This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

Deep Semantic Feature Learning for Software Defect Prediction

Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan

Abstract—Software defect prediction, which predicts defective code regions, can assist developers in finding bugs and prioritizing their testing efforts. Traditional defect prediction features often fail to capture the semantic differences between different programs. This degrades the performance of the prediction models built on these traditional features. Thus, the capability to capture the semantics in programs is required to build accurate prediction models. To bridge the gap between semantics and defect prediction features, we propose leveraging a powerful representation-learning algorithm, deep learning, to learn the semantic representations of programs automatically from source code files and code changes. Specifically, we leverage a deep belief network (DBN) to automatically learn semantic features using token vectors extracted from the programs' abstract syntax trees (AST) (for file-level defect prediction models) and source code changes (for change-level defect prediction models).

We examine the effectiveness of our approach on two file-level defect prediction tasks (i.e., file-level within-project defect prediction and file-level cross-project defect prediction) and two change-level defect prediction tasks (i.e., change-level within-project defect prediction and change-level cross-project defect prediction). Our experimental results indicate that the DBN-based semantic features can significantly improve the examined defect prediction tasks. Specifically, the improvements of semantic features against existing traditional features (in F1) range from 2.1 to 41.9 percentage points for file-level within-project defect prediction, from 1.5 to 13.4 percentage points for file-level cross-project defect prediction, from 1.0 to 8.6 percentage points for change-level within-project defect prediction, and from 0.6 to 9.9 percentage points for change-level cross-project defect prediction.

Index Terms—Defect prediction, quality assurance, deep learning, semantic features.

1 Introduction

Software defect prediction techniques [22], [29], [31], [41], [45], [53], [59], [62], [80], [96], [112] have been proposed to detect defects and reduce software development costs. Defect prediction techniques build models using software history data and use the developed models to predict whether new instances of code regions, e.g., files, changes, and methods, contain defects.

The efforts of previous studies toward building accurate prediction models can be categorized into the two following approaches: The first approach is manually designing new features or new combinations of features to represent defects more effectively, and the second approach involves the application of new and improved machine learning based classifiers. Researchers have manually designed many features to distinguish defective files from non-defective files, e.g., Halstead features [19] based on operator and operand counts; McCabe features [50] based on dependencies; CK features [8] based on function and inheritance counts, etc.; MOOD features [21] based on polymorphism factor, coupling factor, etc.; process features [29], [80] (including number of lines of code added, removed, meta features, etc.); and object-oriented features [3], [11], [49].

- *S. Wang, T. Liu, and L. Tan are with the Department of Electrical and Computer Engineering, University of Waterloo, Canada. E-mail: {song.wang, t67liu, jc.nam, lintan}@uwaterloo.ca*
- *J. Nam is with the School of Computer Science and Electrical Engineering, Handong Global University, Pohang, Korea. E-mail: jc.nam@handong.edu*

Manuscript received xxx xx, 2017; revised xxx xx, 2017.

Traditional features mainly focus on the statistical characteristics of programs and assume that buggy and clean programs have distinguishable statistical characteristics. However, our observations on real-world programs show that existing traditional features often cannot distinguish programs with different semantics. Specifically, program files with different semantics can have traditional features with similar or even the same values. For example, Figure 1 shows an original buggy version, i.e., Figure 1(a), and a fixed clean version, i.e., Figure 1(b), of a method from Lucene. In the buggy version, there is an IOException when initializing variables `os` and `is` before the try block. The buggy version can lead to a memory leak[1] and has already been fixed by moving the initializing statements into the try block in Figure 1(b). Using traditional features to represent these two code snippets, e.g., code complexity features, their feature vectors are identical. This is because these two code snippets have the same source code characteristics in terms of complexity, function calls, raw programming tokens, etc. However, the semantic information in these two code snippets is significantly different. Specifically, the contextual information of the two variables, i.e., `os` and `is`, in the two versions is different. Features that can distinguish such semantic differences are needed for building more accurate prediction models.

To bridge the gap between the programs' semantic information and defect prediction features, we propose leveraging a powerful representation-learning algorithm, namely,

1. https://issues.apache.org/jira/browse/LUCENE-3251

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering


```
1  public void copy(Directory to, String src, String dest)
       throws IOException {
2    IndexOutput os = to.createOutput(dest);
3    IndexInput is = openInput(src);
4    IOException priorException = null;
5
6    try {
7      is.copyBytes(os, is.length());
8    } catch (IOException ioe) {
9      priorException = ioe;
10   }
11   finally {
12     IOUtils.closeSafely(priorException, os, is);
13   }
14 }
```

(a) Original buggy code snippet.

```
1  public void copy(Directory to, String src, String dest)
       throws IOException {
2    IndexOutput os = null;
3    IndexInput is = null;
4    IOException priorException = null;
5    try {
6      os = to.createOutput(dest);
7      is = openInput(src);
8      is.copyBytes(os, is.length());
9    } catch (IOException ioe) {
10     priorException = ioe;
11   } finally {
12     IOUtils.closeSafely(priorException, os, is);
13   }
14 }
```

(b) Code snippet after fixing the bug.

Fig. 1: A motivating example from Lucene.

deep learning [27], to *learn semantic representations of programs automatically*. Specifically, we use the deep belief network (DBN) [26] to automatically learn features from token vectors extracted from source code, and then we utilize these features to build and train defect prediction models.

DBN is a generative graphical model, which learns a semantic representation of the input data that can reconstruct the input data with a high probability as the output. It automatically learns high-level representations of data by constructing a deep architecture [4]. There have been successful applications of DBN in many fields, including speech recognition [58], image classification [9], [43], natural language understanding [56], [86], and semantic search [85].

To use a DBN to learn features from code snippets, we first convert the code snippets into vectors of tokens with the structural and contextual information preserved, and then we use these vectors as the input into the DBN. For the two code snippets presented in Figure 1, the input vectors are `[..., IndexOutput, createOutput(), IndexInput, openInput(), IOException, try, ...]` and `[..., IndexOutput, IndexInput, IOException, try, createOutput(), openInput()...]` respectively (Details regarding the token extraction are provided in Section 3.1). As the vectors of these two code snippets are different, the DBN will *automatically* learn features that can distinguish them.

We examine our DBN-based approach to generating semantic features on both file-level defect prediction tasks (i.e., predict which files in a release are buggy) and change-level defect prediction tasks (i.e., predict whether a code commit is buggy), because most of the existing approaches to defect prediction are on these two levels [2], [24], [29], [67], [82], [90], [94], [103], [104]. Focusing on these two different defect prediction tasks enables us to extensively compare our proposed technique with state-of-the-art defect prediction features and techniques. For file-level defect prediction, we generate DBN-based semantic features by using the complete Abstract Syntax Trees (AST) of the source files, while for change-level defect prediction, we generate the DBN-based features by using tokens extracted from code changes, as detailed in Section 3.

In addition, most defect prediction studies have been conducted in one or two settings, i.e., within-project defect prediction [29], [57], [90], [104] and/or cross-project defect prediction [24], [67], [94], [103]. Thus, we evaluate our approach in these two settings as well.

In this work, we explore the performance of the DBN-based semantic features using different measures under different evaluation scenarios. We first evaluate the prediction performance by using $Precision$, $Recall$, and $F1$, as they are commonly used evaluation measures in defect prediction studies [2], [67], [82], [97], which we refer to as the non-effort-aware scenario in this work. In addition, we also conduct an effort-aware evaluation [52] to show the practical aspect of defect prediction by using `PofB20`, i.e., the percentage of bugs that can be discovered by inspecting 20% lines of code (LOC) [29]. For example, when a team can afford to inspect only 20% LOC before a deadline, it is crucial to inspect the 20% that can assist the developers in discovering the highest number of bugs.

This paper makes the following contributions:

- Shows the incapability of traditional features in capturing the semantic information of programs.
- Proposes a new technique to leverage a powerful representation-learning algorithm, deep learning, to learn *semantic features* from token vectors extracted from programs' ASTs (for file-level defect prediction models) and source code changes (for change-level defect prediction models) automatically.
- Conducts rigorous and large-scale experiments to evaluate the performance of the DBN-based semantic features for defect prediction tasks under both the non-effort-aware and effort-aware scenarios; and
- Demonstrates that DBN-based semantic features can significantly improve defect prediction. Specifically, the improvements of semantic features against existing traditional features (in F1) range from 2.1 to 41.9 percentage points for file-level within-project defect prediction, from 1.5 to 13.4 percentage points for file-level cross-project defect prediction, from 1.0 to 8.6 percentage points for change-level within-project defect prediction, and from 0.6 to 9.9 percentage points for change-level cross-project defect prediction.

The rest of this paper is summarized as follows. Section 2 provides the backgrounds on defect prediction and DBN. Section 3 describes our approach to learning semantic features followed by leveraging the learned features to predict defects. Section 4 presents the experimental setup. Section 5 evaluates the performance of the learned semantic features.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

Section 6 discusses our results and threats to the validity. Section 7 surveys the related work. Section 8 summarizes our study. This paper extends our prior publication [97] presented at the 38th International Conference on Software Engineering (ICSE'16). New materials with respect to the conference version include:

- Examining the effectiveness of the proposed approach for generating semantic features on two change-level defect prediction tasks, i.e., change-level within-project defect prediction (WCDP) and change-level cross-project defect prediction (CCDP). The details are presented in Section 4.5.2 (experimental design), Section 4.8 (WCDP), Section 4.9 (CCDP), Section 5.3 (results of WCDP), and Section 5.4 (results of CCDP).
- New techniques to process incomplete code from source code changes for generating DBN-based features automatically are proposed. The details are presented in Section 3.1.2.
- The model performance assessment scenarios are updated. Both the non-effort-aware and effort-aware evaluation processes are employed to comprehensively evaluate the performance of the DBN-based semantic features on file-level and change-level defect prediction tasks. The details are presented in Section 5.
- New experiments on open-source commercial projects and additional details regarding the experimental design and results are provided. In addition, statistical testing and Cliff's delta effect size analysis are conducted to measure and demonstrate the significance of the prediction performance of the DBN-based semantic features. The details are provided in Section 5.

2 Background

TABLE 1: Defect prediction tasks investigated in this work.

	Within-project	Cross-project
File Level	WPDP	CPDP
Change Level	WCDP	CCDP

This section provides the backgrounds of file-level and change-level defect prediction and deep belief network. Table 1 shows the investigated prediction tasks and their corresponding abbreviations.

2.1 File-level Defect Prediction

Figure 2 presents a typical file-level defect prediction process that is adopted by existing studies [31], [45], [55], [66], [67], [76], [100]. The first step is to label the data as buggy or clean based on post-release defects for each file. One could collect these post-release defects from a Bug Tracking System (BTS) via linking bug reports to its *bug-fixing* changes. Files related to these bug-fixing changes are considered as *buggy*. Otherwise, the files are labeled as *clean*. The second step is to collect the 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 a *training set*, whereas the set of instances used to evaluate the trained models is referred to as a *test set*. As shown in

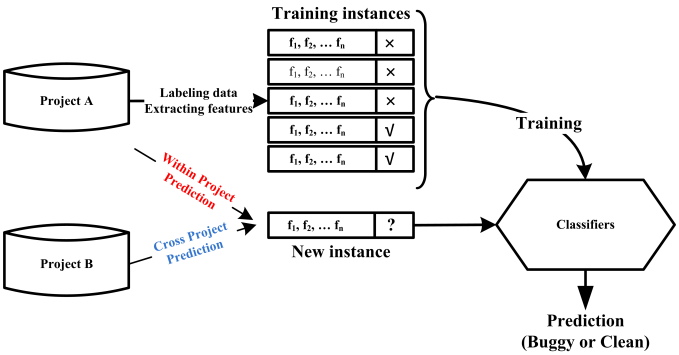


Fig. 2: Defect Prediction Process

Figure 2, when performing within-project defect prediction (following existing work [66], we call this WPDP), the training and test sets are from the same project, i.e., project A. When performing cross-project defect prediction (following existing work [66], we call this CPDP), the prediction models are trained by a training set from project A (source), and a test set is from a different project, i.e., project B (target).

In this study, for file-level defect prediction, we examine the performance of the learned DBN-based semantic features on both WPDP and CPDP.

2.2 Change-level Defect Prediction

Change-level defect prediction can predict whether a change is buggy at the time of the commit so that it allows developers to act on the prediction results as soon as a commit is made. In addition, since a change is typically smaller than a file, developers have much less code to examine in order to identify defects. However, for the same reason, it is more difficult to predict buggy changes accurately.

Similar to file-level defect prediction, change-level defect prediction also consists of the following processes:

- Labeling process: Labeling each change as buggy or clean to indicate whether the change contains bugs.
- Feature extracting process: Extracting the features to represent the changes.
- Model building and testing process: Building a prediction model with the features and labels and then using the model to predict testing data.

Different from labeling file-level defect data, labeling change-level defect data requires further linking of bug-fixing changes to bug-introducing changes. A line that is deleted or changed by a bug-fixing change is a faulty line, and the most recent change that introduced the faulty line is considered a bug-introducing change. We could identify the bug-introducing changes by a *blame* technique provided by a Version Control System (VCS), e.g., git or SZZ algorithm [40]. Such blame techniques are widely used in existing studies [29], [40], [57], [90], [107]. In this work, the bug-introducing changes are considered as buggy, and other changes are labeled clean. Note that, not all projects have a well maintained BTS, and we consider changes whose commit messages contain the keyword "fix" as bug-fixing changes by following existing studies [29], [90].

In this work, similar to the file-level defect prediction, we also examine the performance of DBN-based features on

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

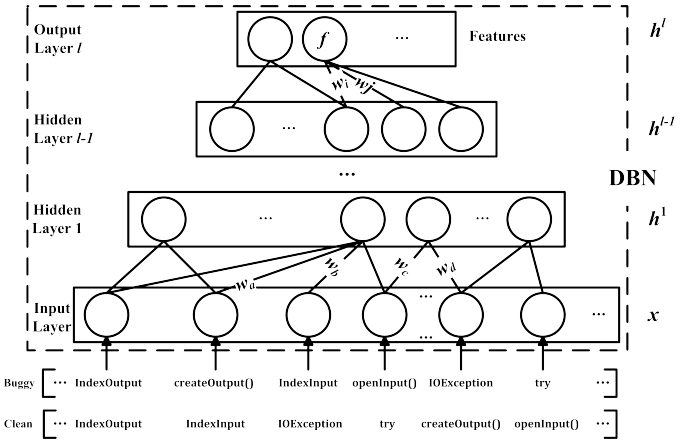


Fig. 3: Deep belief network architecture and input instances of the buggy version and the clean version presented in Figure 1. Although the token sets of these two code snippets are identical, the different structural and contextual information between tokens enables DBN to generate different features to distinguish them.

both change-level within-project defect prediction (WCDP) and change-level cross-project defect prediction (CCDP).

2.3 Deep Belief Network

A deep belief network is a generative graphical model that uses a multi-level neural network to learn a representation from the training data that could reconstruct the semantic and content of the training data with a high probability [4]. DBN contains one *input layer* and several *hidden layers*, and the top layer is the output layer that contains final 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 $k+1$ layers.

To calculate $P(h^k|h^{k+1})$, each pair of two adjacent layers in DBN are trained as a Restricted Boltzmann Machines (RBM) [4]. An 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:

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

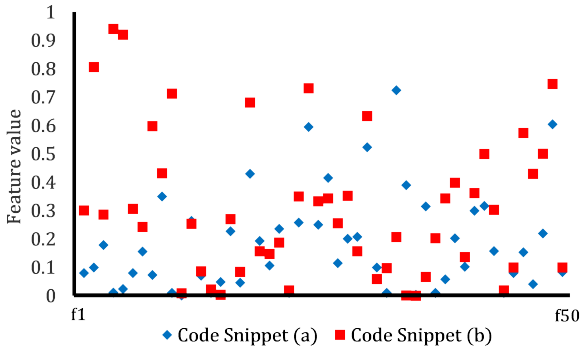


Fig. 4: The distribution of DBN-based features of the two code snippets shown in Figure 1.

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

where n_k is the number of nodes 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 layer $k+1$. $sigm$ is the sigmod function, which serves as the activation function to update the hidden units. We use the sigmod function because it outputs a more smooth range of nonlinear values with a relatively simple computation [20].

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 an 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 **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 4.5.

The DBN model generates features with more complex network connections. These network connections enable DBN models to generate features with multiple levels of abstraction and high-level semantics. DBN features are weighted combinations/vectors of input nodes, which may represent patterns of the usages of input nodes (e.g., methods, control-flow nodes, etc.). We believe such DBN-based features can help distinguish the semantics of different source code snippets, which traditional features cannot handle well. For example, Figure 4 shows the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

distribution of the DBN-based semantic features of the two code snippets shown in Figure 1. Specifically, we use the trained DBN model on project Lucene (details are in Section 4.8) to generate a feature set that contains 50 different features for each of the two code snippets. As we can see in the figure, the distributions of features of the two code snippets are different. Specifically, most of the features of code snippet shown in Figure 1(b) have larger values than those of the features of code snippet shown in Figure 1(a). Thus, the new features are capable of distinguishing these two code snippets with a proper classifier.

3 Approach

In this work, we use DBN to generate semantic features automatically from source files and code changes and further leverage these features to improve defect prediction. Figure 5 illustrates the workflow of our approach to generating features for both file-level defect prediction (inputs are source files) and change-level defect prediction (inputs are source code changes). Specifically, for file-level defect prediction, our approach takes AST node tokens from the source code of the training and test source files as the input, and generates semantic features from them. Then, the generated semantic features are used to build the models for predicting defects. Note that for change-level defect prediction, the input data to our DBN-based feature generation approach are changed code snippets. Since building AST for an incomplete code snippet is challenging, in this work we propose a heuristic approach to extracting important structural and context information from code change snippets (details are in Section 3.1.2). The DBN requires input data in the form of integer vectors, to satisfy this requirement, we first build a mapping between integers and tokens and then convert the token vectors to integer vectors, to generate semantic features, we first use the integer vectors of the training set to build and train a DBN. Then, we use the trained 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.

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

3.1 Parsing Source Code

3.1.1 Parsing Source Code for Files

For file-level defect prediction tasks, we utilize the Java Abstract Syntax Tree (AST) to extract syntactic information from source code files. Specifically, three types of AST node are extracted: 1) nodes of method invocations and class instance creations, e.g., in Figure 3, method `createOutput()` and `openInput()` 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 `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 the names `getAttribute(XXX)` or `setAttribute(XXX)` is not helpful. However, it is useful to know that method declaration nodes exist, 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.

3.1.2 Parsing Source Code for Changes

Different from file-level defect prediction data, i.e., program source files, for which we could build ASTs and extract AST token vectors for feature generation, change-level defect prediction data are changes that developers made to source files, whose syntax information is often incomplete. These changes could have different locations and include code additions and code deletions, which are syntactic incomplete. Thus, building ASTs for these changes is challenging. In this study, for tokenizing changes, instead of building ASTs, we tokenize a change by considering the `code addition`, the `code deletion`, and the `context code` in the change. Code additions are the added lines in a change, code deletions are the deleted lines in a change, and the code around these additions or deletions is considered the context code. For example, Figure 6 shows a real change example from project Lucene. In this change, the code addition contains lines 15 and 16, the code deletion contains lines 7 to 14, and the context contains lines 4 to 6, 17, and 18. Note that the contents of the source code lines in the additions, deletions, and context code are often overlapping, e.g., the deleted line 7 and the added line 15

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

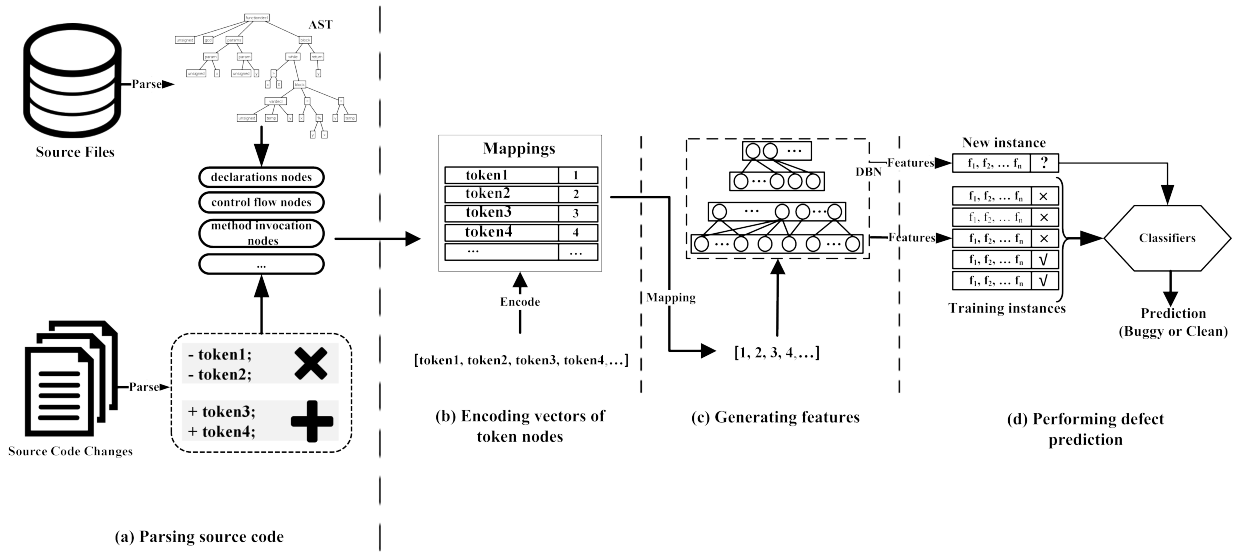


Fig. 5: Overview of our DBN-based approach to generating semantic features for file-level and change-level defect prediction.

```
1  --- a/solr/src/java/org/apache/solr/handler/component/QueryComponent.java
2  +++ b/solr/src/java/org/apache/solr/handler/component/QueryComponent.java
3  @@ -217,14 +217,8 @@ public class QueryComponent extends SearchComponent
4     for (String groupByStr : funcs) {
5         QParser parser = QParser.getParser(groupByStr, "func", rb.req);
6         Query q = parser.getQuery();
7  -      SolrIndexSearcher.GroupCommandFunc gc;
8  -      if (groupSort != null) {
9  -         SolrIndexSearcher.GroupSortCommand gcSort = new SolrIndexSearcher.GroupSortCommand();
10 -         gcSort.sort = groupSort;
11 -         gc = gcSort;
12 -       } else {
13 -         gc =  new SolrIndexSearcher.GroupCommandFunc();
14 -       }
15 +       SolrIndexSearcher.GroupCommandFunc gc = new SolrIndexSearcher.GroupCommandFunc();
16 +       gc.groupSort = groupSort;
17         if (q instanceof FunctionQuery) {
18         gc.groupBy = ((FunctionQuery)q).getValueSource();
```

Fig. 6: A change example from Lucene (commit id is 9535bb795f6d1ec4c475a5d35532f3c7951101da).

contain the same line of code for class instance creation, i.e., `SolrIndexSearcher.GroupCommandFunc gc;`. Thus, to distinguish these lines, we add different prefixes to the raw tokens that are extracted from different types of changed code. Specifically, for the addition, we use prefix "added_", for the deletion, we use prefix "deleted_", and for the context code, we use prefix "context_". The details of the three types of tokens extracted from the example change (in Figure 6) are shown in Table 2.

From Table 2, we could observe that different types of tokens from the changed code snippets contain different information. For example, the context nodes show that the code is changed inside a `for` loop, an `if` statement is removed from the source code in the deletions, and an instantiation of class `GroupCommandFunc` was created in the additions.

Intuitively, DBN-based features generated from different types of tokens may have different impacts on the performance of the change-level defect prediction. To extensively explore the performance of different types of tokens, we build and evaluate change-level defect prediction models with seven different combinations among the three different types of tokens, i.e., **added**: only considers the additions; **deleted**: only considers the deletions; **context**: only considers the context information; **added+deleted**: considers both the additions and the deletions; **added+context**: considers both the additions and the context tokens; **deleted+context**: considers both the deletions and the context tokens; and **added+deleted+context**: considers the additions, deletions, and context tokens together. We discuss the effectiveness of these different combinations in Section 5.

Note that some of the tokens extracted from the changed code snippets are project-specific, which means that they are rare or never appear in changes from a different project. Thus, for change-level cross-project defect prediction we first filter out variable names, and then use `method declaration`, `method invocation`, and `class instantiation` to represent a method declaration, a method call, and an instance of a class instantiation respectively.

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 [25], [39], [92]. To prune noisy data, Kim et al. proposed an effective mislabeling data detection approach named *Closest List Noise Identification* (CLNI) [39]. It identifies the k-nearest

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 2: Three types of tokens extracted from the example change shown in Figure 6.

added	added_SolrIndexSearcher.GroupCommandFunc added_gc added_SolrIndexSearcher.GroupCommandFunc added_gc.groupSort added_groupSort
deleted	deleted_SolrIndexSearcher.GroupCommandFunc deleted_gc deleted_if deleted_groupSort deleted_Notnull deleted_SolrIndexSearcher.GroupSortCommand deleted_gcSort deleted_SolrIndexSearcher.GroupSortCommand deleted_gcSort.sort deleted_groupSort deleted_gc deleted_gcSort deleted_delete else deleted_gc deleted_SolrIndexSearcher.GroupCommandFunc
context	context_for context_QParser context_parser context_QParser.getParser context_Query context_q context_parser.getQuery context_if context_q context_FunctionQuery context_gc.groupBy context_FunctionQuery context_q.getValueSource

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 an 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 to help the DBN learn the common knowledge between the semantic information of buggy and clean instances, we adopt the edit distance similarity computation algorithm [68] to define the distances between instances. The edit distances are sensitive to both the tokens and the 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 to work well. In our benchmark experiments with traditional features, we also perform CLNI to remove the incorrectly labeled data.

In addition, we also filter out infrequent tokens extracted from the source code, which might be designed for a specific file and cannot be generalized to other files. Given a project, if the total number of occurrences of a token is less than three, we filter it out. We encode only the tokens that occur three or more times, which is a common practice in the NLP research field [48]. The same filtering process is also applied to change-level prediction tasks.

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 the DBN to generate semantic features, we first build a mapping between integers and tokens, and encode token vectors to integer vectors. Each token has a unique integer identifier. 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 the DBN. Taking the code snippets in Figure 3 as an example, if we only consider the two versions, the token vectors for the "Buggy" and "Clean" versions would be mapped to [1, 2, 3, 4, 5, 6, ...] and [1, 3, 5, 6, 2, 4, ...] respectively. Through this encoding process, the 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. Note that, in this work we employ the same token mapping mechanism for both the file-level and change-level defect prediction tasks.

3.3 Training the DBN and Generating Features

3.3.1 Training the DBN

As we 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 studies that leveraged DBN models to generate features for NLP [86], [87] and image recognition [9], [43] reported that the performance of DBN-based features is sensitive to these parameters. A few hidden layers can be trained in a relatively short period of time, but result in poor performance as the system cannot fully capture the characteristics of the training datasets. Too many layers may result in overfitting and a slow learning time. Similar to the number of hidden layers, too few or too many hidden nodes or iterations result in either slow learning or poor performance [87]. We show how we tune these parameters in Section 4.5.

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

The DBN requires the values of the input data to range from 0 to 1, while the 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

normalization [102]. In our mapping process, the 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 that these two nodes are different and independent. Thus, the normalized values can still be used as token identifiers since the same identifiers pertain 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, and then obtain semantic features for the training and the 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 instance from both the training and the test datasets, we then build defect prediction models by following the standard defect prediction process described in Section 2. The test data are used to evaluate the performance of the built defect prediction models.

Note that, as revealed in existing work [90], [91], the widely used validation technique, i.e., k-fold cross-validation often introduces nontrivial bias for evaluating defect prediction models, which makes the evaluation inaccurate. In addition, for change-level defect prediction, the k-fold cross-validation may make the evaluation incorrect. This is because the changes follow a certain order in time. Randomly partitioning the dataset into k folds may cause a model to use future knowledge which should not be known at the time of prediction to predict changes in the past. Thus, cross-validation may use information regarding a change committed in 2017 to predict whether a change committed in 2015 is buggy or clean. This scenario would not be a real case in practice, because at the time of prediction, which is typically soon after the change is committed in 2015 for the earlier detection of bugs, the change committed in 2017 is not yet existent.

To avoid the above validation problem, we do not use the k-fold cross-validation in this work. Specifically, for file-level defect prediction, we evaluate the performance of our DBN-based features and traditional features by building prediction models with data from different releases. For change-level defect prediction, we collect the training and test datasets following the time order (Details are in Section 4.3.2) to build and evaluate the prediction models without k-fold cross-validation.

4 Experimental Setup

In this section, we describe the detailed settings for our evaluation experiments. All experiments are run on a 2.5GHz i5-3210M machine with 4GB RAM.

4.1 Research Questions

Table 3 lists the scenarios for the investigated research questions. Specifically, we evaluate the performance of our DBN-based semantic features by comparing it with traditional defect prediction features under each of the four different prediction scenarios. These questions share the following format.

TABLE 3: Research questions investigated in this work.

		Scope	
		Within-project	Cross-project
Level	File	**RQ1**	**RQ2**
	Change	**RQ3**	**RQ4**

RQi ($1 \leq i \leq 4$): Do DBN-based semantic features outperform traditional features at the <**level**> <**scope**> under the non-effort-aware and effort-aware evaluation scenarios?

For example, in **RQ1**, we explore the effectiveness of the DBN-based semantic features for within-project defect prediction at the file-level under both the non-effort-aware and effort-aware evaluation scenarios.

4.2 Evaluation Metrics

4.2.1 Metrics for Non-effort-aware Evaluation

Under the non-effort-aware scenario, we use three metrics: $Precision$, $Recall$, and $F1$. These metrics have been widely adopted to evaluate defect prediction techniques [31], [54], [55], [67], [90], [112]. 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 (or changes) that are truly defective, while false positive is the number of predicted defective ones that are actually not defective. A false negative records the number of predicted non-defective files (or changes) that are actually defective. Higher precision is demanded by developers who do not want to waste their debugging efforts on the non-defective code, while higher recall is often required for mission-critical systems, e.g., revealing additional defects [112]. However, comparing defect prediction models by using only these two metrics may be incomplete. For example, one could simply predict all instances as buggy instances to achieve a recall score of 1.0 (which will likely result in a low precision score) or only classify the instances with higher confidence values as buggy instances to achieve a higher precision score (which could result in a low recall score). To overcome the above issues, we also use the F1 score (i.e., F1), which is the harmonic mean of precision and recall, to measure the performance of the defect prediction.

4.2.2 Metrics for Effort-aware Evaluation

For effort-aware evaluation, we employ `PofB20` [29] to measure the percentage of bugs that a developer can identify by inspecting the top 20 percent lines of code.

To calculate `PofB20`, we first sort all the instances in the test dataset based on the confidence levels (i.e., probabilities

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 4: Cliff's Delta and the effectiveness level [10].

Cliff's Delta (δ)	Effectiveness Level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$\|\delta\| \geq 0.474$	Large

of being predicted as buggy) that a defect prediction model generates for each instance. This is because an instance with a higher confidence level is more likely to be buggy. We then simulate a developer that inspects these potentially buggy instances. We accumulate the lines of code (LOC) that are inspected and the number of bugs identified. The process will be terminated when 20 percent of the LOC in the test data have been inspected and the percentage of bugs that are identified is referred to as the `PofB20` score. A higher `PofB20` score indicates that a developer can detect more bugs when inspecting a limited number of LOC.

4.2.3 Statistical Tests

Statistical tests can help understand whether there is a statistically significant difference between two results. In this work, we used the Wilcoxon signed-rank test to check whether the performance difference between prediction models with DBN-based semantic features and prediction models with traditional features is significant. For example, in RQ3, we want to compare the performance of DBN-based features and traditional features for change-level within-project defect prediction for the projects listed in Table 6. To conduct the Wilcoxon signed-rank test, we first run experiments with these two sets of features and obtain prediction results for each test subject. We then apply the Wilcoxon signed-rank test on the results of the test subjects. The Wilcoxon signed-rank test does not require the underlying data to follow any distribution. In addition, it can be applied to pairs of data and is able to compare the difference against zero. At the 95% confidence level, p-values that are less than 0.05 indicate that the difference between subjects is statistically significant, while p-values that are 0.05 or larger indicate that the difference is not statistically significant.

4.2.4 Cliff's Delta Effect Size Analysis

To further examine the effectiveness of our DBN-based features, following the existing work in [64], [103], we employ Cliff's $delta$ (δ) [10] to measure the effect size of our approach. Cliff's $delta$ is a non-parametric effect size measure that quantifies the amount of difference between two approaches. In this work, we use Cliff's $delta$ to compare the defect prediction models that are built with our DBN-based features to the defect prediction models that are built with traditional features. Cliff's $delta$ is computed using the formula $delta = (2W/mn) - 1$, where W is the W statistic of the Wilcoxon rank-sum test, and m and n are the sizes of the result distributions of two compared approaches. The $delta$ values range from -1 to 1, where $\delta = -1$ or 1 indicates the absence of an overlap between the performances of the two compared models (i.e., all F1 values from one prediction model are higher than the F1 values of the other prediction model, and vice versa), while $\delta = 0$ indicates that the two prediction models completely overlap. Table 4 describes the meanings of the different Cliff's delta values [10].

4.3 Evaluated Projects and Data Sets

In this work, we use different datasets for evaluating file-level and change-level defect prediction tasks. Specifically, for evaluating the performance of DBN-based features on file-level defect prediction, we use publicly available data from the PROMISE data repository, which are widely used for evaluating file-level defect prediction models [24], [31], [66], [67], [103]. For change-level defect prediction, we adopt the dataset from previous studies [29], [90], [104].

The main reason for adopting different datasets for file-level and change-level defect prediction tasks is that using existing widely used datasets enables us to directly compare our approach with existing defect prediction models on the same datasets, which makes the comparison more reliable.

4.3.1 Evaluated Projects for File-level Defect Prediction

To facilitate the replication and verification of our experiments, we use publicly available data from the PROMISE data repository. Specifically, we select all the Java projects from PROMISE[2] whose version numbers are provided. We need the version numbers of each project because we need its source code archive to extract token vectors from the ASTs of the source code to feed our DBN-based feature generation approach. In total, 10 Java projects are collected. Table 5 lists the versions, the average number of source files (excluding test files), and the average buggy rate of each project. The average number of files of the projects ranges from 122 to 815, and the buggy rates of the projects have a minimum value of 9.4% and a maximum value of 62.9%.

4.3.2 Evaluated Projects for Change-level Defect Prediction

We choose six open-source projects: Linux kernel, PostgreSQL, Xorg, Jdt (from Eclipse), Lucene, and Jackrabbit. They are large and typical open source projects covering operating systems, database management systems. These projects have sufficient change histories to build and evaluate change-level defect prediction models and are commonly used in the literature [29], [90], [104]. For Lucene and Jackrabbit, we use manually verified bug reports from Herzig et al. [25] to label the bug-fixing changes, and the keyword search approach [88] is used for the others.

Table 6 shows the evaluated projects for change-level defect prediction. The LOC and the number of changes in Table 6 include only source code (C and Java) files[3] and their changes because we want to focus on classifying source code changes only. Although these projects are written in C and Java, our DBN-based feature generation approach is not limited to any particular programming language. With the appropriate feature extraction approach, our DBN-based feature generation approach can easily be extended to projects in other languages.

Change-level defect data are often imbalanced [23], [29], [34], [35], i.e., there are fewer buggy instances than clean

2. http://openscience.us/repo/defect

3. We include files with these extensions: .java, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx and .h++.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 5: Evaluated projects for file-level defect prediction.

Project	Description	Releases	Avg # Source Files	Avg Buggy Rate (%)
ant	Java based build tool	1.5,1.6,1.7	463.7	21.0
camel	Enterprise integration framework	1.2,1.4,1.6	815	22.5
jEdit	Text editor designed for programmers	3.2,4.0,4.1	297	27.4
log4j	Logging library for Java	1.0,1.1	122	29.1
lucene	Text search engine library	2.0,2.2,2.4	260.7	56.0
xalan	A library for transforming XML files	2.4,2.5	763	32.6
xerces	XML parser	1.2,1.3	446.5	15.7
ivy	Dependency management library	1.4,2.0	296.5	9.4
synapse	Data transport adapters	1.0,1.1,1.2	211.7	25.5
poi	Java library to access Microsoft format files	1.5,2.5,3.0	354.7	62.9

TABLE 6: Evaluated projects for change-level defect prediction in this work. **Lang** is the programming language used for the project. **LOC** is the number of the line of code. **First Date** is the date of the first commit of a project, while **Last Date** is the date of the latest commit. **Changes** is the number of changes collected in this work. **TrSize** is the average size of training data on all runs. **TSize** is the average size of test data on all runs. **NR** is the number of runs for each subject.

Project	Lang	LOC	First Date	Last Date	Changes	TrSize	TSize	Average Buggy Rate (%)	# NR
Linux	C	7.3M	2005-04-16	2010-11-21	429K	1,608	6,864	22.8	4
PostgreSQL	C	289K	1996-07-09	2011-01-25	89K	1,232	6,824	27.4	7
Xorg	C	1.1M	1999-11-19	2012-06-28	46K	1,756	6,710	14.7	6
JDT	Java	1.5M	2001-06-05	2012-07-24	73K	1,367	6,974	20.5	6
Lucene	Java	828K	2010-03-17	2013-01-16	76K	1,194	9,333	23.6	8
Jackrabbit	Java	589K	2004-09-13	2013-01-14	61K	1,118	8,887	37.4	10

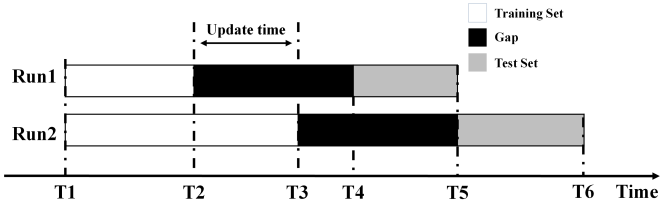


Fig. 7: Change-level data collection process [90].

instances in the training dataset. For example, as shown in Table 6, the average ratio of the buggy and the clean changes is 1.0 to 3.1. The imbalanced data can lead to poor prediction performance [90]. For change-level data, we borrow the data collection process introduced by Tan et al. [90]. Specifically, a gap between the training set and the test set (see Figure 7) is used because the gap allows more time for buggy changes in the training set to be discovered and fixed. For example, the time period between time T2 and time T4 is a gap. In this manner, the training set will be more balanced, i.e., the training set will have a higher buggy rate. A reasonable setup is to make the sum of the gap and the test set, e.g., the duration from time T2 to T5, close to the typical bug-fixing time (i.e., the time from when a bug is introduced until it is fixed). We use the recommended gap values in [90] to collect multiple runs of experimental data, e.g., Linux has four different runs during the given time period (between the `First Date` and `Last Date`) as shown in Table 6. Note that our previous study [90] tuned and evaluated the defect prediction models based on their precision values. In this work, we do not have a bias on either precision or recall, and we tune and evaluate the prediction models based on the harmonic of the precision and recall, i.e., F1 (details are in Section 4.2.1).

Imbalanced data issues occur in both the file-level and the change-level defect data, and as shown in Table 5 and Table 6, most of the examined projects have buggy rates less than 50%. To build optimal defect prediction models, we also perform the re-sampling technique used in existing work [90], i.e., SMOTE [6], on the imbalanced projects.

4.4 Baselines of Traditional Features

4.4.1 Baselines for Evaluating File-level Defect Prediction

To evaluate the performance of semantic features for file-level defect prediction tasks, we compare the semantic features with two different traditional features. Our **first baseline** of traditional features consists of 20 traditional features. Table 7 shows the details of the 20 features and their descriptions. These features and data have been widely used in previous work to build effective defect prediction models [24], [31], [54], [55], [67], [112].

We choose the widely used PROMISE data so that we can directly compare our approach with previous studies. 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 as the input 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 handling the noise (Section 3.2.1). Each instance is represented as a vector of term frequencies of the AST nodes.

4.4.2 Baselines for Evaluating Change-level Defect Prediction

Our baseline features for change-level defect prediction include three types of change features, i.e., `bag-of-words features`, `characteristic features`, and `meta features`, which have been used in previous studies [29], [90].

- `Bag-of-words features`: The bag-of-words feature set is a vector representing the count of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 7: Metrics used for file-level defect prediction.

Metric	Description
WMC	the number of methods used in a given class [8]
DIT	the maximum distance from a given class to the root of an inheritance tree [8]
NOC	the number of children of a given class in an inheritance tree [8]
CBO	the number of classes that are coupled to a given class [8]
RFC	the number of distinct methods invoked by code in a given class [8]
LCOM	the number of method pairs in a class that do not share access to any class attributes [8]
LCOM3	another type of lcom metric proposed by Henderson-Sellers [11]
NPM	the number of public methods in a given class [3]
LOC	the number of lines of code in a given class [3]
DAM	the ratio of the number of private/protected attributes to the total number of attributes in a given class [3]
MOA	the number of attributes in a given class which are of user-defined types [3]
MFA	the number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class [3]
CAM	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods [3]
IC	the number of parent classes that a given class is coupled to [33]
CBM	the total number of new or overwritten methods that all inherited methods in a given class are coupled to [33]
AMC	the average size of methods in a given class [33]
CA	afferent coupling, which measures the number of classes that depends upon a given class [49]
CE	efferent coupling, which measures the number of classes that a given class depends upon [49]
Max_CC	the maximum McCabe's cyclomatic complexity (CC) score [50] of methods in a given class
Avg_CC	the arithmetic mean of the McCabe's clomatic complexity (CC) scores [50] of methods in a given class

occurrences of each word in the text of changes. We employ the snowBall stemmer to group words of the same root, then we use Weka [18] to obtain the bag-of-words features from both the commit messages and the source code changes.

- `Characteristic features`: Inspired by the Deckard tool [28], we use characteristic vectors as features. Characteristic vectors represent the syntactic structure by counting the numbers of each node type in the Abstract Syntax Tree (AST). Bag-of-words and characteristic vectors have different abstraction levels. Although bag-of-words can capture keywords, such as `if` and `while`, it cannot capture abstract syntactic structures, such as the number of statements. Suppose that we are using `if` and `else` node types for characteristic vectors, the characteristic vector of the code before the changes shown in Figure 6 is (1, 1). After obtaining the characteristic vectors for the file before the change and the file after the change, we subtract the two characteristic vectors to obtain the difference. For each change, we use Deckard [28] to automatically generate two characteristic vectors: one for the source code file before the change and one for the source code file after the change. We use the difference between the two characteristic vectors and the characteristic vector of the file after the change as two sets of features.

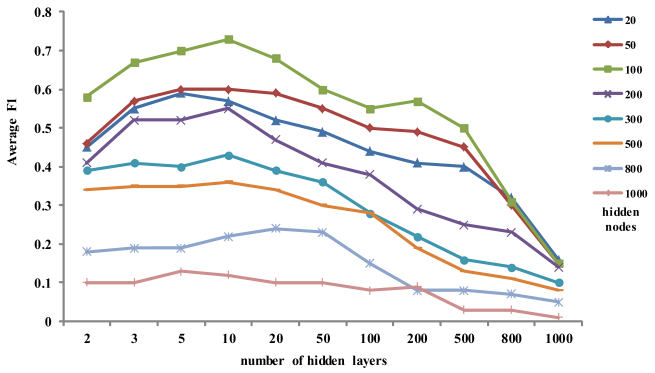


Fig. 8: File-level defect prediction performance with different parameters.

- `Meta features`: In addition to characteristic and bag-of-words vectors, we also use a set of metadata features, which includes the basic information of changes, e.g., commit time, filename, developers, etc. It also contains code change metrics, e.g., the added line count per change, the deleted line count per change, etc.

4.5 Parameter Settings for Training a DBN

Many DBN applications [9], [43], [58] report that an effective DBN requires 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 section, we study the impact of the three parameters on defect prediction models.

4.5.1 Setting Parameters for File-level Defect Prediction

For file-level defect prediction, 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). Each experiment has specific values for the three parameters and runs on the five projects individually. Given an experiment, for each project, we use the older version of the 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 of the project. After this, we use the older version to build a defect prediction model and apply it to the newer version. Finally, we evaluate the specific values of the parameters by the average F1 score of the five projects for file-level defect prediction.

Setting the number of hidden layers and the number of nodes in each layer. Because 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 that 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 i.e., 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 8 illustrates the average F1 scores obtained when tuning the number of hidden layers and the number of nodes in each hidden layer together for file-level defect

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

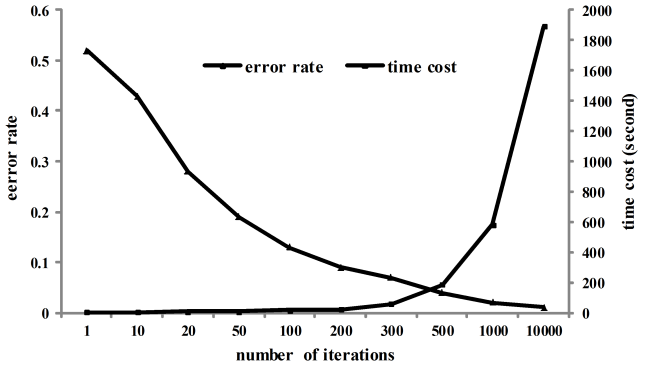


Fig. 9: Average error rates and time costs for different numbers of iterations for tuning file-level defect prediction.

prediction. When the number of nodes in each layer is fixed while increasing the 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 10. If the number of hidden layers remains unchanged, the best F1 score occurs when the number of nodes in each layer is 100 (the top line in Figure 8). 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 the DBN-based features for file-level defect prediction tasks is 100.

Setting the number of iterations. The number of iterations is another important parameter for building an effective DBN. During the training process, the DBN adjusts the weights to narrow down the *error rate* between the reconstructed input data and original input data in each iteration. In general, the higher the number of iterations, the lower the error rate. However, there is a trade-off between the number of iterations and the computational time cost. For tuning the parameters for file-level defect prediction, 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 the error rate to evaluate this parameter. Figure 9 demonstrates that, as the number of iterations increases, the error rate decreases slowly as the corresponding time cost increases exponentially. In this study, we set the number of iterations to 200, with which the average error rate is approximately 0.098 and the time cost is 15 s.

4.5.2 Setting Parameters for Change-level Defect Prediction

For change-level defect prediction, we use the same parameter tuning process as the file-level defect prediction to explore the best parameter values with all the runs of each of the six projects listed in Table 6. For each run of a project, we use its training data to train a DBN with respect to the specific values of the DBN parameters. Then, we use the trained DBN to generate semantic features for both the training and test datasets. Afterward, we use the training dataset to build a defect prediction model and apply it to the test dataset. Last, we evaluate the specific values of the parameters by using the average F1 score of the 41 runs from the six projects.

Note that, for change-level defect prediction, as we described in Section 3.1.2, we have seven different approaches available to extract the source code token vector for a source code change. Our tuning process considers these different types of tokens, the number of hidden layers, and the number of nodes in each layer together. Specifically, for each type of tokens we input them into our DBN model to generate features with different configurations. Similar to our tuning process of file-level defect prediction, for the number of hidden layers, we experiment with 11 discrete values, i.e., 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, i.e., 20, 50, 100, 200, 300, 500, 800, and 1,000. When we evaluate the seven different types of tokens and the two parameters, we set the number of iterations to 50 and keep it constant.

Table 8 shows the F1 scores of the change-level defect prediction with DBN-based semantic features generated by each of the seven types of tokens. Note that among the three basic token types (i.e., `added`, `deleted`, and `context`), the DBN-based features generated by `added` and `deleted` deliver better performance than `context` on all six projects. The improvement could be up to 20.8 percentage points (on project Jdt) and on average the improvement is larger than 8 percentage points. In addition, all the four different combinations, i.e., `added+deleted`, `added+context`, `deleted+context`, and `added+deleted+context`, can generate better performance than the corresponding three basic token types. This may be because the combinations provide more information to the DBN model for generating more effective features to capture buggy changes (a detailed discussion is provided in Section 6.2). Among the four combinations, `added+deleted+context` achieves the best performance.

In this work, we use the combination of `added`, `deleted`, and `context` tokens as input to DBN models to generate features. The corresponding best value of the number of hidden layers is 5 and the best value of the number of nodes in each hidden layers is 50. This means that the number of generated DBN-based features for change-level defect prediction is 50. Additionally, for change-level defect prediction, we also set the number of iterations to 200, with which the average error rate is less than 0.05 and the time cost for feature generation is less than 5 seconds.

4.6 File-level Within-Project Defect Prediction

To examine the performance of our semantic features on file-level within-project defect prediction, we build defect prediction models using three machine learning classifiers, i.e., ADTree, Naive Bayes, and Logistic Regression, which have been widely explored in previous work [31], [54], [55], [67], [112]. We use two consecutive versions of each project listed in Table 5 as the training and test data sets. We use the source code of an older version to train the DBN and generate the training feature set. Then we use the trained DBN to generate features for instances from a newer version. We compare our semantic features with the traditional features as described in Section 4.4. For a fair comparison, we use the same classifiers on these traditional features.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 8: The comparison of F1 scores among change-level defect prediction with different DBN-based features generated by the seven different types of tokens. The F1 scores are measured as a percentage. The best F1 values are highlighted in bold.

Project	added	deleted	context	added+deleted	added+context	deleted+context	added+deleted+context
Linux	39.2	39.8	32.5	39.8	40.1	40.6	**41.3**
PostgreSQL	48.9	49.5	39.6	49.8	51.8	50.1	**55.0**
Xorg	41.1	38.4	30.2	40.7	41.3	41.2	**41.4**
JDT	39.5	30.5	18.7	40.1	39.6	33.3	**41.4**
Lucene	37.2	38.1	31.4	37.8	38.9	38.5	**39.7**
Jackrabbit	45.3	44.7	39.5	45.6	46.6	47.8	**49.9**
Average	41.9	40.2	32.0	42.3	43.1	41.9	**44.8**

4.7 File-level Cross-Project Defect Prediction

Due to a 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 using data from mature projects (called *source projects*), and use the trained models to predict defects for new projects (called *target projects*). However, because the features of source projects and target projects often have different distributions, making an accurate and precise cross-project defect prediction model is still challenging [66].

We believe that the semantic features can capture the common characteristics of defects, which implies that the semantic features trained from one project can be used to predict defects in a different project, and so is 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 projects. Then, DBN-CP trains an ADTree based defect prediction model using data from the source project and uses the built model to perform defect prediction on the target project.

We choose TCA+ [67] as our baseline. To compare with TCA+, we design two different experiments. First, for each of the 16 test versions (which are the target versions in cross-project prediction) from the within-project experiments list in Table 9, we randomly select two source projects that are different from the target projects. Thus, 32 test pairs are collected. Our first experiment can help evaluate the performance of DBN-CP compared to TCA+ and the corresponding within-project defect prediction. Then, to extensively examine the performance of DBN-CP, we use each version from one project as a target project and each version from the other projects as a source project. In total, 606 test pairs are formed.

The reason why we use TCA+ for the comparison that TCA+ is one of the state-of-the-art techniques in cross-project defect prediction [67]. In our reproduction, we follow the processes described in [67]. We first implement all five of their proposed normalization methods and assign them the same conditions as given in the TCA+ paper. We then perform *Transfer Component Analysis* [73] on the source projects and the target projects together, and map them onto the same subspace while minimizing the data difference and maximizing the data variance. Finally, we use the source projects and target projects with the new features to build and evaluate the ADTree-based prediction models.

4.8 Change-level Within-Project Defect Prediction

To examine the effectiveness of the learned DBN-based features for change-level defect prediction tasks, we compare the performance of the DBN-based features to the three types of traditional features described in Section 4.4.2. By examining the combination of these traditional features, we should be able to generate the best performance for change-level defect prediction [29], [90]. In this work, we use the combination as the benchmark for change-level defect prediction.

To generate DBN-based semantic features, for each run of a project listed in Table 6, we use its training data to train a DBN (with the combination of all the tokens in a change as the input to the DBN). Then, we use the trained DBN to generate semantic features for both the training and test datasets. We then use the training data to build a defect prediction model and apply it to the test data. For the classification algorithm, we use ADTree in Weka [18] as the classifier, because it has delivered the best performance in previous work [29], [67], [90].

4.9 Change-level Cross-Project Defect Prediction

Similar to file-level defect models, change-level models also require a large amount of training data to train and build prediction models. However, sufficient training data are not often available when projects are in their initial development phases. To address this limitation, cross-project models for change-level prediction tasks are needed [34]. To explore the performance of the DBN-based semantic features in change-level cross-project defect prediction, we propose a technique called **DBN C**hange-level **C**ross-Project defect **P**rediction (`DBN-CCP`). Specifically, given a source project and a target project, DBN-CCP first trains a DBN by using the source project and generates semantic features for both the source project and the target project. Then, DBN-CCP trains a defect prediction model using data from the source project, and uses the built model to perform defect prediction on the target project.

For evaluating the performance of DBN-CCP, we also choose TCA+ [67] as our baseline. Note that TCA+ requires that the target and source projects have the same features for learning TCA+ based features. As described in Section 4.4.2, in this study we leverage three different types of features for change-level defect prediction, i.e.,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

bag-of-words features, characteristic features, and meta features. Both the bag-of-words features and characteristic features are project-specific and vary for different projects. Thus, for TCA+ on change-level prediction, we only use the meta features.

To extensively evaluate the performance of DBN-CCP, we use each test dataset in all runs from one project as a target dataset and each training dataset in all runs from the other projects as a source dataset to form change-level cross-project test pairs. For example, one test pair could be a training set from Run 1 of Project A and a test set from Run 1 of Project B, a training set from Run 2 of Project A and a test set from Run 1 of Project B, etc. In total, 1,380 test pairs are formed.

5 Results

5.1 RQ1: Performance of semantic features for file-level within-project defect prediction

5.1.1 Non-effort-aware evaluation scenario

We build file-level 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 baselines of traditional features. We conduct 16 sets of file-level within-project defect prediction experiments, each of which uses two versions from the same project (listed in Table 5). The older version is used to train the prediction models, and the newer version is used as the test set to evaluate the trained models.

Table 9 shows the performance of the file-level within-project defect prediction experiments. The highest F1 values 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 (AST nodes). For this comparison, the only difference is the three sets of features, meaning that the same classification algorithm, namely ADTree and the same training and test sets are used.

The results demonstrate that by using the DBN-based semantic features instead of the PROMISE features, we can improve the F1 by 14.2 percentage points on the 16 experiment pairs on average. The average improvements in the precision and recall are 14.7 percentage points and 11.5 percentage points 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.

5.1.2 Effort-aware evaluation scenario

For the effort-aware scenario, we rerun the 16 pairs of file-level within-project defect prediction experiments listed in Table 9, and calculate the `PofB20` of the test data in each experiment based on our setup description in Section 4.2.2.

TABLE 9: 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 the precision, recall, and F1 score respectively and are measured as a percentage. The better F1 values with statistical significance (p-value < 0.05) among the three sets of features are shown with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the **Semantic**. A positive value indicates that the semantic features improve the baseline features in terms of the effect size.

Project	Versions (Tr ⇒ T)	**Semantic***	PROMISE (0.555)	AST (0.656)
		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 10: `PofB20` scores of DBN-based features and traditional features for WPDP. The `PofB20` scores are measured as a percentage. The best values are in bold. The better `PofB20` values with statistical significance (p-value < 0.05) between the two sets of features are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the **Semantic**.

Project	Versions (Tr⇒T)	**Semantic***	PROMISE (0.756)
ant	1.5⇒1.6	**44.3**	16.3
	1.6⇒1.7	**50.2**	23.5
camel	1.2⇒1.4	**33.2**	33.8
	1.4⇒1.6	**30.1**	23.4
jEdit	3.2⇒4.0	**40.1**	29.3
	4.0⇒4.1	**32.6**	17.7
log4j	1.0⇒1.1	**25.0**	21.6
lucene	2.0⇒2.2	**32.1**	14.6
	2.2⇒2.4	**37.9**	23.2
xalan	2.4⇒2.5	**24.5**	8.3
xerces	1.2⇒1.3	**9.1**	7.2
ivy	1.4⇒2.0	**28.3**	15.1
synapse	1.0⇒1.1	**29.6**	13.3
	1.1⇒1.2	**32.5**	12.8
poi	1.5⇒2.5	**38.7**	26.2
	2.5⇒3.0	**25.5**	13.9
Average		**32.1**	18.8

Table 10 presents the `PofB20` of file-level within-project defect prediction models with DBN-based semantic features and the PROMISE features. As we can see, in all the experiments, DBN-based features could achieve better `PofB20` than the corresponding PROMISE features. Compared to the PROMISE features, the improvement could be as much as 26.7 percentage points (ant 1.6 ⇒ ant 1.7) and is, on average, 13.3 percentage points.

We further conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of the DBN-based

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

semantic features and PROMISE features for file-level within-project defect prediction with the 16 experiment pairs under both the non-effort-aware and effort-aware evaluation scenarios. The results suggest that the DBN-based semantic features are significantly better than the PROMISE features.

> Our DBN-based approach is effective in automatically learning semantic features, which significantly improves the performance of file-level within-project defect prediction under both non-effort-aware and effort-aware evaluation scenarios with large effect sizes.

5.1.3 RQ1a: Do semantic features outperform traditional features with other classification algorithms?

To answer this question, we build file-level within-project defect prediction models by using two alternative classification algorithms, i.e., Naive Bayes and Logistic Regression. We conduct 16 sets of file-level within-project defect prediction tests, where the training sets and the test sets are exactly the same as those in RQ1. Table 11 shows the F1 scores of running Naive Bayes and Logistic Regression on semantic features and PROMISE features. 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, the semantic features produce an F1 of 63.0%, which is 7.0 percentage points higher than using PROMISE features. For the same example with Logistic Regression as the classification algorithm, the semantic features achieve an F1 of 91.6%, while using PROMISE features produces an F1 of 50.6% only. Among the experiments with either Naive Bayes or Logistic Regression as the classification algorithm, the semantic features outperform the PROMISE features 14 out of the 16 times. On average, the Naive Bayes based defect prediction model with semantic features achieves an F1 of 60.0%, which is 14.8 percentage points higher than the Naive Bayes with PROMISE features. Similarly, the average F1 of using semantic features with Logistic Regression is 59.7%, which is 10.7 percentage points higher than Logistic Regression with PROMISE features.

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

TABLE 11: 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. The F1 scores are measured as a percentage.

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

TABLE 12: F1 scores of the file-level cross-project defect prediction for target projects explored in RQ1. The F1 scores are measured as a percentage. Better F1 values with statistical significance (p-value < 0.05) between DBN-CP and TCA+ are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to DBN-CP.

Source	Target	Cross-Project		Within-Project
		DBN-CP*	**TCA+ (0.274)**	Semantic Features
camel1.4	ant1.6	**97.9**	61.6	91.4
poi3.0	ant1.6	47.8	**59.8**	
camel1.2	ant1.7	31.2	**35.4**	94.2
jEdit3.2	ant1.7	41.7	**45.5**	
ant1.6	camel1.4	**31.6**	29.2	78.5
jEdit4.1	camel1.4	**69.3**	33.0	
ant1.5	camel1.6	**49.0**	21.3	37.4
lucene2.0	camel1.6	**49.2**	32.1	
xerces1.2	jEdit4.0	**51.9**	32.7	57.4
ivy1.4	jEdit4.0	35.0	**50.2**	
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	
xalan2.5	lucene2.2	**59.4**	56.1	65.1
log4j1.1	lucene2.2	**69.2**	52.4	
poi2.5	lucene2.4	**64.9**	54.4	77.3
xalan2.4	lucene2.4	**61.6**	60.9	
lucene2.2	xalan2.5	**55.0**	53.0	59.5
xerces1.3	xalan2.5	57.2	**58.1**	
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	
synapse1.1	poi2.5	**44.6**	40.6	64.0
ant1.6	poi2.5	**47.5**	44.7	
Average		**53.9**	46.1	64.1

5.2 RQ2: Performance of semantic features for file-level cross-project defect prediction

5.2.1 Non-effort-aware evaluation scenario

To answer this question, we compare our file-level cross-project defect prediction technique DBN-CP with TCA+ [67]. DBN-CP runs on the semantic features that are automatically generated by the DBN, while TCA+ uses the PROMISE features. For a fair comparison, we also provide a benchmark of within-project defect prediction. As described in Section 4.7, our preliminary experimental evaluation includes a set of 32 cross-project test pairs. Each experiment takes two versions separately from two different projects, with one used as the training set and the other used as the test set. The benchmark of the file-level within-project defect prediction uses the data from an older version of the target project as the training set.

Table 12 lists the F1 scores of the DBN-CP, TCA+, and the benchmark within-project defect prediction. The better F1 scores between the DBN-CP and TCA+ are in bold. Regarding the average F1, DBN-CP achieves 53.9%, which is 7.8 percentage points higher than the 46.1% of TCA+. As the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

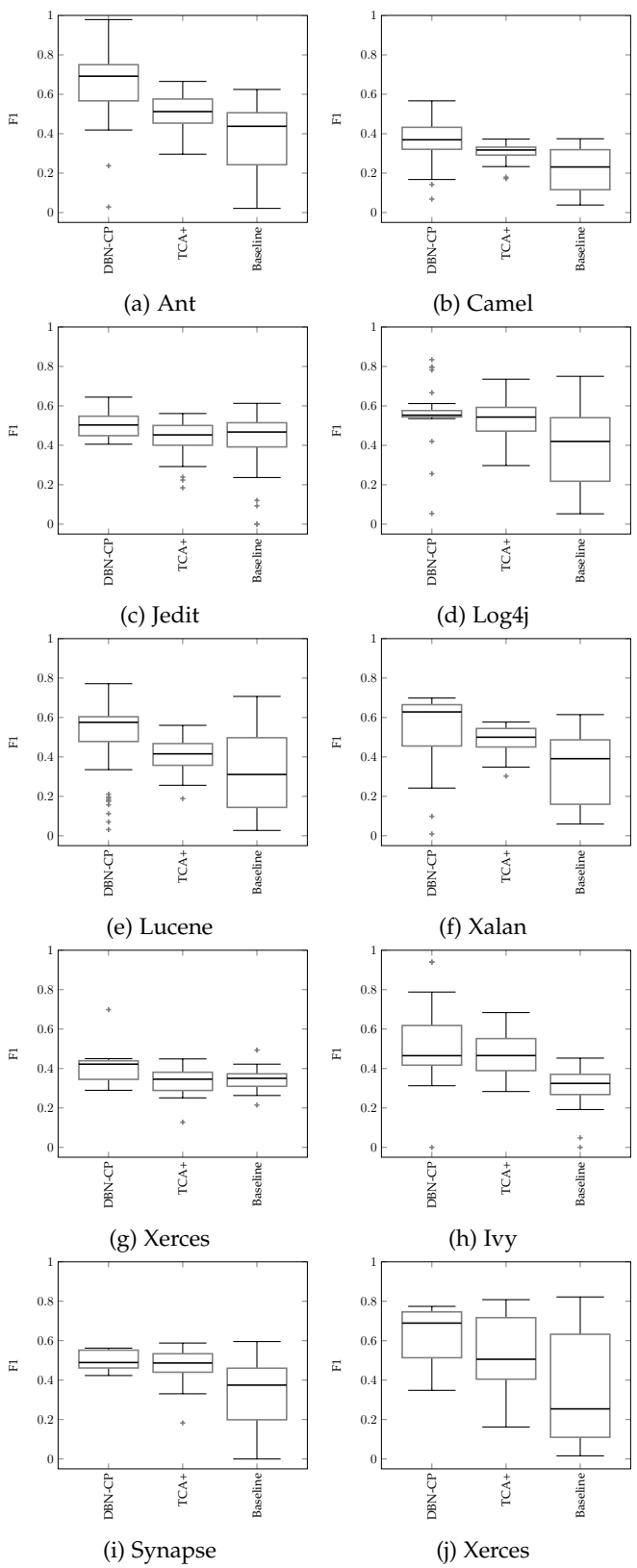


Fig. 10: Results of the DBN-CP, TCA+, and Baseline for CPDP.

source projects are randomly selected, we did not perform statistical tests for these experiments.

As described in Section 4.7, to extensively evaluate the performance of DBN-CP, we use each version from one project as the target project and one version from the other projects as the source project to form a file-level cross-project experiment test pair. This experiment includes 606 test pairs. Specifically, DBN-CP runs on the semantic features and TCA+ runs on generated features by using the PROMISE features. We also provide two benchmarks, i.e., Baseline and Within-Project. Baseline is the result of cross-project defect prediction with the original PROMISE features.

TABLE 13: F1 scores of file-level cross-project defect prediction for all projects listed in Table 5. The F1 scores are measured as a percentage. Better F1 values with statistical significance (p-value < 0.05) between DBN-CP, TCA+, and Baseline are shown with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CP.

Source	Target	DBN-CP	TCA+	Baseline	Within-Project
All Others	ant	**57.3***	50.9 (0.638)	38.9 (0.774)	92.8
	camel	**46.1***	32.7 (0.484)	22.7 (0.685)	57.9
	jEdit	**49.7***	43.9 (0.405)	44.5 (0.261)	59.4
	log4j	**56.2***	52.9 (0.231)	38.7 (0.450)	70.1
	lucene	**43.9***	41.0 (0.522)	31.7 (0.541)	71.2
	xalan	**46.2***	44.7 (0.363)	35.2 (0.577)	59.5
	xerces	**39.7***	33.4 (0.570)	34.6 (0.513)	41.1
	ivy	**41.4***	28.6 (0.148)	31.7 (0.782)	35.0
	synapse	**50.2***	47.9 (0.336)	35.3 (0.636)	56.4
	poi	**63.2***	58.1 (0.307)	34.9 (0.592)	72.2
Average		49.4	43.4 (0.401)	34.8 (0.628)	61.6

Table 13 shows the average F1 scores of the DBN-CP, TCA+, Baseline, and Within-Project defect prediction on each of the file-level projects. Overall, both DBN-CP and TCA+ deliver better performance than Baseline. Moreover, DBN-CP generates a better F1 than TCA+ on all 10 projects listed, and the improvement is as much as 12.8 percentage points (ivy) and is, on average, 6.0 percentage points higher. Compared with the within-project defect prediction, DBN-CP improves the cross-project defect prediction by reducing the gap to approximately 12 percentage points. The statistical tests also show that overall DBN-CP is significantly better than both TCA+ and Baseline.

Figure 10 shows the boxplots of the F1 scores for DBN-CP, TCA+, and Baseline for the 10 projects listed in Table 5. Specifically, each boxplot presents the F1 distribution (median and upper/lower quartiles) of each of the three approaches for cross-project file-level defect prediction. The boxplots indicate that overall, both DBN-CP and TCA+ perform better than Baseline, and that DBN-CP performs better than TCA+ and Baseline on almost all projects.

5.2.2 *Effort-aware evaluation scenario*

For the effort-aware evaluation, we also calculate the `PofB20` for the DBN-CP, TCA+, and Baseline approaches on each of the target projects.

Table 14 shows the `PofB20` of the three file-level cross-project defect prediction models. The highest `PofB20` values among the three approaches are shown in bold. In all the experiments, DBN-CP achieves better `PofB20` than both TCA+ and Baseline. The `PofB20` scores of DBN-CP vary from 21.8 to 37.6 percentage points across the 606 experiments, and the average `PofB20` score of DBN-CP is 29.5 percentage points. Compared to TCA+, the improvement is as high as 21.8 percentage points (Poi) and is, on average,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 14: PofB20 scores of DBN-based features and traditional features for CPDP. PofB20 scores are measured in percentage. Better PofB20 values with statistical significance (p-value < 0.05) among DBN-CP, TCA+, and Baseline are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CP.

Source	Target	DBN-CP	TCA+	Baseline
All Others	ant	**28.3***	28.1 (0.434)	18.3 (0.888)
	camel	**32.7***	14.8 (0.982)	10.7 (1)
	jEdit	**23.2***	21.8 (0.302)	19.6 (0.462)
	log4j	**28.6***	19.1 (0.787)	18.4 (0.855)
	lucene	**30.5***	15.6 (1)	10.9 (1)
	xalan	**37.6***	15.5 (0.900)	14.6 (0.910)
	xerces	**29.1***	22.5 (0.479)	12.9 (0.855)
	ivy	**26.5***	20.1 (0.488)	17.9 (0.789)
	synapse	**21.8***	19.2 (0.133)	15.7 (0.450)
	poi	**36.7***	14.9 (0.994)	11.2 (1)
Average		**29.5**	19.2 (0.650)	15.0 (0.821)

TABLE 15: Overall results of the change-level within-project defect prediction. All values are measured as a percentage. The better F1 values with statistical significance (p-value < 0.05) between the two sets of features are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the Semantic Features in terms of F1.

Projects	Features	P	R	F1
Linux	Change Features [29], [90] (0.821)	28.7	49.5	36.3
	Semantic Features	**32.5**	**56.9**	**41.3***
PostgreSQL	Change Features (1)	44.1	49.0	46.4
	Semantic Features	**51.6**	**58.8**	**55.0***
Xorg	Change Features (0.653)	29.1	51.5	37.2
	Semantic Features	**31.7**	**59.4**	**41.4***
JDT	Change Features (0.421)	32.5	41.5	36.4
	Semantic Features	30.2	**65.9**	**41.4***
Lucene	Change Features (0.136)	33.0	46.8	38.7
	Semantic Features	31.4	**54.0**	**39.7***
Jackrabbit	Change Features (0.525)	46.0	44.6	45.3
	Semantic Features	**49.3**	**50.4**	**49.9***
Average	Change Features (0.593)	36.3	50.6	40.1
	Semantic Features	**37.6**	**56.6**	**44.8**

10.3 percentage points. The results of the Wilcoxon signed-rank test ($p < 0.05$) also indicate that DBN-CP is overall significantly better than both TCA+ and Baseline.

DBN-CP significantly improves the performance of file-level cross-project defect prediction under both non-effort-aware and effort-aware evaluation scenarios with a nontrivial effect. This implies that the semantic features learned by the DBN are effective and are able to capture the common characteristics of defects across projects.

5.3 RQ3: Performance of semantic features for change-level within-project defect prediction

5.3.1 Non-effort-aware evaluation scenario

To answer this question, we use different features to build change-level within-project defect prediction models, e.g., DBN-based semantic features, and three change features described in Section 4.4.2 (i.e., the bag-of-words features, the characteristic features, and the meta features). As we described in Section 4.3.2, in the change-level dataset, each project has multiple runs. Thus, we use the training data from each run to build and train the ADTree based prediction model and evaluate its performance on the test data in this run. To show the overall performance, we use the weighted average precision, recall, and F1 following existing work [29], [90].

Table 15 shows the precision, recall, and F1 of the within-project change-level defect prediction experiments. Overall, the DBN-based features generate better results than the traditional change features in terms of F1. Specifically, for all the projects, the DBN-based features could improve the best existing change features up to 8.6 percentage points in F1, and the improvement is 4.7 percentage points, on average.

TABLE 16: PofB20 scores of the DBN-based features and the traditional features for WCDP. The PofB20 scores are measured as a percentage. The best values are in bold. The better PofB20 values with statistical significance (p-value < 0.05) among these two sets of features are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the DBN-based semantic features.

Project	Semantic Features	Change Features
Linux	**28.6***	25.0 (0.324)
PostgreSQL	**29.2***	8.1 (1)
Xorg	**37.6***	24.8 (0.901)
JDT	**23.8***	14.5 (1)
Lucene	**28.1***	21.9 (0.887)
Jackrabbit	**27.9***	21.3 (0.621)
Average	**29.2**	19.3 (0.789)

5.3.2 Effort-aware evaluation scenario

We further evaluate the DBN-based semantic features and traditional change features for change-level within-project defect prediction with the PofB20 metric.

Table 16 shows the PofB20 of the change-level within-project defect prediction models with DBN-based semantic features and the traditional change features. The DBN-based features could achieve better PofB20 scores than the corresponding change features in all the experiment pairs. The PofB20 scores (measured as a percentage) of DBN-based features vary from 23.8 to 37.6 across the experiments, and the average PofB20 score of the defect prediction models with DBN-based features is 29.2. Compared to the change features, the improvement could be up to 21.1 percentage points (PostgreSQL) and is 9.9 percentage points, on average. In addition, the statistical test, i.e., the Wilcoxon signed-rank test ($p < 0.05$), also suggests that the DBN-based features are overall significantly better than the change features under both the non-effort-aware and effort-aware evaluation scenarios.

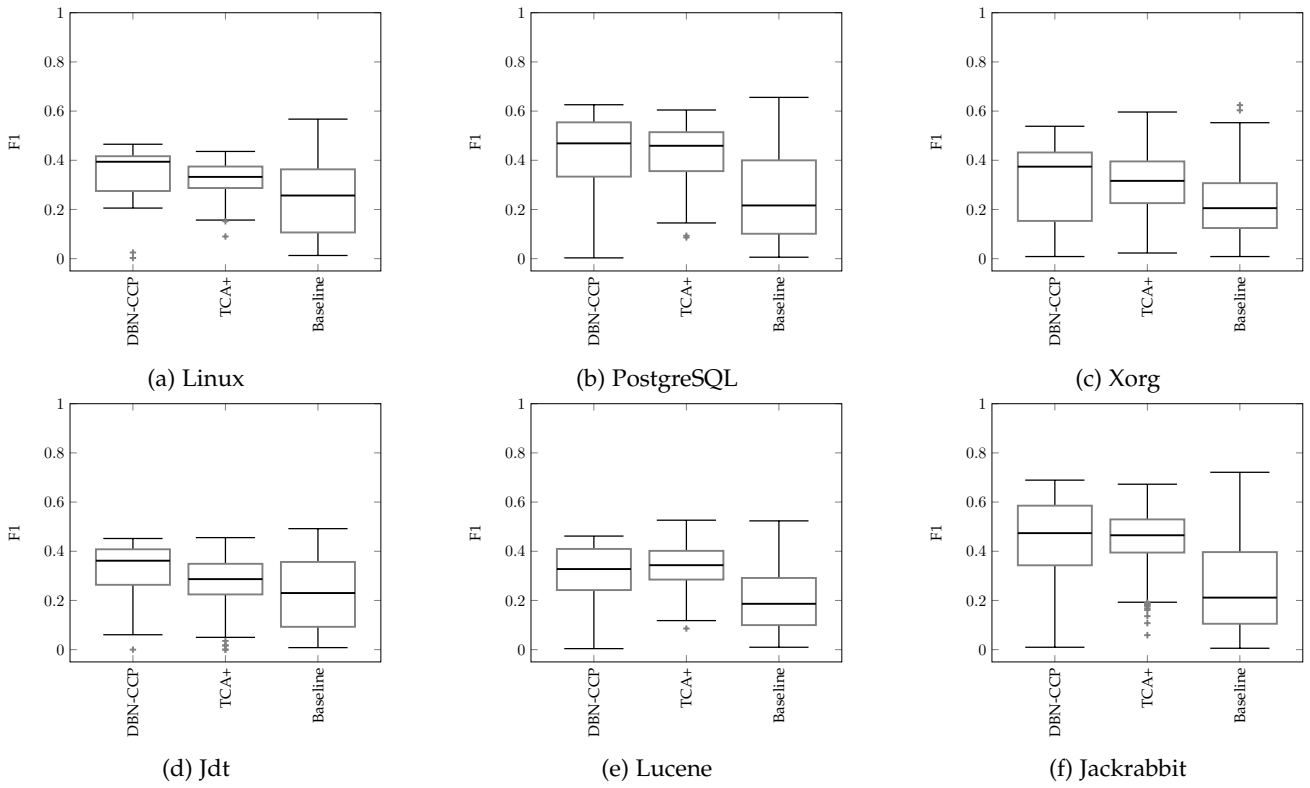
The semantic features automatically learned from the DBN could improve the change-level within-project defect prediction with statistical significance under both non-effort-aware and effort-aware evaluation scenarios with nontrivial effect sizes.

5.4 RQ4: Performance of semantic features for change-level cross-project defect prediction

5.4.1 Non-effort-aware evaluation scenario

To answer this question, we compare our cross-project change-level defect prediction technique DBN-CCP with

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering



(a) Linux (b) PostgreSQL (c) Xorg

(d) Jdt (e) Lucene (f) Jackrabbit

Fig. 11: Results of DBN-CCP, TCA+, and Baseline for CCDP.

TABLE 17: F1 scores of change-level cross-project defect prediction for all projects. The F1 scores are measured as percentages. The better F1 values with statistical significance (p-value < 0.05) between DBN-CCP, TCA+, and Baseline are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to DBN-CCP.

Source	Target	**DBN-CCP**	**TCA+**	**Baseline**	**Within**
All Others	Linux	**35.1***	32.4 (0.295)	24.7 (0.439)	41.3
	PostgreSQL	**44.2***	43.6 (0.130)	25.7 (0.370)	55.0
	Xorg	**31.8***	30.4 (0.128)	22.8 (0.310)	41.4
	JDT	**33.3***	27.3 (0.360)	22.6 (0.566)	41.4
	Lucene	**31.3***	30.2 (0.129)	21.3 (0.520)	39.7
	Jackrabbit	**44.4***	43.3 (0.131)	26.5 (0.463)	49.9
Average		36.7	34.5 (0.196)	23.9 (0.445)	44.7

TABLE 18: `PofB20` scores of the DBN-based features and traditional features for CCDP. The `PofB20` scores are measured as a percentage. The best values are in bold. The better `PofB20` values with statistical significance (p-value < 0.05) among the DBN-CCP, TCA+, and Baseline are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to DBN-CCP.

Source	Target	**DBN-CCP**	**TCA+**	**Baseline**
All Others	Linux	**24.7***	24.1 (0.255)	18.5 (0.500)
	PostgreSQL	**20.7**	20.3 (0.019)	15.9 (0.438)
	Xorg	**22.7***	22.0 (0.110)	19.7 (0.511)
	JDT	**25.6***	22.6 (0.273)	13.5 (0.360)
	Lucene	**18.1**	18.0 (0.030)	17.0 (0.371)
	Jackrabbit	**19.3***	16.4 (0.352)	16.1 (0.343)
Average		**21.9**	20.6 (0.180)	16.8 (0.421)

TCA+. For a fair comparison, we also provide two benchmarks, i.e., Baseline and Within-Project. The Baseline is the result of change-level defect prediction with the original change features. As we described in Section 4.9, we use the test data of one run from one project as the target project and the training data of one run from a different project as the source project to form the change-level cross-project test pairs (in total 1,380 pairs). For each test pair, we build the ADTree based defect prediction model using the three different sets of features.

Table 17 shows the average F1 scores of the DBN-CCP, TCA+, Baseline, and Within-Project for each of the change-level projects. Overall, both DBN-CCP and TCA+ deliver better performance than Baseline. Moreover, DBN-CCP generates a better F1 than TCA+ on all the projects on average. The improvement is as high as 6.0 percentage points and is 2.2 percentage points higher, on average. Compared to the within-project defect prediction, DBN-CCP improves the cross-project defect prediction by reducing the gap to only 8.0 percentage points.

Figure 11 shows the boxplots of the F1 scores for DBN-CCP, TCA+, and Baseline for the six projects listed in Table 6. Specifically, each boxplot presents the F1 distribution (median and upper/lower quartiles) of each of the three approaches for the change-level cross-project defect prediction. The boxplots show that overall both DBN-CCP and TCA+ perform better than Baseline, moreover DBN-CCP performs better than TCA+ and Baseline on almost all projects.

5.4.2 *Effort-aware evaluation scenario*

We also calculate the `PofB20` score for the DBN-CCP, TCA+, and Baseline approaches on each of the target projects when conducting change-level cross-project defect prediction. Table 18 shows the `PofB20` values of the three change-level cross-project defect prediction models. The highest `PofB20`

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 19: Time and space costs of generating semantic features for file-level defect prediction (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

values among the three approaches are shown in bold. DBN-CCP achieves better `PofB20` scores than both TCA+ and Baseline. On average, the `PofB20` score (measured as a percentage) of DBN-CCP is 21.9. Compared to TCA+, the improvement can be up to 3.0 percentage points (Jdt) and is 1.3 percentage points, on average. The results of the Wilcoxon signed-rank test ($p < 0.05$) also indicate that the performance of DBN-CCP is, overall, significantly better than TCA+ and Baseline under both the non-effort-aware and effort-aware evaluation scenarios.

DBN-CCP significantly improves the performance of the change-level cross-project defect prediction compared to the traditional change features with a nontrivial effect.

5.5 Time and Memory Overhead

To understand the space cost of file-level defect prediction, during file-level defect prediction experiments, we keep track of the time cost and memory space cost for our DBN-based feature generation process (details are in Section 3.3). In addition, we also have recorded the time cost for tuning the DBN models in our experiments. The other processes, including parsing source code, handling noise, mapping tokens, building models, and predicting defects, are all common procedures, so we do not analyze their costs.

As described in Section 4.5.1, we tune the three parameters, i.e., the number of hidden layers, the number of nodes in each layer, and the number of iterations, for the randomly selected five projects. To find the best combination among the three parameters, we have $11 \times 8 \times 10$ experiments. In total, the tuning process costs approximately 5 hours.

Table 19 shows the time cost and the memory space cost of each project for generating semantic features. As shown in Table 9, `ant` has two sets of within-project defect prediction experiments, which are `ant 1.5` $\Rightarrow$ `1.6` and `ant 1.6` $\Rightarrow$ `1.7`. On average, it takes the two experiments 15.5 seconds and 2.8 MB memory for the DBN to generate the semantic features for both the training data and the test data. Among all the projects, the time cost of automatically generating the semantic features varies from 8.0 seconds (`ivy`) to 32.0 seconds (`camel`). For the memory space cost, it takes less than 6.5MB for all the examined projects.

In addition, we also keep track of the time and memory space cost for generating DBN-based features for the change-level defect prediction during our experiments. Different from the file-level defect prediction that predicts whether a file contains bugs or not, change-level defect prediction predicts whether a change is buggy or clean. Source files often contain hundreds of LOC, while changes often have fewer lines than files. Thus, both the time and memory costs of generating DBN-based features for changes are smaller than those for files. In our experiments, the average time cost and memory cost of generating DBN-based features for changes are 2.4 seconds and 0.6 MB.

Our DBN-based approach to automatically learning semantic features is applicable in practice.

6 Discussion

6.1 Why Do DBN-based Semantic Features Work?

Our experiments in Section 5 show that compared to traditional features, the DBN-based features that are directly learned from source code deliver significantly better performance for all the four defect prediction tasks investigated in this work. The probable reasons for the outstanding performance of DBN-based features are summarized as follows.

First, the DBN models generate features with more complex network connections. These network connections enable the DBN models to generate features with multiple levels of abstraction and high-level semantics. In this work, the generated DBN features are weighted combinations/vectors of original input source code, which could represent patterns of the usages of the input source code, e.g., method usages, control-flow usages, etc. While traditional features often focus on statistical information of the source code, e.g., LOC, the number of function calls, etc., which cannot capture the semantic information. Although the `Bag-of-words feature` or the `Characteristic feature` (details are in Section 4.4) are derived from the raw programming tokens, they consider each token as an independent feature element and cannot represent the contextual and structural information among the raw tokens. Thus, these features have underperformed.

Second, the DBN-based features are more capable of distinguishing between the semantic information of different code snippets, especially for code snippets that have similar source code characteristics. For example, as shown in Figure 1, the traditional features (e.g., code complexity) of the two code snippets are identical. Training prediction models containing them will degrade the discrimination ability of classifiers and consequently hurt the prediction performance. While the DBN-based features can make a difference, as shown in Figure 4, the different structural and contextual information among tokens of these two code snippets enables the DBN model to generate different features to distinguish between these two code snippets.

6.2 Efficiency of Different Types of Tokens in Change-level Defect Prediction

As described in Section 4.5.2, to achieve better prediction performance for change-level defect prediction, we use the combination of the three types of tokens, i.e., `added`,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 20: Information gain of different types of tokens and their combinations that are used for generating DBN-based semantic features in change-level defect prediction. **Spearman** is the Spearman correlation value between the information gain and the prediction results of different types of tokens for a project.

Project	added	deleted	context	added+deleted	added+context	deleted+context	added+deleted+context	Spearman
Linux	4.2	3.8	3.0	4.3	5.0	4.9	5.1	0.90
Postgresql	5.9	5.5	5.3	6.0	6.7	6.6	7.6	0.96
Xorg	4.8	4.0	4.9	5.5	6.0	5.9	6.1	0.82
JDT	8.1	7.0	7.1	8.0	9.8	8.4	8.9	0.64
Lucene	6.9	6.5	6.3	7.1	7.6	7.4	7.7	0.89
Jackrabbit	8.2	7.4	7.4	8.2	8.6	8.4	8.9	0.96
Average	6.4	5.7	5.6	6.5	7.3	6.9	7.4	0.86

`deleted`, and `context`, to generate DBN-based semantic features. In this section, we further examine the reason why the combination outperforms each of the three types of tokens extracted from changes. One possible reason is that, the combination contains more information than any of the three types of tokens. To explore this, we leverage the information gain [13], which is a widely used metric to measure how much information there is in a given event, to measure the information in each of the three types of tokens and their different combinations.

Specifically, given a document $s = \{a_1 \ ... \ a_n\}$ of length n, a_1 to a_n are tokens in the document s. The information gain of this document $H(s)$ is measured as follows:

$$H(s) = \sum_{i=1}^{n} -p_i \log p_i \tag{9}$$

where p_i is the probability of token a_i in the document s. We use the TF (term frequency) of token a_i to represent its probability in the document s.

To calculate the information gain of a specific type of token in changes, we first collect all seven types of tokens from all the changes in a project. Then, we calculate the information gain of a specific type of tokens extracted from all changes of a project. Table 20 shows the various information gains of the three types of tokens and their combinations. Overall, among the basic three types of tokens, `added` and `deleted` contain more information than `context`. The combination of either two of them could achieve better performance than either of the two types of tokens. In addition, the combination of all the three types of tokens contains more information than any other combinations. We further compute the Spearman correlation between the value of information gain and the prediction result of different types of tokens in a project. The high correlation value (on average 0.86) indicates that the prediction result of DBN-based features generated from a specific type of token has a positive correlation with its information gain. This explains why DBN-based features generated from the combination of all the three types of tokens achieve the best performance.

6.3 Analysis of the Performance

In this section, we evaluate the performance of our proposed DBN-based semantic features on both file-level and change-level defect prediction tasks. We can observe that the improvement of DBN-based semantic features on file-level defect prediction is generally better than change-level defect prediction. The main reason for this phenomenon is that a file generally contains more information than a change. Thus, file-level defect prediction data often provide more context to a DBN model, allowing it to learn more accurate features.

We also note that our approach achieves better performance on some projects than others for file-level with-project defect prediction, e.g., it achieves an F1 of 94.2% on `ant` and an F1 of approximately 80% on `camel`, `lucene`, `poi`, and `jEdit`. This is because we use these projects, i.e., `ant`, `camel`, `lucene`, `poi`, and `jEdit`, as data to train a DBN model for generating features. During the training process, we tune the DBN parameters based on the performance of the defect prediction models with the generated features for the five projects (details are presented in Section 4.5.1). Because the training process is an optimization task to generate features that may produce the best performance for the training dataset, the features fit the training dataset better. Thus, our approach achieves relatively higher F1 values for the five projects (`ant`, `camel`, `lucene`, `poi`, and `jEdit`) than other projects. This may be a risk of overfitting. However, this may also suggest that training a DBN model by using a project's own history data is appropriate when applying our approach to the project.

6.4 Performance on Open-source Commercial Projects

In Section 4, we evaluated the DBN-based semantic features on 15 open source projects (i.e., the projects listed in Table 5 and Table 6). To explore the performance of the DBN-based semantic features on commercial projects, we apply our approach to four additional open-source commercial projects, i.e., Buck[4], Hhvm[5], Guava[6], and Skia[7]. Buck is a build system developed and used by Facebook. Hhvm is a virtual machine, which was also developed and is currently used by Facebook. Guava is a set of Google's core libraries for Java. Skia is a complete 2D graphics library for drawing text, geometries, and images developed and used by Google. These four projects were originally developed and maintained by Facebook and Google and became open-source projects recently. We selected these four projects, because they are the largest Java/C++ projects (in terms of commit size). To collect the change-level data, we use the same approaches as we described in Section 2.2 and Section 4.4 to label the changes and collect the features for each change.

4. https://buckbuild.com/
5. https://hhvm.com/
6. https://github.com/google/guava
7. https://skia.org/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

TABLE 21: The four open-source commercial projects evaluated. **Lang** is the programming language used for the project. **LOC** is the number of the line of code. **First Date** is the date of the first commit of a project, while **Last Date** is the date of the latest commit. **Changes** are the number of changes collected in this work. **TrSize** is the average size of the training data on all runs. **TSize** is the average size of the test data on all runs. **NR** is the number of runs for each subject.

Project	Lang	LOC	First Date	Last Date	Changes	TrSize	TSize	Average Buggy Rate (%)	# NR
Buck (Facebook)	JAVA	296K	2013-04-18	2018-03-23	92K	31k	19k	7.2	3
Hhvm (Facebook)	C++	1M	2010-02-03	2018-04-06	120K	51K	14K	11.6	3
Guava (Google)	JAVA	380K	2009-06-18	2018-03-21	29K	8.6K	8.8K	4.0	2
Skia (Google)	C++	765K	2006-09-20	2018-04-07	147K	48K	22K	30.6	5

The details of the four open-source commercial projects are listed in Table 21.

With these four additional projects, we conduct change-level within-project and change-level cross-project defect prediction tasks to compare the DBN-based semantic features to traditional features under both the non-effort-aware and effort-aware scenarios. Note that we adopt the same procedures to tune the DBN models and generate semantic features as described in Section 4.8 and Section 4.9.

Table 22 shows the results of the change-level within-project prediction on the four projects. Overall, the DBN-based features generate better results than traditional change features in terms of F1, which is consistent with our previous experiment results on pure open-source projects 15. Specifically, for all four projects, DBN-based features could improve the best existing change features up to 6.6 percentage points in F1, and on average the improvement is 5.4 percentage points. These improvements are consistent with our previous experimental results on open-source projects (i.e., the best improvement is 8.6 percentage points and the average improvement is 4.7 percentage points).

TABLE 22: Results of WCDP on the four projects. All values are measured in percentage. Better F1 values with statistical significance (p-value < 0.05) between the two sets of features are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to Semantic Features in terms of F1.

Projects	Features	P	R	F1
Buck	Change Features (0.542)	10.9	39.9	17.2
	Semantic Features	**14.0**	**46.6**	**21.6***
Hhvm	Change Features(0.477)	14.2	**39.8**	20.9
	Semantic Features	**23.6**	33.1	**27.5***
Guava	Change Features (0.685)	7.4	58.0	13.1
	Semantic Features	**10.5**	**63.2**	**18.1***
Skia	Change Features (0.710)	34.5	40.4	37.3
	Semantic Features	**45.2**	**42.9**	**44.0***
Average	Change Features (0.622)	16.7	44.5	22.4
	Semantic Features	**23.3**	**46.5**	**27.8**

TABLE 23: F1 scores of change-level cross-project defect prediction for the four projects. Better F1 values with statistical significance (p-value < 0.05) between DBN-CCP, TCA+, and Baseline are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CCP.

Source	Target	DBN-CCP	TCA+	Baseline	Within
All Others	Buck	**14.3***	11.9 (0.459)	10.8 (0.693)	21.6
	Hhvm	**22.6***	21.7 (0.017)	20.2 (0.112)	27.5
	Guava	7.2	**7.6*** (-0.024)	4.2 (0.407)	18.1
	Skia	**47.9***	38.0 (0.613)	36.9 (0.765)	44.0
Average		**23.0***	19.8 (0.358)	18.0 (0.422)	27.8

TABLE 24: `PofB20` scores of DBN-based features and traditional features for WCDP on the four projects. `PofB20` scores are measured in percentage. Better `PofB20` values with statistical significance (p-value < 0.05) among these two sets of features are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-based semantic features.

Project	**Semantic Features**	**Change Features**
Buck	**28.2***	14.7 (1)
Hhvm	**21.9***	13.3 (0.882)
Guava	**17.4***	15.5 (0.351)
Skia	**27.0***	21.3 (0.655)
Average	**23.6***	16.2 (0.520)

TABLE 25: `PofB20` scores of DBN-based features and traditional features for CCDP on the four projects. Better `PofB20` values with statistical significance (p-value < 0.05) among DBN-CCP, TCA+, and Baseline are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CCP.

Source	Target	DBN-CCP	TCA+	Baseline
All Others	Buck	**25.0***	17.9 (0.801)	14.2 (1)
	Hhvm	13.4	**17.4*** (-0.653)	12.0 (0.455)
	Guava	**14.3***	13.0 (0.625)	10.5 (0.746)
	Skia	**18.2***	17.1 (0.210)	16.9 (0.437)
Average		**18.7***	16.4 (0.623)	13.4 (0.766)

Table 23 shows the results of the change-level cross-project prediction for the four projects. Overall, DBN-CCP generates a better F1 than both TCA+ and Baseline for all four projects on average. The improvement is up to 9.9 percentage points and is 3.2 percentage points on average. The results are also consistent with our previous change-level cross-project defect prediction results listed in 17.

We also calculate the `PofB20` values for both the change-level within-project and cross-project approaches. Table 24 shows the `PofB20` values of the change-level within-project defect prediction models with DBN-based semantic features and the traditional change features. DBN-based features achieve better `PofB20` values than the corresponding change features for all experiment pairs. The improvement is up to 13.5 percentage points (Buck) and is 7.4 percentage points on average. Table 25 shows the `PofB20` values of the three change-level cross-project defect prediction approaches. Similar to our previous results, DBN-CCP achieves better `PofB20` values than both TCA+ and Baseline on average. Compared to TCA+, the improvement is as high as 7.1 percentage points and is 2.3 percentage points, on average.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

DBN-based semantic features outperform traditional features on four open-source commercial projects from Facebook and Google, which indicates that DBN-based semantic features are also applicable for improving defect prediction for open-source commercial projects.

6.5 Threats To Validity

6.5.1 Implementation of TCA+

For the comparative analysis, we compare our cross-project defect prediction models with TCA+ [67], which is the state-of-the-art cross-project defect prediction technique with traditional features. Since the original implementation is not released, we 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 with the data provided by their work. Since our implementation can 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+ [67]. This is because our DBN-based feature generation approach to within-project defect prediction works on data of two different versions from the same project. However, the datasets used in [67] 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. To reduce this threat, we evaluated TCA+ and our approach on the publicly available projects from PROMISE.

6.5.2 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 the 15 projects used in our experiments are not generalizable enough to represent all software projects. Our approach might generate better or worse results for other projects that are not used in the experiments. We mitigate this threat by selecting projects of different functionalities (operating systems, servers, and desktop applications) that are developed in different programming languages (C and Java).

Our approach to generating semantic features is only evaluated on open source projects. While we believe that this approach should be generalizable to proprietary software, evaluating our approach on proprietary software is challenging, because the approach requires AST analysis of source code. We mitigate this threat by applying our approach to four open source commercial projects that were originally developed and maintained by Google and Facebook and are open source now. The performance of these projects suggests our proposed DBN-based semantic features could deliver better results than traditional features.

6.5.3 Labeling Data

Following previous work [40], [88], the labeling process is automatically completed with the annotating or blaming function in VCS. It is known that this process can introduce noise [29], [39]. The noise in the data can potentially harm the performance of defect prediction. Manual inspection of the process shows reasonable precision and recall on open source projects [29]. To mitigate this threat, we use the noise data filtering algorithm introduced in [39].

7 Related Work

7.1 Software Defect Prediction

There are many software defect prediction techniques [15], [22], [29], [31], [41], [45], [53], [59], [61]–[63], [65], [70], [80], [83], [96], [99], [109], [110], [112], most of which leverage features that are extracted from the repositories of projects to train machine learning based classifiers [55]. Commonly used features can be divided into code features and process features [54]. Code features, e.g., Halstead [19], McCabe's cyclomatic complexity [50], CK [8], and MOOD features [21], have been widely examined and used for defect prediction. Recently, process features have been proposed and used for defect prediction. Moser et al. [59] used the number of revisions, authors, past fixes, and ages of files as features to predict defects. Nagappan et al. [62] proposed code churn features, and showed that these features were effective for defect prediction. Hassan et al. [22] used entropy of change features to predict defects. Their evaluation of six projects showed that their proposed features can significantly improve the results of defect prediction in comparison to other change features. Lee et al. [45] proposed 56 micro interaction metrics to improve defect prediction. Their evaluation results on three Java projects showed that their proposed features can improve defect prediction results compared to traditional features. Other process features, including the developers' characteristics [29], [71] and collaboration between developers [45], [55], [76], [100], have also been used to build defect prediction models. In this work, we adopted process features such as the commit time, filename, developers, the added line count, the deleted line count, the changed line count, etc., which are included in the meta features (details are presented in Section 4.4.2). We did not compare the DBN-based semantic features with the meta features alone since we found that combining them with the bag-of-words and characteristic vector outperforms using them alone. The results in Section 5 show that compared with the combination of the three benchmark features (including some process features), our DBN-based semantic features produce better performance.

The main difference between our DBN-based semantic features and the above traditional features is that traditional features are manually encoded and mainly focus on the statistical information of the source code, e.g., LOC, the number of function calls, etc., while our DBN-based semantic features are automatically learned by using deep learning techniques and try to capture patterns of the usages of tokens (e.g., method usages, control-flow usages, etc.) in the source code.

In this work, we rigorously compare the DBN-based semantic features with traditional defect prediction features on two different defect prediction tasks—within-project defect prediction and cross-project defect prediction.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

7.1.1 Within-Project Defect Prediction

Within-project defect prediction uses training data and test data that are from the same project. Many machine learning algorithms have been adopted for within-project defect prediction, including Support Vector Machine (SVM) [12], Bayesian Belief Network [1], Naive Bayes (NB) [93], Decision Tree (DT) [14], [37], [96], Neural Network (NN) [72], [78], and Dictionary Learning [31].

Elish et al. [12] 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. Their results showed that SVM is generally better than, or at least, is competitive against other models, e.g., Logistic Regression, Bayesian techniques, etc. Amasaki et al. [1] proposed an approach to predicting the final quality of a software product by using the Bayesian Belief Network. They evaluated their approach on a closed project, and the results showed that their proposed approach can predict bugs that the Software Reliability Growth Model (SRGM) cannot handle. Wang et al. [96] and Khoshgoftaar et al. [37] examined the performance of Tree-based machine learning algorithms on defect prediction, their results suggested that Tree-based algorithms could help defect prediction. Tao et al. [93] proposed a Naive Bayes based defect prediction model, and they evaluated the proposed approach on 11 datasets from the PROMISE defect data repository. Their experiment results showed that the Naive Bayes based defect prediction models could achieve better performance than J48 (decision tree) based prediction models. Our previous work [97] used the deep belief network to generate semantic features for file-level defect prediction tasks. The new contributions made by this paper are described in Section 1.

In this work, to evaluate the performance of our DBN-based semantic features and the traditional defect prediction features, we built prediction models by using three typical machine learning algorithms, i.e., ADTree, Naive Bayes, and Logistic Regression. Our experiment results show that the learned DBN-based semantic features consistently outperform the traditional defect prediction features on these machine learning classifiers.

Most of the above approaches are designed for file-level defect prediction. For change-level defect prediction, Mockus and Weiss [57] and Kamei et al. [35] predicted the risk of a software change by using change measures, e.g., the number of subsystems touched, the number of files modified, the number of added lines, and the number of modification requests. Kim et al. [38] used the identifiers in added and deleted source code and the words in change logs to classify changes as being defect-prone or clean. Jiang et al. [29] and Xia et al. [104] built separate prediction models with the characteristic features and meta features for each developer to predict software defects in changes. Tan et al. [90] improved the change classification techniques and proposed the online defect prediction models for imbalanced data. Their approach used time sensitive change classification to address the incorrect evaluation introduced by cross-validation. McIntosh et al. [51] studied the performance of change-level defect prediction as software systems evolve. Change classification can also predict whether a commit is buggy or not [75], [77].

In this work, we also compare the DBN-based semantic features with the widely used change-level defect prediction features, and our results suggest that the DBN-based semantic features can also outperform these change-level defect prediction features.

7.1.2 Cross-Project Defect Prediction

Due to the lack of data, it is often difficult to build accurate models for new projects. Some studies [42], [95], [111] have been done on evaluating cross-project defect prediction against within-project defect prediction and show that cross-project defect prediction is still a challenging problem. He et al. [24] showed the feasibility to find the best cross-project models among all available models to predict defects on specific projects. Turhan et al. [94] proposed using a nearest-neighbor filter to improve cross-project defect prediction. Nam et al. [67] proposed TCA+, which adopted a state-of-the-art technique called Transfer Component Analysis (TCA) and the optimized TCA's normalization process to improve cross-project defect prediction. Xia et al. [103] proposed HYDRA, which leverages a genetic algorithm and ensemble learning (EL) to improve cross-project defect prediction. HYDRA requires massive training data and a portion (5%) of labeled data from test data to build and train the prediction models.

TCA+ [67] and HYDRA [103] are the two state-of-the-art techniques for cross-project defect prediction. However, in this work, we only use TCA+ as our baseline for cross-project defect prediction. This is because HYDRA requires that the developers manually inspect and label 5% of the test data, while in real-world practice, it is very expensive to obtain labeled data from software projects, which requires the developers' manually inspection, and the ground truth might not be guaranteed.

Most of the above existing cross-project approaches are examined for file-level defect prediction only. Recently, Kamei et al. [34] empirically studied the feasibility of change-level defect prediction in a cross-project context. In this work, we also examine the performance of the DBN-based semantic features on change-level cross-project defect prediction tasks.

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 the programs' semantic information, while our approach automatically learns the semantic features using a 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. [107] proposed an approach that leveraged deep

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2877612, IEEE Transactions on Software Engineering

learning to generate features from existing features and then used these new features to build defect prediction models. This work was motivated by the weaknesses of logistic regression (LR), which is that LR cannot combine features to generate new features. They used a DBN to generate features from 14 traditional change level features, including the following: 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 features related to developers' experience [107].

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

There also many existing studies that leverage deep learning techniques to address other problems in software engineering [16], [17], [30], [32], [44], [46], [60], [74], [84], [101], [106], [108]. Mou et al. [60] used deep learning to model programs and showed that deep learning can capture the programs' structural information. Deep learning has also been used for malware classification [74], [108], test report classification [32], link prediction in a developer online forum [106], software traceability [30], etc.

How to explain deep learning results is still a challenging question to the AI community. To interpret deep learning models, Andrej et al. [36] used character-level language models as an interpretable testbed to explain the representations and predictions of a Recurrent Neural Network (RNN). Their qualitative visualization experiments demonstrate that RNN models could learn powerful and often interpretable long-range interactions from real-world data. Radford et al. [79] focused on understanding the properties of representations learned by byte-level recurrent language models for sentiment analysis. Their work reveals that there exists a sentiment unit in the well-trained RNNs (for sentiment analysis) that has a direct influence on the generative process of the model. Specifically, simply fixing its value to be positive or negative can generate samples with the corresponding positive or negative sentiment. The above studies show that to some extent deep learning models are interpretable. However, these two studies focused on interpreting RNNs on text analysis. In this work we leverage a different deep learning model, i.e., the deep belief network (DBN), to analyze the ASTs of source code. The DBN adopts different architectures and learning processes from RNNs. For example, an RNN (e.g., LSTM) can, in principle, use its memory cells to remember long-range information that can be used to interpret data it is currently processing, while a DBN does not have such memory cells (details are provided in Section 2.3). Thus, it is unknown whether DBN models share the same property (i.e., interpretable) as RNNs.

Many studies used a topic model [5] to extract semantic features for different tasks in software engineering [7], [47], [69], [70], [89], [105]. Nguyen et al. [70] leveraged a 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, the generated features cannot capture structural information in a source file.

8 Conclusions and Future Work

This work leverages a representation-learning algorithm, i.e., deep learning, to learn semantic representation directly from source code for defect prediction. Specifically, we deploy a deep belief network to learn semantic features from programs' ASTs (for file-level defect prediction models) and source code changes (for change-level defect prediction models) automatically, and leverage the learned semantic features to build prediction models.

We examined the effectiveness of the learned DBN-based semantic features on two file-level defect prediction tasks, i.e., file-level within-project defect prediction (WPDP) and file-level cross-project defect prediction (CPDP), and two change-level defect prediction tasks, i.e., change-level within-project defect prediction (WCDP) and change-level cross-project defect prediction (CCDP). To conduct comprehensive performance evaluations, we employed both non-effort-aware and effort-aware evaluation metrics.

For file-level defect prediction tasks, our evaluations were conducted on 26 versions of data from 10 open source projects. Our results show that the DBN-based semantic features improve WPDP on average by 13.3 percentage points (in F1), and outperform the state-of-the-art CPDP with traditional features on average by 6.0 percentage points. For change-level defect prediction, our evaluations were conducted on more than 1M changes from six open source projects and four open-source commercial projects. The experimental results indicate that the DBN-based semantic features can improve WCDP on average by 5.1 percentage points, and improve the state-of-the-art CCDP technique with traditional change-level features, on average, by 2.9 percentage points. In addition, under the effort-aware evaluation scenario, our DBN-based semantic features can outperform traditional features for both the file-level and the change-level defect prediction.

In the future, we would like to extend our DBN-based approach to generate semantic features for method-level defect prediction, which helps in predicting the buggy methods in software projects. It could be promising to leverage the defect prediction result to facilitate other practices during software development and maintenance. For example, software defect prediction has been used by QA teams to help prioritize test cases [98], enhance static bug finders [81], etc. We plan to explore the potential applications of defect prediction for improving risk management, quality control, and project planning.

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.

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] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE'07*, pages 215–224.

[3] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *TSE'02*, 28(1):4–17.

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

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

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

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

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

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

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

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

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

[13] W. B. Frakes and R. Baeza-Yates. Information retrieval: data structures and algorithms. 1992.

[14] 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.

[15] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *MSR'11*, pages 83–92.

[16] X. Gu, H. Zhang, and S. Kim. Deep code search. In *ICSE'18*, pages 933–944.

[17] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *FSE'16*, pages 631–642.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD'09*, 11(1):10–18.

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

[20] J. Han and C. Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. *From Natural to Artificial Neural Computation*, pages 195–201, 1995.

[21] 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.

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

[23] H. He and E. A. Garcia. Learning from imbalanced data. *TKDE'09*, 21(9):1263–1284.

[24] 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.

[25] 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.

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

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

[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE'07*, pages 96–105.

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

[30] G. Jin, C. Jinghui, and C.-H. Jane. Semantically enhanced software traceability using deep learning techniques. In *ICSE'17*, pages 3–14.

[31] 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.

[32] W. Junjie, C. Qiang, W. Song, and W. Qing. Domain adaptation for test report classification in crowdsourced testing. In *ICSE'17*, pages 83–92.

[33] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *PROMISE'10*, page 9.

[34] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[35] 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'13*, 39(6):757–773.

[36] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. In *ICLR'16 Workshop*.

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

[38] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *TSE'08*, 34(2):181–196.

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

[40] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE'06*, pages 81–90.

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

[42] B. Kitchenham, E. Mendes, and G. H. Travassos. Cross versus within-company cost estimation studies: A systematic review. *TSE'07*, 33(5):316–329.

[43] 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.

[44] 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.

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

[46] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *ICSME'17*, pages 249–260.

[47] 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.

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

[49] R. Martin. Oo design quality metrics. *An analysis of dependencies*, 12:151–170, 1994.

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

[51] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *TSE'17*, pages 412–428.

[52] T. Mende and R. Koschke. Effort-aware defect prediction models. In *CSMR'10*, pages 107–116.

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

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

[55] 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.

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

[57] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[58] 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.

[59] 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.

[60] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI'16*, pages 1287–1293.

[61] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE'05*, pages 284–292.

[62] 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.

[63] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE'06*, pages 452–461.

[64] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *TSE'17*.

[65] J. Nam and S. Kim. CLAMI: Defect prediction on unlabeled datasets. In *ASE'15*, pages 452–463.

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

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

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

[69] 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.

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

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

[72] E. Paikari, M. M. Richter, and G. Ruhe. Defect prediction using case-based reasoning: An attribute weighting technique based upon sensitivity analysis in neural networks. *SEKE'12*.

[73] 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.

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

[75] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects t o assist code audits. In *CCS'15*, pages 426–437.

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

[77] L. Prechelt and A. Pepper. Why software repositories are not used for defect-insertion circumstanc e analysis more often: A case study. *IST'14*, 56(10):1377–1389.

[78] T.-S. Quah and M. M. T. Thwin. Application of neural networks for software quality prediction using object-oriented metrics. In *ICSM'03*, pages 116–125.

[79] A. Radford, R. Jozefowicz, and I. Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

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

[81] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing static bug finders and statistical prediction. In *ICSE'14*, pages 424–434.

[82] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *FSE'12*, pages 61:1–61:11.

[83] J. Ratzinger, M. Pinzger, and H. Gall. Eq-mine: Predicting short-term defects for software evolution. In *FASE'07*, pages 12–26.

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

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

[86] R. Sarikaya, G. E. Hinton, and A. Deoras. Application of deep belief networks for natural language understanding. *TASLP'14*, 22(4):778–784.

[87] F. Seide, G. Li, and D. Yu. Conversational speech transcription using context-dependent deep neural networks. In *INTERSPEECH'11*.

[88] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *FSE'05*, volume 30, pages 1–5.

[89] 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.

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

[91] C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *TSE'16*, 43:1–18.

[92] 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.

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

[94] 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.

[95] 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.

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

[97] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *ICSE'16*, pages 297–308.

[98] S. Wang, J. Nam, and L. Tan. QTEP: Qality-aware test case prioritization. In *FSE'17*, pages 523–534.

[99] S. Watanabe, H. Kaiya, and K. Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *PROMISE'08*, pages 19–24.

[100] 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.

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

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

[103] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang. Hydra: Massively compositional model for cross-project defect prediction. *TSE'16*, 42:977–998.

[104] X. Xia, D. Lo, X. Wang, and X. Yang. Collective personalized change classification with multiobjective search. *TR'16*, 65(4):1810–1829.

[105] 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.

[106] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *ASE'16*, pages 51–62.

[107] 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.

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

[109] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *MSR'14*, pages 182–191.

[110] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *ICSE'16*, pages 309–320.

[111] 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 *FSE'09*, pages 91–100.

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