# A SURVEY ON SOFTWARE DEFECT PREDICTION USING DEEP LEARNING

A PREPRINT

**Akimova E.N. (ORCID 0000-0002-4462-5817)** *
aen15@yandex.ru

**Bersenev A.Yu.***
Alexander.Bersenev@urfu.ru

**Deikov A.A.***
hx0day@hackerdom.ru

**Kobylkin K.S.***
kobylkinks@gmail.com

**Konygin A.V.***
konygin@imm.uran.ru

**Mezentsev I.P.***
ilyamezentcev@gmail.com

**Misilov V.E. (ORCID 0000-0002-5565-0583)** *
v.e.misilov@urfu.ru

April 16, 2021

**ABSTRACT**

Defect prediction is one of key challenges in software development and programming language research for improving software quality and reliability. The problem in this area is to properly characterize the source code, which contains defects with high precision and recall. Developing fault prediction model is challenging task, and many approaches have been proposed through the history. This is especially difficult due to the rarity of defects and the extreme variety of correct source code – the problem of unbalanced distribution.

Traditional studies were focused on hand-crafted features of the code, such as complexity metrics, which are fed into classifiers to identify correct and defective code. However, these features often fail to capture the structural and semantic information of code, which is essential for building accurate models. The recent breakthrough in machine learning technologies, especially, the development of deep learning techniques, led to the fact that many problems began to be solved by these methods.

The survey focuses on deep learning techniques for defect prediction. We analyse the recent works on the topic, study the methods for automatically learning of the semantic and structural features from the code, discuss the strengths and weaknesses of each approach. We also present the survey on the modern code datasets suitable for training models for software defect prediction.

***Keywords*** Defect prediction · Anomaly detection · Program analysis · Code understanding.

## 1 Introduction

According to IEEE Standard Classification for Software Anomalies [17], software defect is an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.

Software defects can cause different problems. Traditional ways of combating software defects, such as testing and code review, require large amounts of time and specialized teams to be performed thoroughly. In order to reduce the needed resources, defect prediction tries to automate defect discovery and thus minimize operational costs, while improving software quality.

Thus, finding defects is a core problem in software engineering and programming language research. The challenge in this domain rests in correctly characterizing source code that contains a defects with high precision and recall.

*N.N. Krasovskii Institute of Mathematics and Mechanics, Ural Federal University

The development and breakthrough of machine learning led to the fact that many tasks began to be solved by machine learning (ML) methods.

Recent advances in both machine learning algorithms and computer hardware (such as supercomputers based on GPUs with AI accelerating modules) allowed the new concepts, such as Deep Learning, to emerge. The main idea is that an artificial neural network with multiple layers is capable to progressively extract the higher-level features from the original data to solve complex problems.

In the field of software defect prediction, the researchers have proposed the representation-learning algorithms to learn semantic representations of programs automatically and use this representation to identify the defect-prone code. Using these implicit features shows better then the previous approaches based on the explicit features, such as code metrics.

Source code remains the main source of data for defect prediction. Some semantic defects are hard to find using only source code. For example, in [5], the bytecode of Kotlin programs is processed to detect the so called compiler-induced anomalies which arise only in the compiled bytecode. Another example is presented in work [23], where to expose the program behavior, C source code is firstly compiled into assembly code which is then used to learn defect features.

In this survey, our main interest lies in techniques devoted to analyzing the source code. Usually, the process of defect prediction consists of following steps (see Fig. 1):

1. Prepare the datasets by collecting the source code samples from repositories of software projects
2. Extract features from the source code
3. Train the model using the train dataset
4. Test the model using the test dataset and assess the performance using quality metrics

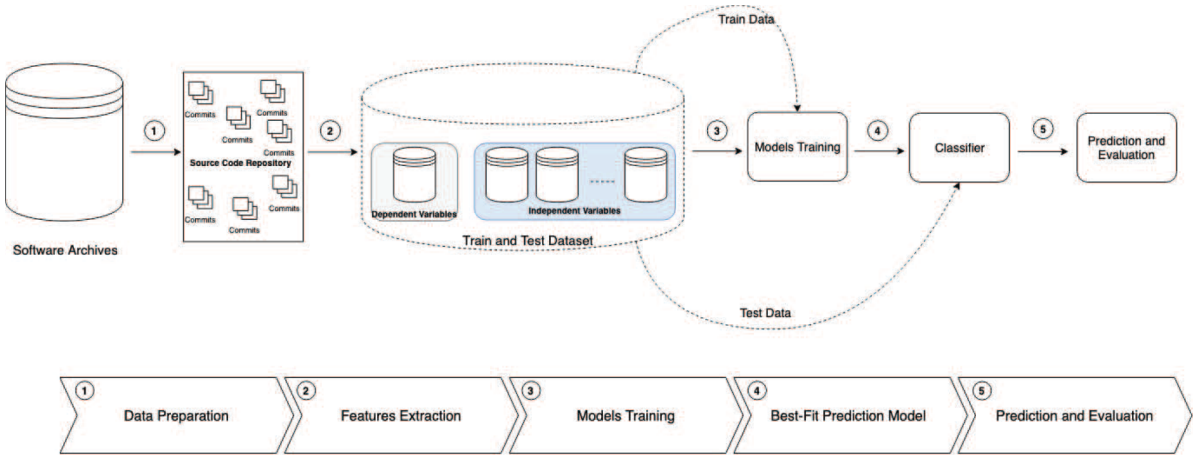


Figure 1: Scheme of defect prediction process.

## 2 Methodology

We performed a systematic literature review on the subject. In this section, we present details of our methodology.

### 2.1 Research Questions

To summarize the work of our survey, let us formulate the following research questions:

- RQ1. What deep learning techniques have been applied for software defect prediction?
- RQ2. What are the key factors contributing to difficulty of the problem?
- RQ3. What are the trends in the primary studies on the problem?

### 2.2 Literature Search and Inclusion or Exclusion Criteria

To collect related papers, we formulated a search string for Google Scholar and Scopus combining the related terms:

("software engineering" AND "deep learning" AND ("defect prediction" OR "anomaly detection") )

To filter the papers with insufficient content and determine the paper quality, we used the following criteria:

- The paper must describe a technique for automatic feature extraction using deep learning and apply it for defect prediction problem
- The paper length must not be less than 6 pages

## 3 RQ1. What techniques have been applied for this problem?

In order to work with the source code, you need to have some representation. On the one hand the representation should be simple as a vector, since most of machine learning algorithms work with vectors. On the other hand the representation should contain all necessary information. There are different ways to build representations for the source code (e.g., [2], [7]).

One way is to create the vector from *hand crafted* features. This approach assumes that an expert invents a set of features and selects best of them. (e.g., [26]). Usually, these features include the statistical characteristics of code, such as its size, code complexity, code churn, or process metrics.

Other way is to view code as a *sequence* of elements, usually code tokens or characters. Most sequence-based models are trained to predict sequences by generating the subsequent element.

One more approach to build the representation of the source code is *abstract syntax trees* (AST). Such models make simplifying assumptions about how a tree is generated, usually following generative NLP models of syntactic trees: they start from a root node, then sequentially generate children top-to-bottom and left-to-right. Syntactic models are trained to generate a tree node conditioned on context defined as the forest of subtrees generated so far. In contrast to sequence models, these models — by construction — generate syntactically correct code.

Usually, building the representation is an intermediate step in solving a problem. On the other hand, some representations are designed to solve several related tasks (for example, CuBERT, CodeBERT). Thus, there is a close connection between all tasks related to code understanding.

Most defect prediction approaches consider defect prediction as a binary classification problem and solve it by classification algorithm. These techniques classify source code into two categories: defect code and correct code.

However, the approaches based on the hand-crafted features, usually do not sufficiently capture the syntax and semantics of source code, which is important for accurate defect prediction. Most traditional code metrics cannot distinguish code fragments with different semantics but similar code structure and complexity. For example, if we switch several lines in the code fragments, traditional code characteristics in terms of lines of code, function calls, raw programming tokens would remain the same.

Modern approaches are usually based on extracting the implicit structural, syntax, and semantic feature from the source code rather than using the explicit hand-crafted ones.

The most popular deep learning techniques for software defect prediction are: Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), Long Short Term Memory (LSTM), and Transformer architecture.

### 3.1 Deep Belief Networks

A Deep Belief Network is a generative graphical model that uses a multilevel neural network. It contains one input layer and several hidden layers. The top layer is the output layer that is used as feature to represent input data. Each layer consists of several stochastic nodes with connection between the layers but not between nodes within each layer as shown in Fig. 2.

Perhaps one of the first works combining AST with deep learning are [36]. The authors propose the "Deeper" approach for software defect prediction on changes level. They use the DBN to generate the new expressive features from existing ones and use these new features in a classical machine learning classifiers. They extract the relations from the traditional change level features, such as number of modified modules, directories, and files, added and deleted lines, and several features related to developer's experience.

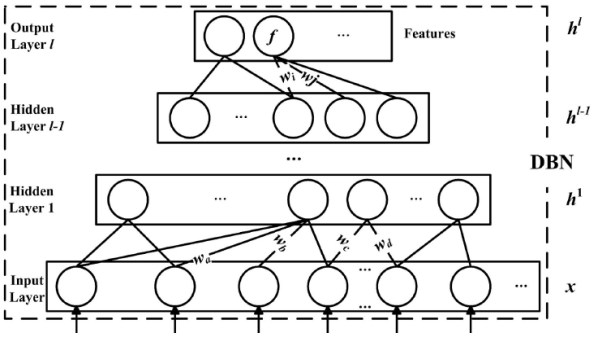


Figure 2: Architecture of Deep Belief Network.

Later, the authors have proposed "TLEL" approach [37], based on the decision tree and ensemble learning. In the inner layer, it combines decision tree and bagging to build a Random Forest model. In the outer layer, it uses random under-sampling to train many different Random Forest models and stacking to ensemble them once more.

The works of Wang et al. [32, 33] also use the DBN, but in a different manner. To bridge the gap between programs' semantics and defect prediction features, the authors have developed a DBN to automatically learn a semantic features from the source code. Their techniques extract semantic feature from program's AST for the file-level defect prediction and from source code changes for the change-level defect prediction models. Then, the authors use classical machine learning classifiers and extracted features to classify source code files whether they are buggy or clean.

DBN however does not naturally capture the sequential order and long-term dependencies in the source code.

### 3.2 Long Short Term Memory

Long Short Term Memory is a recurrent neural network, which maps a sequence of input vectors into a sequence of output vectors. LSTM network consists of LSTM units (see Fig. 3) with memory cells, which stores accumulated memory of the context. This is a key feature allowing LSTM models to learn long-range information and long-term dependencies in the source code.

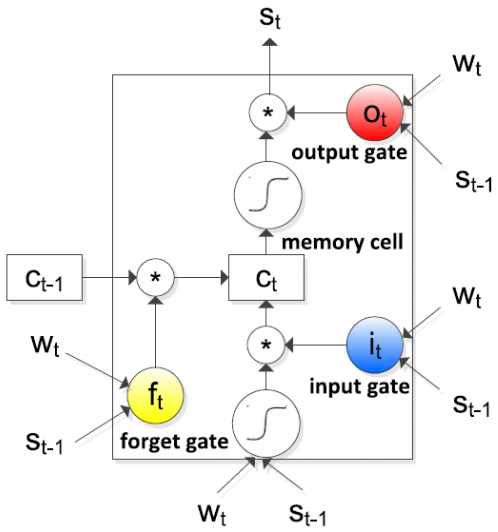


Figure 3: Scheme of LSTM unit.

In [8], the LSTM-based model for learning both semantic and syntactic features of code is constructed. The proposed approach represents the code as a sequence of code tokens, which is fed into a LSTM system to transform code into a feature vector and token state, which captures the token semantics. Later [9], the authors developed Tree-LSTM model using the AST representation as input.

In [13] a neural bug finding technique is proposed. The authors formulate bug detection as classification problem and use neural networks traided on examples of buggy and non-buggy code. The authors use existing static bug detection software to obtain warnings about specific kind of bugs and train a neural model to distinguish code that produces particular warning frome code without such a warning. The code is represented as a tokens sequence and converted to a real-value vector by using the one-hot encoding for each token. Then, a bi-directional RNN with LSTM is used as model.

In [27], inspired by code2vec ([3]), the authors propose a new AST path pair-based source code representation method (PathPair2Vec) and apply it to software project defect prediction. This model extracts the paired path in the sub-AST method of the source code and encodes the combination of symbol sequence and control sequence to form a path vector using the Bi-LSTM architecture. Then, a global attention is used to generate the vector of the entire source code. This final embedding representations is used for classification.

### 3.3 Convolutional Neural Networks

Convolutional Neural Networks are a specialized kind of neural networks for processing data that have a known grid-like topology. CNN have two key characteristics: Sparse Connectictivity and Shared Weights, which can benefit the defect prediction in capturing local structural information of programs. Sparce Connectivity means that the network uses a local connection pattern between units of adjacent layers to generate spatially local correlation of the input data. Shared Weights mean each filter shares the same parameterization. Replicating the filters allows the network to capture the features regardless of their position in the input vector. The scheme of general CNN is shown in Fig. 4

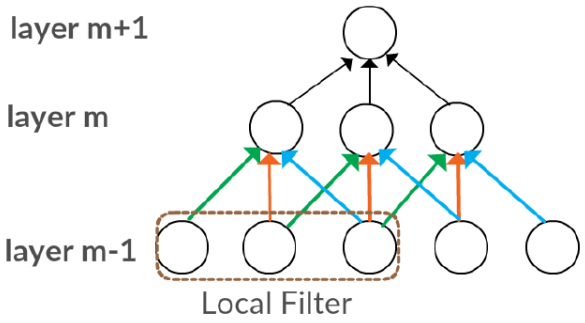


Figure 4: Architecture of Convolutional Neural Network.

In [21], the DP-CNN framework is presented. Based on the program's AST, the token vectors are extracted and encoded as numerical vectors via mapping and word embedding. Then, these vectors are fed into a convolutional neural network to automatically learn semantic and structural features of programs. After that, the combination of learned features and traditional hand-crafted features is used for software defect prediction using logistic regression.

In [15], a deep learning model DeepJIT is presented, that automatically extract features from commit messages and code changes to use them to identify defects. This model is based on the CNN. It uses the convolutional network layers for processing code changes and commit text and the feature combination layer to fuse the two embedding vectors into a single one.

In [35], a framework to learn deep feature representation is proposed. It combines a triplet loss and a weighted cross-entropy loss for DNN training. The model not only retains the original characteristics of the code but also automatically adjusts the distances among the modules with the same or different labels. The random forest is used as classifier.

In [24], a Transferable Hybrid Features Learning with CNN (CNN-THFL) is proposed. This model mines features from token vectors extracted from program's AST and learns the transferable joint features. Combining these deep-learning-generated features with the hand-crafted ones allows the model to perform cross-project defect prediction. Later, in [6], the authors propose a new tree-based convolutional network TBCNN-THFL to perform this task. It uses tree-based continuous bag-of-word for encoding AST nodes to be fed into CNN.

### 3.4 Transformer models

In recent time, inspired by the big success of pre-trained contextual representations in Natural Language Processing, *e.g.*, [22], there has been a rise of attempts to apply these techniques to source code. Usually, these models are based on

the multi-layer transformer architecture [31] shown in Fig. 5. They are pre-trained using massive unlabeled coprora of programs with self-supervised objectives such as masking language modeling and next sentence prediction [18, 11]. The pre-trained model can be fine-tuned for specific tasks using supervised techniques.

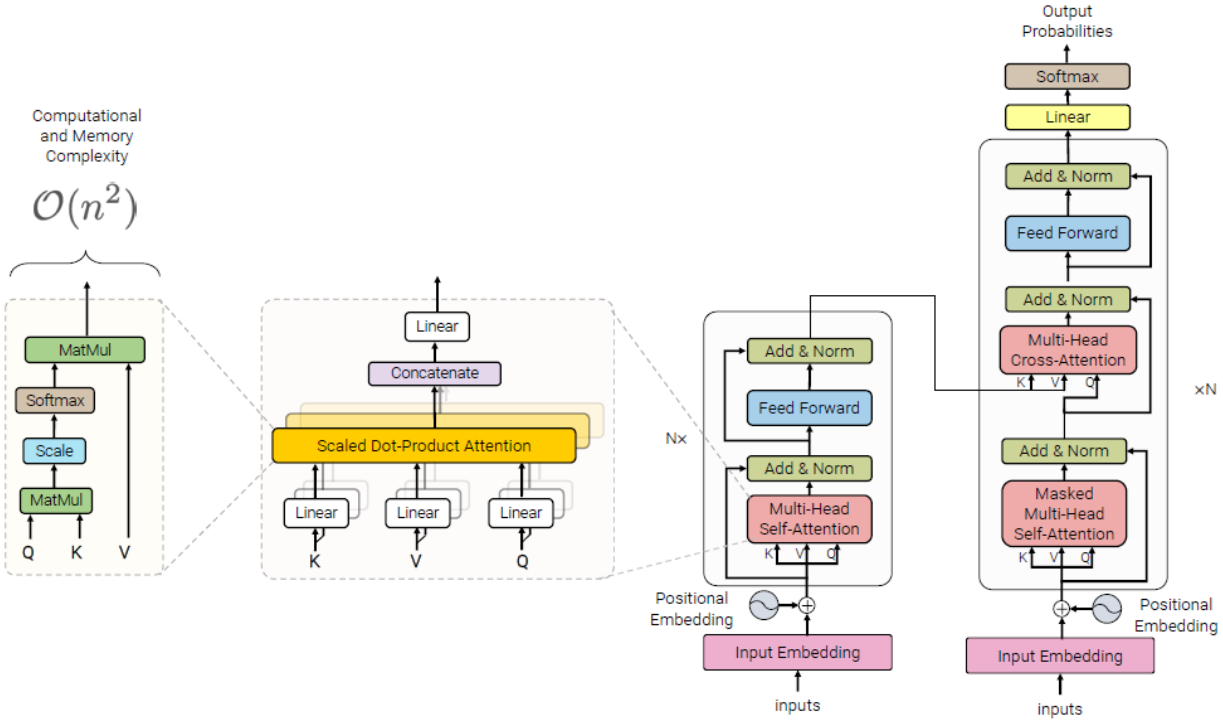


Figure 5: Architecture of multi-layer transformer

In [16] states that the approaches based on the traditional complexity metrics are useless since there is no need for a tool to tell the engineer that longer and more complex code is more defect-prone. The methods [32] of learning features from source code does not guarantee capturing semantic and syntactical similarity and very similar source code can have very different features. They suggest that these features correlate with defects rather than directly cause them. In contrast, they propose the approach based on the self attention transformer encoder for semantic defect prediction. The matrix representing the defectiveness of each token in the fragment is generated. Attention and layer normalization are used as a regularization techniques. The resulting model provides the defect prediction with semantic highlight of defective code regions.

In [18], the authors use a corpus of Python files from GitHub to create a benchmark for evaluating code embeddings on five classification tasks and a program repair task. They train CuBERT model and compare it with various other models, including BiLSTM with word2vec, and Transformer. It is shown that CuBERT outperforms the baseline models consistently.

In [11], a bimodal pre-trained model for programming language and natural language is presented. CodeBERT is based on the multilayer bidirectional transformer neural architecture. The natural language text is presented as a sequence of words, and piece of code is presented as a sequence of tokens. The output of CodeBERT consists of a contextual vector representation of each token, for both natural language and code, as well as, the representation of the aggregated sequence. The resulting model efficiently solves the problems of both code to documentation and natural language code search.

Work [12] presents a multi-layer bidirectional transformer architecture GraphCodeBERT which utilizes three components as input: the source code, paired comments, and data flow graph. This allows the model to consider code structure for code representation. For pre-training tasks, the traditional masked language modeling, as well as, the edge prediction and node alignment of data flow were used. It supports several downstream code-related tasks including code clone detection, code translation, and code refinement.

### 3.5 Other Networks

In [29], a software defect prediction technique based on stacked denoising autoencoders model and two-stage ensemble learning is presented. Stacked denoising autoencoder is used to extract deep representations from the traditional metrics. To address the class-imbalance, the authors use the ensemble learning strategy. Later, the feature selection algorithm was applied to this method [30] to address the feature renundancy problem.

In [40], a model for software defect prediction is presented, called Siamese parallel fully-connected networks (SPFCNN), which combines the advantages of Siamese networks and deep learning into a unified method. A pair of parallel Siamese networks are used to extract the highest-level representation from the high dimensional attributes for SPFCNN training and testing. And the cost-sensitivity features are integrated into SPFCNN to achieve a balance between the classification performance of minority and majority classes.

In [25], the neural forest network is used to learn new feature representations from hand-crafted features. Then, a decision forest is connected after the neural network to perform classification and guide the learning of feature representation. In [41], a new deep forest model is proposed for software defect prediction. This model can identify more important defect features by using a new cascade strategy, which transforms random forest classifiers into a layer-by-layer structure.

In [34], the graph neural network is utilized for identifying software defects via the combination of semantics and context information using abstract syntax tree representation learning. To extract the defect-related information from the source code, the ASTs for buggy and fixed version of a fragment are constructed and pruned using the community detection algorithm which extracts the defect-related subtree. Then, the Graph Neural Network is used to capture the latent defect information.

## 4 RQ2. What are the key factors contributing to difficulty of the problem?

From a machine learning point of view the problem of software defect prediction is considered very complex and very challenging.

### 4.1 Lack of data

One of the difficulties is lack of available large labeled datasets devoted to the defect prediction. To alleviate this problem, one can utilize the pre-trained contextual embeddings. This technique consists in pre-training the language model on a massive corpora of unlabeled source code using self-supervised objectives, such as masked language modeling, next sentence prediction, and replaced token detection. Table 1 presents the popular unlabeled code datasets suitable for this task.

| Dataset | Content | Size | Used in tasks |
|---|---|---|---|
| Py150 | Python source code, AST | 8423 repos, 149993 files | Fine-tuning CUBERT models for Variable-Misuse Classification, Wrong Binary Operator, Swapped Operand, Function-Docstring Mismatch, Exception Type, Variable-Misuse Localization and Repair |
| Big-query github repos | GitHub metadata | 250M commits, Metadata for 2 billion files, metadata for 3 million projects | Pre-training CUBERT model for Variable-Misuse Classification, Wrong Binary Operator,Swapped Operand, Function-Docstring Mismatch, Exception Type, Variable-Misuse Localization and Repair |
| Li-braries.io | Metadata for software packages | 306k packages | Empirical Comparison of Dependency Network |
| GHTor-rent | GitHub metadata | | |

Table 1: List of unlabeled datasets

The pre-trained model then may be fine-tuned for defect prediction using much smaller labeled datasets. Table 2 presents a list of publicly available datasets devoted to defect prediction. Usually, such datasets include pairs of "correct" and "defective" code fragments.

As the other factors affecting the difficuty of constructing datasets, we can highlight the imbalanced distribution of the classes in the data, where the occurence of bug-free code are much higher than defective one. Usually, there are

| Dataset | Content | Size | Used in tasks |
|---|---|---|---|
| SEIP Lab Software Defect Prediction Data | Complexity metrics | 5 subsequent releases of 3 projects from the Java Eclipse community | Data collection and linking |
| PROMISE Software Engineering Repository | Numeric metrics; reported defects (false/true) | 15 000 modules | Defect prediction |
| NASA Defect Dataset | Numeric metrics; reported defects (false/true) | 51 000 modules | Defect prediction |
| GPHR | Java code and metrics | 3526 pairs of fragments, buggy and fixed, code metrics | Defect prediction |
| BugHunter dataset | Hand-crafted metrics; fix-inducing commit; number of reported bugs | 159k pairs for 3 granularity levels (file/class/method), 15 projects | Analyzing the importance of complexity metrics |
| Neural Code Translator Dataset | Pairs of buggy and fixed abstracted method-level fragments | 46k pairs of small fragments (under 50 tokens), 50k pairs of medium fragments (under 100 tokens) | Code refinement |
| BugsInPy | Pair of buggy and fixed Python snippets, manually processed | 493 bugs from 17 projects | Benchmark for testing and debugging tools |

Table 2: List of labeled datasets for defect prediction

fewer defective files or methods in a project than clean ones. In such case, most of conventional and basic classification algorithms tend to classify correctly the major class, which is defect-free code in our case, and ignore the smaller class of defect-prone code. Consequently, this will lead the classifier to poor performance.

To tackle this problem, several oversampling methods are proposed. In works [4, 1] the authors constructed hybrid approaches based on the Synthetic Minority Over-Sampling Technique (SMOTE and SMOTUNED) and ensemble classifiers for detecting software defects in different imbalanced datasets. In [27], the authors calculate the quantity of defective and clean data in each project in the dataset and randomly duplicate parts of the smaller class to balance both classes.

### 4.2 Lack of context

Another problem is the complexity of the context for the code. Unlike the natural texts, the code element may depend on another element located far away, maybe even in another code fragment. Moreover, it is often hard to say if the code element is defective without considering its context. If datasets consists of the pairs "bugged code fragment" and "fixed code fragment", it is often hard to extract the essence of defect.

Approaches based on transformer networks were aimed to NLP problems, where data displays a great deal of locality of reference. Most information about a token can be derived from its neighboring tokens [28, 38]. Thus, most of such architectures consider the source code as a sequence of tokens. Most of Transformer architectures are designed to handle input sequences with length of 512 tokens. Therefore, their applicability to capture the context of the source code is limited.

There are several approaches to alleviate this problem. One of them is modifying the Transformer architecture, improving its ability to comprehend long context [39].

Another approach is to capture the structural and global relations on the code, combining the sequence-based and graph-based models for code representation [14, 12].

Thus, the representing the code context is essential in software defect prediction.

As the other factors affecting the diffity of the software prediction, we can highlight the imbalanced distribution of the classes in the datasets, where the bug-free codes are much higher than defective ones. Usually, there are fewer defective files or methods in a project than clean ones. In such case, most of conventional and basic classification algorithms tend to classify correctly the major class, which is defect-free code in our case, and ignore the smaller class of defect-prone code. Consequently, this will lead the classifier to poor performance.

To tackle this problem, several oversampling methods are proposed. In works [4, 1] the authors constructed hybrid approaches based on the Synthetic Minority Over-Sampling Technique (SMOTE and SMOTUNED) and ensemble classifiers for detecting software defects in different imbalanced datasets. In [27], the authors calculate the quantity of defective and clean data in each project in the dataset and randomly duplicate parts of the smaller class to balance both classes.

## 5 RQ3. What are the trends in the primary studies on the use of deep learning for software defect prediction?

Earliest works such as [36] utilize the deep learning techniques trying to extract deep features from the traditional hand-crafted features. The main drawback of this approach is that the these traditional features usually cannot capture the semantic defference between the correct and defective code. Therefore, the combination of these features would also fail to do so.

Later approaches [9, 21] use the generic or tailored deep learning techniques to extract the semantic and syntactic features directly from the source code, usually, from the abstract syntax trees. These deep learned features are used in combination with the traditional ones in the machine classifiers to produce the accurate defect prediction.

Modern software engineering puts a high emphasis on writing human-readable code. Developers tend to use meaningful identifiers and natural-language documentation in their code. As a result, source code contains substantial information that can be exploited by algorithms originally intended for Natural Language Processing (NLP), such as pre-trained language representations such as BERT [10].

Learning useful models with supervised setting is often difficult, because labeled data is usually limited. Thus, many unsupervised approaches have been proposed recently to utilize the large unlabeled datasets that are more readily available. Usually, this means that pre-training is performed with automatic supervisions without manual annotation of the samples. Then, the model may be fine-tuned for the specific task using much smaller supervised data [19].

The most recent techniques in software engineering are based on using the general-purposed pre-trained models for programming languages [20, 12]. These pre-trained models learn effective contextual representations of the source code from unlabeled datasets using self-supervised objectives. Large corpora of source code is used for pre-training. Usually, the objective are Masked Language Modeling, where at some positions tokens are masked out and model must predict the original token [11]. Utilizing these techniques alleviate the need for task-specific architectures and training on large labeled datasets for each task separately.

## 6 Conclusion

The ever-increasing scale and complexity of modern software projects produces multiple challenges. One of them is predicting potential defective code. Recent developments in the field of machine learning, especially, the deep learning, provide a powerful techniques which utilize learning algorithms for representations of the source code, capturing the semantic and structural information.

This survey presents the latest research progress in software defect prediction using the deep learning techniques, such as Deep Belief Networks, Convolutional Neural Networks, Long Short Term Memory, and Transformer architectures. We formulate the main difficulties of the defect prediction problem, such as lack of data and complexity of context and discuss the ways to alleviate these problems.

Looking at the recent trends of deep learning techniques for defect prediction, we see most potential in using the NLP approaches, such as Transformers models with self-supervised pre-training, in Software Engineering. However, it is important to consider the specifics of the source code, such as structure and long-term dependencies.

**TODO Write about our future contribution (anomalies detection & dataset).**

## References

[1] Amritanshu Agrawal and Tim Menzies. “Is ”Better Data” Better than ”Better Data Miners”? On the Benefits of Tuning SMOTE for Defect Prediction”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 1050–1061. ISBN: 9781450356381. DOI: 10.1145/3180155.3180197. URL: https://doi.org/10.1145/3180155.3180197.

[2] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3212695. URL: https://doi.org/10.1145/3212695.

[3] Uri Alon et al. "Code2vec: Learning Distributed Representations of Code". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290353. URL: https://doi.org/10.1145/3290353.

[4] Hamad Alsawalqah et al. "Hybrid SMOTE-Ensemble Approach for Software Defect Prediction". In: *Software Engineering Trends and Techniques in Intelligent Systems*. Ed. by Radek Silhavy et al. Cham: Springer International Publishing, 2017, pp. 355–366. ISBN: 978-3-319-57141-6.

[5] Timofey Bryksin et al. "Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 455–465. ISBN: 9781450375177. DOI: 10.1145/3379597.3387447. URL: https://doi.org/10.1145/3379597.3387447.

[6] Z. Cai, L. Lu, and S. Qiu. "An Abstract Syntax Tree Encoding Method for Cross-Project Defect Prediction". In: *IEEE Access* 7 (2019), pp. 170844–170853. DOI: 10.1109/ACCESS.2019.2953696.

[7] Zimin Chen and Martin Monperrus. *A Literature Study of Embeddings on Source Code*. 2019. arXiv: 1904.03061 [cs.LG].

[8] H. K. Dam et al. "Automatic Feature Learning for Predicting Vulnerable Software Components". In: *IEEE Transactions on Software Engineering* 47.1 (2018), pp. 67–85. DOI: 10.1109/TSE.2018.2881961.

[9] H. K. Dam et al. "Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 46–57. DOI: 10.1109/MSR.2019.00017.

[10] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://www.aclweb.org/anthology/N19-1423.

[11] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL].

[12] Daya Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. 2021. arXiv: 2009.08366 [cs.SE].

[13] Andrew Habib and Michael Pradel. *Neural Bug Finding: A Study of Opportunities and Challenges*. 2019. arXiv: 1906.00307 [cs.SE].

[14] Vincent J Hellendoorn et al. "Global Relational Models of Source Code". In: *International Conference on Learning Representations (ICLR)*. 2020.

[15] T. Hoang et al. "DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 34–45. DOI: 10.1109/MSR.2019.00016.

[16] Jack Humphreys and Hoa Khanh Dam. "An Explainable Deep Model for Defect Prediction". In: *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. RAISE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 49–55.

[17] "IEEE Standard Classification for Software Anomalies". In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), pp. 1–23. DOI: 10.1109/IEEESTD.2010.5399061.

[18] Aditya Kanade et al. *Learning and Evaluating Contextual Embedding of Source Code*. 2020. arXiv: 2001.00059 [cs.SE].

[19] Aditya Kanade et al. "Learning and Evaluating Contextual Embedding of Source Code". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 5110–5121. URL: http://proceedings.mlr.press/v119/kanade20a.html.

[20] Rafael - Michael Karampatsis and Charles Sutton. *SCELMo: Source Code Embeddings from Language Models*. 2020. arXiv: 2004.13214 [cs.SE].

[21] J. Li et al. "Software Defect Prediction via Convolutional Neural Network". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, pp. 318–328. DOI: 10.1109/QRS.2017.42.

[22] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL].

[23] A. V. Phan and M. Le Nguyen. "Convolutional neural networks on assembly code for predicting software defects". In: *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*. 2017, pp. 37–42. DOI: 10.1109/IESYS.2017.8233558.

[24] Shaojian Qiu et al. "Cross-Project Defect Prediction via Transferable Deep Learning-Generated and Handcrafted Features." In:

[25] Y. Qiu et al. "Automatic Feature Exploration and an Application in Defect Prediction". In: *IEEE Access* 7 (2019), pp. 112097–112112. DOI: 10.1109/ACCESS.2019.2934530.

[26] S. Sharmin et al. "SAL: An effective method for software defect prediction". In: *2015 18th International Conference on Computer and Information Technology (ICCIT)*. 2015, pp. 184–189. DOI: 10.1109/ICCITechn.2015.7488065.

[27] Ke Shi et al. "PathPair2Vec: An AST path pair-based code representation method for defect prediction". In: *Journal of Computer Languages* 59 (2020), p. 100979. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2020.100979. URL: http://www.sciencedirect.com/science/article/pii/S2590118420300393.

[28] Yi Tay et al. *Efficient Transformers: A Survey*. 2020. arXiv: 2009.06732 [cs.LG].

[29] Haonan Tong, Bin Liu, and Shihai Wang. "Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning". In: *Information and Software Technology* 96 (2018), pp. 94–111. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.11.008. URL: https://www.sciencedirect.com/science/article/pii/S0950584917300113.

[30] H. D. Tran, L. T. M. Hanh, and N. T. Binh. "Combining feature selection, feature learning and ensemble learning for software fault prediction". In: *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*. 2019, pp. 1–8. DOI: 10.1109/KSE.2019.8919292.

[31] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[32] S. Wang, T. Liu, and L. Tan. "Automatically Learning Semantic Features for Defect Prediction". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 297–308. DOI: 10.1145/2884781.2884804.

[33] S. Wang et al. "Deep Semantic Feature Learning for Software Defect Prediction". In: *IEEE Transactions on Software Engineering* 46.12 (2018), pp. 1267–1293. DOI: 10.1109/TSE.2018.2877612.

[34] 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* (2020), pp. 1–13. DOI: 10.1109/TR.2020.3040191.

[35] Zhou Xu et al. "LDFR: Learning deep feature representation for software defect prediction". In: *Journal of Systems and Software* 158 (2019), p. 110402. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.110402. URL: https://www.sciencedirect.com/science/article/pii/S0164121219301761.

[36] X. Yang et al. "Deep Learning for Just-in-Time Defect Prediction". In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 17–26. DOI: 10.1109/QRS.2015.14.

[37] Xinli Yang et al. "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction". In: *Information and Software Technology* 87 (2017), pp. 206–220. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.03.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584917302501.

[38] Manzil Zaheer et al. *Big Bird: Transformers for Longer Sequences*. 2021. arXiv: 2007.14062 [cs.LG].

[39] Manzil Zaheer et al. *Big Bird: Transformers for Longer Sequences*. 2021. arXiv: 2007.14062 [cs.LG].

[40] Linchang Zhao et al. "Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks". In: *Neurocomputing* 352 (2019), pp. 64–74. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2019.03.076. URL: https://www.sciencedirect.com/science/article/pii/S0925231219305004.

[41] Tianchi Zhou et al. "Improving defect prediction with deep forest". In: *Information and Software Technology* 114 (2019), pp. 204–216. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2019.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0950584919301466.