Semi-supervised Autoencoder for Defect Prediction

James Hoang
School of Information System
vdthoang.2016@smu.edu.sg

***Abstract*—**

Software defect prediction help developers to find bugs and prioritize their testing efforts. The traditional approaches focus on manually designing features encoding the characteristics of programs and employing different machine algorithms to construct defect prediction models. However, the manual defect features often fail to capture the semantic differences of programs, hence the prediction models may not be accurate. In this paper, we propose bridging the lexical gap between programs' semantics and defect prediction features by projecting natural language statements and programming element. Specifically, we introduce semi-supervised model inspired by autoencoder to learn semantic representation of source code as well as detect program's defect. Our framework is not only learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs), but also construct classification model to detect program elements containing future bug. The experimental results show that our approach significantly outperforms the traditional approaches in defect prediction problem.

Deep learning, defect prediction, bug localization, autoencoder, semi-supervised.

I. Introduction

Software defect prediction techniques [8], [12], [47] have been proposed to detect defects among program elements to help developers to reduce their testing efforts, thus leading to reduce software development costs. Defect prediction tries to construct defect prediction models from software history data, and use these models to predict whether new instances of code regions, e.g., files, changes, and methods, contain defects or any bugs. Traditional approaches try to construct accurate defect prediction models following two different directions: first direction focuses on manually designing a set of features so that it can represent defects more effectively; the second direction focuses on building a new machine learning algorithm to improve the prediction models.

In the past, most researchers have manually designed features to filter buggy source files from non-buggy files. McCabe et al. [22] features focus on a complexity measure for the program elements, CK features [3] based on function and inheritance counts to understand the development of software projects, whereas MOOD features [7] tried to provide an overall assessment of a software system. The other features are constructed based on source code changes like, number of lines of code added, removed, etc. [12], [4]. On the other hand, many machine learning algorithm have been widely used for software defect prediction, including decision tree, logistic regression, Naive Bayes, etc [13]. However, the traditional approaches fail to distinguish code regions of different semantics.

To bridge the gap between programs' semantic information and features used for defect prediction, Wang et al. [41] employed Deep Belief Network (DBN) [10] to automatically learn features from token vectors extracted from programs' ASTs, and then utilize these features to train a defect prediction model. However, Wang approaches [41] build semantic features and defect prediction model independently. Typically, the semantic features only learn from source files without considering the true label of this program element, hence the defect prediction model may not optimize. To tackle this problem, we propose a semi-supervised autoencoder allowing to extract semantic features and optimize the prediction model in one stage. Our proposed framework takes advantage of antoencoder [31] to construct semi-supervised learning model.

This paper makes the following contributions:

- We propose to leverage a powerful representation learning algorithm, namely deep learning, to learn semantic features from token vectors extracted from programs' ASTs automatically and use these features to optimize defect prediction models.
- Our evaluation results on 28 open source Java projects shows that our approach significantly improve the performance of defect prediction by 5.4% compared to traditional approaches.

The rest of this paper is summarized as follows. Section II briefly presents the defect prediction problem and our semi-supervised learning autoencoder. Section III shows the experimental results of our approaches. Section IV presents threat to validity. Section V and Section VI describe the related work and conclusion of our paper.

II. Proposed Approach

A. Defect Prediction

Figure 1 presents the overall framework of file-level defect prediction. Typically, the defect prediction problem is solved by following two specific steps. The first step is to label the program elements as bug or clean based on post-release defects for each file and then we extract the traditional features of these files. These features are briefly introduced in Section V-A. The second step is to construct the classification model [2] used to predict whether a new program element contains bug or clean.

We refer to the software history used for building models as the training set, whereas the new software used to evaluate the trained models as the test set. The classification model are employed on test set to evaluate the performance of our defect prediction model.

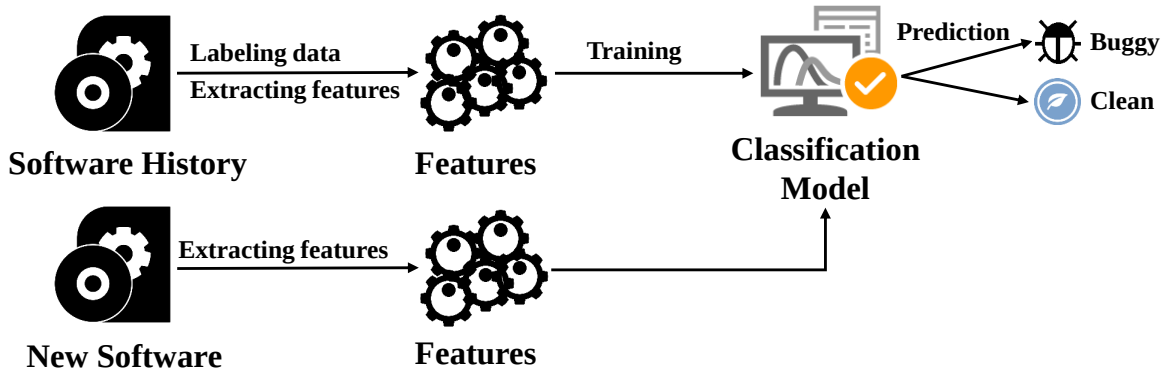

Fig. 1: Defect Prediction Framework

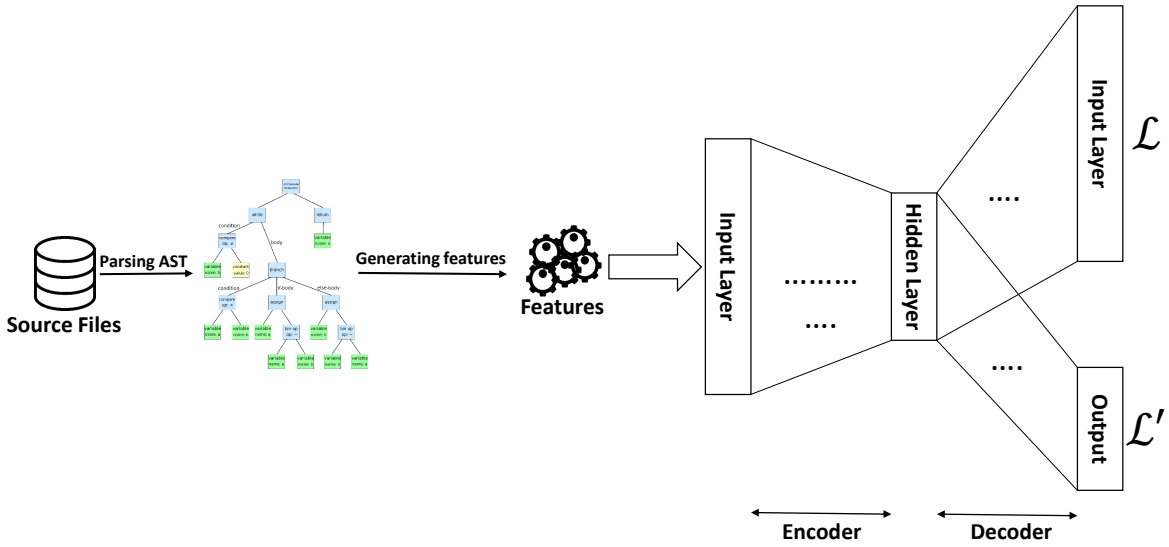

Fig. 2: Semi-supervised AutoEncoder Framework

B. Parsing Source Code and Generating Features

In our approach, we follow Wang et al. [41] to extract source code information to learn semantic features. Typically, the syntactic information from source code is collected based on Java Abstract Syntax Tree (AST) [30]. For each program element, we extract a vector of tokens of the three types of AST nodes: 1) nodes of method invocations and class instance creations, 2) declaration nodes, i.e., method declarations, type declarations, etc. and 3) control-flow nodes such as while statements, catch clauses, if statements, for statements, etc. Note that our semi-supervised learning only takes numerical vectors as inputs, and the lengths of the input vectors are the same. Thus, we apply Wang approaches [41] to map between integers and tokens, and encode token vectors to integer vector. Note that our integer vectors may have different lengths, we append 0 to the integer vectors to make all the lengths consistent and equal to the length of the longest vector. We also note that adding zeros does not affect the results, since it is simply representation transformation to make the vectors acceptable by neural network [41].

C. Semi-supervised Autoencoder

The goal of defect prediction is to detect the potentially source files that may contain bug in the future.

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ denotes the set of program elements of a software project and $\mathcal{Y} = \{y_1, y_2, \ldots, y_n\}$ represents the label of each program element, where n is the number of source files in our collection data. Note that the program element is labeled as 1 if it contains bug, otherwise it will be labeled as 0 which means that it cleans. The source files can be collected from some popular software projects (e.g, ant, camel, lucene, etc.) [1]. Unlike the traditional approaches [43], [41] that independently learn semantic features and construct defect prediction model. Our semi-supervised autoencoder (SSA) combines two different tasks to optimize the classification model for defect prediction problem. Typically, we attempt to learn semantic features function $f : \mathcal{X} \longmapsto \mathcal{X}$ and predict function $f' : \mathcal{X} \longmapsto \mathcal{Y}$, $y_i \in \mathcal{Y} = \{0, 1\}$ indicates whether a source file $x_i \in \mathcal{X}$ contains a bug which can be

[1] http://openscience.us/repo/defect/

obtained by investigating software commit logs and bug report descriptions [6]. These two functions f and f' can be learned by minimizing the following objective function:

$$\min_{f,f'} \sum_{i} \mathcal{L}(f(x_i), x_i) + \mathcal{L}'(f'(x_i), y_i) + \lambda\Omega(f, f') \quad (1)$$

where $\mathcal{L}(\cdot,\cdot)$ and $\mathcal{L}'(\cdot,\cdot)$ are the empirical loss of semantic features and predict functions for the defect prediction problem, respectively. $\Omega(f, f')$ is the regularization terms imposing on the semantic and prediction functions. The trade-off between empirical loss and regularization terms are balanced by λ.

The overall framework of SSA are shown in Figure 2. The SSA model contains two four different parts: parsing abstract syntax tree, generating features, encoder, and decoder. The first two steps are briefly described in Section II-B to feed source files data to our deep neural network. Encoder and decoder are required to learn semantic features as well as defect prediction model. Note that our encoder and decoder steps are inspired by autoencoder [31] which is an unsupervised learning technique. However the original autoencoder only tries to learn the function $f : \mathcal{X} \longmapsto \mathcal{X}$ so that the output values $\hat{\mathcal{X}}$ are similar to input values $\mathcal{X}$. However, SSA attempts to learn semantic features and optimize defect prediction model, thus it takes into account of two functions, i.e., f and f' represent the semantic features and defect prediction respectively. According to Figure 2, our model tries to optimze two different loss functions, i.e., $\mathcal{L}$ and $\mathcal{L}'$ to optimize the defect prediction model. In encoder and decoder steps, we employ a fully connected neural network to fuse middle-level features extracted from source files to generate semantic features, where our network is learn to facilitate the determination on whether the given source code file is related to the given bug report based on the semantic features.

D. Imbalanced Problem in Defect Prediction

In most cases of defect prediction, only few source code files contain bug and a large number of source code files are *clean* [14], hence the imbalanced nature of this type of data increases the learning difficulty of this problem. For this reason, class imbalance learning specializes in tackling classification problems with imbablanced data is helpful for defect prediction problem [42].

To address this problem, we propose to learn the semantic features that may counteract the negative influence of the imbablanced data in the subsequent learning of defect prediction function. Inspired by [46], we introduce an unequal misclassification cost accroding to the imbalance ratio and train the fully connected network in a cost-sensitive manner.

Let r_n denote the ratio cost of incorrectly associating an *clean* source code file to a bug program element and r_p denote the cost of missing a buggy source code file in the training data. The weight of the semi-supervised autoencoder (SSA) networks $\mathcal{W}$ can be learned by minimizing the following objective function following Adam optimization [16].

$$\min_{\mathcal{W}} \sum_{i} \mathcal{L}(f(x_i), x_i) + r_n\mathcal{L}'(x_i, y_i; \mathcal{W})y_i + r_p\mathcal{L}'(x_i, y_i; \mathcal{W})(1-y_i) + \lambda\mathcal{W}^2 \quad (2)$$

where $\mathcal{L}$ and $\mathcal{L}'$ are the loss function for semantic features and defect prediction model, respectively. λ is the trade-off parameter.

E. Setting for Training Semi-supervised Autoencoder

Some deep learning application [10], [31] reports an effective of deep learning models need well-tuned parameters, i.e., 1) the number of hidden layers, 2) the number of nodes in each hidden layer, and 3) the number of iterations. In [41], the authors pointed out that the deep learning model was optimized when they chose 10, 100, 200 as the number of hidden layers, number of nodes in each hidden layer, and number of iterations respectively. For the fair comparison with [41], we use these parameters to train our semi-supervised autoencoder model.

III. Experimental Results

We conduct several experiments to study the performance of the proposed approach and compare it with existing traditional approaches.

A. Evaluation Metrics

To measure defect prediction results, we employ three different metrics: *Precision*, *Recall* and *F1*. These evaluation metrics are widely used to evaluate the performance of defect prediction [23], [24], [29] as well as information retrieval binary classification [20]. Typically, precision is the fraction of retrieved instances that are relevant, recall is the fraction of relevant instances that are retrieved, whereas F1 combines both precision and recall to measure the performance of our model. Below is the equation of these metrics:

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

$$Recall = \frac{TP}{TP + FN} \quad (4)$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (5)$$

where *TP*, *FP*, and *FN* are considered as true positive, false positive, and false negative respectively. True positive is the number of predicted defective files that are truly defective, while false positive is the number of predicted defective files that are actually not defective. False negative records the number of predicted non-defective files that are actually defective. A higher precision makes the manual inspection on a certain amount of predicted defective files find more defects, while an increase in recall can reveal more defects given a project. F1 takes consideration of both precision and recall.

TABLE I: Data descriptions

Project	Number of source files		Average bug rate	
	Training	Testing	Training	Testing
alibaba_druid	446	505	0.078	0.024
apache_calcite	1555	1981	0.177	0.051
apache_commons-math	712	742	0.056	0.038
apache_ignite	684	1204	0.016	0.068
apache_jackrabbit-oak	762	840	0.151	0.056
apache_jclouds	910	972	0.184	0.087
apache_kylin	623	669	0.061	0.142
apache_olingo-odata2	1285	1290	0.040	0.019
apache_phoenix	1378	1408	0.139	0.042
apache_qpid-java	551	504	0.069	0.048
apache_qpid-jms	834	881	0.174	0.102
apache_struts	1429	1485	0.186	0.120
apache_syncope	327	488	0.070	0.115
apache_tika	1321	1209	0.011	0.016
BaseXdb_basex	825	902	0.975	0.092
checkstyle_checkstyle	2234	2428	0.038	0.061
eclipse_jgit	487	493	0.109	0.028
haraldk_TwelveMonkeys	2234	2428	0.038	0.061
izpack_izpack	1430	944	0.326	0.097
jclouds_jclouds	918	1243	0.231	0.092
liquibase_liquibase	200	216	0.220	0.074
metamx_druid	551	504	0.069	0.048
nutzam_nutz	274	663	0.730	0.080
openmicroscopy_bioformats	1014	1227	0.118	0.029
spring-projects_spring-roo	738	727	0.060	0.067
SpringSource_spring-roo	590	753	0.775	0.259
torodb_torodb	1079	1506	0.117	0.069
WindowsAzure_azure-sdk-for-java	400	424	0.030	0.116
Average	921.107	1022.714	0.187	0.075

B. Datasets

We perform several steps to create our benchmark dataset. Firstly, we fetch top 2,500 most popular open-source Java projects from GitHub (sorted by the sum of their number of stars and number of forks). GitHub contains many toy projects, thus, we only consider popular projects similar to prior studies [35], [17]. We only clone the git repositories of projects which use Maven as we leverage Maven to automatically build the projects and construct call graphs from compiled classes. Out of the 2500 projects, 831 projects use Maven. Secondly, we collect 342 Apache Java projects which use Maven and are hosted on GitHub. A er removing the overlapping projects, our dataset contains 1,143 projects.

Next, we ignore projects with less than 150 source files as these projects are too small to employ deep neural netwok. We also filter out projects which have less than 100 tested files. For each project, we have two versions: current version (i.e., latest version as of June 2016) which serves as a test dataset, and previous version (i.e., version one year prior to current version) which serves as a training set. We compile these two versions using *mvn compile:compile*. We ignore projects whose current and/or previous version cannot be compiled. In the end, our dataset has 28 projects.

Table I shows the overview of our dataset containing 28 projects. In average, our data has around 921 source files in training set and more than 1020 program elements in testing set. Average bug rate is 18.7% and 7.5% on training and testing data respectively showing the imbalanced problem in defect prediction [42], [14].

C. Baselines

To evaluate the performance of our approach in defect prediction, we compare with traditional features (see Section V). The first baseline consists some traditional features, including lines of code, cyclomatic complexity [22], total number of methods, total number of public methods, total number of local methods, the depth of inheritance tree, comment density, etc. These features are well described in [8] and have been widely used in previous work [13], [23], [24], [29], [47], [43]. Since they are popular features so we can directly compare our work with previous studies. We note that the traditional features do not contain Abstract Syntax Tree (AST) nodes, which are described in this paper.

The second baseline includes semantic features constructed following Wang et al. approaches [41]. Typically, they tried to employ deep belief network [10] to automatically learn semantic features from token vectors extracted from programs' AST. They also proved that their semantic features significantly improve the performance of defect prediction in software engineering domain.

To build a defect prediction model, we apply three popular machine learning algorithms [2] which are widely used in software engineering domain [41], [42], [13]. They are described as following:

- Decision tree is used to build a predictive model about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). This algorithm is very popular in statistics, data mining and machine learning [37]. Tree models where the target vari-

able can take a finite set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.

- Logistic regression is a predictive analysis and used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables. Logistic regression is used in various applications like: health, statistics, data analysis, etc. [11].
- Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem [39] with strong (naive) independence assumptions between the features. Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. For some types of probability models, Naive Bayes classifiers can be trained very efficiently in a supervised learning setting. In many practical applications, parameter estimation for naive Bayes models uses the method of maximum likelihood [33].

D. Results

This section presents our experimental results. We focus on the performance of our propose approaches, i.e., semi-supervised autoencoder (SSA), and answer the following research question: Does SSA outperform the two baselines: semantic features and traditional features?

To answer this question, we use two different features to build defect prediction models. We run the experiments on 28 sets of software project, each of which uses two versions of the same project (see Table I). The training data, which is the older version of these projects, are used to construct defect prediction models, and the testing data is used to evaluate the performance of our prediction models. Table II shows the precision, recall and F1 of the defect prediction experiments. On average, code features achieve a F1 of 0.354, the semantic features constructed following [41] approaches achieve a F1 of 0.456, and our semi-supervised autoencoder (SSA) achieves a F1 of 0.510. The results demonstrate that we can improve the defect prediction F1 by 5.4% on average on 28 software projects.

We also use code features and semantic features separately to build defect prediction models by using two alternative classification algorithm, i.e., logistic regression and Naive Bayes. Table III shows the precision, recall and F1 scores on SSA vs. two different defect prediction models constructed by code features and semantic features using logistic regression. On average, code features and semantic features achieve F1 of 0.421 and 0.432 respectively. It shows that our SSA model improve the performance of F1 by 8.9% and 7.8% compared to two defect prediction models built using logistic regression algorithm. Table IV presents the results of defect prediction models code features and semantic features using Naive Bayes algorithm, and SSA approaches. We see that our SSA outperforms these two baseline approaches. Table V shows the F1 results of our approach compared to the two code features and semantic features constructed be three different machine learning algorithms, i.e., decision tree, logistic regression, and Naive Bayes. It shows that SSA outperforms the other approaches on 28 software projects in average.

IV. Threats to validity

The projects for this paper contain a large variance in average buggy rates and program elements. Typically, we choose the projects which have more than 200 program elements and more than 10 bugs for running the experiments. However, there is a chance that our projects are not generalizable enough to represent all software projects. Thus, the proposed approach might get better or worse results for the other projects. Furthermore, the proposed semi-supervised autoencoder is only evaluated on open source Java projects. In the future work, we would like to employ the proposed approach on close source software and projects written in different languages (i.e., C++, Python, etc.).

V. Related Work

A. Defect Prediction

The software defect prediction has been studied in the past decade [29], [24], [23], [47], [12], [28], [32], [40]. However, the traditional approaches in defect prediction often manually extract features from historical defect data to construct machine learning classification model [24]. McCabe et al. [22] introduced a graph-theoretic complexity measure for the control program elements which can be considered as a feature in defect prediction. CK features [3] focused on understanding of software development process, while MOOD features [7] provided an overall assessment of a software system to manage the software development projects. These features are widely used in defect prediction. Moser et al. [26] employed the number of revisions of a file, age of a file, number of authors that checked a file, etc. to defect prediction. Nagappan et al. [28] extracted features by considering relationship between its software dependencies, churn measures and post-release failures to build classification model for defect prediction. Lee et al. [19] introduced 56 novel micro interaction metrics (MIMs) leveraging developers' interaction information stored in the Mylyn data, and shown that MIMs significantly improve the performance of defect classification. Jiang [12] showed that individual characteristics and collaboration between developers were useful for defect prediction.

Based on these features, classification models are built to predict the defect among program elements. Elish et al. [5] estimated the capability of Support Vector Machine (SVM) [38] in predicting defect-prone software modules and showed that the prediction performance of SVM is generally better than eight statistical and machine learning models in NASA datasets. Amasaki et al. [1] employed Bayesian belief network (BBN) [21] to predict the amount of residual faults of a software product. Khoshgoftaar et al. [15] showed that the Tree-based machine learning algorithms are efficiently in defect detection. Jing et al. [13] proposed to use the dictionary

TABLE II: Comparison between our approaches vs. two baselines of different features (traditional features and semantic features) using decision tree. P, R, and F1 denote precision, recall, and F1 score respectively. The average F1 scores are highlighted in bold.

Project	Code features			Semantic features			SSA		
	P	R	F1	P	R	F1	P	R	F1
alibaba_druid	0.352	0.951	0.514	0.694	0.051	0.096	0.372	0.083	0.136
apache_calcite	0.561	0.385	0.457	0.457	0.071	0.124	0.466	0.667	0.549
apache_commons-math	0.836	0.351	0.494	0.426	0.388	0.406	0.401	0.393	0.397
apache_ignite	0.428	0.382	0.404	0.426	0.400	0.413	0.511	0.598	0.551
apache_jackrabbit-oak	0.381	0.239	0.294	0.432	0.036	0.066	0.466	0.681	0.553
apache_jclouds	0.424	0.075	0.127	0.392	0.061	0.106	0.509	0.706	0.591
apache_kylin	0.590	0.799	0.679	0.501	0.359	0.418	0.585	0.589	0.587
apache_olingo-odata2	0.485	0.407	0.443	0.466	0.438	0.451	0.414	0.560	0.476
apache_phoenix	0.444	0.310	0.365	0.327	0.528	0.404	0.410	0.559	0.473
apache_qpid-java	0.403	0.225	0.289	0.417	0.327	0.367	0.422	0.625	0.504
apache_qpid-jms	0.462	0.343	0.393	0.299	1.000	0.460	0.468	0.689	0.558
apache_struts	0.316	0.043	0.076	0.326	0.280	0.302	0.547	0.713	0.619
apache_syncope	0.323	0.012	0.024	0.622	0.589	0.605	0.501	0.393	0.441
apache_tika	0.405	0.308	0.350	0.374	0.525	0.437	0.409	0.474	0.439
BaseXdb_basex	0.534	0.248	0.339	0.573	0.878	0.694	0.462	0.964	0.625
checkstyle_checkstyle	0.445	0.814	0.575	0.251	0.211	0.229	0.459	0.476	0.467
eclipse_jgit	0.455	0.364	0.404	0.396	0.857	0.542	0.417	0.429	0.422
haraldk_TwelveMonkeys	0.484	0.510	0.497	1.000	0.517	0.682	0.440	0.476	0.457
izpack_izpack	0.266	0.561	0.361	0.541	0.714	0.616	0.485	0.576	0.527
jclouds_jclouds	0.375	0.071	0.119	0.457	0.408	0.431	0.507	0.605	0.552
liquibase_liquibase	0.467	0.175	0.254	0.479	0.882	0.621	0.449	0.625	0.522
metamx_druid	0.445	0.374	0.407	0.467	0.915	0.619	0.417	0.625	0.500
nutzam_nutz	0.608	0.523	0.562	1.000	0.476	0.645	0.458	0.943	0.616
openmicroscopy_bioformats	0.461	0.286	0.353	0.460	0.781	0.579	0.407	0.583	0.480
spring-projects_spring-roo	0.510	0.250	0.336	0.475	0.497	0.486	0.462	0.510	0.485
SpringSource_spring-roo	0.387	0.208	0.271	0.396	0.930	0.556	0.654	0.831	0.732
torodb_torodb	0.326	0.031	0.056	0.474	0.887	0.618	0.471	0.644	0.544
WindowsAzure_azure-sdk-for-java	0.591	0.389	0.469	0.708	0.938	0.807	0.475	0.469	0.472
Average	**0.456**	**0.344**	**0.354**	**0.494**	**0.534**	**0.456**	**0.466**	**0.589**	**0.510**

TABLE III: Comparison between our approaches vs. two baselines of different features (traditional features and semantic features) using logistic regression. P, R, and F1 denote precision, recall, and F1 score respectively. The average F1 scores are highlighted in bold.

Project	Code features			Semantic features			SSA		
	P	R	F1	P	R	F1	P	R	F1
alibaba_druid	0.289	0.846	0.430	0.277	0.333	0.303	0.372	0.083	0.136
apache_calcite	0.536	0.504	0.519	0.401	0.061	0.106	0.466	0.667	0.549
apache_commons-math	0.507	0.481	0.494	0.806	0.400	0.535	0.401	0.393	0.397
apache_ignite	0.379	0.691	0.490	0.355	0.424	0.386	0.511	0.598	0.551
apache_jackrabbit-oak	0.395	0.537	0.456	0.301	0.053	0.090	0.466	0.681	0.553
apache_jclouds	0.361	0.373	0.367	0.343	0.286	0.312	0.509	0.706	0.591
apache_kylin	0.565	0.572	0.569	1.000	0.318	0.482	0.585	0.589	0.587
apache_olingo-odata2	0.275	0.630	0.383	0.435	0.458	0.446	0.414	0.560	0.476
apache_phoenix	0.370	0.714	0.488	0.521	0.347	0.416	0.410	0.559	0.473
apache_qpid-java	0.301	0.675	0.416	0.410	0.061	0.107	0.422	0.625	0.504
apache_qpid-jms	0.333	0.600	0.428	0.356	0.775	0.488	0.468	0.689	0.558
apache_struts	0.267	0.522	0.353	0.384	0.596	0.467	0.547	0.713	0.619
apache_syncope	0.293	0.134	0.184	1.000	0.541	0.702	0.501	0.393	0.441
apache_tika	0.361	0.731	0.483	0.343	0.875	0.493	0.409	0.474	0.439
BaseXdb_basex	0.460	0.382	0.417	0.439	0.674	0.532	0.462	0.964	0.625
checkstyle_checkstyle	0.423	0.845	0.564	0.329	0.819	0.470	0.459	0.476	0.467
eclipse_jgit	0.317	0.485	0.384	0.368	0.293	0.326	0.417	0.429	0.422
haraldk_TwelveMonkeys	0.386	0.167	0.233	0.465	0.754	0.576	0.440	0.476	0.457
izpack_izpack	0.288	0.152	0.199	0.534	0.605	0.567	0.485	0.576	0.527
jclouds_jclouds	0.338	0.373	0.355	0.403	0.771	0.530	0.507	0.605	0.552
liquibase_liquibase	0.447	0.571	0.502	0.313	0.821	0.453	0.449	0.625	0.522
metamx_druid	0.439	0.583	0.501	0.311	0.847	0.455	0.417	0.625	0.500
nutzam_nutz	0.539	0.658	0.593	0.259	0.760	0.386	0.458	0.943	0.616
openmicroscopy_bioformats	0.329	0.464	0.385	0.364	0.482	0.415	0.407	0.583	0.480
spring-projects_spring-roo	0.366	0.750	0.492	0.581	0.663	0.619	0.462	0.510	0.485
SpringSource_spring-roo	0.344	0.750	0.472	0.337	1.000	0.504	0.654	0.831	0.732
torodb_torodb	0.294	0.138	0.188	0.292	0.972	0.449	0.471	0.644	0.544
WindowsAzure_azure-sdk-for-java	0.418	0.478	0.446	0.332	0.913	0.487	0.475	0.469	0.472
Average	**0.379**	**0.529**	**0.421**	**0.438**	**0.568**	**0.432**	**0.466**	**0.589**	**0.510**

TABLE IV: Comparison between our approaches vs. two baselines of different features (traditional features and semantic features) using Naive Bayes. P, R, and F1 denote precision, recall, and F1 score respectively. The average F1 scores are highlighted in bold.

Project	Code features			Semantic features			SSA		
	P	R	F1	P	R	F1	P	R	F1
alibaba_druid	0.289	0.724	0.413	0.464	0.510	0.486	0.372	0.083	0.136
apache_calcite	0.536	0.333	0.411	0.430	0.340	0.380	0.466	0.667	0.549
apache_commons-math	0.507	0.143	0.223	1.000	0.331	0.497	0.401	0.393	0.397
apache_ignite	0.379	0.408	0.393	0.492	0.542	0.516	0.511	0.598	0.551
apache_jackrabbit-oak	0.395	0.336	0.363	0.349	0.976	0.515	0.466	0.681	0.553
apache_jclouds	0.361	0.129	0.191	0.372	0.068	0.115	0.509	0.706	0.591
apache_kylin	0.565	0.197	0.292	0.417	0.404	0.410	0.585	0.589	0.587
apache_olingo-odata2	0.275	0.259	0.267	0.336	0.830	0.478	0.414	0.560	0.476
apache_phoenix	0.370	0.238	0.290	0.565	0.826	0.671	0.410	0.559	0.473
apache_qpid-java	0.301	0.438	0.357	0.412	0.402	0.407	0.422	0.625	0.504
apache_qpid-jms	0.333	0.457	0.385	0.347	0.536	0.421	0.468	0.689	0.558
apache_struts	0.267	0.174	0.211	0.460	0.600	0.521	0.547	0.713	0.619
apache_syncope	0.293	0.037	0.065	0.308	0.640	0.416	0.501	0.393	0.441
apache_tika	0.361	0.269	0.308	0.462	0.714	0.561	0.409	0.474	0.439
BaseXdb_basex	0.460	0.210	0.288	0.313	0.179	0.227	0.462	0.964	0.625
checkstyle_checkstyle	0.423	0.515	0.465	0.325	0.333	0.329	0.459	0.476	0.467
eclipse_jgit	0.317	0.364	0.339	0.464	0.565	0.510	0.417	0.429	0.422
haraldk_TwelveMonkeys	0.386	0.031	0.058	0.427	0.813	0.560	0.440	0.476	0.457
izpack_izpack	0.288	0.061	0.100	0.373	0.635	0.470	0.485	0.576	0.527
jclouds_jclouds	0.338	0.129	0.187	0.239	1.000	0.386	0.507	0.605	0.552
liquibase_liquibase	0.447	0.333	0.382	0.391	0.537	0.452	0.449	0.625	0.522
metamx_druid	0.439	0.321	0.371	0.462	0.884	0.607	0.417	0.625	0.500
nutzam_nutz	0.539	0.396	0.457	0.347	0.814	0.487	0.458	0.943	0.616
openmicroscopy_bioformats	0.329	0.357	0.342	0.271	0.737	0.396	0.407	0.583	0.480
spring-projects_spring-roo	0.366	0.500	0.423	1.000	0.364	0.534	0.462	0.510	0.485
SpringSource_spring-roo	0.344	0.500	0.408	0.378	0.958	0.543	0.654	0.831	0.732
torodb_torodb	0.294	0.108	0.158	0.388	0.918	0.546	0.471	0.644	0.544
WindowsAzure_azure-sdk-for-java	0.418	0.283	0.338	0.378	0.286	0.325	0.475	0.469	0.472
Average	**0.379**	**0.295**	**0.303**	**0.435**	**0.598**	**0.456**	**0.466**	**0.589**	**0.510**

learning technique to predict software defect. Typically, they introduced a cost-sensitive discriminative dictionary learning (CDDL) approach for software defect classification and prediction.

The main differences between our approach and traditional approaches are as follows. First, existing approaches to defect prediction are based on manually encoded traditional features which are not sensitive to programs' semantic information, while our approach automatically learns semantic features using semi-supervised autoencoder. Second, these features are automatically employed to construct classification model for defect prediction tasks.

B. Deep Learning in Software Engineering

Recently, deep learning algorithms have been widely used to improve research tasks in software engineering. Lam et al. [18] combined deep neural network (DNN) [9] with rVSM [45], a revised vector space model, to improve the performance of bug localization. Raychev et al. [36] o reduced the problem of code completion to a natural-language processing problem of predicting probabilities of sentences and used recurrent neural network [25] to predict the probabilities of the next token. Mou et al. [27] proposed tree-based convolutional neural network (TBCNN) for programming language processing. The results showed that the effectiveness of TBCNN in two different program analysis tasks: classifying programs according to functionality, and detecting code snippets of certain patterns. Pascanu et al. [34] employed recurrent neural network to build malware classification model in software system. Yuan et al. [44] adopted deep belief network (DBN) [10] to predict mobile malware in Android platform. The experimental results showed that deep learning technique is especially suitable for predicting malware in software system.

Yang et al. [43] leveraged DBN to generate features from existing features and used these new features to predict whether a program element contains bugs. It showed that the deep learning algorithm helps to discover more bug than tradition approaches on average across from six large software projects. The existing features were manually designed based on change level: i.e., the number of modified subsystems, code added, code deleted, the number of files change, etc. In 2016, Wang et al. [41] also employed DBN to learn semantic features from source code. However, the existing features were extracted from abstract syntax tree since [40] claimed that Yang features [43] were fail to distinguish the semantic difference among source code. The evaluation on ten popular source projects showed that the semantic features significantly improved the performance of defect detection. Different to the existing works that semantic features and defect prediction model are built independently, thus the semantic features only learn from source code without considering the label of this program element which may decrease the performance of defect prediction model. To tackle this problem, we propose a semi-supervised learning autoencoder allowing to extract

TABLE V: Comparison between our approaches vs. two baselines of different features (traditional features and semantic features) using three different machine learning algorithms, i.e., decision tree, logistic regression and Naive Bayes. The performance is estimated using F1 score. The average F1 scores are highlighted in bold.

Project	Code features			Semantic features			SSA
	DT	LR	NB	DT	LR	NB	
alibaba_druid	0.514	0.430	0.413	0.096	0.303	0.486	0.136
apache_calcite	0.457	0.519	0.411	0.124	0.106	0.380	0.549
apache_commons-math	0.494	0.494	0.223	0.406	0.535	0.497	0.397
apache_ignite	0.404	0.490	0.393	0.413	0.386	0.516	0.551
apache_jackrabbit-oak	0.294	0.456	0.363	0.066	0.090	0.515	0.553
apache_jclouds	0.127	0.367	0.191	0.106	0.312	0.115	0.591
apache_kylin	0.679	0.569	0.292	0.418	0.482	0.410	0.587
apache_olingo-odata2	0.443	0.383	0.267	0.451	0.446	0.478	0.476
apache_phoenix	0.365	0.488	0.290	0.404	0.416	0.671	0.473
apache_qpid-java	0.289	0.416	0.357	0.367	0.107	0.407	0.504
apache_qpid-jms	0.393	0.428	0.385	0.460	0.488	0.421	0.558
apache_struts	0.076	0.353	0.211	0.302	0.467	0.521	0.619
apache_syncope	0.024	0.184	0.065	0.605	0.702	0.416	0.441
apache_tika	0.350	0.483	0.308	0.437	0.493	0.561	0.439
BaseXdb_basex	0.339	0.417	0.288	0.694	0.532	0.227	0.625
checkstyle_checkstyle	0.575	0.564	0.465	0.229	0.470	0.329	0.467
eclipse_jgit	0.404	0.384	0.339	0.542	0.326	0.510	0.422
haraldk_TwelveMonkeys	0.497	0.233	0.058	0.682	0.576	0.560	0.457
izpack_izpack	0.361	0.199	0.100	0.616	0.567	0.470	0.527
jclouds_jclouds	0.119	0.355	0.187	0.431	0.530	0.386	0.552
liquibase_liquibase	0.254	0.502	0.382	0.621	0.453	0.452	0.522
metamx_druid	0.407	0.501	0.371	0.619	0.455	0.607	0.500
nutzam_nutz	0.562	0.593	0.457	0.645	0.386	0.487	0.616
openmicroscopy_bioformats	0.353	0.385	0.342	0.579	0.415	0.396	0.480
spring-projects_spring-roo	0.336	0.492	0.423	0.486	0.619	0.534	0.485
SpringSource_spring-roo	0.271	0.472	0.408	0.556	0.504	0.543	0.732
torodb_torodb	0.056	0.188	0.158	0.618	0.449	0.546	0.544
WindowsAzure_azure-sdk-for-java	0.469	0.446	0.338	0.807	0.487	0.325	0.472
Average	**0.354**	**0.421**	**0.303**	**0.456**	**0.432**	**0.456**	**0.510**

semantic features and construct classification model for defect prediction. We evaluate the effectiveness of our proposed approaches against Wang approaches [40] and the traditional machine learning algorithms (i.e., naive bayes, logistic regression, and random forest).

VI. Conclusion

Our paper presents a semi-supervised autoencoder to learn semantic features as well as optimize defect prediction model. Typically, we take advantage of deep learning autoencoder to learn semantic features from token vectors extracted from programs' ASTs automatically, and optimize these feature to construct classification model for predicting defects. Our evaluation on 28 software projects shows that our approaches could significantly improve the performance of defect prediction compared to two traditional approaches, i.e., code features and semantic features. In the future, we would like to extend our automatically semantic feature generation approach to C/C++ projects for defect prediction. In addition, it would be promising to leverage our approach to automatically build defect prediction model at different levels, i.e., change level, module level, or package level, etc. instead of file level.

VII. Acknowledgments

The authors thank the anonymous researchers for their feedback which help to improve the paper. This work has been partially supported by the National Research Funding of Singapore.

References

[1] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno. A bayesian belief network for assessing the likelihood of fault content. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 215–226. IEEE, 2003.

[2] C. M. Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

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

[4] F. B. e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[5] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[7] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[8] 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.

[9] R. Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[10] G. E. Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[11] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[12] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289. IEEE, 2013.

[13] 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*, pages 414–423. ACM, 2014.

[14] T. M. Khoshgoftaar, K. Gao, and N. Seliya. Attribute selection and imbalanced data: Problems in software defect prediction. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 137–144. IEEE, 2010.

[15] T. M. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 203–214. IEEE, 2002.

[16] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] P. S. Kochhar, D. Wijedasa, and D. Lo. A large scale study of multiple programming languages and code quality. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 563–573. IEEE, 2016.

[18] 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.

[19] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321. ACM, 2011.

[20] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[21] K. T. McAbee, N. P. Nibbelink, T. D. Johnson, and H. T. Mattingly. Bayesian-belief network model.

[22] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[23] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2007.

[24] 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.

[25] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[26] 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.

[27] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *CoRR, abs/1409.5718*, 2014.

[28] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 364–373. IEEE, 2007.

[29] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

[30] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[31] A. Ng. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[32] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong. Topic-based defect prediction (nier track). In *Proceedings of the 33rd international conference on software engineering*, pages 932–935. ACM, 2011.

[33] J.-X. Pan and K.-T. Fang. Maximum likelihood estimation. *Growth Curve Models and Statistical Diagnostics*, pages 77–158, 2002.

[34] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE, 2015.

[35] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[36] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

[37] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[38] J. A. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[39] V. N. Vapnik and V. Vapnik. *Statistical learning theory*, volume 1. Wiley New York, 1998.

[40] 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.

[41] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[42] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[43] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 17–26. IEEE, 2015.

[44] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[45] 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 *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24. IEEE Press, 2012.

[46] Z.-H. Zhou and X.-Y. Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):63–77, 2006.

[47] 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.