Towards Balanced Defect Prediction with Better Information Propagation

Abstract

Defect prediction, the task of predicting the presence of defects in source code artifacts, has broad application in software development. Defect prediction faces two major challenges, *label scarcity*, where only a small percentage of code artifacts are labeled, and *data imbalance*, where the majority of labeled artifacts are non-defective. Moreover, current defect prediction methods ignore the impact of *information propagation* among code artifacts, and this negligence leads to performance degradation. In this paper, we propose DPCAG, a novel model to address the above three issues. We treat code artifacts as nodes in a graph, and learn to propagate influence among neighboring nodes iteratively in an EM framework. DPCAG dynamically adjusts the contributions of each node and selects high-confidence nodes for data augmentation. Experimental results on real-world benchmark datasets show that DPCAG improves performance compare to the state-of-the-art models. In particular, DPCAG achieves substantial performance superiority when measured by Matthews Correlation Coefficient (MCC), a metric that is widely acknowledged to be the most suitable for imbalanced data.

Introduction

The presence of undetected defects is a serious threat to software quality and security. For instance, the Heartbleed Bug was inadvertently introduced in OpenSSL in 2012 and remained unpatched for more than two years. Heartbleed can be exploited to steal confidential information in the SSL/TLS protocol, including passwords and session cookies. Consequently, a large number of users were forced to modify their passwords on popular web sites including Amazon Web Services and GitHub.

Defect prediction aims at building classifiers to predict whether code artifacts (e.g. classes, methods) are defective or not (Bowes, Hall, and Petric 2018). Many defect prediction models have been proposed over the years (Wang, Liu, and Tan 2016; Fan et al. 2019; Li et al. 2017; Qu et al. 2018). However, most of them do not effectively handle the following three main challenges that arise from the inherent characteristics of source code: (1) information propagation among code artifacts needs to be well represented; and (2) label information (i.e. defective or not) for code artifacts is lacking; and (3) the categories of data are severely imbalanced (i.e. non-defective artifacts are the vast majority).

Code artifacts communicate with each other via method invocation and message passing, i.e. *information propagation*. The information propagated among code artifacts may impact whether an artifact is defective or not. For instance, a class containing a defective method is defective, but a subclass that overrides this method may not be defective. If the information from this overridden method is not properly propagated, the subclass may be incorrectly classified as defective by the classifier. Information propagation among code artifacts is not well utilized by existing methods (Fan et al. 2019; Wang, Liu, and Tan 2016). Instead, these models treat each code artifact as a separate data instance and extract features of each instance to train a classifier. Some models such as Node2defect (Qu et al. 2018) treat individual classes as nodes, and utilize Deepwalk in the node2vec algorithm (Grover and Leskovec 2016) to propagate information among nodes. However, none of them specifically distinguishes the information propagation between different code artifacts. Hence, a mechanism to precisely describes the information propagation needs to be designed.

Label scarcity refers to the absence of labeled artifacts within source code. For instance, in three of the ELFF datasets (Shippey et al. 2016), DrJava, Genoviz and Jmol, the proportions of code artifacts with label information are 4.53%, 5.57% and 7.18%, respectively. Other datasets in ELFF also suffer from the label scarcity issue to a similar degree. The lack of labeled data makes it more difficult to train a defect prediction model with high accuracy. Recent works such as DP-CNN (Li et al. 2017) use Convolutional Neural Network (CNN) to extract features from the Abstract Syntax Tree (AST) and train the classifier. Our experimental results show that the performance of DP-CNN in larger datasets is worse than in smaller datasets, as larger datasets are more susceptible to label scarcity. Therefore, mechanisms need to be investigated to address the label scarcity challenge.

Class imbalance occurs when the vast majority of labeled code artifacts are non-defective and it is a wide-spread phenomenon. For instance, the proportions of defective code artifacts among all labeled code artifacts are 1.79%, 3.60% and 5.13% respectively in the three ELFF datasets mention above. If no measure is taken to handle the imbalance be-

tween the two categories (i.e. defective and non-defective), models will prefer to label code with the majority category during the classification process. Over-sampling is one of the common methods to handle this challenge. Models such as NSGLP (Zhang, Jing, and Wang 2017) use Laplacian Sampling and expand the minor category to solve this challenge. These approaches bring up two new problems: (1) classes and methods of the same category are not distinguished; and (2) additional noise may be introduced. However, different classes and methods have different features, and the classifier may misjudge their labels, leading to poor performance, while noise will further degrade performance. Hence, a re-weighting method that adjusts weights for each class and method may be more suitable.

In this paper, we propose a semi-supervised method, named Defect Prediction on Code Artifact Graph (DPCAG), for defect prediction task. We construct a Code Artifact Graph and pre-train the embeddings of the code artifacts as features. We further propose a novel propagation control mechanism to automatically learn an information propagation matrix to indicate the intensity of information propagation. Finally, we train a classifier in an Expectation-Maximization (EM) framework (Dempster, Laird, and Rubin 1977).

A key component of DPCAG is the EM-based classifier. In the E-step, we combine embeddings with the propagation matrix to propagate on a GNN model to estimate the label distribution and update the embeddings. During this stage, a data augmentation mechanism is used to expand the training set to handle label scarcity. In the M-step, we propagate labels instead of embeddings on another GNN model, and maximize the probability of the label distribution. We employ a dynamic weighting loss function to overcome the class imbalance.

Our contributions can be summarized as follows.

- We propose a novel semi-supervised method, named DPCAG, utilize the information of code artifacts and the structure information between code artifacts for the defect prediction task while effectively addresses the three main challenges described above.
- We employ a propagation matrix to adequately capture the information transfer between code artifacts, and propose a data augmentation mechanism to add high-similarity samples to the training set while overcoming the label scarcity, and we also design a loss function which dynamically changes the weights of code artifacts to alleviate the effect of class imbalance issue.
- Our method outperforms state-of-the-art models in the Defect Prediction task through an extensive empirical study. It is particularly noteworthy that our method achieves substantial superiority when measured by Matthews Correlation Coefficient (MCC), a metric that has been widely recognized as the appropriate metric for performance measurement on imbalanced data.

Related Work

Defect Prediction is a task to predict whether code artifacts are defective (Bowes, Hall, and Petric 2018). Most Defect Prediction models focus on extracting better features from code artifacts. They use statistics of code artifacts and process metrics as features (Nagappan and Ball 2005; Bibi et al. 2006) to train the classifiers. Recent embedding-based models prefer to use embeddings of Abstract Syntax Trees (ASTs) as features while ASTs contain syntax and semantic information. For instance, DP-CNN (Li et al. 2017) encodes the token vectors from ASTs and generates features via CNN. DBN-CP (Wang, Liu, and Tan 2016) utilizes Deep Belief Networks to extract semantic features from ASTs to train the classifiers. DP-ARNN (Fan et al. 2019) utilizes word embeddings from word2vec (Mikolov et al. 2013) and employs attention mechanisms to generate features. However, these state-of-the-art models simply ignore the class imbalanced and information propagation.

Several models attempt to use propagate information among code artifacts. For example, node2defect (Qu et al. 2018) uses the class dependency graph to encode the classes in code, and uses node2vec (Grover and Leskovec 2016) to randomly propagate to generate the embeddings of classes as features. However, these methods construct graphs in which adjacent nodes are of the same type (e.g. classes), which limits their applicability to data that contains different types of code artifacts (e.g. both classes and methods).

Other studies focus on class imbalance issues. For instance, ORB (Cabral et al. 2019) proposed an over-sampling mechanism to handle the class imbalance issue by scoring different categories according to the bias and re-sample according to the score. NSGLP (Zhang, Jing, and Wang 2017) rates the artifacts by Laplacian Sampling and expands the minor class to address class imbalance. However, these models are highly dependent on the quality of sampling and sampling may introduce noises.

Graph Convolution Network (GCN) has been shown to (1) graph structure is used to represent the relationship among code artifacts (Atzeni and Atzori 2017), and (2) our model modify to describe information propagation process. GCN (Kipf and Welling 2017) is one of the Graph Neural Network (GNN) architectures that classifies nodes in graphs. The convolution operation in GCN computes the Laplacian matrix for nodes in the graph and then multiplies the representations of the nodes. Related models such as GraphSAGE (Hamilton, Ying, and Leskovec 2017) sample from neighborhood nodes and aggregate the information to predict the label of unlabeled nodes. GAT (Velickovic et al. 2018) use the attention mechanism to learn the coefficient weights between nodes in the propagation step. GMNN (Qu, Bengio, and Tang 2019) uses conditional random fields to model the joint distribution of target labels and utilizes the EM algorithm for model optimization. However, none of these models addresses the class imbalanced problem, which is a key issue in the Defect Prediction task.

Problem Definition

We use parser (Atzeni and Atzori 2017) to convert the source code of a given project to a graph $G = (V, E)$, where V is the set of nodes and represents the code artifacts, i.e. classes and methods, and E is the set of edges that represents the

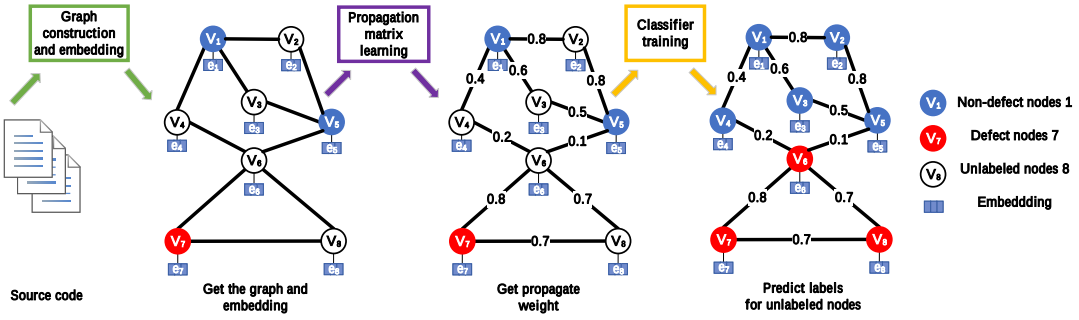

Figure 1: The overall framework of DPCAG.

relationships between nodes, such as inheritance relationship, association relationship, and invocation relationship. Nodes can be separated into two disjoint subsets: labeled nodes V_l and unlabeled nodes V_u, such that $V = V_l \cup V_u$. Furthermore, labeled nodes can also be separated into two sub-categories: non-defect nodes and defect nodes, that is, $V_l = V_{neg} \cup V_{pos}$.

Each node $v_i \in V$ can be represented by its d-dimensional embedding $\mathbf{e}_i \in \mathbb{R}^d$. To simplify the notation, we use matrix $\mathbf{X} \in \mathbb{R}^{|V| \times d}$ to represent the embedding of all nodes in G.

Assuming a particular and consistent ordering of nodes in sets V, V_l and V_u, we define $p(\mathbf{Y}), p(\mathbf{Y}_l), p(\mathbf{Y}_u)$ to be the label distributions of these nodes respectively.

We treat the defect prediction task cast as a binary classification task. Our goal is to predict the label of every unlabeled node. Hence, the defect prediction task is reduced to maximize the likelihood of the labels of unlabeled nodes.

Formally, given the Graph $G = (V, E)$ and the node embedding $\mathbf{X}$, our goal is to learn a defect prediction model parameterized by θ to maximize the log likelihood of label distribution of unlabeled nodes, i.e., $\theta^* = \arg\max_\theta \log p(\mathbf{Y}_u|\mathbf{Y}_l, \mathbf{X}, G; \theta)$. For convenience, we use $\log p_\theta(\mathbf{Y}_u|\mathbf{Y}_l, \mathbf{X}, G)$ to represent $\log p(\mathbf{Y}_u|\mathbf{Y}_l, \mathbf{X}, G; \theta)$

Method

The overall framework of our method is shown in Figure 1. Our method can be divided into three main steps.

1. **Graph construction and embedding.** To utilize the dependency relationships in the source code, we convert the source code into a graph G as described above, and compute the embeddings $\mathbf{X}$ of nodes V to facilitate the training of the model.
2. **Propagation matrix learning.** To utilize propagate information among nodes in the graph G, we use labels $\mathbf{Y}_l$ to construct and optimize the propagation matrix $\mathbf{A}^*$, the elements in which indicate the propagation intensity.
3. **Classifier training.** To classify nodes V_u (i.e. learn the label distribution of V_u), we combine label information $\mathbf{Y}_l$ with the propagation matrix $\mathbf{A}^*$ and features $\mathbf{X}$ to train an EM-based classifier while employing data augmentation and a dynamic weighted loss function to overcome the label scarcity and class imbalance issues.

Graph construction & embedding

Given a project, we parse its source code by using CodeOntology (Atzeni and Atzori 2017) and extract triples in Resource Description Framework, and construct the graph G. Finally, we employ a graph representation learning method to pre-train the embeddings $\mathbf{X}$ of nodes V.

Propagation matrix learning

As we mentioned before, information propagation among code artifacts has an impact on whether an artifact is defective. Thus, ignoring information propagation may lead to sub-optimal performance of the classifier. For instance, in Figure 1, node v_6 may be a defect node as it utilizes the information of defect node v_7. As shown in the graph in the middle, the strength of information propagation for the edge between nodes v_6 and v_7 is 0.8, indicating the strong influence of v_7 has on v_6.

Propagation matrices used in current methods are all exclusively based on graph structure information, such as adjacency matrix and Laplacian matrix (Kipf and Welling 2017; Qu, Bengio, and Tang 2019). These matrices do not utilize feature information or label information and may affect prediction performance. Hence, we use the graph structure information to construct a propagation matrix, and then combine label information to automatically optimize it.

Formally, for the propagation matrix $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$, we define the initial value of each element $\mathbf{A}_{ij}$ as:

$$\mathbf{A}_{ij} = \begin{cases} 0, & v_i \text{ is not connected with } v_j \\ \frac{1}{\sqrt{d(v_i) \times d(v_j)}}, & v_i, v_j \in V_u \\ \frac{a}{\sqrt{d(v_i) \times d(v_j)}}, & v_i \in V_{neg} \text{ or } v_j \in V_{neg} \\ \frac{b}{\sqrt{d(v_i) \times d(v_j)}}, & v_i \in V_{pos} \text{ or } v_j \in V_{pos} \end{cases} \quad (1)$$

where $d(i)$ represents the degree of node v_i in graph G, and a, b are hyperparameters.

After initializing matrix $\mathbf{A}$, we use LPA (Zhu and Ghahramani 2002) to optimize it automatically.

$$\mathbf{A}^* = \arg\min_A \mathcal{L}_{lpa}(\mathbf{A}) = \arg\min_A \sum_{v_i \in V_l} J(\hat{y}_i^{lpa}, y_i) \quad (2)$$

where the $J()$ is the cross-entropy loss and $\hat{y}_i^{lpa}$ is the label predicted by LPA.

As a result, $\mathbf{A}^*$ contains both graph structure information and label information, and we use $\mathbf{A}^*$ in the classifier to control the propagation of information.

The EM-based Defect Prediction Model

In the defect prediction task, defects could occur internally within an artifact and at the interface between artifacts. Thus, whether a node is defective is determined by its features and the labels from nodes which have dependency relationships with it. However, current methods do not effectively utilize these dependency relationships as the labels of the majority of nodes $\mathbf{Y}_u$ are unknown.

To make full use of features from each node and labels from their neighboring nodes, we propose an EM-based semi-supervised classifier by considering $\mathbf{Y}_u$ as hidden variables. In the Expectation step, we use the node embeddings $\mathbf{X}$ to learn the distribution $\log q_\theta$ to obtain labels of unlabeled nodes. In the Maximization step, we use labels of neighboring nodes and the dependency relations $\mathbf{A}^*$ to learn distribution $\log p_\theta$ and maximize the expectation $\mathbb{E}_{\log q_\theta} \log p_\theta$.

Specifically, in our classifier, we optimize the evidence lower bound (ELBO) of log-likelihood instead of optimizing the log-likelihood directly, as follows.

$$\log p_\theta(\mathbf{Y}_u|\mathbf{Y}_l, \mathbf{X}, \mathbf{A}^*) \geq \mathbb{E}_{\log q_\theta(\mathbf{Y}_l|\mathbf{X},\mathbf{A}^*)}[\log p_\theta(\mathbf{Y}|\mathbf{Y}_l, \mathbf{X}, \mathbf{A}^*) - \log q_\theta(\mathbf{Y}_l|\mathbf{X}, \mathbf{A}^*)] \quad (3)$$

The main steps of the classifier are summarized in Algorithm 1. In the E-step, we fix $\log p_\theta$ and update $\log q_\theta$. We use dynamic weighted loss and data augmentation to address the class imbalance and label scarcity issues respectively. In the M-step, we fix $\log q_\theta$ and update $\log q_\theta$, and again use the dynamic weighted loss to address the class imbalance issue. We will show the detail of E-step M-step in the next sub-section and the last sub-section.

Algorithm 1: training DPCAG classifier

Input:
node embedding $\mathbf{X}$; propagation matrix $\mathbf{A}^*$;
The labels of labeled nodes $\mathbf{Y}_l$
Output:
The label distribution q_θ

```
while not converged do
    (E-step):
        Use X and A* to estimate a distribution q_θ.
        Use dynamic weighted loss to optimize q_θ, let
        q_θ approach p_θ and update X (addressing class
        imbalance)
        Use X for data augmentation to expand Y_l
        (address label scarcity)
    (M-step):
        Get label distribution Y by using q_θ and X
        Use Y, Y_l and A* to update distribution p_θ
        Use dynamic weighted loss to optimize p_θ and
        maximize E_{q_θ} p_θ
end
```

Expectation step

In the Expectation step, our goal is to estimate the label distribution $\log q_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{A}^*)$. We employ a two-layer GNN to predict the label distribution $\log q_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{A}^*)$ while updating $\mathbf{X}, \theta$ and expanding the training set. Specifically, the Expectation step has the following three main steps.

1. Learning the distribution. It has been shown that the representations of nodes (i.e. $\mathbf{X}$) are influenced by their neighboring nodes (Qu, Bengio, and Tang 2019). Hence, we employ a GNN to propagate influence from neighboring nodes, through the propagation matrix $\mathbf{A}^*$. Specifically, we use a two-layers GNN, named GNNq, to update node embeddings from neighboring nodes and learn the distribution $\log q_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{A}^*)$. Nodes in GNNq are node embeddings $\mathbf{X}$, and edge weights are given by the propagation matrix $\mathbf{A}^*$.

$$\log q_\theta(\mathbf{Y}^{t+1}|\mathbf{X}^{t+1}, \mathbf{A}^*) = \sigma'(\mathbf{A}^*\sigma(\mathbf{A}^*\mathbf{X}^t\mathbf{W}_{q1})\mathbf{W}_{q2}), \quad (4)$$

where $\mathbf{W}_{q1} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{q2} \in \mathbb{R}^{h\times 2}$ are the weight matrices of the first layer and second layer of GNNq respectively, where h denotes the dimension, σ and σ' are the activation functions $ReLU$ and $softmax$, and $\mathbf{X}^t$ denotes the t-th iteration of $\mathbf{X}$.

Every row in $\log q_\theta(\mathbf{Y}^{t+1}|\mathbf{X}^{t+1}, \mathbf{A}^*)$, denoted $\mathbf{pred}_i \in \mathbb{R}^2$, represents a two-dimensional array for node v_i, where two values represent the predicted probability of whether the node is defective or not.

2. Optimization by dynamic weighted loss function. We design the dynamic weighted loss function to alleviate the impact of class imbalance. Specifically, we use $\mathbf{pred}_i \in \mathbb{R}^2$ to represent the probability of $v_i \in V_l$ from $\log q_\theta(\mathbf{Y}^{t+1}|\mathbf{X}^{t+1}, \mathbf{A}^*)$, and $\mathbf{real}_i \in \mathbb{R}^2$ represents the probability of $v_i \in V_l$ in $\log p_\theta(\mathbf{Y}|\mathbf{Y}_l, \mathbf{X}, \mathbf{A}^*)$ where $\mathbf{Y}^t$ denotes the t-th iteration of $\mathbf{Y}$. Let $\mathbf{gap}_i = \mathbf{pred}_i - \mathbf{real}_i$, the loss function can be defined as follows, where $\circ$ is element-wise product:

$$\begin{aligned}\mathcal{L} &= L(\log q_\theta(\mathbf{Y}^{t+1}|\mathbf{X}^{t+1}, \mathbf{A}^*, \log p_\theta(\mathbf{Y}|\mathbf{Y}_l, \mathbf{X}, \mathbf{A}^*) \\ &= \sum_{v_i \in V_l} ||-\log(\mathbf{pred}_i) \circ \mathbf{real}_i \circ (1 + \mathbf{gap}_i)||_1. \end{aligned} \quad (5)$$

We multiply $-\log(\mathbf{pred}_i)$ in the loss function as it can dynamically adjust the contribution of v_i to the loss function. We multiply $1+(\mathbf{gap}_i)$ as it amplifies the penalty when the classifier misclassifies. In contrast, the loss is almost unchanged when v_i is correctly classified.

3. Data Augmentation. After computing the loss in the previous step by backpropagation, we use the updated embedding $\mathbf{X}^{t+1}$ to select highly reliable nodes and add them to the training set. For each unlabeled node $v \in V_u$, let $s' = \max_{v' \in V_{neg}} Sim(\mathbf{e}_v, \mathbf{e}_{v'})$ be the highest non-defect confidence score, and let $s'' = \max_{v' \in V_{pos}} Sim(\mathbf{e}_v, \mathbf{e}_{v'})$ be the highest defect confidence score. Given a pre-defined threshold t, node v is added to V_{neg} if $s' > max(s'', t)$. Otherwise, node v is added to V_{pos} if $s'' > max(s', t)$.

In our model, we use Cosine Similarity as the similarity function. The expanded training set is then used for the subsequent training process.

Maximization step

In the Maximization steps, we fix the distribution $\log p_\theta$, and use the label information of neighboring nodes to learn the distribution $\log p_\theta$ and maximizing $\mathbb{E}_{\log q_\theta} \log p_\theta$. Similar to node embedding, the label of each node in V is also influenced by the labels of its neighboring nodes (Qu, Bengio, and Tang 2019). Hence, we employ another GNN, named GNNp, to propagate this information. The nodes in GNNp represent updated node labels and edge weights are given by the propagation matrix $\mathbf{A}^*$. For convenience, we use $\log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}^t, \mathbf{A}^*)$ instead of $\log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}_l^t, \mathbf{X}^{t+1}, \mathbf{A}^*)$ because $\mathbf{X}$ is an irrelevant variable to $\log p_\theta$ and $\mathbf{Y}_l^t \approx \mathbf{Y}^t$ before propagation where $\mathbf{Y}^t$ denotes the t-th iteration of $\mathbf{Y}$.

$$\log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}^t, \mathbf{A}^*) = \sigma'(\mathbf{A}^*\sigma(\mathbf{A}^*\mathbf{Y}^t\mathbf{W}_{p1})\mathbf{W}_{p2}), \quad (6)$$

where $\mathbf{W}_{p1} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{p2} \in \mathbb{R}^{h\times 2}$ are the weight matrices of the first layer and second layer of GNNq, hyperparameter h denotes the dimension, $\sigma()$ and $\sigma'()$ are the activation functions $ReLU$ and $softmax$.

Similar to the loss function in the Expectation step, let $\mathbf{pred}_i \in \mathbb{R}^2$ represent the probability of $v_i \in V_l$ in $\log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}^t, \mathbf{A}^*)$, $\mathbf{real}_i \in \mathbb{R}^2$ represent the probability of $v_i \in V_l$ in $\log q_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{A}^*)$, and $\mathbf{gap}_i = \mathbf{pred}_i - \mathbf{real}_i$. We directly calculate the distance between the two distributions instead of calculate their KL divergence $\mathbb{E}_{q_\theta(\mathbf{Y}|\mathbf{X},\mathbf{A}^*)}[\log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}^t, \mathbf{A}^*)]$:

$$\begin{aligned}\mathcal{L} &= L(\mathbb{E}_{q_\theta(\mathbf{Y}|\mathbf{X},\mathbf{A}^*)}[\log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}^t, \mathbf{A}^*)]) \\ &= L(\log q_\theta(\mathbf{Y}|\mathbf{X}, \mathbf{A}^*), \log p_\theta(\mathbf{Y}^{t+1}|\mathbf{Y}^t, \mathbf{A}^*)) \\ &= \sum_{v_i\in V_l} ||-\log(\mathbf{pred}_i) \circ \mathbf{real}_i \circ (1 + \mathbf{gap}_i)||_1\end{aligned} \quad (7)$$

In this way, we have used embedding $\mathbf{X}$, label $\mathbf{Y}$ and propagation matrix $\mathbf{A}^*$ in one iteration. The training process stops when the loss of E-step and M-step stops decreasing and we regard $\log q_\theta$ as the final label distribution.

Experiments

Datasets and metrics

We use three datasets from ELFF (Shippey et al. 2016), namely DrJava, Genoviz and Jmol, to evaluate model performance. Among all datasets in ELFF, DrJava is the largest datasets; Jmol is the most commonly used small-scale dataset; and Genoviz is also a commonly used dataset which it is between DrJava and Jmol in terms of size. Their brief statistics information is shown in Table 1. It can be clearly observed that all three datasets exhibit significant issues of label scarcity and class imbalance. Among all nodes, labeled nodes account for 4.53%, 5.57% and 7.12% in the three datasets respectively. Furthermore, defect nodes make up only 1.8%, 3.6% and 4.9% of labeled nodes respectively. For each dataset, the set of labeled nodes is divided into training, validation and test sets with the ratio of 90-5-5.

We use Matthews Correlation Coefficient (MCC) as the main evaluation metric, along with accuracy (ACC), F1-score. It has been widely acknowledged in the literature that MCC is a more appropriate metric for measuring classification performance on imbalanced datasets (Boughorbel, Jarray, and El-Anbari 2017; Song, Guo, and Shepperd 2019; Shepperd, Bowes, and Hall 2014), making it especially suitable for defect prediction. MCC is computed from the four elements in the confusion matrix, namely true positive (tp), true negative (tn), false positive (fp), and false negative (fn).

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}. \quad (8)$$

Baselines

We compare our model to a number of state-of-the-art defect prediction models: an influential traditional model DBN-CP (Wang, Liu, and Tan 2016), the CNN-based model DP-CNN (Li et al. 2017), the Transformer-based model DP-ARNN (Fan et al. 2019), and the network embedding-based model Node2defect (Qu et al. 2018). Furthermore, we evaluate our model against Graph Markov Neural Network (GMNN) (Qu, Bengio, and Tang 2019). GMNN is a state-of-the-art semi-supervised classification model that combines Markov networks and graph neural networks.

Note that DBN-CP, DP-CNN and DP-ARNN are not end-to-end models, and an additional classifiers needs to be employed to make defect prediction on test data. We employ Random Forest (RF) and Logistic Regression (LR) for these models.

Experimental settings

For our model, the learning rate lr is set to 0.006 and the dropout rate p set to 0.5. The hyperparameter a is set to 25 for all three datasets. The hyperparameter b is set to 1300 in DrJava and 2000 in both Genoviz and Jmol. The dimension of hidden layer is $h = 20$. We use RMSProp (Tieleman and Hinton 2012) as the optimizer. The confidence threshold for adding nodes to the training set is $t = 0.8$. In every iteration, the epoch for the Expectation step and the Maximization step is set to 200. We select the values of hyperparameters according to the result of validation set.

Results

The main results of the defect prediction task are shown in Table 2. A number of important observations can be made.

1. In general, our model outperforms all the state-of-the-art baselines on all three datasets in all but one case (ACC on Jmol), especially prominently in MCC and F1.
2. More importantly, the most substantial performance disparity can be observed in the values of MCC, where MCC is more faithfully to measure a model's performance on imbalanced data. Compared to the second-best method for each dataset, our model outperforms them by 48.2%, 44.6%, and 24.6% respectively. These results clearly demonstrate the superiority of our model, especially on imbalanced and large datasets.

Dataset	Nodes	Edges	Labeled Nodes	Defect Nodes	Training	Validation	Test
DrJava	255,512	1,281,752	11,575 (4.5%)	208 (1.8%)	10,417	579	579
Genoviz	81,591	355,133	4,548 (5.6%)	164 (3.6%)	4,093	228	227
Jmol	32,792	134,303	2,355 (7.2%)	115 (4.9%)	2,119	118	118

Table 1: Brief statistics of the three datasets. Percentage values of labeled nodes are measured against all nodes, and the percentage values of defect nodes are measured against labeled nodes.

3. In comparison, accuracy values for different models are all high and close to each other, reflecting ACC's insensitivity to the class imbalance issue. In other words, a model only needs to classify unlabeled nodes as the majority class (non-defect) to achieve high accuracy values. For nodes that need to be classified as defect nodes, our model outperforms the second-best method for each dataset by 46.1%, 16.7%, and 9.1%. This shows that our model can correctly classify the defect nodes and help us find the code with defects in real tasks.

In conclusion, these observations clearly demonstrate that our model is effective in dealing with issues of label scarcity and class imbalance while making full use of information propagation in the defect prediction task. In the next section, we further analyze the effect of the main components on model performance.

Model Analysis

In this section, we present ablation studies and additional analyses of our model. DrJava is chosen as the dataset for analysis as it is the largest dataset in ELFF, and also the one with the most label scarcity and class imbalance issues among the three. Therefore, it can reflect the performance of our model most appropriately.

Why does DPDAG use EM FrameWork

In our model, we use the EM-based classifier to utilize embeddings $\mathbf{X}$ and labels $\mathbf{Y}$ of the nodes and their neighborhood. We examine the necessity of using the EM framework by building a supervised classifier with embeddings $\mathbf{X}$ as features, which have been propagated, and with node labels $\mathbf{Y}_l$ as ground-truth. In other words, GNNq in the Expectation step is used to build this classifier with all the hypermeters kept the same.

The comparison shows that our model significantly outperforms the GNNq-based classifier by 19.2%, 8.8%, and 7.6% on MCC in the three datasets. It demonstrates that our EM-based classifier can effectively utilize label information from neighboring nodes.

Analysis of information propagation

We design this analysis to verify the impact of the propagation matrix on model performance. The propagation matrix can be regarded as the key to the propagation process on GNNs, where the elements of these matrices represent the intensity of information propagation between nodes.

In this experiment, we compare our automatically optimized propagation matrix $\mathbf{A}^*$, defined in Formula (2) with three different propagation matrices: adjacency matrix $\mathbf{AD} \in \mathbb{R}^{|V|\times|V|}$, propagation matrix of GMNN (Qu, Bengio, and Tang 2019) $\mathbf{GM} \in \mathbb{R}^{|V|\times|V|}$, and a propagation matrix $\mathbf{A}$ fixed at the initial values as defined in Formula (1). For $v_i, v_j \in V$, if v_i is not connected with v_j, $\mathbf{AD}_{ij} = \mathbf{GM}_{ij} = 0$; otherwise $\mathbf{AD}_{ij} = 1$ and $\mathbf{GM}_{ij} = \frac{1}{\sqrt{d(v_i)\times d(v_j)}}$, where $d(v_i)$ is the degree of v_i.

The result is presented in Table 3. It shows that our propagation matrix has the best performance. The performance difference can be attributed to the following main reasons. (1) The adjacency matrix $\mathbf{AD}$ only contains coarse-grained information about whether the nodes in the graph are related. (2) The propagation matrix of GMNN $\mathbf{GM}$ adds degree information of nodes to the adjacency matrix, but ignores the different impact of different categories of nodes (unlabeled, defect or non-defect). (3) Propagation matrix $\mathbf{A}$ considers different categories of nodes based on $\mathbf{GM}$ but fixes their influences. (4) In contrast, our automatically optimized propagation matrix $\mathbf{A}^*$ makes full use of label information to fine-tune the matrix to further improve performance.

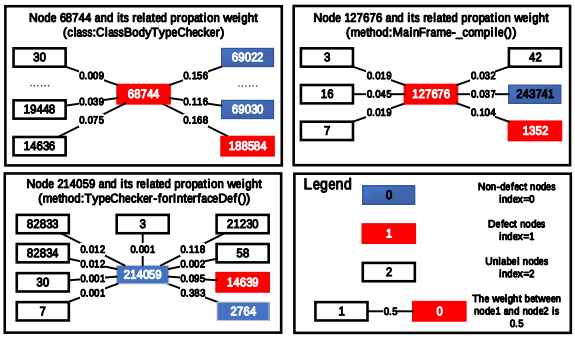

Figure 2: Three examples of correct classification by distinguishing different weight

To further demonstrate our case, we present three representative nodes in DrJava in Figure 2, which shows predictions of the center nodes by color-coding and the edge weights as learned by our propagation matrix A^*. In all three cases, the label of the center node is correctly predicted by our method and misclassified by other methods (such as GMNN). As we can see, each center node is connected with the same-label node with the highest edge weight.

Analysis of label scarcity

To demonstrate data augmentation's effectiveness in overcoming the label scarcity issue, we experiment with the node

Model	DrJava			Genoviz			Jmol		
Metrics	MCC	F1	ACC	MCC	F1	ACC	MCC	F1	ACC
DBN-CP (LR)	0.075	0.091	0.965	0.162	0.182	0.960	0.324	0.353	0.907
DBN-CP (RF)	0.476	0.375	0.982	0.395	0.400	0.973	0.584	0.533	0.940
DP-CNN (LR)	0.196	0.200	0.972	0.278	0.242	0.889	0.493	0.533	0.881
DP-CNN (RF)	0.388	0.266	0.981	0.403	0.285	0.977	0.584	0.533	**0.941**
DP-ARNN (LR)	0.163	0.174	0.934	0.212	0.173	0.832	0.445	0.485	0.855
DP-ARNN (RF)	0.389	0.267	0.981	0.572	0.500	0.982	0.526	0.533	0.923
Node2defect	0.363	0.333	0.979	0.461	0.444	0.977	0.598	0.636	0.932
GMNN	0.431	0.444	0.974	0.403	0.285	0.978	0.377	0.421	0.906
Ours	**0.958**	**0.857**	**0.993**	**0.907**	**0.667**	**0.997**	**0.844**	**0.695**	0.940

Table 2: Results of the defect prediction task. Best results are **bolded** and second best results are underlined

Propagation matrix	MCC	F1	ACC
Adjacency Matrix **AD**	0.102	0.125	0.951
GMNN **GM**	0.355	0.370	0.941
Fixed **A**	0.810	0.814	0.991
Ours $\mathbf{A}^*$	0.958	0.857	0.993

Table 3: Performance on different propagation matrices.

similarity threshold t to control the number of nodes that are added to the training set. When t decreases, the number of nodes added to the training set increases.

Training data	MCC	F1	ACC
Original training set (100%)	0.918	0.758	0.987
t=0.9 (117.56%)	0.920	0.814	0.991
t=0.8 (129.08%)	0.958	0.857	0.993
t=0.7 (153.18%)	0.957	0.799	0.989

Table 4: Effect of the data augmentation.

The result is presented in Table 4, which shows that, as t decreases, model performance first increases and then decreases. The main reason is that during the propagation process, every node in the training set spreads information to adjacent nodes as a source. The more sources of propagation, the more information each node convolves to adjacent nodes through GNNs, and the more accurate the classification result is. However, adding too many nodes may introduce noise and cause performance degradation. As a result, we chose $t = 0.8$ as the final parameter value.

Analysis of class imbalance

In this experiment, we examine whether our loss function can effectively alleviate the impact of categories imbalance. We compare our loss function, defined in Formulas (5) and (7) with three other loss functions: Cross-Entropy loss (CEL) , Mean square error (MSE) and the loss function of GMNN (Qu, Bengio, and Tang 2019), which maximizes the sum of the probability that each node is correctly classified.

We define the *imbalance rate* of a node as the number of nodes in the majority category in its neighboring nodes divided by the number of nodes in the minority category.

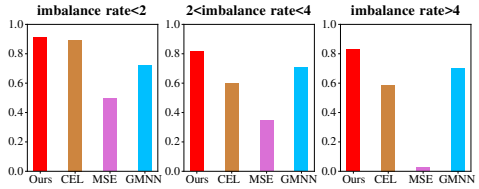

Figure 3: The MCC performance of loss functions on different imbalance rates.

The result is presented in Figure 3. It shows that as the imbalance rate increases from < 2 to > 4, our loss function consistently and increasingly outperforms the other loss functions. Compared with other loss functions, our loss function can dynamically adjust the contribution weight for loss and embedding of each node, which helps the model to overcome the impact of class imbalance.

Conclusion

In this paper, we proposed a novel semi-supervised model named DPCAG for the defect prediction task. DPCAG combined the propagation control mechanism, the data augmentation mechanism and the dynamic weighted loss to solve three challenges: (1) properly represent the information propagation among code artifacts; and (2) lacking label information; and (3) the imbalance of categories of data. Experimental results on the DrJava, Genoviz and Jmol showed that three mechanisms in DPCAG achieved significant performance boost compare with state-of-the-art methods.

For future work, we will use the semantic information in Resource Description Framework triples to further optimize the propagation matrix to improve the performance in the Defect Prediction task. We will try other methods to deal with class imbalance issue.

References

Atzeni, M.; and Atzori, M. 2017. CodeOntology: RDF-ization of Source Code. In d'Amato, C.; Fernández, M.; Tamma, V. A. M.; Lécué, F.; Cudré-Mauroux, P.; Sequeda, J. F.; Lange, C.; and Heflin, J., eds., *Proceedings on International Semantic Web Conference*, volume 10588 of *Lecture Notes in Computer Science*, 20–28. Springer.

Bibi, S.; Tsoumakas, G.; Stamelos, I.; and Vlahavas, I. P. 2006. Software Defect Prediction Using Regression via Classification. In *Proceedings of 2006 IEEE/ACS International Conference on Computer Systems and Applications*, 330–336. IEEE Computer Society.

Boughorbel, S.; Jarray, F.; and El-Anbari, M. 2017. Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric. *Public Library of Science* 12(6): e0177678.

Bowes, D.; Hall, T.; and Petric, J. 2018. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal* 26(2): 525–552.

Cabral, G. G.; Minku, L. L.; Shihab, E.; and Mujahid, S. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In Atlee, J. M.; Bultan, T.; and Whittle, J., eds., *Proceedings of the 41st International Conference on Software Engineering*, 666–676. IEEE / ACM.

Dempster, A. P.; Laird, N. M.; and Rubin, D. B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39(1): 1–22.

Fan, G.; Diao, X.; Yu, H.; Yang, K.; and Chen, L. 2019. Software Defect Prediction via Attention-Based Recurrent Neural Network. *Scientific Programming* 2019: 6230953:1–6230953:14.

Grover, A.; and Leskovec, J. 2016. node2vec: Scalable Feature Learning for Networks. In Krishnapuram, B.; Shah, M.; Smola, A. J.; Aggarwal, C. C.; Shen, D.; and Rastogi, R., eds., *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 855–864. ACM.

Hamilton, W. L.; Ying, Z.; and Leskovec, J. 2017. Inductive Representation Learning on Large Graphs. In Guyon, I.; von Luxburg, U.; Bengio, S.; Wallach, H. M.; Fergus, R.; Vishwanathan, S. V. N.; and Garnett, R., eds., *Proceedings on Annual Conference on Neural Information Processing Systems 2017*, 1024–1034.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings on International Conference on Learning Representations*. OpenReview.net.

Li, J.; He, P.; Zhu, J.; and Lyu, M. R. 2017. Software Defect Prediction via Convolutional Neural Network. In *Proceedings of 2017 IEEE International Conference on Software Quality, Reliability and Security*, 318–328. IEEE.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient Estimation of Word Representations in Vector Space. In Bengio, Y.; and LeCun, Y., eds., *Proceedings of 1st International Conference on Learning Representations*.

Nagappan, N.; and Ball, T. 2005. Use of relative code churn measures to predict system defect density. In Roman, G.; Griswold, W. G.; and Nuseibeh, B., eds., *Proceedings of 27th International Conference on Software Engineering*, 284–292. ACM.

Qu, M.; Bengio, Y.; and Tang, J. 2019. GMNN: Graph Markov Neural Networks. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 5241–5250. PMLR.

Qu, Y.; Liu, T.; Chi, J.; Jin, Y.; Cui, D.; He, A.; and Zheng, Q. 2018. node2defect: using network embedding to improve software defect prediction. In Huchard, M.; Kästner, C.; and Fraser, G., eds., *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 844–849. ACM.

Shepperd, M. J.; Bowes, D.; and Hall, T. 2014. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions on Software Engineering* 40(6): 603–616.

Shippey, T.; Hall, T.; Counsell, S.; and Bowes, D. 2016. So You Need More Method Level Datasets for Your Software Defect Prediction?: Voilà! In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 12:1–12:6. ACM.

Song, Q.; Guo, Y.; and Shepperd, M. J. 2019. A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering* 45(12): 1253–1269.

Tieleman, T.; and Hinton, G. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4(2): 26–31.

Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *Proceedings of 6th International Conference on Learning Representations*. OpenReview.net.

Wang, S.; Liu, T.; and Tan, L. 2016. Automatically learning semantic features for defect prediction. In Dillon, L. K.; Visser, W.; and Williams, L., eds., *Proceedings of the 38th International Conference on Software Engineering*, 297–308. ACM.

Zhang, Z.; Jing, X.; and Wang, T. 2017. Label propagation based semi-supervised learning for software defect prediction. *Automated Software Engineering* 24(1): 47–69.

Zhu, X.; and Ghahramani, Z. 2002. Learning from Labeled and Unlabeled Data with Label Propagation. Technical report, School of Computer Science, Carnegie Mellon University.