Can Code Representation Boost IR-Based Test Case Prioritization?

Lin Yang, Junjie Chen, Hanmo You, Jiachen Han, Jiajun Jiang, Zhe Sun, Xinqi Lin, Fang Liang, Yuning Kang
College of Intelligence and Computing, Tianjin University, Tianjin, China
{linyang, junjiechen, youhanmo, hanjiachen2, jiangjiajun,
sunzhe158328, linxinqi, fr_3122355109, kangyuning}@tju.edu.cn

***Abstract*—Test case prioritization (TCP) aims to schedule the execution order of test cases for earlier fault detection. A recent study has demonstrated that the information-retrieval-based (IR-based) TCP approaches achieve the state-of-the-art effectiveness. The current IR-based TCP approaches leverage lexical similarity between test cases and code changes to guide TCP while ignoring rich code semantics, which may limit their effectiveness to some degree. In this paper, we conduct the first study to explore whether code semantic information can further boost IR-based TCP. Here, we studied two types of code representation methods (i.e., general-purpose and task-associated models) and explored two modes of utilizing the code representation embeddings (i.e., unsupervised and supervised modes) for IR-based TCP. Our results demonstrate that incorporating code semantics through the supervised mode of code representation can achieve a 16.96% improvement in the efficiency of fault detection over the state-of-the-art IR-based TCP approach (which is based on lexical similarity).**

***Index Terms*—Test Case Prioritization, Code Representation, Information Retrieval, Empirical Study**

I. Introduction

Regression testing has been widely used to ensure the quality of software evolution (i.e., not damaging the existing functionalities of the software). However, due to the large scale of regression test cases and the high frequency of software evolution, regression testing is extremely costly [1], [2], [3]. As demonstrated by the existing study [1], more than 80% of testing time are consumed by regression testing. To improve the efficiency of regression testing, test case prioritization (TCP) has been proposed, which aims to schedule the execution order of test cases in order to detect faults as early as possible [2], [3], [4], [5], [6], [7].

In the literature, many TCP approaches have been proposed [5], [4], [8], [9], [10]. Most of them utilize code coverage information (e.g., statement coverage or branch coverage) to prioritize test cases. However, this kind of TCP approach suffers from some major limitations during practical use [5], [11]. First, collecting code coverage information involves substantial extra time and storage overheads [8]. Second, coverage-based TCP approaches collect code coverage information from a previous version and then prioritize test cases for the latest version of the software. However, they do not consider the information of the version under test, and the information collected from a previous version may be out of date with the latest version (especially when the code changes between versions are significant), leading to ineffective prioritization [5].

To alleviate the above limitations, information retrieval based (IR-based) TCP approaches have been proposed in recent years [9], [4], which prioritize test cases based on the similarity between test cases and code changes. As demonstrated by existing work [9], [4], [12], [13], [14], code changes tend to be more likely to introduce faults in software regression, and thus the test cases more relevant to code changes can be more likely to detect faults. The key insight behind IR-based TCP is that the developers tend to use similar terms from code changes for constructing test cases [9]. Therefore, the textual relationship serves as a connection between test cases and code changes, thereby converting the TCP problem into a conventional IR problem. IR-based TCP can avoid the extra time and storage overheads incurred by code coverage collection and, meanwhile, incorporate the information from the version under test through analyzing code changes. Specifically, it formulates the TCP problem as an IR problem by treating code changes as queries and test cases to be prioritized as a collection of documents. In particular, the recent study has demonstrated that the state-of-the-art IR-based TCP approach (i.e., BM25-based TCP approach) has outperformed the state-of-the-art coverage-based TCP approaches [4]. Although IR-based TCP has more practical significance, current IR-based TCP approaches only measure the similarity between code changes and test cases at the lexical level based on some traditional algorithms (e.g., BM25 [15] or TF-IDF [16]), but ignores rich semantic information embodied in code. Such ignorance may limit the effectiveness of IR-based TCP, but it is still unexplored till now.

Due to the state-of-the-art effectiveness and practicability of IR-based TCP, it is definitely valuable to further improve its effectiveness. In this work, we conducted the first empirical study to explore whether incorporating code semantic information is helpful to boost IR-based TCP. Specifically, we adopted code representation learning to extract semantic information from both code changes and test cases, since this kind of methods have been demonstrated to be effective in many software engineering tasks [17], [18], [19], [20]. In our study, we adopted two widely used code representation methods (i.e., Doc2Vec [21] and ASTNN [22]) as representatives to explore whether this direction is promising for IR-based TCP.

*Junjie Chen is the corresponding author for this work.

Doc2Vec is a typical general-purpose code representation method, which builds a model by predicting the next word (i.e., token in code representation) in the document (i.e., code snippet) following the idea of CBOW [23]. ASTNN is a typical task-associated code representation method that builds a model by incorporating both textual and structural information from the AST corresponding to a code snippet based on the data for a downstream task.

In particular, we investigated two usage modes of code representation to boost IR-based TCP. (1) *Unsupervised mode*: After representing test cases and code changes as vectors via a pre-trained code representation model, we adopt an unsupervised way (i.e., similarity calculation between vectors) to solve the TCP task. (2) *Supervised mode*: After obtaining vectors through code representation, we then build a model based on a supervised learning algorithm to solve our TCP task by treating these vectors as feature vectors and labeling them according to our task, which can predict the fault-triggering probability of a test case to be prioritized.

Our experimental results on two Java benchmarks (i.e., one is from the existing study [4], another is collected by us from GitHub) show that: (1) The semantic information encoded by the pre-trained code representation models (especially Doc2Vec) can make the similarity between fault-triggering test cases and code changes higher than that between non-fault-triggering test cases and code changes. This is the foundation on which code representation may boost IR-based TCP. (2) Utilizing the pre-trained code representation models for the IR-based TCP task in the unsupervised mode (i.e., similarity calculation between vectors) performs slightly worse than the state-of-the-art IR-based TCP approach (i.e., the BM25-based approach [4]), where Doc2Vec performs better than ASTNN. (3) Code representation has the potential to improve the effectiveness of IR-based TCP in the supervised usage mode. However, the improvement depends on the code representation method adopted. In particular, ASTNN in the supervised usage mode achieves a 18.64% improvement over the state-of-the-art BM25-based TCP approach in terms of average APFD. That is, during our exploration in the extensive study, we first obtained a negative conclusion on utilizing code representation for boosting IR-based TCP (i.e., in the unsupervised mode), but finally found a possible way that can largely improve the effectiveness of IR-based TCP (i.e., in the supervised mode) and is worthy for future exploration.

This paper makes the following major contributions:

- We are the first to propose the incorporation of code semantic information to boost IR-based TCP.
- We conducted an extensive study to explore the effectiveness of two usage modes of code representation for IR-based TCP. In particular, certain type of code representation used in the supervised mode can effectively boost IR-based TCP.
- We delivered a series of findings and released our code [24] for promoting future research and practice.

The remaining of this paper is organized as follows. Section II introduces basic background about TCP and code representation. Section III presents the design of our extensive study. Section IV reports our experimental results and findings. Section V discusses the generality of our study and the threats to validity in our study. Section VI presents the related work. Section VII concludes our work.

II. Background

A. IR-based Test Case Prioritization

Test case prioritization (TCP) aims at finding the optimal execution order of test cases, which satisfies a given property or target, such as detecting as many faults as possible within a given execution time budget [25]. Formally, when providing a set of test cases T, we use PT to denote all possible execution orders of test cases in T, and we use f to denote the function that maps an instance $pt \in PT$ to the desired property or target, such as the aforementioned number of detected faults. In this way, the target of TCP can be formalized as finding a $pt \in PT$ that $\forall pt' \in PT, f(pt) \succeq f(pt')$, where $\succeq$ denotes the superiority relation. In particular, in this paper, we define the target function f as the average percentage of fault detection (to be defined in Section IV-B1), and $\succeq$ will be $\geq$ for the numeric comparison.

As described before, coverage-based and information retrieval (IR)-based TCP approaches are two widely studied categories of TCP. The former have been widely studied in the last decades [26], while the latter were less explored. The most recent study [4] reported that IR-based TCP approaches can achieve competitive effectiveness compared with coverage-based TCP approaches, but require less execution and storage overheads. However, most of the existing IR-based approaches are still simple as they rarely consider the deep semantic information of source code. For example, the latest IR-based TCP approach (i.e., BM25-based TCP approach [4]), which achieves the best effectiveness over all IR-based TCP approaches, only considers the lexical features of source code like natural language (i.e., the term frequency and the length of data objects). Many existing studies [27], [28], [29] have revealed that the semantic information of the source code was valuable and proved to be effective in boosting many downstream applications, such as code search [27] and fault prediction [30].

In this paper, our aim is to perform the first study to fill the gap and explore the possibility of incorporating semantic information from source code to boost the effectiveness of IR-based TCP.

B. Code Representation

Source code representation is a fundamental task in many application scenarios related to program analysis in the software engineering community, such as code search [27], code clone detection [31], [32], and program repair [33]. The target of such a task is to learn a vector representation of source code, which hopefully can embed syntactic and/or semantic characteristics delivered by human developers in source code. However, it is usually not easy to achieve that. Unlike natural

language, whose semantics usually can be conveyed by keywords, the semantics of source code are usually complex and embedded by rigorous computation logic. In recent decades, researchers have developed different methods to improve the performance of code representation for different tasks.

In the early stage, source code was usually embedded based on syntactic features. For example, Jiang et al. [31] proposed Deckard, which uses the number of different types of AST nodes to represent the source code for code clone detection. Based on this, Jiang et al. [33] further incorporated text similarities (i.e., variable and method names) to measure the semantic distance of the source code for automatic program repair. However, these approaches can hardly take advantage of the deep semantics of source code. Recently, researchers proposed using deep learning models to aid in the embedding of source code. For example, TECCD [32] uses random walk over the AST of the source code to better encode the syntactic features, while ASTNN [22] uses a tree-based bidirectional GRU [34] encoder to extract both the syntactic and semantic features of the code. These approaches have been demonstrated to be much more effective [35] in extracting features from the source code and thus can produce better code representations. In this paper, we will explore whether IR-based TCP will also benefit from the advanced progress in source code representation by incorporating typical code representation methods into the TCP task.

III. Study Design

In this section, we present the design of our empirical study. Intuitively, if code representation can be helpful to the task of IR-based TCP, it should be able to make the semantic similarity between code changes and fault-triggering test cases higher than that between code changes and non-fault-triggering test cases. This is the foundation of our exploration. Therefore, our study addresses the first research question (**RQ1**): *Is the semantic similarity (measured by code representation) between code changes and fault-triggering test cases higher than that between code changes and non-fault-triggering test cases?* In particular, after representing both code changes and test cases as vectors, respectively, we designed four strategies of calculating semantic similarity based on these vectors (to be presented in Section III-D).

After validating the foundation, we investigated whether code representation can improve the effectiveness of IR-based TCP. In practice, there are two typical usage modes of code representation, and we investigated the influence of them on the effectiveness of IR-based TCP in our study. (1) *Unsupervised mode*: after representing code changes and test cases as vectors through code representation, we adopted an unsupervised method (i.e., calculating the semantic similarity between vectors) to solve the TCP task. (2) *Supervised mode*: we built a model based on some supervised learning algorithm to solve the TCP task by treating these vectors through code representation as feature vectors and labeling them according to the task, which can predict the fault-triggering probability of a test case to be prioritized.

Therefore, our study addresses the second research question (**RQ2**): *Can code representation improve the effectiveness of IR-based TCP in the unsupervised mode?* Accordingly, our study addresses the third research question (**RQ3**): *Can code representation improve the effectiveness of IR-based TCP in the supervised mode?*

A. Studied Code Representation Methods

In general, there are two types of code representation methods: *general-purpose* and *task-associated* code representations. The former builds a model by learning from the general corpus without specific labels, while the latter builds a model based on the data specific to a downstream task. In the study, we selected a typical code representation method for each type (Doc2Vec [21] from the general purpose code representation and ASTNN [22] from the task-associated code representation), respectively.

As the first study investigating whether code representation can boost IR-based TCP, we used the two typical methods as the representative. Actually, there are some more advanced code representation methods [27], [28], but our study aims to explore the feasibility of boosting IR-based TCP by code representation rather than evaluating all the code representation methods in the task. Moreover, many advanced methods strictly limit the maximum length of their input code [36], [37], and the two selected methods are more general in this aspect. Therefore, we adopted the two methods as the representative for the first exploration, and in the future, we can investigate more advanced code representation methods to further improve the performance. In the following, we introduce the two methods in detail.

1) Doc2Vec: Doc2Vec [21] is a typical general-purpose code representation method to produce fixed-length semantic representation for variable-length text. Inspired by Word2Vec [23], Doc2Vec designs a distributed memory model of paragraph vectors (PV-DM) to learn semantic representation of a document. It trains a language model by predicting the next word in the document following the idea of CBOW [23]. Specifically, each word in the document is first represented as a fixed-length vector, and the vocabulary V becomes a weight matrix W. Then, for a given T-length document $[w_1, w_2, \ldots, w_T]$, the goal of Doc2Vec is to maximize the log probability during the training process:

$$\frac{1}{T}\sum_{t=k}^{T-k} log(p(w_t|v; w_{t-k}, \ldots, w_{t+k})) \tag{1}$$

Here, $p(w_t|w_{t-k}, \ldots, w_{t+k})$ is the probability that the word w_t appears in the context of $w_{t-k}, \ldots, w_{t+k}$. In addition to the word sequence, Doc2Vec adds a vector $v(t)$ before each context as the embedding for the document, and it finally serves as the document representation. Further, the probability of the next word is calculated via a multi-class classifier as shown in Formula 2:

$$\begin{aligned} p(w_t|v(t); w_{t-k}, \ldots, w_{t+k}) &= \frac{e^{y_t}}{\sum_{i \in V} e^{y_i}} \\ y_i &= Uh(v(i); w_{i-k}, \ldots, w_{i+k}) + b \end{aligned} \tag{2}$$

In this formula, U and b are the weight matrix and bias vector, respectively. h is the hidden state of the document and the context, which is constructed by the concatenation of word embeddings and document representation.

By treating code as text, Doc2Vec can be used for code representation. In our study, we used the pre-trained language model through Doc2Vec on the publicly available code dataset (constructed by Allamanis et al. [38]) to represent both code changes and test cases. This dataset contains over 352 million lines of code from 14,807 Java projects on GitHub. When applying Doc2Vec to code representation in our study, we pre-processed code by splitting combined tokens (e.g., getInputStream) to separate tokens based on two popular naming schemes (i.e., Camel Case [39] and Snake Case [39]). Then, a token after preprocessing in the source code is regarded as a word and a method is regarded as a document, and thus the pre-trained Doc2Vec model can represent a method as a vector.

2) ASTNN: ASTNN [22] is a typical task-associated code representation method, it extracts textual information (i.e. tokens in source code) and structural information from an AST corresponding to a method for code representation. Specifically, ASTNN transforms an AST to a sequence of statement-level subtrees. For each subtree, ASTNN designs a tree-based Non-Linear Aggregation network, where the representation of each non-leaf node is calculated by its child nodes through a Non-Linear Layer. The hidden state of the non-leaf node n (with C child nodes) can be calculated as shown in Formula 3.

$$h_n = \sigma(W_n^T v_n + \sum_{i \in C} h_i + b_n) \tag{3}$$

In this formula, W_n refers to the Linear Weight matrix, v_n refers to the token embedding obtained by Word2Vec [23], and b_n refers to the bias vector. Subsequently, ASTNN adopts a MaxPooling Layer to obtain the representation of the subtree. Finally, ASTNN adopts a Bidirectional Gated Recurrent Unit Network (GRU) [34] to encode the subtree sequence as the final AST representation.

For the sake of cost-effectiveness, we also adopted the pre-trained ASTNN model in our study. Among all the released pre-trained ASTNN models, the one based on the code clone detection task [22] is the closest to our task. This is because both tasks aim to capture the semantic relationship between *code snippets* (even though they target different types of code snippets and aim to capture different kinds of semantic relationship). Specifically, it was built on BigCloneBench [40], which is one of the most widely used benchmarks for code clone detection in Java code. It contains more than 6 million pairs of known clone and non-clone method extracted from the big data interproject repository *IJaDataset* [41] (25,000 subject systems, totaling 365M LOC).

As introduced before, in RQ2, we investigated the effectiveness of the unsupervised mode of code representation for boosting IR-based TCP by calculating the similarity between vectors represented by the two studied pre-trained models. Details on how to utilize the unsupervised mode of code representation to prioritize test cases will be presented in Section III-D. After that, in RQ3, we designed a supervised learning framework based on the two pre-trained models and investigated the effectiveness of the supervised mode of code representation for TCP. The corresponding supervised learning framework to prioritize test cases will be presented in Section IV-C.

TABLE I
BASIC INFORMATION OF BENCHMARKS

	#Proj	#Job	Code Change		Test Case	
			#class	#method	#class	#method
D2980	123	2,980	54.85K	684.59K	3.83M	523.89K
D237	8	237	1.38K	63.62K	472.82K	57.88K

B. Benchmarks

In the study, we evaluated our studied IR-based TCP approaches on two benchmarks (denoted as D2980 and D237 in this paper for ease of presentation).

D2980: This benchmark is the one used by Peng et al. [4] to evaluate the effectiveness of the existing IR-based TCP approaches. It consists of 2,980 test jobs (from 2,042 Travis builds) of 123 popular Java projects from GitHub, all of which are built with Maven [42]. Each test job in this benchmark contains test failures, but the Travis build of the corresponding prior commit is "passed", which can ensure that the test failures are caused by the code changes between the two commits. More details about this benchmark can be found on its homepage [43].

D237: This benchmark is collected by us in this study. Since we need to build a model with a supervised learning algorithm on the data specific to our task in RQ3, we used the D2980 benchmark as the training data and then constructed the D237 benchmark to measure the TCP effectiveness. Specifically, following the process of constructing the D2980 benchmark, we manually collected 237 test jobs of 8 popular Java projects from GitHub (i.e., Apache Commons CLI [44], Apache Commons Compress [45], Apache Commons Codec [46], Apache Commons CSV [47], Apache Commons Math [48], Jackson-Core [49], JacksonXML [50], JFreeChart [51]). Note that there is no overlapped project between the two benchmarks, in order to avoid data leakage.

Table I shows the basic information of the two benchmarks, including the number of included Java projects, the total number of test jobs, the total number of classes and methods involved in code changes, and the total number of test classes and test methods.

C. Data Preparation

For each test job in both benchmarks, it contains both code changes and test cases (also indicating fault-triggering test cases). As demonstrated by the existing study [4], treating classes that include code changes as queries can achieve better effectiveness for IR-based TCP than using only the code

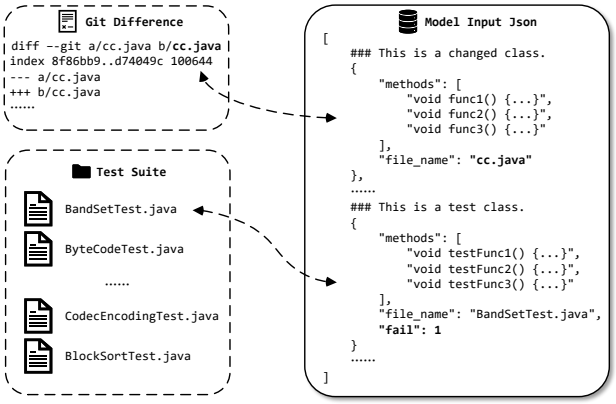

Fig. 1. Overview of data preparation

changes. Therefore, in our study, we also adopted this setting. Furthermore, following the existing study [4], we prioritize test cases at the test class level, and the D2980 benchmark provides only the information whether a test class fails or passes corresponding to code changes. Please note that we also discussed the generality of the study on method-level TCP in Section V-A. That is, each test class is regarded as a document. In particular, to encode semantic information more precisely, we applied each pre-trained code representation model to each method (i.e., each method in each changed class or each test method in each test class to be prioritized) and then designed four strategies of similarity calculation based on these method-level vectors for TCP (to be presented in Section III-D). Therefore, we organized the information from each test job as shown in Figure 1, that is, the changed classes (each of which consists of a set of methods) and the test classes (each of which consists of a set of test methods).

D. Similarity Calculation Strategies for TCP

As described above, when boosting IR-based TCP through code representation, we applied a pre-trained code representation model (built with Doc2Vec or ASTNN) to represent each test method or each method in changed classes as a vector. More formally, a test class (TC) consists of a set of test methods (TM_1, TM_2, ..., TM_{nt}). A test method (TM_i) is encoded by a pre-trained code representation model as vector $tv_i = \{tx_{i1}, tx_{i2}, \ldots, tx_{im}\}$. Also, the changed classes contain a set of methods (SM_1, SM_2, ..., SM_{ns}), each of which (SM_j) is encoded by the pre-trained code representation model as the vector $sv_j = \{sx_{j1}, sx_{j2}, \ldots, sx_{jm}\}$. Based on those method-level vectors, we design four strategies of similarity calculation in our IR-based TCP enhanced by code representation. All four strategies share the following workflow:

1 Measuring the cosine similarity between tv_i ($1 \leq i \leq nt$) in each test class and sv_j ($1 \leq j \leq ns$) in changed classes;
2 Determining the similarity between a test method and code changes by using the **Mean** or **Max** similarity between the test method and all the methods in changed classes (we call the similarity *test-method-level similarity* for ease of presentation);
3 Determining the similarity between a test class and code changes by using the **Mean** or **Max** test-method-level similarity of all test methods in the test class (we call the similarity *test-class-level similarity* for ease of presentation);

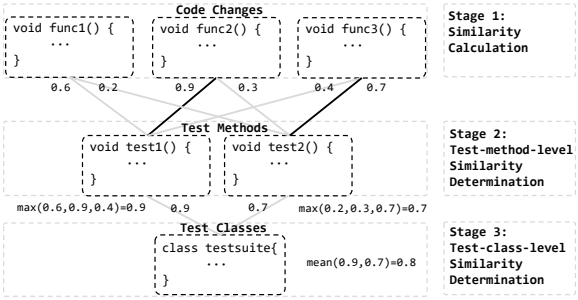

Fig. 2. Illustrating example of the Max-Mean strategy

According to the ways of determining the test-method-level similarity and the test-class-level similarity, we have four specific strategies to calculate the similarity between test classes and code changes in total. For ease of presentation, we denote them as **Mean-Mean**, **Mean-Max**, **Max-Mean**, and **Max-Max**, respectively, where the one before "-" refers to the way of determining test-method-level similarity and the one after "-" refers to the way of determining test-class-level similarity. As shown in Figure 2, we also use an example for a better illustration by taking the **Max-Mean** strategy as representative.

In particular, *in the unsupervised mode of code representation, we prioritized test classes in the descending order of their test-class-level similarity*. Due to the four similarity calculation strategies, we implemented four specific IR-based TCP approaches enhanced by code representation in the unsupervised mode, which will be evaluated in RQ2. Please note that if some test classes also involve code changes, we put these test classes to the beginning of our prioritization result as their original order in the test suite following the existing work [4] (regardless of the supervised or unsupervised mode).

E. Implementation

We implemented our IR-based TCP approaches enhanced by code representation in Python 3.6 and Numpy 1.14.2. For the two pre-trained code representation models, we directly adopted the public implementations of Doc2Vec from gensim 3.5.0 [52] and ASTNN from the author-provided Github repository [53]. All our experiments were conducted on a server with an Ubuntu 20.04 system with Intel(R) Xeon(R) Silver 4214 @ 2.20GHz CPU, 256GB memory, and NVIDIA GeForce RTX 2080 Ti GPU.

We also released our code and experimental data on our homepage for promoting future research and practical use. More details about our implementations can be found in our code package:**https://github.com/youhanmo/sstcp/**.

IV. Results and Analysis

In this section, we answer each research question by introducing the corresponding experimental process and analyzing the corresponding results.

A. RQ1: Semantic similarity between test classes and code changes

*1) **Process**:* To answer RQ1, for each pre-trained model (based on Doc2Vec or ASTNN) on each test job in the D2980 benchmark, we calculated the test-class-level similarity for each test class by using each of the four strategies, respectively. Therefore, we divided the test-class-level similarity results into two groups according to whether the test class failed or passed on the test job for comparison. We show the results of the comparison with box plots in Figure 3.

*2) **Results**:* From Figure 3, for Doc2Vec, the box representing failed test classes is always higher than that representing passed test classes. However, for ASTNN, the box representing failed test classes is relatively close to that representing passed test classes. That is, the similarity between test classes and code changes based on Doc2Vec is more distinguishable for fault-triggering test classes (failed test classes) and non-fault-triggering test classes (passed test classes) than that based on ASTNN. We suspect the reason may be that the pre-trained ASTNN model used in our study is built based on the task of code clone detection. Although it is close to our task to some extent, it does not intend to distinguish failed and passed test cases. However, the pre-trained Doc2Vec model is a general-purpose model and thus may perform stably well on most of the tasks.

Furthermore, regardless of ASTNN or Doc2Vec, the Max-Max strategy performs the best among the four strategies. On the one hand, it can achieve the highest similarity between test classes and code changes, regardless of failed test classes or passed test classes. On the other hand, it can better distinguish the similarity for failed test classes and passed test classes. One possible reason is that a fault tends to be caused by a small portion of code changes and affects a small set of test methods (even a single test method). Identifying the small set of failed test methods and their associated code changes and then using their similarity to measure the similarity at the test class level can be more effective. To some degree, the MAX operation can help retain the effect of these failed test methods and their associated code changes, while the Mean operation can weaken the effect.

To sum up, the foundation on which code representation may boost IR-based TCP is validated, especially for the pre-trained Doc2Vec model and the Max-Max strategy.

> **Answer to RQ1**: Code changes tend to be more semantically relevant to fault-triggering test cases than non-fault-triggering test cases through code representation, where the pre-trained Doc2Vec model with the Max-Max strategy achieves the most significant result.

B. RQ2: Effectiveness of IR-based TCP based on the Unsupervised Usage Mode of Code Representation

*1) **Metric**:* Following the existing work on test case prioritization [9], [4], [8], [5], [2], [54], [55], [56], [57], we adopted APFD (i.e. Average Percentage of Failure Detection) [26] to measure the effectiveness of each studied IR-based TCP approach. If the APFD value is larger, it indicates that the effectiveness of the TCP approach is better. The APFD value ranges from 0 to 1, which is calculated as shown in Formula 4. In this formula, TF_i refers to the first test case in the prioritized result that detects the i_{th} fault, n refers to the number of test cases to be prioritized, and m refers to the number of faults detected by the entire test suite.

$$APFD = 1 - \frac{\Sigma_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{4}$$

*2) **Process**:* RQ1 has validated the foundation of applying code representation to the task of IR-based TCP. In RQ2, we investigated the effectiveness of IR-based TCP approaches enhanced by code representation in the unsupervised mode by comparing with the state-of-the-art IR-based TCP approach (i.e., BM25-based approach) on the D2980 benchmark. In particular, we studied two pre-trained code representation models with four prioritization strategies in the unsupervised mode, and thus we implemented eight code-representation-enhanced IR-based TCP approaches in this experiment. For each studied IR-based TCP approach on each test job (i.e., each pair of code changes and test cases to be prioritized), we obtained the prioritization result and measured its APFD value.

*3) **Results**:* Figure 4 shows the comparison results between our eight code-representation-enhanced IR-based TCP approaches (based on pre-trained code representation models in the unsupervised mode) and the state-of-the-art BM25-based TCP approach. As expected, the combinations of a pre-trained code representation model and a similarity calculation strategy that can better distinguish failed test classes and passed test classes by measuring the semantic similarity with code changes (shown in Figure 3), indeed achieve the larger APFD values, indicating that the corresponding TCP approaches achieve better effectiveness. Specifically, the TCP approach based on the pre-trained Doc2Vec model with the Max-Max strategy performs the best among all the eight approaches enhanced by code representation. Its mean APFD value across all test jobs is 0.72.

However, the best code-representation-enhanced IR-based TCP approach still performs slightly worse than the state-of-the-art BM25-based TCP approach. The average APFD value of the latter across all the test jobs is 0.75. The conclusion actually goes against our intuition, that is, code semantic information can be more helpful to capture the correlation between code snippets than textual information. The possible reason lies in that our used code representation models are pre-trained for the general purpose (Doc2Vec) or the other task (ASTNN for the task of code clone detection), and thus cannot well capture the semantic relationship between

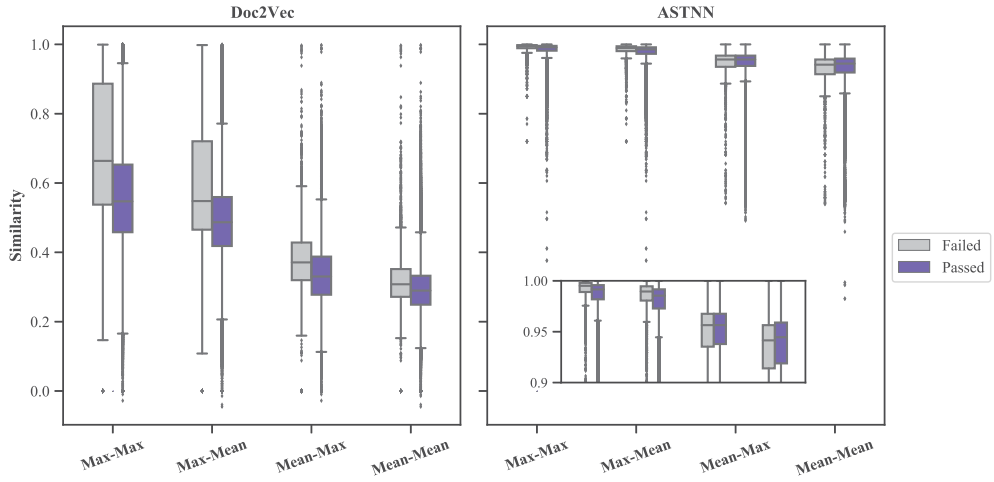

Fig. 3. Class-Level similarity distribution of code representation methods on D2980.

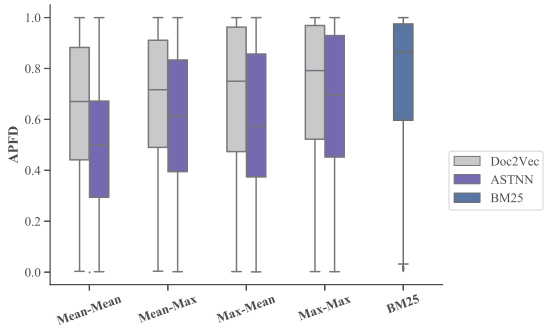

Fig. 4. APFD value distribution of the studied approaches (unsupervised mode) on D2980 for class-level TCP

code changes and test cases required by the task of IR-based TCP. Inspired by the practice of supervised mode of code representation, its effectiveness on a specific task can be improved by using the code-representation vectors to build the supervised model of solving the specific task based on the corresponding labeled data. This further motivates our third research question presented in the next section.

Answer to RQ2: The unsupervised usage mode of the pre-trained code representation models cannot effectively boost IR-based TCP to outperform the state-of-the-art BM25-based TCP approach.

C. RQ3: Effectiveness of IR-based TCP based on the Supervised Usage Mode of Code Representation

1) ***Supervised Learning Framework****:* After code representation, it is also common to utilize these vectors to solve the target task in a supervised mode, i.e., building a model with a supervised machine learning or deep learning algorithm on the training data specific to the target task. For each training instance, the feature vector is the code representation vector while the label is assigned according to the specific task. In this experiment, we aim to investigate whether the supervised usage mode of code representation can help improve the effectiveness of IR-based TCP. In our TCP task, each test class can be regarded as an instance. For each test class, we extracted the pair of test and changed method with the largest semantic similarity based on the Max-Max strategy (due to its effectiveness demonstrated in Section IV-B), and then concatenated the vectors of this pair of methods through code representation as the feature vector. Then, we labeled an instance as 1 (if the test class is fault-triggering) or 0 (if the test class is non-fault-triggering).

Based on the training data, we applied some machine learning or deep learning algorithm to build a supervised model, which can predict the fault-triggering probability of a test class to be prioritized. As the descending order of the predicted probabilities for the test classes to be prioritized, the prioritization result can be obtained. For ease of presentation, we call such IR-based TCP enhanced by code representation in the supervised mode ***SSTcp***. We also use Figure 5 to further illustrate the workflow of *SSTcp*.

To adequately investigate the effectiveness of *SSTcp*, in this experiment, we also studied the two code representation methods (i.e., Doc2Vec and ASTNN). Furthermore, we studied three widely used supervised learning algorithms to build the prediction model, including Random Forest (RF) [58], Logistic Regression (LR) [59], and Neural Network (NN) [60]. In total, we implemented 6 specific *SSTcp* approaches. In particular, we adopted the implementations of RF and LR provided by scikit-learn [61] and NN provided by PyTorch to implement the *SSTcp* approaches. We determined their parameter settings based on a small dataset and released the parameter settings on our project homepage.

2) Process: In the experiment, we used the D2980 benchmark as training data and evaluated the effectiveness of *SSTcp* on the D237 benchmark. This is because the D2980 benchmark contains more data, which is helpful in building

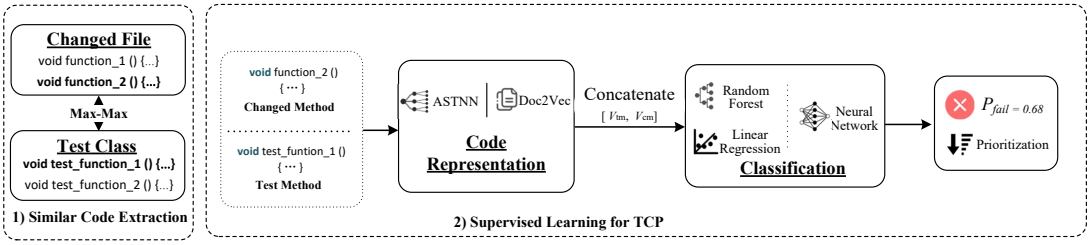

Fig. 5. Overview of *SSTcp*

TABLE II
COMPARISON IN TERMS OF APFD ON D237 FOR CLASS-LEVEL TCP

Metric	Doc2Vec				ASTNN				BM25
	SIM	LR	RF	NN	SIM	LR	RF	NN	
Avg.	0.57	**0.64**	0.54	**0.64**	0.59	0.38	0.33	**0.67**	0.56
Mid	0.58	**0.68**	0.56	**0.68**	0.59	0.36	0.27	**0.71**	0.59

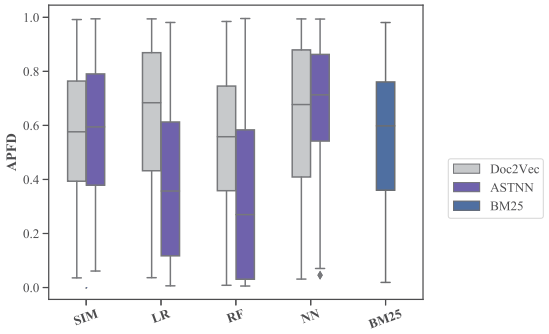

Fig. 6. APFD value distribution of the studied approaches (supervised mode) on D237 for class-level TCP

the prediction model with supervised machine learning or deep learning. In particular, there is no overlap between the two benchmarks, and thus this setting can avoid data leakage. Since the number of failed test cases tends to be much smaller than the number of passed test cases, our training data suffer from the imbalance issue. To alleviate this issue, we randomly selected negative samples (i.e., the instances for passed test cases) from the training data and made the ratio of negative samples to positive samples be 3:1 for model training, following the idea of Hard Negative Mining [62], [63]. To reduce the influence of randomness, we repeated the experiment ten times and calculated the average results. Here, we also adopted the state-of-the-art BM25-based TCP approach as the baseline and used APFD as the metric to measure the effectiveness of each studied TCP approach.

*3) **Results**:* Table II and Figure 6 present the comparison results between our six *SSTcp* approaches and the state-of-the-art BM25-based TCP approach. We also reported the results of the most effective IR-based TCP approach based on code representation in the unsupervised mode (i.e., the approach based on Doc2Vec or ASTNN with the Max-Max strategy) for sufficient comparison (denoted as **SIM** in Table II and Figure 6). In Table II, we report the average and median APFD values for all test jobs in the D237 benchmark for each TCP approach. In Figure 6, we used box plots to show the distribution of the APFD values on all test jobs for each TCP approach.

From Table II and Figure 6, we can see that ASTNN with NN performs much better than the state-of-the-art BM25-based TCP approach and the IR-based TCP approach based on code representation in the supervised mode when using the Max-Max strategy. To investigate the statistical significance of the difference, we further adopted the Wilcoxon rank sum test [64], [65], [66] at the significance level of 0.05. The result confirms that ASTNN significantly outperforms BM25 with *p-value*=0.001 (< 0.05). In particular, the effect size is 2.74 based on the Hedges' g metric [67]. On the contrary, although the average APFD of Doc2Vec with NN is slightly better, the difference between it and the BM25 is not statistically significant (*p-value*=0.19).

The results demonstrate that code representation can boost IR-based TCP in the supervised usage mode to some degree. However, the improvement can vary and depend on the selected code representation since only ASTNN with NN can significantly outperform the baseline method.

In addition, we found that NN performs more stably and better than RF and LR in terms of the APFD results, regardless of Doc2Vec or ASTNN. For example, NN improves RF and LR by 18.52% and 0.14% when using Doc2Vec respectively and by 103.03% and 76.32% when using ASTNN respectively, in terms of average APFD. The results demonstrate that NN is more suitable for our task in the supervised mode of code representation than the other two. The possible reason for the superiority of NN lies in that the mechanism behind NN is different from LR and RF. NN learns the relationship between labels and code representation vectors via an additional network and thus can fit the difference between fault-triggering test cases and non-fault-triggering ones. On the contrary, LR and RF try to find a "boundary" between the two types of test cases, and thus their performance is highly relevant to the quality of code representation. Actually, *SSTcp* can also integrate other machine learning or deep learning algorithms to build the prediction model, in addition to the three algorithms studied. In the future, we will further improve the effectiveness of *SSTcp* by incorporating more advanced machine learning or deep learning algorithms.

Answer to RQ3: Code representation has the potential to improve the effectiveness of IR-based TCP in the supervised usage mode. However, the improvement depends on the adopted code representation method. In particular, ASTNN can effectively improve the effectiveness of IR-based TCP.

V. Discussion

A. SSTcp on method-level TCP

Our study has demonstrated that code representation can boost IR-based TCP at the test-class level in the supervised mode. In the literature, method-level TCP is also common. Hence, we further discussed whether *SSTcp* could be generalized to boost IR-based TCP at the test-method level. To apply *SSTcp* to method-level TCP, it treats each test method as an instance. Specifically, it identifies the changed method that has the largest semantic similarity to the test method and then concatenates the vectors of this pair of methods as the feature vector. It labels an instance as 1 (if the test method is fault-triggering) or 0 (if it is non-fault-triggering). After dealing with the imbalance issue in the training data as before, it builds a model to predict the probability of triggering a fault for a test method to be prioritized. Finally, the method-level prioritization result can be produced according to the descending order of the predicted probabilities of the test methods to be prioritized.

This experiment shares the same process as that for RQ3 (presented in Section IV-C). Table III and Figure 7 present the results of our *SSTcp* approaches (considering both Doc2Vec and ASTNN with the same NN, RF and LR algorithms as RQ3), the state-of-the-art BM25-based TCP approach, and the most effective approach based on code representation in the unsupervised mode (denoted as SIM in the table and figure), in terms of APFD. We can obtain the same conclusion as in RQ3. That is, *SSTcp* with NN can also outperform the state-of-the-art BM25-based TCP approach and the most effective approach based on code representation in the unsupervised mode, for method-level TCP. For example, the former improves the latter two by 50.00% and 134.38% when using Doc2Vec respectively and by 34.00% and 13.56% when using ASTNN respectively, in terms of average APFD. We also performed the Wilcoxon rank sum test [64] to evaluate whether *SSTcp* with NN significantly outperforms the BM25-based TCP approach here. Different from the results from RQ3, the p-values are also much smaller than 0.05, demonstrating the significant performance of *SSTcp* with NN in method-level TCP. The results further confirm the power of code representation in supervised mode to improve the effectiveness of IR-based TCP. In the future, we can incorporate more advanced code representation methods and learning algorithms for supervised model training, in order to further boost IR-based TCP.

B. Lexical Information vs Semantic Information

According to our empirical study, utilizing pre-trained code presentation models in the unsupervised usage mode cannot outperform the state-of-the-art BM25-based approach, which just utilizes lexical information from code changes and test cases. However, when we adopted the supervised mode of the pre-trained code representation models, it can largely outperform the lexical-information-based TCP approach. The findings indicate that when trying to solve a specific task, the semantic information extracted by code representation for general purpose or other tasks may not perform better than the lexical information. When incorporating information from the target task (such as through our supervised learning framework *SSTcp*), the encoded semantic information can be more useful for the target task. In the future, we may also fine-tune the pre-trained models based on the labeled data specific to our task to obtain more precise semantic information for our task, and then further improve the effectiveness of IR-based TCP, instead of the current way of building a supervised model based on the vectors from the pre-trained code representation model.

In addition, we also found that for a few test jobs, *SSTcp* cannot outperform the lexical-information-based approach, indicating that the two kinds of information may be complementary for the task of IR-based TCP to some degree. In the future, we may combine both information in an effective way to further boost IR-based TCP.

TABLE III
Comparison in terms of APFD on D237 for method-level TCP

Metric	Doc2Vec				ASTNN				BM25
	SIM	LR	RF	NN	SIM	LR	RF	NN	
Avg.	0.32	0.72	0.52	**0.75**	0.59	0.40	0.38	**0.67**	0.50
Mid	0.26	0.82	0.49	**0.83**	0.58	0.41	0.34	**0.71**	0.47

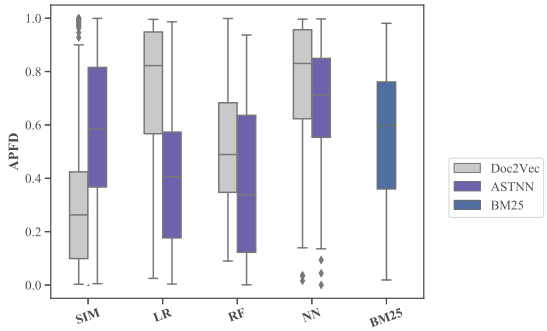

Fig. 7. APFD value distribution of the studied approaches (supervised mode) on D237 for method-level TCP

C. Threats to validity

The *internal* threat to validity lies mainly in the implementation of our experimental scripts and the TCP approaches. To reduce this threat, we adopted the implementation of the compared approach released by the authors, and carefully checked all of our source code.

The *external* threat to validity lies mainly in the benchmarks and code representation models used in our study. In our study, we used both the benchmark released by the existing work

and the benchmark collected by us for a sufficient evaluation. However, they may not represent other projects with other programming languages. To reduce this threat, we plan to extend our study on more diverse projects in the future. In addition, to mitigate the threat in model selection, we adopted Doc2Vec (representing general purpose methods) and ASTNN (representing task-associated methods) as representatives because of their good performance and easy to deploy. The latest studies [68], [69] have demonstrated that large language models (LLMs) are effective in many applications. However, the existing study [70], [71] showed that ASTNN can achieve comparative effectiveness compared to the representative LLM CodeBERT in code clone detection, which is a very close task to ours. Furthermore, LLMs tend to achieve suboptimal performance without fine-tuning on targeted tasks [72], [73] and it is also hard and time consuming to construct the fine-tuning data in our application. As a result, we do not include LLMs in our study. Nevertheless, our conclusion that code representation can boost the IR-based TCP will not be affected, and its performance can potentially be further improved by more advanced and effective representation models. In the future, we plan to carry out more comprehensive studies to investigate which models are more effective in our task by incorporating more diverse representation models. Furthermore, since the BM25-based TCP method was not designed for code representation, we did not include it when studying the supervised usage mode. In the future, we plan to conduct more comprehensive studies to investigate the performance of the supervised methods with traditional features.

The *construct threats to validity* lie mainly in the metric, randomness, the studied code representation methods, and the parameter settings in our study. Following existing studies [4], [9], [8], [5], [2], [54], [55], [56], [57], we adopted APFD to measure TCP effectiveness. In the future, we will extend our study with more testing properties (e.g., execution time and severity) and then use more comprehensive metrics (e.g. $APFD_c$ [74]) to evaluate the effectiveness of TCP more adequately. To reduce the threat of randomness due to the problem of imbalance (in Section IV-C), we repeated this experiment ten times and then used the average results as representative. Regarding the studied code representation methods, as explained in Section III-A, we selected one method from each type as the representative for the first exploration and can study more code representation methods in the future. To reduce the threat from parameter settings, we determined the parameter settings based on a small dataset and released them on our project homepage.

VI. Related work

To improve testing effectiveness and efficiency, many TCP approaches have been proposed in the literature. The major difference between them is the information resource used to guide the TCP process. Specifically, test coverage is a widely used information resource, e.g. statement coverage [26], branch coverage [75], function coverage [76], method coverage [77] and so on. Furthermore, Lin et al. [78] proposed to incorporate the fault detection capability of test cases in historical versions. Hettiarachchi et al. [56] utilized a fuzzy expert system to assign risk values on requirements as prioritization guidance. Wang et al. [55] proposed a quality-aware TCP approach based on the fault proneness of test cases, while Chen et al. [57] used log information to boost black-box test case prioritization. Lou et al. [5] proposed to employ mutation analysis as TCP guidance. Furthermore, a series of TCP approaches used IR methods, including TF-IDF [16], BM25 [15], LSI [79], and LDA [80], which guide the TCP process by measuring textual similarity between test cases and targeted code snippets.

In addition, many studies have been conducted to compare the performance of different TCP approaches. For example, Yoo and Harman [25] conducted a systematic survey on test case optimization, which explains TCP from different perspectives. Lu et al. [11] evaluated the effectiveness of TCP approaches in the scenario of practical software evolution. Henard et al. [81] compared the performance of black-box and white-box TCP approaches, while Luo et al. [82] compared the performance of static and dynamic TCP approaches. Finally, Lou et al. [83] studied TCP approaches in a finer-grained granularity, where TCP approaches were classified according to various features, e.g., algorithms used, criteria, measurements, scenarios, constraints, etc. Differently from them, our study aims at exploring the possibility of improving the IR-based TCP approach by incorporating state-of-the-art source code representation methods, which is orthogonal to existing studies.

In this study, we conducted the first study to explore the possibility of using code representation methods to boost IR-based TCP approaches. In the future, we plan to incorporate more advanced code representation methods into our task.

VII. Conclusion

Due to the state-of-the-art effectiveness of IR-based TCP and the ignorance of the code semantic information of the existing IR-based TCP approach, we conducted the first study to explore whether the effectiveness of IR-based TCP can be further improved by incorporating code semantic information. Specifically, we studied a typical general-purpose code representation model (i.e., Doc2Vec) and a typical task-associated model (i.e., ASTNN) to encode semantic information from both code changes and test cases. Additionally, we investigated the influence of two modes of code representation usage (i.e., unsupervised and supervised modes). Our results show that, regardless of Doc2Vec or ASTNN, code representation cannot effectively boost IR-based TCP in the unsupervised mode, but largely outperform the state-of-the-art IR-based TCP approach in the supervised mode (the average improvement is 16.96% in terms of APFD).

VIII. Acknowledge

This work was supported by the National Natural Science Foundation of China Grant Nos. 62322208, 62002256, 62232001, 62202324.

REFERENCES

[1] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 29:1–29:32, 2017.

[2] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 10:1–10:31, 2014.

[3] D. You, Z. Chen, B. Xu, B. Luo, and C. Zhang, "An empirical study on the effectiveness of time-aware test case prioritization techniques," in *SAC*. ACM, 2011, pp. 1451–1456.

[4] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *ISSTA*. ACM, 2020, pp. 324–336.

[5] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *ISSRE*. IEEE Computer Society, 2015, pp. 46–57.

[6] Q. Luo, K. Moran, D. Poshyvanyk, and M. D. Penta, "Assessing test case prioritization on real faults and mutants," in *ICSME*. IEEE Computer Society, 2018, pp. 240–251.

[7] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 33, no. 9, pp. 1041–1054, 2021.

[8] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1258–1275, 2012.

[9] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *ICSE (1)*. IEEE Computer Society, 2015, pp. 268–279.

[10] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *ICST*, 2016, pp. 266–277.

[11] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *ICSE*. ACM, 2016, pp. 535–546.

[12] M. Wen, R. Wu, and S. Cheung, "Locus: locating bugs from software changes," in *ASE*. ACM, 2016, pp. 262–273.

[13] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *PROMISE*. ACM, 2011, p. 2.

[14] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2021, pp. 2169–2182.

[15] S. E. Robertson, S. Walker, and M. Hancock-Beaulieu, "Experimentation as a way of life: Okapi at TREC," *Inf. Process. Manag.*, vol. 36, no. 1, pp. 95–108, 2000.

[16] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manag.*, vol. 24, no. 5, pp. 513–523, 1988.

[17] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *ACL*. Association for Computational Linguistics, 2020, pp. 4998–5007.

[18] L. Chen, W. Ye, and S. Zhang, "Capturing source code semantics via tree-based convolution over api-enhanced AST," in *CF*. ACM, 2019, pp. 174–182.

[19] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in *ICSE*. IEEE, 2021, pp. 1161–1173.

[20] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *MSR*. ACM, 2020, pp. 243–253.

[21] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML*, ser. JMLR Workshop and Conference Proceedings, vol. 32. JMLR.org, 2014, pp. 1188–1196.

[22] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *ICSE*. IEEE / ACM, 2019, pp. 783–794.

[23] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR (Workshop Poster)*, 2013.

[24] "Homepage," Accessed: January 2023. [Online]. Available: https://github.com/youhanmo/sstcp/

[25] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verification Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *ICSM*. IEEE Computer Society, 1999, pp. 179–188.

[27] 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*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[28] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*. OpenReview.net, 2021.

[29] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "FIRA: fine-grained graph-based code change representation for automated commit message generation," in *ICSE*. ACM, 2022, pp. 970–981.

[30] Y. Shi, Y. Mao, T. Barnes, M. Chi, and T. W. Price, "More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code," in *EDM*. International Educational Data Mining Society, 2021.

[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *ICSE*. IEEE Computer Society, 2007, pp. 96–105.

[32] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai, "TECCD: A tree embedding approach for code clone detection," in *ICSME*. IEEE, 2019, pp. 145–156.

[33] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*. ACM, 2018, pp. 298–309.

[34] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR*, 2015.

[35] M. Lei, H. Li, J. Li, N. Aundhkar, and D. Kim, "Deep learning application on code clone detection: A review of current knowledge," *J. Syst. Softw.*, vol. 184, p. 111141, 2022.

[36] S. Ding, J. Shang, S. Wang, Y. Sun, H. Tian, H. Wu, and H. Wang, "Ernie-doc: A retrospective long-document modeling transformer," in *ACL/IJCNLP (1)*. Association for Computational Linguistics, 2021, pp. 2914–2927.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017, pp. 5998–6008.

[38] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 81:1–81:37, 2018.

[39] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, "On the naming of methods: A survey of professional developers," in *ICSE*. IEEE, 2021, pp. 587–599.

[40] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *ICSME*. IEEE Computer Society, 2014, pp. 476–480.

[41] "Ambient software evoluton group," Accessed: January 2023. [Online]. Available: http://secold.org/projects/seclone

[42] "Maven," Accessed: January 2023. [Online]. Available: https://maven.apache.org/

[43] "Empirically revisiting and enhancing ir-based test-case prioritization." Accessed: January 2023. [Online]. Available: https://sites.google.com/view/ir-based-tcp

[44] "Apache commons cli," Accessed: January 2023. [Online]. Available: https://github.com/apache/commons-cli

[45] "Apache commons compress," Accessed: January 2023. [Online]. Available: https://github.com/apache/commons-compress

[46] "Apache commons codec," Accessed: January 2023. [Online]. Available: https://github.com/apache/commons-codec

[47] "Apache commons csv," Accessed: January 2023. [Online]. Available: https://github.com/apache/commons-csv

[48] "Apache commons math," Accessed: January 2023. [Online]. Available: https://github.com/apache/commons-math

[49] "Jackson," Accessed: January 2023. [Online]. Available: https://github.com/FasterXML/jackson-core

[50] "Jacksonxml," Accessed: January 2023. [Online]. Available: https://github.com/FasterXML/jackson-dataformat-xml

[51] "Jfreechart," Accessed: January 2023. [Online]. Available: https://github.com/jfree/jfreechart

[52] "Gensim," Accessed: January 2023. [Online]. Available: https://github.com/RaRe-Technologies/gensim

[53] "Astnn," Accessed: January 2023. [Online]. Available: https://github.com/zhangj111/astnn

[54] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *ICST*. IEEE Computer Society, 2016, pp. 266–277.

[55] S. Wang, J. Nam, and L. Tan, "QTEP: quality-aware test case prioritization," in *ESEC/SIGSOFT FSE*. ACM, 2017, pp. 523–534.

[56] C. Hettiarachchi, H. Do, and B. Choi, "Risk-based test case prioritization using a fuzzy expert system," *Inf. Softw. Technol.*, vol. 69, pp. 1–15, 2016.

[57] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang, "Exploring better black-box test case prioritization via log analysis," *ACM Trans. Softw. Eng. Methodol.*, p. to appear, 2022.

[58] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[59] R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin, "LIBLINEAR: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, 2008.

[60] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[62] P. F. Felzenszwalb, R. B. Girshick, D. A. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 9, pp. 1627–1645, 2010.

[63] K.-K. Sung and T. Poggio, "Example-based learning for view-based human face detection," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 20, no. 1, pp. 39–51, 1998.

[64] F. Wilcoxon, S. Katti, and R. A. Wilcox, "Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test," *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.

[65] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *ICSE*. IEEE, 2021, pp. 397–409.

[66] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 30:1–30:35, 2020.

[67] L. V. Hedges and I. Olkin, *Statistical methods for meta-analysis*. Academic press, 2014.

[68] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," *CoRR*, vol. abs/2302.04026, 2023.

[69] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *ICER (1)*. ACM, 2022, pp. 27–43.

[70] E. Choi, N. Fuke, Y. Fujiwara, N. Yoshida, and K. Inoue, "Investigating the generalizability of deep learning-based clone detectors."

[71] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "Bridging pre-trained models and downstream tasks for source code understanding," in *ICSE*. ACM, 2022, pp. 287–298.

[72] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *CoRR*, vol. abs/2302.05020, 2023.

[73] V. Pallagani, B. Muppasani, K. Murugesan, F. Rossi, B. Srivastava, L. Horesh, F. Fabiano, and A. Loreggia, "Understanding the capabilities of large language models for automated planning," *arXiv preprint arXiv:2305.16151*, 2023.

[74] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE*. IEEE Computer Society, 2001, pp. 329–338.

[75] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 195–209, 2003.

[76] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 159–182, 2002.

[77] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *ISSRE*. IEEE Computer Society, 2004, pp. 113–124.

[78] C. Lin, C. Chen, C. Tsai, and G. M. Kapfhammer, "History-based test case prioritization with software version awareness," in *ICECCS*. IEEE Computer Society, 2013, pp. 171–172.

[79] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *J. Am. Soc. Inf. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.

[80] X. Wei and W. B. Croft, "Lda-based document models for ad-hoc retrieval," in *SIGIR*. ACM, 2006, pp. 178–185.

[81] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *ICSE*. ACM, 2016, pp. 523–534.

[82] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *SIGSOFT FSE*. ACM, 2016, pp. 559–570.

[83] Y. Lou, J. Chen, L. Zhang, and D. Hao, "Chapter one - A survey on regression test-case prioritization," *Adv. Comput.*, vol. 113, pp. 1–46, 2019.