SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

An Investigation on the Usage of Source Code Embeddings in Test Case Prioritization and Selection

Alessandro Escher

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

An Investigation on the Usage of Source Code Embeddings in Test Case Prioritization and Selection

Eine Untersuchung zur Verwendung von Quellcode-Embeddings in der Testfall-Priorisierung und -Auswahl

Author:	Alessandro Escher
Examiner:	Prof. Dr. Alexander Pretschner
Supervisor:	Raphael Nömmer
Submission Date:	April 22, 2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

A Escher

Munich, April 22, 2025 Alessandro Escher

Acknowledgments

I would first like to thank my advisor, Raphael Noemmer, for his consistent support and guidance throughout the development of this thesis. His feedback and willingness to invest time in discussing ideas and reviewing my work were a great help, and I sincerely appreciate his involvement in this process. I also thank my friends and family for proofreading and giving valuable feedback to improve this thesis. I am especially grateful to the CQSE GmbH and Prof. Alexander Pretschner along with Jakob Rott for allowing this collaboration to happen. CQSE GmbH also generously provided access to their project and its source code, which was an important component for the practical part of this thesis. They also offered a virtual machine for running my experiments which allowed me to process and evaluate a larger number of projects and commits. Additionally, they granted access to a paid third party API service which added to the diversity of embedding models I was able to evaluate. Their support made a significant contribution to the success of this thesis, and I am very grateful for their involvement.

Abstract

Regression testing is a critical process in modern software development, ensuring that new changes do not introduce unintended bugs. However, as software grows, executing an entire test suite after every modification becomes increasingly expensive. Test Case Prioritization (TCP) addresses this challenge by reordering test cases so that fault-revealing tests execute earlier, reducing feedback time and debugging effort.

In this thesis, we investigate the use of source code embeddings generated by Large Language Models for TCP as an alternative to traditional coverage-based and information retrieval (IR) techniques. In particular, we propose a Code Embedding-based Test Case Prioritization (CETCP) approach and evaluate its performance against established baselines, including a BM25-based IR method and a TimeSort baseline.

The evaluation consists of experiments on Defects4J projects, as well as a large-scale industry project called Teamscale, employing both the Average Percentage of Fault Detection (APFD) metric and its cost-aware variant (APFDc) to assess failure detection performance. We conclude that, when incorporating test execution time into the TCP calculation, our approach using the CodeXEmbed model significantly outperforms both a TimeSort baseline and a BM25-based IR method in terms of APFDc. However, results based on APFD alone indicate only marginal improvements on open source projects and inferior performance on the industry project, with no statistically significant differences observed.

These findings suggest that, while embedding-based techniques offer promising advantages for cost-aware test prioritization, their effectiveness may depend on the evaluation context and the characteristics of the underlying project. Further improvements in the information collection and embedding generation are required to make our approach more competitive and robust.

Contents

Abbreviations

RQ Research Question

VCS Version Control System

CI Continuous Integration

LoC Lines of Code

LLM Large Language Model

RNN Recurrent Neural Network

MLM Masked Language Modelling

CLM Causal Language Modeling

AST Abstract Syntax Tree

TSO Test Suite Optimization

TCS Test Case Selection

TCP Test Case Prioritization

TSM Test Suite Minimization

CETCP Code Embeddings-Based Test Case Prioritization

APFD Average Percentage of Failure Detection

APFDc	Average Percentage of Failure Detection per Cost
IR	Information Retrieval
LDA	Latent Dirichlet Allocation
TF-IDF	Term Frequency-Inverse Document Frequency
BM25	Best Matching 25
UMAP	Uniform Manifold Approximation and Projection
t-SNE	t-Distributed Stochastic Neighbor Embedding

1. Introduction

Software systems evolve constantly, with frequent changes introducing new features and bug fixes. To ensure that these changes introduce as few regressions as possible, extensive regression testing is required. However, having to wait for the full test suite to complete after every commit to get feedback or not being able to run the whole test suite due to its size raises issues. Especially in large-scale projects with many thousands of test cases, test suites can take hours or even days to complete and often require a substantial amount of resources [1–4]. This challenge has led to the development of various Test Suite Optimization (TSO) techniques in order to better handle large test suites. One of these techniques is Test Case Prioritization (TCP), which aims to reorder test cases such that any failures are detected as early as possible. By prioritizing fault-revealing test cases, TCP improves feedback time for developers.

Traditional TCP techniques often rely on code coverage-based heuristics, prioritizing tests that execute the modified code or those that produce a lot of coverage [5]. While effective in some cases, these approaches have limitations. They may not always correlate with fault detection capability and require detailed and up to date coverage information, which is not always available and often requires a high maintenance effort [6–8]. The whole system has to be instrumented and profiled after every major change and the coverage data has to be stored for each version of the system. Additionally, TCP approaches using the coverage information rely on old information gathered from the previous version of the code base, missing newly added code and tests [6]. To address these limitations, recent research has explored Information Retrieval (IR) methods and machine learning-based approaches that leverage textual or structural similarities between test cases and the modified code. In particular, code embeddings—vectorized representations of source code—may offer a promising alternative for TCP by capturing semantic relationships between test cases and code changes without requiring explicit coverage data or expensive training procedures.

In this thesis we investigate whether embeddings generated by pre-trained code models can serve as an effective alternative to traditional TCP techniques. We develop a prototype implementation of a Code Embeddings-Based Test Case Prioritization (CETCP) approach that ranks test cases based on their similarity to code modifications using vectorized representations called embeddings. To evaluate the effectiveness of this approach, we conduct an extensive empirical study on both Defects4J [9]—a well-known benchmark dataset of real-world software defects—and Teamscale, a large-scale industry project. The evaluation compares CETCP against various baselines, including a random ordering, a BM25-based [10] IR approach, and a time-based sorting strategy. Performance is measured using Average Percentage of Failure Detection (APFD) and APFDc (cost-aware APFD) to assess fault detection efficiency and the impact of execution time.

The rest of this thesis is structured as follows. Chapter 2 gives insight into concepts and terminology required for this topic, Chapter 3 gives an overview of alternative and established TCP approaches. Chapter 4 explains how the prototype developed in this thesis works while Chapter 5 contains the empirical evaluation in terms of fault detection capabilities. Lastly, Chapter 6 and Chapter 7 contain our suggestions for possible future work based on this work and our concluding thoughts, respectively.

2. Terms and Background

This section explains terminology required to understand the topics that are discussed in this thesis and also gives an overview over the concepts that are applied.

2.1. Version Control Systems and Continuos Integration Pipelines

Most projects use a Version Control System (VCS) such as *Git*[1]. VCSs are a tool that help manage changes to source code, allowing multiple developers to collaborate in parallel while maintaining a complete history of the modifications made to the system. It enables tracking of code changes, reverting the code base to previous states, and resolving conflicts when multiple contributors work on the same code.

Most VCSs utilize Continuous Integration (CI) pipelines, which automate the process of building and validating code changes. When developers push updates to the remote repository, CI pipelines automatically run a predefined test suite to detect any issues with the changes. This ensures that faulty code is identified before being merged into the main development branch.

2.2. Test Suite Optimization Techniques

Test Suite Optimization is a very generic umbrella term that encompasses many different research areas and techniques. In this thesis we focus on a subset of techniques known as regression testing. This section explains the concept of regression testing and gives an overview of different approaches and explains some of the terminology.

2.2.1. Faults and Failures

When discussing test suite optimization techniques it is important to distinguish between the terms faults, failures and errors. A fault describes a mistake in the source code which causes the system to behave in an unintended manner. This state of unexpected behaviour is called an error. Should this error state lead to wrong

[1]`https://git-scm.com/about/`

output—whether that be incorrect output data or a crash of the system—then a failure has occurred. The goal of test suite optimization techniques like TCP is to prioritize those test cases that produce a failure in the system so that the underlying fault can be identified and fixed before being integrated into the system. One fault can produce multiple failures, in which case fixing the underlying fault may resolve multiple failing test cases at once.

2.2.2. Regression Testing Approaches

In regression testing, a test suite is run whenever modifications are made to a system, e.g. by a commit to the VCS, to ensure no breaking changes are introduced. If the test suite passes without failures, then the changes can be fully integrated into the code base. Should one or more tests fail, the changes are rejected and a developer needs to investigate the failing test cases and fix the underlying issues. TCP, Test Case Selection (TCS) and Test Suite Minimization (TSM) are techniques that aim to optimize a test suite by temporarily or permanently changing it. The main goal of all these approaches is to reduce the time required for the modified test suite to provide feedback on whether the newly introduced changes contain a fault. This is done while preserving as much of the full test suite's fault detection capability as possible. For smaller systems where execution time of the whole test suite is not a concern, running the whole test suite is usually a preferable approach. In practice however, where industry systems have large test suites that can take many hours or even days to run [1], much of a project's cost is spent on testing [2, 3, 5]. The goal of approaches like TCP, TCS and TSM is to minimize the time it takes for the developer to get this valuable feedback. TCS and TSM aim to achieve this by reducing the number of tests that are run. TCS does so by selecting a subset of test cases depending on the current changes and letting only those tests be executed, whereas TSM permanently removes redundant or unnecessary test cases from the test suite, independent from any specific set of changes. TCP does not reduce the number of tests that are run, ensuring that the whole test suite is always executed, but instead reorders them such that the most relevant test cases are executed first. This way, the developer will see any failing test cases at the beginning of the test suite and can already start working on a fix while the rest of the suite completes. Choosing which tests are important for a given change is a difficult task and relevant test cases might be missed in an approach like TCS, resulting in a lower fault detection capability [5]. This could make TCP a favorable approach to TCS when it comes to industry adoption, since the former does not incur the risk of missing any failing tests.

2.3. Information Retrieval

IR deals with search problems on unstructured data such as natural language texts and finds many applications in everyday activities such as web searching [10]. Given a large set of documents and a query, the goal is to find the documents that best answer or relate to this query [10]. This process starts with the collection and preprocessing of the documents which includes tokenization, the removal of very common words called stop words and the normalization of tokens to reduce variations caused by linguistic differences, such as verb conjugations and plural forms [10]. Stop words are frequently occurring words—such as "the," "is," "and" and "of"—that typically do not contribute meaningful information for distinguishing relevant documents. Since these words appear so frequently across documents, they add little value in differentiating between them during retrieval [10].

Once the preprocessing stage is finished, the documents in the collection are indexed. In this phase, data such as the frequency and position of a term is collected and stored for each document. This makes the querying phase much more efficient than a simple linear scan of the text, at the cost of a higher storage overhead [10]. The last stage of IR is the retrieval and ranking phase. First, relevant documents are retrieved from the overall document corpus based on the query. These selected documents are then sorted in descending order of relevance to the query [11].

Traditional retrieval models, such as Term Frequency-Inverse Document Frequency (TF-IDF), score words based on their importance in a document relative to the entire collection. TF-IDF assigns higher weights to terms that appear frequently in a document but penalizes those that are common across many documents [10]. It does so by computing the product of two values: term frequency (TF), which represents how often a term appears in a specific document, and inverse document frequency (IDF), which reduces the weight of terms that appear in a large number of documents. Equation 2.1 shows the different components of TF-IDF, where t represents a term, d a document of a set of documents D and $f_{t,d}$ the number of times the term t appears in document d.

$$\begin{aligned}&\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)\\ &\quad\text{TF}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \quad \text{IDF}(t) = \log\left(\frac{|D|}{\text{DF}(t)}\right) \quad \text{DF}(t) = |\{d \in D \mid t \in d\}|\end{aligned} \tag{2.1}$$

While effective for basic retrieval, it does not capture deeper semantic relationships between words and is susceptible to duplicated content in documents [10].

A more advanced and widely applied ranking function is Best Matching 25 (BM25) [10, 11]. Unlike TF-IDF, which assumes term importance increases linearly with frequency, BM25 introduces a saturation function, meaning that repeated occurrences of a term in a document contribute less additional relevance beyond a certain point.

Additionally, BM25 normalizes document length, addressing the issue where longer documents tend to contain more query terms simply due to their size, which can skew rankings in TF-IDF. BM25 achieves this by using tuning parameters k_1 and b, where k_1 controls term frequency saturation and b determines the degree of length normalization [10, 11]. Equation 2.2 shows a widely used version of the BM25 formula [10], where d is a document, q the query and *avgdl* the average document length of the corpus.

$$\mathrm{BM25}(d,q) = \sum_{t \in q} \mathrm{IDF}(t) \cdot \frac{f_{t,d} \cdot (k_1 + 1)}{f_{t,d} + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\mathrm{avgdl}}\right)} \tag{2.2}$$

However, both TF-IDF and BM25 primarily rely on exact word matching. To enable more conceptual retrieval, Latent Dirichlet Allocation (LDA) applies topic modeling, assuming that documents are mixtures of topics which in turn are distributions over words. By identifying hidden topic structures within a document collection, LDA allows retrieval based on conceptual similarity instead of exact keyword matching [10, 11].

2.4. Code Model Embeddings

Recent advancements in machine learning techniques have enabled significant improvements in the embeddings based representation of source code [12]. Code embeddings are vector representations of code snippets that capture syntactic and semantic information, making them useful for various software engineering tasks [13, 14]. Large Language Models (LLMs) are deep learning models trained on massive amounts of text and/or code to understand and generate natural language or structured text like source code. These models are usually based on the *Transformer* [15] architecture. A *Transformer* consists of multiple layers of attention heads and feed-forward networks, enabling it to learn complex relationships between tokens in an input sequence [12, 15]. In contrast to previous methods like Recurrent Neural Networks (RNNs), Transformers process input in parallel, which makes them highly scalable and a much more powerful alternative [16].

Researches often publish pre-trained versions of their model, which are first trained on large generic datasets and can then be fine-tuned for specific downstream tasks, such as code search, summarization, or clone detection. Popular pretraining objectives for LLMs include Masked Language Modelling (MLM), where parts of the input are hidden and then predicted, and Causal Language Modeling (CLM), where the model predicts the next token in a sequence. Code embeddings can be extracted from these models by passing a code snippet through the Transformer and retrieving the corresponding hidden state representations from specific layers, often using the output from a special classification token or an average pooling over all output token embeddings [12, 17]. Several specialized LLMs have been developed for processing source code. The following is an overview over the models that are used in this thesis:

CodeBERT A transformer-based model pre-trained on source code and natural language for tasks like code search and summarization. It is trained with MLM and Replaced Token Detection (RTD) to understand both syntax and semantics in source code. [18]

UniXcoder An improvement on traditional encoder-decoder architectures that adds bidirectional attention mechanisms, enabling more effective comprehension of code structure. It also integrates structural information through Abstract Syntax Tree (AST) based representations in the training phase, improving performance in code understanding tasks such as clone detection and code search. [19]

CodeT5+ A generative Transformer model based on *T5* (Text-to-Text Transfer Transformer) that supports both encoding and decoding tasks. It uses an encoder-decoder architecture with a variety of different components that can be combined depending on the downstream task [20]. In this thesis we use the *CodeT5+ Embedding*[2] variant.

CodeXEmbed A family of open-source embedding models designed for code and text retrieval tasks [21]. The 2B variant used in this thesis is called *SFR-Embedding-Code-2B_R*, contains approximately 2.61 billion parameters and is initialized from the *Gemma* [22] model. It was specifically designed for code embedding and has been found to perform better than other open source models when it comes to various retrieval scenarios, including text-to-code, code-to-text, and code-to-code retrieval [21]. We opted for the *2B* variant as a compromise between accuracy and computational efficiency, as the overall performance improvement between the smallest *400M* and the *2B* variant is much more significant than the difference between the *2B* and *7B* versions [21].

OpenAI's text-embedding-3 A more recent closed-source embedding model offered by OpenAI[3], optimized for both text and code representations. It provides embeddings optimized for retrieval-based applications (based on work by Kusupati et al. [23]). In this thesis we use the *text-embedding-3-small* variant[4].

[2]`https://huggingface.co/Salesforce/codet5p-110m-embedding`
[3]`https://openai.com/index/new-embedding-models-and-api-updates/`
[4]`https://platform.openai.com/docs/guides/embeddings/#embedding-models`

Model	**Architecture**	**# Parameters**	**Embedding Size**
CodeBERT	Encoder-only Transformer	125M	768
UniXcoder	Unified Encoder-Decoder Transformer	125M	768
CodeT5+ Embedding	Encoder-only Transformer	110M	256
SFR-Embedding-Code-2B_R	Encoder-only Transformer	2.6B	2304
OpenAI Embedding Small	Transformer	Not specified	1536

Table 2.1.: Overview of code models used in this thesis

Table 2.1 contains an overview of the models and their parameters. By leveraging code model embeddings, we hope to improve upon traditional IR techniques by capturing more complex semantic relationships and being less reliant on exact keyword matches and lexical similarity.

3. Related Work

Optimizing a test suite entails maximizing its effectiveness. This is commonly measured by the achieved code coverage and fault detection rate for a given cost in execution time [3]. With this goal, a wide variety of techniques have been suggested and evaluated over time. This section gives an overview of popular TCP approaches that have been extensively researched as well as newer techniques that have been gaining popularity in recent years.

3.1. Coverage Based Test Case Prioritization

Coverage based TCP techniques aim to maximize the coverage, whether that be on the statement, branch or method level, as early as possible [24]. The idea of a high coverage being desirable is intuitive, as a fault can only be detected if a test is executed that triggers it. This is especially true for changed code, where the likelihood of a fault is much higher, especially if it remains untested [25]. The type of coverage that is used for TCP can vary in granularity, from statement level to function level. Yoo et al. [24] and DiNardo et al. [26] found that with a coarser granularity, such as function coverage, the fault detection capability of the TCP approach worsens. However, using more fine-grained coverage comes at the cost of lower scalability [26].

Another variable that has been examined is the prioritization technique, i.e. the logic by which the tests are ordered. Researchers differentiate between total, additional and modified coverage. Prioritization by total coverage means that for each test, all the statements or functions it covers are counted. In contrast, additional coverage prioritization considers only the new coverage that a test contributes—meaning it accounts only for statements or functions that have not been covered by any previously prioritized tests [24, 26]. Lastly, modified coverage prioritization only considers the coverage of changed code. Di Nardo et al. [26] examined the impact of using different granularities and prioritization approaches on an industrial software projects. They confirmed that more fine grained coverage information performs better and also found that additional coverage prioritization techniques significantly outperform the total coverage approach. Using modification information did not result in significant improvements and was therefore not deemed worth the additional effort [26].

Coverage TCP techniques are among the most researched approaches and have been popular for some time [5, 24, 27]. However with the high effort and maintenance cost required to gather the necessary coverage data, researchers have been steadily looking to find cheaper and easier to manage alternatives [5].

3.2. Information Retrieval Based Test Case Prioritization

IR is a technique that has seen extensive research and application on natural language problems such as querying information from a large collection of documents [10]. Saha et al. were the first to apply this concept to TCP in the regression testing stage by introducing *REPiR* [28]. By treating the test classes and methods in the test suite as a collection of documents and the changed code as the query, they were able to apply the same principles that had been studied for natural language documents to source code. They examined different granularities for the document collection, i.e. on the class and on the method level, as well as various change collection approaches for the query construction [28]. *REPiR* outperformed all evaluated coverage-based TCP techniques, regardless of the document or change retrieval technique [28]. Saha et al. tried to further improve upon these results by incorporating structural information but found that this did not lead to any improvement in fault detection.

Peng et al. developed an IR approach that prioritizes test classes [29]. They experimented with different additional sources of information such as test execution time and historical failure frequencies and examined the impact of different IR retrieval models, among them TF-IDF, BM25 and LDA [29], which we explained in Section 2.3. The evaluation was performed on almost 3000 CI jobs from 123 Java projects. The results show that when using optimal parameters, their IR-only approach was able to outperform coverage-based techniques, although not by a statistically significant margin and fell short of outperforming an execution time-only based prioritization when using a cost-cognizant metric such as the Average Percentage of Failure Detection per Cost (APFDc) [29]. However, when adding test execution time and historical failure information alongside IR, the new approach was able to outperform their other techniques and baselines [29].

3.3. Other Test Case Prioritization Techniques

Classic TCP approaches use coverage, execution times and historical failure information to prioritize test suites [5, 30]. However, a wide variety of techniques from different computer science domains have been used and applied to the problem of TCP.

Miranda et al. equate test suite optimization to a "big data problem" [4] and applied approaches from this domain such as Shingling, Minhashing and Locality-Sensitive Hashing. Their core idea relies on the notion that TCP techniques should strive for diversity, i.e. dissimilarity between test cases, and on the fact that highly efficient algorithms exist in the big data domain to find dissimilar elements in a large set. They evaluated their *FAST* approach on C projects and some Java projects from the Defects4J [9] dataset and found that this approach is more efficient and scalable with no loss in effectiveness when compared to other popular baselines such as coverage and other similarity based TCP techniques [4].

Genetic algorithms have also been applied to TCP. The general idea is to have an initial starting population, where each element is an ordered test suite. Each member of the population is assessed by a fitness function, often using information such as code or requirements coverage [31]. The most fit members are kept onto the next generation and new members are created from them by applying crossover and mutation operations in hopes of creating even better candidates, mimicking the natural process of evolution. Genetic algorithms have the potential to perform better than greedy coverage approaches [32], however they can suffer from a bad initial population, leading to long convergence times and getting stuck in local minima unable to find the optimal test case ordering [33].

Mattis et al. attempted to use code embedding models as well as generative LLMs for TCP [34]. They used the UniXcoder [19] model to generate embeddings of tests and changes and use these in a similar manner to the IR concepts outlined in Section 3.2. They also attempted to use generative LLMs to gather how likely the LLM would have been to generate any of the test suite's tests as an adequate test case for a given change. This likelihood is then used as the test's priority. They evaluated both of these approaches on a small dataset consisting of three small python projects and found that their embedding approach was able to slightly outperform a BM25 baseline, whereas the LLM approach did not perform well. We aim to expand upon this work by evaluating a larger number of code models, including newer and larger models than UniXcoder, and also by using a bigger and more representative dataset such as Defects4J [9] and an industry project.

4. Code Embeddings Based Test Case Prioritization

This chapter illustrates the prototype that was implemented in this thesis. It is a regression TCP approach, meaning the prioritization is always run in the context of one or more VCS commits with code changes. The goal is to find test cases that are most similar to a code change, and have those tests be run first. The prototype is implemented inside the software Teamscale[1], which is a software quality tool developed by CQSE GmbH[2]. In the context of our research it is used for syntax and static analysis, such as identifying code changes and test case implementations as well as storing and analyzing test execution data.

Figure 4.1 shows an overview of the pipeline, consisting of collecting the source code of both tests and changed methods, generating embedding vectors and using those to rank the tests cases for TCP. Details about each step can be found in the following sections.

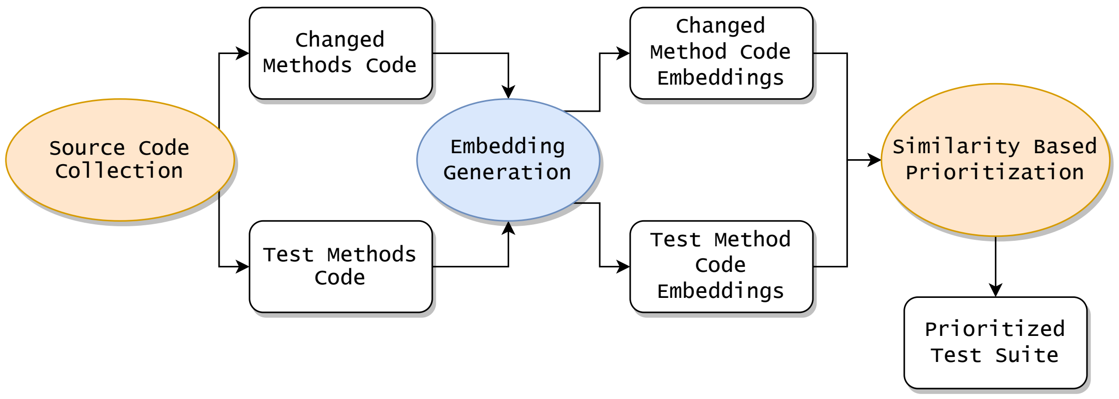

Figure 4.1.: An overview showing the pipeline of the prioritization approach. Orange elements are performed in Java code whereas blue elements are executed in Python.

[1]https://teamscale.com/
[2]https://teamscale.com/about-us

4.1. Source Code Collection

There are two components for which code embeddings have to be generated, the code changes of the target commit and the source code of the tests. The change information is gathered from the underlying VCS, in this case Git[3]. For each line of code that has been changed, the surrounding method is determined by parsing the syntax tree of the code file. The whole code of the method including its signature is saved in a dictionary mapping from the method path to its source code. This way, there is always a syntactically complete context for each change, which is important for many code models due to their pre-training objective[4] [35]. If there are two or more disjoint changed regions within the same method, they are merged and the method source code is only saved once. Changes to comments are ignored, as they do not represent a possible source for faults. Changes such as module imports are not considered, as any breaking change would most likely already be discovered in the system's build phase and not by a test case. Any other changes outside of a method, e.g. to class attributes, are not considered due to scoping issues. Most of the time such changes also come with changes inside method bodies, such as the renaming of a variable or the changing of its type, meaning the change would still be partially included.

The second group of source code that has to be collected are the test cases. This is dependent on the language of the system and the testing framework that is used. In this thesis the focus is set on projects written in Java, for which the most popular testing framework is *JUnit*[5]. *JUnit* 4 and 5 tests are usually annotated with the *@Test* annotation, making it easy to detect them and to collect the test implementations. *JUnit 3* tests are detected by parsing through methods that are in the project's test folder and looking for methods that start with the keyword "*test*".

4.2. Embedding Generation

All vector representations are generated using *Python* by applying a pre-trained code model like the ones described in Section 2.4. The size of this embedding can vary in length depending on the model, from 768 floating point values for *CodeBERT* [18] and *UniXcoder* [19] to 2304 for the *CodeXEmbed* [21] model. An embedding vector is generated for every single code block, which in this case are changed methods and test methods as described in Section 4.1. For open source models, the model data and

[3]`https://git-scm.com/`
[4]see e.g. MLM or CLM in Section 2.4
[5]`https://junit.org/`

the tokenizer are retrieved from the *Huggingface*[6] library via its python module[7]. The model is loaded locally in its pre-trained state and is then used to generate embeddings according to the available documentation.
For *CodeBERT* [18], the embeddings are extracted from the *[CLS]* token[8].
UniXcoder [19] offers a custom python class that contains functionality for tokenization and embedding retrieval[9] by averaging over the last hidden layer output of each token.
For *CodeT5+* [20] a version of the model is used called *codet5p-110m-embedding*[10] which was specifically made for the purpose of embeddings generation.
SFR-Embedding-Code-2B_R offers a custom class[11] intended to be used to reproduce the results of Liu et al [21] on the *Coir* benchmark [36]. The queries —which in this case are the changed methods—are prepended with an instruction formulated in natural language. The documents in the corpus —in this case the test implementations—are structured such that the test's path is prepended to the method source code, which is analogous to the title and text body in the IR context. Figure A.1 shows the relevant code snippets.
For the closed source model from *OpenAI*, the corresponding python module[12] is used according to the official documentation[13]. The API can handle batches of up to 2048 inputs, which in our case are code methods, with a maximum length of 8096 tokens for each input. The price is calculated based on the amount of tokens that are embedded and currently stands at 0.02$ per $1,000,000$ tokens for the *text-embedding-3-small* variant[14].

The generated embedding vectors are then stored in a JSON file as a dictionary mapping from the path of the code entity to the embedding vector.

4.3. Test Case Prioritization

Once all code objects have been embedded, the sorting phase of the TCP approach is comparable to that of IR approaches as laid out in Section 3.2, where the changed code

[6]`https://huggingface.co`
[7]`https://pypi.org/project/huggingface-hub/`
[8]`https://github.com/microsoft/CodeBERT/tree/c0de43d3aaf38e89290f1efb771f8de845e7a489?tab=readme-ov-file#codebert`
[9]`https://github.com/microsoft/CodeBERT/tree/c0de43d3aaf38e89290f1efb771f8de845e7a489/UniXcoder#1-code-and-nl-embeddings`
[10]`https://huggingface.co/Salesforce/codet5p-110m-embedding#how-to-use`
[11]`https://huggingface.co/Salesforce/SFR-Embedding-Code-2B_R/discussions/9#67a10598ce3af048c276787b`
[12]`https://pypi.org/project/openai/`
[13]`https://platform.openai.com/docs/guides/embeddings`
[14]`https://platform.openai.com/docs/pricing#embeddings`

sections are compared (or queried) against the test implementations. The goal is to prioritize those test cases that show a strong correlation to the current changes by being similar in the latent embedding space. This similarity calculation is achieved by using the cosine similarity between two embedding vectors, where c is the embedding vector of the code change and t that of the test implementation.

$$s(c,t) = \frac{c \cdot t}{\|c\|\|t\|} \tag{4.1}$$

First, the embedding vectors generated in Section 4.2 are loaded. Then, for each test embedding, its similarity to every collected change is measured, as explained in Section 4.1. Each test is then placed into a change bucket to which it shows the strongest match. After processing all tests, the test cases within each bucket are ranked in descending order based on their similarity score to the corresponding change. Finally, tests are selected from the buckets in a round-robin fashion until all buckets are depleted. This technique will be referred to as CETCP and is further illustrated in Algorithm 1. A second approach also uses the execution time of a test case in the similarity score calculation. This is achieved by dividing the cosine similarity by the test's runtime in seconds, adding 1 to avoid division by 0 for tests that have executions times that might be rounded to 0, such as unit tests.

$$s_{time}(c,t) = \frac{s(c,t)}{time_t + 1} \tag{4.2}$$

Algorithm 1 Code Embeddings-Based Test Case Prioritization (CETCP)

```
Require: Set of test embeddings $T$, Set of change embeddings $C$
    return List of Prioritized test cases
1: for each $t \in T$ do
2:     $best\_match \leftarrow \arg\max_{c \in C} s(c, t)$
3:     $B_{best\_match} \leftarrow B_{best\_match} \cup \{t\}$
4: end for
5: for each bucket $B$ do
6:     Sort $B$ by $s(c, t)$ in descending order
7: end for
8: Initialize $P \leftarrow \emptyset$
9: while $\exists B \neq \emptyset$ do
10:     for each $B \neq \emptyset$ in round-robin order do
11:         $t \leftarrow \arg\max_{t \in B} s(c, t)$
12:         $B \leftarrow B \setminus \{t\}$
13:         $P \leftarrow P \cup \{t\}$
14:     end for
15: end while
16: return $P$
```

5. Empirical Evaluation

This chapter presents the Research Questions (RQs) that we aim to answer and illustrates the methodology that was used to achieve this. We evaluate a CETCP approach and compare it to a set of baselines, including a BM25 based IR approach. The evaluation is performed on a dataset of open source *Java* projects and a closed source industry project.

5.1. Research Questions

The aim of this thesis is to evaluate whether embeddings from code models with a greater focus on the semantics of code are a viable alternative to other TCP techniques like IR and to determine which code models work well for this goal. Therefore we formulate the following research questions.

RQ1 Which of the selected code models delivers the best results for TCP?

RQ2 How does CETCP compare to a traditional IR TCP approach?

RQ3 How does the CETCP approach perform in a cost aware environment?

RQ4 How do the different approaches scale in regards to computation time?

5.2. Objects of Study

The main study object of this thesis is the *Defects4J* dataset published by Just et al. [9]. It consists of a set of curated Java projects that contain various faults in source code. Each fault has its own revision where all code changes that are unrelated to the fault have been removed. Additionally, flaky tests have been manually filtered out such that all occurring test failures are caused by the underlying fault. This is important because such non-deterministic failures cannot be reliably detected by code changes and would therefore skew the results. All faults present in the dataset occurred naturally, i.e. there are no seeded or synthetically added bugs, allowing for a more realistic and diverse set of bugs. The faults were collected by parsing open source repositories for commits that mentioned fixing bugs in the commit message. By inspecting the commit and locating the code changes that contributed to the fix and reverting them, the authors of the dataset were able to create a buggy version for each code fix found this way [9].

The second set of failures is collected from a large industry project called *Teamscale*[1], which is also used in the implementation of the prototype as described in Chapter 4. *Teamscale* consists of a frontend written in *Javascript/Typescript* and a backend written in *Java*. Failing tests were collected from the project's CI pipeline in *GitLab*[2]. The tests were then programmatically filtered to weed out most flaky failures by checking if a test case had multiple failing runs followed by a passing run for the same commit and by seeing if the test failed on other branches at the same time, indicating a flaky failure possibly related to network connectivity issues. Due to the size and complexity of the system, a manual inspection of failures such as the one performed by the *Defects4J* [9] team was not feasible. Relevant information about the projects of the dataset used in this thesis can be found in Table 5.1.

Project	**LoC**	**Buggy Commits**	**Tests**	**$\sim$ Failing Tests**
Chart	329K	26	2200	3.54
Closure	601K	174	18,400	3.13
Jsoup	29K	93	750	1.56
Lang	169K	65	2300	1.92
Math	305K	106	3600	1.66
Mockito	93K	38	1500	3.11
Time	142K	26	4100	2.85
Teamscale	1.6M	48	11,600	4.15

Table 5.1.: Information about the projects used to evaluate the prototype of this thesis. Lines of Code (LoC) and the number of tests are retrieved from the most recent commit in the dataset. The number of failing tests is the average over all evaluated commits.

5.3. Methodology

This section explains how we collect the necessary data for our empirical evaluation and which metrics and baselines we use to evaluate our prototype's performance.

[1]`https://teamscale.com/`
[2]`https://about.gitlab.com/`

5.3.1. Baselines

To put our results into context, we perform TCP with three baselines.

Random This baseline simply sorts the test cases in a random order, using the *Java Collections.shuffle* method. Since it does not apply any prioritization strategy, it provides a lower bound for performance evaluation. Comparing CETCP against this baseline helps assess whether the learned embeddings provide meaningful prioritization beyond random chance. To minimize the impact of outliers, we run this baseline 100 times and only report the average performance across all iterations.

IR This technique uses an IR approach based on the BM25 method as described in Section 2.3. The implementation is based on the *Apache Lucene*[3] package. This baseline represents a strong, well-established heuristic for test prioritization and is particularly relevant for answering RQ 2, as it allows us to compare CETCP against a non-learning-based method. This baseline can also be combined with test execution data, where the IR score is divided by the execution time of a test similar to the formula used by the CETCP approach in Equation 4.2.

Time Sort This baseline sorts tests in ascending order by their execution time, prioritizing faster tests. This approach assumes that running faster tests earlier can provide quicker feedback. The performance of CETCP when incorporating execution time information (RQ 3) can be compared against this baseline to determine whether CETCP can produce a meaningful improvement upon it, as time-sort has been found to be a powerful baseline not trivial to outperform when using a cost-aware evaluation metric [8, 29].

The Random and IR baselines are directly used to evaluate RQs 1 and 2, while the Time Sort baseline is particularly relevant for RQ3, as it isolates the impact of test execution time on prioritization performance.

5.3.2. Evaluation Metrics

To quantify the effectiveness of a TCP approach in prioritizing failing test cases earlier, we use the widely adopted APFD metric [5, 24]. APFD evaluates how early on failing tests are executed by considering their position in the prioritized test suite. With n as the total number of test cases in the test suite, m as the total number of faults and T_i as the position of a failing test case revealing fault i, the APFD formula is as follows [24]:

$$APFD = 1 - \frac{\sum_{i=1}^{m} T_i}{nm} + \frac{1}{2n} \tag{5.1}$$

[3] `https://lucene.apache.org/`

To answer **RQ1**, we perform a test suite prioritization for each code embedding model from the ones outlined in Section 2.4. We then compare the APFD values of the resulting prioritized test suites, both on the open source dataset as well as the industry project.

For **RQ2** we additionally perform a test suite prioritization using the BM25 IR approach. The APFD is then compared to the results of the CETCP approach from *RQ1*.

Lastly, for **RQ3** we perform a test suite prioritization using execution data as explained in Section 4.3. This is done for both the code models as well as the IR baseline. Since the APFD metric has no notion of test execution times and implicitly assumes that each test case has the same execution cost, we use the APFDc metric to answer *RQ3*. APFDc is an extension of the APFD metric which also considers the cost of a test, which in our case is the test execution time, and also adds a notion of severity for each fault [5, 24]. Let n be the number of test cases in the test suite, m the number of faults, T_i the position of the test that reveals fault i, t_j the execution cost of test case j and f_i the severity of fault i, then the formula of the APFDc is given as [24]:

$$APFD_c = \frac{\sum_{i=1}^{m} \left(f_i \times \left(\sum_{j=T_i}^{n} t_j - \frac{1}{2} t_{T_i} \right) \right)}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{5.2}$$

For our evaluation we do not quantify the severity of faults, so we set all f_i to 1.

For *RQ3*, we only utilize the open source projects as collecting the necessary execution data from the industry project for the whole test suite proved to be too time consuming.

5.3.3. Collecting Evaluation Data

To evaluate the effectiveness of different TCP techniques, we automate the collection of evaluation data using a Python script. This script systematically processes all commits for a given Defects4J [9] project and performs TCP using all code models outlined in Section 2.4 and the baselines from Subsection 5.3.1. Computations were performed on a *g2-standard-8 Google Cloud VM*[4]. The evaluation for the Defects4J projects follows these steps:

1. Checkout of Buggy Versions: Using the Defects4J executable[5], we employ its *checkout*[6] functionality to retrieve the buggy version of each commit locally. The list of all bug IDs for a project is obtained via the *query*[7] command.

[4]`https://cloud.google.com/compute/docs/gpus#l4-gpus`

[5]`https://github.com/rjust/defects4j/blob/88c6225ee54c4fa0c7c00c50762333fe64d1426f/framework/bin/defects4j`

[6]`http://defects4j.org/html_doc/d4j/d4j-checkout.html`

[7]`https://defects4j.org/html_doc/d4j/d4j-query.html`

2. Test Suite Prioritization Execution: A new Teamscale project is created for each buggy commit revision. The script executes TCP on this revision using the selected prioritization technique. The prioritized test suite and associated metadata are stored in a JSON file.

3. Running Baseline Prioritization: After storing the initial TCP results, the baselines (*Random*, *IR*) are executed on the same revision. Their outputs, including prioritized test orders and metadata, are similarly stored.

4. APFD Calculation: To compute the APFD, we use the Defects4J *query* function to retrieve the *triggering_tests* attribute. This provides the set of failing test cases for each buggy version, allowing us to measure how quickly faults are detected by the different prioritization methods.

5. APFDc and Execution Time Data Collection: The *checkout* and *test*[8] functionalities of Defects4J are used to run the full test suite. To extract execution times, the *defects4j.build.xml*[9] file is modified to generate a JUnit test report containing timing information. The generated report is uploaded to Teamscale, making test execution times accessible for all prioritization methods. This execution time data is stored alongside the prioritized test suite results in the JSON files to be later used in the APFDc calculation.

The evaluation on the industry project follows a similar pattern, with the exception that all commits appear in the same repository and the failing tests are gathered from the CI platform.

5.3.4. Dimensionality Reduction for Visualization

In order to showcase the information that is contained within embedding vectors for our discussion Section 5.5, we can visualize each embedded element in a coordinate system. To achieve this, we generate embeddings for the commit and code model in question, normalize them using the $L2$ norm and use Uniform Manifold Approximation and Projection (UMAP) [37], a novel dimensionality reduction technique, to reduce the embedding vectors to two dimensions. This allows us to plot all embeddings in a coordinate system, as shown in Figure 5.2. We use the implementation of the python module *umap-learn*[10] and *plotly*[11] for the plot, for details see Figure A.2. UMAP

[8] `https://defects4j.org/html_doc/d4j/d4j-test.html`

[9] `https://github.com/rjust/defects4j/blob/88c6225ee54c4fa0c7c00c50762333fe64d1426f/framework/projects/defects4j.build.xml`

[10] `https://umap-learn.readthedocs.io/en/latest/index.html`

[11] `https://plotly.com/python/plotly-express/`

attempts to keep both local and global structure from the higher dimensional data into lower dimensions, making it a more adequate choice for our application than other state of the art visualization techniques such as t-Distributed Stochastic Neighbor Embedding (t-SNE) that loose the global structure [37, 38].

5.4. Results

Project	**Random**	**IR**	**CodeBERT**	**CodeT5+**	**UniXcoder**	**OpenAI**	**CodeXEmbed**
Chart	49.97	97.73	51.11	94.93	91.3	**98.19**	98.15
Closure	49.73	82.22	60.36	76.01	74.5	83.17	**89.83**
Jsoup	50.04	88.66	51.35	88.56	81.65	90.54	**91.7**
Lang	49.79	**98.02**	52.86	95.48	86.43	95.47	97.07
Math	49.97	95.7	56.43	89.89	86.87	95.48	**97**
Mockito	49.89	83.51	61.53	86.04	89.61	88.63	**92.21**
Time	49.85	90.08	53.23	88.33	76.77	90.67	**91.83**
Teamscale	52.61	**85.77**	67.81	79.41	78.23	80.37	76.19
Average	50.23	91.46	56.84	87.33	83.17	90.32	**91.75**

Table 5.2.: APFD results for the open source and industry projects, scaled to an interval of $[0, 100]$. The best performing method is highlighted in bold for each project.

Table 5.2 showcases the APFD results over both the open source projects and the industry project as well as an average APFD over all projects for the baselines and all code models. The best performing TCP approach is highlighted in bold. The APFD values were scaled to an interval of $[0, 100]$ and rounded to two decimal digits for formatting and spacing purposes.

For every Defects4J project, we map all failures to one fault, meaning we assume that all failing test cases are caused by the same bug. This is because all commits in the Defects4J dataset were filtered down to contain only changes related to a single bug and all failing tests are caused only by the fault [9]. For our APFD and APFDc calculations (see Equation 5.1 and Equation 5.2 respectively), this means that $m = 1$ and T_i is the position of the first failing test. We discuss consequences of this approach in Section 5.5 and Section 5.6. For the industry project, we assume each failure is caused by a separate fault, as we do not filter out any code changes or failing tests and have no information about the number of bugs that were introduced in a commit.

Figure 5.1 shows a plot for each TCP technique. The dots represent the APFD values from Table 5.2 and the lines show the minimum and maximum range of APFD values

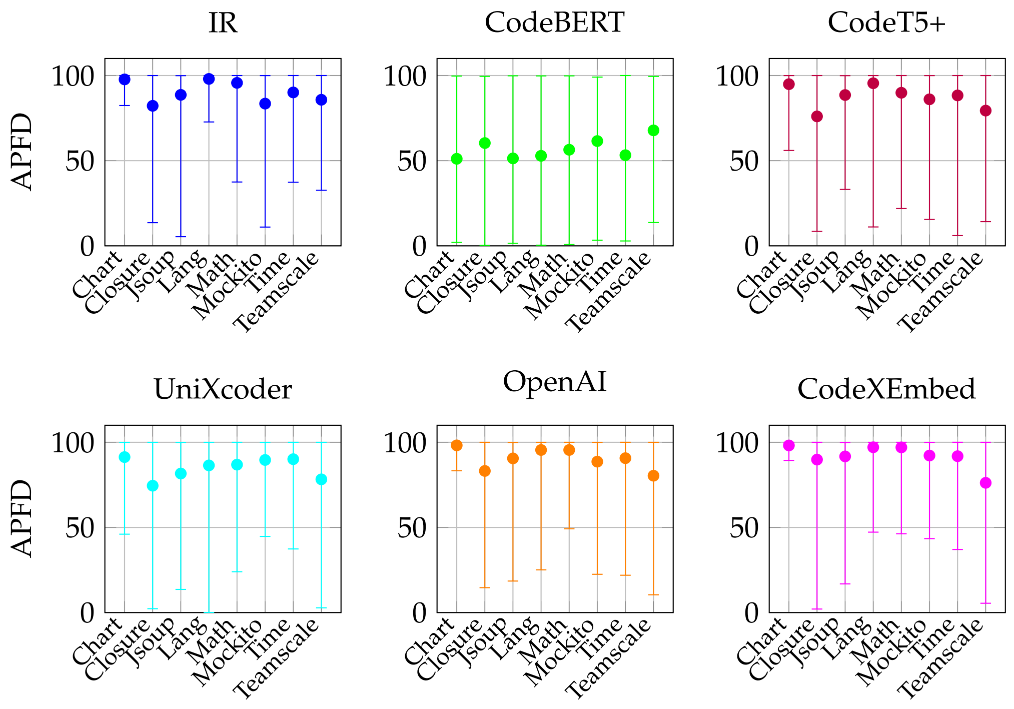

Figure 5.1.: Mean APFD values (dots) with bars showing the min/max range of APFD values achieved over the commits of each project.

that were achieved across the commits of each project. The random baseline was left out of this figure as the variance of APFD values is between 0 and 100 as is to be expected from such an approach.

Table 5.3 contains the APFDc values of the baselines and the two open source code models that performed best in terms of APFD. Once again the APFDc values were scaled to an interval of $[0, 100]$ and rounded to two decimal digits. The best performing approach is highlighted in bold for each project.

In Table 5.4 we write down measured computation time results of all evaluated approaches. The values were measured during the evaluation for RQ1 and include the time it takes to gather the changed and test method source code as well as the model loading time for the CETCP approaches. Computational performance was not considered during the development of the prototype, so these results serve only to get a general idea of how long each technique takes to generate a prioritized test suite. It is also important to note that values depend strongly on the project, as the computation time scales with the number and LoC of test cases in the project's test suite.

Project	Random	Time-Sort	IR	CodeT5+	CodeXEmbed
Chart	50.26	87.96	94.41	95.17	**96.62**
Closure	49.95	75.98	87.12	82.91	**92.92**
Jsoup	49.97	80.49	90.65	90.02	**93.67**
Lang	50.12	96.61	**99.27**	98.17	98.63
Math	49.89	89.07	96.89	92.87	**97.4**
Mockito	49.66	84.96	90.97	88.27	**95.43**
Time	50.18	86.58	91.79	88.36	**93.5**
Average	50.00	85.95	93.01	90.82	**95.45**

Table 5.3.: APFDc results for the open source projects, scaled to an interval of $[0, 100]$. The best performing method is highlighted in bold for each project.

Project	IR	CodeBERT	CodeT5+	UniXcoder	OpenAI	CodeXEmbed
Chart	8.2	76.2	66.7	204.9	138	639.4
Closure	71.9	212.9	216.5	593.8	257	1635.2
Jsoup	2.6	16.9	16.1	41.7	12.2	117.9
Lang	17.6	55.9	51.8	138.2	31.5	623.7
Math	16.2	64.8	56.3	186.9	176.3	884.8
Mockito	28.1	25.8	23	103	15.1	202.8
Time	60.9	317.8	314.4	579.2	400.2	1547.2
Teamscale	23.2	207.6	216.5	768.3	339.5	3085.8

Table 5.4.: Computation times in seconds for relevant TCP techniques across all projects, rounded to one digit.

The random and time sort baselines were left out from Table 5.4 since their computation times are negligibly small and are not the focus of this thesis.

5.5. Discussion

This section gives our interpretation and thoughts on the results showcased in Section 5.4 as well as our answers to our research questions posed in Section 5.1.

5.5.1. RQ1: Best Code Model

Code Model Comparisons

When looking at the APFD results from Table 5.2 it becomes clear that newer models outperform older models. E.g. CodeBERT [18], which is the oldest model examined in this thesis, performs poorly and is not always able to convincingly outperform the random baseline. From Figure 5.1 we can also see that the variance is between $[0, 100]$, similar to that of a random baseline, showing that CodeBERT tends to regularly struggle in extracting meaningful embeddings.

Newer but similarly sized models in terms of parameter count such as CodeT5+ [20] and UniXcoder [19] perform much better and are often closer to the IR baseline and sometimes to the top performing code models. Still, neither of them performs the best for any of the projects. Between these two models, CodeT5+ usually outperforms UniXcoder, often by a large margin, sometimes even being close to the top performers such as e.g. for the Defects4J [9] projects *Jsoup*, *Lang* and *Time*. This is most likely due to the fact that CodeT5+ is a newer variation of the CodeT5 family, having been released a year after UniXcoder, and that the variant we used in this thesis was specifically made to generate code embeddings[12]. Judging by the min/max bars from Figure 5.1, both models show a similar variance in performance, albeit across different projects. Overall between these two models, CodeT5+ [20] could be considered as an alternative to larger or more expensive models, trading accuracy for model size and computation cost.

The two best performing models are OpenAI's *text-embedding-3-small* and *SFR-Embedding-Code-2B_R* from the CodeXembed [21] family. They consistently outperform all other code models, albeit on a smaller margin for Defects4J projects *Mockito* and *Time*. The superior performance of these models when compared to the other code models can be attributed to two facts. The CodeXEmbed model is by far the largest model in this lineup, with $\sim$ 2.6 billion parameters compared to the other much smaller open source models, which have a parameter count in the neighbourhood of $\sim$ 120

[12]`https://huggingface.co/Salesforce/codet5p-110m-embedding`

million. A larger parameter count has been found to generally lead to increased performance [39]. Additionally, CodeXEmbed was designed with retrieval tasks in mind, including code-to-code retrieval. The CodeXEmbed model we used is the *2B* variant, however a larger version with ~ 7 billion parameters exists[13] which performs slightly better than the *2B* variant in the evaluation performed by Liu et al. [21]. This larger variant comes at the cost of taking much more GPU memory and producing bigger embedding vectors, further increasing computational cost. We opted for the *2B* variant as a compromise between accuracy and computational efficiency (see Section 2.4 for details).

For the closed source OpenAI embedding model we can make no definitive conclusions as to why it performs better, as we have no details about the model such as its architecture or parameter count. The only available publications about OpenAI's embedding models we could find at the time of writing is their research on the usage of contrastive pre-training by Neelakantan et al. [40], however that paper related to a previous version of the embedding models and it is unclear what changes were made to the newer iterations. It should be noted that another embedding variant exists, namely OpenAI's *text-embedding-3-large*, which performs slightly better on natural language benchmarks than the smaller variant used in this thesis, according to OpenAI's own internal research[14]. We chose the smaller variant in this thesis due to it being significantly cheaper.

Most code models perform better on the Defects4J projects than on the industry project. One reason for this is that the buggy commits in the Defects4J projects contain only changes that are related to the fault [9], whereas the industry projects contains unfiltered changes. This removes a lot of noise and irrelevant information from the change data, making it easier to focus on the relevant code changes and find suitable test cases. Indeed, the best performing code models CodeXEmbed [21] and the OpenAI embedding model achieve extremely high APFD values on Defects4J project like *Chart, Lang* and *Math*. The performance is so good on these projects that it would be difficult to measure any general improvements of an approach, possibly making these projects less adequate for future evaluations. We discuss this issue further in Section 5.6 and Section 6.2. Looking at Figure 5.1 we also notice that projects with many bug ids tend to showcase a larger variance in performance, such as for the projects *Closure* and *Jsoup*. This suggests that, to obtain more generalizable results, the number of evaluated commits per project is a more important metric and should be prioritized over simply including a large number of projects, especially since entire projects such as *Chart* and *Lang* can offer limited interpretive value.

[13]`https://huggingface.co/Salesforce/SFR-Embedding-Mistral`

[14]`https://openai.com/index/new-embedding-models-and-api-updates/`

In conclusion for RQ1, the best performing open source model out of the ones we evaluated is the *2B* variant of the CodeXEmbed [21] family, with the closed source OpenAI embedding model performing almost as good. However the most appropriate code model can depend on the use case. If the model should be self-hosted, e.g. due to privacy issues, then the CodeXEmbed or CodeT5+ [20] models are the most suitable. The CodeXEmbed model is quite large, with $\sim 2.6B$ parameters at 32 bits per parameter, it takes up about 10 GB of GPU memory. Users may choose to load the model at lower precisions (8 or 16 bit), which significantly reduces the memory requirements but still makes the loaded model quite large. The CodeT5+ model has a much smaller memory footprint while still maintaining reasonably good performance. If privacy is not an issue, then the OpenAI *text-embedding-3* variants make for a computationally faster solution without sacrificing accuracy, albeit with the downside of having to pay depending on the amount of embedded tokens[15].

Performance Breakdown on Industry Commits

To gain a better understanding on why some code models perform better or worse on certain projects, we take a closer look at a small subset of commits. For Teamscale, we first look at the worst performing commit in terms of APFD for the CodeXEmbed model. CodeXEmbed achieved an APFD of only 5.47 for the commit in question. Since Teamscale is a closed source industry project, we cannot share direct code snippets. However, it is possible to share anonymized information such as a visualization of the embeddings. Dimensionality reduction and the visualization of embedding vectors is performed as outlined in Subsection 5.3.4 and the result for this commit can be seen in Figure 5.2. Since UMAP keeps both local and global structure, we can draw conclusions from local neighbourhoods we find in the visualized data as well as the global distances of elements.

The commit contained changes to various generic type-parameterized getter methods inside an index class in the persistent storage module. These methods are used in many different places throughout the project and don't contain a lot of useful information in their definition on the context they might be used in, which can be seen by the fact that they do not seem to belong to any distinct cluster or group of test methods in the visualization of Figure 5.2. Furthermore, the failing test case is an end-to-end UI test. The group of tests surrounding it are all end-to-end tests that test a UI component which allows the user to create and change which rules and checks should be used for the project being monitored with Teamscale. The failing test performs a set of UI actions to make changes to a setting and saves these changes. The only connection to any method relating to persistence is hidden behind a helper method in the superclass that

[15] `https://platform.openai.com/docs/guides/embeddings/#embedding-models`

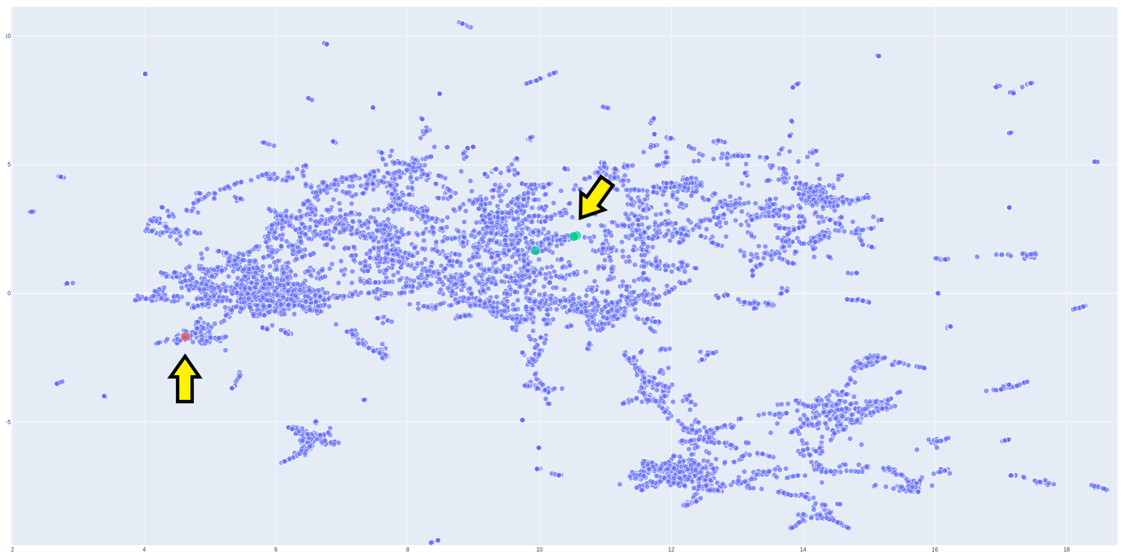

Figure 5.2.: A visualization of CodeXEmbed embeddings for a Teamscale test suite using UMAP as a reduction technique. The dots represent Test Methods, Changed Code Methods and Failing Tests. Arrows highlight the changed code and failing test elements

is invoked inside the test, meaning no actual reference to any of the changed methods is made. These references would also be hidden behind different layers of abstraction, since the test is written to interact with the frontend, whereas the changed code is in the backend. Teamscale uses the PageObject pattern[16], where pages of a website are represented as *Java* objects which offer methods to perform UI actions. This introduces another layer of abstraction, since many low level method calls are now hidden inside multiple helper methods. While it has many advantages for maintainability and code reusability [41], it also makes it increasingly difficult to gather semantic information about a test case using only its immediate source code. This is an inherent problem that large projects with an intricate architecture are bound to encounter. While adding additional context can alleviate this issue, it is difficult to predict which information has to be added since the failing tests are not known beforehand in a real application scenario.

To also give an opposing example, we pick the best performing commit for CodeXEmbed on Teamscale, which achieved an APFD of 99.99. The code changes in this commit related to a method that determines the binary size of an object and was part of a utility class. The failing test case creates such an object and then checks various attributes,

[16]`https://martinfowler.com/bliki/PageObject.html`

among them the binary size. In this case, the relation between code change and failing test is much more immediate, which can be confirmed by looking at the embedding plot of the CodeXEmbed model shown in Figure 5.3. This commit contained none of

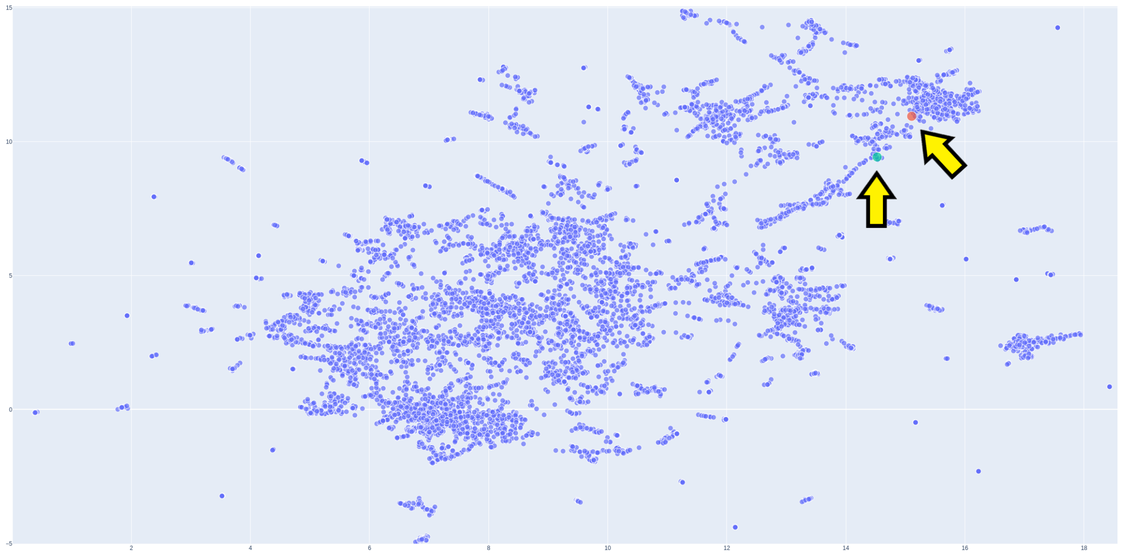

Figure 5.3.: A visualization of CodeXEmbed embeddings for a Teamscale test suite using UMAP as a reduction technique. The dots represent Test Methods, Changed Code Methods and Failing Tests. The arrows highlights the changed code and failing test elements

the difficulties of the worst performing commit, namely the changes and failing test did not belong to different layers, the test case had less nested helper functions and there was a more direct correlation between the changed code method identifiers and the failing test method.

5.5.2. RQ2: IR vs Code Models

When comparing the APFD values of the IR baseline to the best performing code model CodeXEmbed in Table 5.2, we see that CodeXEmbed outperforms the baseline in most projects. The only other code model that is able to achieve higher APFD values than IR is OpenAI's embedding model. However, the difference in APFD between the code models and the baseline is sometimes very small, such as e.g. for projects *Chart*, *Lang* and *Time*. Only the industry project stands out in this regard, with the IR baseline performing well and better than the code models, especially CodeXEmbed. A possible explanation for this is that the *Apache Lucene* implementation collects more context information such as the surrounding class declaration, which can help with heavily nested code as is the case for the industry project.

To assess whether the differences between the IR baseline and the code models are statistically significant, we perform two Wilcoxon signed-rank tests comparing IR with the two best-performing models, namely the OpenAI embedding model and CodeXEmbed. The Wilcoxon signed-rank test is a non-parametric test used to compare two related samples, without assuming a normal distribution [42]. This makes it well-suited for our APFD data, which is collected across multiple projects and may not be normally distributed. The test outputs a p-value, which indicates the probability of observing the measured difference between the approaches if there were no true underlying performance difference. A low p-value (typically below 0.05) suggests that the difference is statistically significant. By applying this test, we can evaluate whether the improvements of CodeXEmbed and the OpenAI embedding model over the IR baseline are meaningful. We implement the test using the Python library *SciPy*[17].

The result of the Wilcoxon signed-rank test between the IR baseline and the CodeX-Embed results is $p = 0.3125$ and $p = 0.7422$ for the IR and OpenAI embedding model. Both values are well above the $p = 0.05$ threshold commonly required for statistical significance [42]. Therefore we conclude for RQ2 that we were not able to measure a statistically relevant improvement from our CETCP approaches to a BM25 based IR baseline based on APFD.

5.5.3. RQ3: Impact of Execution Data

A purely execution time based prioritization strategy has been found to be a strong baseline, especially when evaluating with a cost aware metric such as APFDc [8, 29]. To determine whether code embeddings can improve upon such a baseline, we use the APFDc values from Table 5.3. Looking at the APFDc values per TCP approach, we notice that CodeXEmbed performs very good across all of the Defects4J [9] dataset, with the lowest APFDc value being 92.92. It is also the best performing approach for all projects except for *Lang*, where the IR baseline is the best with an extremely high APFDc of 99.27. However, all approaches seem to achieve good APFDc performance on this project since even the TimeSort baseline has an APFDc of 96.61. This suggests that most failures in this project occur mainly in unit tests with extremely low execution times. Indeed, many of the tests of the *Lang* project had a recorded execution time of 0 milliseconds because they were too quick for the test framework to measure. Every test with such an execution time can be seen as free since they have a cost of 0 in the APFDc calculation. However, this is rather an exception in the dataset. The TimeSort baseline can be seen as an indicator for this phenomenon, as a high APFDc score would indicate

[17]`https://docs.scipy.org/doc/scipy/reference/stats.html`

sorting tests by execution time is enough to almost immediately find all failures. We use the *Closure* project as an opposing example to this, where the TimeSort baseline performs comparatively poor as opposed to the CodeXEmbed approach, which still achieves a good APFDc value. In this project, the test suite is much larger with more longer running tests and less test cases that have a recorded execution time of 0.

As introduced in Subsection 5.5.2, we perform another set of Wilcoxon signed rank tests. The results of this statistical test is $p = 0.0156$ for the comparison of the TimeSort baseline and the CodeXEmbed model. Since the value is well below the $p = 0.05$ significance level, we can confidently reject the null hypothesis that the median difference in performance is not significant. Therefore we can conclude that the embeddings of the CodeXEmbed model add significant value to the execution time information. When comparing the IR to CodeXEmbed, we get $p = 0.0469$ which, while barely below the significance level, allows us to reject the null hypothesis again. In contrast to the result of Subsection 5.5.2, we can therefore conclude that our CETCP approach with the CodeXEmbed model is able to outperform the IR baseline when using the execution time as a cost metric. One possible reason for this general trend is that, while BM25 has hyperparameters to control term saturation and document length normalization as explained in Section 2.3, the IR approach will inevitable tend to prioritize tests that are larger in terms of LoC, since these tests contain many different terms and are therefore counted as relevant for a much larger spectrum of queries. Longer test methods can tend to have longer execution times, although this is not universally the case. Code models do not suffer from this issue, since they do not work on the basis of term frequency. A tuning of hyperparameters k_1 and b (see Equation 2.2) could combat this issue but they would most likely not be able to completely get rid of this problem

For Table 5.3, it is also important to note that the APFDc values did not include the industry project, as collecting this data was not feasible in our time frame. Since the IR baseline outperformed the CodeXEmbed model on Teamscale in terms of APFD, it might have also done so in terms of APFDc which could influence the outcome of the statistical test. More evaluations on industry projects would be required to conclusively examine this, which we also mention in Section 6.2.

5.5.4. RQ4: Computation Time Comparison

When looking at Table 5.4 it is immediately noticeable that IR is quicker by a large margin than all code model approaches. This is no surprise, as the IR approach uses the open source library *Lucene*[18], which as a project of the *Apache Foundation*[19] that has been

[18] `https://lucene.apache.org/`
[19] `https://www.apache.org/`

worked on and refined for many years and is still actively maintained. Considerable research[20] and thought has gone into performance optimization, among other areas.

Comparing the computation times between the various code models, we notice that for CodeBERT [18], CodeT5+ [20] and UniXcoder [19] there is no clear correlation between model size or performance and computation time. All models have a similar parameter count (see Table 2.1 for reference) yet UniXcoder runs for significantly longer than both CodeBERT and CodeT5+. CodeT5+, despite being the best performing model out of those three in terms of APFD, is the fastest one. The OpenAI embedding variant we used is also very fast. However, since the model is closed source, we can only speculate that this performance is likely due to the use of highly optimized, computationally powerful servers handling the API requests. The CodeXEmbed [21] model is the slowest one by often a huge margin, which can be most likely traced back to its large parameter count and to the fact that we load the model into memory for each request, and do so with a full precision of 32 bit per parameter.

Improvements in this regard can be achieved by two ways. Firstly, the model has to only be loaded once and can be reused for concurrent requests, regardless of the project or commit chosen. We loaded the model on each new request due to implementation details, as we had to regularly switch between different models in the implementation and evaluation phase. In addition to this, most of the models support being loaded with a lower precision, 16, 8 or even 4 bit per parameter through a process known as quantization [43]. This way, the smaller model can be loaded faster, takes up less memory and the inference takes less time [43], i.e. the embeddings are generated quicker. Of course, this lowered precision leads to a degradation in accuracy, however most often a reduction to 16 and even 8 bits doesn't have particularly noticeable effects [39, 43].

The second source for optimization is batching, a practice where multiple inputs are placed into a multidimensional vector and given as input to the LLM. This utilizes GPU memory more efficiently and limits unnecessary copying of data between different memory sections or caches [16, 44].

5.6. Threats to Validity

In this section, we discuss potential threats to the validity of our evaluation and the steps taken to mitigate them. We differentiate between internal and external threats to validity.

[20] `https://cwiki.apache.org/confluence/display/lucene/LucenePapers`

5.6.1. Internal Threats

Internal validity concerns factors that could affect the correctness of our results and conclusions. One potential threat is the implementation of the TCP techniques and their evaluation. Errors in the experimental setup, such as incorrect processing of test execution results or incorrect implementation of the prioritization logic, could influence our results. To mitigate this, we base parts of our approach on an already proposed technique by Mattis et al. [34] and we ensure that all baselines and the CETCP approach are executed using the same automated pipeline, reducing the likelihood of inconsistencies. For processing of change and execution data, we use *Teamscale*, a software quality tool that has been developed and refined for a long time and sees a lot of commercial use by different companies[21].

Another internal threat is the use of Defects4J as an evaluation dataset. Defects4J isolates bug-related changes, meaning that test cases are only evaluated on controlled modifications rather than full-scale, real-world software changes [9]. While this setup may not capture all complexities of real projects, it ensures that our prototype can focus specifically on the fault inducing changes rather than unrelated changes. It is useful for an initial baseline performance evaluation, so that core issues in the prioritization approach can be detected in the early stages of the prototype. If a TCP technique fails to correlate code changes to test cases on a filtered dataset like Defects4J, it is unlikely to perform any better on real unadjusted projects. Additionally, Defects4J is widely used in the research community, making it a reasonable benchmark for empirical evaluation and ensuring the reproducibility of our results.

5.6.2. External Threats

External validity concerns the generalizability of our findings beyond the specific datasets and settings used in this study. One key threat is that Defects4J may not fully represent the challenges of real-world software development, where code changes are often larger and contain non-bug-related modifications (e.g. refactorings or documentation updates). To address this, we complement our evaluation with a large-scale industry project, where test cases are prioritized without any pre-filtering of changes. This additional dataset helps assess the effectiveness of CETCP in a more realistic environment.

Another external threat is that our results depend on the specific test suites and fault characteristics of the evaluated projects. Different software systems may have varying fault distributions and test suite behaviors, possibly affecting how well CETCP generalizes to other domains. However, by evaluating multiple projects from Defects4J

[21] `https://teamscale.com/updates-and-publications`

as well as an industry-scale project, we aim to provide a broader understanding of the approach’s performance. As with any empirical evaluation, the generalizability of our results is not guaranteed since the possible combinations of programming languages, frameworks and testing methodologies is too vast to cover at once [45]. While in this research we only focused on tests written in Java, we tried to diversify our testing data by using industry as well as open source projects of varying sizes and application domains. Further empirical results for different projects and programming languages would be required to improve on this. Practical options for this are given in Section 6.2.

6. Future Work

This chapter contains ideas for future work that could be done based on our implementation and evaluation results.

6.1. Expanding the Implementation

One key area for future work is improving the quality of code embeddings used for test case prioritization. Currently, we rely on pre-trained models, but fine-tuning encoder models on more specialized datasets could improve their ability to extract information from code for the purpose of similarity calculations. Although code similarity isn't a typical downstream task and creating a dataset for it is an inherently difficult task [40], datasets for related tasks such as code search or clone detection could be used. Alternatively, a code similarity dataset could be created by selecting a subset of entries from a larger source code dataset such as *CodeNet* [46], similar to what Guo et al. did for UniXcoder [19] in their published fine-tuning code[1].

Furthermore, our study is limited to specific embedding models. Future work could explore new code models as well as the larger versions of OpenAI's embedding model or CodeXEmbed mentioned in Subsection 5.5.1, which may provide better embedding representations at the cost of increased computational overhead.

Another important aspect is computational performance optimization. The current approach processes test cases sequentially, and prioritization can become computationally expensive for large test suites. Potential improvements include batching to allow parallel processing of multiple test cases and quantization methods to reduce the memory and computation cost of embedding models. Applying these optimizations would make CETCP more scalable, particularly for large-scale industry projects.

Besides improving the code embeddings, the source code collection and prioritization process, outlined in Section 4.1 and Section 4.3 respectively, can also be improved. More context, such as the surrounding class definition or the inclusion of helper methods, could add further information to the embeddings of both the changed methods and test methods. This could help bridge contextual gaps created by programming patterns and

[1]`https://github.com/microsoft/CodeBERT/tree/c0de43d3aaf38e89290f1efb771f8de845e7a489/UniXcoder/downstream-tasks/zero-shot-search`

encapsulation, however care has to be put into choosing the right amount of context to add, as many code models have a token input length limit. Next, the prioritization approach, which follows a round robin principle, could be adapted to the changes that are being evaluated. One example for this would be to assign different weights and importance to each change bucket, e.g. determined by its size in LoC. This way, more tests relating to larger changed methods will be prioritized earlier than tests relating to shorter methods such as e.g. *getter* or *setter* functions, where the former would presumably have a higher risk of introducing faults.

6.2. Expanding the Evaluation

The evaluation in this thesis is primarily based on Defects4J [9] and the industry project *Teamscale*. While these datasets provide valuable information, expanding the evaluation would improve the generalizability of our results.

One immediate extension is to consider more projects from Defects4J to increase the diversity of software systems used in the evaluation. Additionally, datasets such as Bugs.jar [47] could provide access to a broader range of real-world defects and test failures, allowing us to validate CETCP's performance on a more diverse set of software projects. These datasets need to fulfill certain criteria in order to be applicable to our type of research, namely there have to be fault-inducing code changes, at least one failure and ideally some filtering of flaky failures. We found other datasets such as Bugswarm [48] which have a similar structure do Defects4J [9], however we concluded that it lacked enough projects with multiple commits that fit our criteria.

Beyond open-source datasets, increasing the number of industry projects in the evaluation would provide better insights into CETCP's practical applicability, especially given the fact that the code models underperformed on the industry project compared to the IR baseline. Industry projects often contain different testing and development practices, larger test suites, and more complex failure patterns, which makes them an essential component of any benchmark for real-world applicability. Furthermore, industry projects regularly have test suites that take hours or even days to complete [1], making the issue of TSO all the more relevant.

7. Conclusions

In this thesis, we investigated the use of code embeddings for TCP as an alternative to traditional IR and coverage-based approaches. We developed a prototype implementation of CETCP, which ranks test cases based on their similarity to code modifications using vectorized representations from pre-trained code models. To evaluate CETCP, we applied it both with and without the inclusion of test execution data. In the evaluation we compared our approach against three baselines: random ordering, a traditional IR method using BM25, and an execution time-based sorting strategy.

To assess the effectiveness of CETCP, we conducted an empirical study on both Defects4J [9], a widely used benchmark dataset, and *Teamscale*, a large-scale industry project. The evaluation measured performance using APFD and APFDc to analyze fault detection efficiency and the impact of execution time.

Our findings show that larger pre-trained code models, such as CodeXEmbed [21], perform better than smaller models in the context of TCP. However, despite this performance, we found no statistically significant advantage of code embeddings over the BM25-based IR baseline when relying only on the generated embeddings and not considering the execution costs of tests. This suggests that while code embeddings are able to capture meaningful semantic information, their advantage in TCP remains limited compared to more established and polished IR techniques.

When incorporating test execution times into the prioritization process, CodeXEmbed significantly outperformed both the BM25-based IR and TimeSort baselines in terms of APFDc, as confirmed by a Wilcoxon signed-rank test. This indicates that prioritization based on code model embeddings may be particularly beneficial in environments where test execution time is an important factor, such as for large projects with longer running test suites.

These results highlight the potential of embedding-based approaches for TCP but also show the need for further improvements. Future work could explore larger models or attempt to improve the performance of pre-trained models through fine-tuning and by providing more context to the code models. Additionally, expanding the evaluation to larger and more diverse datasets could provide deeper insights into the generalizability of embedding-based TCP methods.

Overall, this thesis contributes to the ongoing research on machine learning-based software testing and provides a comprehensive evaluation of code embedding approaches in the context of TCP. While our initial prototype and its results do not immediately demonstrate a clear and decisive advantage over IR-based methods, they provide a foundation for future research into the role of code models in test prioritization.

A. Code Snippets

A.1. CodeXEmbed

```python
def get_detailed_instruct(task_description: str, query: str) -> str:
    return f'Instruct: {task_description}\nQuery: {query}'

class CustomCodeXEmbedModel2B(base_model.__class__):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def encode_queries(self, queries: List[str], batch_size: int = 12, max_length: int = 1024, **kwargs) -> np.ndarray:
        task_description = "Given Code or Text, retrieve relevant content."
        all_queries = [get_detailed_instruct(task_description, query) for query in queries]
        return self.encode_text(all_queries, batch_size, max_length)

    def encode_corpus(self, corpus: List[Dict[str, str]], batch_size: int = 12, max_length: int = 1024, **kwargs) -> np.ndarray:
        all_texts = [doc["title"] + " " + doc["text"] for doc in corpus]
        return self.encode_text(all_texts, batch_size, max_length)

```

Figure A.1.: Code Snippet showing how queries and the corpus are structured for the embeddings generation of *SFR-Embedding-Code-2B_R*. Taken and adjusted from https://huggingface.co/Salesforce/SFR-Embedding-Code-2B_R/discussions/9#67a10598ce3af048c276787b

A.2. UMAP Dimensionality Reduction

```python
from sklearn.preprocessing import normalize
import umap.umap_ as umap

def get_reduced_embeddings(embeddings: np.array, n_components=2):
    normalized_embeddings = normalize(embeddings, norm='l2', axis=1)
    reducer = umap.UMAP(n_components=n_components, n_neighbors=15,
    ↪  min_dist=0.1, metric='cosine', random_state=42)
    return reducer.fit_transform(normalized_embeddings)

```

Figure A.2.: Code Snippet showing the dimensionality reduction parameters for UMAP

Bibliography

[1] S. Amann and E. Jürgens, "Change-Driven Testing," in *The Future of Software Quality Assurance*, S. Goericke, Ed., Springer Verlag, 2019, ch. 1.

[2] K. Herzig, "Testing and continuous integration at scale: Limits, costs, and expectations," in *Proceedings of the 11th International Workshop on Search-Based Software Testing*, ser. SBST '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 38, ISBN: 9781450357418. DOI: `10.1145/3194718.3194731`.

[3] 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, May 2017, ISSN: 0360-0300. DOI: `10.1145/3057269`.

[4] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 222–232, ISBN: 9781450356381. DOI: `10.1145/3180155.3180210`.

[5] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *Journal of King Saud University - Computer and Information Sciences*, vol. 33, no. 9, pp. 1041–1054, 2021, ISSN: 1319-1578. DOI: `https://doi.org/10.1016/j.jksuci.2018.09.005`.

[6] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012. DOI: `10.1109/TSE.2011.106`.

[7] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 419–429. DOI: `10.1109/ICSE.2019.00055`.

[8] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 656–667, ISBN: 9781450355735. DOI: `10.1145/3236024.3236053`.

[9] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440, ISBN: 9781450326452. DOI: 10.1145/2610384.2628055.

[10] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[11] K. Hambarde and H. Proença, "Information retrieval: Recent advances and beyond," *IEEE Access*, vol. PP, pp. 1–1, Jan. 2023. DOI: 10.1109/ACCESS.2023.3295776.

[12] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2377–2388, ISBN: 9781450392211. DOI: 10.1145/3510003.3510050.

[13] B. Thi-Mai-Anh and N. Nhat-Hai, "On the value of code embedding and imbalanced learning approaches for software defect prediction," in *Proceedings of the 12th International Symposium on Information and Communication Technology*, ser. SOICT '23, Ho Chi Minh, Vietnam: Association for Computing Machinery, 2023, pp. 510–516, ISBN: 9798400708916. DOI: 10.1145/3628797.3628963.

[14] S. Kotsiantis, V. Verykios, and M. Tzagarakis, "Ai-assisted programming tasks using code embeddings and transformers," *Electronics*, vol. 13, no. 4, 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13040767.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 9781510860964.

[16] B. Fu, F. Chen, P. Li, and D. Zeng, "Tcb: Accelerating transformer inference services with request concatenation," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22, Bordeaux, France: Association for Computing Machinery, 2023, ISBN: 9781450397339. DOI: 10.1145/3545008.3545052.

[17] Y. Zhao, L. Gong, H. Zhang, Y. Yu, and Z. Huang, "How to get better embeddings with code pre-trained models? an empirical study," 2023. DOI: https://doi.org/10.48550/arXiv.2311.08066. arXiv: 2311.08066 [cs.SE].

[18] 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," 2020. arXiv: `2002.08155 [cs.CL]`.

[19] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[20] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," 2023. arXiv: `2305.07922 [cs.CL]`.

[21] Y. Liu, R. Meng, S. Joty, S. Savarese, C. Xiong, Y. Zhou, and S. Yavuz, "Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval," 2024. arXiv: `2411.12644 [cs.SE]`.

[22] G. Team, T. Mesnard, C. Hardin, *et al.*, "Gemma: Open models based on gemini research and technology," 2024. arXiv: `2403.08295 [cs.CL]`.

[23] A. Kusupati, G. Bhatt, A. Rege, M. Wallingford, A. Sinha, V. Ramanujan, W. Howard-Snyder, K. Chen, S. Kakade, P. Jain, and A. Farhadi, "Matryoshka representation learning," 2024. arXiv: `2205.13147 [cs.LG]`.

[24] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012, ISSN: 0960-0833. DOI: `10.1002/stv.430`.

[25] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer, "Did we test our changes? assessing alignment between tests and development in practice," in *2013 8th International Workshop on Automation of Software Test (AST)*, 2013, pp. 107–110. DOI: `10.1109/IWAST.2013.6595800`.

[26] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 302–311. DOI: `10.1109/ICST.2013.27`.

[27] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001. DOI: `10.1109/32.962562`.

[28] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 268–279. DOI: `10.1109/ICSE.2015.47`.

[29] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 324–336, ISBN: 9781450380089. DOI: `10.1145/3395363.3397383`.

[30] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, Dec. 2021, ISSN: 1573-7616. DOI: `10.1007/s10664-021-10066-6`.

[31] A. Bajaj and O. P. Sangwan, "A systematic literature review of test case prioritization using genetic algorithms," *IEEE Access*, vol. 7, pp. 126 355–126 375, 2019. DOI: `10.1109/ACCESS.2019.2938260`.

[32] H. Huynh, N. Pham, T. N. Nguyen, and V. Nguyen, "Segment-based test case prioritization: A multi-objective approach," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '24, ACM, Sep. 2024, pp. 1149–1160. DOI: `10.1145/3650212.3680349`.

[33] Y. Yang, L. Wang, N. Cha, and H. Li, "A test case prioritization based on genetic algorithm with ant colony and reinforcement learning improvement," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2023, pp. 1588–1593. DOI: `10.1109/COMPSAC57700.2023.00245`.

[34] T. Mattis, L. Böhme, E. Krebs, M. C. Rinard, and R. Hirschfeld, "Faster feedback with ai? a test prioritization study," in *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, ser. Programming '24, Lund, Sweden: Association for Computing Machinery, 2024, pp. 32–40, ISBN: 9798400706349. DOI: `10.1145/3660829.3660837`.

[35] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 1–10, ISBN: 9781450392730. DOI: `10.1145/3520312.3534862`.

[36] X. Li, K. Dong, Y. Q. Lee, W. Xia, Y. Yin, H. Zhang, Y. Liu, Y. Wang, and R. Tang, "Coir: A comprehensive benchmark for code information retrieval models," 2024. arXiv: `2407.02883 [cs.IR]`.

[37] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," 2020. arXiv: `1802.03426 [stat.ML]`.

[38] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008.

[39] R. Jin, J. Du, W. Huang, W. Liu, J. Luan, B. Wang, and D. Xiong, "A comprehensive evaluation of quantization strategies for large language models," 2024. arXiv: `2402.16775 [cs.CL]`.

[40] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, J. Heidecke, P. Shyam, B. Power, T. E. Nekoul, G. Sastry, G. Krueger, D. Schnurr, F. P. Such, K. Hsu, M. Thompson, T. Khan, T. Sherbakov, J. Jang, P. Welinder, and L. Weng, "Text and code embeddings by contrastive pre-training," 2022. arXiv: `2201.10005 [cs.CL]`.

[41] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 108–113. DOI: `10.1109/ICSTW.2013.19`.

[42] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics: Methodology and Distribution*, S. Kotz and N. L. Johnson, Eds. New York, NY: Springer New York, 1992, pp. 196–202, ISBN: 978-1-4612-4380-9. DOI: `10.1007/978-1-4612-4380-9_16`.

[43] R. Gong, Y. Ding, Z. Wang, C. Lv, X. Zheng, J. Du, H. Qin, J. Guo, M. Magno, and X. Liu, "A survey of low-bit large language models: Basics, systems, and algorithms," 2024. arXiv: `2409.16694 [cs.AI]`.

[44] S. M. Nabavinejad, M. Ebrahimi, and S. Reda, "Throughput maximization of dnn inference: Batching or multi-tenancy?," 2023. arXiv: `2308.13803 [cs.DC]`.

[45] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, "The case for context-driven software engineering research: Generalizability is overrated," *IEEE Software*, vol. 34, no. 5, pp. 72–75, 2017. DOI: `10.1109/MS.2017.3571562`.

[46] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021. arXiv: `2105.12655 [cs.SE]`.

[47] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 10–13, ISBN: 9781450357166. DOI: `10.1145/3196398.3196473`.

[48] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 339–349. DOI: `10.1109/ICSE.2019.00048`.