2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Automatically Learning Semantic Features for Defect Prediction

Song Wang,Taiyue Liu and Lin Tan
Electrical and Computer Engineering, University of Waterloo, Canada
{song.wang, t67liu, lintan}@uwaterloo.ca

ABSTRACT

Software defect prediction, which predicts defective code regions, can help developers find bugs and prioritize their testing efforts. To build accurate prediction models, previous studies focus on manually designing features that encode the characteristics of programs and exploring different machine learning algorithms. Existing traditional features often fail to capture the semantic differences of programs, and such a capability is needed for building accurate prediction models.

用DBN从AST中分析特征

To bridge the gap between programs' semantics and defect prediction features, this paper proposes to leverage a powerful representation-learning algorithm, deep learning, to learn semantic representation of programs automatically from source code. Specifically, we leverage Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs).

Our evaluation on ten open source projects shows that our automatically learned semantic features significantly improve both within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) compared to traditional features. Our semantic features improve WPDP on average by 14.7% in precision, 11.5% in recall, and 14.2% in F1. For CPDP, our semantic features based approach outperforms the state-of-the-art technique TCA+ with traditional features by 8.9% in F1.

1. INTRODUCTION

Software defect prediction techniques [12, 18, 20, 23, 27, 32, 38, 40, 52, 62, 70] have been proposed to detect defects and reduce software development costs. Defect prediction techniques build models from software data, and use the models to predict whether new *instances* of code regions, e.g., files, changes, and methods, contain defects. Efforts of previous studies towards building accurate prediction models fall into two main directions: one is manually designing new *features* or new combinations of features to represent defects more effectively; the other is using new and improved machine learning algorithms. Researchers have manually designed many features to distinguish defective files from non-defective files, e.g., Halstead features [10] based on operator and operand counts, McCabe features [31] based on dependencies, CK features [5] based on function and inheritance counts, etc., MOOD features [11] based on polymorphism factor, coupling factor, etc., code change features [18] include number of lines of code added, removed, etc., and other object-oriented features [7]. Meanwhile, many machine learning algorithms have been adopted for software defect prediction, including Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), Neural Network (NN), and Dictionary Learning [20].

Programs have well-defined *syntax*, which can be represented by Abstract Syntax Trees (ASTs) [15] and have been successfully used to capture programming patterns [44, 46]. In addition, programs have *semantics*, which is hidden deeply in source code [65]. It has been shown that programs' semantic information is useful for tasks such as code completion and bug detection [15, 28, 44, 46, 60]. Such semantic information should also be useful for characterizing defects for improving defect prediction. Specifically, in order to make accurate predictions, the features need to be discriminative: capable of distinguishing one instance of code region from another.

However, existing traditional features cannot distinguish code regions of different semantics. Program files with different semantics can have traditional features with the same values. For example, Figure 1 shows two Java files, `File1.java` and `File2.java`, both of which contain an `if` statement, a `for` statement, and two function calls. Using traditional features to represent these two files, their feature vectors are identical, because these two files have the same source code characteristics in terms of lines of code, function calls, raw programming tokens, etc. However, the semantic information is different. Features that can distinguish such semantic differences should enable the building of more accurate prediction models.

To bridge the gap between programs' semantic information and features used for defect prediction, this paper proposes to leverage a powerful representation-learning algorithm, namely deep learning [17], to *learn semantic representation of programs automatically* and use the representation to improve defect prediction. Specifically, we use Deep Belief Network (DBN) [16] to automatically learn features from token vectors extracted from programs' ASTs, and then utilize these features to train a defect prediction model.

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884804

```
1 int i = 9;
2 if (i == 9) {
3   foo();
4   for (i = 0; i < 10;
        i++) {
5     bar();
6   }
7 }
```

File1.java

```
1 int i = 9;
2 foo();
3 for (i = 0; i < 10; i
      ++) {
4   if (i == 9) {
5     bar();
6   }
7 }
```

File2.java

Figure 1: A Motivating Example

DBN is a generative graphical model, which learns a representation that can reconstruct training data with a high probability. It automatically learns high-level representation of data by constructing a deep architecture [2]. We have seen successful applications of DBN in many fields, including speech recognition [37], image classification [6, 25], natural language understanding [35, 55], and semantic search [54].

To use a DBN to learn features from code snippets, we convert the code snippets into vectors of tokens with structural and contextual information preserved, and use these vectors as input to the DBN. For the two code snippets in Figure 1, the input vectors will be `[..., if, foo, for, bar, ...]` and `[..., foo, for, if, bar, ...]` respectively. Since the vectors of these two files are different, DBN will *automatically* learn features to distinguish these two code snippets (details are in Figure 3 and Section 3.3). 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.
- We leverage the semantic features learned automatically by DBN to improve both within-project defect prediction (WPDP) and cross-project defect prediction (CPDP).
- Our evaluation results on ten open source Java projects show that the automatically generated semantic features improve both WPDP and CPDP. For WPDP, our semantic features achieve an average improvement of precision by 14.7%, recall by 11.5%, and F1 by 14.2% compared to traditional features. For CPDP, our semantic feature based approach outperforms the state-of-the-art technique TCA+ [42] built on traditional features by 8.9% in F1.

The rest of this paper is summarized as follows. Section 2 provides backgrounds on defect prediction and DBN. Section 3 describes our proposed approach to learn semantic features from source code automatically, and leverage these learned features to predict defects. Section 4 shows the experimental setup. Section 5 evaluates the performance of learned semantic features. Section 6 and Section 7 present threats to our work and related work respectively. We conclude this paper in Section 8.

2. BACKGROUND

This section provides the backgrounds of file-level defect prediction and deep belief network.

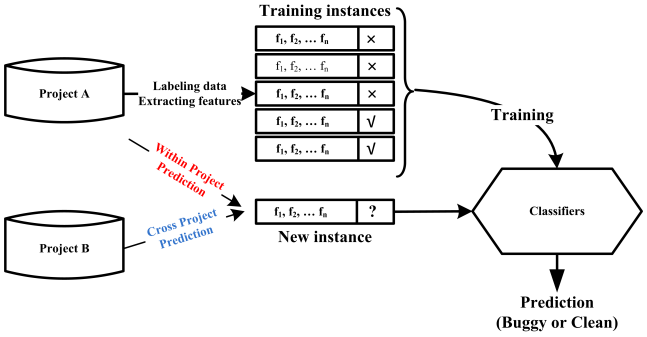

Figure 2: Defect Prediction Process

2.1 Defect Prediction

Figure 2 presents a typical file-level defect prediction process that is adopted by existing studies [20, 27, 34, 41, 42, 51, 64]. The first step is to label data as buggy or clean based on post-release defects for each file. A file is *buggy* if the file contains bugs. Otherwise, the file is *clean*. The second step is to collect corresponding traditional features of these files. Instances with features and labels are used to train machine learning classifiers. Finally, trained models are used to predict new instances as buggy or clean.

We refer to the set of instances used for building models as the *training set*, whereas the set of instances used to evaluate the trained models as the *test set*. As shown in Figure 2, when performing within-project defect prediction (following existing work [41], we call this WPDP), the training and test sets are from the same project A. When performing cross-project defect prediction (following existing work [41] we call this CPDP), prediction models are trained by training set from a project A (source), and test set is from a different project B (target).

In this study, we examine the performance of learned semantic features on both WPDP and CPDP.

2.2 Deep Belief Network

A Deep Belief Network is a generative graphical model that uses a multi-level neural network to learn a representation from training data that could reconstruct the semantic and content of input data with a high probability [2]. DBN contains one *input layer* and several *hidden layers*, and the top layer is the output layer that used as features to represent input data as shown in Figure 3. Each layer consists of several stochastic nodes. The number of hidden layers and the number of nodes in each layer vary depending on users' demand. In this study, the size of learned semantic features is the number of nodes in the top layer. The idea of DBN is to enable the network to reconstruct the input data using generated features by adjusting weights between nodes in different layers.

DBN models the joint distribution between input layer and the hidden layers as follows:

$$P(x, h^1, ..., h^l) = P(x|h^1)(\prod_{k=1}^{l} P(h^k|h^{k+1})) \quad (1)$$

where x is the data vector from input layer, l is the number of hidden layers, and h^k is the data vector of k^{th} layer ($1 \leq k \leq l$). $P(h^k|h^{k+1})$ is a conditional distribution for the adjacent k and $k+1$ layer.

To calculate $P(h^k|h^{k+1})$, each pair of two adjacent layers in DBN are trained as a Restricted Boltzmann Machines (RBM) [2]. A RBM is a two-layer, undirected, bipartite graphical model where the first layer consists of observed data variables, referred to as *visible nodes*, and the second layer consists of latent variables, referred to as *hidden nodes*. $P(h^k|h^{k+1})$ can be efficiently calculated as:

$$P(h^k|h^{k+1}) = \prod_{j=1}^{n_k} P(h_j^k|h^{k+1}) \quad (2)$$

$$P(h_j^k = 1|h^{k+1}) = sigm(b_j^k + \sum_{a=1}^{n_{k+1}} W_{aj}^k h_a^{k+1}) \quad (3)$$

where n_k is the number of node in layer k, $sigm(c) = \frac{1}{1+e^{-c}}$, b is a bias matrix, b_j^k is the bias for node j of layer k, and W^k is the weight matrix between layer k and $k+1$.

DBN automatically learns W and b matrices using an iteration process. W and b are updated via log-likelihood stochastic gradient descent:

$$W_{ij}(t+1) = W_{ij}(t) + \eta \frac{\partial log(P(v|h))}{\partial W_{ij}} \quad (4)$$

$$b_k^o(t+1) = b_k^o(t) + \eta \frac{\partial log(P(v|h))}{\partial b_k^o} \quad (5)$$

where t is the t^{th} iteration, η is the learning rate, $P(v|h)$ is the probability of the visible layer of a RBM given the hidden layer, i and j are two nodes in different layers of the RBM, W_{ij} is the weight between the two nodes, and b_k^o is the bias on the node o in layer k.

To train the network, one first initializes all W matrices between two layers via RBM and sets the biases b to $\mathbf{0}$. They can be well-tuned with respect to a specific criterion, e.g., the number of training iterations, error rate between reconstructed input data and original input data. In this study, we use the number of training iterations as the criterion for tuning W and b. The well-tuned W and b are used to set up a DBN for generating semantic features for both training and test data. Also, we discuss how these parameters affect the performance of learned semantic features in Section 4.4.

3. APPROACH

In this work, we use DBN to generate semantic features automatically from source code and leverage these features to improve defect prediction. Figure 4 illustrates the workflow of our approach according to the motivating example in Figure 1. Our approach takes tokens from the source code of the training and test sets as input, and generates semantic features from them, which are then used to build and evaluate the models for predicting defects. Specifically, our approach first extracts a vector of tokens from the source code of each file in both the training and test sets. Since DBN requires input data in the form of integer vectors, we build a mapping between integers and tokens and convert the token vectors to integer vectors. To generate semantic features, we first use the integer vectors of the training set to build a DBN. Then, we use the DBN to automatically generate semantic features from the integer vectors of the training and test sets. Finally, based on the generated semantic features, we build defect prediction models from the training set, and evaluate their performance on the test set.

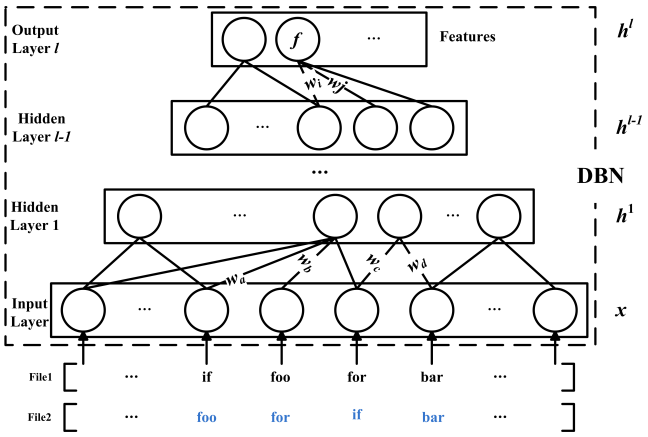

Figure 3: Deep belief network architecture and input instances of File1.java and File2.java. Although the token sets of these two files are identical, the different structural and contextual information between tokens enables DBN to generate different features to distinguish these two files.

Our approach consists of four major steps: 1) parsing source code into tokens, 2) mapping tokens to integer identifiers, which are the expected inputs of DBN, 3) leveraging DBN to automatically generate semantic features, and 4) building defect prediction models and predicting defects using the learned semantic features of the training data and the test data.

3.1 Parsing Source Code

For our study, we need to extract syntactic information from source code for DBN to learn semantic features. We utilize Java Abstract Syntax Tree (AST) to extract syntactic information from source code. Three types of AST nodes are extracted as tokens: 1) nodes of method invocations and class instance creations, e.g., in Figure 3, method `foo()` and `bar()` are recorded as their method names, 2) declaration nodes, i.e., method declarations, type declarations, and enum declarations, and 3) control-flow nodes such as `while` statements, `catch` clauses, `if` statements, `throw` statements, etc. Control-flow nodes are recorded as their statement types, e.g., an `if` statement is simply recorded as `if`. In summary, for each file, we obtain a vector of tokens of the three categories.

We exclude AST nodes that are not one of these three categories, such as assignment and intrinsic type declaration, because they are often method-specific or class-specific, which may not be generalizable to the whole project. Adding them may dilute the importance of other nodes.

Since the names of methods, classes, and types are typically project-specific, methods of an identical name in different projects are either rare or of different functionalities. Thus, for cross-project defect prediction, we extract all three categories of AST nodes, but for the AST nodes in categories 1) and 2), instead of using their names, we use their AST node types such as `method declarations` and `method invocations`. Take project `xerces` as an example. As an XML parser, it consists of many methods named `getXXX` and

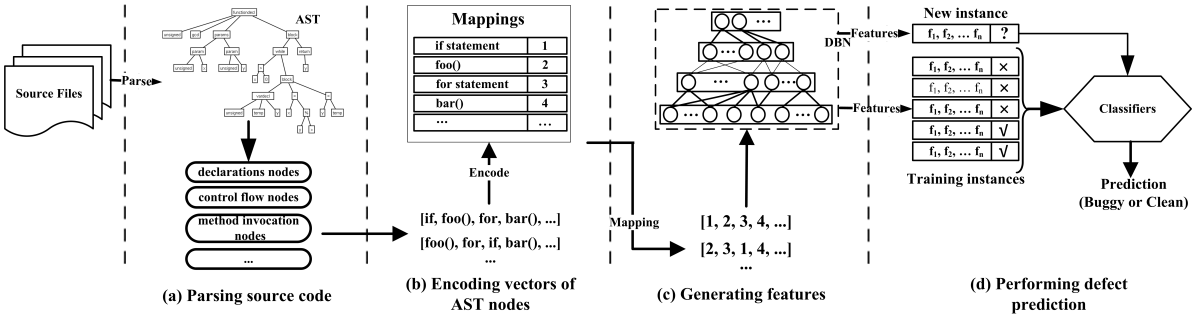

Figure 4: Overview of our proposed DBN-based feature generation and defect prediction

`setXXX`, where `XXX` refers to XML-specific keywords including `charset`, `type`, and `href`. Each of these methods contains only one method invocation statement, which is in form of either `getAttribute(XXX)` or `setAttribute(XXX)`. Methods `getXXX` and `setXXX` do not exist in other projects, while `getAttribute(XXX)` and `setAttribute(XXX)` have different meanings in other projects, so using their names `getAttribute(XXX)` or `setAttribute(XXX)` is not helpful. But it is useful to know that there exist method declaration nodes, and only one method invocation node is under each of these method declaration nodes, since it might be unlikely for a method with only one method invocation inside to be buggy. In this case, compared with using the method names, using the AST node types `method declaration` and `method invocation` is more useful since they can still provide partial semantic information.

3.2 Handling Noise and Mapping Tokens

3.2.1 Handling Noise

Defect data are often noisy and suffer from the mislabeling problem. Studies have shown that such noises could significantly erode the performance of defect prediction [14,22,58]. To prune noisy data, Kim et al. proposed an effective mislabeling data detection approach named *Closest List Noise Identification* (CLNI) [22]. It identifies the k-nearest neighbors for each instance and examines the labels of its neighbors. If a certain number of neighbors have opposite labels, the examined instance will be flagged as noise. However, such approach cannot be directly applied to our data because their approach is based on the Euclidean Distance of traditional numerical features. Since our features are semantic tokens. The difference between the values of two features only indicates that these two features are of different tokens.

To detect and eliminate mislabeling data, and help DBN learn common knowledge between the semantic information of buggy and clean files, we adopt the edit distance similarity computation algorithm [43] to define the distances between instances. The edit distances are sensitive to both the tokens and order among the tokens. Given two token sequences A and B, the edit distance $d(A, B)$ is the minimum-weight series of edit operations that transform A to B. The smaller $d(A, B)$ is, the more similar A and B are.

Based on edit distance similarity, we deploy CLNI to eliminate data with potential incorrect labels. In this study, since our purpose is not to find the best training or test set, we do not spend too much effort on well tuning the parameters of CLNI. We use the recommended parameters and find them work well. In our benchmark experiments with traditional features, we also perform CLNI to remove the incorrectly labeled data.

Additionally, we filter out infrequent AST nodes, which might be designed for a specific file and be hardly generalized to other files. For a project, if the total number of occurrences of a token is less than three, we filter the token out. We encode only the tokens that occur three or more times, which is a common practice in the NLP research field [30].

3.2.2 Mapping Tokens

DBN takes only numerical vectors as inputs, and the lengths of the input vectors must be the same. To use DBN to generate semantic features by using DBN, we first build a mapping between integers and tokens, and encode token vectors to integer vectors. Each token has a unique integer identifier while different method names and class names will be treated as different tokens. Since 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. Adding zeros does not affect the results, and it is simply a representation transformation to make the vectors acceptable by DBN. Taking code snippets in Figure 3 as an example, if we consider only "File1" and "File2", the token vectors for "File1" and "File2" would be mapped to [1, 2, 3, 4] and [2, 3, 1, 4] respectively. Through this encoding process, method invocation information and inter-class information are represented as integer vectors. In addition, some program structure information is preserved since the order of tokens remains unchanged.

3.3 Training DBN and Generating Features

3.3.1 Training DBN

To generate semantic features for distinguishing buggy and clean files, we need to first train DBN by using the training data. As discussed in Section 2, to train an effective DBN for learning semantic features, we need to tune three parameters, which are: 1) *the number of hidden layers*, 2) *the number of nodes in each hidden layer*, and 3) *the number of training iterations*. Existing work that leveraged DBN to generate features for NLP [55] and image recognition [6,25] reported that the performance of DBN-generated features is sensitive to these parameters. We show how we tune these parameters in Section 4.4.

To simplify our model, we set the number of nodes to be the same in each layer. Through these hidden layers and nodes, DBN obtains characteristics that are difficult

to be observed but are capable of capturing semantic differences. For each node, DBN learns probabilities of traversing from this node to the nodes of its top level. Through back-propagation validation, DBN reconstructs the input data using generated features by adjusting weights between nodes in different layers.

DBN requires the values of input data ranging from 0 to 1, while data in our input vectors can have any integer values due to our mapping approach. To satisfy the input range requirement, we normalize the values in the data vectors of the training and test sets by using min-max normalization [66]. In our mapping process, integer values for different tokens are just identifiers. One token with a mapping value of 1 and one token with a mapping value of 2 only means these two nodes are different and independent. Thus, the normalized values can still be used as token identifiers since the same identifiers still keep the same normalized values.

3.3.2 Generating Features

After we train a DBN, both the weights w and the biases b (details are in Section 2) are fixed. We input the normalized integer vectors of the training data and the test data into the DBN respectively, and then obtain semantic features for the training and test data from the output layer of the DBN.

3.4 Building Models and Performing Defect Prediction

After we obtain the generated semantic features for each file in both the training data and the test data, we build and train defect prediction models by following the standard defect prediction process described in Section 2, and then we use the test data to evaluate the performance of the built defect prediction models.

4. EXPERIMENTAL SETUP

We conduct several experiments to study the performance of the proposed semantic features and compare them with existing traditional features. We run experiments on a 2.5GHz i5-3210M machine with 4GB RAM.

4.1 Evaluation Metrics

To measure defect prediction results, we use three metrics: *Precision*, *Recall*, and $F1$. These three metrics are widely adopted to evaluate defect prediction techniques [20, 33, 34, 42, 70]. Here is a brief introduction:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \quad (6)$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \quad (7)$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (8)$$

Precision and recall are composed of three numbers in terms of *true positive*, *false positive*, and *false negative*. 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.

4.2 Evaluated Projects and Data Sets

To facilitate the replication and verification of our experiments, we use publicly available data from the PROMISE data repository. We select all Java open source projects from PROMISE[1] whose version numbers are provided. We need the version numbers of each project because we need to extract token vectors from ASTs of source code to feed our DBN-based feature generation approach. In total, 10 Java projects are collected. Table 1 shows the versions, the average number of files, and the average buggy rate of each project. The numbers of files of the projects range from 150 to 1,046, and the buggy rates of the projects have a minimum value of 13.4% and a maximum value of 49.7%.

4.3 Two Baselines of Traditional Features

To evaluate the performance of semantic features in defect prediction, we compare semantic features with traditional features. Our **first baseline** of traditional features consists of 20 traditional features, including lines of code, operand and operator counts, number of methods in a class, the position of a class in inheritance tree, and McCabe complexity measures, etc. The 20 traditional features are available for PROMISE data, and the work from He et al. [13] contains the full list of the 20 features, which are well described in their Table II. These features and data have been widely used in previous work [20, 33, 34, 42, 70]. We choose the widely used PROMISE data so that we can directly compare our work with previous studies. Note that, for a fair comparison, we also perform the noise removal approach described in Section 3.2.1 on the PROMISE data.

The traditional features from PROMISE do not contain AST nodes, which were used by our DBN models. For a fair comparison, our **second baseline** of traditional features is the AST nodes that were given to our DBN models, i.e., the AST nodes in all files after fixing noise (Section 3.2.1). Each instance is represented as a vector of term frequencies of the AST nodes.

4.4 Parameter Settings for Training a DBN Model

Many DBN applications [6, 25, 37] report that an effective DBN needs 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 this study, since we leverage DBN to generate semantic features, we need to consider the impact of the three parameters. We tune the three parameters by conducting experiments with different values of the parameters on `ant` (1.5, 1.6), `camel` (1.2, 1.4), `jEdit` (4.0, 4.1), `lucene` (2.0, 2.2), and `poi` (1.5, 2.5) respectively. Each experiment has specific values of the three parameters and runs on the five projects individually. Given an experiment, for each project, we use the older version of this project to train a DBN with respect to the specific values of the three parameters. Then, we use the trained DBN to generate semantic features for both the older and newer versions. After that, we use the older version to build a defect prediction model and apply it to the newer version. Lastly, we evaluate the specific values of the parameters by the average F1 score of the five projects in defect prediction.

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

Table 1: Evaluated projects

Project	Description	Releases	Avg Files	Avg Buggy Rate (%)
ant	Java based build tool	1.5,1.6,1.7	488	13.4
camel	Enterprise integration framework	1.2,1.4,1.6	1,046	18.7
jEdit	Text editor designed for programmers	3.2,4.0,4.1	645	19.2
log4j	Logging library for Java	1.0,1.1	150	49.7
lucene	Text search engine library	2.0,2.2,2.4	402	35.8
xalan	A library for transforming XML files	2.4,2.5	992	29.6
xerces	XML parser	1.2,1.3	549	15.7
ivy	Dependency management library	1.4,2.0	311	20.0
synapse	Data transport adapters	1.0,1.1,1.2	220	22.7
poi	Java library to access Microsoft format files	1.5,2.5,3.0	416	40.7

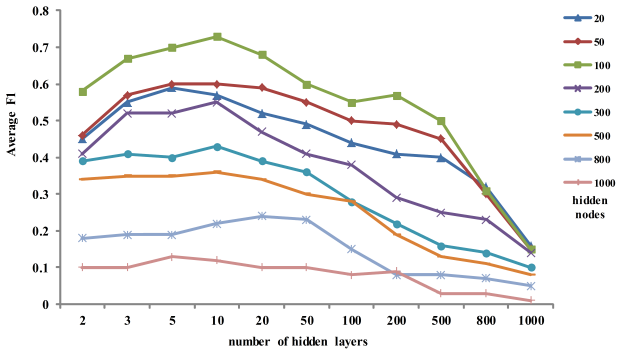

Figure 5: Defect prediction performance with different parameters

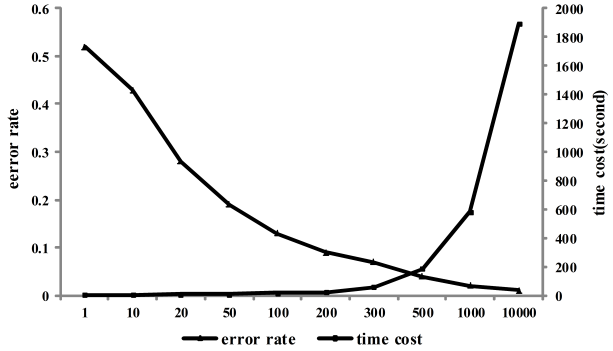

Figure 6: Average error rate and time cost for different numbers of iterations

4.4.1 *Setting the number of hidden layers and the number of nodes in each layer*

Since the number of hidden layers and the number of nodes in each hidden layer interact with each other, we tune these two parameters together. For the number of hidden layers, we experiment with 11 discrete values include 2, 3, 5, 10, 20, 50, 100, 200, 500, 800, and 1,000. For the number of nodes in each hidden layer, we experiment with eight discrete values include 20, 50, 100, 200, 300, 500, 800, and 1,000. When we evaluate these two parameters, we set the number of iterations to 50 and keep it constant.

Figure 5 illustrates the average F1 scores for tuning the number of hidden layers and the number of nodes in each hidden layer together. When the number of nodes in each layer is fixed, with increasing number of hidden layers, all the average F1 scores are convex curves. Most curves peak at the point where the number of hidden layers is equal to 10. If the number of hidden layers remains unchanged, the best F1 score happens when the number of nodes in each layer is 100 (the top line in Figure 5). As a result, we choose the number of hidden layers as 10 and the number of nodes in each hidden layer as 100. Thus, the number of DBN-based features for each project is 100.

4.4.2 *Setting the number of iterations*

The number of iterations is another important parameter for building an effective DBN. During the training process, DBN adjusts weights to narrow down *error rate* between reconstructed input data and original input data in each iteration. In general, the bigger the number of iterations, the lower the error rate. However, there is a trade-off between the number of iterations and the time cost. To balance the number of iterations and the time cost, we choose the same five projects to conduct experiments with ten discrete values for the number of iterations. The values range from 1 to 10,000. We use error rate to evaluate this parameter. Figure 6 demonstrates that, as the number of iterations increasing, the error rate decreases slowly with the corresponding time cost increases exponentially. In this study, we set the number of iterations to 200, with which the average error rate is about 0.098 and the time cost is about 15 seconds.

4.5 Within-Project Defect Prediction

To examine the performance of our semantic features in within-project defect prediction, we build defect prediction models using three machine learning classifiers, including ADTree, Naive Bayes, and Logistic Regression, which have been adopted in previous work [20, 33, 34, 42, 70]. To obtain the training and test data, we use two consecutive versions of each project listed in Table 1. We use the source code of an older version to train DBN and generate the training data. Then we use the trained DBN to generate features for a newer version, which are the test data. We compare our semantic features with the traditional features as described in Section 4.3. For a fair comparison, we use the same classifiers on these traditional features.

Defect data are often imbalanced [57], which might affect the accuracy of defect prediction. Table 1 shows that most of our examined projects have buggy rates less than 50% and so are imbalanced. To obtain optimal defect prediction

models, we perform the re-sampling technique used in existing work [57], i.e., SMOTE, on our training data for both semantic features and traditional features.

4.6 Cross-Project Defect Prediction

Due to the lack of defect data, it is often difficult to build accurate prediction models for new projects. To overcome this problem, cross-project defect prediction techniques train prediction models by using data from mature projects or called *source projects*, and use the trained models to predict defects for new projects or called *target projects*. However, since the features of source projects and target projects often have different distributions, making an accurate and precise cross-project defect prediction is still challenging [41].

We believe that our proposed semantic features can capture the common characteristics of defects, which implies that the semantic features trained from a project can be used to predict a different project, and so applicable in cross-project defect prediction. To measure the performance of the semantic features in cross-project defect prediction, we propose a technique called **DBN C**ross-Project Defect **P**rediction (*DBN-CP*). Given a source project and a target project, DBN-CP first trains a DBN by using the source project and generates semantic features for both the two projects. Then, DBN-CP trains an ADTree based defect prediction model using data from the source project, and then use the built model to perform defect prediction on the target project. We choose TCA+ [42] as our baseline. To compare with TCA+, we randomly pick 1 or 2 versions from each project, in total we have 11 target projects, and for each target project, we randomly select 2 source projects that are different from the target projects. Thus, 22 test pairs are collected.

The reason why we compare with TCA+ is that TCA+ is the state-of-the-art technique that reports the best performance in cross-project defect prediction. Since TCA+ is not publicly available, we have reimplemented our own version of it. In our reproduction, we follow the processes described in [42], we first implement all their proposed five normalization methods and assign them with the same conditions as given in TCA+ paper. We then perform *Transfer Component Analysis* [49] on source projects and target projects together, and map them onto the same subspace while minimizing data difference and maximizing data variance. Finally, we use the source projects and target projects with the new features to build and evaluate ADTree-based prediction models.

5. RESULTS

This section presents our experimental results. We focus on the performance of our proposed semantic features and answer the following research questions (RQ):

RQ1: Do semantic features outperform traditional features for within-project defect prediction?

To answer this question, we use different features to build within-project defect prediction models to compare the impact of three sets of features: semantic features that are automatically learned by DBN, PROMISE features, and AST features. The latter two are the two baselines of traditional features. We conduct 16 sets of within-project defect prediction experiments, each of which uses two versions of the same project (listed in Table 1). The older version is used to train prediction models, and the newer version is used as the test set to evaluate the trained models.

Table 2: Comparison between semantic features and two baselines of traditional features (PROMISE features and AST features) using ADTree. Tr denotes the training set version and T denotes the test set version. P, R, and F1 denote precision, recall, and F1 score respectively and are measured in percentage. The best F1 scores are highlighted in bold.

Project	Versions (Tr->T)	Semantic	PROMISE	AST
		P R F1	P R F1	P R F1
ant	1.5->1.6	88.0 95.1 **91.4**	44.8 51.1 47.7	40.5 51.4 45.3
	1.6->1.7	98.8 90.1 **94.2**	41.8 77.1 54.2	41.2 54.7 47.0
camel	1.2->1.4	96.0 66.4 **78.5**	24.8 75.2 37.3	32.3 55.6 40.2
	1.4->1.6	26.3 64.9 37.4	28.3 63.7 **39.1**	29.7 51.5 38.3
jEdit	3.2->4.0	46.7 74.7 **57.4**	44.7 73.3 55.6	45.8 47.4 46.6
	4.0->4.1	54.4 70.9 **61.5**	46.1 67.1 54.6	50.4 40.4 44.8
log4j	1.0->1.1	67.5 73.0 **70.1**	49.1 73.0 58.7	55.4 38.6 45.5
lucene	2.0->2.2	75.9 56.9 **65.1**	73.3 38.2 50.2	69.5 37.4 48.4
	2.2->2.4	66.5 92.1 **77.3**	70.9 52.7 60.5	65.9 53.1 58.8
xalan	2.4->2.5	65.0 54.8 **59.5**	64.7 43.2 51.8	60.1 43.5 50.5
xerces	1.2->1.3	40.3 42.0 **41.1**	16.0 46.4 23.8	25.5 22.0 23.6
ivy	1.4->2.0	21.7 90.0 **35.0**	22.6 60.0 32.9	31.6 28.6 30.0
synapse	1.0->1.1	46.0 66.7 **54.4**	45.5 50.0 47.6	51.5 45.7 48.4
	1.1->1.2	57.3 59.3 **58.3**	51.1 55.8 53.3	50.7 40.5 49.0
poi	1.5->2.5	76.1 55.2 **64.0**	73.7 44.8 55.8	70.0 31.6 43.5
	2.5->3.0	81.6 79.0 **80.3**	75.0 75.8 75.4	72.1 46.3 55.6
Average		63.0 70.7 **64.1**	48.3 59.2 49.9	49.5 43.0 44.7

Table 2 shows the precision, recall, and F1 of the within-project defect prediction experiments. The highest F1 of the three sets of features are shown in bold. For example, by using `ant 1.6` as the training set, and `ant 1.7` as the test set, the F1 of using semantic features is 94.2%, while the F1 is only 54.2% with the first baseline of traditional features (from PROMISE), and the F1 is 47.0% with the second baseline of traditional features (based on AST nodes). For this comparison, the only difference is the three sets of features, meaning that the same classification algorithm, namely ADTree, the same parameters and the same training and test sets are used.

On average, semantic features achieve a F1 of 64.1%, while the PROMISE features achieve a F1 of 49.9%, and the AST features achieve a F1 of 44.7%. The results demonstrate that by using the semantic features automatically learned by DBN instead of the PROMISE features, we can improve the defect prediction F1 by 14.2% on average on 16 data sets. The average improvement in the precision and recall is 14.7% and 11.5% respectively.

Since the DBN algorithm has randomness, the generated features vary between different runs. Therefore, we run our DBN-based feature generation approach five times for each experiment. Among the runs, the difference in the generated features is at the level of 1.0E-20, which is too small to propagate to precision, recall, and F1. In other words, the precision, recall, and F1 of all five runs are identical.

> The proposed DBN-based approach is effective in automatically learning semantic features, which improves the performance of within-project defect prediction.

RQ1a: Do semantic features outperform traditional features with other classification algorithms? We use semantic features and PROMISE features separately to build defect prediction models by using two alternative

Table 3: Comparison of F1 scores between semantic features and PROMISE features using Naive Bayes and Logistic Regression. Tr denotes the training set version and T denotes the test set version. F1 scores are measured in percentage. The best F1 scores are highlighted in bold.

Project	Version (Tr->T)	Naive Bayes		Logistic Regression	
		Semantic	PROMISE	Semantic	PROMISE
ant	1.5->1.6	**63.0**	56.0	**91.6**	50.6
	1.6->1.7	**96.1**	52.2	**92.5**	54.3
camel	1.2->1.4	**45.9**	30.7	**59.8**	36.3
	1.4->1.6	**48.1**	26.5	34.2	**34.6**
jEdit	3.2->4.0	**58.3**	48.6	**55.2**	54.5
	4.0->4.1	**60.9**	54.8	**62.3**	56.4
log4j	1.0->1.1	**72.5**	68.9	**68.2**	53.5
lucene	2.0->2.2	**63.2**	50.0	**63.0**	59.8
	2.2->2.4	**73.8**	37.8	62.9	**69.4**
xalan	2.4->2.5	**45.2**	39.8	**56.5**	54.0
xerces	1.2->1.3	**38.0**	33.3	**47.5**	26.6
ivy	1.4->2.0	34.4	**38.9**	**34.8**	24.0
synapse	1.0->1.1	47.9	**50.8**	**42.3**	31.6
	1.1->1.2	**57.9**	56.5	**54.1**	53.3
poi	1.5->2.5	**77.0**	32.3	**66.4**	50.3
	2.5->3.0	**77.7**	46.2	**78.3**	74.5
Average		**60.0**	45.2	**59.7**	49.0

classification algorithms—Naive Bayes and Logistic Regression. We conduct 16 sets of within-project defect prediction, where the training sets and the test sets are exactly same as those in RQ1. Table 3 shows the F1 scores of running Naive Bayes and Logistic Regression on semantic features and PROMISE features. We compare the performance of semantic features and PROMISE features under different classification algorithms. The better F1 scores are in bold. Take `ant` as an example, when the model is built on Naive Bayes, by choosing version `1.5` as the training set and `1.6` as the test set, semantic features produce a F1 of 63.0%, which is 7.0% higher than using PROMISE features. For the same example with Logistic Regression as the classification algorithm, semantic features achieve a F1 of 91.6%, while using PROMISE features produces a F1 of 50.6% only. Among the experiments with either Naive Bayes or Logistic Regression as the classification algorithm, semantic features outperform PROMISE features in 14 out of the 16 times. On average, Naive Bayes based defect prediction model with semantic features achieves a F1 of 60.0%, which is a 14.8% improvement over Naive Bayes with PROMISE features. Similarly, the average F1 of using semantic features with Logistic Regression is 59.7%, which is a 10.7% improvement over Logistic Regression with PROMISE features.

> The semantic features automatically learned from DBN improve within-project defect prediction and the improvement is not tied to a particular classification algorithm.

RQ2: Do semantic features outperform traditional features for cross-project defect prediction?

In order to answer this question, we compare our proposed cross-project defect prediction technique DBN-CP with TCA+ [42]. DBN-CP runs on the semantic features that are automatically generated by DBN, while TCA+ uses PROMISE features. For a fair comparison, we also provide a benchmark of within-project defect prediction. We conduct 22 sets of cross-project defect prediction experiments. Each experiment takes two versions separately from two different projects, while one is as the training set and the other one is as the test set. Different from DBN-CP and TCA+, the benchmark of within-project defect prediction uses the data from an older version of the target project as the training set.

Table 4: F1 scores of cross-project defect prediction. F1 scores are measured in percentage. The best F1 scores between DBN-CP and TCA+ are highlighted in bold.

Source	Target	Cross-Project		Within-Project
		DBN-CP	TCA+	Semantic Features
ant1.6	camel1.4	**31.6**	29.2	78.5
jEdit4.1	camel1.4	**69.3**	33.0	
camel1.4	ant1.6	**97.9**	61.6	91.4
poi3.0	ant1.6	47.8	**59.8**	
camel1.4	jEdit4.1	**61.5**	53.7	61.5
log4j1.1	jEdit4.1	**50.3**	41.9	
jEdit4.1	log4j1.1	**64.5**	57.4	70.1
lucene2.2	log4j1.1	**61.8**	57.1	
lucene2.2	xalan2.5	**55.0**	53.0	59.5
xerces1.3	xalan2.5	57.2	**58.1**	
xalan2.5	lucene2.2	**59.4**	56.1	65.1
log4j1.1	lucene2.2	**69.2**	52.4	
xalan2.5	xerces1.3	38.6	**39.4**	41.1
ivy2.0	xerces1.3	**42.6**	39.8	
xerces1.3	ivy2.0	**45.3**	40.9	35.0
synapse1.2	ivy2.0	**82.4**	38.3	
ivy1.4	synapse1.1	**48.9**	34.8	54.4
poi2.5	synapse1.1	**42.5**	37.6	
ivy2.0	synapse1.2	43.3	**57.0**	58.3
poi3.0	synapse1.2	51.4	**54.2**	
synapse1.2	poi3.0	**66.1**	65.1	80.3
ant1.6	poi3.0	**61.9**	34.3	
Average		**56.8**	47.9	63.2

Table 4 contains the F1 scores of DBN-CP, TCA+, and the benchmark within-project defect prediction. The better F1 scores between DBN-CP and TCA+ are in bold. Take an example of the experiment where the source project (training set) is from `camel 1.4` and the target project (test set) is from `ant 1.6`. Running DBN-CP with `camel 1.4` as the training set and `ant 1.6` as the test set produces a F1 of 97.9%, while running TCA+ on the same sets produces a F1 of 61.6%. The within-project defect prediction for this experiment uses semantic features with `ant 1.5` as the training set and `ant 1.6` as the test set, which are the same sets with the experiment that uses `ant 1.6` as the test set in Table 2. In this experiment, running DBN-CP achieves a F1 score of 97.9%, which is even higher than the F1 score of within-project defect prediction with a value of 91.4%.

From the point of average F1, DBN-CP achieves 56.8%, which is 8.9% higher than the 47.9% of TCA+. Compared with within-project defect prediction, DBN-CP makes progress for cross-project defect prediction by reducing the gap to only 6.4%.

> Our proposed DBN-CP improves the performance of cross-project defect prediction. The semantic features learned by DBN are effective and able to capture the common characteristics of defects across projects.

RQ3: What is the time and space cost of the proposed DBN-based feature generation process?

While we conduct the 16 sets of within-project defect prediction experiments for RQ1, we keep track of the time cost and memory space cost for our proposed DBN-based feature generation process (refer to Section 3.3), in which DBN

Table 5: Time and space cost of generating semantic features (s: second)

Project	Generating Features	
	Time (s)	Memory (MB)
ant	15.5	2.8
camel	32.0	5.5
jEdit	18.1	3.3
log4j	10.1	2.2
lucene	11.1	2.4
xalan	29.6	6.2
xerces	13.9	5.8
ivy	8.0	2.2
synapse	8.5	1.9
poi	11.9	4.4

generates semantic features automatically by using noise-handled data. For the other processes, including parsing source code, handling noise, mapping tokens, building models, and predicting defects, they are all common procedures, so we do not analyze their costs.

Table 5 shows the time cost and the memory space cost for generating semantic features. Give an example of `ant`, Table 2 shows that `ant` has two sets of within-project defect prediction experiments, which are `ant 1.5 ->1.6` and `ant 1.6 ->1.7`. On average, it takes the two experiments 15.5 seconds and 2.8MB memory for DBN to generate semantic features for both the training data and the test data.

Among all the projects, the time cost of automatically generating semantic features varies from 8.0 seconds (`ivy`) to 32.0 seconds (`camel`). As for the memory space cost, it takes less than 6.5MB for all the examined projects.

> Using our proposed DBN-based approach to automatically learn semantic features is applicable in practice.

6. THREATS TO VALIDITY

Implementation of TCA+.

For the comparative analysis, we compare our proposed CPDP approach with TCA+ [42], which is the state-of-the-art CPDP technique. Since the original implementation is not released, we have reimplemented our own version of TCA+. Although we strictly followed the procedures described in their work, our new implementation may not reflect all the implementation details of the original TCA+. We test our implementation using data provided by their work, since our implementation could generate the same results, we are confident that our implementation reflects the original TCA+.

In this work we did not evaluate our DBN-based feature generation approach on projects used for evaluating TCA+ [42]. There are two reasons. First, our DBN-based feature generation approach to within-project defect prediction works on data of two different versions from the same project, while datasets used in [42] only provided one version of defect data for each of their eight projects, which are unsuitable for evaluating our approach to within-project defect prediction. Second, some of their examined projects are C/C++ projects. The current implementation of our DBN-based feature generation approach focuses on Java projects, and all of the ten evaluated projects in this work are Java projects. Despite the threats, our comparison should be fair, since we apply TCA+ and our approach on the same projects, which are publicly available data from PROMISE and are biased toward neither TCA+ nor our approach.

Project selection.

The examined projects in this work have a large variance in average buggy rates. We have tried our best to make our dataset general and representative. However, it is still possible that these ten projects are not generalizable enough to represent all software projects. Given projects that are not included in the ten projects, our proposed approach might generate better or worse results. Our proposed semantic features generation approach is only evaluated on open source Java projects. Its performance on closed source software and projects written in other languages is unknown.

7. RELATED WORK

7.1 Software Defect Prediction

There are many software defect prediction techniques [12, 18, 20, 23, 27, 32, 38, 40, 47, 52, 62, 70]. Most defect prediction techniques leverage features that are manually extracted from labeled historical defect data to train machine learning based classifiers [34]. Commonly used features can be divided into static code features and process features [33]. Code features include Halstead features [10], McCabe features [31], CK features [5], and MOOD features [11], which are widely examined and used for defect prediction. Recently, process features have been proposed and used for defect prediction. Moser et al. [38] used the number of revisions, authors, past fixes, and ages of files as features to predict defects. Nagappan et al. [40] proposed code churn features, and shown that these features were effective for defect prediction. Hassan et al. [12] used entropy of change features to predict defects. Lee et al. [27] proposed 56 micro interaction metrics to improve defect prediction. Other process features, including developer individual characteristics [18, 48] and collaboration between developers [27, 34, 51, 64], were also useful for defect prediction.

Based on these features, many machine learning models are built for two different defect prediction tasks—within-project defect prediction and cross-project defect prediction.

7.1.1 Within-Project Defect Prediction

Within-project defect prediction (WPDP) uses training data and test data that are from the same project. Many machine learning algorithms have been adopted for WPDP, including Support Vector Machine (SVM) [8], Bayesian Belief Network [1], Naive Bayes (NB) [59], Decision Tree (DT) [9, 21, 62], and Dictionary Learning [20].

Elish et al. [8] evaluated the capability of SVM in predicting defect-prone software modules, and they compared SVM against eight statistical and machine learning models on four NASA datasets. Amasaki et al. [1] proposed an approach to predict the final quality of a software product by using the Bayesian Belief Network. Tao et al. [59] proposed a Naive Bayes based defect prediction model, they evaluated the proposed approach on 11 datasets from the PROMISE defect data repository. Wang et al. [62] and Khoshgoftaar et al. [21] examined the performance of Tree-based machine learning

algorithms on defect prediction, their results suggested that Tree-based algorithms could help defect prediction. Jing et al. [20] introduced the dictionary learning technique to defect prediction. They proposed a cost-sensitive dictionary learning based approach to improve defect prediction.

7.1.2 Cross-Project Defect Prediction

Due to the lack of data, it is often difficult to build accurate models for new projects. To address this issue, cross-project defect prediction (CPDP) models are trained by using data from other projects. Watanabe et al. [63] proposed an approach for CPDP by transforming the target dataset to the source dataset by using the average feature values. Turhan et al. [61] proposed to use a nearest-neighbor filter to improve CPDP. Nam et al. [42] proposed TCA+, which adopted a state-of-the-art technique called Transfer Component Analysis (TCA) and optimized TCA's normalization process to improve CPDP. They evaluated TCA+ on eight open-source projects, results shown TCA+ significantly improved CPDP. Nam et al. [41] and Jing et al. [19] used different approaches to address the heterogeneous data problem in cross-project defect prediction.

The main differences between our approach and existing approaches for within-project defect prediction and cross-project defect prediction 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 DBN and uses these features to perform defect prediction tasks. Second, since our approach requires only the source code of the training and test projects, it is suitable for both within-project defect prediction and cross-project defect prediction.

7.2 Deep Learning and Semantic Feature Generation in Software Engineering

Recently, deep learning algorithms have been adopted to improve research tasks in software engineering. Yang et al. [68] proposed an approach that leveraged deep learning to generate features from existing features and then used these new features to predict whether a commit is buggy or not. This work was motivated by the weaknesses of logistic regression (LR) that LR can not combine features to generate new features. They used DBN to generate features from 14 traditional change level features: the number of modified subsystems, modified directories, modified files, code added, code deleted, line of code before/after the change, files before/after the change, and several developer experience related features [68].

Our work differs from the above study mainly in three aspects. First, we use DBN to learn semantic features directly from source code, while features generated from their approach are relations among existing features. Since the existing features used cannot distinguish many semantic code differences, the combination of these features would still fail to distinguish the semantic differences. For example, if two changes add the same line at different locations in the same file, the traditional features used cannot distinguish the two changes. Thus, the generated new features, which are combinations of the traditional features, would also fail to distinguish the two changes. Second, we evaluate the effectiveness of our generated features using different classifiers and for both within-project and cross-project defect prediction, while they use LR only for within-project defect prediction. Third, we focus on file level defect prediction, while they work on change level defect prediction.

Other studies leverage deep learning to address other problems in software engineering. Lam et al. [26] combined deep learning algorithms and information retrieval techniques to improve fault localization. Raychev et al. [53] reduced the code completion problem to a natural language processing problem and used deep learning to predict the probabilities of next tokens. White et al. [65] leveraged deep learning to model program languages for code suggestion. Similarly, Mou et al. [39] used deep learning to model programs and showed that deep learning can capture programs' structural information. In addition, deep learning has also been used for malware classification [50, 69], acoustic recognition [24, 36, 37], etc.

Many studies used topic model [3] to extract semantic features for different tasks in software engineering [4, 29, 45, 47, 56, 67]. Nguyen et al. [47] leveraged topic model to generate features from source code for within-project defect prediction. However, their topic model handled each source file as one unordered token sequence. Thus, its generated features cannot capture structural information in a source file. Chen et al. [4] used topic model to generate features for source files to help explain their defect-proneness. Liu et al. [29] proposed to use topic model to generate features from comments and identifiers in source code. Then they further used these features to model class cohesion.

In this work, we leverage DBN to automatically learn semantic features from token vectors extracted from programs' ASTs for both WPDP and CPDP.

8. CONCLUSIONS AND FUTURE WORK

This paper proposes to leverage a representation-learning algorithm, deep learning, to learn semantic representation directly from source code for defect prediction. Specifically, we deploy Deep Belief Network to learn semantic features from token vectors extracted from programs' ASTs automatically, and leverage the learned semantic features to build machine learning models for predicting defects.

Our evaluation on ten open source projects shows that the automatically learned semantic features could significantly improve both within-project and cross-project defect prediction compared to traditional features. Our semantic features improve the within-project defect prediction on average by 14.7% in precision, 11.5% in recall, and 14.2% in F1 comparing with traditional features. For cross-project defect prediction, our semantic features based approach improves the state-of-the-art technique TCA+ built on traditional features by 8.9% in F1.

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 generate features for predicting defects at other levels, such as change level, module level, and package level.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno. A Bayesian Belief Network for Assessing the Likelihood of Fault Content. In *ISSRE'03*, pages 215–226.

[2] Y. Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[4] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan. Explaining software defects using topic models. In *MSR'12*, pages 189–198.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *TSE'94*, 20(6):476–493.

[6] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR'12*, pages 3642–3649.

[7] F. B. e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *JSS'94*, 26(1):87–96.

[8] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *JSS'08*, 81(5):649–660.

[9] N. Gayatri, S. Nickolas, A. Reddy, S. Reddy, and A. Nickolas. Feature selection using decision tree induction in class level metrics dataset for software defect predictions. In *WCECS'10*, pages 124–129.

[10] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[11] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *TSE'98*, 24(6):491–496.

[12] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE'09*, pages 78–88.

[13] Z. He, F. Peters, T. Menzies, and Y. Yang. Learning from open-source projects: An empirical study on defect prediction. In *ESEM'13*, pages 45–54.

[14] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *ICSE'13*, pages 392–401.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE'12*, pages 837–847.

[16] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation'06*, 18(7):1527–1554.

[17] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science'06*, 313(5786):504–507.

[18] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *ASE'13*, pages 279–289.

[19] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *FSE'15*, pages 496–507.

[20] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *ICSE'14*, pages 414–423.

[21] T. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *Software Metrics'02*, pages 203–214.

[22] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *ICSE'11*, pages 481–490.

[23] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498.

[24] S. Kombrink, T. Mikolov, M. Karafiát, and L. Burget. Recurrent neural network based language modeling in meeting recognition. In *INTERSPEECH'11*, pages 2877–2880.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems'12*, pages 1097–1105.

[26] A. Lam, A. Nguyen, H. Nguyen, and T. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. In *ASE'15*, pages 476–481.

[27] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *FSE'11*, pages 311–321.

[28] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *FSE'05*, pages 306–315.

[29] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *ICSM'09*, pages 233–242.

[30] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[31] T. J. McCabe. A complexity measure. *TSE'76*, (4):308–320.

[32] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *FSE'08*, pages 13–23.

[33] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *TSE'07*, 33(1):2–13.

[34] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *ASE'10*, 17(4):375–407.

[35] A. Mnih and G. E. Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems'09*, pages 1081–1088.

[36] A. Mohamed, D. Yu, and L. Deng. Investigation of full-sequence training of deep belief networks for speech recognition. In *INTERSPEECH'10*, pages 2846–2849.

[37] A.-r. Mohamed, G. E. Dahl, and G. Hinton. Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):14–22, 2012.

[38] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*, pages 181–190.

[39] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718*, 2014.

[40] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM'07*, pages 364–373.

[41] J. Nam and S. Kim. Heterogeneous defect prediction. In *FSE'15*, pages 508–519.

[42] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *ICSE'13*, pages 382–391.

[43] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 2001.

[44] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *ICSE'15*, pages 858–868.

[45] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *ASE'12*, pages 70–79.

[46] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *FSE'09*, pages 383–392.

[47] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong. Topic-based defect prediction. In *ICSE'11*, pages 932–935.

[48] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *PROMISE'10*, pages 19:1–19:10.

[49] S. J. Pan, I. Tsang, J. Kwok, and Q. Yang. Domain adaptation via transfer component analysis. *Neural Networks, IEEE Transactions on*, pages 199–210, 2011.

[50] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *ICASSP'15*, pages 1916–1920.

[51] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *FSE'08*, pages 2–12.

[52] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *ICSE'13*, pages 432–441.

[53] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI'14*, pages 419–428.

[54] R. Salakhutdinov and G. Hinton. Semantic hashing. *RBM'07*, 500(3):500.

[55] R. Sarikaya, G. E. Hinton, and A. Deoras. Application of deep belief networks for natural language understanding. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 22(4):778–784, 2014.

[56] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *FSE'11*, pages 365–375.

[57] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *ICSE'15*, pages 99–108.

[58] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. ichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *ICSE'15*, pages 812–823.

[59] W. Tao and L. Wei-hua. Naive bayes software defect prediction model. In *CiSE'10*, pages 1–4.

[60] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *FSE'14*, pages 269–280.

[61] 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 Softw. Engg.*, 14(5):540–578, 2009.

[62] J. Wang, B. Shen, and Y. Chen. Compressed c4. 5 models for software defect prediction. In *QSIC'12*, pages 13–16.

[63] S. Watanabe, H. Kaiya, and K. Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, pages 19–24, 2008.

[64] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *PROMISE'07*, pages 8–8.

[65] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *MSR'15*, pages 334–345.

[66] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[67] X. Xie, W. Zhang, Y. Yang, and Q. Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *PROMISE'12*, pages 19–28.

[68] X. Yang, D. Lo, X. xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *QRS'15*, pages 17–26.

[69] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: Deep learning in android malware detection. In *SIGCOMM'14*, pages 371–372.

[70] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE'07*, pages 9–9.