Can CodeT5 embeddings be adapted for efficient Code Clone Detection and Retrieval?

Chinmay Rane, M.Tech

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Data Science)

Supervisor: Professor David Gregg

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Chinmay Rane

August 19, 2022

Permission to Lend and/or Copy

Can CodeT5 embeddings be adapted for efficient Code Clone Detection and Retrieval?

Chinmay Rane, Master of Science in Computer Science
University of Dublin, Trinity College, 2022

Supervisor: Professor David Gregg

With the increase in the number of programs written every day, it becomes challenging to maintain a large software repository. Owing to this, large-scale code clone detection and retrieval have become a necessity. There exist several works offering solutions to solve the problem. However, most of the works have trouble maintaining a balance between accuracy and scalability. Classical approaches have high scalability but lower precision whereas recent neural network-based models have high precision but suffer from scalability. In this work, we show how CodeT5 a recent neural network-based model could be modified to reduce its usage during clone retrieval. Particularly, we fine-tune the architecture to extract code embeddings rich in semantic and syntactic information. Through experiments on the BigCloneBench dataset, we assess the efficacy of the generated code embeddings and show how our proposed Nearest Neighbor-based retrieval approach fetches clones in real-time while achieving comparable accuracy to the original CodeT5 architecture.

Keywords: Clone Detection, Clone Retrieval, Deep Learning, Neural Networks, Code Embeddings, K-Nearest Neighbor Search

Acknowledgments

Firstly, I would like to thank my supervisor, Professor David Gregg, for giving me the freedom to work and experiment with my ideas. My deepest gratitude to David for guiding me throughout, supporting me through difficult times, and motivating me to do my best. Secondly, I would like to express my appreciation to Professor Stefan Weber for providing invaluable suggestions for improving my dissertation. I would also like to thank Mr. Gul Aftab Ahmed for showing me the direction, especially during the implementation phase.

I am grateful to my colleagues who constantly encouraged me and provided me with steady consolation and direction during the dissertation. Finally, I would like to thank my wonderful parents for their constant love and support which provided me with solidarity to successfully complete the thesis. I am grateful for the endless sacrifices they have made in making me stand where I am today. So, I dedicate this dissertation to my parents.

CHINMAY RANE

University of Dublin, Trinity College
August 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code clones are defined in the literature as two identical or similar pieces of code [55]. Code clone detection is an interesting area of research concerning locating code clones within a software repository. There are numerous reasons for creating code clones — modifying existing software code, reformatting the code, rewriting the code base in a faster and stable programming language [51]. Code clones are known to introduce problems in software maintenance such as bug propagation [60]. For instance, a bug identified in the original code needs to be fixed in all the duplicated code fragments. Thus, there exists a need to detect code clones in order to allow the programmers to efficiently manage (reuse) the code fragments. This helps save time, effort, and organization resources. Other advantages of detecting code clones include copyright infringement detection, plagiarism detection [7, 24], code maintainability, and bug detection.

Several tools and techniques [16, 19, 58, 72, 74, 5, 21, 22, 25] have been proposed for detecting code clones from the software code base. However, maintaining a balance between accuracy and scalability is still an active area of research. Code clone retrieval is another interesting challenge that aims at retrieving code clones from a given code repository. There has been a lot of work in this space. Most of the works in this field typically use information retrieval methods [18, 17, 64, 30, 34, 29, 12]. Recent code retrieval approaches focus on leveraging deep learning techniques [50, 28] to solve the problem.

This chapter is structured as follows: Section 1.1 presents the motivation behind this research work. Section 1.2 describes certain code clone specific terminologies and details. Further, the problem statement is identified in Section 1.3, followed by research questions in Section 1.4 and the objectives enumerated in Section 1.5. Finally, Sections 1.6 and 1.7 presents a brief overview of the thesis and its organization respectively.

1.1 Motivation

Today, in most organizations we find multiple software engineers and developers contributing to the company's large software code base. With the enormous amount of source code being generated every day, large-scale code clone detection becomes a necessity. Detecting clones help the developers review and maintain the code base efficiently. In addition, it offers a quick solution to bug propagation. In other words, developers can quickly detect and fix bugs such as security errors in all clones arising due to a potential security flaw in the original code.

Code clone retrieval is generally combined with Integrated Development Environments (IDEs) to offer engineers easy solutions to enhance object-oriented programming. We know that it is impossible for a person to manually monitor the whole code repository and come up with recommendations to improve the code. This is where IDEs such as Visual Studio Code, and PyCharm come into play. They provide users with intelligent suggestions of similar codes already present in the code repository to save their time in implementing the function from scratch and thus, promote code reusability.

A code clone detection and retrieval system need to be accurate and scalable in order to find clones in real-time. Classical systems utilize information retrieval methods such as suffix trees, hashing, bags of tokens, and combinatorial search for detecting and retrieving codes. However, recent neural network-based approaches are able to surpass the accuracy of classical methods by a large margin. Recent works incorporate deep learning and employ huge neural network architectures for both code clone detection and retrieval. These architectures result in exceptional accuracy in code clone detection but do not scale well for code clone retrieval. This is because they perform pair-wise code comparisons to rank similar codes in the database consuming a large amount of time.

This motivates us to adopt deep learning techniques and focus on reducing the time complexity of the neural network approach while maintaining accuracy. We aim to minimize the use of such computationally-heavy neural network-based architectures by relying on its embeddings (representations; explained in Section 2.4.1) instead. Comparing code embeddings is much faster than comparing codes using these architectures. We expect the embeddings to retain rich semantic and syntactic information, thus helping us to efficiently detect and retrieve code clones.

1.2 Background

This section provides a short description of the code clone terminologies. Further, it introduces the types of code clones recognized in the literature [61, 60].

1.2.1 Code clone terminologies

Code Block. Set of programming statements that are defined within a finite scope (generally enclosed within brackets).

Code fragment. A continuous piece of code, specified by the tuple (l, s, e) where 'l', 's', and 'e' represent the source file, start, and end of the fragment respectively.

Clone pair. A pair of similar code fragments, represented by the tuple (f_1, f_2, ϕ) where f_1 and f_2 indicate the pair of clones and ϕ denotes their clone type.

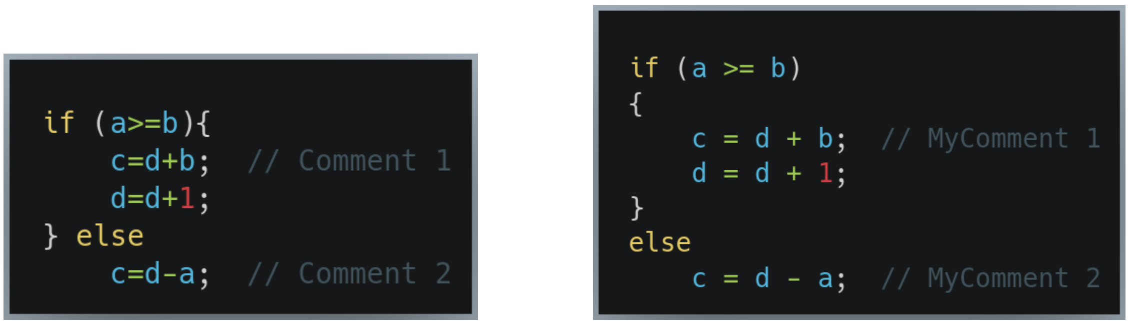
Clone class. A distinct set of code fragments that are similar. It is represented by the tuple (f_1, f_2, ..., f_n, ϕ)

Inter-Project Clone. A clone pair (f_1, f_2, ϕ) identified in two different software repositories.

Intra-Project Clone. A clone pair (f_1, f_2, ϕ) identified within a single software repository.

1.2.2 Code clone types

Code clones are categorized into four types depending upon their complexity and similarity [56]. These are described below:



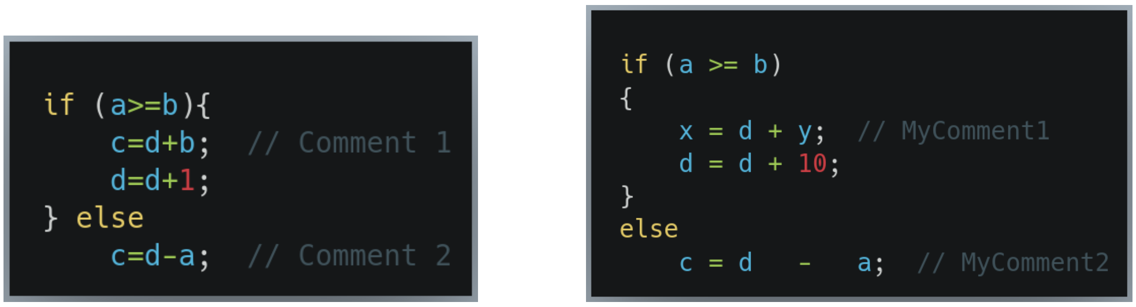
```
if (a>=b){
    c=d+b;   // Comment 1
    d=d+1;
} else
    c=d-a;   // Comment 2
```

```
if (a >= b)
{
    c = d + b;   // MyComment 1
    d = d + 1;
}
else
    c = d - a;   // MyComment 2
```

Figure 1.1: Type-1 Clones

Type-1. Includes two identical code fragments having differences in comments, layout, and whitespaces. Fig. 1.1 represents an example of Type-1 clone.

Type-2. Includes two similar code fragments with differences in their variable names and values. Note that Type-2 clones also include Type-1 differences. An example of a Type-2 clone is displayed in Fig. 1.2

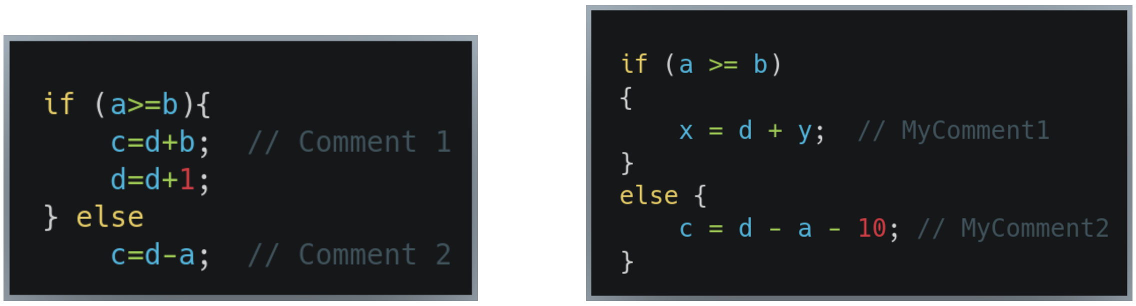


```
if (a>=b){
     c=d+b;   // Comment 1
     d=d+1;
} else
     c=d-a;   // Comment 2
```

```
if (a >= b)
{
    x = d + y;   // MyComment1
    d = d + 10;
}
else
    c = d    -    a;   // MyComment2
```

Figure 1.2: Type-2 Clones

Type-3. Includes two code fragments that have syntactic similarity but differ at the statement level. These are generally modifications of one another where statements are either added, replaced, or deleted. Type-3 clones encompass the Type-1 and Type-2 differences as well. Fig. 1.3 shows an example of a Type-3 clone.

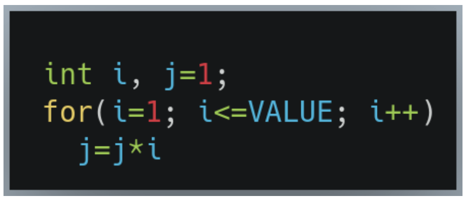


```
if (a>=b){
     c=d+b;   // Comment 1
     d=d+1;
} else
     c=d-a;   // Comment 2
```

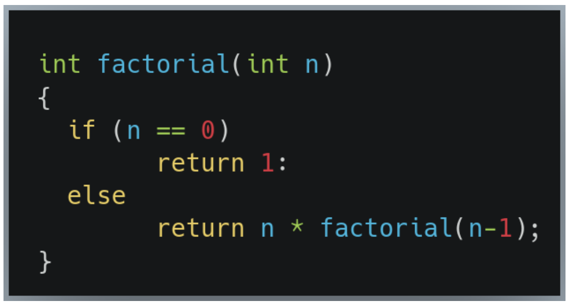
```
if (a >= b)
{
    x = d + y;   // MyComment1
}
else {
    c = d - a - 10; // MyComment2
}
```

Figure 1.3: Type-3 Clones

Type-4. Includes two code fragments that are syntactically dissimilar but semantically similar. Type-4 clones are minor variations of a single functionality. In comparison, Type-3 clones look syntactically similar but are semantically different. Fig. 1.4 illustrates an example of Type-4 clone.



```
int i, j=1;
for(i=1; i<=VALUE; i++)
  j=j*i
```



```
int factorial(int n)
{
  if (n == 0)
        return 1:
  else
        return n * factorial(n-1);
}
```

Figure 1.4: Type-4 Clones

1.3 Problem Statement

While classical code clone detection approaches [20, 39] do an exceptional job of identifying Type-1 and Type-2 clones on a large scale, they struggle with detecting Type-3 and Type-4 clones. In real life, most of the code clones exist as Type-3 and Type-4 clones [53]. Hence, there exists a requirement for a system that is able to identify semantically similar clones whilst being scalable.

A few works that achieve high accuracy and scalability on code clone detection applications make assumptions about the target domain [32, 6]. For instance, [6] detects clones only in android applications. Such code clone detection approaches are domain-specific and do not work well when applied on large-scale domain-independent datasets [56].

Recent works [13, 69] for code clone detection and retrieval leverage deep learning architectures (CodeBERT, CodeT5 resp.) to achieve remarkable accuracy at the expense of scalability. These works require quadratic time complexity $O(n^2)$ for retrieving clone pairs in the dataset. This is because these approaches perform pairwise code comparisons to identify the clone pairs with high similarity. In other words, to identify all clones in the repository, every code fragment needs to be compared against another resulting in $O\left(n^2\right)$ comparisons or $(n \times (n-1)/2)$ comparisons to be precise, where n is the number of code fragments in the dataset.

These aforementioned challenges bring the need for developing a system to address code clone detection and retrieval in an efficient and scalable way. An ideal system must have low space and time complexity and must quickly adapt to the ever-growing amount of source code every day.

1.4 Research Questions

According to the problems stated, the following research questions can be identified:

Research Question 1. Can we modify the CodeT5 architecture to make it scalable for code clone detection and retrieval?

CodeT5 [69] is a transformer-based architecture trained for code understanding and generation tasks. The transformer architecture is developed by Vaswani et al. [66] for language processing tasks, that work by learning the interrelation between words in the sentence using a novel "*attention*" mechanism (explained in section 3.3.1). The CodeT5 achieves state-of-the-art results in code clone detection and other downstream tasks. CodeT5 takes as input two code fragments, produces a single code embedding, and further employs a classifier to detect the level of similarity between them.

Research Question 2. Does the CodeT5 representations capture different functionalities of code corpus?

The purpose of this question is to verify whether the code embeddings generated by the Codet5 model capture rich syntactic and semantic information present in the code fragments. In this work, we will plot visualizations to derive insights to address the question.

Research Question 3. How effective is the Nearest Neighbor search on the CodeT5 embeddings?

Using CodeT5-generated embeddings, we aim to perform a nearest neighbor search on the code repository in order to rank and identify the closest clone pair given a query code. This would help answer the question — "Whether the CodeT5 embeddings can be adapted to efficient code clone retrieval?". In addition, we aim to analyze the accuracy and scalability of the proposed information retrieval system using CodeT5-generated embeddings.

1.5 Research Objectives

We formulate the following set of research objectives based on the research questions posed in the previous subsection.

1. To assess whether the modified CodeT5 architecture results in high performance.
2. To explore whether the CodeT5-generated embeddings distinctly capture code characteristics.
3. To identify whether the generated code embeddings can be segregated into unique categories.
4. To figure out the effectiveness of CodeT5-generated embeddings for code clone retrieval at scale.

1.6 Thesis Overview

The thesis presents an approach to detecting and retrieving code clone pairs while requiring lower memory and time when compared to using the original CodeT5. The CodeT5 architecture is adopted and modified in an attempt to make it scalable. The code clones are detected by fine-tuning the pre-trained model in a supervised learning setup. Two different training strategies are evaluated for producing semantically rich code embeddings. The results using these strategies are further compared against standard classification metrics.

The fine-tuned model is used to generate embeddings for every code in the dataset. The generated embeddings are then assessed to verify whether they capture meaningful interpretations (characteristics of the codes in the dataset). Ideally, the embeddings of clone pairs must appear closer in the feature space and farther from the non-clone pairs. Finally, the generated embeddings are used in the code clone retrieval task. Retrieving code clones using code embeddings has the advantage of being faster and space efficient than pairwise code comparison using the original CodeT5 architecture.

1.7 Thesis Structure

The flow of the thesis is presented as follows:

Chapter 2 [**Literature Review**] **-** Presents existing code clone detection and retrieval approaches.

Chapter 3 [**Methodology**] **-** Describes the dataset used in the study, followed by the text pre-processing techniques used. Further, explains the original CodeT5 architecture and the proposed methodology to train the modified CodeT5 architecture.

Chapter 4 [**Experiments and Results**] **-** Describes the evaluation metrics used, followed by results discussion and visualization.

Chapter 5 [**Conclusion and Future Work**] **-** Concludes the thesis and identifies the limitations and presents the future work that could be carried out in this direction.

Chapter 2

Literature Review

Code clone detection and retrieval have been an interesting area of research for a long time and we can find a significant amount of work done in this direction. This chapter reviews the research work carried out in the literature by taking as reference some of the most comprehensive surveys conducted [60, 51, 38, 1, 75]. Firstly, we discuss "why code clones often arise in practice". Secondly, we highlight the problems caused by the presence of code clones. Further, we review the classical code clone detection and retrieval approaches, followed by the deep learning approaches. Finally, we talk about the measures to evaluate code clones.

2.1 Why do code clones often arise in practice?

Code clones exist in a software repository for numerous reasons. Some of the reasons identified in the literature are:

1. **Promote code reusability**

   A widely used practice in the development community is to copy-paste-modify code from various web sources such as *StackOverflow, Github Gists.* This creates code clones within the software repository. Developers predominantly use this programming practice in an attempt to save the time and effort required to implement everything from scratch. However, it is worth noting that this sort of copying from random sources is completely unacceptable in most commercial software development environments. It involves a massive breach of copyright and leaves the company very prone to being sued for breach of copyright. A work [27] discovered that developers also clone the existing software repository and modify the code to adapt it to their use case. This is totally legal but can be bad a software engineering practice, depending on how large are the cut and pasted fragments.

2. Maintenance Benefits

Existing codes are well-tested and approved by a large team/community. Therefore, it makes sense to build new code on top of the existing ones. Rewriting codes from scratch consumes time and can introduce new errors. In certain companies, developers are often asked to reuse the code and adapt it according to the new client's requirements.

3. Starter code in IDEs

Several IDEs such as *Eclipse, IntelliJ IDEA* provide the user with the flexibility to choose the type of project he/she wants to create based on the specifications such as Java, Maven, Gradle, Android, etc. When the user selects an option, the IDE automatically generates the boilerplate code for the user. Another example is IDEs automatically generating getter setter methods for class variables. This also contributes to creating code clones.

4. Project Deadlines

Often, developers are subjected to pressure due to project deadlines. Thus, the developers can make bad choices under pressure and copy code from other sources to adapt it to fit their specifications.

5. Accidental Cloning

It is nearly impossible to monitor the organization's large software repository manually. Developers may unintentionally write code that already exists in the code repository. This creates duplicated code fragments that can be solved by IDEs automatically suggesting the developers to import the existing code to promote code reusability.

2.2 Problems caused by code clones

While code clones can be beneficial in saving time and resources for the organization. They are known to introduce problems for the organization as well. Some of the well-known issues identified in the literature are:

1. Bug propagation

If the original code contains bugs or errors, they get propagated to all its clones [36]. Thus, creating a major problem in code maintenance. For instance, a flaw in the original code can result in a potential security breach, hence, the software maintainers need to take immediate actions to resolve the issue across all the clones.

2. Introduce new errors

Many a time, developers copy inconsistent and incomplete code from various sources and merge them [56]. They might overlook important information such as the required package versions and their dependencies. This introduces new errors in their software system as a result of incompatibility of the copied codes.

2.3 Classical clone detection and retrieval techniques

Classical code clone detection and retrieval techniques utilize information retrieval methods. They typically vary based on the way of representing source codes, comparison algorithms used, and the computational complexity. Some of the classical techniques include suffix trees [25, 32], bags of tokens [25, 36, 4], hashing, combinatorial search, abstract syntax trees (ASTs) [5, 67, 73], text representations [4, 8, 23, 10], and program dependence graphs (PDGs) [31, 33, 37], etc. These approaches are described in brief in the subsections below:

2.3.1 Text-based approaches

Text-based techniques use traditional natural language processing techniques such as text cleaning (comments removal, whitespace removal), and text normalization (stemming, lemmatization) for pre-processing of the source codes. They further employ string-matching algorithms to identify code clones. For instance, Baker et al. [3] detect clones by running a string-matching algorithm line-by-line. Ducasse et al. [10] uses a string-based Dynamic Pattern Matching algorithm to detect clones and proposes a solution that is language-independent.

2.3.2 Token-based approaches

Unlike text-based approaches, a token-based approach applies a transformation to the source code and converts it into a sequence of tokens. This is generally done by parsing the code using a lexical analyzer. The sequence of tokens is further scanned to detect duplicated subsequences, that are returned as clones. Token-based approaches identify clones with high recall and are robust to code changes such as spacing and formatting [51]. CP-Miner [36] and Baker's *Dup* [4] are two state-of-the-art token-based code clone detection techniques.

2.3.3 AST-based approaches

Abstract syntax tree-based approaches convert the programs into a parse tree-based structure using a compiler of the programming language. Further, they employ a tree-matching algorithm to detect similar sub-trees. The source codes corresponding to the sub-trees are returned as clones. While AST-based approaches have high precision, they suffer from scalability issues. This is because of the lack of efficient algorithms for approximately matching ASTs. Moreover, the trees can have a large depth and consume a significant amount of memory. A few pioneering works done in this direction using AST-based techniques include Yang's approach [73] and Baxter's *CloneDR* [5].

2.3.4 PDG-based approaches

Unlike the aforementioned approaches, PDG-based approaches utilize semantic information of the program to generate code representations. They work similar to ASTs but also encode information pertaining to control flow and data flow within the source code. Therefore, they are more robust to changes within the code such as reordered statements, insertion, and deletion of code. Komondoor and Horowitz [31] used a variable dependency graph to represent the source code. They converted the task of finding clones into detecting isomorphic subgraphs within the program dependency graph.

2.3.5 Hybrid approaches

Hybrid approaches encompass a blend of the aforementioned approaches. Researchers have tried to combine multiple program representations in order to balance high precision, recall and achieve scalability. For instance, Koschke et al. [32] proposed a token-based approach with ASTs for representing the source codes, and his approach resulted in linear scaling with time, making it appealing for large-scale applications. Another pioneering work by Jiang et al. [22] includes a hybrid approach to clone detection. They represent the program using AST and convert it into a vector representation. Further, they employ a hashing algorithm to cluster the vectors. The vectors within the same cluster are returned as clones.

2.4 Deep learning-based clone detection and retrieval methods

Deep learning-based techniques use Neural Networks (NN) [72, 2, 50] to address clone detection and retrieval. These techniques generally differ in the way they represent the programs or train various NN architectures. State of the art deep learning models have high precision and recall in clone detection but do not scale well in retrieval tasks. This section provides a brief summary of the existing works employing deep learning techniques for both tasks. Some excellent survey papers highlighting the pros and cons of deep learning techniques for clone detection are [61, 35].

Code2Vec. Alon et al. [2] proposed a NN-based Code2Vec algorithm that transforms the code fragments into continuous distributed representations. They represent each source code fragment as a fixed length contiguous vector (code embeddings; explained in Section 2.4.1) that is able to retain semantic information within the code. The code embeddings are able to solve analogical reasoning tasks similar to Word2vec and Glove. Word2Vec [42] and Glove [45] are algorithms used in natural language processing to represent words as continuous vector spaces that capture word-to-word relationships. Code2Vec was originally developed to predict the method name given the code snippet but was later used in other works [47, 26] for performing semantic code clone detection.

RNNs and LSTMs. White et al. [72] use traditional Recurrent NNs (RNNs) for modeling structural and syntactical information present within the code fragments. They achieved 93% precision at detecting method-level and file-level clones. Wei et al. [71] used Long Short-Term Memory (LSTM) to learn code representations by modeling ASTs. The representations are learned by minimizing a hash function which computes the hamming distance between the hash codes of the code pairs.

Ensemble learning. Sheneamer and Kalita [59] extract syntactic and semantic features from ASTs and PDGs to train machine learning and deep learning classifiers for code clone detection. They further combine multiple classifiers in an ensemble to predict the clone type.

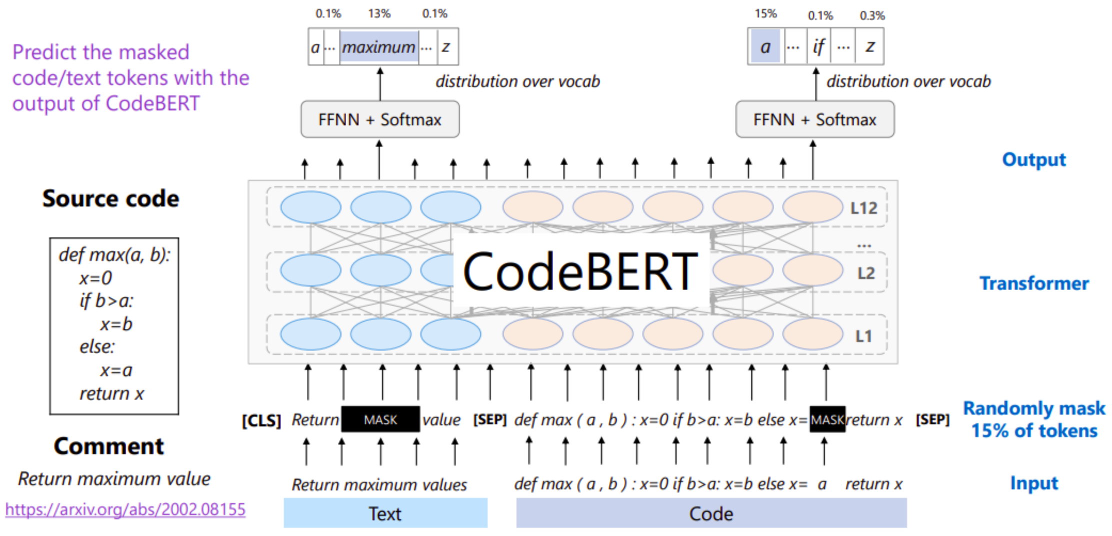


Figure 2.1: Pre-training CodeBERT on Masked Language Modeling[1]

CodeNN. Gu et al. [14] proposed CodeNN, a deep neural network to perform code search on large datasets. CodeNN embeds the programs and their descriptions into a high-dimensional vector space referred to as embeddings (explained in Section 2.4.1), such that the description and its respective code snippet appear closer in the feature space. Thus, given a description of the method, CodeNN automatically retrieves matching code snippets from the database.

CodeBERT. CodeBERT [13] is a state-of-the-art neural network developed for code understanding and generation tasks. It learns neural representations of natural language and programming language pairs by modeling them using a popular neural network architecture. Particularly, it uses BERT [9] as its underlying architecture and pre-trains its parameters using two different training objectives — Masked Language Modeling and a novel Replaced Token Detection task. The training takes as input the program description along with the actual program and returns a single representation containing rich information about the program and its description. The pre-trained architecture is then fine-tuned for the multiple downstream applications such as code search, clone detection, code summarization, etc. Figure 2.2 shows CodeBERT being trained on the Masked Language Modeling objective.

Graph NN. Wang et al. [68] were the first authors to use Graph Neural Networks (GNNs) to detect semantic clones. They proposed FA-AST (Flow Augmented AST), a

[0]Code Intelligence

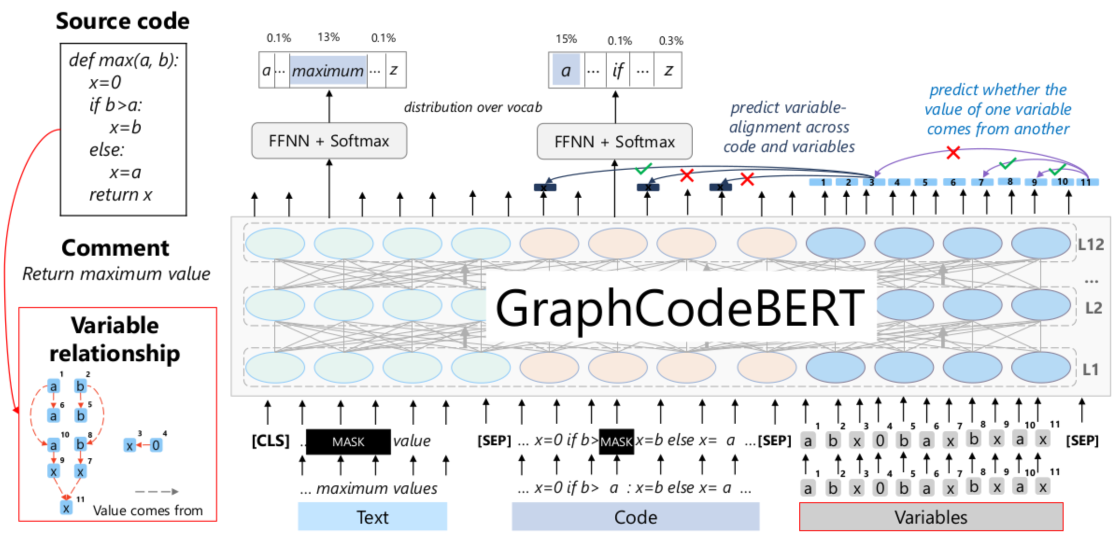


Figure 2.2: Pre-training GraphCodeBERT on Masked Language Modeling[2]

technique that leverages control and data flow graphs to enhance the ASTs. The authors employ GNNs on FA-AST to measure the similarity between the code pairs.

GraphCodeBERT [15] was developed to solve an important limitation of CodeBERT, that is, to incorporate structural information of the code. CodeBERT represents programs as a sequence of tokens and ignores the inherent structure of code which can provide crucial information about the code semantics. Therefore, the authors use the program's data flow graph to encode semantic information relating to the dependencies between the variables. GraphCodeBERT is pre-trained on three tasks — Masked Language Modeling, predicting dependency edges in the code structure, and aligning representations between source code and code structure. The proposed model improves CodeBERT results and achieves state-of-the-art performance on four downstream tasks.

CodeT5. Wang et al. proposed CodeT5 [70], an encoder-decoder-based Transformer model that leverages user-defined identifiers to capture semantic properties in the code. The authors extend a Seq2Seq-based T5 model [49] (developed for natural language understanding and generation tasks) for code understanding and generation applications. They propose a novel identifier-aware pre-training objective to model the crucial token-type (identifier) information from the programming language. CodeT5 results in state-of-the-art performance on fourteen code-related sub-tasks defined in CodeXGLUE [41] (including code clone detection) and significantly outperforms the prior architectures.

[1]Code Intelligence

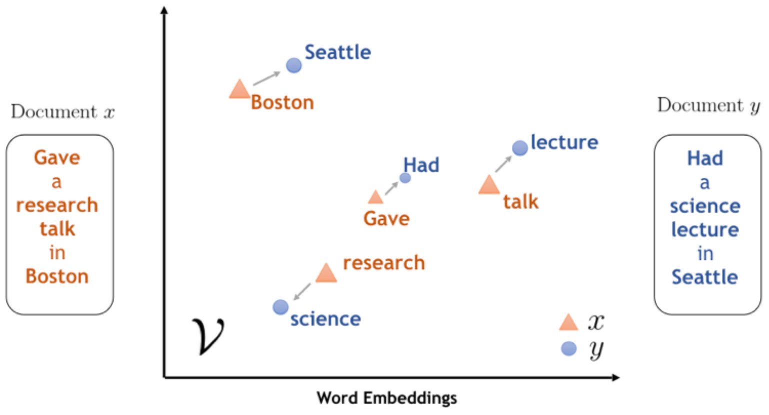


Figure 2.3: Visualization of word embeddings for non-stop words in documents x and y produced by Word2Vec algorithm. Orange triangles and blue dots represent words in the documents x and y respectively[3]

2.4.1 Embeddings

Embeddings refer to a low-dimensional continuous space in which one can project the high-dimensional vectors. They capture rich semantic information present in the inputs such that the inputs which are related have their embeddings closer in the feature space. The closeness between the embeddings is determined by the level of similarity between the inputs.

In this section, we discuss a few popular word embedding algorithms and later introduce code embeddings. Our idea is to use the code embeddings to perform both code clone detection and retrieval in order to minimize the use of computationally-heavy architectures.

Word embeddings

Word embeddings are a way of representing each word in the input text. The concept of word embeddings became popular after Tomas Mikolov proposed Word2Vec algorithm [42] in 2013. A year later, Pennington et al. proposed GloVe (Global Vectors for word representation) [44], an improved word embedding algorithm which models the co-occurrence word matrix to capture word-to-word relations. Both the algorithms assign a single representation for every word in the corpus. These word representations can be used for several downstream applications including machine translation, text summarization, text classification, etc.

All the word embedding algorithms have a common property — the embeddings of similar words appear closer in space and those of non-related words occur farther in space. The illustration in Fig. 2.3 shows how non-stop words in both documents relate to each other in the embedding space. This leads us to think about how the authors enforce this property in the word embeddings. In reality, they design loss functions that guide the neural network to generate such representations. For instance, the Word2Vec algorithm uses negative sampling loss to impose the property.

One major drawback of the Word2Vec and GloVe algorithms is that they are unable to address the issue of polysemy. Polysemy refers to the co-existence of words having multiple meanings. For example, the word "bank" can refer to a river bank or a place to store money safely. Thus, the word "bank" should have several different embeddings based on the context. However, the algorithms assign a single representation to every word. This is where the idea of contextual embeddings evolved. Recent architectures such as BERT [9], ELMo [46] produce contextual embeddings for every word based on the surrounding words (context).

Code embeddings

Similar to word embeddings, we aim to generate code embeddings using CodeT5 architecture for every code in the repository. These code embeddings will be used for clone detection and retrieval. The idea of using code embeddings emerged to minimize the use of the architecture during inference. For clone detection, we will compare embeddings of the code pairs to determine the level of similarity between them. Similarly, for clone retrieval, we will experiment using the nearest neighbor search for comparing the query code embedding with the pre-computed embeddings of the codes in the dataset.

[3] Word embeddings - IBM Research

2.5 Conclusion

This chapter provided a brief overview of the existing works in code clone detection and retrieval. Initially, we described some of the classical approaches, followed by the more recent deep learning-based approaches. The classical approaches suffer from either low precision or low recall. Neural networks overcome their limitations but suffer from scalability issues. So, we can view it as a tradeoff between precision, recall, and scalability. In this work, we aim to achieve high precision, and high recall while reducing the time complexity compared to $O(n^2)$ pairwise comparison, by utilizing CodeT5 embeddings instead of the architecture for pairwise code comparisons.

Chapter 3

Methodology

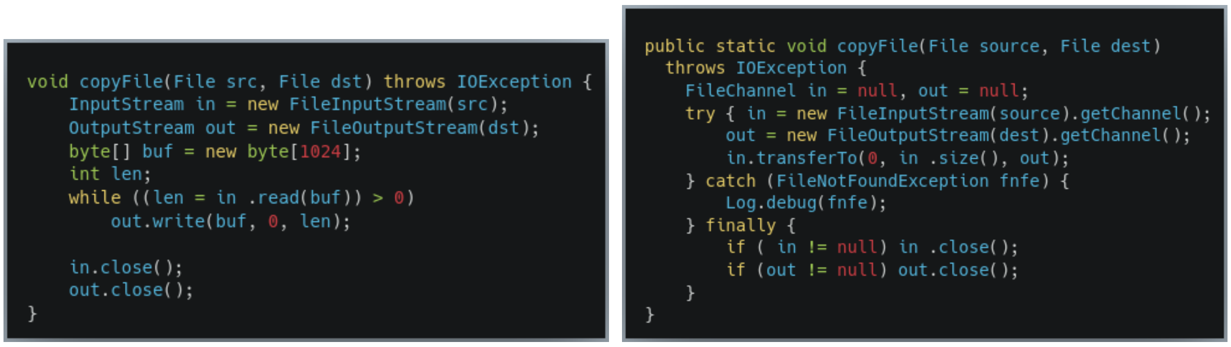
3.1 Data Overview

CodeXGLUE [41] is a benchmark dataset containing a collection of 10 code-related tasks across 14 datasets. There exists two popular datasets for clone detection and retrieval defined in CodeXGLUE which are BigCloneBench [63] and POJ-104 [43] respectively. There is another popular dataset for code clone detection which is the Google Code Jam repository[1] (GCJ). We use the filtered BigCloneBench dataset for both tasks because it is commonly used in existing clone detection works. In addition, it contains enough code fragments and a large vocabulary size to fit within our computational limits. Table 3.1 shows the statistics of the aforementioned datasets. The section below describes the BigCloneBench dataset and discusses the pre-processing performed on the codes.

Table 3.1: Characteristics of the datasets

Characteristics	**BigCloneBench**	**GCJ**	**POJ**
Code fragments	9,134	1,669	52,000
Vocabulary Size	77,535	8,033	-
Average code length	32.89	58.79	35.25
True clone pairs	336,498	275,570	12,974,000
False clone pairs	2,080,088	1,116,376	1,339,000,000
Language	Java	Java	C/C++

[1]https://code.google.com/codejam/contests.html



```java
void copyFile(File src, File dst) throws IOException {
    InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dst);
    byte[] buf = new byte[1024];
    int len;
    while ((len = in .read(buf)) > 0)
        out.write(buf, 0, len);

    in.close();
    out.close();
}
```

```java
public static void copyFile(File source, File dest)
  throws IOException {
    FileChannel in = null, out = null;
    try { in = new FileInputStream(source).getChannel();
        out = new FileOutputStream(dest).getChannel();
        in.transferTo(0, in .size(), out);
    } catch (FileNotFoundException fnfe) {
        Log.debug(fnfe);
    } finally {
        if ( in != null) in .close();
        if (out != null) out.close();
    }
}
```

Figure 3.1: A sample true clone pair in the BigCloneBench dataset classified as Weak Type-3 clone

3.1.1 BigCloneBench

BigCloneBench is a widely used benchmarking dataset for comparing clone detection algorithms. It holds a collection of 6 million true clone pairs and 260k false clone pairs in Java programming language covering 10 different functionalities. BigCloneBench includes all four types of code clones (Type-1/2/3/4) with different strengths. Wang et al. [68] shared a filtered version of the BigCloneBench dataset by removing programs that were not labeled as true/false clone pairs.

The filtered BigCloneBench dataset consists of 9,134 programs written in Java covering several functionalities such as file transfer, SQL queries, etc. The training dataset includes 901,028 equally balanced true and false clone pairs. The validation and test set consists of 415,416 clone pairs each. Fig. 3.1 illustrates a true clone pair in the dataset. The clone pair programs shown in the figure implement the function of copying the content of a source file into the destination file in two similar ways. We use the filtered dataset for our experiments. From here on, we will refer to the filtered version of the dataset as BigCloneBench (BCB) unless stated otherwise.

3.2 Data Pre-processing

This section describes the pre-processing steps applied to all the programs in the BigCloneBench dataset. During inference, the pre-processing steps are applied to every query code to convert it into a standard form to be input to the architecture. Pre-processing is required to clean and normalize the source codes to improve the model performance. We employ the following steps to pre-process the codes

Removal of unnecessary code. The irrelevant parts of the source code such as the whitespaces and the comments are discarded. This leaves us with only the crucial parts of the code such as the function name, its parameters, user-defined identifiers, variable names, and values.

Tokenization. Tokenization is the process of splitting the code fragments into smaller units called tokens. It is one of the crucial steps involved in improving the model performance by handling out-of-the-vocabulary (OOV) words. Similar to BERT and GPT, we employ CodeT5's pre-trained Byte-Pair-Encoding (BPE) tokenizer which is trained as proposed by Radford et al. [48]. The BPE tokenizer reduces the sequence length and is determined to work better on the understanding and generation tasks [69].

Code-specific features. As proposed in CodeT5, we extract the code-specific features by leveraging the token-type (identifier) information in the source code. The identifiers are code tokens that are common to many programming languages capturing rich code semantics. The CodeT5 tokenizer uses the identifier information and converts the code into features to be utilized during training.

Encoding programs. The original CodeT5 architecture is pre-trained on natural language description and program pairs (NL-PL) or program-program pairs (PL-PL). It concatenates the tokens of the bimodal inputs by putting a $[SEP]$ token between them. $[SEP]$ acts as a separation between the PL-PL or the NL-PL pairs. Since we modify the architecture to take as input a single code we use $[PAD]$ tokens after the code tokens to pad the tokens to their maximum length and ignore the $[SEP]$ token. Our input sequence can be represented as $([CLS], c_1, c_2, c_3, ..., c_n, [PAD], [PAD], ..., [PAD])$ where $[CLS]$ represents the classification token, c and n represents the individual code tokens and number of tokens in the program respectively.

3.3 Original CodeT5 Architecture

A simplified version of the original CodeT5 architecture is depicted in Fig. 3.2. This section explains the working of the CodeT5 architecture. In the next section, we explain the modified architecture and depict how it solves the limitations of the original model.

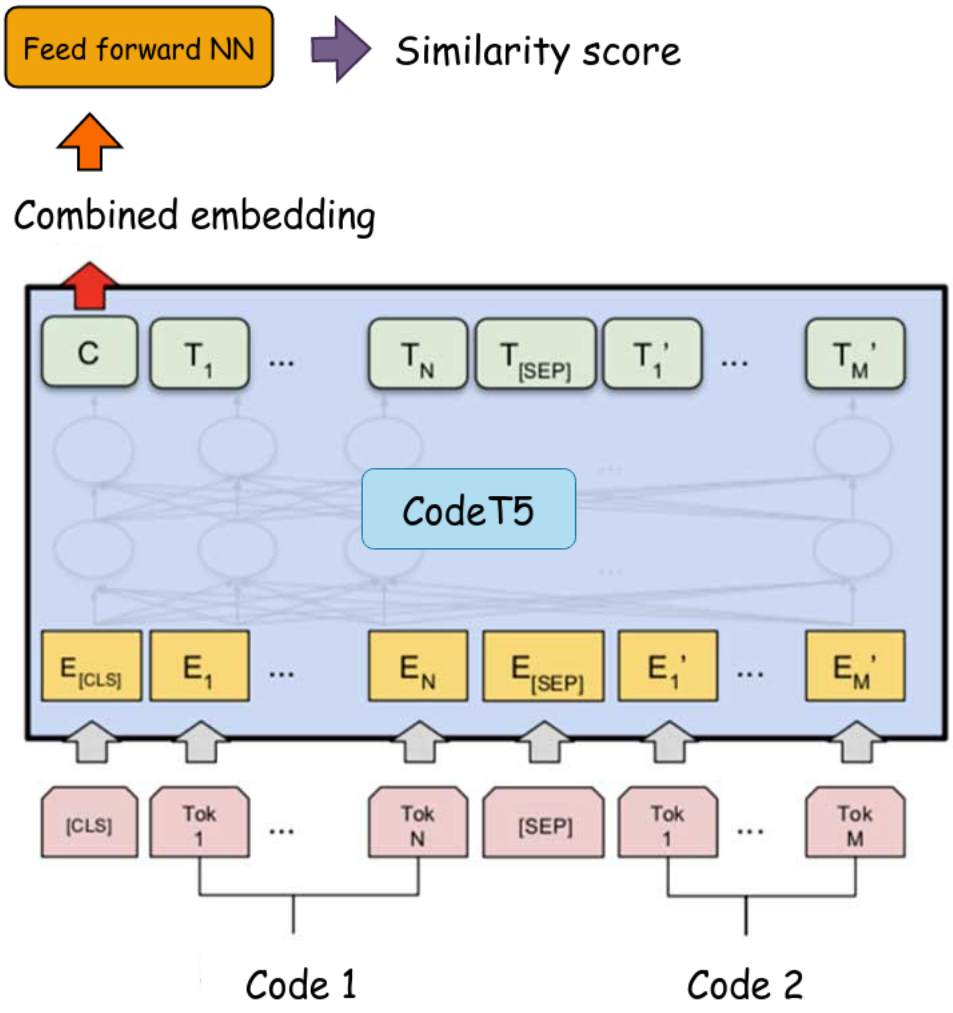


Figure 3.2: Working of the CodeT5 architecture

3.3.1 Working of CodeT5 architecture

As can be seen in Fig. 3.2, the CodeT5 architecture takes as input a sequence of code-code (PL-PL) pairs, performs certain computations, and returns the embeddings corresponding to each input token. The embedding of the first token $[CLS]$ contains a rich summary of both the codes which is further used for performing clone detection and retrieval. The main computation performed by the architecture is defined as *attention* proposed by [66] (explained in Section 3.3.1). The CodeT5 architecture is pre-trained on 3 different objective functions (Masked Span Prediction, Identifier Tagging, and Masked Identifier Prediction) providing feedback to the model parameters for enriching the code understanding. We briefly describe the attention operation along with the objective functions below.

Attention. The encoder of the CodeT5 performs an attention operation on the input tokens to understand how each token of the first code relates to its own tokens as well as to the tokens of the second code. In other words, the attention mechanism generates contextualized token embeddings capturing inter-code (between code-1 and code-2) and intra-code (within code-1/code-2) relationships. The self-attention operation generates three vectors K, Q, V for each token which act as the key, query, and value pairs. The self-attention operation is defined by the equation 3.1.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.1}$$

where d_k represents the dimension of the key vector and $softmax$ is a non-linear function applied to convert the raw scores into normalized probabilities.

Masked Span Prediction (MSP). Denoising has proved to be effective in improving model understanding against a wide variety of text-related tasks [49]. It enables the model to predict noised tokens using the denoised ones. Thus, it learns the interaction between noised and denoised tokens. The MSP objective function is a type of denoising objective in which some code tokens are initially corrupted using certain noising functions and given as input to the model for reconstructing the tokens. The model leverages the non-corrupted tokens to predict the corrupted ones. The CodeT5 authors employ identifier-aware denoising to improve the code understanding process. The objective can be represented by equation 3.2.

$$\mathcal{L}_{\mathrm{MSP}}(\theta) = \sum_{t=1}^{k} -\log P_\theta \left(x_t^{\mathrm{mask}} \mid \mathbf{x}^{\backslash \mathrm{mask}}, \mathbf{x}_{<t}^{\mathrm{mask}} \right) \tag{3.2}$$

where x_t^{mask} is the masked token to be predicted, $\mathbf{x}^{\backslash \mathrm{mask}}$ represents the masked input, and $\mathbf{x}_{<t}^{\mathrm{mask}}$ represents the sequence generated till time t. The goal is to optimize the model parameters θ to minimize the loss function.

Identifier Tagging (IT). CodeT5 introduces this novel objective function to amalgamate code-specific structural information in the architecture. The objective of this function is to label each token of the sequence and hence it is also referred to as a sequence labeling task. Sequence labeling aims at predicting whether or not each token is an identifier in the programming language. Essentially, the authors compute Binary Cross Entropy loss at each token given by the equation 3.3.

$$\mathcal{L}_{\mathrm{IT}}(\theta) = \sum_{i=1}^{m} -[y_i \log p_i + (1 - y_i)\log(1 - p_i)] \tag{3.3}$$

where p_i is the probability over the vocabulary, y_i is the actual label (1 if the token is an identifier, 0 if not) and θ represents the model parameters.

Masked Identifier Prediction (MIP). In this objective function, all the identifiers in the program are masked and the model is trained to predict them, given the natural language description of the program. This objective enforces the model to learn about code semantics. The model predicts the identifiers in an auto-regressive fashion. The loss function is similar to equation 3.2 and represented by equation 3.4.

$$\mathcal{L}_{\mathrm{MIP}}(\theta) = \sum_{j=1}^{|I|} -\log P_\theta(I_j \mid \mathbf{x}^{\backslash \mathbf{I}}, \mathbf{I}_{<t}) \tag{3.4}$$

where I_j is the predicted identifier, $\mathbf{x}^{\backslash \mathbf{I}}$ is the masked input, and $\mathbf{I}_{<t}$ is the sequence predicted till time t.

The overall CodeT5 pre-training loss function is a combination of the three loss functions with equal weightings given in the equation 3.5.

$$\mathcal{L}_{final}(\theta) = \mathcal{L}_{\mathrm{MSP}}(\theta) + \mathcal{L}_{\mathrm{IT}}(\theta) + \mathcal{L}_{\mathrm{MIP}}(\theta) \tag{3.5}$$

3.3.2 Opportunity of improving the CodeT5 architecture

The pre-trained CodeT5 model achieves remarkable results in detecting all types of clones with great precision and recall [69]. This behavior is expected because the pre-training three objectives guide the model in understanding the code syntax and semantics well. Coming to the clone retrieval task, CodeT5 retrieves clones with great accuracy because it is capable of distinguishing between codes and calculating the level of similarity between code pairs.

However, it has a major drawback in clone retrieval — pairwise code comparisons for determining the similarity level, thus requiring $O(n^2)$ comparisons ($(n \times (n-1)/2)$ times architecture usage) and consuming a great deal of time. As can be seen in Fig. 3.2, the architecture takes a single input containing the code-1 and code-2 tokens concatenated and compute a combined embedding containing rich information about the relationship between both the programs. This embedding is further fed into a 2-layer feedforward

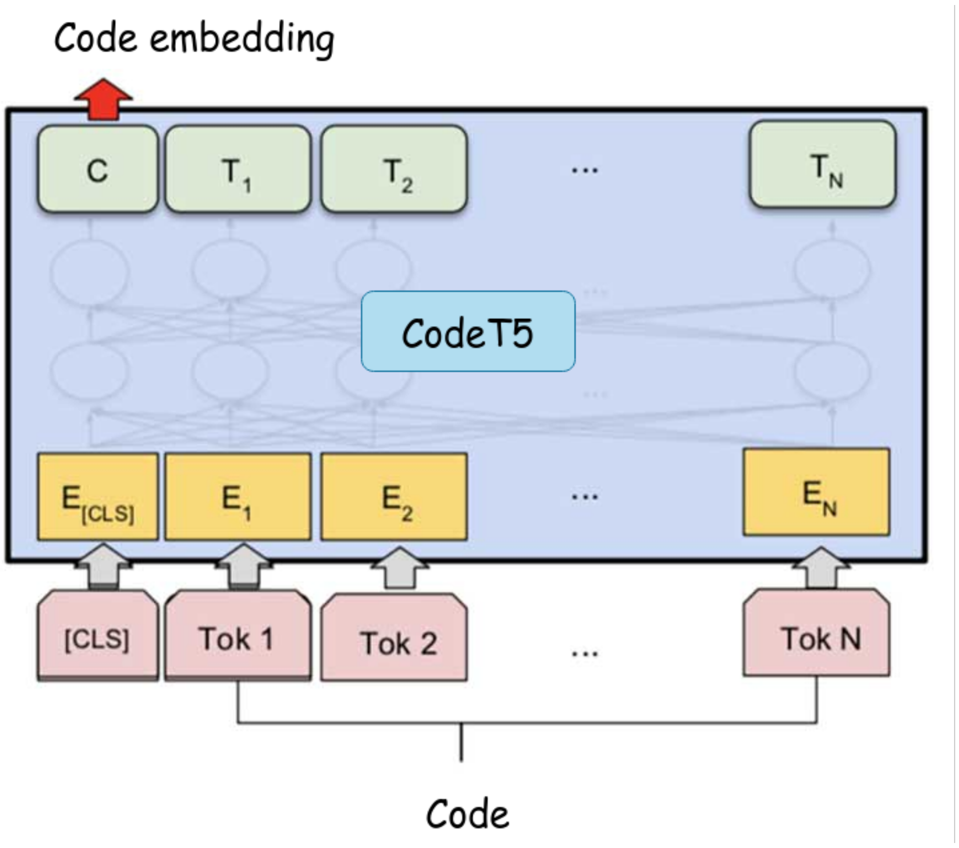


Figure 3.3: Working of the proposed modified CodeT5 architecture

neural network which outputs a similarity score. The similarity score is then used for ranking the codes during clone retrieval.

Could we reduce the number of times the neural network architecture is used during inference for code retrieval? Yes, in the next section, we discuss how the architectural design can be modified to reduce its usage from $O(n^2)$ to only $O(n)$ during code retrieval for n queries.

3.4 Proposed modified CodeT5 architecture

Our modified CodeT5 architecture is illustrated in Fig. 3.3. Instead of taking two codes as input to the model as described in the original architecture, we generate embedding for each code individually. For the clone detection task, we compute raw embeddings for both the code separately and then experiment with two different training strategies to learn these embeddings. Once the embeddings are trained, we compute and store them for every code in the BCB dataset.

Our idea is to use the code embeddings for comparison during clone retrieval rather than using the whole architecture. This proposal is based on the assumption that the embeddings capture syntactic and semantic information present in the program. We conduct experiments to assess the efficacy of these embeddings in Chapter 4. If the assumption holds true, then the clone retrieval task reduces to comparing these embeddings, which is faster and more efficient than using architecture for pairwise comparison.

3.5 Training Procedure

In this section, we describe two different training strategies that we use for learning the parameters of the modified CodeT5 architecture. One training objective is to minimize the cosine distance between the embeddings of code clones and maximize the distance between the embeddings of non-clones, referred to as the Cosine Embedding loss. Another training objective is to compute the Binary Cross Entropy loss in order to perform clone detection and automatically learn code embeddings during this process. These two strategies are described in the following subsections.

Note that we fine-tune the model instead of training it from scratch. Fine-tuning is a training strategy wherein we take a model already pre-trained on a certain task and apply it to our task. The idea is that the pre-trained model already understands the patterns in the data. Fine-tuning a pre-trained model allows for generating better results while saving a significant amount of training time and cost.

3.5.1 Cosine Embedding loss

In this training strategy, we aim to bring the embeddings of the true clone pairs closer to the common embedding space and simultaneously push the embeddings of non-clone pairs farther from each other. We use a cosine similarity metric to determine the level of similarity between each code pair. Using the cosine similarity, we define a cosine embedding loss and plan to minimize the loss by updating the model parameters. In this way, the model learns to project the code into an embedding vector rich in code syntax and semantics.

Cosine similarity. Cosine similarity represents the level of similarity between two vectors. In our case, we project two codes into embedding vectors using the architecture and compute the cosine similarity. If the two vectors point in the same direction they have a high level of similarity and vice versa. The cosine similarity is essentially the dot product of both vectors normalized by the product of their magnitude. The cosine similarity outputs a floating point value in the range [0, 1], where the value 0 (1 resp.) indicates the highest level of dissimilarity (similarity resp.) between vectors. Given two vectors A and B, the cosine similarity is defined by equation 3.6.

$$Cosine\ Similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||} \tag{3.6}$$

Cosine Embedding (CE) loss. The CE loss is derived from the cosine similarity. This loss is used to measure whether the two vectors are similar or dissimilar. If the two vectors of true (false resp.) clone pairs are predicted dissimilar (similar resp.), then the loss penalizes the model parameters. In other words, if the prediction and the actual label do not match the loss function takes a high value. The CE loss for each sample is defined in the equation 3.7.

$$l_{CE}(x_1, x_2, y) = \begin{cases} 1 - \cos(x_1, x_2) & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \gamma), & \text{otherwise} \end{cases} \tag{3.7}$$

where (x_1, x_2) are two embedding vectors, y is the true label and γ is the margin which is a tunable hyperparameter. We use the default value of the margin ($\gamma = 0$). The CE loss encourages the cosine angle between the embedding vectors to be small if both vectors are similar [62].

Fine-tuning CodeT5 parameters on CE loss. Fig. 3.4 depicts how the architecture is fine-tuned by minimizing the CE loss. We instantiate a single pre-trained CodeT5 architecture and input code-1 and code-2 tokens to extract their embeddings individually. Thus, there is parameter sharing (single CodeT5 instance) as illustrated in the figure. Once, the code embeddings are generated, the CE loss is computed and the model parameters are updated using gradient descent. The gradient descent [54] is an optimization algorithm to adjust the model parameters in order to minimize the loss function.

Inference on clone detection. During clone detection inference, we input the code tokens to fine-tuned architecture. Specifically, we input code-1 and code-2 tokens individually to the model to extract their embeddings. The cosine similarity between the embeddings is computed which determines the level of similarity between the code pairs. Finally, we compare the similarity score with a certain threshold. If the similarity lies

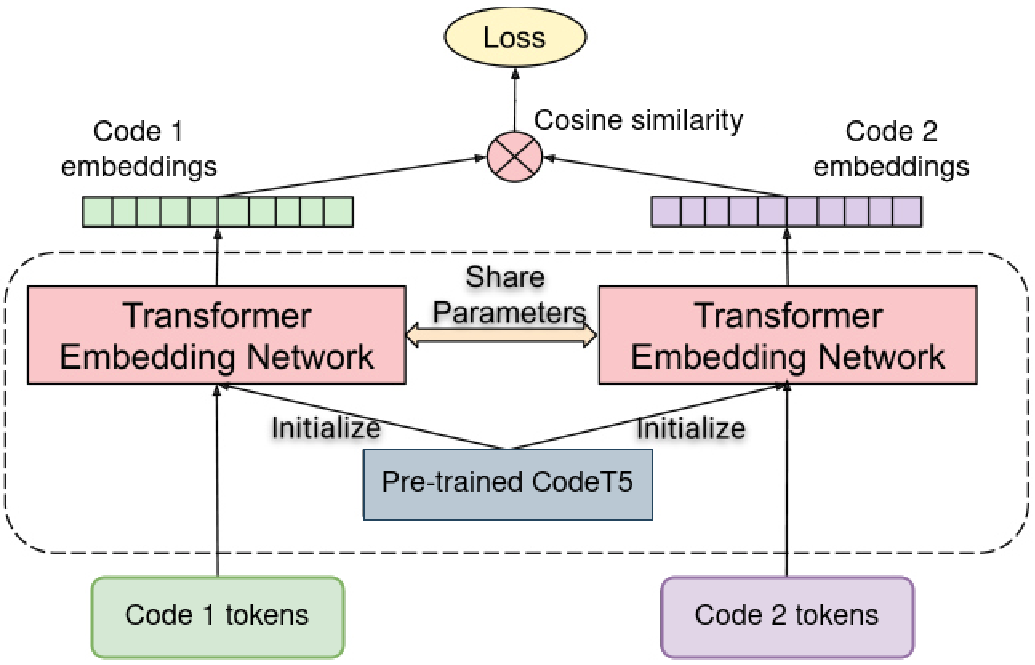


Figure 3.4: Learning code embeddings by minimizing cosine embedding loss

above the threshold, the code pairs are claimed to be true clones and vice versa.

Inference on clone retrieval. Once the architecture is fine-tuned, we pre-compute and store the embeddings corresponding to all the codes within the dataset. During clone retrieval inference, the input is a single query code. We use the architecture only once to extract the query code embeddings. We then compute the cosine similarity between the query code embeddings and all the pre-computed embeddings. The codes in the dataset are ranked according to their similarity with the query code. Finally, we choose and retrieve the $top - k$ codes as the clone pairs.

3.5.2 Binary Cross Entropy loss

In this strategy, we train the architecture to perform clone detection and automatically learn the parameters to generate the code embeddings during the process. This strategy differs from the previous conceptually and also by the way of computing loss on the embedding vectors. Let's first define and state the characteristics of the loss function.

Binary Cross Entropy (BCE) loss. It is a common loss function used in binary classification problems. In our case, given a code pair, we aim to predict whether they are true clone pairs or not. The BCE loss penalizes the model by returning a high value

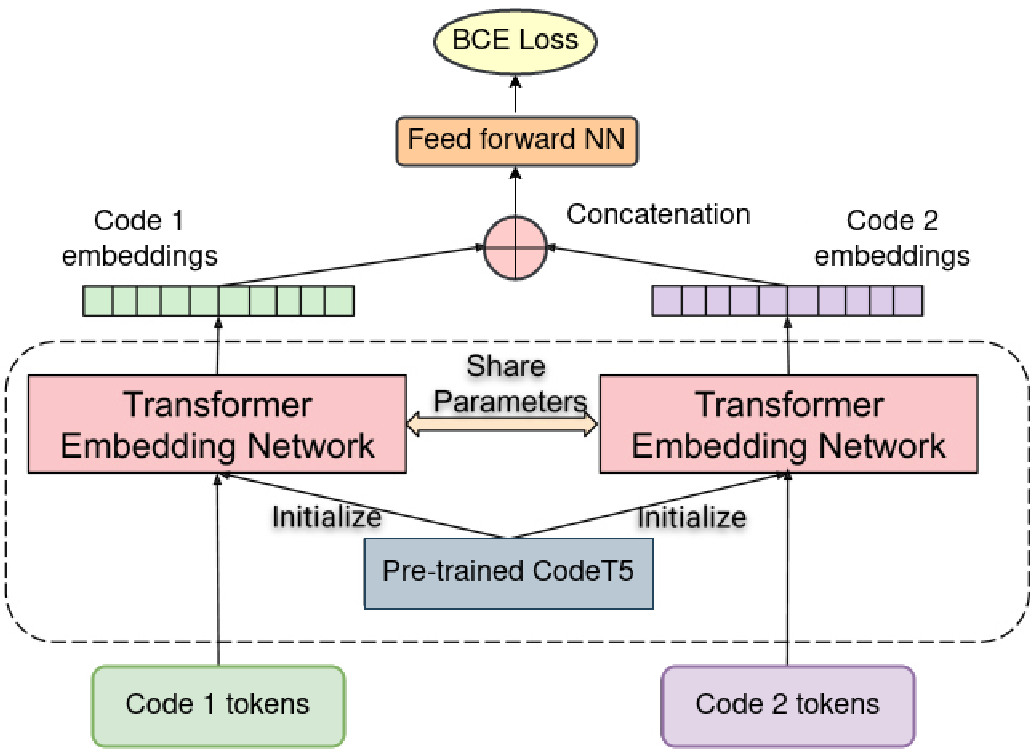


Figure 3.5: Learning code embeddings by minimizing binary cross entropy loss

for every wrong prediction. It can be represented by equation 3.8.

$$l_{BCE}(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}[\, y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)\,] \tag{3.8}$$

where N is the total number of samples, y is the true label ('1' if true clone pair otherwise '0'), and $\hat{y}$ is the predicted label. We train the model to minimize the loss and automatically learn the parameters to produce code embeddings.

Fine-tuning CodeT5 parameters on BCE loss. Fig. 3.5 illustrates how the architecture is fine-tuned by minimizing the BCE loss. The whole process is similar to fine-tuning the architecture on CE loss till the point the embeddings are extracted individually. After that, the code embeddings are concatenated horizontally and given to a feedforward neural network (classifier) for classification. The output of the classifier is a probability indicating the level of similarity between the code pair.

Feedforward neural network. The original CodeT5 architecture uses a 2-layered feedforward neural network taking as input a single embedding of 768 dimensions. Similar to the original CodeT5 model, we design a feedforward neural network that consists of two dense layers with non-linear functions following each dense layer. The only difference between the original CodeT5 feedforward neural network with ours is the number of input dimensions. We have input as a 1,536-dimensional vector as we concatenate code-1 and code-2 embeddings each having a 768-dimensional vector. The 2-layer classifier architecture is presented in Table 3.2. The sigmoid function takes as input raw scores (logits) and converts them into probability between [0, 1]. The classifier takes as input concatenated code embeddings and computes the weighted average over the embeddings, followed by the non-linear activation functions (Tanh or Sigmoid). The total number of trainable parameters in the classifier is 1,181,954. Note that the architecture including the classifier network is trained in an end-to-end manner.

Table 3.2: Layers in the Feedforward Neural Network

Layer	**Input shape**	**Output Shape**	**# Parameters**
Linear-1	[1, 1536]	[1, 768]	1,180,416
Tanh	[1, 768]	[1, 768]	-
Linear-2	[1, 768]	[1, 1]	1,538
Sigmoid	[1, 1]	[1, 1]	-

Inference on clone detection. During clone detection inference, we use the CodeT5 architecture to generate embeddings corresponding to both the codes individually. Further, our feedforward neural network predicts a probability indicating the level of similarity between the codes. If the probability is greater than a threshold ($p \geq 0.5$), then we classify the code pair as true clones.

Inference on clone retrieval. We pre-compute and store the embeddings using the fine-tuned architecture on the codes present in the entire dataset. During inference, we fetch the query code embeddings using the architecture. We experiment with the nearest neighbor search algorithm (KNN) using the euclidean distance as the comparison metric. The codes are now ranked according to their distances with the query code embeddings and the $top - k$ codes are retrieved as the clone pairs.

Chapter 4

Experiments and Results

This chapter describes the experiments conducted to assess the performance of the modified CodeT5 architecture for both clone detection and retrieval. The results obtained from each experiment are analyzed and discussed.

4.1 Experimental Setup

In the following subsection, we describe the specific implementation details and choices for reproducing our work on the BCB dataset. In subsection 4.1.2, we introduce the evaluation metrics used for assessing the performance of our system on both clone detection and retrieval task.

4.1.1 Implementation details

Our work is carried out in Python programming language (version 3.7.12). In particular, we use the PyTorch deep learning framework (version 1.11.0) for model implementation, training, and optimization. We use the CodeT5 implementation[1] provided in the HuggingFace transformers library (version 4.18). The model is fine-tuned for 2 epochs on both training strategies — Cosine Embedding loss (CE) and Binary Cross Entropy (BCE) loss. We use the Kaggle[2] platform for training the models which provide NVIDIA Tesla P100 GPUs for parallel computation. Training the model for 2 epochs took between 40-45 hours on the P100 GPU accelerators.

We used the AdamW [40] as the optimization algorithm with a learning rate of $5e-5$

[1]https://huggingface.co/Salesforce/codet5-small
[2]https://www.kaggle.com/

and epsilon value of $1e-8$, as given in the CodeT5 implementation[3]. The training and validation batch size was kept as 16 and 128 respectively. The maximum token length of codes was kept at 128 to retain maximum information as well as stay within the computational limits of Kaggle.

4.1.2 Evaluation Metrics

The clone detection and retrieval systems are evaluated against a set of standard metrics identified in the literature [21, 22, 52, 57]. These metrics help determine the efficiency of the algorithm and allow the users to choose the system that best fits their use case. The metrics we use in our work for clone detection and retrieval are described below.

Clone Detection

For clone detection, we use precision, recall, and f1-score as the metrics for comparing the performance of the existing works with ours. These metrics are defined below:

1. **Precision**

Precision is a standard classification metric that determines "What fraction of total clone pairs predicted by the model are actually true clone pairs?" The true clone pairs identified by the model are referred to as True Positives (TP) and the falsely predicted clone pairs are referred to as False Positives (FP). Therefore, the precision can be represented by the equation 4.1.

$$Precision\ (P) = \frac{TP}{TP + FP} \tag{4.1}$$

The value of precision varies in the range $[0, 1]$. A precision of 1 indicates that there are no false positives predicted.

2. **Recall**

Recall determines "What fraction of actual clone pairs are accurately predicted by the system as true clone pairs. The number of true clone pairs correctly identified by the model is called True Positives (TP) and the true clone pairs incorrectly classified by the model are called False Negatives (FN). Hence, the recall is represented by the equation 4.2.

[3]https://github.com/salesforce/CodeT5

$$Recall\ (R) = \frac{TP}{TP + FN} \tag{4.2}$$

Recall takes values in the range $[0, 1]$, where a value of 1 indicates all of the true clone pairs correctly identified as clones by the model and vice versa. Note that a trivial classifier can have a recall of 1 by simply classifying every code pair as a true clone.

3. F1 score

F1 score is a performance metric that is a combination of precision and recall. Comparing two or more classifiers using precision and recall is difficult. Assume if classifier A has high precision but classifier B has a high recall, then it is difficult to determine which classifier performs the best. Due to this reason, the F1 score was designed as a single metric to compare the performance of two or more classifiers. It is defined as the harmonic mean of precision and recall and represented by the equation 4.3.

$$F1\ score = \frac{2 \times P \times R}{P + R} \tag{4.3}$$

F1 score takes value in the range $[0, 1]$, where a value of 1 indicates the best possible classifier. A high F1 score in clone detection indicates a model that is able to distinguish between positive and negative clone pairs.

Clone Retrieval

Clone retrieval is the task of fetching clones from a repository of codes, given a query code. For clone retrieval, we assess the performance of the systems against two simple metrics: time and accuracy. The specific details of how we calculate the time and accuracy of the algorithm are defined below.

1. Execution time

We calculate the total time required for the system to answer n queries. In other words, for each query, we start the timer the moment the system parses query code and end when the system retrieves k potential clone pairs. We now sum the total time required for answering all n queries. Ideally, we want the system to consume minimum time to retrieve the clone pairs.

2. Accuracy

The accuracy is calculated for n queries by diving the number of correct clone retrieval by the total number of queries. For every query, we consider an accurate clone retrieval, if the system retrieves a true clone pair in the top k retrieved codes.

4.2 Results

In this section, we describe the experiments conducted and present the results. We perform three experiments — Experiment 1 illustrates our results on the clone detection task using our modified neural network architecture and presents a comparison with the existing works. In experiment 2, we visualize the code embeddings and identify the functionalities in the data captured by the embeddings. In experiment 3, we perform clone retrieval and assess the performance of our proposed solution.

4.2.1 Experiment 1

We divide the experiment into two parts. First, we present and compare the results of two of our proposed training strategies — CE and BCE loss. Second, we compare the results of our best model with the existing works against the clone detection metrics defined in the previous subsection.

Experiment 1.1

Table 4.1 depicts the results of our modified CodeT5 architecture fine-tuned on two training strategies along with the original CodeT5 results on the clone detection task. The first row in the table represents the results of the original CodeT5 architecture (with concatenated code tokens as described in Section 3.3) on the clone detection task. The second row represents the original architecture directly used without any fine-tuning on single code input. The third and the fourth row represent the modified CodeT5 architecture fine-tuned on CE and BCE losses respectively.

As we can see from Table 4.1, the original CodeT5 performs the best on all metrics, followed by our modified CodeT5 architecture fine-tuned on BCE loss. One reason why our fine-tuned CodeT5 architecture lags behind the original CodeT5 architecture can be attributed to the fact that the CodeT5 authors use a maximum code token length of 400. However, we truncate the maximum code token length to 128 to stay within our computational limits, thus losing a lot of useful information. Note that the original CodeT5 results were taken from their Github[4] repository and not from their work [69], as they

[4]https://github.com/salesforce/CodeT5/issues/55#issuecomment-1178517051

Table 4.1: Results of original and fine-tuned modified CodeT5 architecture on the clone detection task. Figures in **bold** (resp. underline) refers to the best (resp. second best) value on the metric.

Architecture	**Precision**	**Recall**	**F1 score**
Original CodeT5	**0.9526**	**0.9474**	**0.9500**
Original CodeT5 (Single Code input)	0.2076	0.2519	0.2276
Our training strategies			
Fine-tuned CodeT5 (CE loss)	0.6242	0.9346	0.7485
Fine-tuned CodeT5 (BCE loss)	0.8972	0.9074	0.9023

claimed to have reported incorrect result in the paper.

We can see in the second row that if single code tokens are given as input to the original CodeT5 architecture without fine-tuning, we get very poor results on the task. This is expected because the original CodeT5 is pre-trained on concatenated code-1 and code-2 tokens. It can be clearly observed in the table that fine-tuning modified CodeT5 using BCE loss achieves a higher F1-score (0.9023) than using CE loss (0.7485). Thus, we can conclude that fine-tuning modified CodeT5 by minimizing BCE loss tends to provide better code embeddings than CE loss. To verify this claim, we perform Experiment-2 in which we visualize the code embeddings and generate meaningful interpretations.

Experiment 1.2

Table 4.2 compares the results of some of the notable works against our best result on the clone detection task. All the results are generated on the BCB testing dataset.

The results of the existing works are arranged in ascending order of F1 scores. We can observe that the classical clone detection approaches using ASTs such as RtvNN [72] and Deckard [22] have high precision but low recall values and thus a lower F1-score. Works such as CDLH [71] and ASTNN [75] which model ASTs using classical deep learning architectures (such as RNNs and LSTMs) obtain better precision and recall scores than RtvNN and Deckard.

Table 4.2: Results on code clone detection. Figures in **bold** (resp. underline) represents the best (resp. second best) score on the metric

Architecture	**Precision**	**Recall**	**F1 score**
RtvNN [72]	0.95	0.01	0.01
Deckard [22]	0.93	0.02	0.03
CDLH [71]	0.92	0.74	0.82
ASTNN [75]	0.92	0.94	0.93
RoBERTa [15]	0.949	0.922	0.935
CodeBERT [13]	0.947	0.934	0.941
FA-AST [68]	**0.96**	0.94	**0.95**
GraphCodeBERT [15]	0.948	**0.952**	**0.95**
CodeT5 [70]	0.952	0.947	**0.95**
Ours	0.897	0.907	0.902

Recent deep learning architectures such as RoBERTa [15], CodeBERT [13], GraphCodeBERT [15], CodeT5 [69] and FA-AST [68] compete to attain top results on the task. Three architectures — FA-AST, GraphCodeBERT, and CodeT5, currently achieve state-of-the-art F1 scores (0.95) on the clone detection task. Our best model which is the modified CodeT5 architecture fine-tuned on BCE loss obtains an F1 score of 0.902. The main reason why our best model drops in the F1 score is because we use a maximum code token length of 128 as opposed to all the other architectures which use a maximum code sequence length greater than 400. Our focus was never to achieve top results in clone detection but to assess whether the CodeT5 embeddings are capable of capturing essential semantic and syntactic code information. Our secondary aim is to identify whether the code embeddings could be useful for clone detection and retrieval. Thus, we can conclude that our best model achieves a considerable performance on the clone detection task.

4.2.2 Experiment 2

In this experiment, we aim to verify whether the code embeddings extracted from the fine-tuned CodeT5 architecture capture several functionalities present in the code corpus. In turn, this will confirm that CodeT5 is able to comprehend semantic and syntactic information from the programs. We will now explain the steps we take to verify this assumption.

Firstly, we run the fine-tuned modified architecture using BCE loss on all the code snippets present in the dataset and extract corresponding code embeddings. Each code embeddings are 768-dimensional continuous vectors representing useful code-specific information. Once, the code embeddings are extracted, we apply a popular dimensionality reduction technique, T-SNE, to reduce the 768D vector into 2D. This is done so that we can visualize the code embeddings in a 2D cartesian plane and derive meaningful interpretations. We explain the T-SNE algorithm briefly below and then visualize the code embeddings in 2D.

T-SNE. T-distributed Stochastic Neighbor Embeddings [65] is a dimensionality reduction technique, majorly used to visualize high-dimensional data in 2D or 3D. Let's briefly understand how it works. The algorithm randomly initializes 2D embeddings for all high-dimensional data points. It then transforms the similarities between the high-dimensional data and low-dimensional data points to joint probabilities and minimizes the KL divergence (distance) between the two probability distributions. This way the low-dimensional embeddings are trained to reflect/mimic the similarity present in the high-dimensional data points. The KL divergence between two probability distributions $P(x)$ and $Q(x)$ belonging to χ probability space data points is given in the equation 4.4.

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \cdot \log\left(\frac{P(x)}{Q(x)}\right) \tag{4.4}$$

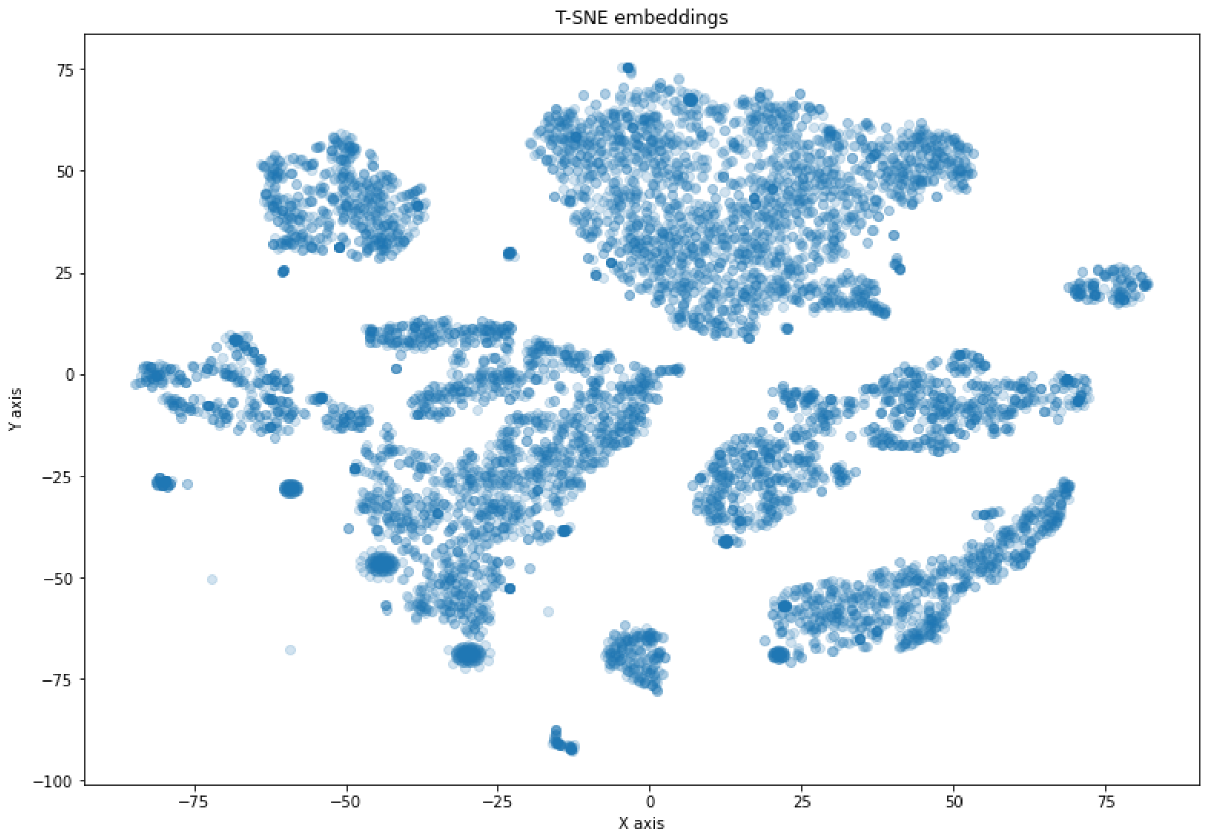


Figure 4.1: Visualizing code embeddings in the 2D cartesian plane. Each blue dot represents a distinct code fragment in the corpus

Visualizing 2D code embeddings

Fig. 4.1 illustrates the 768D code embeddings plotted in 2D by reducing the dimensionality using the T-SNE algorithm. Every blue dot in the figure represents a unique code fragment in the BCB dataset. We can see that the codes can be grouped into a certain number of categories. The codes within a category might belong to similar functionalities. If this is the case, then our assumption regarding code embeddings capturing the code-specific details holds true. Since there exists a distinct separation between the groups, we will now try grouping the code embeddings into k clusters to gather more information about what each cluster represents. We experiment with two different clustering algorithms — KMeans and DBSCAN.

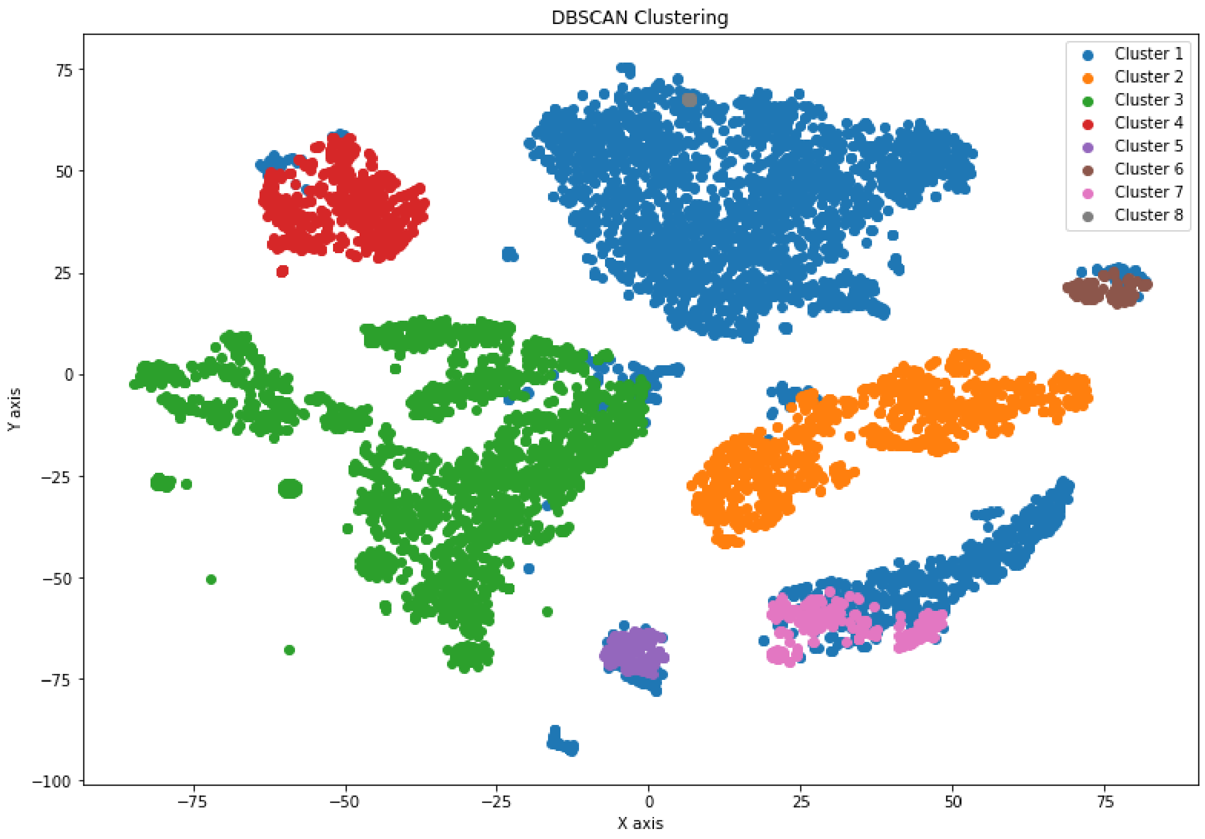


Figure 4.2: Clusters identified in the code embeddings by DBSCAN algorithm

DBSCAN. Density-Based Spatial Clustering of Applications with Noise [11] is a clustering algorithm that automatically determines the optimal number of clusters. It can is known to discover clusters possessing an arbitrary shape. The algorithm finds core data points present in a high-density region and expands clusters from them. We apply the DBSCAN algorithm to the extracted 768D code embeddings to identify inherent clusters present in the code corpus.

We can observe from Fig. 4.2 that the number of clusters identified by the DBSCAN algorithm is suboptimal. There are eight clusters identified by the algorithm. However, some of the clusters overlap each other and are not distinctly separated. Therefore, we pivot to the KMeans clustering algorithm with the hope to get optimal clustering to identify the functionalities encoded in the clustered code embeddings.

KMeans clustering. KMeans clustering is an iterative algorithm that divides n observations into k distinct clusters (categories). The algorithm begins with initializing k centroids that act as representative of each cluster. The algorithm works in two stages — (i) Cluster assignment and (ii) Centroid movement. After centroid initialization, the algorithm finds the nearest centroid for each data point and assigns it the corresponding cluster. Once all data points are assigned a cluster, the centroids of each cluster are translated to the mean of the coordinates of all data points contained within that cluster. This process is repeated till the algorithm is converged.

We apply KMeans clustering on the 768D code embeddings to group them into categories with the intention of understanding what each category represents. The optimal number of clusters k is determined using the elbow-curve method. Fig. 4.3 shows the elbow curve plotted for varying values of k against two distance metrics — inertia and distortion. Inertia is defined as the sum of squared distances from every data point to its nearest centroid. While distortion is the average of squared distances from every cluster centroid to all points contained within the centroid.

Choosing K in KMeans clustering. From Fig. 4.3 we determine the optimal value of number of clusters k. Typically, we find the point which acts as an elbow, meaning that, the point after which the curve is a straight line. We can observe from both subfigures that after $k = 7$, the curves flatten, indicating the optimal cluster value. Therefore, we choose $k = 7$ and apply the KMeans clustering algorithm to the extracted code embeddings.

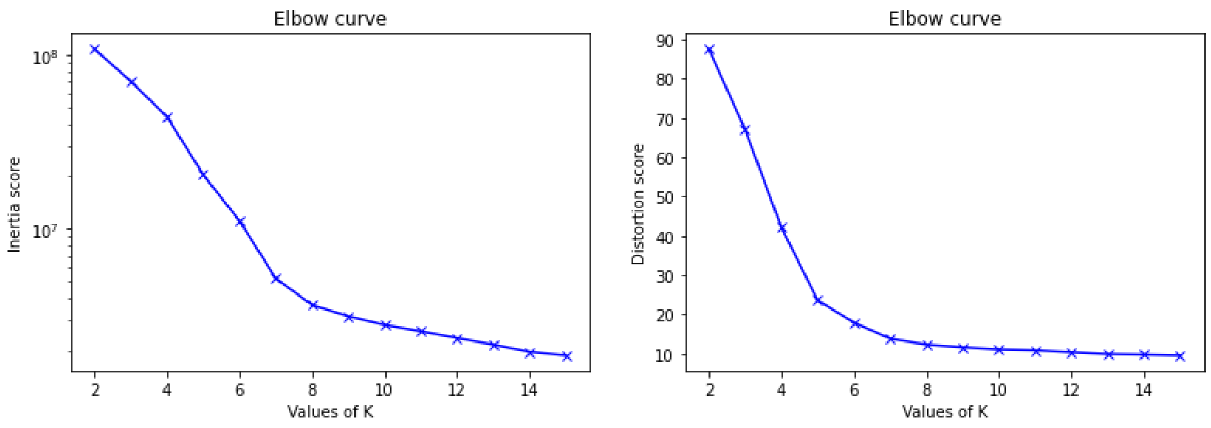


Figure 4.3: Determining the optimal number of clusters in KMeans using elbow curve

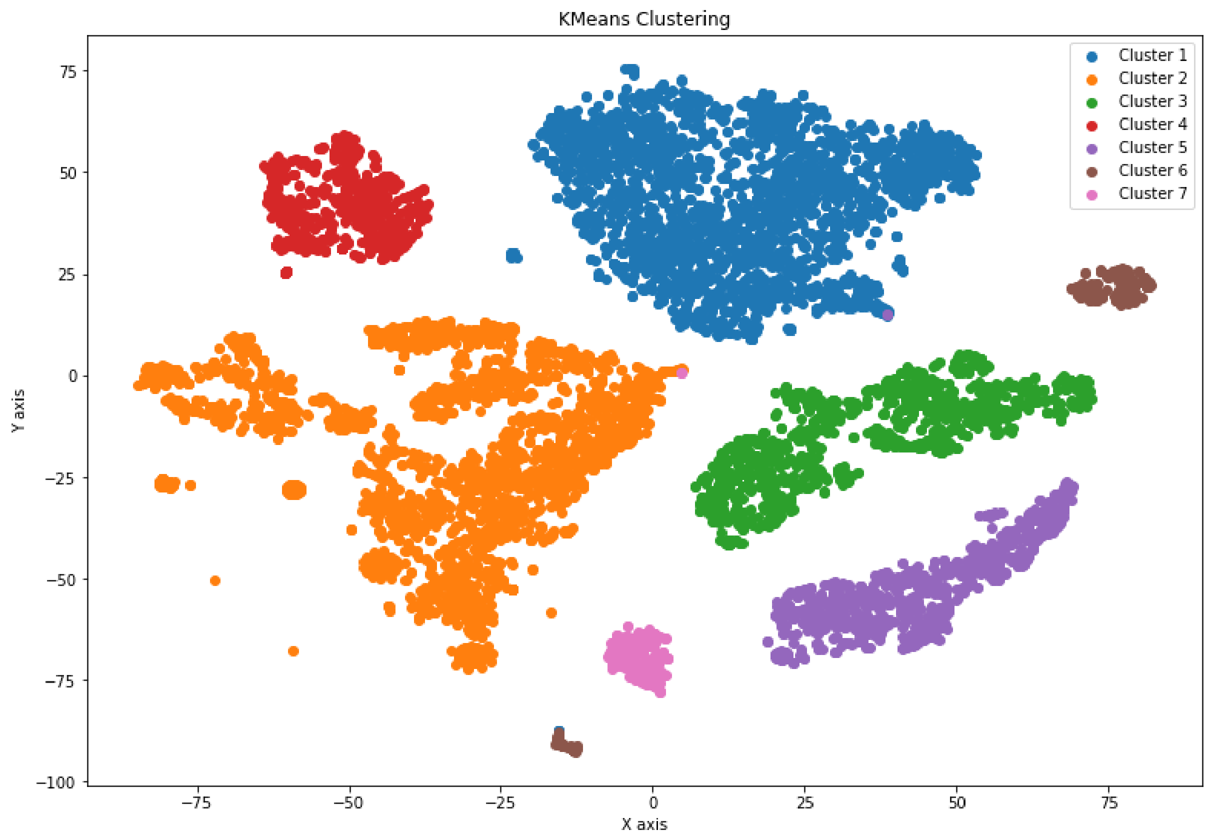


Figure 4.4: Clusters identified in the code embeddings by KMeans clustering algorithm

Visualizing clusters identified using KMeans. Fig. 4.4 depicts the seven clusters identified by the algorithm. The algorithm seems to have We can spot some misclassification such as the points at the bottom center of the figure are considered by KMeans to be within cluster 6 (brown), however, the points seem to be closer to cluster 7 (pink). Nevertheless, we can see that KMeans clustering does a better job at identifying distinct clusters than the DBSCAN algorithm. We are now interested in finding out what each cluster represents. The next subsection talks about how we figure out the functionalities encoded by data points within each cluster.

Visualizing functionalities encoded by each cluster

For visualizing the functionalities encoded, we consider all the codes within each cluster, identified by the KMeans algorithm. For each cluster, we extract the 50 most frequent words in the codes encompassed within the cluster. By visualizing the top frequent words, we can get an idea of the characteristics of the codes in every cluster.

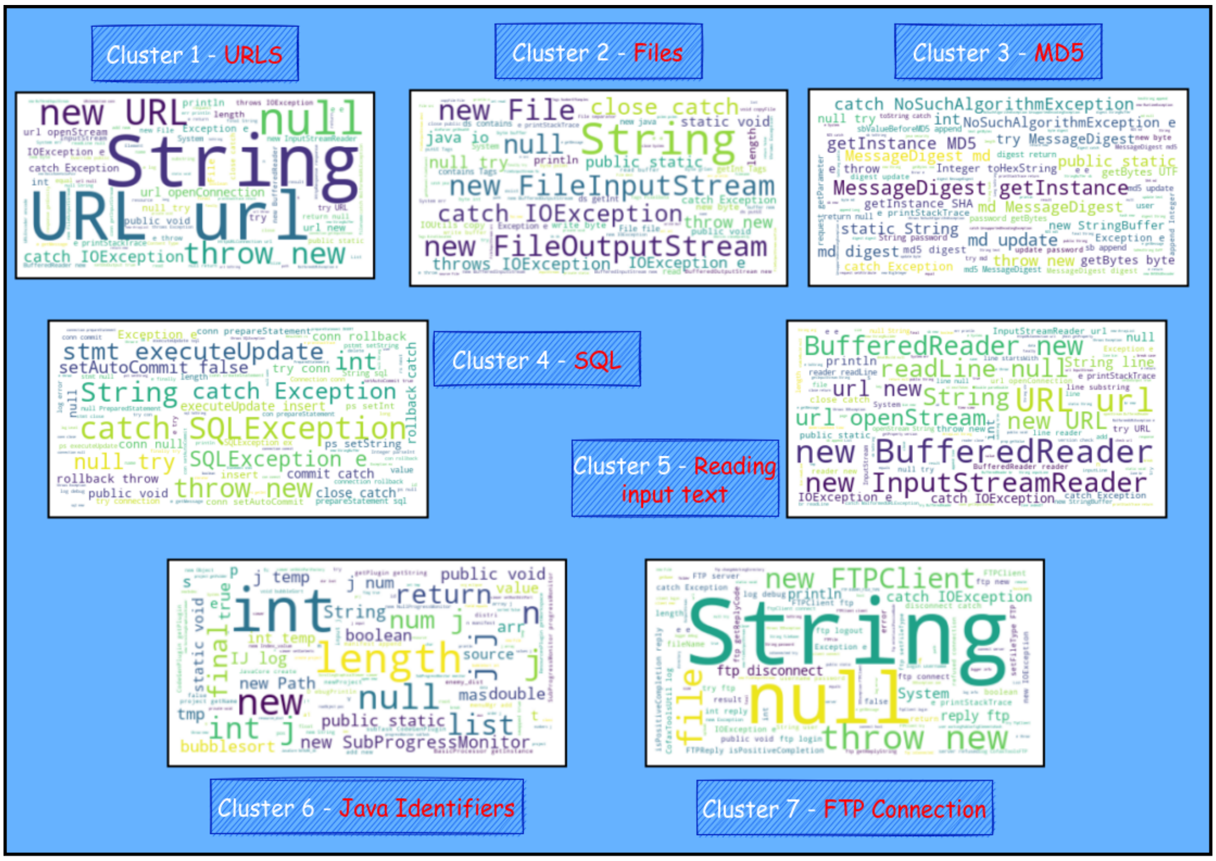

Cluster 1 - URLS
Cluster 2 - Files
Cluster 3 - MD5
Cluster 4 - SQL
Cluster 5 - Reading input text
Cluster 6 - Java Identifiers
Cluster 7 - FTP Connection

Figure 4.5: Functionalities encoded within the seven clusters identified by KMeans algorithm

Fig. 4.5 illustrates a word cloud depicting the top 50 words present in each of the seven clusters. We can see that each cluster represents a unique functionality. The codes within the cluster can be considered slight variations of that functionality and hence can be categorized as clones. The functionalities represented by each cluster are:

1. **Cluster 1 (URLS)** - It can be seen from the figure that a few top words in this cluster are [”URL”, ”openconnection”, ”new URL”, ”String”]. Thus, we can conclude this cluster contains codes related to opening an URL connection, perhaps, fetching a resource from the URL, and closing the connection.

2. **Cluster 2 (Files)** - The top words identified in the cluster are [”File”, ”String”, ”FileInputStream”, ”FileOutputSteam”, ”contains”, ”try”, ”catch”, ”java io”, ”copyFile”]. Hence, we can conclude that this cluster contains codes associated with reading file inputs and producing a file output. It also contains codes for copying the contents of one file to another.

3. **Cluster 3 (MD5)** - The top words identified in this cluster are [”MessageDigest”,

"MD5", "SHA", "NoSuchAlgorithmException", "String password", "update password"]. Looking at the words we can comment that this cluster encompasses codes that are responsible for securely hashing and storing user passwords. MD5 is a cryptographic hashing algorithm popularly used to convert variable-length string input to a fixed 128-length hashed output.

4. **Cluster 4 (SQL)** - The top 50 words represented by codes in this cluster include ["SQLException", "SQL rollback", "insert", "executeUpdate", "SetAutoCommit false"]. We can conclude that this cluster contains codes executing different SQL queries.

5. **Cluster 5 (Reading input text)** - The top words identified in this cluster are ["new BufferedReader", "new InputStream", "readLine", "substring", "String line"]. We can comment that this cluster encodes the program that reads and stores the stream of text in the buffer. Then, applies various functionalities to the text such as getting a substring from the text.

6. **Cluster 6 (Java identifiers)** - Some of the top words present in this cluster are ["int", "final", "static", "public", "void", "list"]. Looking at the words we can claim that this cluster represents the codes carrying out simple java operations. Most of the top words identified are keywords or identifiers present in the java programming language.

7. **Cluster 7 (FTP connection)** - The top words identified in this cluster are ["new FTPClient", "FTP disconnect", "FTP Server", "FTPReply isPositiveCompletion", "FTP login"]. By observing the words, we can tell that this cluster encompasses codes containing functions to establish FTP (File Transfer Protocol) connection, transfer files and data, and then close the connection.

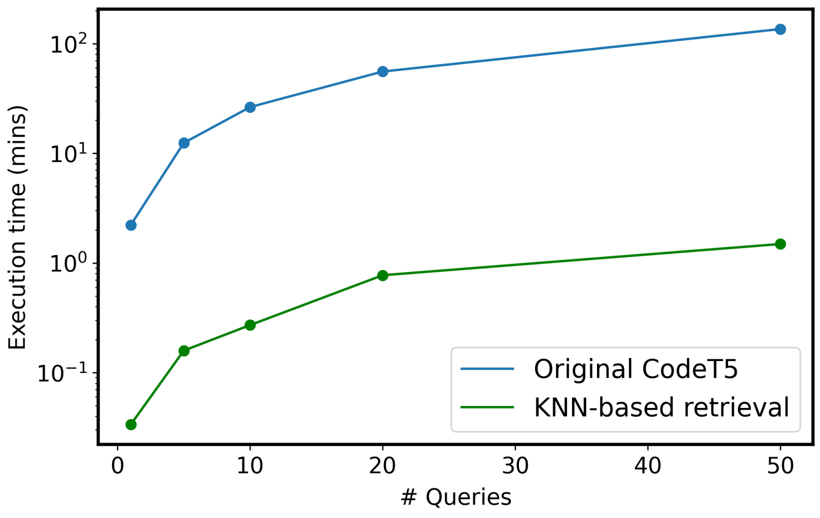


Figure 4.6: Total time taken to answer n queries

4.2.3 Experiment 3

In this experiment, we are interested in finding out the total time required by both the original CodeT5 and our proposed KNN-based retrieval system to answer n queries. Further, we compare through visualization how accurate both systems are at answering the queries.

KNN-based retrieval. We pre-compute and store the code embeddings of the existing codes in the dataset. In the BCB testing dataset, we have the code pairs as input. We consider the first code to be the query code. Now, we use our fine-tuned architecture to extract the query code embeddings and employ KNN to compare the query code embeddings with existing ones and retrieve the $top - k$ best matching code clones. We choose $k = 20$, thus retrieving 20 codes closest to the query code. In most websites, we observe somewhere between 10-25 elements being retrieved for a search query and hence the choice of the k.

Execution time

Fig. 4.6 illustrates the time-based comparison of the original CodeT5 architecture and our proposed solution. We can see that the original architecture takes a great deal of time as the number of queries increases, in the order of 10^2 minutes. However, the proposed KNN-based retrieval system takes around 1 minute to address 50 queries. Thus we can conclude that the proposed KNN-based retrieval system is scalable.

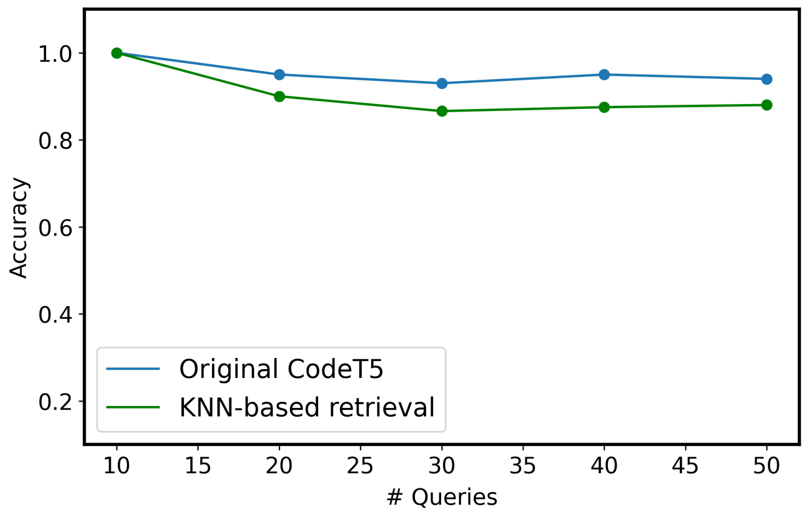


Figure 4.7: Accuracy of the systems in response to n queries

For every query, the original CodeT5 architecture runs through all the existing code in the dataset, concatenates the query code tokens with the existing codes one by one, and produces a similarity score. Further, the codes in the dataset are ranked in descending order according to their similarity score. Since the architecture runs for n times (n represents total codes in the corpus) for each query it ends up consuming a lot of time. However, in our case, we run the fine-tuned architecture once for every query to extract its embeddings and then use the KNN algorithm for comparing the embeddings.

Accuracy

Fig. 4.7 depicts the accuracy of both systems in response to n queries. We define accuracy as a fraction of total queries containing the true clone pair in the top k retrieved clone pairs by the system. We can observe from the figure that the performance of both systems is comparable. There exists a drop in the accuracy of our proposed KNN-based retrieval system when compared to the original CodeT5 model. This drop can be attributed to two factors — (i) We truncate the maximum code token length to 128 which is way below the 400 token limit used in the original CodeT5; (ii) In the original CodeT5 architecture, the model gets both codes as inputs. This is beneficial as the model is able to capture the interrelation between the codes. However, in our case, the fine-tuned model generates embeddings by looking at the codes independently.

Note that for the inference the true clone pairs were picked randomly from the BCB testing dataset. The first code in the clone pair is considered the query code. From Fig.

4.7 it is visible that the average accuracy of our proposed KNN-based system is close to 0.9, whereas the average accuracy of the original CodeT5-based retrieval model is close to 0.96. Thus, we can conclude that our proposed KNN-based retrieval system has a comparable accuracy and can retrieve code clones in lesser time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this study, we propose to modify the architecture of CodeT5 in an attempt to reduce its usage during clone retrieval. CodeT5 is a recent neural network that achieves state-of-the-art results on a variety of code understanding and generation tasks, including clone detection. The CodeT5 architecture is not designed to perform clone retrieval due to the large number of pairwise code comparisons using the architecture. This brings us to the motivation for the work. We propose to modify the architecture to generate code embeddings for each code separately as opposed to their original way of taking both code pairs as input. Comparing two code embeddings is way faster than comparing two codes using the architecture. We make an assumption that the embeddings are able to retain useful semantic and syntactic language information.

We experiment with two different training strategies using Cosine Embedding loss and Binary Cross Entropy loss. We confirm through evaluation that fine-tuning the modified architecture using BCE loss results in better performance on the clone detection task as compared to CE loss. As a result, we used the fine-tuned model on BCE loss for carrying out further experiments. For clone detection, the code embeddings for both code pairs are concatenated and passed through a 2-layered feedforward neural network which outputs a similarity score. Finally, we compare the similarity score to a threshold and determine whether the codes are clones. Our proposed system achieves an F1 score of 0.902 on clone detection.

Next, we verify the assumption made about code embeddings by visualizing them and extracting meaningful interpretations. First, the embeddings are converted from high-dimensional space to low-dimensional space for visualization purposes. Further, KMeans algorithm is employed to group the embeddings into 7 clusters. We identify by visualizing through word clouds what each cluster represents. We conclude that the programs within a cluster encode minor variations of the same functionality. This confirms that the code embeddings are capable of capturing rich code-specific information.

For clone retrieval, we propose a nearest neighbor-based retrieval system that addresses queries in real-time. The proposed system takes 10^2 order of execution time (mins) lesser as compared to the original CodeT5 for responding to 50 queries while achieving comparable accuracy. Thus, we confirm that the code embeddings can be used for both clone detection and retrieval.

5.2 Limitations

There are a few limitations of our proposed approach for both clone detection and retrieval. In clone detection, we separately extract code embeddings for the code pairs using the modified CodeT5 architecture. As a result, the code embeddings are not able to capture the interrelation between the code pairs unlike in the original CodeT5 architecture, resulting in a loss of performance. In addition, due to constrained computation resources, we truncate the code tokens to a maximum length of 128 which might result in unreliable code embeddings for codes with token length greater than 128.

In clone retrieval, our proposed KNN-based retrieval computes euclidean distance across 768 dimensions. Although the computation takes a fraction of a second for a small number of queries, however, it will take a significant time given a million queries. Can we do better? Yes, perhaps by reducing the dimensionality while preserving the inter-similarity between code pairs as in the high-dimensions, we can gain in time.

5.3 Future Scope

In this work, we performed clone detection and retrieval on a BCB dataset consisting of 9,134 java programs. Hence, at the moment our proposed system works only for code fragments written in java. This could be extended to include a multilingual programming dataset. Most languages have similar syntactical structures, thus the model can better understand similar and dissimilar properties across various programming languages. There could be a unified model developed in the future, that would be able to perform multilingual clone detection and retrieval.

In the future, we can hope to witness more work leveraging code embeddings to perform code understanding and generation tasks. Code embeddings offer several advantages including reduced time for pairwise comparison and significant accuracy in retrieval. This could be the way forward to perform large-scale code clone detection and other code-specific tasks such as code summarization, code search, etc.

Another direction could be to perform knowledge distillation of huge neural network-based architectures to develop a smaller model that captures the same knowledge learned by the bigger network. This can help speed up both the clone detection and retrieval process. Several others works are focused on designing loss functions to better train the architecture. Future work could involve experimenting with a combination of both CE and BCE losses to train optimal code embeddings. There is a possibility of training code embeddings using negative sampling loss [42] which is used to generate word embeddings such as Word2Vec.

Many works have found data pre-processing to be a better solution than designing huge neural network-based architectures for code-related tasks. An extensive code pre-processing can result in even a smaller neural network better able to comprehend the code-specific properties and produce remarkable results. Code processing is still an active area of research and it would be interesting to witness how future works perform feature engineering to extract meaningful features from the programs.

Bibliography

[1] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[3] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[6] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[7] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*, pages 7–13, 2011.

[8] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–12. Citeseer, 2004.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*,

pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[10] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE, 1999.

[11] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[12] M. R. Farhadi, B. C. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi. Scalable code clone search for malware analysis. *Digital Investigation*, 15:46–60, 2015.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

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

[15] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[16] A. Gupta, B. Suri, and S. Misra. A systematic literature review: code bad smells in java source code. In *International Conference on Computational Science and Its Applications*, pages 665–682. Springer, 2017.

[17] M. Hammad, Ö. Babur, H. A. Basit, and M. van den Brand. Clone-advisor: recommending code tokens and clone methods with deep learning and information retrieval. *PeerJ Computer Science*, 7:e737, 2021.

[18] M. Hammad, Ö. Babur, H. A. Basit, and M. Van Den Brand. Clone-seeker: Effective code clone search using annotations. *IEEE Access*, 10:11696–11713, 2022.

[19] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A pdg-based approach. In *2011 18th Working Conference on Reverse Engineering*, pages 3–12. IEEE Computer Society, 2011.

[20] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *2010 IEEE International Conference on Software Maintenance*, pages 1–9. IEEE, 2010.

[21] Y. Jia, B. Binkley, M. Harman, J. Krinke, and M. Matsushita. A proposed approach to fast and precise clone detection. *Proceedings of IWSC*, 2009.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[23] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183, 1993.

[24] N. Juillerat and B. Hirsbrunner. An algorithm for detecting and removing clones in java code. In *Proceedings of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change, SeTra*, volume 2006, pages 63–74, 2006.

[25] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

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

[27] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[28] I. Keivanloo and J. Rilling. Source code clone search. In *Code Clone Analysis*, pages 121–134. Springer, 2021.

[29] I. Keivanloo, J. Rilling, and P. Charland. Internet-scale real-time code clone search via multi-level indexing. In *2011 18th Working Conference on Reverse Engineering*, pages 23–27. IEEE, 2011.

[30] I. Keivanloo, C. K. Roy, and J. Rilling. Towards source code clone search via information retrieval.

[31] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001.

[32] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 309–318. IEEE, 2012.

[33] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[34] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 167–176, 2010.

[35] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software*, 184:111141, 2022.

[36] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSdi*, volume 4, pages 289–302, 2004.

[37] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, 2006.

[38] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)*, 54(9):1–40, 2021.

[39] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *29th International Conference on Software Engineering (ICSE'07)*, pages 106–115. IEEE, 2007.

[40] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[41] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[43] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[44] J. Pennington, R. Socher, and C. Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[45] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[46] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[47] A. Prasad. *Code Clone Detection Using Code2Vec*. University of California, Irvine, 2020.

[48] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[49] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

[50] C. Ragkhitwetsagul and J. Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, 24(4):2236–2284, 2019.

[51] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[52] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[53] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[54] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[55] N. Saini, S. Singh, et al. Code clones: Detection and management. *Procedia computer science*, 132:718–727, 2018.

[56] H. Sajnani. *Large-scale code clone detection.* University of California, Irvine, 2016.

[57] H. Sajnani, V. Saini, and C. Lopes. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process*, 27(6):402–429, 2015.

[58] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[59] A. Sheneamer and J. Kalita. Semantic clone detection using machine learning. In *2016 15th IEEE international conference on machine learning and applications (ICMLA)*, pages 1024–1028. IEEE, 2016.

[60] A. Sheneamer and J. Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[61] G. Shobha, A. Rana, V. Kansal, and S. Tanwar. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security*, pages 645–655, 2021.

[62] S. Sudholt and G. A. Fink. Evaluating word string embeddings and loss functions for cnn-based word spotting. In *2017 14th iapr international conference on document analysis and recognition (icdar)*, volume 1, pages 493–498. IEEE, 2017.

[63] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE, 2014.

[64] R. Tairas and J. Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, 2009.

[65] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[66] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[67] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pages 128–135. IEEE, 2004.

[68] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[69] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[70] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[71] H. Wei and M. Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.

[72] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[73] W. Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

[74] Y. Yang, Z. Ren, X. Chen, and H. Jiang. Structural function based code clone detection using a new hybrid technique. In *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, volume 1, pages 286–291. IEEE, 2018.

[75] H. Zhang and K. Sakurai. A survey of software clone detection from security perspective. *IEEE Access*, 9:48157–48173, 2021.