10-2021

Assessing generalizability of CodeBERT

Xin ZHOU
Singapore Management University, xinzhou.2020@phdcs.smu.edu.sg

DongGyun HAN
Singapore Management University, dhan@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Assessing Generalizability of CodeBERT

Xin Zhou, DongGyun Han, and David Lo
School of Computing and Information Systems, Singapore Management University
xinzhou.2020@phdcs.smu.edu.sg, {dhan, davidlo}@smu.edu.sg

2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 978-1-6654-2882-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICSME52107.2021.00044

***Abstract*—Pre-trained models like BERT have achieved strong improvements on many natural language processing (NLP) tasks, showing their great generalizability. The success of pre-trained models in NLP inspires pre-trained models for programming language. Recently, CodeBERT, a model for both natural language (NL) and programming language (PL), pre-trained on code search dataset, is proposed. Although promising, CodeBERT has not been evaluated beyond its pre-trained dataset for NL-PL tasks. Also, it has only been shown effective on two tasks that are close in nature to its pre-trained data. This raises two questions: Can CodeBERT generalize beyond its pre-trained data? Can it generalize to various software engineering tasks involving NL and PL?**

Our work answers these questions by performing an empirical investigation into the generalizability of CodeBERT. First, we assess the generalizability of CodeBERT to datasets other than its pre-training data. Specifically, considering the code search task, we conduct experiments on another dataset containing Python code snippets and their corresponding documentation. We also consider yet another dataset of questions and answers collected from Stack Overflow about Python programming. Second, to assess the generalizability of CodeBERT to various software engineering tasks, we apply CodeBERT to the just-in-time defect prediction task. Our empirical results support the generalizability of CodeBERT on the additional data and task. CodeBERT-based solutions can achieve higher or comparable performance than specialized solutions designed for the code search and just-in-time defect prediction tasks. However, the superior performance of the CodeBERT requires a tradeoff; for example, it requires much more computation resources as compared to specialized code search approaches.

***Index Terms*—pre-trained model, generalizability, CodeBERT**

I. Introduction

With recent success of deep learning, applying deep learning became popular to automate Software Engineering (SE) tasks [1]–[4]. However, deep learning solutions usually require numerical vector representation of the object of interest as input [3]–[11]. This numerical vector representation is also called embedding. Recently, there are many works on source code embedding; they learn an embedding for a piece of code [3], [4], [9]–[11]. Though these source code embedding models work well in their dedicated tasks, the generalizability of their embeddings to other downstream tasks requires further investigation. For example, Kang et al. found that source code token embeddings extracted by code2vec [9] cannot be readily utilized for other downstream tasks, i.e. code comments generation, code authorship identification, and code clone detection [12].

Recently, in the natural language processing (NLP) area, BERT [13] has been a breakthrough in learning embedding of a natural language word. BERT outperformed state-of-the-art techniques by a large margin on *many* NLP tasks, such as question answering, natural language inference, named entity recognition, sentiment binary classification, text classification, etc. [14]–[19]. BERT is a typical pre-trained model, which has generalizability across *many datasets and downstream NLP tasks*. Inspired by a variant of BERT, i.e., RoBERTa [20], Feng et al. [21] proposed CodeBERT, a pre-trained model for source code and natural language texts. The released CodeBERT model is pre-trained on a code search dataset (CodeSearchNet) provided by Husain et al. [22], containing 2.1 million bimodal code-documentation pairs and 6.4 million unimodal code snippets.

CodeBERT has been evaluated on two natural language and programming language (NL-PL) based tasks: code search and code documentation generation [21]. For its evaluation, however, they used data from CodeSearchNet. Generalizability of BERT has been demonstrated on NLP data outside its pre-trained dataset [23]–[25]; however, this has not been demonstrated for CodeBERT. This leads us to investigate the following research question:

RQ1 Can CodeBERT generalize beyond its pre-trained data?

To answer RQ1, we evaluated the CodeBERT pre-trained model for the code search task on test datasets *other than* its pre-training corpus. First, we conducted experiments on a public dataset containing Python code snippets and their corresponding documentations (i.e., code comments) collected by Barone and Sennrich [26]. The nature of this dataset (code-documentation pairs) is the same as the dataset used in the original CodeBERT evaluation [21]. Second, we used another dataset of questions and answers about Python programming collected from Stack Overflow [27]. Using this dataset, we can further test the generalizability of CodeBERT on NL-PL paired data of a slightly different nature (than the one considered in the original evaluation). Moreover, while the original CodeBERT evaluation compares CodeBERT with generic models (e.g., CNN, RNN) as baselines, in this study, we use specialized deep learning approaches designed for code search, i.e., NCS [28] and UNIF [1] as baselines.

The results of our experiments show that CodeBERT can outperform all specialized approaches by about 31–38% in terms of the average mean reciprocal rank (MRR) scores, showing its generalizability across different datasets on the code search task. However, we also find that the superior

2576-3148/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSME52107.2021.00044

performance of CodeBERT comes at a cost; CodeBERT is 8–23 times slower than the baselines in producing a ranked list of code snippets given a query.

Considering another aspect of generalizability, CodeBERT was only evaluated on two tasks involving natural language and programming language (NL-PL tasks): code search and code documentation generation. Generalizability of BERT has been demonstrated on many different tasks [13], [14], [14]–[19]; however, this has not been demonstrated for CodeBERT. This leads us to investigate the following research question:

RQ2 Can CodeBERT generalize to more NL-PL tasks?

To answer RQ2, we evaluated the CodeBERT pre-trained model on another NL-PL task: just-in-time (JIT) defect prediction task, which is to predict the defectiveness of commits considering their NL descriptions and PL code snippets. We choose the JIT defect prediction task because it is a popular task (investigated in many prior works [2], [29]–[34]), is of high relevance to software engineers, and involves both source code and natural language. It is also of different nature compared to the two NL-PL tasks considered in the original CodeBERT evaluation: in the code search task, it takes NL descriptions as input and produces code; in the documentation generation task, it takes code as input and produces NL descriptions; in this task, it takes NL descriptions plus code as input and produces defectiveness label (i.e., defective or clean). We choose two recent deep learning based approaches, i.e., DeepJIT [2] and CC2Vec [34] as baselines.

The results of our experiments show that a simple application of the CodeBERT model can achieve performance near the state-of-the-art approach and outperform one recently published approach by 3.7–4.2% in terms of AUC scores, which shows the generalizability of CodeBERT to this NL-PL downstream task.

The main contributions of our work are:

- We assess the generalizability of CodeBERT in terms of its ability to work beyond its pre-trained data, and provide empirical evidence that demonstrates its effectiveness.
- We assess the generalizability of CodeBERT on another popular automated software engineering task (just-in-time defect prediction) and provide evidence that demonstrates its effectiveness.
- In our assessments, we compare CodeBERT with recently proposed *specialized* approaches for code search and just-in-time defect prediction. This comparison was missing in the original paper (CodeBERT was only compared to generic solutions, e.g., CNN).

The paper is organized as follows. Section II presents some background information on CodeBERT and the downstream tasks. Section III presents our experiments to answer RQ1 and their findings. Section IV describes our experiments to answer RQ2 and their findings. Section V presents our discussion. Section VI analyzes the threats to validity. Section VII discusses the related work. Conclusion and future work are presented in Section VIII.

II. Background

A. Pre-training of CodeBERT

CodeBERT is a bimodal pre-trained model for both programming language (PL) and natural language (NL) [21]. CodeBERT is pre-trained on a large-scale code search dataset CodeSearchNet provided by Husain et al. [22]. Though CodeSearchNet is for the code search task, its NL queries for code snippets are documentation of the code snippets (i.e., code comments) rather than real questions asked by programmers. CodeBERT is pre-trained considering two objectives. The first objective is masked language modeling (MLM), which considers the generation of masked code and NL tokens [13]. MLM is proven effective to learn a generalizable model to downstream tasks [13], [20], [35]. The second objective is the replaced token detection (RTD) proposed by Clark et al. [36] to consider the identification of replaced code and NL tokens. CodeBERT uses bimodal data (NL-PL pairs) to train the model for the MLM target and uses unimodal data to continue to train the model for the RTD target.

B. Fine-tuning of CodeBERT

As the pre-trained model is trained on its pre-training datasets, the trained model may not fit a specific downstream task at the beginning [12]. To use pre-trained CodeBERT in a specific downstream task, we need to fine-tune CodeBERT on datasets of that specific task [13], [21].

In this study, we mainly utilize CodeBERT for classification, and simply leverage CodeBERT as a sentence encoder: embed a sequence of tokens (either NL tokens or PL tokens) to a fixed dimension vector (embedding) that contains the semantic meaning of the sequence at the input.

As CodeBERT requires a specific format of the input, we need to do pre-processing on the input (a sequence of NL or PL tokens): special tokens *[CLS]* and *[EOS]* are added at the start and the end of the whole sequence respectively. In

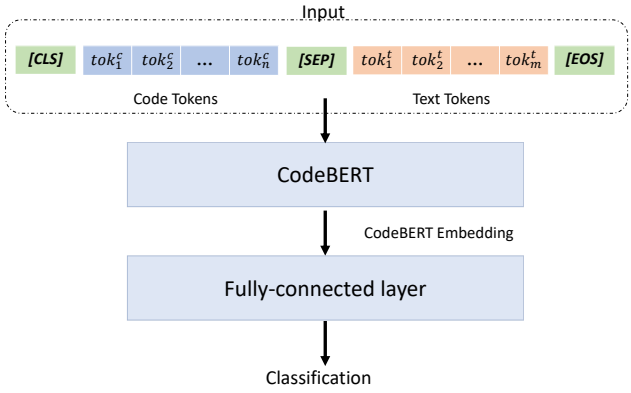

Fig. 1. Example of CodeBERT fine-tuning framework for classification. *[CLS]* and *[EOS]* are special tokens added at the start and the end of the whole sequence respectively and *[SEP]* is a separation token added between different segments. We use the token embedding of *[CLS]* at the output as the CodeBERT embedding for the whole input.

addition, if the given data is a pair of sequences, the special token *[SEP]* should be added in the middle of two sequences, as shown in Figure 1.

For classification, CodeBERT first encodes the input (a sequence of NL or PL tokens) into its CodeBERT embedding. As shown in Figure 1, for tasks of paired data (NL-PL pairs in code search), a pair of data (two sequences of NL or PL tokens) will be concatenated into one single sequence. This concatenated sequence is then fed into CodeBERT to get an embedding representing this pair. Following the previous work [21], we use the token embedding of *[CLS]* at the output as the embedding for the whole input in this study. Then, it feeds the embedding to a fully connected layer and either a softmax layer (for multi-class classification) or sigmoid function (for binary classification) to compute the classification. Then the loss is computed based on the gap between the classification and the ground truth, e.g., the defectiveness label (defective or clean) in the JIT defect prediction, and the loss is back-propagated to adjust the parameters of CodeBERT and the fully-connected layer. The tuned CodeBERT will fit these specific datasets better and is applied on a test data.

C. Downstream tasks

We choose two NL-PL downstream tasks to assess the generalizability of CodeBERT.

a) Code search: The task of code search is given a natural language query as input, find the most semantically similar code snippet to this query in a collection of code snippet candidates. Code search task can assess models' ability to capture the semantic similarity between code snippets and NL texts. In other words, the ability to distinguish semantically related NL-PL pairs from unrelated pairs. For this task, we select RoBERTa [20] and two recent specialized deep learning based code search tools: Neural Code Search (NCS) [28] and UNIF [1] as baselines.

b) Just-in-time defect prediction: Just-in-time defect prediction is a task of predicting whether a given commit is buggy or clean according to the extracted features from NL commit messages and PL code changes [37], [38]. A commit consists of code changes and a commit message summarizing the commit. A commit may modify several different files in a project at the same time, which means code changes spread over the files. It is necessary to consider features extracted from different files to make a prediction, because code changes in any one of the files may bring in a bug to a project. We select two recent specialized deep learning approaches, namely DeepJIT [2] and CC2Vec [34] as baselines.

III. RQ1: Can CodeBERT generalize beyond its pre-trained data?

In this part, we evaluated the CodeBERT pre-trained model for the code search task to test its generalizability to datasets *other than* its pre-training data.

We first used a public dataset of Python code snippets and their corresponding documentations (code comments) that was released by Barone and Sennrich [26], denoted as the Doc-Code dataset; this dataset has the same nature (code-documentation pairs) as the CodeSearchNet dataset. In a typical scenario of code search, users typically use questions about programming instead of entering code comments as queries. Thus, we also conducted experiments on another dataset containing questions and answers about Python programming collected from Stack Overflow [27], denoted as the Question-Code dataset. Using this dataset, we can further assess whether CodeBERT can be generalized to a dataset of question-answers pairs. For the Question-Code dataset, specifically, we used the *Single-Code Answer Posts* data collected by Yao et al. [27], which paired a complete code snippet in an (accepted) answer with the question title as a question-code pair.

TABLE I
Statistics of the code search datasets

Datasets	Question-Code	Doc-Code
Train	51,176	67,864
Validation	17,058	22,621
Test	17,060	22,623
Unique tokens in PL code	136,552	127,398
Unique tokens in NL text	25,117	67,895
Common tokens percentage	7.86%	22.91%

A. Datasets

We conducted our experiments on two publicly available datasets: the Doc-Code dataset and the Question-Code dataset. Table I shows the detailed statistics of the two datasets, including the number of code-text pairs in the training, validation, and test set respectively, the number of unique tokens in PL code and NL text, and the average percentage of common unique tokens shared between code snippets and their corresponding NL texts (documentations or questions). In the Doc-Code dataset, a code snippet shares 22.91% of unique tokens with its corresponding documentation on average. However, for the Question-Code dataset, code snippets and questions have only 7.86% of unique tokens in common on average, which indicates questions and theirs corresponding code snippet answers are less similar than the documentations and code snippets in the Doc-Code dataset in terms of the co-occurrence of tokens.

B. Baselines and CodeBERT for Code Search

Many deep learning based specialized code search approaches [1], [22], [28], [39], [40] have two separate embedding modules: the *Code Embedding Network* to embed a code snippet into a vector, and the *Text Embedding Network* to embed a NL query into a vector. These two modules are shown in Figure 2. If a code snippet and a description have similar semantic meaning, their embedded vectors should be close to each other in the joint embedding space; and for an unrelated code and description pair, their embeddings should be far away from each other [39]. This goal is represented by the ranking loss function [41], [42]:

$$L(\theta) = \sum_{\langle c, q^+, q^- \rangle \in P} \max(0, \eta - \cos(c, q^+) + \cos(c, q^-))$$

where c is a code embedding of a code snippet and q^+ is the embedding of the corresponding description of the code snippet c, called a positive sample; q^- is the embedding of a negative sample of code snippet c, which is randomly chosen from the pool of descriptions and is not likely to have a similar semantic meaning to code snippet c; P denotes the training data set. Intuitively, the ranking loss encourages the cosine similarity score between a code snippet and its corresponding NL text to be large, and the cosine similarity scores between a code snippet and irrelevant NL descriptions to be small.

In this study, we select two recently published specialized models for code search: namely NCS [28], UNIF [1], and an NLP model, RoBERTa [20], as baselines. We will briefly introduce the selected baselines and the usage of CodeBERT.

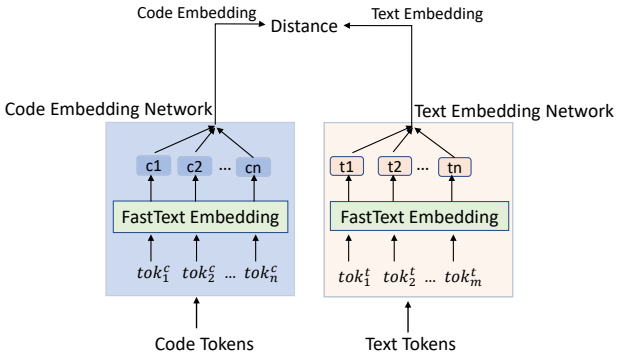

Fig. 2. The framework of Neural Code Search (NCS) and UNIF

*a) **NCS**:* The neural code search (NCS) [28] is an unsupervised learning model for the code search task. For the input layer, NCS directly uses FastText [43] token embeddings. FastText is similar to the Word2Vec model, which can learn a token-level representation according to a given corpus. For the code embedding module, NCS simply sums the token embeddings of a code snippet by their corresponding TF–IDF weights:

$$c = \sum_{i=1}^{n} \text{tfidf}(e_i) \cdot e_i$$

where e_i is the embedding for the i-th token in this code snippet and $\text{tfidf}(e_i)$ is the TF-IDF value of the token e_i. For query embedding module, NCS simply calculates the average of embedding of all tokens in a query.

*b) **UNIF**:* UNIF [1] is a supervised version of NCS. For code embedding, UNIF also directly utilizes the FastText model for initialization of token embeddings. But UNIF tunes the token embeddings during the training process. For the code embedding module, instead of using fixed weights like TF-IDF values, they used the attention mechanism [9] to compute attention weights. The attention mechanism can highlight the different importance of tokens in a code snippet. Given the embedding of all tokens in a code snippet, i.e. $e_1, e_2, ..., e_n$, the attention weight α_i for e_i is computed as follows:

$$\alpha_i = \frac{\exp(a_c \cdot e_i^T)}{\sum_{j=1}^{n} \exp(a_c \cdot e_j^T)}$$

where a_c is a code context vector trained according to the loss function. After getting the weights α_i, the single vector for this code snippet is computed as a weighted average across all tokens in this code snippet. For the query embedding module, UNIF also calculates the average of embeddings of all tokens in a query as query embedding.

*c) **CodeBERT**:* CodeBERT modeled the code search task as a binary classification problem: given any code-documentation pair, the model identifies whether this pair is semantically related or not. They first concatenated the code snippet and the query into one single sequence: *[CLS], w1, w2, ..wn, [SEP], c1, c2, ..., cm, [EOS]*, where *[CLS]* and *[EOS]* are special tokens added at the start and the end of the concatenated sequence respectively and *[SEP]* is a separation token added between the natural language query $(w_1, ..., w_n)$ and the code snippet $(c_1, ..., c_m)$. Second, they fed the concatenated sequence into the CodeBERT and considered the *[CLS]* representation returned by the CodeBERT as the CodeBERT embedding. Then the CodeBERT embedding was fed into a fully connected layer and a sigmoid function to give prediction. As the framework of CodeBERT and RoBERTa is the same, for brevity sake, we omit the explanation for RoBERTa.

C. Experimental Details

Train-Test Pipeline: The data split of each code search dataset is shown in Table I. Datasets are divided into the training, validation and test sets. We use the same data split for all methods, including the three baseline methods. Before training, NCS and UNIF require FastText token embeddings for initialization. We use the FastText library[1] to train a FastText model on each dataset, which can turn a code or NL token into a corresponding FastText token embedding.

NCS, which is an unsupervised code search method, does not learn the embeddings of code or NL texts but rather directly uses the weighted average of FastText token embeddings as the embeddings of code or NL texts. For training UNIF, RoBERTa and CodeBERT, which are supervised code search methods, we first build a positive-negative balanced dataset from the training set of each dataset by pairing the corresponding description and an irrelevant description sampled from the training set to each code snippet in the training set. UNIF then is trained on the built balanced dataset to minimize the ranking loss function. For RoBERTa and CodeBERT, the code search task is modeled as a binary classification problem; specifically, we assign a label 1 for the pair of code and its corresponding description and assign 0 for the pair of code and the sampled irrelevant description. Thus, these two methods are trained to predict the label of each pair of data in the balanced dataset.

For validation, similar to the training set, we build another positive-negative balanced dataset from the validation set of each dataset. We use the validation loss of the balanced dataset built upon the validation set to tune the models. We tune each model using a grid search procedure [44] considering the following set of hyper-parameters and their

[1]https://github.com/facebookresearch/fastText

TABLE II
MRR SCORES OF ALL CODE SEARCH MODELS ON THE DOC-CODE AND QUESTION-CODE DATASETS

Datasets	Models	MRR
Doc-Code	CodeBERT	**0.98**
	RoBERTa	0.97
	NCS	0.60
	UNIF	0.67
Question-Code	CodeBERT	**0.74**
	RoBERTa	0.50
	NCS	0.43
	UNIF	0.43

possible values: the batch size is in $\{8, 16, 32\}$ and the learning rate is in $\{5e^{-6}, 1e^{-5}, 5e^{-5}\}$. We select the combination of hyper-parameters that lead to the smallest validation loss. The resultant model trained using the selected hyper-parameters is used for evaluation on the test sets.

For testing, the evaluation is performed on a hold-out test set which is the test set of the each dataset. We use the same evaluation method for all approaches. When evaluating, for each NL query Q in the test set, we considered the corresponding code snippet C as the positive sample because C is semantically related to the query Q. To assess the ability of a model to distinguish the semantically related code snippet from irrelevant code snippets, we randomly sampled K irrelevant code snippets C^- from the test set as negative samples and mix the correct code snippet C with the irrelevant code snippets. Following previous work [27], [45], we set $K = 49$. In this case, for each query, there are 50 code snippet candidates and among them, only one code snippet is semantically related to that query. We then pair the query with all code candidates and input each pair into the trained model to get a prediction score. Then we rank all code snippets from high to low according to scores.

Evaluation Metric: Following the previous works [1], [27], [39], [45], the code search performance of a model is evaluated by the average mean reciprocal rank (MRR) metric [46] across the whole test set T:

$$MRR = \frac{1}{|T|}\sum_{i=1}^{|T|}\frac{1}{rank_i},$$

where $rank_i$ is the rank of the corresponding code snippet C for the i-th query in all code snippets candidates of the i-th query. A higher MRR score indicates a better code search performance.

D. Results and Discussions

a) Generalizability of CodeBERT: The results of the code search for two datasets are shown in Table II. For the Doc-Code dataset, CodeBERT outperformed all specialized code search approaches with a large margin: CodeBERT achieved a 0.98 MRR score, which means that it can rank the correct code snippet in the first place for most of the queries. However, two specialized approaches NCS and UNIF only achieved MRR scores of 0.60 and 0.67 respectively. For the Doc-Code dataset, RoBERTa also achieved a very high MRR score of 0.97, which is slightly worse than CodeBERT.

As RoBERTa can also achieve such a high MRR score, it indicates that the Doc-Code dataset seems to be simple for BERT's variants like RoBERTa and CodeBERT. As shown in Table I, the Doc-Code dataset contains many common tokens between code and documentation (22.91% on average), which makes it easier to predict the relationship between two sequences of tokens.

For the Question-Code dataset, CodeBERT outperformed all specialized code search approaches by 31% in terms of the MRR scores: CodeBERT achieved an MRR score of 0.74, while NCS and UNIF both achieved 0.43 MRR score. For the Question-Code dataset, RoBERTa got an MRR score of 0.50, which is slightly better than NCS and UNIF.

Comparing CodeBERT with RoBERTa in the Question-Code dataset, CodeBERT has a large improvement over RoBERTa (more than 24% MRR score), indicating the superiority of CodeBERT over the NLP pre-trained model RoBERTa on the NL-PL tasks. In addition, the significant drop of performance of RoBERTa on the Question-Code dataset indicates that this dataset requires a model to capture more semantic meanings of both code and NL texts to do prediction.

Generally, CodeBERT can perform better compared to other recent specialized code search approaches, showing its generalizability on different datasets other than its pre-trained dataset for the code search task.

b) Time efficiency of CodeBERT: Though CodeBERT showed superior performance compared to the specialized code search approaches, the time efficiency of the CodeBERT can be a concern in practice. We report the training (fine-tuning, for CodeBERT and RoBERTa) and evaluation time of all the approaches in Table III. We ran all the approaches on a desktop computer with Nvidia GeForce RTX 2080 Ti and Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz.

TABLE III
TRAINING (TRAIN.) AND EVALUATION (EVAL.) TIME IN MINUTES OF CODEBERT AND BASELINES

Phase	Datasets	NCS	UNIF	RoBERTa	CodeBERT
Train.	Doc-Code	-	16	253	231
	Question-Code	-	12	211	186
Eval.	Doc-Code	7	3	64	56
	Question-Code	5	2	53	46

For the training time, CodeBERT required 231 minutes and 186 minutes for the Doc-Code dataset and the Question-Code dataset respectively, which are 14–15 times longer than UNIF. The training time for NCS is zero because it is an unsupervised method that does not require training. For the evaluation time, CodeBERT required 8–9 times as much evaluation time as NCS's for the two datasets and required 18–23 times as much evaluation time as what UNIF needed. In addition, CodeBERT is slightly faster than RoBERTa for both training and evaluation.

RQ1 Main Findings: CodeBERT is generalizable to datasets other than its pre-trained dataset for the code search task. CodeBERT outperforms the recent specialized deep learning based code search approaches consistently across the two datasets. The improvements achieved by the CodeBERT model range from 31% to 38% in terms of the average mean reciprocal rank metric. However, the superior performance of the CodeBERT comes at a cost; CodeBERT requires 8–23 times more time than that of the specialized code search baselines in producing an output given a query.

IV. RQ2: Can CodeBERT generalize to more NL-PL tasks?

In this part, we evaluated the CodeBERT model on another NL-PL task, i.e., just-in-time defect prediction, and compare it with the recent specialized approaches to assess its generalizability to this NL-PL task. We aim to investigate whether CodeBERT can transfer the knowledge from the code search dataset (CodeSearchNet) to the just-in-time defect prediction task.

We choose the just-in-time defect prediction task for three reasons. First, just-in-time defect prediction is a popular task that has been studied in many previous works [2], [29]–[34]. This task can potentially identify defects in an early stage (even before new code changes are introduced into the code base), which is one of the best cost-saving practice [2]. Second, this task involves both source code, i.e., code changes, and natural language, i.e., commit messages, which is suitable to assess CodeBERT's generalizability to a specific NL-PL task. Third, this task is dissimilar to the evaluated two NL-PL tasks. The code search task retrieves code given NL descriptions as input considering the semantic similarity between code and NL descriptions. The documentation generation task produces NL descriptions given code as input. This task predicts the defectiveness of commits considering their NL descriptions and PL code snippets as input.

A. Just-in-time defect prediction

Just-in-time (JIT) defect prediction is the task of predicting whether a given commit is buggy or not according to the extracted features from commit messages and code changes [37], [38]. The core part of this task is to design a feature extraction model that can properly capture the *buggy probability* of a commit. Machine learning techniques with carefully hand-crafted features are common solutions to JIT defect prediction [29]–[32]. Recently, some works use deep learning techniques, like deep belief network [33] and convolutional neural network (CNN) [2], to solve JIT defect prediction and outperform the hand-crafted features with a large margin. We choose two deep learning based approaches as baselines: DeepJIT [2] and CC2Vec [34]. DeepJIT used a specific CNN [47] to extract features from code changes and commit messages [2]. CC2Vec is a deep learning based model that learns the distributed representation of code changes guided by their log messages [34].

B. Dataset

We use the same datasets used by Hoang et al. [34]: OpenStack dataset and Qt dataset, which are commits collected from the OpenStack and Qt software projects respectively by McIntosh and Kamei [48]. The Qt dataset contains 25,704 commits and among them 1,825 commits are defective. The OpenStack dataset contains 13,304 commits in total and 1,627 of them are defective. We use the data released in the replication package of CC2Vec[2] and use the original data split. For OpenStack dataset, the training set contains 11,973 commits and the test set has 1,331 commits. For Qt dataset, the training set contains 23,133 commits and the test set has 2,571 commits.

C. Baselines

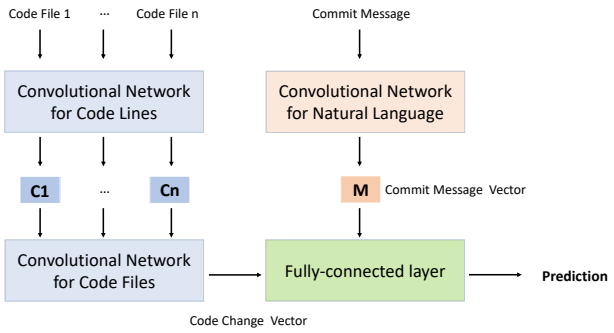

Fig. 3. The framework of DeepJIT

1) DeepJIT: DeepJIT is an end-to-end deep learning framework that first represents code tokens and commit message words in fixed dimension word embedding and then uses CNN to automatically extract features from code changes embedding and commit messages embedding [2]. As both code changes and commit messages are considered as input, the JIT defect prediction model is related to both natural language and programming language. In DeepJIT, the embeddings of commit message and code change tokens are randomly initialized and tuned during the training process. After the initialization, all tokens in both the commit message and code changes are turned into a fixed dimension vector. As shown in Figure 3, for a commit message (a sequence of tokens), DeepJIT uses a specific CNN model for text classification task [47] to fuse the embeddings of all tokens in this commit message into a single vector, called the commit message vector.

As code changes are a list of code lines, DeepJIT uses CNN, in a hierarchical way, to fuse the code changes into a single vector. First, for all code lines, DeepJIT uses a CNN module to embed a code line (a sequence of code tokens) into a vector representing that line. After the first CNN module, all code lines in the code changes are embedded into vectors. Then, DeepJIT uses a second CNN module to fuse all embeddings of all code lines into a single vector, which is referred to as code change vector; this vector represents the whole code changes for this commit. Finally, the model concatenates

[2]https://github.com/CC2Vec/CC2Vec

vectors for code changes and commit messages and feeds the concatenated vector into a fully connected layer for prediction.

2) CC2Vec: CC2Vec is an end-to-end deep learning model that learns distributed representations of code changes supervised by the words in log messages. The training goal of CC2Vec is to learn an embedding of a code change that can predict whether a word from the word vocabulary exists in its log message or not. As log messages are the descriptions of the semantic meaning of corresponding code changes, this training goal can force the learned embeddings of code changes to capture the semantic meaning of code changes [34]. CC2Vec uses a hierarchical attention network (HAN) [49] as the feature extraction layer to aggregate the embeddings of all tokens in a code change into a single vector that captures the semantic meaning of the whole code change. CC2Vec is not a model designed for just-in-time defect prediction, but a model to learn the representation of code changes. However, Hoang et al. [34] concatenated the feature extracted by CC2Vec with the original DeepJIT to boost the performance of DeepJIT.

D. JIT defect prediction using CodeBERT

To utilize CodeBERT in the just-in-time defect prediction task, we need to design a model for JIT defect prediction. Hereafter, we call it as CodeBERT4JIT for short. As we mentioned before, CodeBERT essentially is a sentence encoder, which encodes a sequence of tokens (sentence) into a single vector and this vector contains the semantic meaning of the sentence. In addition, CodeBERT is a bimodal sentence encoder that can encode both code lines and NL descriptions.

Inspired by the framework of DeepJIT, as shown in Figure 3, we find that the first layer of CNNs for both code lines and commit messages (the convolutional network for code lines and the convolutional network for the commit message) are encoders that turn a sentence (i.e., a sequence of NL or PL tokens) into a single vector. The function of these CNNs is the same as the CodeBERT, i.e., encoding a sequence into a vector that contains semantic meanings of the sequence. Thus, we simply replace the CNN sentence encoders in DeepJIT with the CodeBERT model to build a simple application of the CodeBERT model for the just-in-time defect prediction task. We do not design a complex network because we only want to see the ability and generalizability of CodeBERT on this task. A complicated design makes it harder to investigate the effectiveness of CodeBERT in the just-in-time defect prediction task.

Given a commit message m which is a sequence of NL tokens $[w_1, w_2, ..., w_{|m|}]$, we feed it into the CodeBERT and get its sentence embedding:

$$Z_m = \text{CodeBERT}(m)$$

where $Z_m \in R^d$ is the CodeBERT embedding of this commit message and d is the dimension of the embedding output by the CodeBERT. A code change C contains changed code lines in (potentially different) source code files $[C_1, C_2, ..., C_{|C|}]$ and C_i stands all changed code lines in i-th source code file. We aim to embed these code lines into a single vector Z_C that contains the semantic meaning of this code change C.

We first concatenate all code lines in one source code file (C_i) into one single sentence ($\hat{C}_i$) and then feed this concatenated sentence into the CodeBERT to get its sentence embedding:

$$Z_{C_i} = \text{CodeBERT}(\hat{C}_i)$$

where $Z_{C_i} \in R^d$ is the CodeBERT embedding of the code file C_i. For code files in each code change, i.e. C_i, we compute its CodeBERT embedding vector Z_{C_i}. These embedding vectors are then fed into a CNN module and a max-pooling layer to form an embedding vector (Z_C) for the code change C.

To fuse the embeddings of different files, a filter $f \in R^{k \times d}$ is utilized to a window of k files to compute a new feature:

$$x_i = \alpha(f * C_{i:i+k-1} + b_i)$$

where $*$ is a sum of element-wise product, $\alpha(.)$ is a non-linear activation function, and $b_i \in R$ is the bias value. The filter f is applied to every k-files of the code change. The generated features are then concatenated to form a vector X such that:

$$X = [x_1, x_2, ..., x_{|C|-k+1}]$$

Following prior work [2], for each filter, we then use a max-pooling layer [50] to process the feature vector X to obtain the highest value:

$$Z_C = \max_{1 \le i \le |C|-k+1} x_i$$

After getting the embedding for the commit message and the code change, we concatenate these two embeddings to generate a feature representation, i.e., Z representing this commit:

$$Z = Z_m \oplus Z_C$$

where $\oplus$ is the concatenation operator. Finally, the vector Z for a commit is then fed into a fully-connected (FC) layer with a sigmoid function to map the high dimensional vector to a 1-dimensional vector to compute a prediction on this binary classification task:

$$y = \text{sigmoid}(\alpha(w_h \cdot Z + b_h))$$

where $\cdot$ is a dot product, w_h is a weight matrix of the FC layer, b_h is the bias term, $\alpha(.)$ is a non-linear activation function, and sigmoid$(.)$ is a function to do normalization on the prediction.

E. Experimental Details

Train-Test Pipeline: We use the data released in the replication package of CC2Vec and the dataset is divided into the training set and test set. We use 10% data of the training set for validation. The data split is the same for all approaches in this study.

For training of two baselines, we use scripts in the replication packages and use the default hyper-parameters in the packages. For training CodeBERT, we set the learning rate as $1e^{-5}$, the batch size as 8, the max sequence length (the maximum number of tokens considered in a sequence) as

120. We use the Adam optimizer to update model parameters. As JIT defect prediction datasets often have an imbalance problem: only a few commits are buggy while a large number of commits are clean, we use the same loss function as the previous work [2] to address this issue.

For validation, we tune the hyper-parameters of each model by observing the performance of the model trained using a set of hyper-parameter values on the validation set. Specifically, we tune the hyper-parameters using a grid search procedure with the following set of parameters and their possible values: the learning rate is in $\{5e^{-6}, 1e^{-5}, 5e^{-5}\}$ and the max sequence length is in $\{100, 120, 150\}$. We choose the combination of hyper-parameters that lead to the best performance in the validation process. The model that performs best in the validation process is used for evaluation on the test sets.

For testing, following the previous work [2], the evaluation is performed on the hold-out test sets which are the test sets of the released datasets. For each approach (CodeBERT4JIT, DeepJIT, and CC2Vec), we feed all commits in the test sets into the chosen model (the best-performing model identified in the validation process) to get the predictions (defective or clean) and calculate the AUC scores based on the predictions and the ground truths.

Evaluation Metric: Similar to the previous studies [2], [34], [48], to evaluate the effectiveness of the JIT approaches, we use the area under the receiver operator characteristics curve (AUC), a threshold-independent metric to measure their ability to differentiate between defective or benign commits. The values of AUC range between 0 (the worst prediction) and 1 (the best prediction).

F. Results and Findings

a) Generalizability of CodeBERT: Table IV presents that the proposed approach using CodeBERT outperforms the specialized approach DeepJIT by 0.037 and 0.042 improvement in terms of AUC scores on the OpenStack dataset and Qt dataset respectively. Though this simple application of CodeBERT on the JIT defect prediction cannot outperform the state-of-the-art approach CC2Vec on all datasets, CodeBERT achieved comparable results with CC2Vec on the OpenStack dataset. Considering that CC2Vec is dedicatedly designed for code changes while CodeBERT is not designed for this task, the great performance of CodeBERT indicates its generalizability to this specific NL-PL tasks.

We also employed an ablation test [51], [52] to analyze the contribution of each component of commit data, by only feeding code changes or a commit message in a commit and evaluate their AUC scores separately. As Table V shows, for predicting a defective commit, the proposed model benefits more from code changes than the commit message. For code changes, CodeBERT4JIT achieved AUC scores of 0.775 and 0.771 for the OpenStack dataset and the Qt dataset respectively, while it can only got AUC scores of 0.731 and 0.726 by only using commit messages.

TABLE IV
AUC RESULTS OF JUST-IN-TIME DEFECT PREDICTION

Datasets	DeepJIT	DeepJIT+CC2Vec	CodeBERT
OpenStack	0.771	0.797	**0.808**
Qt	0.765	**0.822**	0.807

TABLE V
AUC SCORES OF VARIANT MODELS USING DIFFERENT PARTS OF DATA

Model	OpenStack	Qt
Commit Message	0.731	0.726
Code Changes	0.775	0.771
All	0.808	0.807

RQ2 Main Findings: A simple adaptation of CodeBERT (CodeBERT4JIT) outperformed the specialized approach DeepJIT by 3.7–4.2% in terms of AUC and achieved comparable performance with the state-of-the-art approach on the OpenStack dataset, indicating its generalizability to this specific NL-PL task.

V. DISCUSSION

A. Lessons Learned

From the 2 tasks above, we see that the pre-trained CodeBERT model is promising for NL-PL tasks. In the code search task, the fine-tunned CodeBERT model outperformed the specialized approaches, i.e., NCS and UNIF, by 31% to 38% in terms of the average mean reciprocal rank metric. In the case of the JIT defect prediction task, fine-tuned CodeBERT achieved comparable performance with the state-of-the-art approach (CC2Vec) on the OpenStack dataset. Though the performance of CodeBERT was worse than that of the state-of-the-art approach on the Qt dataset, it still outperformed a recent specialized approach DeepJIT consistently on two datasets by 3.7–4.2% in terms of the receiver operator characteristics curve metric. **Based on these promising results, we encourage researchers to consider the fine-tuning of pre-trained CodeBERT approaches as the baseline in future work involving NL-PL tasks.**

In the JIT defect prediction task, though CodeBERT achieved improvements compared to DeepJIT, CC2Vec brought larger improvements to DeepJIT than the CodeBERT. Compared with the superior performance of the CodeBERT on the code search task, the relatively worse performance in the JIT defect prediction task indicates that CodeBERT may not be equipped to perfectly adapt to the dataset that is different from the one it has been pre-trained on. Howard and Ruder had raised this issue before in the NLP domain [53]. They suggested that even using a very diverse corpus for pre-training, the data of the target task will likely come from a different distribution. To mitigate this issue, they proposed task-adaptive fine-tuning, which fine-tuned the pre-trained model with the pre-training objective on the task training data. This approach brought significant improvements for the pre-trained model in their experiments. Thus, researchers can consider the task-adaptive fine-tuning approach, which further fine-tunes the pre-trained CodeBERT model on the task-specific datasets with the pre-training objectives: MLM and RTD.

It is recommended to fine-tune the CodeBERT on the downstream task datasets with the pre-training objectives first.

B. Challenges in Computation Time

Though CodeBERT showed great performance on the 2 tasks above, the time efficiency of the CodeBERT can be a concern in practice, especially for retrieval-related tasks like code search.

The solution of CodeBERT for code search is expensive in terms of computation time when doing realistic code search tasks with over millions of code snippet candidates for each query in the search corpus. To illustrate this issue, we report the evaluation time for one query when the search space is 10,000 (i.e., one correct code snippets and 9,999 unrelated code snippets) in Table VI. In this setting, CodeBERT took 34.2 seconds to provide recommendations for one query, which is 9–24 times slower than NCS and UNIF. Teevan et al. [54] found that slightly slower retrieval can lead to a dramatic drop in the perceived quality of results. Eric and Jake [55] also found that increasing the load time of the result page in Google by 100 milliseconds can lead to a decrease in the number of searches per person. For CodeBERT, 34 seconds of waiting time for one query may not be acceptable for users and this waiting time will go up when the number of code snippets in the search corpus increases.

Another issue about the computation time of CodeBERT is related to the framework. Suppose we have A queries waiting to be answered by a code search system that has B code snippets in the search corpus. Note that both A and B can be a large number. For specialized code search approaches like NCS and UNIF, they have separate Code Embedding Network and Text Embedding Network as shown in Figure 2. In this case, they only need to encode the B code snippets once by the trained model and store all code snippet embeddings. When a new query comes, they can encode the new query once by the trained model and compute the cosine similarity scores between this query's embedding and all stored the code snippet embeddings. So for specialized code search approaches such as NCS and UNIF, they only need to encode all queries and code snippets into corresponding embeddings separately for A+B times in total.

However, when it comes to the framework of CodeBERT, the code snippet and the corresponding query are concatenated into one single sequence that is fed to the CodeBERT model as a whole. In this case, we cannot split the embedding process of code and queries separately. To build a realistic code search system by this implementation, it needs to encode for $A \times B$ times to get embeddings for all possible code-query pairs.

TABLE VI
EVALUATION TIME IN SECONDS FOR ONE QUERY WHEN THE SEARCH SPACE IS 10,000

Dataset	NCS	UNIF	CodeBERT
Question-Code	3.8	1.4	34.2

There are some potential solutions to mitigate this issue. For instance, to reduce the time complexity of the CodeBERT in the code search task, we can separately encode the PL code snippets and the NL descriptions by the CodeBERT model, instead of encoding a sentence concatenated by an NL-PL pair. By doing so, the number of times the CodeBERT needs to encode for A queries and B codes snippets drops from $A \times B$, for all possible pairs between code and queries, to A+B, for code and queries separately. However, the effectiveness of this idea requires further investigation in future work.

Nogueira and Cho also addressed this issue in the document retrieval task before [15]. They proposed a two-step approach to mitigate this computation time issue: first, a large number of possibly relevant documents to a given question were ranked by a standard and fast mechanism, such as BM25, to narrow down the search space [15]; second, they used a more powerful but more computationally-intensive method, e.g., BERT, to re-rank the top K documents returned by the first step. Their approach outperformed the previous state of the art by 27% (relative) in terms of MRR@10 [15]. We will leave the exploration of a feasible code search approach that combines both powerful models like CodeBERT and efficient models like the NCS in future work.

VI. THREATS TO VALIDITY

Threats to Internal Validity. Threats to internal validity are concerned with the factors that may affect our results. In the code search task, we have compared CodeBERT against two recently published approaches for the code search task [1], [28]. Since the source code of these two approaches was not made publicly available, we needed to re-implement our version of those techniques. We tried our best to closely follow the description in their original work. We released a replication package for others to check. For the JIT defect prediction task, we use scripts in the replication packages of baselines and retain the default hyperparameters in the replication packages unchanged.

Threats to External Validity. Threats to external validity are concerned with the generalizability of our findings. We have conducted experiments on two different NL-PL tasks with four publicly available datasets and these datasets are diverse from several aspects, e.g., scale, type of query, number of PL code per data item, etc. The two NL-PL tasks are also of different nature: the code search takes NL description as input and produces the code snippets; the JIT defect prediction takes both NL descriptions and PL code snippets as input and predicts a defectiveness label. While we try to ensure the diversity of datasets and tasks, our findings may not generalize to all NL-PL tasks and datasets. Further studies are needed to confirm our results for other NL-PL tasks. Besides, in this work, we focus only on the embedding of a whole code snippet or a code-text pair. In other words, we use CodeBERT to embed a sequence of NL/PL tokens into a single vector to represent this sequence. We haven't investigated other granularity of CodeBERT embeddings, e.g., embeddings of source code tokens.

Threats to Construct Validity. Threats to construct validity are concerned with the evaluation metrics we choose. For the code search task, we followed the previous works to use the average mean reciprocal rank (MRR) metric [1], [27], [39], [45]. For the JIT defect prediction, we followed Hoang et al. [2], [34] to use the area under the receiver operator characteristics curve (AUC), which is a performance measure recommended for assessing the discriminatory power of defect prediction models [56].

VII. Related Work

Recently, in the natural language processing (NLP) area, BERT outperformed the word2vec techniques by a large margin on *many* NLP tasks [14]–[19], showing its superiority over word2vec. Inspired by the success of pre-trained embeddings for natural languages [13], Feng et al. proposed CodeBERT [21], a pre-trained model for both NL and PL. In this paper, we investigate the generalizability of CodeBERT in terms of additional datasets and an additional downstream task, as compared to the state-of-the-art approaches that are specifically designed for downstream tasks.

There are some pre-trained models for *PL or NL-PL* data. Kanade et al. used masked language modeling and next sentence prediction as the pre-training objectives to train a BERT model on Python source code, namely CuBERT [57]. The authors evaluated the CuBERT on several downstream tasks and outperformed generic baselines like LSTM and Transformers. Svyatkovskiy et al. proposed a multi-layer generative transformer model for code (GPT-C) [58], which is a variant of the GPT-2 [59]. GPT-C is trained from scratch on a large unsupervised source code dataset. Buratti et al. proposed a transformer-based language model called C-BERT [60], which is pre-trained on top-100 starred GitHub C language repositories. C-BERT achieved good performance in the abstract syntax tree (AST) tagging task and the vulnerability identification task. Guo et al. proposed a pre-trained model for programming language considering the inherent structure of code [61]. They used masked language modeling, edge prediction, and node alignment as pre-training targets and outperformed the baselines on several downstream tasks such as code refinement and code translation.

Lachaux et al. proposed neural source-to-source translator by leveraging the cross-lingual language model (XLM) [62]. They aimed to design a better approach based on unsupervised principles like XLM, while we aimed to evaluate the generalizability of CodeBERT. Mastropaolo et al. examined the ability of a pre-trained Text-To-Text Transfer Transformer (T5) model to generalize to several downstream tasks such as automatic bug-fixing and code summarization [63]. In our work, we evaluated the generalizability of CodeBERT on different downstream tasks which are not covered by their work. Whether T5 or CodeBERT is better is still unknown, which may need further investigation in the future, and outside the scope of this work. Recently, Tian et al. evaluated the benefit of pretrained BERT model on predicting the correctness of patches in the program repair task [64]. They evaluated a BERT model trained on NL texts, while we evaluated a CodeBERT model trained on both NL and PL data. Besides, they focused on the program repair task only, while our work focused on two other NL-PL tasks. Another closely related work, that is developed in parallel with our work, is the work by Lu et al. named CodeXGLUE that is described in their arXiv manuscript [65]. The main contribution of their work is different from ours; they create a benchmark dataset for ten programming language tasks while we assess the generalizability of the CodeBERT model. Still, as part of their study, they evaluate CodeBERT on several tasks. Our work is different from theirs in several aspects: (1) CodeXGLUE used generic deep learning models (e.g. BiLSTM) as baselines, while we chose the state-of-the-art approaches of each task as baselines. Thus, our study can shed light on whether CodeBERT is competitive against the state-of-the-art. (2) They did not consider the just-in-time defect prediction task that is considered in this work. Their findings and our findings complement each other to provide empirical evidence for the effectiveness of pretrained models like CodeBERT.

VIII. Conclusion and future work

In this work, we have conducted experiments to study on the generalizability of CodeBERT, the first pre-trained model for both natural language (NL) and programming language (PL). Our experiments include the code search task and the just-in-time defect prediction task. For the code search task, our experimental results reveal that the fine-tuned CodeBERT model outperforms the specialized code search approach by 31% to 38% in terms of MRR metric on datasets that are not included in its training dataset and have different query types. The experiment results indicate CodeBERT is generalizable to more data that is not included in its training dataset.

For the just-in-time defect prediction task, a simple application of CodeBERT achieves comparable performance the state-of-the-art on one dataset and outperforms a recent specialized approach by 3.7–4.2% in terms of the AUC scores. It indicates that CodeBERT is generalizable to datasets of a specific NL-PL task, though the tasks and datasets are of different nature of evaluated tasks: the code search and the code documentation generation. In the future, we are interested in a few directions: (1) investigating the effectiveness of the CodeBERT model for other SE tasks (e.g., bug localization), and (2) applying the CodeBERT model to improve the baselines of further downstream tasks.

Dataset and Code. The dataset and code for this work are available at: https://github.com/Xin-Zhou-smu/Assessing-generalizability-of-CodeBERT.

Acknowledgement. This research / project is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Singapore Data Science Consortium.

REFERENCES

[1] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[2] T. Hoang, K. H. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "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)*, pp. 34–45, 2019.

[3] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *ArXiv*, vol. abs/1710.11054, 2017.

[4] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 95–104, 2019.

[5] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *ArXiv*, vol. abs/1803.04497, 2018.

[6] M. White, M. Tufano, M. Martinez, M. Martin, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 479–490, 2019.

[7] D. Azcona, P. Arora, I. Hsiao, and A. Smeaton, "user2code2vec: Embeddings for profiling students based on distributional representations of source code," *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, 2019.

[8] Z. Chen and M. Martin, "The remarkable role of similarity in redundancy-based program repair," *ArXiv*, vol. abs/1811.05703, 2018.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1 – 29, 2019.

[10] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *ArXiv*, vol. abs/1808.01400, 2019.

[11] D. DeFreez, A. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," *ArXiv*, vol. abs/1802.07779, 2018.

[12] H. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1–12, 2019.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[14] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing [review article]," *IEEE Computational Intelligence Magazine*, vol. 13, pp. 55–75, 2018.

[15] R. Nogueira and K. Cho, "Passage re-ranking with bert," *ArXiv*, vol. abs/1901.04085, 2019.

[16] M. A. S. Cabezudo, M. Inácio, A. C. Rodrigues, E. Casanova, and R. F. D. Sousa, "Natural language inference for portuguese using bert and multilingual information," in *PROPOR*, 2020.

[17] C. Liang, Y. Yu, H. Jiang, S. Er, R. Wang, T. Zhao, and C. Zhang, "Bond: Bert-assisted open-domain named entity recognition with distant supervision," *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[18] M. Munikar, S. Shakya, and A. Shrestha, "Fine-grained sentiment classification using bert," *2019 Artificial Intelligence for Transforming Business and Society (AITB)*, vol. 1, pp. 1–5, 2019.

[19] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?" *ArXiv*, vol. abs/1905.05583, 2019.

[20] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *ArXiv*, vol. abs/1907.11692, 2019.

[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *EMNLP*, 2020.

[22] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *ArXiv*, vol. abs/1909.09436, 2019.

[23] R. Zellers, Y. Bisk, R. Schwartz, and Y. Choi, "Swag: A large-scale adversarial dataset for grounded commonsense inference," in *EMNLP*, 2018.

[24] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100, 000+ questions for machine comprehension of text," in *EMNLP*, 2016.

[25] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," in *BlackboxNLP@EMNLP*, 2018.

[26] A. V. Miceli Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," in *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Taipei, Taiwan: Asian Federation of Natural Language Processing, Nov. 2017, pp. 314–319.

[27] Z. Yao, D. S. Weld, W. Chen, and H. Sun, "Staqc: A systematically mined question-code dataset from stack overflow," *Proceedings of the 2018 World Wide Web Conference*, 2018.

[28] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018.

[29] A. Mockus and D. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, pp. 169–180, 2000.

[30] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. Godfrey, "Investigating code review quality: Do people and participation matter?" *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 111–120, 2015.

[31] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, pp. 181–196, 2008.

[32] Y. Kamei, E. Shihab, B. Adams, A. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, pp. 757–773, 2013.

[33] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 17–26, 2015.

[34] T. Hoang, H. Kang, J. L. Lawall, and D. Lo, "Cc2vec: Distributed representations of code changes," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 518–529, 2020.

[35] C. Sun, A. Myers, C. Vondrick, K. Murphy, and C. Schmid, "Videobert: A joint model for video and language representation learning," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 7463–7472, 2019.

[36] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *ArXiv*, vol. abs/2003.10555, 2020.

[37] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, pp. 33–45, 2016.

[38] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, pp. 531–577, 2011.

[39] X. Gu, H. Zhang, and S. Kim, "Deep code search," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, 2018.

[40] W. Ye, R. Xie, J. lei Zhang, T. xiang Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," *Proceedings of The Web Conference 2020*, 2020.

[41] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, 2011.

[42] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov, "Devise: A deep visual-semantic embedding model," in *NIPS*, 2013.

[43] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[44] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: A big comparison for nas," *ArXiv*, vol. abs/1912.06059, 2019.

[45] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 826–831, 2018.

[46] E. Voorhees, "The trec-8 question answering track report," in *TREC*, 1999.

[47] Y. Kim, "Convolutional neural networks for sentence classification," in *EMNLP*, 2014.
[48] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, pp. 412–428, 2018.
[49] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *HLT-NAACL*, 2016.
[50] I. Goodfellow, Y. Bengio, and A. C. Courville, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.
[51] B. Korbar, A. M. Olofson, A. P. Miraflor, K. M. Nicka, M. A. Suriawinata, L. Torresani, A. Suriawinata, and S. Hassanpour, "Deep learning for classification of colorectal polyps on whole-slide images," *Journal of Pathology Informatics*, vol. 8, 2017.
[52] J. Liu, W.-C. Chang, Y. Wu, and Y. Yang, "Deep learning for extreme multi-label text classification," *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017.
[53] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *ACL*, 2018.
[54] J. Teevan, K. Collins-Thompson, R. W. White, S. Dumais, and Y. Kim, "Slow search: Information retrieval without time constraints," in *HCIR '13*, 2013.
[55] E. Schurman and J. Brutlag, "Performance related changes and their user impact," in *velocity web performance and operations conference*, 2009.
[56] C. Tantithamthavorn, A. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Transactions on Software Engineering*, vol. 46, pp. 1200–1219, 2020.
[57] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *ICML*, 2020.
[58] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: code generation using transformer," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.
[59] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.
[60] L. Buratti, S. Pujar, M. A. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, and G. Domeniconi, "Exploring software naturalness through neural language models," *ArXiv*, vol. abs/2006.12641, 2020.
[61] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *ArXiv*, vol. abs/2009.08366, 2020.
[62] M.-A. Lachaux, B. Rozière, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *ArXiv*, vol. abs/2006.03511, 2020.
[63] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader-Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 336–347, 2021.
[64] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 981–992, 2020.
[65] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *ArXiv*, vol. abs/2102.04664, 2021.