Deep Learning for Just-In-Time Defect Prediction

Xinli Yang*, David Lo†, Xin Xia*‡, Yun Zhang*, and Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
zdyxl@zju.edu.cn, davidlo@smu.edu.sg, {xxia, yunzhang28, sunjl}@zju.edu.cn

***Abstract*—Defect prediction is a very meaningful topic, particularly at change-level. Change-level defect prediction, which is also referred as just-in-time defect prediction, could not only ensure software quality in the development process, but also make the developers check and fix the defects in time. Nowadays, deep learning is a hot topic in the machine learning literature. Whether deep learning can be used to improve the performance of just-in-time defect prediction is still uninvestigated.**

In this paper, to bridge this research gap, we propose an approach *Deeper* which leverages deep learning techniques to predict defect-prone changes. We first build a set of expressive features from a set of initial change features by leveraging a deep belief network algorithm. Next, a machine learning classifier is built on the selected features. To evaluate the performance of our approach, we use datasets from six large open source projects, i.e., Bugzilla, Columba, JDT, Platform, Mozilla, and PostgreSQL, containing a total of 137,417 changes. We compare our approach with the approach proposed by Kamei et al. [1]. The experimental results show that on average across the 6 projects, Deeper could discover 32.22% more bugs than Kamei et al's approach (51.04% versus 18.82% on average). In addition, Deeper can achieve F1-scores of 0.22-0.63, which are statistically significantly higher than those of Kamei et al.'s approach on 4 out of the 6 projects.

Keywords*—*Deep Learning, Just-In-Time Defect Prediction, Deep Belief Network, Cost Effectiveness

I. Introduction

To improve software quality, much effort is invested to the process of testing and debugging. However, in most cases, developers usually have limited resources and tight schedules, and thus could not pay too much effort to such process. Defect prediction techniques are proposed to help prioritize software testing and debugging; they can recommend software components that are likely to be defective to developers. Much research has been done on defect prediction; these techniques construct predictive classification models built on features such as lines of code, code complexity and number of modified files [2], [3], [4]. Prior studies mainly focus on predicting defects at coarse granularity level, such as file, package, or module [4], [5], [6].

In recent years, several research studies propose *just-in-time defect prediction techniques* that are able to predict defective changes (i.e., commits to a version control system) [1], [7]. Just-in-time defect prediction is more practical because it can not only ensure software quality in the development process, but also make the developers check and fix the defects just at the time they are introduced. The advantage of just-in-time defect prediction includes: (1) it leads to smaller amount of code to be reviewed because only individual changes (rather than entire files or packages) need to be reviewed [8]; (2) it leads to an easier assignments of developers to fix bugs because we can easily identify the authors of the changes that introduce defects. In a recent work, Kamei et al. perform a large-scale empirical study on just-in-time defect prediction, and propose the usage of the logistic regression algorithm to build a prediction model [1].

Deep learning is one of the most promising area in the machine learning literature [9], and it is adopted in many research areas and proves to be very effective, particularly in image processing [10] and speech recognition [11]. However, whether deep learning could be used to improve the performance of just-in-time defect prediction is still uninvestigated. In this paper, to bridge this research gap, we propose an approach named *Deeper* to address the just-in-time defect prediction problem by leveraging deep learning techniques. *Deeper* contains two phases: a feature selection phase and a machine learning phase. In the feature selection phase, we extract a set of expressive features from an initial set of basic features by leveraging Deep Belief Network [9]. In the machine learning phase, we build a classifier based on the selected features.

To evaluate Deeper, we use two widely used metrics which were also used to evaluate the approach proposed by Kamei et al. [1]: cost effectiveness [12], [13], [14], [15], and F1-score [7], [12], [16], [17]. Cost effectiveness evaluates prediction performance considering a given cost threshold, e.g., a certain percentage of code to inspect. For example, when a team has limited resources to inspect potentially buggy lines of code, it is crucial that by manually inspecting the top percentages of lines that are likely to be buggy, developers can discover as many bugs as possible. We measure cost effectiveness as the percentage of bugs that can be discovered by inspecting the top 20% LOC based on the confidence levels that a change classification technique outputs (PofB20) [3], [1]. In addition, we also evaluate our method using the F1-score [7], [12], [16], [17], which is a summary measure that combines both precision and recall. F1-score is a good evaluation metric when there is enough resources to inspect all predicted buggy changes. A higher F1-score means that a method can detect more buggy changes.

We have performed experiments on 6 large-scale software projects from different communities, i.e., Bugzilla, Columba, JDT, Platform, Mozilla, and PostgreSQL, containing a total of 137,417 changes. We compare our approach with the approach proposed by Kamei et al. [1]. The experimental results show that on average across the 6 projects, Deeper can discover up to 39.96% more bugs than the Kamei et al's approach (58.09% versus 18.13% for Mozilla). In addition, Deeper can achieve

‡Corresponding author.

F1-scores of 0.22-0.63, which are statistically significantly higher than those of Kamei et al.'s approach on 4 out of the 6 projects.

The main contributions of this paper are:

1) To our best knowledge, it is the first time deep learning is used to improve the performance of just-in-time defect prediction. We propose a novel approach Deeper which leverages a Deep Belief Network to achieve a better performance.
2) We compare our method with Kamei et al.'s approach on 6 large software projects. The experiment results show that our method can achieve a better performance than Kamei et al.'s approach.

The rest of our paper is organized as follows. Section II introduces the background and motivation of our work. Section III presents the overall framework of our approach and elaborates the techniques that we use in our approach. Section IV describes our experiments and the results. Section V discusses the related work. Conclusion and future work are presented in the last section.

II. Background

In this section, we first introduce the basic concepts of just-in-time defect prediction in Section II-A. Next, we present the motivation of using deep learning in Section II-B.

A. Just-in-time Defect Prediction

Just-in-time defect prediction aims to predict if a particular file involved in a commit (i.e., a change) is buggy or not. Traditional just-in-time defect prediction techniques typically follow the following steps:

1) **Training Data Extraction.** For each change, label it as buggy or clean by mining a project's revision history and issue tracking system. Buggy change means the change contains bugs (one or more), while clean change means the change has no bug.
2) **Feature Extraction.** Extract the values of various features from each change. Many different features have been used in past change classification studies.
3) **Model Learning.** Build a model by using a classification algorithm based on the labeled changes and their corresponding features.
4) **Model Application.** For a new change, extract the values of various features. Input these values to the learned model to predict whether the change is buggy or clean.

B. Motivation of Using Deep Learning

Kamei et al. propose a just-in-time defect prediction technique which leverages the advantages of logistic regression (LR) [18]. However, Logistic regression has two weaknesses. First, in logistic regression, the contribution of each feature is calculated independently, which means that LR can not combine different features to generate new features. For example, given two features x and y, if $x \times y$ is a more important and really-needed feature, it is not enough to input only x and y because logistic regression can not generate the new feature $x \times y$. Second, logistic regression performs well only when input features and output labels are in linear relation. Due to these two weaknesses, the selection of input features becomes crucial when using Logistic regression. The bad selection of features may be not in linear relation with output labels, leading to bad training performance or even training failure.

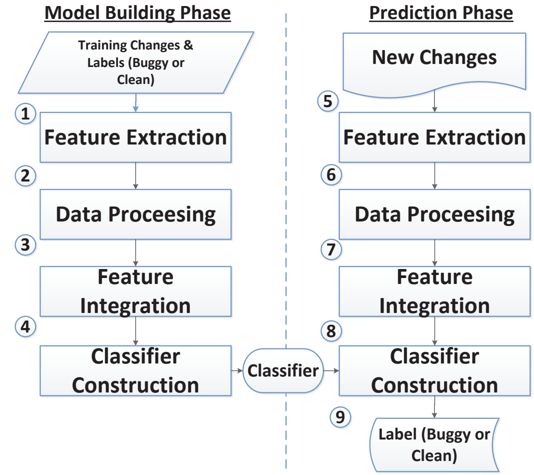


Fig. 1. The Overall Framework of *Deeper*.

The severe problem leads us to adopt Deep Belief Network (DBN) [9], which is one of the state-of-the-art deep learning approaches. The biggest advantage of DBN over Logistic regression is that DBN can generate a more expressive feature set from the initial feature set. The generated feature set, which can include $x \times y$, x^y, and even more complicated nonlinear combination of the initial features, is more powerful to express the nature of a problem. If we input these generated features instead of the initial set of basic features, the above two weaknesses with logistic regression can be overcome.

III. Our Proposed Approach

In this section, we present the details of our proposed approach *Deeper*. We first present the overall framework of Deeper, and then we describe in detail the individual steps in the overall framework.

A. Overall Framework

Figure 1 presents the overall framework of our proposed approach *Deeper*. The framework mainly contains two phases: a model building phase and a prediction phase. In the model building phase, our goal is to build a classifier (i.e., a statistical model) by leveraging deep learning and machine learning techniques from historical changes with known labels (i.e., buggy or clean). In the prediction phase, this classifier would be used to predict if an unknown change would be buggy or clean.

Our framework first extracts a number of features from a set of training changes (i.e., changes with known status) (Step 1). Features are various quantifiable characteristics of changes that could potentially distinguish changes that are buggy from those that are clean. In this paper, we use the 14 basic features proposed by Kamei et al. [1] as shown in Table I. Next, we perform data preprocessing on the collected features

TABLE I. FOURTEEN BASIC CHANGE MEASURES

Name	Description
NS	The number of modified subsystems [19]
ND	The number of modified directories [19]
NF	The number of modified files [20]
Entropy	Distribution of modified code across each file [21]
LA	Lines of code added [22]
LD	Lines of code deleted [22]
LT	Lines of code in a file before the change [23]
FIX	Whether or not the change is a defect fix [24]
NDEV	The number of developers that changed the modified files [24]
AGE	The average time interval between the last and the current change [25]
NUC	The number of unique changes to the modified files [21]
EXP	Developer experience [19]
REXP	Recent developer experience [19]
SEXP	Developer experience on a subsystem [19]

(Step 2)[1]. The data preprocessing contains two sub-steps: data normalization and resampling. In the data normalization sub-step, we transform the values of all features to values in the interval from 0 to 1. Due to the class imbalance phenomenon [26], the number of clean changes is much more than the buggy changes, thus in the resampling sub-step, we perform random under-sampling [27] to make the number of buggy and clean changes equal. Then, a deep learning technique such as Deep Belief Network (DBN) is used to generate and integrate advanced features from the initial features (Step 3)[2]. The advanced features are linear combinations of the initial features. After we generate the advanced features, our framework next constructs a classifier (i.e., a statistical model) based on the advanced features of the training changes (Step 4).[3] In this paper, we use logistic regression [18] to build the classifier.

In the prediction phase, the classifier is then used to predict whether a change with an unknown label is buggy or clean. For each of such changes, our framework first extracts the values of the same set of initial features (Step 5), preprocess the changes (i.e., using data normalization) (Step 6), and generate and integrate the same advanced features as we do in the model building phase (step 7). Next, these features are input into the classifier (Step 8). This step would output the prediction result which is one of the following labels: buggy or clean (Step 9). Note that to mimic reality (we do not have the class labels: buggy or not), we do not perform random under-sampling in the prediction phase.

B. Data Preprocessing

1) Data Normalization: Considering that the values of the 14 basic change features are not in the same order of magnitude, we perform data normalization on the these features. In this paper, we use the min-max method to do the normalization. It transforms all values to values in the interval $[0, 1]$. Given a feature f, we denote the maximum and minumum value for f as $max(f)$ and $min(f)$ respectively. For each value f_i of the feature f, the normalized value z_i is computed as

$$z_i = \frac{x_i - min(f)}{max(f) - min(f)}$$

2) Random Under-Sampling: Random under-sampling [26] is one of the effective approaches to deal with unbalanced data. It randomly deletes data belonging to the majority class (in our case: non-buggy changes) until the amount of data in the majority class is approximately equal to the minority. This step is essential and important for defect prediction because it helps the learned classifier not to be biased to the majority class and thus it can improve the performance of the classifier [27], [28].

C. Feature Integration

Deep Belief Network (DBN) is an advanced deep learning algorithm [9], [29], [10]. It consists of several Restricted Boltzmann's Machines (RBM). The RBM, which is used for feature detection, is a two-layer network. The first layer is the input layer, which contains several visible units. And the second is the hidden layer, which contains several hidden units. The hidden units are also called feature detectors, which output the detected features. The units in the two layers are connected symmetrically, but the units in the same layer are not connected. The numbers of units in the two layers are variable with different research areas and generally is chosen empirically or experimentally. Figure 2 presents the structure of a simple RBM, which contains three visible units and two hidden units.

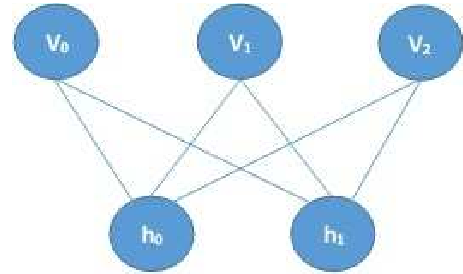


Fig. 2. The Structure of a Restricted Boltzmann's Machine.

The principle of RBM is to minimize the energy of the network. The energy function of RBM is defined as follows:

$$E(v,h) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

In the above equation, w represents the weights connecting visible units and hidden units and a and b are the offsets of the visible and hidden units respectively. Knowing the energy function, we can obtain the joint probability of the two layers of RBM, which is:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)}, where\ Z = \sum_{v,h} e^{-E(v,h)}$$

The probability distribution function about v is:

$$P(v) = \sum_h P(v,h) = \sum_h \frac{e^{-E(v,h)}}{Z}$$

From the above formulas, in order to minimize the energy of RBM, we should maximize $P(v)$, i.e., minimize $-P(v)$.

[1]Detail information of this step is presented in Section III-B.
[2]Detail information of this step is presented in Section III-C.
[3]Detail information of this step is presented in Section III-D.

Therefore, with the partial derivative of $-P(v)$ to w, a and b as well as optimization algorithm such as the method of gradient descent or conjugate gradients, we can obtain optimized w, a and b for RBM.

A single RBM is not the best way to detect features. Instead, we stack several RBMs in the way that the hidden layer of the former RBM is the visible layer of the next RBM. In addition, we connect a classifier to the last RBM in which the hidden layer of the last RBM is the input layer of the classifier. The whole network structure is also called Deep Belief Network (DBN). The power of DBN is it's ability to extract a more expressive feature set.

In our algorithm, we use the DBN which contains three stacked RBMs and a Logistic regression classifier. Since the dimensions of input (in our case, 14 basic features) and output (in our case, 2 labels) are fixed, what we can change is the numbers of hidden layers and the numbers of hidden units. For the number of hidden layers, we choose three hidden layers, which is a general configuration followed by Hiton et al. [9]. For the numbers of hidden units, we try a wide range of numbers using a strategy similar to greedy search. We first change the number of units in the first hidden layer and fix the other two, and then we change the number of units in the second hidden layer and fix the other two, and so on. We find that the performance do not vary much for a wide range of configurations. Therefore, the whole network structure we finally choose has layers of size 14-20-12-12-2, which means that the first RBM has 14 visible units and 20 hidden units, the second RBM has 20 visible units and 12 hidden units, the third RBM has 12 visible units and 12 hidden units, and the classifier has 12 input units and 2 output units.

D. Classifier Construction

Logistic regression [18] models the relationship between features and labels as a parametric distribution $P(y|x)$, where y refers to the label of a data point (in our case: a change), and x refers to the data point represented as a set of features. The parameters of this distribution is directly estimated from the training data. Let $x = \{x_{f_1}, x_{f_2}, ...x_{f_m}\}$ denotes the vector representation of features of a data point x, and x_{f_i} denotes the value of the i-th feature of x, and $W = \{w_0, w_1, w_2, ...w_m\}$ denotes the weight vector associated to the features in x, w_0 is a bias parameter, and $w_i, i \in \{1, 2, ...m\}$ is the weight of the i-th feature of x (i.e., x_{f_i}). Given a new change x, we compute the confidence scores for x to be buggy and clean, denoted as $Conf_{buggy}(x)$ and $Conf_{clean}(x)$, as follows:

$$Conf_{buggy}(x) = \frac{1}{1 + exp(w_0 + \sum_{i=1}^{m} w_i \times x_{f_i})}$$

$$Conf_{clean}(x) = \frac{exp(w_0 + \sum_{i=1}^{m} w_i \times x_{f_i})}{1 + exp(w_0 + \sum_{i=1}^{m} w_i \times x_{f_i})}$$

From the two confidence scores, we compute the output score $Out(x)$ as:

$$Out(x) = \frac{Conf_{buggy} - Conf_{clean}}{LOC(x)}$$

TABLE II. STATISTICS OF THE DATASETS USED IN OUR STUDY

Project	Time	# Instances	% Buggy
Bugzilla	1998.08-2006.12	4620	36%
Columba	2002.11-2006.07	4455	31%
JDT	2001.05-2007.12	35386	14%
Platform	2001.05-2007.12	64250	14%
Mozilla	2000.01-2006.12	98275	5%
PostgreSQL	1996.07-2010.05	20431	25%

In the above equation, $LOC(x)$ refer to the number of LOC changed in x. If $Out(x) \geq 0$, we predict the change as buggy; else we predict it as clean.

IV. EXPERIMENTS AND RESULTS

In this section, we evaluate the effectiveness of *Deeper*. The experimental environment is an Intel(R) Core(TM) T6570 2.10 GHz CPU, 4GB RAM desktop running Windows 7 (32-bit). We first present our experiment setup and evaluation metrics in Sections IV-A and IV-B respectively. We then present 4 research questions and our experiment results that answer the 4 research questions in Section IV-C.

A. Experiment Setup

We evaluate *Deeper* on six datasets from six well-known open source projects, which are Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla and PostgreSQL. These datasets are also used by Kamei et al. [1]. Table II summarizes the statistics of each dataset, containing the period of each dataset, the total number of instances (i.e., changes), and the proportions of the defective changes. Note that due to batch processing of RBM (100 instances per batch) and 10-fold cross validation, we don't use all instances for experiments. Instead, we use the multiples of 1000 instances for simplicity. For example, Bugzilla has 4620 instances and we only use the first 4000 instances in our experiments. Also note that all the datasets are unbalanced. The most unbalanced dataset, Mozilla, contains only 5% defects, while the most balanced dataset, Bugzilla, contains 36% defects.

We use 10 times ten-fold cross validation [18] to evaluate the performance of Deeper. We randomly divide the dataset into 10 folds, in which 9 folds are used as training dataset, and the remaining one fold is used as testing dataset. To further reduce the bias due to training set selection, we run ten-fold cross validation 10 times and record the average performance. Cross validation is a standard evaluation setting, which is widely used in software engineering studies [30], [31].

B. Evaluation Metrics

We use two evaluation metrics to evaluate the performance of our approach. The first one is cost effectiveness and the other is F1-score.

1) Cost Effectiveness: Cost effectiveness is often used to evaluate defect prediction approaches [13], [15], [14]. Cost effectiveness is measured by computing the percentage of buggy changes found when reviewing a specific percentage of the lines of code. To compute cost-effectiveness, given a number of changes, we firstly sort them according to their likelihood to be buggy. We then simulate to review the changes one-by-one from the highest ranked change to the lowest

and record buggy changes found. Using this process we can obtain the percentage of buggy changes found when reviewing different percentages of lines of code (1% to 100%).

TABLE III. CONFUSION MATRIX

	Predicted Buggy	Predicted Clean
True Buggy	TP	FN
True Clean	FP	TN

2) F1-score: The F1-score is a commonly-used measure to evaluate classification performance [18], [16]. It combines Precision and Recall and can be derived from a confusion matrix, as shown in Table III. The confusion matrix lists all four possible prediction results. If an instance is correctly classified as "buggy", it is a true positive (TP); if an instance is misclassified as "buggy", it is a false positive (FP). Similarly, there are false negatives (FN) and true negatives (TN). Based on the four numbers, Precision, Recall and F1-score are calculated. Precision is the ratio of correctly predicted "buggy" instances to all instances predicted as "buggy" (*Precision* = $\frac{TP}{TP+FP}$). Recall is the ratio of the number of correctly predicted "buggy" instances to the actual number of "buggy" instances (*Recall* = $\frac{TP}{TP+FN}$). Finally, F1-score is a harmonic mean of Precision and Recall: *F1-measure* = $\frac{2*Recall*Precision}{Recall+Precision}$. F1-score is often used as a summary measure to evaluate if an increase in precision outweighs a reduction in recall (and vice versa).

C. Research Questions

We compare *Deeper* against two baselines. The first baseline is an approach using a standard Logistic Regression. For this baseline, we ignore unbalanced-data preprocessing and don't use DBN. It is referred to as *LR* in the following text. The second baseline is the approach proposed by Kamei et al. [1]. The approach uses Random Under-Sampling and Logistic Regression but doesn't use DBN. It is referred to as *Kamei et al.'s approach* in the following text. Our experiments are designed to answer the following research questions:

RQ1 How effective is Deeper?

Motivation. To validate the effectiveness of Deeper, we need to compare it with the above two baselines. When comparing with *LR*, we can test the effectiveness of both unbalanced-data preprocessing and DBN. When comparing with *Kamei's approach*, we can test the effectiveness of DBN only.

Approach. We use the above two evaluation metrics, i.e., cost effectiveness and F1-score, to make comparisons. They are commonly-used measures to evaluate the performance of a defect prediction approach. To make our results more convincing, we perform 10-fold cross validation 10 times and report the average results.

For cost effectiveness, we record the percentage of buggy instances found when adding every one percentage of lines of code reviewed. So we will have 100 average values corresponding to the percentage of buggy instances found when reviewing 1% to 100% lines of code. We specifically focus on the percentage of buggy instances found when reviewing 20% lines of code, which is referred to as PofB20 [3]. For F1-score, we calculate the average of the 100 F1-score values that we obtain after performing 10 times 10-fold cross validation. We use this average value to compare with the baselines.

In addition, we also calculate p-value and cliff delta to better investigate whether or not our approach improve the baselines significantly.

TABLE IV. POFB20 VALUES OF DEEPER AND THE TWO BASELINES

Project	LR(%)	Kamei et al.'s (%)	Deeper(%)
Bugzilla	21.35	21.44	42.80
Columba	12.52	12.29	41.00
JDT	18.31	17.79	55.77
Platform	25.71	24.92	61.87
Mozilla	18.85	18.13	58.09
PostgreSQL	17.82	18.38	46.70
Average	19.09	18.82	51.04

TABLE V. PRECISION OF DEEPER AND THE TWO BASELINES

Project	LR	Kamei et al.'s	Deeper
Bugzilla	0.7019	0.5475	0.5557
Columba	0.6312	0.4865	0.4693
JDT	0.4527	0.2490	0.2597
Platform	0.5271	0.2321	0.2640
Mozilla	0.5836	0.1240	0.1321
PostgreSQL	0.6988	0.5036	0.4573
Average	0.5992	0.3571	0.3564

TABLE VI. RECALL OF DEEPER AND THE TWO BASELINES

Project	LR	Kamei et al.'s	Deeper
Bugzilla	0.4026	0.7027	0.7207
Columba	0.3104	0.6486	0.6703
JDT	0.0304	0.6613	0.6883
Platform	0.0320	0.7085	0.7003
Mozilla	0.0397	0.6084	0.6820
PostgreSQL	0.2819	0.6016	0.6799
Average	0.1828	0.6552	0.6903

TABLE VII. F1-SCORE OF DEEPER AND THE TWO BASELINES

Project	LR	Kamei et al.'s	Deeper
Bugzilla	0.5106	0.6147	0.6264
Columba	0.4148	0.5550	0.5493
JDT	0.0568	0.3616	0.3769
Platform	0.0603	0.3496	0.3833
Mozilla	0.0742	0.2058	0.2213
PostgreSQL	0.4014	0.5480	0.5463
Average	0.2530	0.4391	0.4506

Results. Tables IV, V, VI and VII present the PofB20, Precision, Recall and F1-score values of Deeper as compared with those of the two baselines, respectively. From these tables, we can conclude several points.

First, from Table IV, we can see that using our approach, on average, over 50% of the buggy instances can be found by reviewing only 20% of the lines of code, which is a substantial improvement as compared to the results achieved by the two baselines. The PofB20 values of our approach range from 41% to 62%. For each dataset, the values exceed those of the two baselines substantially.

Second, from Tables V to VII, we can find that in terms of Precision, LR is the best performer by achieving an average precision of 60%. However, in terms of Recall, LR performs the worst. Instead, in terms of Recall, Deeper is the best performer by achieving an average recall of 69%. Also, in

terms of F1-score, which is the summary of the above two indicators, Deeper is the best performer by achieving an average F1-score of 45%.

Third, although the precision of LR is the best, its recall value is very low. This is especially the case for three datasets where the proportions of buggy changes are less than 15%. For those datasets, LR's recall values are only about 3%, while Deeper and *Kamei et al.'s approach* achieve much larger recall values of more than 60%. The result indicates that LR is not good enough for defect prediction and unbalanced-data processing is essential and important.

Fourth, when comparing Deeper with *Kamei et al.'s approach*, we find that almost all the recall values of Deeper are higher than those of *Kamei et al.'s approach*, but the F1-score values of *Kamei et al.'s approach* are larger than Deeper in two datasets, which is due to the relatively low precision of our approach in these two datasets. However, considering the setting of defect prediction, recall is much more important than precision, because what we want to find as many buggy changes as possible even though we need to sacrifice a little more review cost to inspect non-buggy instances. In summary, DBN is useful and overall Deeper is more effective than *Kamei et al.'s approach*.

TABLE VIII. MAPPINGS OF CLIFF'S DELTA VALUES TO EFFECTIVENESS LEVELS [32]

Cliff's Delta (δ)	Effectiveness Level
$-1 <= \delta < 0.147$	Negligible
$0.146 <= \delta < 0.33$	Small
$0.33 <= \delta < 0.474$	Medium
$0.474 <= \delta <= 1$	Large

To better demonstrate the superiority of our approach, we perform the Wilcoxon statistical test and compute the p-value. We also compute the Cliff's delta. Wilcoxon statistical test is often used to check if the difference in two means is statistically significant (which corresponds to a p-value of less than 0.05). Cliff's delta is often used to check if the difference in two means are substantial. The range of Cliff's delta is in [-1, 1], where -1 or 1 means all values in one group are smaller or larger than those of the other group, and 0 means the data in the two groups is similar. The mappings between Cliff's delta scores and effectiveness levels are shown in Table VIII. By computing the p-value and Cliff's delta, the extent of which our approach improves over the two baselines can be more rigorously assessed.

TABLE IX. P-VALUES OF DEEPER COMPARED WITH THE TWO BASELINES IN TERMS OF POFB20

Project	**With LR**	**With Kamei et al.'s**
Bugzilla	<2.2e-16	<2.2e-16
Columba	<2.2e-16	<2.2e-16
JDT	<2.2e-16	<2.2e-16
Platform	<2.2e-16	<2.2e-16
Mozilla	<2.2e-16	<2.2e-16
PostgreSQL	<2.2e-16	<2.2e-16

Tables IX, X, IX and X present p-values and Cliff's deltas of Deeper compared with the two baselines for each of the six datasets. From the four tables, we can see the effectiveness of our approach more clearly. In terms of cost effectiveness,

TABLE X. CLIFF'S DELTAS OF DEEPER COMPARED WITH THE TWO BASELINES IN TERMS OF POFB20

Project	**With LR**	**With Kamei et al.'s**
Bugzilla	1(large)	1(large)
Columba	1(large)	1(large)
JDT	1(large)	1(large)
Platform	1(large)	1(large)
Mozilla	1(large)	1(large)
PostgreSQL	1(large)	1(large)

TABLE XI. P-VALUES OF DEEPER COMPARED WITH THE TWO BASELINES IN TERMS OF F1-SCORE

Project	**With LR**	**With Kamei et al.'s**
Bugzilla	<2.2e-16	1.836e-06
Columba	<2.2e-16	0.05783
JDT	<2.2e-16	<2.2e-16
Platform	<2.2e-16	<2.2e-16
Mozilla	<2.2e-16	<2.2e-16
PostgreSQL	<2.2e-16	0.1955

Deeper improves the performance of the baselines significantly and substantially in all datasets. In terms of F1-score, Deeper improves the performance of the baselines significantly and substantially in four out of the six datasets. In the two datasets where the performance of Deeper is worse than that of Kamei et al.'s approach (i.e., Columba and PostgreSQL) the p-values are larger than 0.05 (not significant) and the Cliff's deltas are less than 0.147 (negligible). Thus for these datasets, our approach is not significantly and substantially worse than Kamei et al.'s approach.

Deeper is more effective than the two baselines and DBN is effective in improving the effectiveness of just-in-time defect prediction. On average, by reviewing only 20% lines of code, over 50% of the buggy changes can be found with our approach.

RQ2 How effective is Deeper when different percentages of LOC are inspected?

Motivation. We have validated the effectiveness of Deeper in terms of cost effectiveness and F1-score through the first research question. Deeper has shown a large improvement in cost effectiveness. We want to go further by showing the amount of buggy instances found when reviewing different amount of lines of code using Deeper. Given the same amount of lines of code reviewed, the more buggy instances found, the more useful an approach is. Note that since RQ1 has showed the weakness of LR, we won't compare our approach with LR in this and the following research questions.

Approach. We record the percentage of buggy instances found when adding every one percentage of lines of code reviewed. So we will have 100 average values corresponding to the percentage of buggy instances found when reviewing 1% to 100% lines of code. We can generate a figure whose x-axis represents the percentage of code reviewed and y-axis represents the percentage of defects found for each dataset. In each chart there are two lines, one for Deeper and the other for *Kamei et al.'s approach*.

Results. Figure 3 shows six charts comparing the cost effectiveness of our approach and *Kamei et al.'s approach* for

TABLE XII. CLIFF'S DELTA OF DEEPER COMPARED WITH THE TWO BASELINES IN TERMS OF F1-SCORE

Project	With LR	With Kamei et al.'s
Bugzilla	0.9832(large)	0.2645(small)
Columba	0.9956(large)	-0.0906(negligible)
JDT	1(large)	0.5295(large)
Platform	1(large)	0.9748(large)
Mozilla	1(large)	0.6531(large)
PostgreSQL	1(large)	-0.0484(negligible)

different percentages of LOC inspected. The red solid curve corresponds to Deeper and the blue dashed curve corresponds to *Kamei et al.'s approach*. From the charts, we can see that the red solid curves are always more convex than the blue dashed curves, which means that our approach can always detect more buggy changes than *Kamei et al.'s approach* in the whole range of percentages of LOC inspected. Take the project Platform as an example, when we inspect 20% LOC, Kamei et al.'s approach can find near 25% buggy instances while Deeper can find 62% buggy instances. When we inspect 40% LOC Kamei et al.'s approach can find around 47% buggy instances while Deeper can find 80% buggy instances. Therefore, the performance of our approach is much better than Kamei et al.'s approach in terms of the cost effectiveness.

Deeper can identify more buggy changes than Kamei et al.'s approach for a wide range of lines of code inspected.

RQ3 What is the effect of varying the amount of training data on the effectiveness of Deeper?

Motivation. For some projects, the amount of training data (i.e., changes known to be buggy or non-buggy) can be limited. Thus, in this research question, we want to investigate the effectiveness of Deeper when the amount of training data is reduced.

Approach. In RQ1 and RQ2, we perform 10-fold cross validations which means that 90% of the data are used for training and 10% of data are used for testing. In this RQ, we perform 2-fold to 10-fold cross validations on the datasets. To make the results more convincing, we also perform each kind of cross validation 10 times. For each dataset, we plot two curves on one chart showing the F1-score and cost effectiveness values for 2-folds to 10-folds cross validations.

Results. Figure 4 presents the F1-score (blue dashed line) and cost effectiveness values (red solid line) for different cross validations. In the figure, we notice the fluctuations of the curves are not large. For example, in Bugzilla, its F1-score achieves the maximum of 0.67 when we perform 8-fold cross validation and it has the minimum of 0.59 when we perform 6-fold cross validation. The cost effectiveness are all between 0.42 and 0.44. In addition, there is no extraordinary minimum. For example, in Columba, its F1-score, and cost effectiveness is from 0.49 – 0.53, and 0.38 – 0.41, respectively. Therefore, we can note that Deeper can work with different amount of training data and the performance will not vary much.

Deeper is able to work well for reduced amount of training data.

RQ4 How much time does it take for Deeper to run?

Motivation. Now that we have examined the effectiveness and the applicability of our approach (with reduced training data), we shall test the efficiency of Deeper in the end. The efficiency of an approach is also an important indicator to evaluate whether or not the approach is good enough.

Approach. In order to answer the question, we measure the training and testing time of Deeper. The training time includes the time taken for data preprocessing, feature selection and classifier training. The testing time is the time taken to process the testing dataset until the prediction results are generated. For Just-In-Time Defect Prediction, the testing time is more crucial than training time. Training can be done offline and the resultant statistical model can be used to assign defect labels to many changes. For each dataset, we consider a collection of 4000 changes (3,600 are used to build a statistical model, and 400 are used to test the model).

TABLE XIII. TRAINING TIME OF KAMEI ET AL.'S APPROACH AND DEEPER (IN SECONDS)

Project	Kamei et al.'s Approach	Our Approach
Bugzilla	1.17	13.03
Columba	0.69	10.41
JDT	0.62	8.31
Platform	0.81	9.86
Mozilla	0.64	7.60
PostgreSQL	0.78	10.67
Average	0.79	9.98

TABLE XIV. TESTING TIME OF KAMEI ET AL.'S APPROACH AND DEEPER (IN SECONDS)

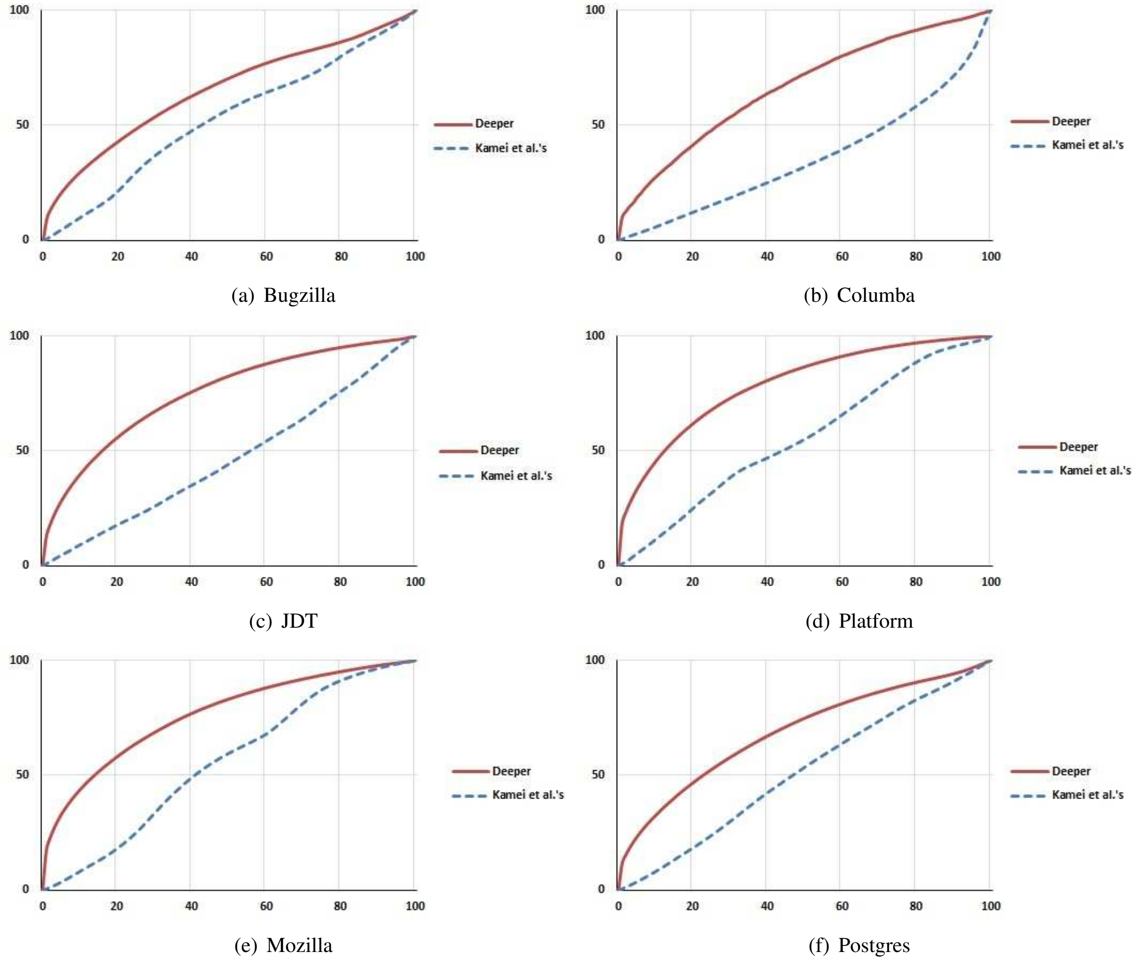
Project	Kamei et al.'s Approach	Our Approach
Bugzilla	0.0022	0.0015
Columba	0.0012	0.0011
JDT	0.0018	0.0031
Platform	0.0021	0.0069
Mozilla	0.0044	0.0098
PostgreSQL	0.0063	0.0028
Average	0.0030	0.0042

Results. Tables XIII and XIV present the training time and testing time of Deeper and *Kamei et al.'s Approach* on the six datasets. From Table XIII, it takes about 10 seconds for our approach to finish training a statistical model, while *Kamei et al.'s approach* only needs less than 1 second. Still, 10 seconds is not an unacceptable cost. In addition, the testing time of both approaches are very small.

Deeper needs around 10 seconds to build a statistical model from 3,600 changes and a negligible amount of time to predict the defect labels of 400 changes.

D. Threats to Validity

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and implementations. Still, there could be errors that we did not notice. Threats to external validity relate to the generalizability of our results. We have evaluated our approach on 137,417 changes from six open source projects. In the future, we plan to reduce this threat further by analyzing even more datasets from more open source and commercial software projects. Threats to construct validity refer to the suitability of our evaluation



(a) Bugzilla (b) Columba (c) JDT (d) Platform (e) Mozilla (f) Postgres

Fig. 3. Cost Effectiveness Trends for the Six Datasets

metrics. We use cost effectiveness and F1-score which are also used by past software engineering studies to evaluate the effectiveness of various prediction techniques [7], [12], [16], [33], [34], [17], [14], [13], [35]. Thus, we believe there is little threat to construct validity.

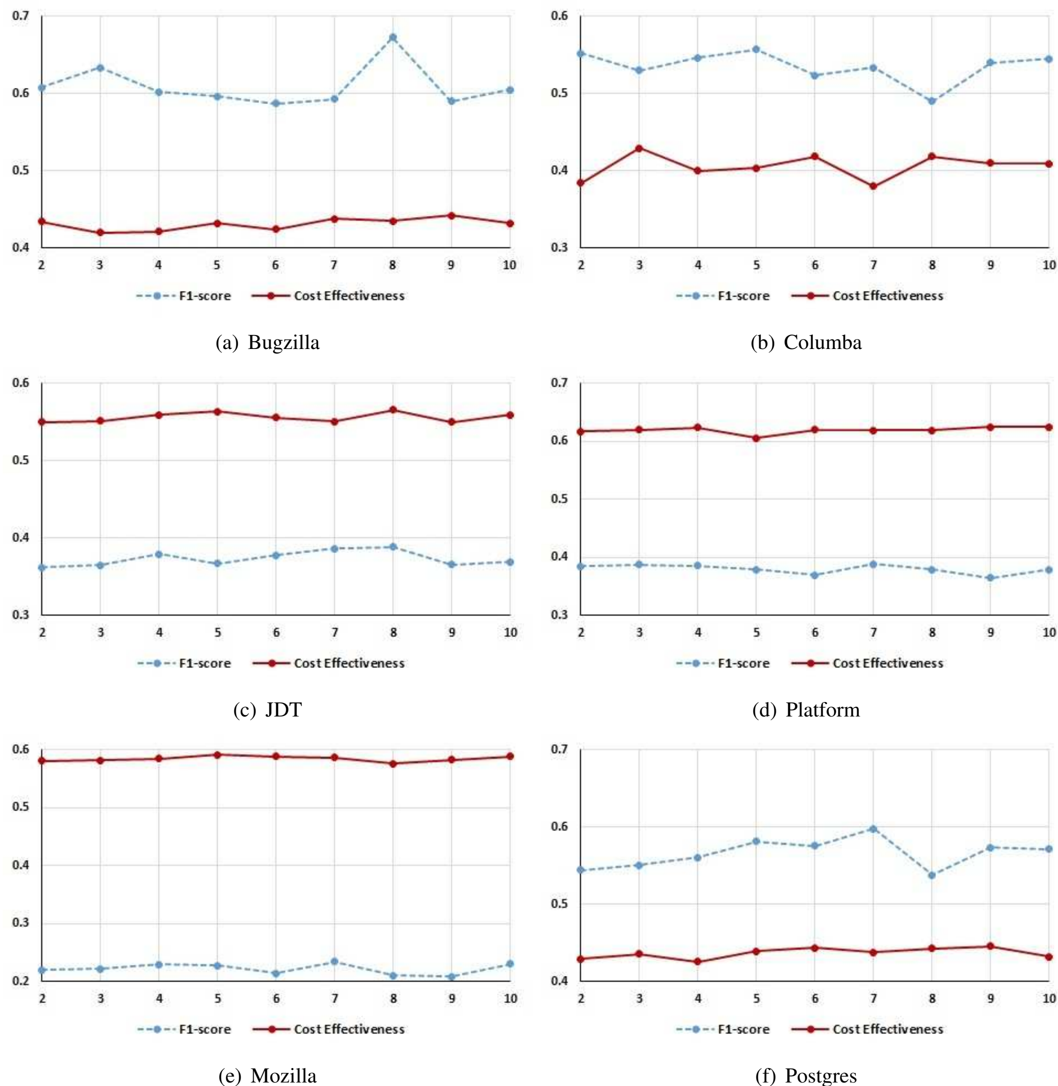
V. Related Work

We classify related work into two parts: studies on defect prediction and studies on deep learning. Due to the page limit, the survey here is by no means complete.

A. Defect Prediction

There are some prior studies on just-in-time defect prediction. Mockus et al. predict defects at change-level in a telecommunication system [19]. They propose a number of measures that characterize a change including change diffusion, change size, change purpose and so on, and use logistic regression to do prediction. All the change measures satisfy three basic conditions: The measure can be computed automatically from changes, the measure can be obtained immediately after changes, and the measure can reflect a property of changes. Kim et al. predict defects at change-level in 12 open source projects [7]. They use Support Vector Machine to predict whether or not a change will lead to a bug. Kamei et al. perform a large-scale empirical study of just-in-time defect prediction [1]. They choose 14 change measures that perform well in traditional defect prediction research and build Logistic Regression models to predict if changes are defective or not.

There are many studies on traditional defect prediction. Zimmermann et al. use network analysis to analyze dependencies between various pieces of code, which can help managers to identify central program units that are more likely to be defective [5]. Zimmermann et al. propose a cross-project defect prediction approach; they train a model on a source project which is selected considering several factors, and use the model on a given target project [36]. Turhan et al. employ a k-nearest neighbor algorithm for cross-project defect prediction, which selects 10 nearest instances from source projects to be used as training data for a target project [6]. D'Ambros et al. present a benchmark for defect prediction and provide an extensive comparison of well-known approaches used for defect prediction in their survey [4]. Rahman et al. analyze code metrics from several different perspectives, and build prediction models across 12 large open source projects to understand the performance, stability, portability and stasis of different sets of metrics for defect prediction [13]. Nam et al. propose TCA+, a novel approach to make feature distributions in source projects similar to that of target projects, which can improve the performance of cross-project defect



(a) Bugzilla (b) Columba

(c) JDT (d) Platform

(e) Mozilla (f) Postgres

Fig. 4. Two-to-ten fold validation results on six datasets

prediction [16].

B. Deep Learning

Hinton et al. pioneer the effectiveness of deep networks [9], [29]. In 2006, Hinton et al. proposed an effective way called ”pretraining” to initialize the weights of deep networks [9]. With pretraining, deep autoencoder networks, which are made up of a stack of Restricted Boltzmanns Machines, can be fine-tuned much efficiently to learn low-dimensional codes. They experiment on many image datasets as well as document datasets and results demonstrate that deep autoencoder networks work much better than principal components analysis (PCA). They also illustrate that the low-dimensional codes obtained by deep autoencoder networks can be used for classification and regression with excellent performance. Also in 2006, Hinton et al. proposed a fast learning algorithm for DBN [29]. They use so-called ”complementary priors” to eliminate the explaining-away effects that make inference difficult in deep belief networks. The algorithm performs greedy and layer-wise training by using complementary priors. They experiment on MNIST dataset and results show that the algorithm gives better performance than the best discriminative learning algorithms. In 2007, Hinton writes a paper about learning feature detectors using multiple-layer networks [10]. He presents the limitations of multilayer generative models and how RBM becomes the key to finding an efficient learning algorithm for deep generative models. He also introduces how to learn many layers of features by composing RBMs in detail.

Nowadays, with the deeper understanding of deep learning, Deep Belief Networks have been studied and used in more and more applications and fields such as image processing, speech recognition, stock prediction and so on [37]. Ranzato et al. present a novel and efficient algorithm to learn a sparse representation for DBN [38]. Mohamed et al. achieve better performance in speech recognition using DBN [39]. Zhu et al. construct a stock decision support system based on DBN for helping stock brokers in buying and selling stocks [40]. In our work, we try to apply DBN to a software engineering task namely Just-In-Time Defect Prediction.

VI. Conclusion and Future Work

In this paper, we propose a deep learning approach for just-in-time defect prediction. The approach first extracts a set of expressive features from an initial set of basic change measures

using Deep Belief Network (DBN), and then trains a classifier based on the extracted features using Logistic Regression. We evaluate our approach on datasets taken from six large open source projects and use two evaluation metrics which are F1-score and cost effectiveness. We compare our approach with two baselines, i.e., a standard Logistic Regression algorithm and the approach proposed by Kamei et al. [27]. The results show that our approach is the best in terms of the two metrics. Our approach achieve an average recall of 69% and an average F1-score of 45%. For cost effectiveness, our approach can identify over 50% defective changes by reviewing only 20% lines of code, which is much more than the defective changes that can be identified by the two baselines.

In the future, we plan to improve the performance of our approach by optimizing parameters of DBN and perform experiments on more datasets to reduce the threats to external validity. We also plan to try other classifiers to find if there exists better classifiers when combined with DBN.

Acknowledgment. This research was partially supported by China Knowledge Centrefor Engineering Sciences and Technology (No. CKCEST-2014-1-5), National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01), and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *TSE*, vol. 39, no. 6, pp. 757–773, 2013.

[2] 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*, vol. 17, no. 4, pp. 375–407, 2010.

[3] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *ASE*, 2013, pp. 279–289.

[4] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[5] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE*, 2008, pp. 531–540.

[6] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[7] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *TSE*, vol. 34, no. 2, pp. 181–196, 2008.

[8] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *ICSM*, 2010, pp. 1–10.

[9] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[10] G. E. Hinton, "Learning multiple layers of representation," *Trends in cognitive sciences*, vol. 11, no. 10, pp. 428–434, 2007.

[11] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams *et al.*, "Recent advances in deep learning for speech research at microsoft," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, 2013, pp. 8604–8608.

[12] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *FSE*, 2012, p. 61.

[13] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *ICSE*, 2013, pp. 432–441.

[14] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *ESEC/FSE*, 2013, pp. 147–157.

[15] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *ISSRE*, 2007, pp. 215–224.

[16] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *ICSE*, 2013, pp. 382–391.

[17] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *ICST*, 2013, pp. 252–261.

[18] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, 2006.

[19] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[20] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *ICSE*, 2006, pp. 452–461.

[21] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*, 2009, pp. 78–88.

[22] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005, pp. 284–292.

[23] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," *TSE*, vol. 35, no. 2, pp. 293–304, 2009.

[24] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *TSE*, vol. 31, no. 6, pp. 511–526, 2005.

[25] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *TSE*, vol. 26, no. 7, pp. 653–661, 2000.

[26] H. He and E. A. Garcia, "Learning from imbalanced data," *TKDE*, vol. 21, no. 9, pp. 1263–1284, 2009.

[27] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *ESEM*. IEEE, 2007, pp. 196–204.

[28] T. M. Khoshgoftaar, X. Yuan, and E. B. Allen, "Balancing misclassification rates in classification-tree models of software quality," *Empirical Software Engineering*, vol. 5, no. 4, pp. 313–330, 2000.

[29] G. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[30] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *CSMR-WCRE*, 2014, pp. 134–143.

[31] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *MSR*, 2013, pp. 287–296.

[32] N. Cliff, *Ordinal methods for behavioral data analysis*, 2014.

[33] X. X. J. S. Yun Zhang, David Lo, "An empirical study of classifier combination for cross-project defect prediction," in *COMPSAC*. IEEE, 2015.

[34] X. X. Y. T. Xiao Xuan, David Lo, "Evaluating defect prediction approaches using a massive set of metrics: An empirical study," in *SAC*. IEEE, 2015.

[35] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *MSR*, 2013, pp. 409–418.

[36] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *ESEC/FSE*, 2009, pp. 91–100.

[37] M. Längkvist, L. Karlsson, and A. Loutfi, "A review of unsupervised feature learning and deep learning for time-series modeling," *Pattern Recognition Letters*, vol. 42, pp. 11–24, 2014.

[38] Y.-l. Boureau, Y. L. Cun *et al.*, "Sparse feature learning for deep belief networks," in *Advances in neural information processing systems*, 2008, pp. 1185–1192.

[39] A.-r. Mohamed, G. E. Dahl, and G. Hinton, "Acoustic modeling using deep belief networks," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, no. 1, pp. 14–22, 2012.

[40] C. Zhu, J. Yin, and Q. Li, "A stock decision support system based on dbns," *Journal of Computational Information Systems*, vol. 10, no. 2, pp. 883–893, 2014.