Review

A Survey on Software Defect Prediction Using Deep Learning

Elena N. Akimova [1,2,*], Alexander Yu. Bersenev [1,2], Artem A. Deikov [1,2], Konstantin S. Kobylkin [1,2], Anton V. Konygin [1], Ilya P. Mezentsev [1,2], and Vladimir E. Misilov [1,2]

[1] Krasovskii Institute of Mathematics and Mechanics, Ural Branch of RAS, S. Kovalevskaya Street 16, Ekaterinburg, Russia; Alexander.Bersenev@urfu.ru (A.Yu.B.); hx0day@hackerdom.ru (A.A.D); kobylkinks@gmail.com (K.S.K); konygin@imm.uran.ru (A.V.K.); ilyamezentcev@gmail.com (I.P.M); v.e.misilov@urfu.ru (V.E.M.)

[2] Ural Federal University, Mira Street 19, Ekaterinburg, Russia;

* Correspondence: aen15@yandex.ru

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 identify the defective source code with high accuracy. Developing fault prediction model is a challenging problem, and many approaches have been proposed through the history. The recent breakthrough in machine learning technologies, especially, the development of deep learning techniques, led to the fact that many problems can be solved by these methods. Our survey focuses on the deep learning techniques for defect prediction. We analyse the recent works on the topic, study the methods for automatical learning of the semantic and structural features from the code, discuss the open problems, and present the recent trends in the field.

Keywords: Defect prediction; Anomaly detection; Program analysis; Code understanding; Neural Networks; Deep Learning

MSC: 68N30; 68T07

1. Introduction

According to the IEEE Standard Classification for Software Anomalies [1], a 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. Common ways for finding the software defects are the manual testing and code review. The main drawback of these methods is that they are quite expensive in terms of time and effort. The automatic approaches to the Software Defect Prediction (SDP) would allow one to recude the costs and improve quality of the software projects.

Thus, Software Defect Prediction is an important problem in the fields of the software engineering and programming language research. The task is to identify the defective code with high accuracy (in terms of the precision and recall).

The development and breakthrough of machine learning led to the fact that many tasks can be solved by the these methods.

Recent advances in the fields of artificial neural networks and machine learning, as well as, the increasing power of the modern computers (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.

For the problem of software defect prediction, the researchers have proposed the representation-learning algorithms to learn semantic representations of programs au-

Citation: Akimova, E. N.; Bersenev, A. Yu.; Deikov, A. A.; Kobylkin, K. S.; Konygin, A. V.; Mezentsev, I. P.; Misilov, V. E. A Survey on Software Defect Prediction Using Deep Learning. *Mathematics* **2021**, *1*, 0. https://doi.org/

Academic Editor:

Received:
Accepted:
Published:

tomatically and use this representation to identify the defect-prone code. Using these implicit features shows better results then the previous approaches based on the explicit features, such as the code metrics [2].

The software defect prediction is a rapidly developing field, and the state-of-the-art surveys on the topic [3–5] do not sufficiently cover the recent works describing the cutting-edge techniques. For example, recent advances in the related fields of Natural Language Processing (NLP) provided the new powerful tools such as Transformer language models. These techniques were later successfully applied to the software engineering tasks.

The goal of our survey is to describe these latest achievements taking into account the newest primary studies published in 2019–2021. We hope that this survey can be useful for researchers and practitioners in the software defect prediction, code understanding, and other related fields.

Some semantic defects are hard to find using only source code. For example, in [6], 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 [7] where to expose the program behavior, the assembly code (generated from the C source code by the compiler) is used to learn the defect features.

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

1. Prepare the dataset by collecting the source code samples from repositories of the software projects (or choose the suitable existing dataset).
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 the quality metrics.

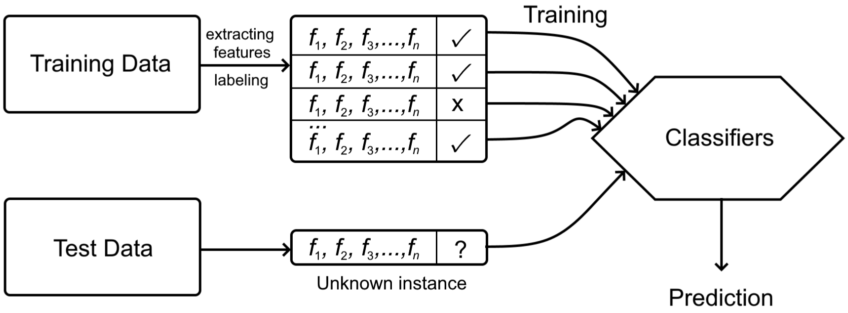
Training Data
extracting features
labeling
$f_1, f_2, f_3,\ldots,f_n$ ✓
$f_1, f_2, f_3,\ldots,f_n$ ✓
$f_1, f_2, f_3,\ldots,f_n$ x
...
$f_1, f_2, f_3,\ldots,f_n$ ✓
Training
Classifiers
Test Data
$f_1, f_2, f_3,\ldots,f_n$?
Unknown instance
Prediction

Figure 1. Scheme of the the process of constructing the defect prediction model.

The survey is structured as follows. Section 2 briefly describes the methodology of our survey. Section 3 presents the overview on the various deep learning techniques applied to the defect prediction. In Section 4, we outline the main difficulties of the problem. Section 5 presents the study of the latest trends in the techniques and methods for defect prediction. Section 6 concludes the study and offers our vision on the future developments on the field.

2. Methodology

We reviewed the primary studies 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 to the 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 use of deep learning for the software defect prediction?

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 keywords "software engineering", "deep learning", and "defect prediction".

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 to defect prediction problem
- The paper length must not be less than 6 pages

3. RQ1. What techniques have been applied to this problem?

In order to work with the source code, we need to have its representation. On the one hand, this 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. The numerical vector representing the source code is called an “embedding”.

There are different ways to represent the source code. Moreover, we need different granularities for different tasks, *e.g.*, for code completion we need token-level embedding and for function clone detection we need function embedding. For the software defect prediction problem, various levels of granularity are used, such as sub-system, component, file/class, method, and change (see [8,9] for more info on various code embeddings).

One way is to create the vector from the hand crafted features. This approach assumes that an expert invents a set of features and selects best of them, (e.g., [10,11]). Usually, these features include the statistical characteristics of code, such as its size, code complexity, code churn, or process metrics.

Other way is to create the numerical vector by processing the source code.

One way to represent the code is a sequence of elements. Usually, they are code tokens or characters [12]. The neural networks based on the sequences are usually trained to predict the subsequent element.

Another approach to build the representation of the source code is the abstract syntax trees (AST) [13]. The nodes of the tree corresponds to the statement and operators, and the leaves represent the operands and values. The tree-based models are trained to predict the code by generating new nodes taking into account the existing tree structure.

The most common approach to defect prediction is to use some classification algorithm to divide the source code into two categories: defect code and correct one (*e.g.* [14]).

However, the approaches based on the hand-crafted features usually do not sufficiently capture the syntax and semantics of the source code. Most traditional code metrics cannot distinguish code fragments if these fragments have the same structure and complexity but implement different functionality. For example, if we switch several lines in the code fragments, traditional features, such as the number of lines of code, number of function calls, number of tokens would remain the same (see [2]). Thus, the semantic information is more important for defect prediction than these metrics.

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 the 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 [15] generative models are based on a multilevel neural network. This network contains one input layer, one output layer, and multiple hidden layers. The output layer generates a feature vector representing the data fed to the input layer. Each layer consists of the stochastic nodes. The important feature of the DBN is that the nodes are only connected to the nodes in the adjacent layers, but not to the nodes within the same layer as shown in Fig. 2.

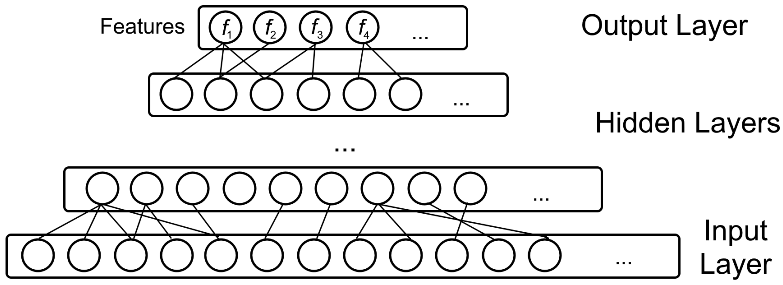

Figure 2. Architecture of the Deep Belief Network.

Perhaps one of the first works combining AST with the deep learning is [16]. The authors propose the approach for software defect prediction on changes level. The DBN (which is fed by the traditional code metrics) generates the new expressive features and use them in a classical machine learning classifiers. They extract the relations from the traditional code metrics, such as number of modified modules, directories, and files, added and deleted lines, and several features related to developer's experience. Later the authors have proposed the "TLEL" approach [17] based on the decision tree and ensemble learning for classification.

The works of Wang et al. [2,18] also use the DBN, but in a different manner. For predicting the defects on the basis of the code semantics, the authors have developed a DBN to automatically learn a semantic features from the source code. As the input for the network, the programs' AST and source code changes are used for the cases of file-level and change-level prediction, respectively. Then, the authors use the classical machine learning classifiers and extracted features to classify source code files whether they are buggy or clean.

The main drawback of the DBN is that it does not sufficiently capture the context of the code elements, such as the order of statement execution and function calls.

3.2. Long Short Term Memory

The Long Short Term Memory [19] is a subtype of the recurrent neural network specialized for processing the data sequences. The LSTM network consists of LSTM units (see Fig. 3). The key element of the unit is a memory cell, which allows the unit to store the values for a short, as well as, for a long time intervals. This provides the LSTM-based models the ability to capture the long-range context information from the source code.

The LSTM-based model was used in work [11] for learning both semantic and syntactic features of code. 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

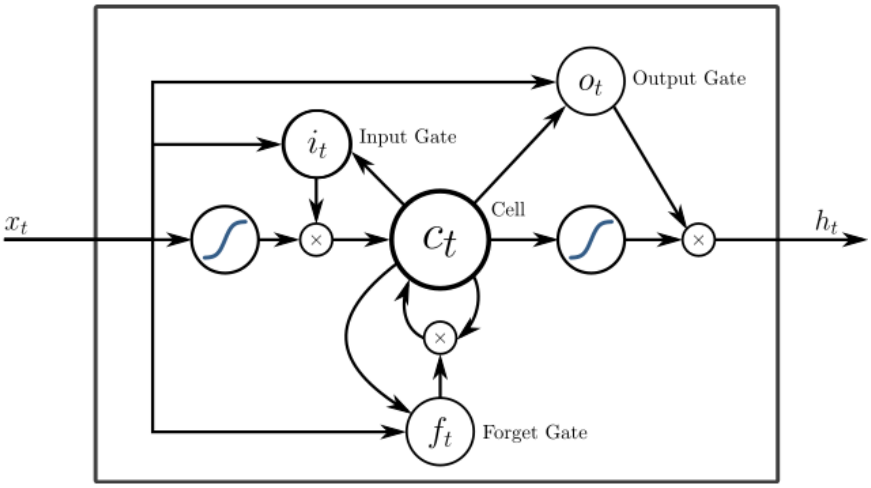

Figure 3. Scheme of the LSTM unit.

a token state representing the semantic information of the token. Later the Tree-LSTM model was developed using the AST representation as input [20].

A neural bug finding technique is proposed in [21]. The authors train a neural network on examples of the defective and correct code, and then use the resulting binary classifier for bug detection. To prepare a labeled dataset, the authors use the existing static bug detection software to identify the specific kind of bugs. 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 network with LSTM is used as model.

In [22], the authors propose a model for defect prediction on the base of AST path pair representation. To process the code, the path in the AST is extracted as combination of symbol sequence and control sequence. These sequences are fed to a Bi-LSTM network to generate a path vector. Then, all the vectors are combined using the global attention technique to generate the vector for the entire code fragment. These final embedding representations are used for classification.

3.3. Convolutional Neural Networks

The Convolutional Neural Networks [23] are a type of neural network specialized for processing the data with a mesh-like structure. This network is characterized by two important features. Firstly, the local connection pattern between the units is repeated over the entire network. It allows the network to capture the short-term structural context of the source code. Secondly, the each unit have the same parameters. It allows the network to learn the information on the code element irrespective of its position in the code. The scheme of general CNN is shown in Fig. 4.

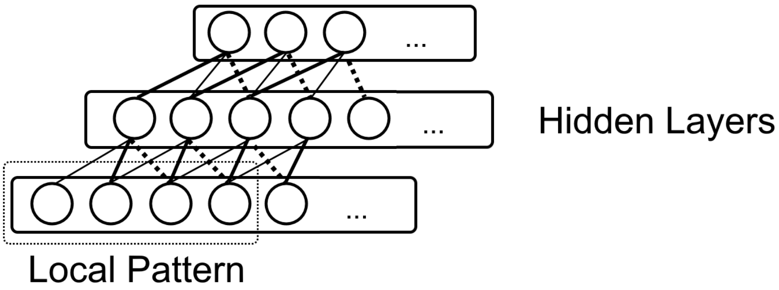

Figure 4. Architecture of the Convolutional Neural Network.

The work [24] presents the model based on the CNN architecture. Based on the program's AST, the token vectors are extracted and converted to numerical vectors. Then, these vectors are fed into a CNN. After that, the combination of the extracted semantic and structural features and code metrics is used for software defect prediction applying the logistic regression.

A deep learning model to predict defects on the basis of the commit messages and code changes is developed in work [25]. This model is based on the CNN. It uses the convolutional network layers for processing the code changes and commit text and the feature combination layer to fuse these two embedding vectors into a single one.

Another deep learning-based model for defect prediction is proposed in work [26]. The training of the neural network utilizes the triplet loss technique and the weighted cross-entropy loss technique. The random forest is used as a classifier.

In [27], the features learning technique based on CNN is proposed. This model extract features from token vectors in the AST of the code and learns the transferable joint features. Combining these deep-learning-generated features with the hand-crafted ones allows the model to perform the cross-project defect prediction. Later the authors propose a new tree-based convolutional network to perform this task [28]. It uses the tree-based continuous bag-of-word for encoding the AST nodes to be fed into CNN.

3.4. *Transformer models*

In recent time, the big success of pre-trained contextual representations in the NLP, *e.g.*, [29], led to a rise of attempts to apply these techniques to source code. Usually, these models are based on the multi-layer Transformer architecture [30] shown in Fig. 5. They are pre-trained using the massive unlabeled corpora of programs with the self-supervised objectives, such as masking language modeling and next sentence prediction [31,32]. After the pre-training phase, the model can be fine-tuned for specific tasks using the supervised techniques.

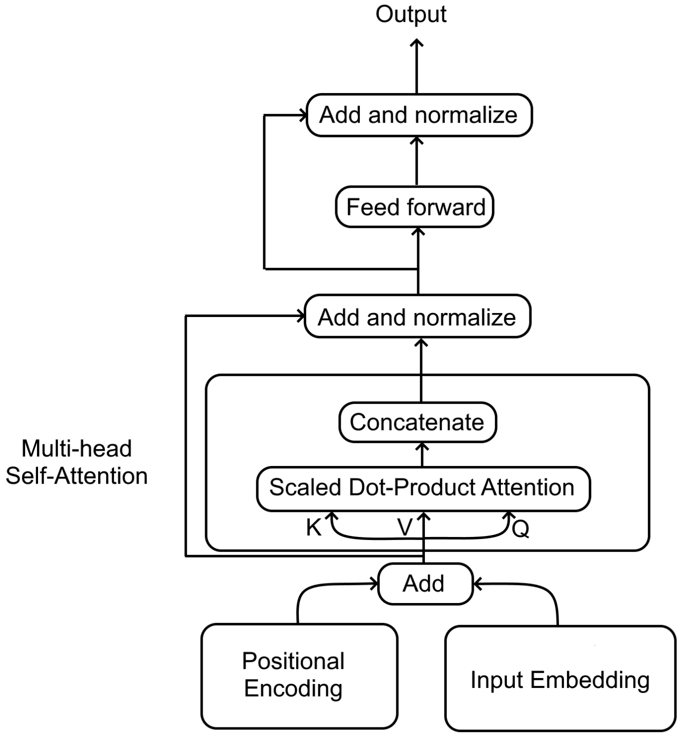

Figure 5. Architecture of the multi-layer transformer.

The authors of work [33] state 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 of learning features from the source code does not guarantee capturing semantic and syntactical similarity, and very similar source code can have very different features. These features can correlate with defects rather than directly cause them. In contrast, the authors propose the approach based on the self attention transformer encoder to the 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 technique. The resulting model provides the defect prediction with the semantic highlight of defective code regions.

The CuBERT model is presented in work [31]. The authors use a corpus of Python files from the GitHub to create a benchmark for evaluating code embeddings on five classification tasks and a program repair task. They train their model and compare it with various other models including the BiLSTM and Transformer. It is shown that the CuBERT outperforms the baseline models consistently.

A bimodal language model called CodeBERT is presented in work [32]. It is based on the multilayer bidirectional Transformer neural architecture. To prepare the data, the natural language text is represented as a sequence of words, and the source code is presented as a sequence of tokens. The output of the CodeBERT model is a contextual vector learned from the natural language and source code, as well as, the aggregated sequence. The resulting model efficiently solves the problems of both code to the documentation and natural language code search.

Work [34] presents a multi-layer bidirectional transformer architecture GraphCodeBERT, which utilizes three components as input: the source code, paired comments, and data flow graph. Data flow graph represents relations between variables, for example, where the value of a variable comes from. This allows the model to consider the 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 graph were used. It supports several downstream code-related tasks including the code clone detection, code translation, and code refinement.

3.5. Other Networks

In [35], a software defect prediction technique based on stacked denoising autoencoders model is presented. The stacked denoising autoencoder is used to extract higher-level features from the traditional metrics. The two-stage ensemble learning is used for classification. To address the class imbalance, the authors use the ensemble learning strategy. Later, the feature selection algorithm was applied to this method to address the feature renundancy problem [36].

A model for the software defect prediction was constructed in work [37] on the base of the Siamese parallel fully-connected networks. This model utilizes the paired parallel Siamese networks architecture and the deep learning approach. The network produces the high-level features that are used for classification. To address the imbalance between the minority and majority classes, the network takes into account the cost-sensitivity features.

The neural forest networks are used to learn feature representations in work [38]. To perform classification, a decision forest is used. It also guides the learning of the neural network. In [39], a new deep forest model is proposed for the software defect prediction. To detect the essential defect features, it uses the cascade learning strategy, which consists in reforming a set of the random forest classifiers into a layered network.

The graph neural network to predict the software defects is constructed in work [40]. It extracts the semantics and context features from the AST of the code fragments. To capture the defect-related information from the source code, the ASTs for the 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?

The problem of software defect prediction is considered very complex and very challenging for the machine learning models based on the neural networks.

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 the 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.

Table 1. List of unlabeled datasets.

Dataset	Content	Size	Used in tasks
Bigquery github repos [41]	Python source code	4M files	Pre-training CuBERT model
Py150 [42]	Python source code, AST	8 423 repos, 149 993 files	Fine-tuning CuBERT model
Js150 [43]	Javascript source code, AST	150 000 source files	Code Summarization; Defect Prediction
Datasets for [44]	Java source code	9 500 projects, 16M samples in the largest one	Code summarization
GitHub Java Corpus [45]	Java source code	11 000 projects	Language Modeling
CodeNN Dataset [46]	C# source code and summaries	66 015 fragments	Code Captioning
Dataset for [6]	Kotlin source code, AST, bytecode	47 751 repos, 932 548 files, 4 044 790 functions	Anomaly detection, defect prediction
Dataset for [47]	C# source code	29 projects, 2.9M lines of code	Variable Misuse detection

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

Table 2. List of labeled datasets.

Dataset	Content	Size	Used in tasks
SEIP Lab Software Defect Prediction Data [48]	Complexity metrics	5 subsequent releases of 3 projects from the Java Eclipse community	Data collection and linking
PROMISE Software Engineering Repository [49]	Numeric metrics; reported defects (false/true)	15 000 modules	Defect prediction
NASA Defect Dataset [50]	Numeric metrics; reported defects (false/true)	51 000 modules	Defect prediction
REPD datasets [51]	Numeric metrics, semantic features, reported defects	10885 fragments in the largest one	Defect prediction
GPHR [52]	Java code and metrics	3526 pairs of fragments, buggy and fixed, code metrics	Defect prediction
BugHunter[53]	Java source code; 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
GitHub Bug DataSet [54]	Java source code; code metrics; number of reported bugs and vulnerabilities	15 projects; 183k classes	Bug prediction
Unified Bug Dataset [55]	Java source code; code metrics; number of reported bugs	47 618 classes; 43 744 files	Bug prediction
Neural Code Translator Dataset [56]	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 [57]	Pair of buggy and fixed Python snippets, manually processed	493 bugs from 17 projects	Benchmark for testing and debugging tools

Dataset	**Content**	**Size**	**Used in tasks**
Draper VDISC Dataset [58]	C and C++ source code, labeled for potential vulnerabilities	1.27M functions	Vulnerability Detection
Refactory Dataset [59]	Python source code	2 442 correct and 1 783 buggy program	Program repair
Defect4J [60]	Java source code	835 pairs of buggy and fixed fragments	Software testing research
BugSwarm[61]	Java and Python source code	3 232 pairs of buggy and fixed fragments	Software testing research
BuGL [62]	C, C++, Java, and Python source code; issues; pull requests	54 projects; 151k closed issues; 10 187 pull requests	Bug localization
Bugs.jar [63]	Java source code	1 158 pairs of buggy and and fixed fragments	Program repair

As the other factors affecting the difficuty of constructing datasets, we can highlight that the distribution of the classes in the real code projects is often imbalanced. Usually, there are fewer buggy files or methods in a project than the correct ones. This may lead to the situation where the common classifiers would correctly detect the major class (correct code) and ignore the much smaller class of the defect-prone code. This will lead to bad performance of the model.

To address this imbalance, several oversampling methods are proposed. In works [64, 65], the authors constructed hybrid approaches. It is based on the Synthetic Minority Over-Sampling Technique (SMOTE and SMOTUNED) for preparing the datasets and ensemble approaches for classifying the defective and correct code. In [22], the authors takes into account the proportion of the correct and defective code in each project in the dataset. To balance the classes, they duplicate the elements of the smaller class.

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 dataset consists of the pairs of bugged and fixed code fragments, it is often hard to extract the essence of defect.

Approaches based on the Transformer networks were aimed to NLP problems where data display a great deal of locality of reference. Most information about a token can be derived from its neighboring tokens [66]. Thus, most of such models represent the source code as a sequence of tokens.

The traditional Transformer architectures based on self-attention matrices do not scale well because of quadratic complexity. Usually, they are designed to handle the input sequences with limited length (usually, 512 or 1024 tokens) [66,67]. Therefore, their applicability to understand the context of the source code is limited.

There are several modifications to the Transformer architecture that improve its ability to comprehend long sequences [68–70]. These approaches alleviate the problem of limited length of the input, giving the Transformers a potential to work with a complex context of the source code.

Another approach is to capture the structural and global relations on the code, combining the sequence-based and graph-based models for code representation [34,71].

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

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

Earliest works such as [16] utilize the deep learning techniques trying to extract the implicit features from the traditional explicit features (such as code metrics). The main drawback of this approach is that these traditional features usually cannot capture the semantic difference between the correct and defective code. Therefore, the combination of these features would also fail to do this [24].

Later approaches [20,25] 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 development often prioritize writing the human-readable source code. This includes using the meaningful names for the functions and variables and writing the code documentation in natural language. This leads to the situation where we can extract the semantic information from the source code using the techniques originally intended for the NLP, such as the pre-trained language representations such as BERT [72].

Learning useful models with supervised setting is often difficult because labeled data are 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 [41].

The most recent techniques in software engineering are based on using the general-purposed pre-trained models for programming languages [34,73]. These models learn to "understand" the source code from unlabeled datasets using the self-supervised objectives. Large corpus of source code is used for pre-training. Usually, the objective is the Masked Language Modeling where at some positions the tokens are masked out and the model must predict the original token [32]. Utilizing these techniques alleviates the need for the task-specific architectures and training on large labeled datasets for each task separately.

6. Conclusions

One of the major challenges in the modern software engineering is predicting defective code. Recent developments in the field of machine learning, especially the multi-layered neural networks and deep learning algorithms, provide powerful techniques, which utilize learning algorithms for representations of the source code that captures the semantic and structural information.

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

Taking into account the latest trends in the machine learning techniques for the software defect prediction problem, we believe that progress in this field will be achieved largely due to the implementation of the following ideas.

- To reduce the requirements for the size of the labeled datasets, one should use the self-supervised training on large corpora of the unlabeled data. In addition, it is necessary to use the unlabeled data for the pre-training of related tasks and to contribute to the fact that the trained models will have a deeper and more comprehensive understanding of the source code. This, in the turn, will allow one to find the deeper defects.

- To leverage the latest advances in the machine learning techniques in the natural language processing in the programming languages, we are already seeing the successful migration of these methods to solve various code understanding problems. For example, optimization of the self-attention mechanism for the transformers will allow one to use them for long sequences, which, in the turn, will lead to a more complete consideration of the code context for finding the defects.
- Often a defect is not limited to a single line of code or one function, and there are various ways to fix it. For example, a bug can be fixed either inside the function or at calling this function. Thus, the defect ceases to have specific coordinates inside the source file. In addition, not being an explicit defect, a line of code can become defective at a certain point in time. A changed context may lead to the fact that the purpose of the code changes, and, therefore, the old implementation no longer corresponds to the new requirements or specifications.

All this leads to a blurring of the concept of a defect. Thus, we come to the concepts of “potentially defective” code or “strange” code. In this regard, as promising problems, we want to note the task of finding an atypical (or anomalous) code and the task of the code refinement. These task require good representations of the code and code changes, taking into account the specifics of the source code, such as structure and context.

It is difficult to state which of the state-of-the-art models performs in the best way. There are no universally accepted standard benchmarks for the problem and different researchers utilize different performance metrics and use different data. Thus, the experimental results from the primary works cannot be directly compared. The existing comparative studies such as [74] show that while the state-of-the-art deep learning techniques usually perform better than standard deep learning and traditional metrics-based ones (achieving the increase of F1 from 60% up to 80% in some cases). None of the approaches achieves a consistently high performance in terms of recall, precision, and accuracy sufficient for the practical application. Thus, the defect prediction problem remains an open one.

Author Contributions: All authors (E.N.A., A.Yu.B., A.A.D., K.S.K., A.V.K., I.P.M., V.E.M.) contributed equally. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AST	Abstract Syntax Tree
CNN	Convolutional Neural Network
DBN	Deep Belief Network
DL	Deep Learning
LSTM	Long Short Term Memory
NLP	Natural Language Processing
SDP	Software Defect Prediction

References

1. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* **2010**, pp. 1–23. doi:10.1109/IEEESTD.2010.5399061.
2. Wang, S.; Liu, T.; Tan, L. Automatically Learning Semantic Features for Defect Prediction. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 297–308. doi:10.1145/2884781.2884804.

3. Omri, S.; Sinz, C. Deep Learning for Software Defect Prediction: A Survey. Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops; Association for Computing Machinery: New York, NY, USA, 2020; ICSEW'20, p. 209–214. doi:10.1145/3387940.3391463.
4. Yang, Y.; Xia, X.; Lo, D.; Grundy, J. A Survey on Deep Learning for Software Engineering, 2020, [arXiv:cs.SE/2011.14597].
5. Shen, Z.; Chen, S. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques. *Security and Communication Networks* **2020**, *2020*, 8858010. doi:10.1155/2020/8858010.
6. Bryksin, T.; Petukhov, V.; Alexin, I.; Prikhodko, S.; Shpilman, A.; Kovalenko, V.; Povarov, N. Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler. Proceedings of the 17th International Conference on Mining Software Repositories; Association for Computing Machinery: New York, NY, USA, 2020; MSR '20, p. 455–465. doi:10.1145/3379597.3387447.
7. Phan, A.V.; Le Nguyen, M. Convolutional neural networks on assembly code for predicting software defects. 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), 2017, pp. 37–42. doi:10.1109/IESYS.2017.8233558.
8. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* **2018**, *51*. doi:10.1145/3212695.
9. Chen, Z.; Monperrus, M. A Literature Study of Embeddings on Source Code, 2019, [arXiv:cs.LG/1904.03061].
10. Sharmin, S.; Arefin, M.R.; Wadud, M.A.; Nower, N.; Shoyaib, M. SAL: An effective method for software defect prediction. 2015 18th International Conference on Computer and Information Technology (ICCIT), 2015, pp. 184–189. doi:10.1109/ICCITechn.2015.7488065.
11. Dam, H.K.; Tran, T.; Pham, T.; Ng, S.W.; Grundy, J.; Ghose, A. Automatic Feature Learning for Predicting Vulnerable Software Components. *IEEE Transactions on Software Engineering* **2018**, *47*, 67–85. doi:10.1109/TSE.2018.2881961.
12. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space, 2013, [arXiv:cs.CL/1301.3781].
13. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 783–794. doi:10.1109/ICSE.2019.00086.
14. Pradel, M.; Sen, K. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* **2018**, *2*. doi:10.1145/3276517.
15. Bengio, Y. Learning Deep Architectures for AI. *Found. Trends Mach. Learn.* **2009**, *2*, 1–127. doi:10.1561/2200000006.
16. Yang, X.; Lo, D.; Xia, X.; Zhang, Y.; Sun, J. Deep Learning for Just-in-Time Defect Prediction. 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 17–26. doi:10.1109/QRS.2015.14.
17. Yang, X.; Lo, D.; Xia, X.; Sun, J. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* **2017**, *87*, 206–220. doi:https://doi.org/10.1016/j.infsof.2017.03.007.
18. Wang, S.; Liu, T.; Nam, J.; Tan, L. Deep Semantic Feature Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering* **2018**, *46*, 1267–1293. doi:10.1109/TSE.2018.2877612.
19. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Comput.* **1997**, *9*, 1735–1780. doi:10.1162/neco.1997.9.8.1735.
20. Dam, H.K.; Pham, T.; Ng, S.W.; Tran, T.; Grundy, J.; Ghose, A.; Kim, T.; Kim, C.J. Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 46–57. doi:10.1109/MSR.2019.00017.
21. Habib, A.; Pradel, M. Neural Bug Finding: A Study of Opportunities and Challenges, 2019, [arXiv:cs.SE/1906.00307].
22. Shi, K.; Lu, Y.; Chang, J.; Wei, Z. PathPair2Vec: An AST path pair-based code representation method for defect prediction. *Journal of Computer Languages* **2020**, *59*, 100979. doi:https://doi.org/10.1016/j.cola.2020.100979.
23. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press, 2016. http://www.deeplearningbook.org.
24. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software Defect Prediction via Convolutional Neural Network. 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp. 318–328. doi:10.1109/QRS.2017.42.
25. Hoang, T.; Khanh Dam, H.; Kamei, Y.; Lo, D.; Ubayashi, N. DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 34–45. doi:10.1109/MSR.2019.00016.
26. Xu, Z.; Li, S.; Xu, J.; Liu, J.; Luo, X.; Zhang, Y.; Zhang, T.; Keung, J.; Tang, Y. LDFR: Learning deep feature representation for software defect prediction. *Journal of Systems and Software* **2019**, *158*, 110402. doi:https://doi.org/10.1016/j.jss.2019.110402.
27. Qiu, S.; Lu, L.; Cai, Z.; Jiang, S. Cross-Project Defect Prediction via Transferable Deep Learning-Generated and Handcrafted Features.
28. Cai, Z.; Lu, L.; Qiu, S. An Abstract Syntax Tree Encoding Method for Cross-Project Defect Prediction. *IEEE Access* **2019**, *7*, 170844–170853. doi:10.1109/ACCESS.2019.2953696.
29. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach, 2019, [arXiv:cs.CL/1907.11692].
30. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention Is All You Need, 2017, [arXiv:cs.CL/1706.03762].
31. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code, 2020, [arXiv:cs.SE/2001.00059].

32. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; Zhou, M. CodeBERT: A Pre-Trained Model for Programming and Natural Languages, 2020, [arXiv:cs.CL/2002.08155].
33. Humphreys, J.; Dam, H.K. An Explainable Deep Model for Defect Prediction. Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. IEEE Press, 2019, RAISE '19, p. 49–55.
34. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; Tufano, M.; Deng, S.K.; Clement, C.; Drain, D.; Sundaresan, N.; Yin, J.; Jiang, D.; Zhou, M. GraphCodeBERT: Pre-training Code Representations with Data Flow, 2021, [arXiv:cs.SE/2009.08366].
35. Tong, H.; Liu, B.; Wang, S. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology* **2018**, *96*, 94–111. doi:https://doi.org/10.1016/j.infsof.2017.11.008.
36. Tran, H.D.; Hanh, L.T.M.; Binh, N.T. Combining feature selection, feature learning and ensemble learning for software fault prediction. 2019 11th International Conference on Knowledge and Systems Engineering (KSE), 2019, pp. 1–8. doi:10.1109/KSE.2019.8919292.
37. Zhao, L.; Shang, Z.; Zhao, L.; Zhang, T.; Tang, Y.Y. Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks. *Neurocomputing* **2019**, *352*, 64–74. doi:https://doi.org/10.1016/j.neucom.2019.03.076.
38. Qiu, Y.; Liu, Y.; Liu, A.; Zhu, J.; Xu, J. Automatic Feature Exploration and an Application in Defect Prediction. *IEEE Access* **2019**, *7*, 112097–112112. doi:10.1109/ACCESS.2019.2934530.
39. Zhou, T.; Sun, X.; Xia, X.; Li, B.; Chen, X. Improving defect prediction with deep forest. *Information and Software Technology* **2019**, *114*, 204–216. doi:https://doi.org/10.1016/j.infsof.2019.07.003.
40. Xu, J.; Wang, F.; Ai, J. Defect Prediction With Semantics and Context Features of Codes Based on Graph Representation Learning. *IEEE Transactions on Reliability* **2020**, pp. 1–13. doi:10.1109/TR.2020.3040191.
41. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code. Proceedings of the 37th International Conference on Machine Learning; III, H.D.; Singh, A., Eds. PMLR, 2020, Vol. 119, *Proceedings of Machine Learning Research*, pp. 5110–5121.
42. Raychev, V.; Bielik, P.; Vechev, M. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* **2016**, *51*, 731–747. doi:10.1145/3022671.2984041.
43. Raychev, V.; Bielik, P.; Vechev, M.; Krause, A. Learning Programs from Noisy Data. *SIGPLAN Not.* **2016**, *51*, 761–774. doi:10.1145/2914770.2837671.
44. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating Sequences from Structured Representations of Code, 2019, [arXiv:cs.LG/1808.01400].
45. Allamanis, M.; Sutton, C. Mining source code repositories at massive scale using language modeling. 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 207–216. doi:10.1109/MSR.2013.6624029.
46. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.
47. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs, 2018, [arXiv:cs.LG/1711.00740].
48. Mauša, G.; Galinac-Grbac, T.; Dalbelo-Bašić, B. A systematic data collection procedure for software defect prediction. *Computer Science and Information Systems* **2016**, *13*, 173–197.
49. Sayyad Shirabad, J.; Menzies, T. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.
50. Shepperd, M.; SOng, Q.; Sun, Z.; Mair, C. NASA MDP Software Defects Data Sets, 2018. doi:10.6084/m9.figshare.c.4054940.v1.
51. Afric, P.; Sikic, L.; Kurdija, A.S.; Silic, M. REPD: Source code defect prediction as anomaly detection. *Journal of Systems and Software* **2020**, *168*, 110641. doi:https://doi.org/10.1016/j.jss.2020.110641.
52. Fei, W.; Jun, A.; Jiaxi, X. Software Defect Prediction Based on Graph Representation Learning. *IEEE Transactions on Software Engineering* **2020**.
53. Ferenc, R.; Gyimesi, P.; Gyimesi, G.; Tóth, Z.; Gyimóthy, T. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software* **2020**, *169*, 110691. doi:https://doi.org/10.1016/j.jss.2020.110691.
54. Tóth, Z.; Gyimesi, P.; Ferenc, R. A Public Bug Database of GitHub Projects and Its Application in Bug Prediction. Computational Science and Its Applications – ICCSA 2016; Springer International Publishing: Cham, 2016; pp. 625–638.
55. Ferenc, R.; Tóth, Z.; Ladányi, G.; Siket, I.; Gyimóthy, T. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal* **2020**, pp. 1–60.
56. Tufano, M.; Watson, C.; Bavota, G.; Penta, M.D.; White, M.; Poshyvanyk, D. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *CoRR* **2018**, *abs/1812.08693*.
57. Widyasari, R.; Sim, S.Q.; Lok, C.; Qi, H.; Phan, J.; Tay, Q.; Tan, C.; Wee, F.; Tan, J.E.; Yieh, Y.; Goh, B.; Thung, F.; Kang, H.J.; Hoang, T.; Lo, D.; Ouh, E.L. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020; Devanbu, P.; Cohen, M.B.; Zimmermann, T., Eds. ACM, 2020, pp. 1556–1560. doi:10.1145/3368089.3417943.
58. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 2018, pp. 757–762. doi:10.1109/ICMLA.2018.00120.

59. Hu, Y.; Ahmed, U.Z.; Mechtaev, S.; Leong, B.; Roychoudhury, A. Re-factoring based Program Repair applied to Programming Assignments. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE/ACM, 2019, pp. 388–398.
60. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 437–440.
61. Tomassi, D.A.; Dmeiri, N.; Wang, Y.; Bhowmick, A.; Liu, Y.; Devanbu, P.T.; Vasilescu, B.; Rubio-González, C. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. ICSE. IEEE / ACM, 2019, pp. 339–349.
62. Muvva, S.; Rao, A.E.; Chimalakonda, S. BuGL – A Cross-Language Dataset for Bug Localization, 2020, [arXiv:cs.SE/2004.08846].
63. Saha, R.K.; Lyu, Y.; Lam, W.; Yoshida, H.; Prasad, M.R. Bugs.Jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs. Proceedings of the 15th International Conference on Mining Software Repositories; Association for Computing Machinery: New York, NY, USA, 2018; MSR ’18, p. 10–13. doi:10.1145/3196398.3196473.
64. Alsawalqah, H.; Faris, H.; Aljarah, I.; Alnemer, L.; Alhindawi, N. Hybrid SMOTE-Ensemble Approach for Software Defect Prediction. Software Engineering Trends and Techniques in Intelligent Systems; Silhavy, R.; Silhavy, P.; Prokopova, Z.; Senkerik, R.; Kominkova Oplatkova, Z., Eds.; Springer International Publishing: Cham, 2017; pp. 355–366.
65. Agrawal, A.; Menzies, T. Is "Better Data" Better than "Better Data Miners"? On the Benefits of Tuning SMOTE for Defect Prediction. Proceedings of the 40th International Conference on Software Engineering; Association for Computing Machinery: New York, NY, USA, 2018; ICSE ’18, p. 1050–1061. doi:10.1145/3180155.3180197.
66. Tay, Y.; Dehghani, M.; Bahri, D.; Metzler, D. Efficient Transformers: A Survey, 2020, [arXiv:cs.LG/2009.06732].
67. Tay, Y.; Dehghani, M.; Abnar, S.; Shen, Y.; Bahri, D.; Pham, P.; Rao, J.; Yang, L.; Ruder, S.; Metzler, D. Long Range Arena: A Benchmark for Efficient Transformers, 2020, [arXiv:cs.LG/2011.04006].
68. Zaheer, M.; Guruganesh, G.; Dubey, A.; Ainslie, J.; Alberti, C.; Ontanon, S.; Pham, P.; Ravula, A.; Wang, Q.; Yang, L.; Ahmed, A. Big Bird: Transformers for Longer Sequences, 2021, [arXiv:cs.LG/2007.14062].
69. Fiok, K.; Karwowski, W.; Gutierrez, E.; Davahli, M.R.; Wilamowski, M.; Ahram, T.; Al-Juaid, A.; Zurada, J. Text Guide: Improving the quality of long text classification by a text selection method based on feature importance, 2021, [arXiv:cs.AI/2104.07225].
70. Beltagy, I.; Peters, M.E.; Cohan, A. Longformer: The Long-Document Transformer, 2020, [arXiv:cs.CL/2004.05150].
71. Hellendoorn, V.J.; Sutton, C.; Singh, R.; Maniatis, P.; Bieber, D. Global Relational Models of Source Code. International Conference on Learning Representations (ICLR), 2020.
72. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 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); Association for Computational Linguistics: Minneapolis, Minnesota, 2019; pp. 4171–4186. doi:10.18653/v1/N19-1423.
73. Karampatsis, R.M.; Sutton, C. SCELMo: Source Code Embeddings from Language Models, 2020, [arXiv:cs.SE/2004.13214].
74. Herbold, S.; Trautsch, A.; Grabowski, J. A Comparative Study to Benchmark Cross-Project Defect Prediction Approaches. *IEEE Transactions on Software Engineering* **2018**, *44*, 811–833. doi:10.1109/TSE.2017.2724538.