Review of Automatically Learning Semantic Features for Defect Prediction

Felix Schober
Technical University of Munich
Mail: felix.schober@tum.de

Abstract—**As software projects get bigger and more complex, companies and institutions need to focus their testing efforts on parts of the software that are more likely to contain defects. Software defect prediction tries to automatically predict classes that contain code which does not behave in an expected way. Wang et al. propose a new approach for software defect prediction that utilizes abstract syntax trees and deep belief networks. They achieve an average F1 score of 64.1% for "within-project defect prediction" and 56.8% for "cross-project defect prediction". Both results outperform traditional features and algorithms.**

I. Introduction

According to a study performed by the university of Cambridge, the annual global cost caused by software defects is estimated to be around 312 billion dollars [5]. Software defects increase development and maintenance costs and decrease customer satisfaction [15].

To minimize these negative effects it is crucial to eliminate software defects as soon as possible especially since bugs or defects are harder and more time consuming to fix at a later point in the development process or after the product is released [3].

A report of the Microsoft Corporation observes around 10 - 20 software defects per 1000 lines of code during development and testing and 0.5 defects per 1000 lines of code after a product is released [18]. Since software projects get bigger and more complex the number of built-in software defects rises, which can cause software to fail. Depending on the scenario such failures can cause inconvenience (failure in a consumer product) or even lead to severe safety issues (failure in a power station or plane). Nevertheless, exhaustive software testing is time- and resource-consuming and thus not an optimal strategy to spread testing effort over all parts of the software since software defects are usually not uniformly distributed [14]. Boehm and Papaccio showed that only approximately 20% of the modules account for 80% of the software defects [4]. Accordingly, it is reasonable to focus testing efforts on modules or classes that are more likely to contain software defects.

In previous studies, software defect prediction was done using handcrafted features such as Halstead features, McCabe features or code metrics like lines of code or polymorphism factor [24]. Classifiers like Support Vector Machines, Decision Trees or Neural Networks were then used to decide whether a piece of code contained a defect or not [24, 25].

Recently, Mou et al. proposed a new approach for software defect prediction using abstract syntax trees in combination with convolutional neural networks [21].

Wang et al. pursue a slightly different approach. They try to predict software defects in Java open source projects by using abstract syntax trees which are transformed by a deep belief network to reduce the dimension. Then, those feature vectors are the input to an alternating decision tree classifier which does the final prediction.

This paper reviews Wang et al.'s approach and is organized as follows: in Section II the technical and machine learning background is briefly introduced. Section III then puts the machine learning technique into context and describes the approach proposed by Wang et al.. Section IV presents the results which are discussed in Section V. Section VI gives a short overview of relevant papers citing Wang et al.. Finally, section VII draws a conclusion and offers suggestions for future research.

II. Background

A. Abstract Syntax Trees

Abstract Syntax Trees (ASTs) are used to capture essential semantic information of a program in the form of a tree. Each node in the tree represents an operation in the source code like condition statements ("if"), operators (">") or assignments (variable assignment). ASTs are language-specific and are created through hand written parsers in a bottom-up fashion [13]. Figure 1 shows an AST of the code in listing 1.

Listing 1: Pseudocode for an AST example

```
a=6
b=8
c=(a+b)*3
```

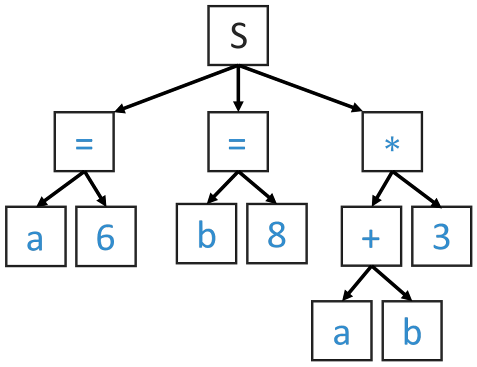

Figure 1: Figure of an abstract syntax tree. Representation of pseudo code from listing 1. Source: Own representation

B. Restricted Boltzmann Machines

1) General Principles: A Restricted Boltzmann Machine (RBM) is a shallow, two layer neural network with the goal to reconstruct a data pattern with a high probability. It consists of stochastic binary nodes that are partitioned into two sets: "visible nodes" and "hidden nodes". These nodes are connected to other nodes with bidirectional connections. To make learning easier there cannot be connections between two nodes of the same set. However, all visible nodes are connected to all hidden nodes. Furthermore, the connections can be strengthened or weakened by a weight for each connection. Weights can take on arbitrary positive or negative real values [1]. Each node has a binary state that either be on (state 1) or off (state 0). Figure 2 depicts a RBM with three visible nodes (green nodes) and two hidden nodes (blue nodes).

Visible nodes represent the real world input (for example in case of an image, each node would represent one pixel [12] or in case of defect prediction one AST-token). Hidden nodes then try to abstract the real world input. They are named "hidden" because the state of a hidden node cannot be directly observed through the input. However, the probability for a certain state of a hidden node can be determined by a forward propagation of the visible node states. Given an input-vector v, the probability that a binary hidden node h_j is turned-on $p(h_j = 1)$ is calculated by the following equation

$$p(h_j = 1|v) = \sigma(b_j + \sum_i v_i * w_{ij}) \tag{1}$$

where $\sigma(x)$ is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$ (or another binary threshold function), w_{ij} is the weight-term of the connection between the visible node v_i and the hidden node h_j. b_j is a bias-term that is always added to node h_j regardless of the input [12]. Equivalently, the probability for an active visible node v_i, given the set of hidden nodes h, is

$$p(v_j = 1|h) = \sigma(a_i + \sum_j h_j * w_{ij}) \tag{2}$$

with a_i as the bias-term of the visible node v_i [9].

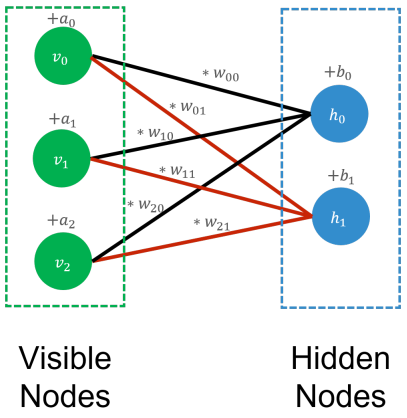

Figure 2: Figure of a RBM with three visible- and two hidden nodes. States of the hidden nodes are calculated by multiplying input states from the visible nodes with the connection weights for h_0 (depicted as black connections) or by the weights for h_1 (red connections). Source: Own representation

2) RBM Training: RBMs can be trained supervised, like a normal neural network with backpropagation and labels for the training inputs, or unsupervised using alternating Gibbs-Sampling [1] and Contrastive Divergence (CD) [9, 6, 17].

Unsupervised learning is done by continuously repeating two steps:

1) First, the probability for node activations in the hidden layer, given some data vector v, is calculated. Nodes are turned-on with the probability $p(h_j = 1|v)$ or turned-off with the probability $1 - p(h_j = 1|v)$ (equation 1). The hidden nodes now have a stochastic binary activation state which is derived from the data vector [9] (hidden nodes are either on or off).
2) Those hidden binary unit activations are then used as the input vector in the "reconstruction-step". The new states of the visible units v'_i are calculated by applying equation 2. This step is called "reconstruction" or "dreaming" because the network tries to reconstruct the input v' it would like to see given the hidden input vector h. The difference between the real world input v from step 1 and the reconstructed vector v' is called "reconstruction error".
3) Finally, the real-valued probabilities from the reconstruction vector v' calculated in step 2 are used to calculate the activations of the hidden nodes once again.

4) Step 2 and 3 are repeated until learning eventually stops due to the nature of weight adjustments described below ($v \approx v'$).

To actually learn, a RBM adjusts the weights between the two layers. The weight change is then

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_i \rangle_{dream}) \quad (3)$$
$$\Delta a_i = \epsilon(v_i - \langle v_i h_j \rangle_{data}) \quad (4)$$
$$\Delta b_j = \epsilon(b_j - \langle v_i h_j \rangle_{data}) \quad (5)$$

where Δw_{ij} is the weight change for w_{ij} and ϵ is the learning rate. $\langle v_i h_j \rangle_{data}$ is number of times the input node v_i and the hidden node h_j were on together in the data-step where v_i represents actual data. Similarly, $\langle v_i h_i \rangle_{dream}$ shows the number of times v_i and h_j were on together during the "reconstruction"/"dreaming" phase. The biases a_i and b_j are updated in a similar fashion.

This process is infinitely repeated until $v \approx v'$. Since $\langle v_i h_j \rangle_{data}$ is subtracted by $\langle v_i h_i \rangle_{dream}$ this means that the learning will automatically stop when the model eventually reproduces the original input ($v \approx v'$).

Figure 3 visualizes the Gibbs-Sampling process.

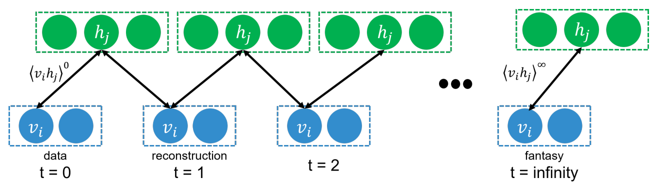

Figure 3: Figure of a RBM learning with Gibbs-Sampling. Source: Own representation after [9]

This process can be shortened considerably by using CD introduced by Hinton in 2002 [10]. Instead of repeating Gibbs-Sampling an infinite number of times with a random data vector, CD only uses a finite number of Gibbs-Sampling steps. To compensate for the reduced number of Gibbs-Sampling steps, a data input has to be chosen so that it is already very close to the mean of the sample data distribution.

Although this is only a rough approximation of the log likelihood gradient of the training data, this method works well enough for most applications and is much faster [12, 9, 17].

C. *Deep Belief Networks*

The main problem of shallow models like RBMs is that functions might need exponentially more nodes in a k layer architecture than in a $k+1$ layer architecture [2]. This means that for some problems deep networks are more efficient than shallow models.

Deep Belief Networks (DBNs) invented by Hinton, Osindero and Teh in 2006 combine the concept of deep architectures with RBMs [11]. They form a composition of multiple stacked RBMs. Figure 4 shows the general architecture of a DBN with stacked RBMs. DBNs can be trained very similarly to RBMs:

1) Learning begins by training the first two bottom layers v and h_1 of the DBN on a training sample v as a RBM using CD.
2) Activations of the hidden units h_1 from the trained RBM using v are computed. Then, h_1 is used as the input to train the next RBM layer h_2.
3) This process continues until the last RBM is reached.
4) (Optional) Weights can be fine-tuned by training the DBN supervised using backpropagation starting from the top layer h_l.

The advantage of this architecture and training procedure is that layers can be trained greedily layer by layer in an unsupervised, as well as supervised fashion [11].

This allows a DBN to be applied in many different scenarios. It is therefore possible to pretrain a DBN with large amounts of unlabeled data and then use the last layer as a feature detector for another classifier or neural network [16].

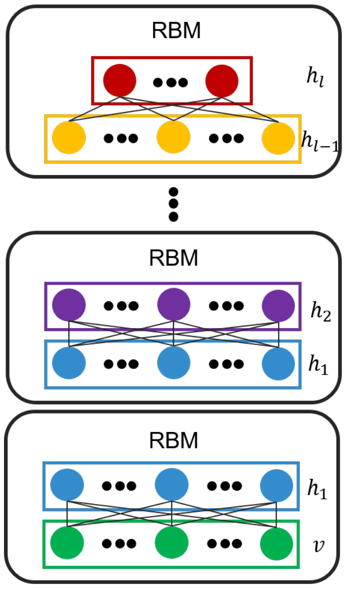

Figure 4: Figure of a DBN. A DBN consists of multiple stacks of RBMs. The bottom layer v represents the input whereas the topmost layer can be used as a feature vector or a soft-max classification layer [12, 16]. Source: Own representation after [16]

III. Defect Prediction using DBNs

Wang et al. propose a novel way to predict defects in Java source code. As described in section I they use Abstract Syntax Trees, DBNs and different classifiers to predict software defects.

A. *Parsing*

First, they parse Java source code to generate one AST for each class of a Java project. Next, they extract three types of syntax tree nodes that might be relevant for the defect prediction (shown below) and exclude all others:

- Method invocations & Class instance creations
- Method declarations & Type declarations & Enum declarations
- Control-flow nodes:
  - While statements
  - If statements
  - Catch clauses
  - Throw statements

B. Mapping

In the next step this list of AST-nodes is then mapped to numerical features, so that each unique node is assigned a unique identifier. Since methods and types are often project specific they treat those nodes as generic "method declarations" and "method invocations" when conducting cross-project defect prediction. During within-project prediction method declarations and invocations are assigned a unique token depending on their name/type, so that e.g. `foo()` has a different identifier than `bar()`.

Unfortunately, defect data sets often contain noise [24], so before the data can be used, the authors remove incorrect labeling with an adapted Closest List Noise Identification (CLNI) that is based on the "Edit distance" instead of the "Euclidean distance". In addition, they also remove tokens that occur less than three times throughout the project. This has proven to be an effective way to reduce complexity.

Finally, since most neural networks require input scaled within the interval $[0, 1]$ they apply min-max normalization.

C. Semantic DBN Features

Wang et al. use the mapped, cleaned and transformed semantic AST-features to train a DBN unsupervised.

For within-project defect prediction the DBN is trained with an older version of the test source project as the training dataset.

Instead of using an older version of the same project as the training input for the DBN during cross-project defect prediction, they use the source code of a different project as a training dataset.

After the training of the DBN is done, the DBN is used to generate features for both the training and test dataset.

D. Defect Prediction Classifier

After the data is preprocessed and the DBN feature vectors are generated for each class the final defect prediction is then done by an Alternating Decision Tree (ADTree) which is a generalized decision tree developed by Freund and Mason in 1999 [7].

E. DBN Parameter Tuning

Wang et al. tune the following three parameters of their DBN to increase the F1 score:

- Number of layers
- Number of nodes in each layer
- Number of iterations

They conduct experiments with different values for their parameters and test them on five different projects. The parameter value with the best average F1 score is then chosen for the evaluation. They achieve the best results with ten hidden layers and 100 nodes in each layer, so that they generate a 100 dimensional feature vector. To keep training within reasonable limits (15 seconds), they set their number of training iterations to 200.

IV. Discussion of Results

The authors answer four research questions:

1) Can DBN features outperform traditional features?
2) Can DBN features outperform traditional features with different classification algorithms?
3) Can DBN features outperform traditional features for cross-project defect prediction?
4) What is the time and space cost for DBN feature based defect prediction?

Wang et al. test their hypothesis on ten open source Java projects. They obtain the information about the number of defects from the PROMISE defect repository [20] which specifies the number of defects in a class along other, more traditional source code metrics.

A. Within Project Defect Prediction

To answer the first research question, they test the performance of their proposed semantic DBN features for within-project defect prediction against 20 traditional features from the PROMISE dataset [20, 14], as well as the AST features without the DBN using the same ADTree classifier as a baseline.

Table I shows five out of the 16 tests that were performed. They achieved an average F1-Score of 64.1% for their DBN features, whereas the baselines only achieved a F1-score of 49.9% for the traditional features and 44.7% for the AST features [24]. There is only one instance where traditional features outperformed the proposed approach (Camel 1.4 -> Camel 1.6).

Source -> Target			DBN	PROMISE	AST
ant	1.5	1.6	<u>91.4</u>	47.7	45.3
ant	1.6	1.7	<u>94.2</u>	54.2	47.0
camel	1.4	1.6	37.4	<u>39.1</u>	38.3
log4j	1.0	1.1	<u>70.1</u>	58.7	45.5
xerces	1.2	1.3	<u>41.1</u>	23.8	23.6
average over all experiments			<u>64.1</u>	49.9	44.7

Table I: Results of within-project defect prediction for five different projects. Best results are underlined. Source: Wang et al. [24]

B. Classifier Performance

The second question is tested by performing experiments using Naïve Bayes and Logistic Regression for both traditional features and DBN features. DBN features outperform traditional features with both classifiers. The results of this experiment show that the increased performance of the DBN features is not tied to a specific classifier [24].

C. Cross Project Defect Prediction

To test cross-project defect prediction state of the art technique TCA+ introduced by Nam et al. [22] was used as a baseline algorithm. According to the authors of TCA+, cross-project defect prediction is significantly harder than within-project defect prediction. Most machine learning algorithms assume that there is a similar data distribution between the training and test dataset which is true for within-project defect prediction but not for cross-project defect prediction. However, similar to the results in the previous section, the F1 score of the proposed DBN features outperformed TCA+ on average by 8.9%. The total average F1 score over all tests was 56.8% [24].

D. Time and Space Cost

Finally, the question is tested if DBN features are feasible in practice, or if computational costs are too high.

For this they take the time and memory costs of the DBN feature generation. The time it takes to parse the source code is not considered.

The time to generate the features varies from 8 seconds to 32 seconds and the memory cost is less than 6.5MB [24].

V. Assessment of Results

A. F1 Prediction Variance

There is a considerable variation of the results for within-project defect prediction (Standard deviation (SD) 15.5%) and for cross-project defect prediction (SD 11.5%). There are instances for which the approach works well (Apache Ant: 94.2%) but also projects were the classifier has low predictive power (Apache Ivy: 35.0%) [24]. In case of the Ivy project, the classification method is barely better than an average random classifier, that has a F1 score of 28.3% or a classifier which always predicts that the class contains a bug 32.8%.

B. Parameter Tuning

A possible explanation for the variance of the defect prediction might be the parameter tuning described in Section III-E. Wang et al. try to find a model that fits every project, although the projects differ in the number of available training samples and input dimensions. A model with fewer layers/nodes might be better for a project with a small amount of classes because it has fewer trainable parameters. Koru and Liu support the assumption that prediction models might need to change from one project to another since defect patterns are different across projects [15].

Additionally, they do not mention if they tuned hyper-parameters like learning rate or mini batch size which can have a significant impact on performance [9].

C. Time Cost

Furthermore, the authors state that the proposed approach would be applicable in practice because the feature generation and classification is reasonably fast. However, they do not take parsing and noise cleaning into account. For large projects the parsing of source code into Abstract Syntax Trees can take a long time (up to five minutes[1]) for a project with 1090 classes (Ant 1.6 & Ant 1.7).

D. AST Feature Generation

As described in Section III-A the proposed AST features only consist of nine different statements. They contain no information about nesting, calculations or variables. Listing 2 and Listing 3 show code examples that are semantically different but the AST feature vector would be the same for both examples.

Listing 2: Code A

```
# Complicated
# calculation
a = 42
b = 'a'
if a == b:
    foo(a)
    bar(b)
```

Listing 3: Code B

```
a = b = 10
if True:
    foo()
bar()
return -1
```

If Code A would contain a defect the DBN would not be able to detect this defect since it does not know that there is a difference between the defect in code A and code B, which works within expected parameters.

VI. Cited by Analysis

The paper by Wang et al. has a current citation count of four, which shows interest in the field of defect prediction given that the paper is very recent. Two of the papers focus on implementing software defect prediction themselves [23, 26]. Mens focuses on the maintenance of large software ecosystems [19] and Gu et al. use a Recurrent Neural Network Encoder-Decoder to generate API usage sequences [8]. All of these papers, however, only briefly mention the work done by Wang et al. as an example for recent innovations in the field of defect prediction.

[1]Parsing was done on a Intel Core i5-6500 @ 3.20GHz

VII. Conclusion

Wang et al. propose a new approach for defect prediction which significantly outperforms existing defect prediction models. They show that AST features in combination with deep belief networks have a high predictive power compared to traditional features. In addition, their approach is not only able to work on different versions of the same project but also across different projects which is important for the cold-start problem.

For within-project defect prediction they achieve an average F1 score of 64.1% while the baselines accomplish an average F1 score of 49.9% and 44.7%. Although cross-project defect prediction is notably harder they still beat the current state of the art baseline TCA+. They accomplish an average F1 score 56.8% while TCA+ achieves a mean F1 score of 47.9%.

Future work could focus on how convolutional neural networks in combination with abstract syntax tree proposed by Mou et al. [21] compare to deep belief networks on the same dataset. Additionally, it should be tested how data augmentation can help to increase classification accuracy for within and cross-project defect prediction.

References

[1] D. Ackley, G. Hinton, and T. Sejnowski. "A learning algorithm for boltzmann machines." In: *Cognitive Science* 9.1 (1985), pp. 147–169.

[2] Y. Bengio. "Learning Deep Architectures for AI." In: *Foundations and Trends® in Machine Learning* 2.1 (2009), pp. 1–127.

[3] B. W. Boehm and P. N. Papaccio. "Understanding and Controlling Software Costs." In: *IEEE Transactions on Software Engineering* 14.10 (1988), pp. 1462–1477.

[4] B. W. Boehm and P. N. Papaccio. "Understanding and Controlling Software Costs." In: *IEEE Transactions on Software Engineering* 14.10 (1988), pp. 1462–1477.

[5] F. Brady. *Financial Content: Cambridge University study states software bugs cost economy $312 billion per year | CJBS Insight*. 2013.

[6] M. a. Carreira-Perpiñán and G. E. Hinton. "On Contrastive Divergence Learning." In: *Artificial Intelligence and Statistics* 10 (2005), p. 17.

[7] Freund, Yoav and L. Mason. "The alternating decision tree learning algorithm." In: *International Conference on Machine Learning* 99 (1999), pp. 124–133.

[8] X. Gu, H. Zhang, D. Zhang, and S. Kim. "Deep API Learning." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 631–642.

[9] G. Hinton. "A Practical Guide to Training Restricted Boltzmann Machines A Practical Guide to Training Restricted Boltzmann Machines." In: *Computer* 9.3 (2010), p. 1.

[10] G. E. Hinton. "Training products of experts by minimizing contrastive divergence." In: *Neural Computation* 14.8 (2002), pp. 1771–1800.

[11] G. E. Hinton, S. Osindero, and Y.-W. Teh. "A Fast Learning Algorithm for Deep Belief Nets." In: *Neural Computation* 18.7 (2006), pp. 1527–1554.

[12] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks." In: *Science* 313.5786 (2006), pp. 504–507.

[13] J. Jones. "Abstract syntax tree implementation idioms." In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)* (2003), pp. 1–10.

[14] M. Jureczko and L. Madeyski. "Towards identifying software project clusters with regard to defect prediction." In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10* (2010), p. 1.

[15] a. G. Koru and H. Liu. "Building Effective diction Models in Practice." In: *IEEE Software* 22.6 (2005), pp. 23–29.

[16] P. Lin, S.-W. Fu, S.-S. Wang, Y.-H. Lai, and Y. Tsao. "Maximum Entropy Learning with Deep Belief Networks." en. In: *Entropy* 18.7 (July 2016), p. 251.

[17] X. Ma and X. Wang. "Average Contrastive Divergence for Training Restricted Boltzmann Machines." en. In: *Entropy* 18.2 (Jan. 2016), p. 35.

[18] S. C. McConnell. *Code Complete*. 2nd Editio. Vol. 136. 1. Redmond: Microsoft Press, 2011, p. 952.

[19] T. Mens. "An Ecosystemic and Socio-Technical View on Software Maintenance and Evolution." In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 1–8.

[20] T. Menzies, R. Krishna, and D. Pryor. *The Promise Repository of Empirical Software Engineering Data*. 2015.

[21] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. "Convolutional neural networks over tree structures for programming language processing." In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI '16)* (2016), pp. 1287–1293.

[22] J. Nam, S. J. Pan, and S. Kim. "Transfer defect learning." In: *Proceedings - International Conference on Software Engineering* (2013), pp. 382–391.

[23] R. Queiroz, T. Berger, and K. Czarnecki. "Towards predicting feature defects in software product lines." In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development - FOSD 2016* (2016), pp. 58–62.

[24] S. Wang, T. Liu, and L. Tan. "Automatically learning semantic features for defect prediction." In: *Proceedings - International Conference on Software Engineering* 14-22-May (2016), pp. 297–308.

[25] W. E. Wong and Y. Qi. “Bp neural network-based effective fault localization.” In: *International Journal of Software Engineering and Knowledge Engineering* 19.4 (2009), pp. 573–597.

[26] F. Wu, X.-Y. Jing, X. Dong, J. Cao, B. Xu, and S. Ying. “Cost-Sensitive Local Collaborative Representation for Software Defect Prediction.” In: *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE. 2016, pp. 102–107.