with a vector by using the corresponding column of the decomposition matrix.

C. Semantic Similarity Matrix Generation

Given a set of documents, $D = D_1, D_2, \ldots D_n$, and their related vectors $V = V_1, V_2, \ldots, V_n$. The semantic similarity between any two documents could be calculated by taking the cosine between their corresponding vectors. Such cosine similarity is a measure reflecting how similar of two documents without considering their size. It calculates cosine of the angle between two vectors in a multi-dimensional space. A smaller angle has a higher cosine value, meaning that two documents are more similar. In this step, we generate a matrix to represent the semantic similarity between any two files. The similarity matrix is a square matrix ($N \times N$ matrix) that consists of a set of source files. Each cell in this matrix, $cell(f_i, f_j)$, shows the semantic similarity between files i and j, that is, the semantic similarity of documents D_i and D_j.

D. Semantic Dependency Graph (SDG) Generation

Based on the semantic similarity matrix, we further transfer it into a semantic dependency graph, SDG. $SDG =< V, E >$, where V is the set of source files and E is the set of edges. Each of the edges, $e(f_i, f_j)$, indicates that file i semantically depends on file j. The weight of each edge will be the value of the corresponding semantic similarity. In this case, we consider there exists an edge from file A to B (i.e. file A semantically depends on file B), if the similarity weight between A and B satisfies: $w(A, B) >= C$, where C is a threshold indicting the significance of the semantic similarity. If a value is larger than the threshold, we consider there is a semantic dependency. Following the well-know Pareto principle [28], we set C to be 0.8, meaning that when the $w(A, B) >= 0.8$, we consider the semantic dependency from file A to B to be significant. In addition, to substantiate this setting, we randomly selected one hundred pairs of files and manually examined their lexicon information. We have observed that, when the similarity value of two files is larger than 0.8, we can explicitly identify similar or even identical keywords, identifiers, or comments.

E. SDG-based Semantic Metrics Definition

Given the semantic similarity matrix and semantic dependency graph, $SDG =< V, E >$, we formally define eleven metrics which represent the semantic characteristics of each file as follows:

Dependent Count (DCT): DCT_{f_i} is calculated as the total number of files that semantically depend on the file. A larger value of *DCT* often suggests a more influential and complicated file.

Sum of Dependent Weight (SDW): File i's SDW is calculated as the sum of the semantic weight of the edges formed by file i and the files that semantically depend on it. $SDW_{f_i} = \sum w(f_i, f_j)$, where $i \neq j$, and

[1]https://github.com/SDGSemantic/SDG_Semanticcode
[2]https://scitools.com/

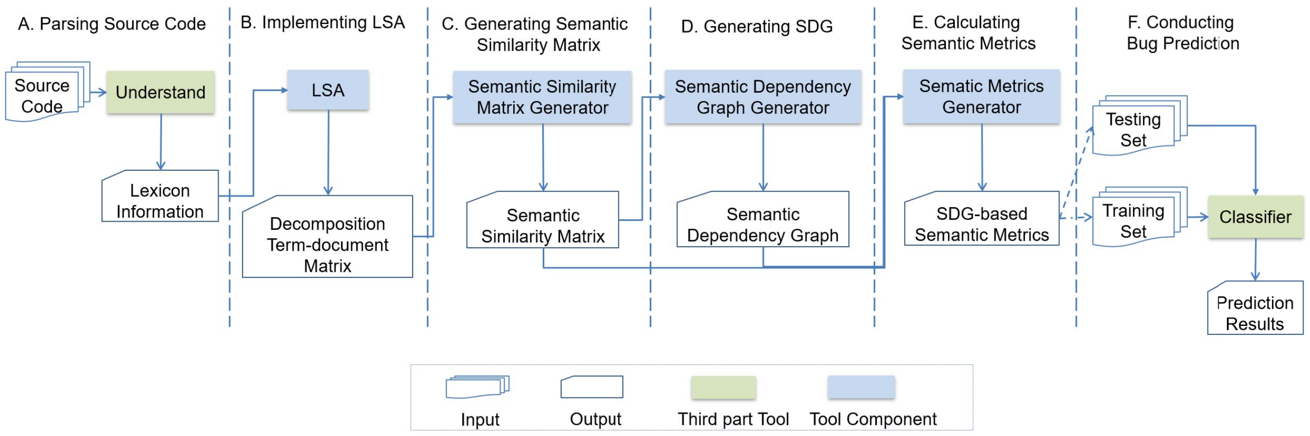
A. Parsing Source Code
B. Implementing LSA
C. Generating Semantic Similarity Matrix
D. Generating SDG
E. Calculating Semantic Metrics
F. Conducting Bug Prediction
Source Code
Understand
Lexicon Information
LSA
Decomposition Term-document Matrix
Semantic Similarity Matrix Generator
Semantic Similarity Matrix
Semantic Dependency Graph Generator
Semantic Dependency Graph
Sematic Metrics Generator
SDG-based Semantic Metrics
Testing Set
Training Set
Classifier
Prediction Results
Input
Output
Third part Tool
Tool Component

Fig. 1: Overview of Our SDG-based Semantic Metrics Extraction and Bug Prediction

$j = [1, 2, 3, ..., m]$, m is the total number of files that semantically depends on $file_i$.

Sum of Semantic Similarity (SSS): File i's SSS is calculated as the sum of the semantic similarities between file i and other files in the same project. $SSS_{f_i} = \sum w(f_i, f_k)$, where $i \neq k$, and $k = [1, 2, 3, ..., n]$, n is the total number of files in a project. Unlike *SDW* that sums the weight of each semantic dependency from file i according to a SDG, SSS takes all semantic similarities related to the file i into account.

Average of Dependency Weight (ADW): File i's ADW represents the average of the semantic weight of the edges formed by file i and the files that semantically depend on it: $ADW_{f_i} = \frac{SDW_{f_i}}{DC_{f_i}}$.

Average of Semantic Similarity (AS): File i's AS represents the average of the semantic similarities between file i and other files in the same project: $AS_{f_i} = \frac{SSS_{f_i}}{N}$, where N is the number of files that are semantically similar to file i, that is, the semantic similarity value is larger than 0.

Longest Dependency Path (LDP): LDP_{f_i} is calculated as the number of files involved in the longest path of file i in the semantic dependency graph. A larger LDP represents more files are transitively connected, which may increase complexity, since changes to a file may be propagated to other files at the same path.

The above six semantic metrics are calculated based on the direct dependencies among files. Next, we define a suite of semantic metrics by taking the transitivity of dependency into account. Considering file i as a starting point, the BFS algorithm is applied to traverse the semantic dependency graph. In this way, we can obtain a group of files that directly or indirectly depends on file i. We named such a file group as the *Impact Space*. For each file, a particular impact space would be generated. From a file's impact space, we calculate the other five semantic metrics as follows:

Impact Space File Count (ISF): ISF_{f_i} is the total number of files involved in file i's impact space. This measures a file's transitive impact on the whole project in terms of semantics.

Sum of Files' Weight (SFW): SFW_{f_i} is the sum of similarity weights between file i and the other files involved in its impact space.

Average of Files' Weight (AFW): AFW_{f_i} is the average of similarity weights between file i and the other files in its impact space: $AFW_{f_i} = \frac{SFW_{f_i}}{ISF_{f_i}}$.

Impact Space Weight (ISW): Each Impact space can form a sub graph of the semantic dependency graph, G. ISW_{f_i} is the sum of the weights of all edges in the sub graph (i.e. the impact space).

Average Weight of Impact Space (AWIS): $AWIS_{f_i}$ is the average weight of file i's impact space: $AWIS_{f_i} = \frac{ISW_{f_i}}{n}$, where n is the number of edges in the impact space.

Given a project's source code, our tool could automatically generate its corresponding semantic similarity matrix and semantic dependency graph, then extract and calculate each source file's semantic metrics.

F. Bug Prediction Model Development

We build models to predict bug-prone files by applying three supervised machine learning algorithms: Alternating Decision Tree, Naive Bayes and Logistic Regression, which have been widely used in various bug prediction models. Next, we briefly described all the machine learning techniques used in our experiments.

- ***Alternating Decision Tree (ADTree)*** is a classifier consisting of decision nodes and prediction nodes. Each decision node contains a predicate condition, and each prediction node has a single number. To classify an instance, the ADTree consider all decision nodes are true and sum all prediction nodes that have been traversed.
- ***Naive Bayes (NB)*** is one of the simple probabilistic classifiers. Naive Bayes algorithm leverages Bayes Theorem to calculate the conditional probability of all classes from the training data, and assumes the attributes to be independent. Naive Bayes has been shown to be a very effective classification algorithm in various studies.
- ***Logistic Regression (LR)*** is an extension of the linear regression model. Logistic regression transfers its output to be a probability value based on the logistic sigmoid function. Based on the probability values, the logistic regression could model the probabilities for classification problems by assigning two possible outcomes.

For each predictive study, we first generate the SDG-based semantic metrics, and use the training set to train the prediction models, then apply the trained models on testing set to validate their performance on bug prediction.

V. Evaluation

A. Research Questions

We proceed our empirical study by investigating the following research questions:

RQ1: *Is it possible to build effective bug prediction models based on our proposed SDG-based semantic metrics?* In this question, we investigate the performance of the developed prediction models by using our proposed semantic metrics.

RQ2: *Do the SDG-based semantic metrics outperform traditional syntactic metrics on bug prediction?* Syntactic metrics have been widely used in bug prediction. In this question, we attempt to understand whether our SDG-based semantic metrics are better predictors than the traditional syntactic metrics for building bug prediction models.

RQ3: *Do the SDG-based semantic metrics have a better performance on bug prediction when compared to the state of the art?* Leveraging two state-of-the-art studies that automatically learn semantic features from source code as the baseline, we further investigate whether our SDG-based metric could be used to build better bug prediction models.

RQ4: *What is the performance of our approach in terms of execution time and space?* Finally, we present whether our approach is applicable in practice by examining its execution time and space.

B. Subjects

In this paper, we construct our subjects from the PROMISE projects[3] listed in [29], which is an open data set and has been widely used in many prior bug prediction studies [10], [30], [7], [27]. To guarantee a fair and consistent comparison, we select five projects that have been used in the previous studies that we compare with, including: Log4j[4] – a Java-based logging utility; Lucene[5] – a search engine software library; Synapse[6] – a lightweight Enterprise Service Bus; Xalan[7] – a software library for transforming XML; and Xerces[8] – a library for manipulating XML documents.

Table II shows the basic facts of our prediction datasets. For each experiment, we use a former release as the training set, and the latter release as the testing set. Column "# files" indicated the number of files in a dataset. Using the first row as an example, the experiment uses the release 1.0 of project Log4j to train the prediction model, and the release 1.1 to do testing. Release 1.0 contains 119 source files, and Release 1.1 contains 104 source files.

In this work, we aim to train and validate bug prediction models for classifying files to be bug-prone or not bug-pone. Since the supervised algorithms are used for building prediction models, we first need to get each source file in the dataset labeled as *bug-prone* or *not bug-prone*. Following the prior studies of [29], [7], a file will be identified to be *bug-prone*, if the file has been changed for bug fixes at least once in revision history of testing release. Otherwise this file will be labeled as *not bug-prone*. To examine whether a file has been changed for bug fixes, we need to mine a project's revision history and bug reports by matching bug ticket IDs in commits. More details about the construction of dataset can be referred in [29].

TABLE II: Dataset of Each Experimental Study

Project	Training Set		Testing Set	
	Release	#Files	Release	#Files
Log4j	1.0	119	1.1	104
Lucene	2.0	186	2.2	238
Lucene	2.2	238	2.4	334
Synapse	1.0	157	1.1	222
Synapse	1.1	222	1.2	256
Xalan	2.4	676	2.5	762
Xerces	1.2	439	1.3	452

VI. Results

RQ1: Is it possible to build effective bug prediction models based on our proposed SDG-based semantic metrics?

To answer this question, we first leverage three widely used machine learning algorithms, ADTree, Naive Bayes and Logistic Regression to develop prediction models, respectively. Then we apply the derived models on seven sets of experimental studies and examine their performance. For each experiment, the dataset contains two consecutive releases of a project, where the previous release is set as the training set and the

[3]https://github.com/opensciences/opensciences.github.io/tree/master/repo/defect

[4]http://logging.apache.org/

[5]https://lucene.apache.org/

[6]https://synapse.apache.org/

[7]https://xalan.apache.org/

[8]https://xerces.apache.org/

latter release is set as the testing set. In this paper, we adopt *F-measure* to evaluate the performance of each prediction model. *F-measure* has been widely used in previous bug prediction studies [10], [31], [7], [27]. It takes both precision and recall into consideration for assessing how precise and how robust a classification is [32], [33].

We have summarized the experimental results in Table III. Using *Log4j 1.0→1.1* as an example, we can observe that, when using our SDG-based semantic metrics, the prediction models based on ADTree, NB, LR classifiers could achieve a F-measure value of 0.62, 0.56 and 0.65, respectively. According to the prior studies [34], [35], a *F-measure* value larger than or equal to 0.5 indicates a promising prediction. This means that all the three derived prediction models by using the SDG-based semantic metrics can effectively predict bug-prone files.

Considering all experiments together, we can find that most of models based on the SDG-based semantic metrics could achieve promising prediction performance: 90.5% (19/21) of the predictions get a *F-measure* greater than 0.5. It means that using our semantic metrics to build a bug prediction model is promising. As shown at the last row, the average F-measure values are 0.593, 0.633, and 0.731, with respect to the models based on ADTree, NB and LR. Therefore, we believe that the prediction model derived from our SDG-based semantic metrics can effectively predict bug-prone files.

TABLE III: Performance of Each Prediction Model

Project	Datasets	DT	NB	LR
Log4j	1.0→1.1	0.620	0.560	0.650
Lucene	2.0→2.2	0.582	0.803	0.847
Lucene	2.2→2.4	0.545	0.743	0.756
Synapse	1.0→1.1	0.616	0.551	0.616
Synapse	1.1→1.2	0.583	0.546	0.530
Xalan	2.4→2.5	0.444	0.497	0.938
Xerces	1.2→1.3	0.761	0.731	0.777
Average		0.593	0.633	0.731

Answer to RQ1: Using our proposed semantic metrics, we could build promising bug prediction models by applying Decision Tree, Naive Bayes, and Logistic Regression classifiers.

RQ2: Do the SDG-based semantic metrics outperform syntactic metrics on bug prediction?

Syntactic metrics have been widely used for bug prediction. To further validate the usefulness of our SDG-based semantic metrics on bug prediction, we compare the SDG-based metrics with traditional syntactic metrics. In this case, we directly leverage the twenty traditional syntactic metrics listed in [10] for this comparison. These syntactic metrics contain lines of code, number of operands and operators, Fan-in, Fan-out, C&K metrics [9], and McCabe's complexity [8], etc.. All of these metrics have been well defined and widely used in previous studies of bug prediction [5], [31], [10], [6], [7].

Table IV shows the comparison results of 21 experiments between prediction models using the SDG-based semantic metrics and the selected syntactic metrics. The higher F-measure value of each comparison has been highlighted.

Using *Lucene 2.2→2.4* as an example, where release 2.2 is used as the training set, and release 2.4 is used as the testing set, we can observe that: 1) F-measure values of SDG-based bug prediction models are 0.582, 0.803 and 0.847, with respect to ADTree, NB and LR classifiers; 2) F-measure values of syntactic bug prediction models are 0.502, 0.500 and 0.598, respectively. This suggests that our SDG-based semantic metrics outperform the studied syntactic metrics in this experiment.

Considering all experiments together, we can obtain the following observation: in 71.4% (15/21) of all experiments, the bug prediction models using our semantic defect prediction achieve a better performance than the ones using the syntactic metrics. More specifically, we can see that:

- ***SDG + ADTree vs Syntactic + ADTree.*** In 4 of 7 comparisons, the SDG-based semantic metrics outperform the studies syntactic metrics. Referring to the average values shown at the last row, we can see that the average F-measure of prediction models using the SDG-based metrics is 0.593, which is 16.7% higher than the models using syntactic metrics.
- ***SDG + NB vs Syntactic + NB.*** In 5 of 7 comparisons, the SDG-based semantic metrics outperform the selected syntactic metrics. The average F-measure of prediction models using the SDG-based metrics is 0.633, which is 31.3% higher than the models using syntactic metrics.
- ***SDG + LR vs Syntactic + LR.*** In 6 of 7 comparisons, the SDG-based semantic metrics outperform the studies syntactic metrics. The average F-measure of prediction models using the SDG-based metrics is 0.731, which is 47.1% higher than the models using syntactic metrics.

According to the above observations, we believe that our SDG-based semantic metrics outperform the selected syntactic metrics on bug prediction.

Answer to RQ2: The models using our proposed SDG-based semantic metrics could achieve a better performance on bug prediction than the models using traditional syntactic metrics.

RQ3: Do the SDG-based semantic metrics have a better performance on bug prediction when compared to the state of the art?

To further evaluate the effectiveness of our SDG-based semantic metrics on bug prediction, we leverage two state-of-the-art approaches as the baseline to conduct the comparisons. The first baseline is the DBN-based (Deep Belief Network) bug prediction model proposed by Wang et al. [7]. Their work uses a DBN model to automatically generate semantic features from source code, and applies ADTree, NB and LR classifiers

TABLE IV: Performance of Models Using SDG-based Metrics and Syntactic Metrics

Project	Datasets	SDG-Based			Syntactic		
		DT	NB	LR	DT	NB	LR
Log4j	1.0→1.1	0.620	0.560	**0.650**	0.687	0.689	0.535
Lucene	2.0→2.2	**0.582**	**0.803**	**0.847**	0.502	0.500	0.598
Lucene	2.2→2.4	0.545	**0.743**	**0.756**	0.605	0.378	0.694
Synapse	1.0→1.1	**0.616**	**0.551**	**0.616**	0.476	0.508	0.316
Synapse	1.1→1.2	**0.583**	0.546	0.530	0.530	0.565	0.533
Xalan	2.4→2.5	0.444	**0.497**	**0.938**	0.518	0.398	0.540
Xerces	1.2→1.3	**0.761**	**0.731**	**0.777**	0.238	0.333	0.266
Average		0.593	0.633	0.731	0.508	0.482	0.497

to predict bug-prone files. The second baseline is the CAP-CNN model proposed by [27]. CAP-CNN model is a deep learning model which could automatically generate semantic features from source code to predict bug-prone files.

Table V shows the comparison results between the SDG-based models and the DBN-based models, from which we can obtain the following observations:

- ***SDG+ADTree vs DBN+ADTree.*** In 3 of 7 comparisons, the SDG-based models perform better on bug prediction. Using *Log4j 1.0→1.1* as an example, the F-measure of SDG+DT model is 0.62, and the F-measure of DBN+DT is 0.701. According to the average value, we can see that the average F-measure of SDG-based models is 0.593. It is similar to the average value of DBN-based models, which is 0.608.
- ***SDG+NB vs DBN+NB.*** In 5 of 7 comparisons, the SDG-based models could obtain a better prediction performance. The average F-measure of SDG-based models is 0.633, which is 11.2% higher than the average value (0.569) of DBN-based models.
- ***SDG+LR vs DBN+LR.*** The SDG-based models achieve a better prediction performance in 5 of 7 comparisons as well. The average F-measure of SDG-based models is as high as 0.731, which is 29.6% higher than the average value (0.564) of DBN-based models.

Considering all experiments together, in 61.9% (13 of 21) of all experiments, the SDG-based models are better than DBN-based models on bug prediction. For 5 of 7 experimental datasets, the models using our SDG-based semantic metrics could achieve the best prediction performance in terms of the F-measure values (*the highest F-measure has been colored in each set of experiments*). Using *Lucene 2.0→2.2* as an example, SDG+LR could obtain the best performance with a F-measure value of 0.847. Therefore, we believe that our SDG-based metrics outperform the semantic features learned by DBN-based models on bug prediction.

Table VI reports the comparison results between the SDG-based models and the CAP-CNN models. The CAP-CNN is a deep learning model that could automatically learn semantic features from a project's source code. Unlike the other approaches, which need to apply external classifiers for bug prediction, CAP-CNN model can directly leverage the learned features for predicting bug-prone files. From Table VI, we can get the following observations:

- When being compared with the ***CAP-CNN*** models, both types of ***SDG+ADTree*** and ***SDG+NB*** models could just obtain a better prediction performance in 3 of 7 comparisons.
- But the ***SDG+LR*** models could achieve a better prediction performance than the ***CAP-CNN*** models in 4 of 7 comparisons.

When taking all experiments into account, the SDG-based models outperfrom the CAP-CNN models in just 10 of 21 experiments. However, for 5 of 7 experimental datasets, the models using our SDG-based semantic metrics could achieve a better prediction performance than CAP-CNN models (*the highest F-measure has been colored in each set of experiments*). In the experiments of *Log4i 2.0→2.2 and Lucene 2.2→2.4*, CAP-CNN could achieve the best performance. But the best performances in the other 5 sets of experiments are from the SDG-based models.

Based on the above results, we believe our SDG-based semantic metrics outperform the semantic features that are learned by the state-of-the-art models on bug prediction.

Answer to RQ3: Compared to the state of the arts, the models using our SDG-based semantics metrics perform better on bug prediction for most of experimental datasets.

RQ4: What is the performance of our approach in terms of execution time and space?

To answer this question, we examine the time cost and memory space cost for the generation process of the SDG-based semantic metrics. As introduced in Section IV, the whole process consists of LSA implementation, semantic similarity matrix and SDG generations, and semantic metrics extraction. The time complexity of SDG generations and semantic metrics extractions are both $O(n^2)$, where n is the number of files in a project. For each project, we repeatedly run each experiment for 10 times and calculated the average time and space costs. In each case, we need calculate the semantic metrics for both training set and testing set. Thus we take the sum of time cost in two steps to be the execution time, and use the maximum space cost in two steps as the execution space. Our running machine is a laptop with i5-8265U CPU@1.6GHz and a 8GB RAM.

Table VII shows the time cost and the used memory space

TABLE V: Performance of SDG-based Models and DBN-based Models

Project	Datasets	SDG-based			DBN-based		
		DT	NB	LR	DT	NB	LR
Log4j	1.0→1.1	0.620	0.560	0.650	0.701	0.725	0.682
Lucene	2.0→2.2	0.582	**0.803**	**0.847**	0.651	0.632	0.630
Lucene	2.2→2.4	0.545	**0.743**	**0.756**	0.773	0.738	0.629
Synapse	1.0→1.1	**0.616**	**0.551**	**0.616**	0.544	0.479	0.423
Synapse	1.1→1.2	**0.583**	0.546	0.530	0.583	0.579	0.541
Xalan	2.4→2.5	0.444	**0.497**	**0.938**	0.595	0.452	0.565
Xerces	1.2→1.3	**0.761**	**0.731**	**0.777**	0.411	0.380	0.475
Average		0.593	0.633	0.731	0.608	0.569	0.564

TABLE VI: Performance of SDG-based Models and CAP-CNN Models

Project	Datasets	SDG-based			CAP-CNN
		DT	NB	LR	
Log4j	1.0→1.1	0.620	0.560	0.650	0.754
Lucene	2.0→2.2	0.582	0.803	0.847	0.743
Lucene	2.2→2.4	0.545	0.743	0.756	0.771
Synapse	1.0→1.1	0.616	0.551	0.616	0.577
Synapse	1.1→1.2	0.583	0.546	0.530	0.555
Xalan	2.4→2.5	0.444	0.497	0.938	0.631
Xerces	1.2→1.3	0.761	0.731	0.777	0.609

when generating semantic metrics. Using *Log4j 1.0→1.1* as an example, we can observe that, on average, it takes 2.9 minutes and 111.1MB memory to generate semantic metrics for all files. Considering all the studied projects together, we can summarize that, the time cost of automatically generating semantic metrics ranges from 2.9 minutes (*Log4j 1.0→1.1*) to 111.6 minutes (*Xalan 2.4→2.5*). For the memory space cost, from 106.3 MB (*Synapse 1.0→1.1*) to 146.0 MB (*Xalan 2.4→2.5*) memory was used for each of the empirical study. Given these results, we believe that the time and space used by our approach are affordable and our approach could be applied in practice.

TABLE VII: Execution Time and Space for Generating SDG-based Semantic Metrics

Project	Datasets	Time(min)	Space(MB)
Log4j	1.0→1.1	2.9	111.1
Lucene	2.0→2.2	9.0	116.6
Lucene	2.2→2.4	16.8	118.9
Synapse	1.0→1.1	13.5	106.3
Synapse	1.1→1.2	18.5	110.5
Xalan	2.4→2.5	111.6	146.0
Xerces	1.2→1.3	59.4	126.0

Answer to RQ4: With respect to the execution time and space, our approach for generating SDG-based semantic metrics is applicable in practice.

VII. Limitations and Threat to Validity

In this section, we give a brief discussion of the limitations of our approach, and threats to validity.

A. *Limitation*

First, so far we only defined eleven semantic metrics, which is just a small amount compared with the amount of existing syntactic metrics. We thus cannot claim that these eleven metrics could totally represent the semantic characteristics of a source file. However, we have considered both direct dependencies among files and the transitivity of semantic dependency, and the prediction results have demonstrated that the eleven semantic metrics are at least adequate to predict bug-prone files. In addition, our approach is scalable, whenever we identify new metrics, it is easy to extend our approach to incorporate the new metrics for experiments.

Second, since the original implementations of the compared approaches [7], [27] are not released, we directly adopt the prediction results from the compared studies in our comparisons, without reimplementing their approaches. This limits the selection of our experimental datasets. In this paper, to guarantee a fair comparison, we only select the dataset used by the baseline. Since we didn't find the datasets of project Ivy and Poi, and the datasets of project Camel are mismatched, we just analyze five open-source projects with seven sets of experiments. We admit this is both a limitation of our study and a threat to external validity.

B. *Threat to Validity*

First, a threat of construct validity is from the model performance indicator we adopt. The performance indicators could influence the conclusion of defect prediction. In this work, we leverage F-measure to indicate the performance of bug prediction, because it has been widely used in many previous studies [31], [10], [7], [27].

Second, we used a code analysis tool, Understand, to capture the lexicon information of each source file. Consequently, any imprecision in the tool could negatively impacts our analysis. This causes a threat to internal validity. However, our approach does not inherently depend on Understand: any code analysis tool that can extract and export lexicon information, e.g., identifiers, comment, etc. into readable formats could be used for our approach.

Third, a threat to external validity is in our data set. Like we mentioned in above section, we only analyzed five open-source projects with seven sets of experiments, thus we can not assure that the evaluation results can be generalizable to all projects. We plan to apply our approach on more open-source or industrial projects having different sizes and in different domains in the future.

VIII. Conclusion

In this paper, we have formally defined a suite of semantic metrics, and created a tool to automatically generate the semantic metrics from source files. More specifically, we first presented how to implement LSA for constructing semantic information from source files. Given the semantic information, we generated a semantic similarity matrix and proposed a novel model of Semantic Dependency Graph (SDG), which explicitly depicts semantic dependencies among source files. Based on the SDG, we could extract the defined semantic metrics. Finally, we presented how to use the SDG-based semantic metrics to build bug prediction models.

We have validated the effectiveness of our proposed semantic metrics on bug prediction by conducting seven sets of experiments. The evaluation results have shown that: 1) our proposed SDG-based semantic metrics can be used to develop effective bug prediction models; 2) the SDG-based semantic metrics outperform traditional syntactic metrics on bug prediction; 3) Compared to the state of the art that learn semantic features automatically, our SDG-based approach could build bug prediction models with better performance; 4) In terms of execution time and space, our approach is affordable in practice. In conclusion, we believe our approach bridges the gap between semantics and bug prediction. Our proposed SDG-based semantic metrics are useful for building promising bug prediction models.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under the grant No. 62002129, the Hubei Provincial Natural Science Foundation of China under the grant No. 2020CFB473, and the Fundamental Research Funds for the Central Universities under the grant No. CCNU19TD003.

References

[1] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[2] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th International Conference on Software Engineering*, pp. 452–461, 2006.

[3] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, 2007.

[4] M. Tim, B. Andrew, M. Andrian, Z. Thomas, and C. David, "Local vs. global models for effort estimation and defect prediction," in *Proc. 26thIEEE/ACM International Conference on Automated Software Engineering*, pp. 343–351, 2011.

[5] 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*, ESEM '12, pp. 171–180, 2012.

[6] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering*, p. 414423, 2014.

[7] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 297–308, 2016.

[8] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, Dec. 1976.

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994.

[10] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 45–54, 2013.

[11] T. T. Nguyen, H. A. Nguyen, N. H.Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, p. 383392, 2009.

[12] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, p. 858868, 2015.

[13] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, p. 306315, 2005.

[14] A. Hindle, E. T. Barr, Z. Su, M. G. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, p. 837847, 2012.

[15] T. K. Landauer and S. T.Dumais, "A solution to Plato's problem: The latent semantic analysis theory of the acquisition, induction, and representation of knowledge," *Psychological Review*, vol. 104, pp. 211–240, 1997.

[16] K. S. Jones, "Index term weighting," *Information Storage and Retrieval*, vol. 9, no. 11, pp. 619 – 633, 1973.

[17] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numer. Math.*, vol. 14, no. 5, p. 403420, 1970.

[18] B. Croft, D. Metzler, and T. Strohman, *Search Engines: Information Retrieval in Practice*. Pearson, 1st ed., 2009.

[19] M. Lan, C. L. Tan, J. Su, and Y. Lu, "Supervised and traditional term weighting methods for automatic text categorization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 4, pp. 721–735, 2009.

[20] R. W. Selby and V. R. Basili, "Analyzing error-prone system structure," *IEEE Transactions on Software Engineering*, vol. 17, pp. 141–152, Feb. 1991.

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

[22] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[23] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 14–24, 2012.

[24] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, p. 169180, 2019.

[25] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 404–415, 2016.

[26] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Improving bug localization with an enhanced convolutional neural network," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 338–347, 2017.

[27] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *2018 IEEE International Conference on Data Mining (ICDM)*, pp. 1049–1054, 2018.

[28] G. E. Box and R. D. Meyer, "An analysis for unreplicated fractional factorials," *Technometrics*, vol. 28, no. 1, pp. 11–18, 1986.

[29] 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*, 2010.

[30] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1054–1068, 2013.

[31] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 382–391, 2013.

[32] N. Chinchor, “Muc-4 evaluation metrics,” in *Proceedings of the 4th Conference on Message Understanding*, pp. 22–29, 1992.

[33] M. Sokolova and G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Inf. Process. Manage.*, vol. 45, no. 4, pp. 427–437, 2009.

[34] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.

[35] F. Rahman and P. Devanbu, “How, and why, process metrics are better,” in *Proceedings of the 2013 International Conference on Software Engineering*, p. 432441, 2013.