Middle East Technical University
Informatics Institute

SEMANTIC METHODS IN SOFTWARE DEFECT PREDICTION TECHNIQUES

Advisor Name: Altan Koçyiğit
(METU)

Student Name: Şükrücan Taylan Işıkoğlu
(SM)

January 2024

TECHNICAL REPORT
METU/II-TR-2024-

Orta Doğu Teknik Üniversitesi
Enformatik Enstitüsü

YAZILIM HATASI TAHMİNİ TEKNİKLERİNDE ANLAMSAL YÖNTEMLER

Danışman Adı: Altan Koçyiğit
(ODTÜ)

Öğrenci Adı: Şükrücan Taylan Işıkoğlu
(SM)

Ocak 2024

TEKNİK RAPOR
ODTÜ/II-TR-2024-

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Internal Use)	2. REPORT DATE
3. TITLE AND SUBTITLE **SEMANTIC METHODS IN SOFTWARE DEFECT PREDICTION TECHNIQUES**	
4. AUTHOR (S) Ş. Taylan Işıkoğlu	**5. REPORT NUMBER (Internal Use)** **METU/II-TR-2024-**
6. SPONSORING/ MONITORING AGENCY NAME(S) AND SIGNATURE(S) Software Management Master's Programme, Department of Information Systems, Informatics Institute, METU Advisor: Altan Koçyiğit	Signature:
7. SUPPLEMENTARY NOTES	

8. ABSTRACT (MAXIMUM 200 WORDS)

The traditional methods in software defect prediction use software metrics that are collected from the source code. However these methods have an important shortcoming: it is possible that two source code segments, where one is buggy and one is not, have the same software metrics. Software metrics are not descriptive enough to discern defective code. Recently semantic methods have been explored. These methods use the source code directly and extract semantic information using methods that involve deep learning. This research presents a survey of the use of semantic methods in software defect prediction.

9. SUBJECT TERMS	10. NUMBER OF PAGES
software defect prediction, semantic, semantic methods, deep learning	**17**

TABLE OF CONTENTS

1. INTRODUCTION
2. BACKGROUND INFORMATION
    1. Software Defect Prediction
    2. Abstract Syntax Tree
    3. Long Short-Term Memory
    4. Deep Belief Network
    5. Word Embedding
    6. Attention
    7. Graph Representation Learning
3. RELATED WORK
4. METHODOLOGY
5. SOFTWARE DEFECT PREDICTION METHODS
6. CONCLUSION
7. REFERENCES

LIST OF TABLES

1. Table of Search Results
2. Table of Studies

LIST OF FIGURES

1. Non-Defective Code
2. Defective Code
3. An LSTM Cell
4. Graph Representation of AST

LIST OF ACRONYMS

- SDP: Software Defect Prediction
- CPDP: Cross Project Defect Prediction
- WPDP: Within Project Defect Prediction
- LSTM: Long Short-Term Memory
- DBN: Deep Belief Network
- AST: Abstract Syntax Tree
- VCS: Version Control System
- CBOW: Continuous Bag of Words
- LR: Logistic Regression
- RF: Random Forest
- CNN: Convolutional Neural Network
- RNN: Recurrent Neural Network
- NLP: Natural Language Processing

1. INTRODUCTION

Software Defect Prediction(SDP) is a way to predict potential bugs in the software. Traditional methods try to achieve this by using software metrics such as Halstead Complexity Measures, object oriented measures like inheritance depth and coupling, McCabe Complexity Metrics etc.[1]. These metrics provide a quantitative measure of software complexity. Software complexity is a good indicator of how prone the particular software is to defects. Therefore they can be used as features for machine learning methods to train and predict software defects; methods such as Support Vector Machine(SVM), Naive Bayes, Random Forest etc. are trained using software metrics obtained from buggy and non-buggy code. The resulting model is used with metrics from other pieces of software to predict potential bugs in them.

```
struct stack_t
{
    // ...
};

typedef struct stack_t stack_t;

void push_to_stack(stack_t *s, int val);
void pop_from_stack(stack_t *s, int *val);

int main()
{
    stack_t s;
    for (int i = 0; i < 10; ++i) {
        int val = 0;
        push_to_stack(&s, i);
        // ...
        pop_from_stack(&s, &val);
    }
    return 0;
}
```

Fig. 1 Non-Defective Code

```
struct stack_t
{
    // ...
};

typedef struct stack_t stack_t;

void push_to_stack(stack_t *s, int val);
void pop_from_stack(stack_t *s, int *val);

int main()
{
    stack_t s;
    for (int i = 0; i < 10; ++i) {
        int val = 0;
        pop_from_stack(&s, &val);
        // ...
        push_to_stack(&s, i);
    }
    return 0;
}
```

Fig. 2 Defective Code

However these kinds of methods have an important shortcoming; two different pieces of code, one of them defective and the other one being clean, can have same software metrics. For example in the code in Fig. 1 pushes to the stack and pops from it at the end of the loop. It is a defect to pop from the stack when it is empty, like it is done in Fig. 2. However both codes exhibit the same software metrics in terms of complexity. When non-buggy code and buggy code have the same metrics, it becomes impossible to predict defects using traditional methods that use software metrics only. To overcome such shortcoming, one possible way is to look at semantic information that is inside the software code. Such methods can learn complicated relations between the statements in the code itself and predict bugs more accurately. This is usually achieved by creating an Abstract Syntax Tree(AST) representation of the source code and training deep learning models on them using buggy and non-buggy code to learn how to predict defects in software.

This project aims to explore recent works that employ semantic methods for SDP. It will do so by making a survey of most cited research papers that have been released in the last 5 years. The structure of this survey is as follows: section 2 presents the basic topics that are used in the surveyed methods. Section 3 goes over similar surveys done in the area of semantic methods in software defect prediction. Section 4 presents the methodology used in creating this survey. Section 5 provides the survey of methods. Section 6 provides a conclusion to the work.

2. BACKGROUND INFORMATION

In this section, some of the important concepts used in the semantic SDP methods are explained.

2.1. Software Defect Prediction

To predict buggy software modules from non-buggy software modules, data is collected. Then modules are labeled for training purposes. A machine learning algorithm is trained with this data to predict whether a software module contains a bug or not. There are two categories of software defect prediction depending on the data it is used to train the algorithm and the test data that is used to predict an outcome. If the data used to train the algorithm and the data that is used to test the algorithm come from the same software project it is called Within Project Defect Prediction(WPDP). If the data used for training purposes is from different project then the data used to make predictions, it is called Cross Project Defect Prediction(CPDP)[1]. The performance of the algorithm highly depends on the kind of prediction done and the research done differentiates between these two categories.

2.2. Abstract Syntax Tree

Abstract Syntax Tree(AST) is a tree data structure that encapsulates the syntactic information of the source code. Source code is made of tokens and each token has a specific type, like function call, variable declaration etc. The tokenized source code is converted into a tree structure which represents each token with a node containing its data and its type. The relationship between the nodes and the type of the nodes can then be used to extract semantic information about the source code.

2.3. Long Short-Term Memory

Long Short-Term Memory(LSTM)[2] is a type of Recurrent Neural Network(RNN). A typical cell can be seen in Fig. 1. Conventional RNN's suffer from vanishing and exploding gradient issues due to the backpropagation. LSTM networks solve this problem by introducing three gates: input, output and forget gates. These gates control the flow of error propagation and ensure a constant error flow. It uses the input of current time step and previous time step to control the gates and update the cell state. LSTM networks excel at working with long sequences of data.

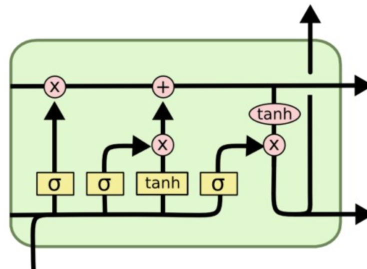

Fig. 3 An LSTM Cell

2.4. Deep Belief Network

Deep Belief Network(DBN)[3] is a type of unsupervised neural network architecture based on layers of Restricted Boltzmann Machines(RBM). They are generative models, which mean they model the data as a probability distribution. It consists of an input layer and several hidden layers. Each hidden layer is used to model the joint probability distribution between input and consequent layers.

2.5. Word Embedding

Word embedding is a way to represent words as numerical vectors. These numerical vectors are placed in vector space using their semantic meanings. This means that words which are similar semantically are placed closer in the vector space. A widely used method is word2vec[4]. Word embedding is used to convert text into a representation that neural networks can work with.

2.6. Attention

Attention[5] is a mechanism used in modelling long sequential data in neural networks. Traditional sequential processing favors recent information over older information. To keep the older context relevant for longer, attention mechanism is introduced. An attention mechanism consists of Query(Q), Value(V) and Key(K) layers. Output is calculated by calculating a weight from Q and K vectors, which is then multiplied by the V vector to get the output. The output is weighted with consideration to importance, this helps filtering unimportant data out and highlighting important data, much like giving an attention to specific subject.

2.7. Graph Representation Learning

Graphs are structures that consist of nodes and connections. They are used for representing relationships between objects. For example AST can be represented by a graph like in Fig. 2. Graph Representation Learning[6] is a method used in machine learning where the input consists of graphs. The algorithm learns features that represent the structure of the graph and can perform various tasks such as node classification, graph classification and community detection.

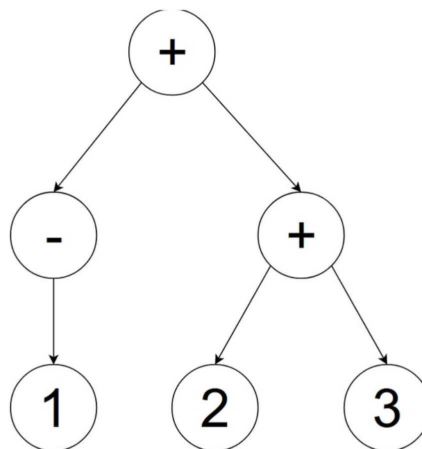

Fig. 4 Graph Representation of AST

3. RELATED WORKS

Akimova et al.[7] provide a survey of SDP methods for automatic feature extraction that uses deep learning. They explored the deep learning methods used in SDP and they explored the common challenges deep learning methods face. They found a lack of large datasets that are labelled. They suggest the use of large language models that are trained on source code using self-supervised methods to alleviate this problem. Another issue they found is the imbalance of classes in labelled data. Due to the nature of SDP, there are much more non-defective code than there are defective code. They suggest the use of oversampling methods like SMOTE to overcome the imbalance. Finally they consider the lack of context searching in deep learning methods. Long term dependencies can be important to finding defects in source code and to overcome this problem they suggest transformer architecture. Lastly, they introduced the trends in using deep learning for SDP. They found large language models which are aimed at Natural Language Processing(NLP) tasks to be gaining traction in newer studies. Omri et al.[8] provide an overview of deep learning in SDP. First they go over traditional methods in two sections; CPDP and WPDP. Finally they go over latest SDP techniques that use deep learning. Abdu et al.[9] explore SDP methods that use semantic features obtained from the source code. They explore main motivations behind using semantic features in SDP and introduce deep learning techniques used to extract semantic features from source code. They comparatively examine the methods based on their performance. They point out that source code can have differences which causes bugs, while the metrics used in traditional methods cannot adequately capture the differences enough to predict the defects. They suggest using semantic methods which can capture the difference in semantic information between two pieces of code better. They go over available datasets, both labelled and unlabelled which can be used for supervised training tasks aswell as large language models. They compare the performance of deep learning techniques with existing SDP models. The following work will summarize the most impactful studies in the last 5 years, some of which have not been covered before, as well as several works that came out in 2023.

4. METHODOLOGY

This work aims to explore semantic methods, which use the semantic information inside the source code to predict defects in software. For this work done in between years 2018-2023 are considered only. The following queries were used to search for related works:

- “software defect prediction” AND “semantic”
- “software fault prediction” AND “semantic”

The databases used are IEEE Xplore, Hindawi, Google Scholar, ACM Digital Library, MDPI, PLOS ONE, Springer Link.

Table 1. Table of Research Results

Database	Number of Results
IEEE Xplore	116
Hindawi	257
ACM Digital Library	74
MDPI	41
PLOS ONE	209
Springer Link	354

The related works were sorted by their citation counts. Studies that were not using a semantic method to predict software defects were excluded. Works with at least 20 citations are included in the survey. This methodology however, favors older works in the literature since they will be getting more citations. To include more recent works, this work also includes works with highest citations from 2023 that are below 20 citations. A total of 16 studies are included in this survey.

5. SOFTWARE DEFECT PREDICTION METHODS

Table 2. Table of Studies

Author	Methodology	Highlights
Wang et al.[10]	DBN	One of the earliest works to introduce semantic method to SDP.
Liang et al.[11]	LSTM	Enhanced the capability of LSTM using the word embedding CBOW.
Dam et al.[12]	Tree-Based LSTM	Improved the performance by utilizing the tree structure of the AST.
Fan et al.[13]	Bidirectional LSTM	Utilized Bidirectional LSTM with an attention layer to perform SDP.
Chen et al.[14]	Deep Transfer Learning for Defect Prediction(DTL-DP)	Novel methodology that converts source code into images and utilizes image classification networks.
Wang et al.[15]	Gated Hierarchical LSTM	Utilizes both traditional features and features extracted from AST's using gated merge layer.
Xu et al.[16]	GNN	They created their database by utilizing GitHub pull requests.
Deng et al.[17]	Bidirectional LSTM	They use Bidirectional LSTM to perform SDP.
Huo et al.[18]	CNN	Novel method that includes code comments aswell as source code in building SDP.
Cai et al.[19]	Tree Based Embedding CNN	They create a method that tries to improve CPDP by using TCA to create transferable features.
Pan et al.[20]	CodeBERT	They use a language model that combines programming language and natural language queries to perform SDP.
Qui et al.[22]	Transfer CNN	They train a CNN using both source and target project to improve CPDP performance.
Munir et al.[23]	Gated Recurrent Unit-LSTM	Provides statement level defect prediction, unlike other methods which are file level or change level.
Wang et al.[24]	Graph CNN	They construct graph representation of the source code and use it to perform SDP.
Shen et al.[25]	BERT + Graph CNN	They use a language model to generate features which are then used to train GCNN to perform SDP.
Yao et al.[28]	Tree-based CNN	They perform defect feature mining by first training a TCC with defective code to perform SDP.

Wang et al.[10] build a software defect prediction model using Deep Belief Network(DBN). They extract token vectors from AST representation of the source code and perform defect prediction on file-level and change-level. File-level defect prediction process predicts whether a given source file contains software defects while change-level defect prediction process focuses on commits made in a Version Control System(VCS) like Git. They use AST of Java language and extract 3 types of nodes from the AST: methods invocations, declarations and control flow statements. They exclude any other AST node. Since methods invocations and declarations are project specific, they record them as their node type to improve CPDP. The DBN based defect prediction model outperforms state of the art methods in WPDP by 13% and CPDP by 6% for file level defect prediction. For change level defect prediction the DBN method outperforms the state of the art methods by 5.1% and 2.9% for WPDP and CPDP respectively.

Liang et al.[11] train a SDP model using Long-Short Term Memory(LSTM) neural network. They extract token sequence from the AST and convert it into real valued vectors using a word embedding model. The word embedding model used is the Continuous Bag of Words(CBOW) model provided by word2vec. From AST, 3 types of nodes are extracted: Method invocation and class instance creation, declaration and control flow nodes. Method invocation and class instance creation and declaration nodes are recorded using their names. These names are further preprocessed into substrings to convert different styles of namings used between projects to a uniform style. This method improves upon state of the art methods in both WPDP and CPDP.

Dam et al.[12] use a tree-based LSTM network to perform SDP. They parse the source code into an AST representation. The root of AST represents the whole source file and its children are the elements of source file such as class declarations, method declarations, method body contents etc. The AST nodes are converted into real valued vectors in the embedding step using ast2vec methodology. An embedding matrix is trained along the LSTM model to perform word embedding. The AST is recursively traversed and each node is fed into the LSTM. The hidden state and context vectors are calculated for each node and is then used to calculate the same vectors for the parent. The output of children nodes are combined to be fed into the parent node's LSTM. Finally a classifier is used at the top node to predict whether given source file contains defects or not. They used Logistic Regression and Random Forest classifiers in this paper. They used two dataset, one provided by Samsung's own code base called Tizen Operating System and another from PROMISE dataset. They have found that for WPDP and Samsung dataset, RF classifier outperformed the LR classifier. The RF classifier performed very well across four metrics: F-measure, Recall, Precision and Area Under the ROC Curve. All metrics provided a value of 0.9 for RF while LR only provided high recall but provided many false positives. For WPDP and PROMISE dataset they observed LR performing better. They explained this difference due to the fact that PROMISE dataset having smaller number of data points. Overall they have found their method to have higher recall but lower precision than state of the art method in WPDP. They note that higher recall is preferable to higher precision because the cost of

missing defects are higher than cost of having false positives. They achieved similar results for CPDP, where there is high recall and low precision. They have found that their predictions are considerably better than random prediction from AOC metric.

Fan et al.[13] propose an defect prediction via attention based recurrent neural network(DP-ARNN). They focus on file level WPDP. They parse the source code into AST and perform node filtering. They pick nodes of method invocations, nodes of declarations and nodes of control flow and drop every other node. They perform depth first traversal to form token vectors. Then they perform an integer mapping of tokens where each token maps into a unique integer. To handle class imbalance in the SDP training data they perform oversampling to not lose important data. They selected 7 open source projects from Apache to form dataset. Their arhitecture consist of an embedding layer, Bidirectional-LSTM(BiLSTM) layer, an attention layer, two fully connected layers and an output layer. The embedding layer is trained together with the whole network. They use a BiLSTM network because a bug might be caused by not only samples from before, but also from after the training sample. Going bidirectional means the network can learn from both directions. They compare their work to two different deep learning methods, namely CNN and RNN(BiLSTM without attention mechanism) networks. They found that DP-ARNN improves upon compared works, specifically 14% on F1 measure and 7% on AUC metric.

Chen et al.[14] propose a Deep Transfer Learning for Defect Prediction(DTL-DP) that avoids intermediary representations like AST and instead converts the source code into an image. This way they can also leverage the already existing image classification networks capabilities. DTL-DP is made of two stages: source code visualization and DTL. For source code visualization they convert each character in the source code into pixels by using their ASCII values as pixel intensity. They implement a novel data augmentation method by converting consecutive 3 pixel values into permutations of RGB(RGB, BGR, GBR...). This way they can increase the amount of data by 6 fold. Their network architecture consists of an input layer, 5 convolutional layers from AlexNet(conv1-conv5), an attention layer and 4 fully connected layers acting as the classifier. They used the PROMISE dataset for their work. DTL-DP performs competitively with state of the art method in WPDP and outperforms other deep learning based methods.

Wang et al.[15] use a Gated Hierarchical LSTM(GH-LSTM) architecture to perform SDP. They use both semantic features and traditional features in their work. They first process source code into AST representation. They extract three types of AST nodes: method invocation, declaration and control flow types. Then they train a word embedding model called GloVe. For traditional SDP features they selected 18 code metrics provided by the PROMISE dataset. They feed both the embedded features and traditional features hierarchically into LSTM networks. The output of the LSTM networks are then fed into a gate layer composed of fully connected network to filter out the data. Resulting features are concatenated to create a combined feature, which is fed into a last fully connected layer to

classify the code as buggy or clean. To evaluate their model they have picked 10 projects from PROMISE dataset. They found that GH-LSTM improves upon the performance of state of the art WPDP tasks.

Xu et al.[16] deploy graph representation learning model to train a model. They created GitHub pull request(GHPR) dataset where they queried GitHub repositories for defect fixing pull requests(PR). Pull requests are better than git commits because they contain less noise due to being reviewed before being accepted. They collected the code before the fix as defected sample and code after the fix as clean sample. For projects, they selected them based on two criterias: projects having at least 1000 forks and projects being Java language projects. They choose projects with at least 1000 forks because that indicated more PR activities. They collected PR's that have been merged into master branch and marked as a defect in the PR. Then they extracted the source code from commits in the PR. The source code is parsed into AST representation and mapped into integers. The fixes applied to defects are only a small part of the AST, so to remove redundant information and noise the AST tree's are pruned using Louvain Algorithm. This leaves AST subtrees that are related to the defect. They use a topic model to capture the semantic high level meaning of the source code. The AST subtrees are treated as the graph and the concept features extracted from topic model are used as graph attributes. These are given as input to the GNN. Their work improves upon the performance of state of the art DP.

Deng et al.[17] propose DP-LSTM which is based on Bidirectional LSTM. They parse the source code into AST representation, create a mapping between integers and AST nodes and perform word embedding to create numerical vectors from tokens. Token sequences are fed into the BiLSTM network to train it. A Logistic Regression classifier is used to classify outputs as buggy/non-buggy. They choose PROMISE dataset to evaluate their model. They found that their model outperforms state of the art methods in F-measure.

Huo et al.[18] introduce Convolutional Neural Network for Comments Augmented Program(CAP-CNN). They include the comments in the source code to generate semantic information. Their claim is that comments include extra information on describing the structure of the code. They split source code into source code and comments part. They form vectors using word2vec model. Since code comments and code itself have different structure, they use two CNN's to process embedding vectors. The resulting features are then fed into a fully connected network to classify the source code as defective or not. They propose a novel training method to overcome the problem that not every code contains comments. They found that they can improve the performance of SDP over state of the art methods.

Cai et al.[19] developed a method to improve CPDP tasks called tree-based-embedding convolutional neural network with transferable hybrid feature learning(TBCNN-THFL). They note that real valued

vectors obtained from AST's of the source code do not represent the semantic distance between different projects really well. They used PROMISE dataset for their work. They parse the source code into AST; they ignore nodes other than the following three: method invocation, declaration nodes and control flow nodes. To capture the context, they use the parent and the children of a node together with the central node to train CBOW word embedding. Then they use a 1-D convolutional layer, a maximum pool layer, a fully connected layer and an output layer. They augment the features generated by the output layer by concatenating handcrafted features used in [26]. They feed the augmented features into TCA[27] to generate transferable features. They use a LR classifier to decide whether source code is buggy or not. To overcome the data imbalance problem they use synthetic minority oversampling(SMOTE). Their method shows improvement over state of the art methods.

Pan et al.[20] introduce CodeBERT, a pre-trained programming language model for SDP tasks. CodeBERT[21] is a transformer based natural language/programming language pre-trained model created by Microsoft. They first choose a prediction model for their work. Traditional prediction model takes in the source code as input and outputs whether it contains a bug or not. Since CodeBERT can also understand natural language they create two more prediction models: one which takes source code and a declarative sentence such as "The code is buggy" and outputs whether the sentence matches the source code; another one which takes a list of keywords like "bug", "defect", "error" etc. and the source code, which outputs whether the list matches the source code. They tokenize the source code and perform a mapping to integers, they use BertTokenizer provided by the CodeBERT. They perform random oversampling to deal with the class imbalance. They use both pre-trained CodeBERT model and training it from scratch using the same architecture. For pre-trained model they reuse the weights of the encoder-decoder part and reset the weights of the classification parts. They use the PROMISE dataset for their work. They found that the pre-trained CodeBERT model outperforms the one trained from scratch and it is also 4 times faster to train. They also found that both sentence based and keyword based prediction models outperform traditional models when performing SDP.

Qui et al.[22] offer a novel method for CPDP called Transfer Convolutional Neural Network(TCNN). They introduce a matching layer into CNN for matching different distributions of different projects to improve CPDP. They build an AST representation from source code first and perform a mapping into integers for each node. They feed both source project with labels and target project without labels into the CNN. They calculate classification loss from source project's output and calculate a distribution divergence between source and target projects using the matching layer. The aim of the training process is minimizing both classification error and distribution divergence. Finally they combine the generated features with traditional features into what is called transferable joint features and use a LR classifier to get the output. They used PROMISE dataset for their work. They found that TCNN outperforms traditional metrics in CPDP. It also provides better SDP performance compared to the state of the art method.

Munir et al.[23] propose a statement-level defect prediction method called Attention-Based Gated Recurrent Unit(GRU)-LSTM(DP-AGL). Traditional methods focus on file, function or class defect prediction. They parse the source code into AST structure and pick 32 metrics from nodes. These metrics include but not limited lo literal string, literal count, variable count etc. They embed tokens into the metrics vector to form a matrix where rows are the lines of the program and columns are the metrics/tokens. They train a GRU-LSTM layer and an attention layer. They used Code4Bench for C/C++ dataset in their work. They compared their work to SLDeep model and a Random Forest based SDP model and they have found it to be more effective.

Wang et al.[24] provide a cross version defect prediction system(CVDP) in their work. CVDP only focuses on parts of code that may be affected by the changes. They look for keywords such as "fix", "bug", "defect" in version control system commits. The commits contain the code segment that included the defective code and the code segment that fixed the defect. They first construct AST from the functions in the code. From the AST, they create control flow relationship. This control flow relation ship forms a graph where the edges of the graph follow the execution of the code statements. They also model data flow dependencies by creating three types of edges between nodes: variable definition edge, variable use edge and variable modification edge. From the graph they have constructed, they extract both features about nodes and features about changes. The code features include features like whether the node is a control flow node or not, number of function calls etc. The code change features includes features like number of revisions, number of modified lines etc. The constructed graph and features matrix is used for training the Graph Convolutional Neural Network(GCNN). An output layer using Soft Max as classifier is used to detect whether the code change includes a defect or not. Their work outperformed 3 other CVDP methods in F1 score. They also compared their work to a CPDP work called ADCNN, using the same datasets and found that it also performed 27% higher in F1 score.

Shen et al.[25] use a GCNN based on a pre-trained BERT model. They first parse the source code into AST representation. They use BERT to extract code semantic features from the code. They extract descriptive features from code comments using Latent Dirichlet Assignmnet(LDA) and fuse the features together by concatenating them together. They perform oversampling to deal with the class imbalance problem. Specifically they use GraphSMOTE to generate a graph and create a balanced graph. They use GraphSAGE model to process the data. They use an output layer using Soft Max as classifier. They compared their work to Pan's CNN, Seml and GCN2defect and it outperformed all of them respectively by 9.7%, 6.6% and 4.9% in F1 measure.

Yao et al.[28] introduce Program Semantic Feature Mining(PSFM) method for SDP. They use the code text and syntax structure information to create semantic information for the program. They first extract the AST of the source code. They create a token sequence by traversing the AST by breadth first traversal. They use Tree-based CNN(TBCNN) to create syntax structure information. They use

an attention module with the token sequence to create text information. They fuse these two information using cross multiplication. Finally they use a CNN network to mine defect features. They feed the features into LR classifier to detect whether the source code is buggy or not. They found that their work performed best compared to other deep learning based methods.

6. CONCLUSION

This survey presented the latest, most cited works in software defect prediction that focus on semantic methods. A total of 17 studies were surveyed, from the last 5 years where studies were picked according to their citation counts. To avoid only including older studies from 2023 were also included. All the works converted the source code into AST representation to extract semantic information from the source code eventually. While most works directly converted AST representation into vector representations, some works utilized the fact that AST is a graph, that the nodes and their relationships contain extra information. There are also works that try to enhance the results by including the traditional metrics together with features created by the neural networks. Most studies use LSTM and CNN to train on the vector representations while several studies utilized BERT language model, specifically CodeBERT pre-trained model for programming languages. They utilize natural language prompts with source code information to predict defects in the code. Semantic methods showed a significant improvement over traditional methods in performance. One of the challenges all works faced was the imbalance of datasets. The datasets contain more clean pieces of data compared to the buggy pieces of data. Most works decided to use oversampling instead of undersampling to avoid losing information. Some works utilized SMOTE technique to perform oversampling. All the previous works focus on performing SDP on a single programming language. Future works are recommended to consider cross programming language SDP. This will increase the applicability of the semantic SDP methods. Also it enables even more data for a model as they will no longer be constrained to data from single language.

7. REFERENCES

[1] S. Kumar and S. S. Rathore, Software Fault Prediction A Road Map. Berlin, Germany: Springer, 2018.

[2] S. Hochreiter and J. Schmidhuber, ‘Long Short-term Memory’, Neural computation, vol. 9, pp. 1735–1780, 12 1997.

[3] Y. Bengio, Learning Deep Architectures for AI, 2009.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ‘Efficient Estimation of Word Representations in Vector Space’, arXiv [cs.CL]. 2013.

[5] A. Vaswani et al., ‘Attention Is All You Need’, arXiv [cs.CL]. 2023.

[6] W. Ju et al., ‘A Comprehensive Survey on Deep Graph Representation Learning’, arXiv [cs.LG]. 2023.

[7] E. N. Akimova et al., ‘A Survey on Software Defect Prediction Using Deep Learning’, Mathematics, vol. 9, no. 11, 2021.

[8] S. Omri and C. Sinz, ‘Deep learning for software defect prediction: A survey’, in Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops, 2020, pp. 209–214.

[9] A. Abdu, Z. Zhai, R. Algabri, H. A. Abdo, K. Hamad, and M. A. Al-antari, ‘Deep Learning-Based Software Defect Prediction via Semantic Key Features of Source Code—Systematic Survey’, Mathematics, vol. 10, no. 17, 2022.

[10] S. Wang, T. Liu, J. Nam and L. Tan, "Deep Semantic Feature Learning for Software Defect Prediction," in IEEE Transactions on Software Engineering, vol. 46, no. 12, pp. 1267-1293, 1 Dec. 2020, doi: 10.1109/TSE.2018.2877612.

[11] H. Liang, Y. Yu, L. Jiang and Z. Xie, "Seml: A Semantic LSTM Model for Software Defect Prediction," in IEEE Access, vol. 7, pp. 83812-83824, 2019, doi: 10.1109/ACCESS.2019.2925313.

[12] H. K. Dam et al., ‘A deep tree-based model for software defect prediction’, arXiv [cs.SE]. 2018.

[13] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, ‘Software Defect Prediction via Attention-Based Recurrent Neural Network’, Scientific Programming, vol. 2019, p. 6230953, Apr. 2019.

[14] J. Chen et al., ‘Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction’, in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, South Korea, 2020, pp. 578–589.

[15] H. Wang, W. Zhuang and X. Zhang, "Software Defect Prediction Based on Gated Hierarchical LSTMs," in IEEE Transactions on Reliability, vol. 70, no. 2, pp. 711-727, June 2021, doi: 10.1109/TR.2020.3047396.

[16] J. Xu, F. Wang and J. Ai, "Defect Prediction With Semantics and Context Features of Codes Based on Graph Representation Learning," in IEEE Transactions on Reliability, vol. 70, no. 2, pp. 613-625, June 2021, doi: 10.1109/TR.2020.3040191.

[17] J. Deng, L. Lu, and S. Qiu, ‘Software defect prediction via LSTM’, IET Software, vol. 14, no. 4, pp. 443–450, 2020.

[18] X. Huo, Y. Yang, M. Li and D. -C. Zhan, "Learning Semantic Features for Software Defect Prediction by Code Comments Embedding," 2018 IEEE International Conference on Data Mining (ICDM), Singapore, 2018, pp. 1049-1054, doi: 10.1109/ICDM.2018.00133.

[19] Z. Cai, L. Lu and S. Qiu, "An Abstract Syntax Tree Encoding Method for Cross-Project Defect Prediction," in IEEE Access, vol. 7, pp. 170844-170853, 2019, doi: 10.1109/ACCESS.2019.2953696.

[20] C. Pan, M. Lu, and B. Xu, ‘An Empirical Study on Software Defect Prediction Using CodeBERT Model’, Applied Sciences, vol. 11, no. 11, 2021.

[21] Z. Feng et al., ‘CodeBERT: A Pre-Trained Model for Programming and Natural Languages’, CoRR, vol. abs/2002.08155, 2020.

[22] S. Qiu, H. Xu, J. Deng, S. Jiang, and L. Lu, ‘Transfer Convolutional Neural Network for Cross-Project Defect Prediction’, Applied Sciences, vol. 9, no. 13, 2019.

[23] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, and S. Qayyum, ‘Attention based GRU-LSTM for software defect prediction’, PLOS ONE, vol. 16, no. 3, pp. 1–19, 03 2021.

[24] Z. Wang et al., ‘BugPre: an intelligent software version-to-version bug prediction system using graph convolutional neural networks’, Complex \& Intelligent Systems, vol. 9, no. 4, pp. 3835–3855, Aug. 2023.

[25] H. Shen, X. Ju, X. Chen and G. Yang, "EDP-BGCNN: Effective Defect Prediction via BERT-based Graph Convolutional Neural Network," 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), Torino, Italy, 2023, pp. 850-859, doi: 10.1109/COMPSAC57700.2023.00114.

[26] M. Jureczko and L. Madeyski, ``Towards identifying software project clusters with regard to defect prediction,'' in Proc. 6th Int. Conf. Predictive Models Softw. Eng., 2010, p. 9.

[27] S. J. Pan, I. W. Tsang, J. T. Kwok and Q. Yang, "Domain Adaptation via Transfer Component Analysis," in IEEE Transactions on Neural Networks, vol. 22, no. 2, pp. 199-210, Feb. 2011, doi: 10.1109/TNN.2010.2091281.

[28] W. Yao, M. Shafiq, X. Lin, and X. Yu, ‘A Software Defect Prediction Method Based on Program Semantic Feature Mining’, Electronics, vol. 12, no. 7, 2023.