A Comparative Study of Three Transformer-Based Models for Embedding-Powered XGBoost Code Clone Detection

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
May 2026
Lauri Talvitie

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Lauri Talvitie: A Comparative Study of Three Transformer-Based Models for Embedding-Powered XGBoost Code Clone Detection

Master of Science (Tech) Thesis, 57 p., 3 app. p.
Software Engineering
May 2026

Code cloning is a common yet potentially harmful practice in software development, which can degrade maintainability and increase the need for debugging. The first objective of this thesis is to investigate different types of code clones and the current approaches that are used to detect them. The second objective is to compare the performance of three different transformer-based models in detecting code clones. The study investigates how two smaller models specified on code-related tasks perform against larger general purpose Large Language Model.
The research methods included a literature review and method development. The literature review is used to gather foundation for the current state of code clone detection, including the clone types and different clone detection approaches. For the method development, a code clone detection pipeline is constructed, by utilizing CodeT5+, GraphCodeBERT, and Llama 3.2 1B in generating code embeddings that are furthermore used to train XGBoost binary classifier.
The results indicate that the code-specific models, CodeT5+ and GraphCodeBERT, perform significantly better than the larger general-purpose LLM Llama 3.2 1B model. The results show that pre-training data plays more crucial role in the model's performance than only the sheer size of the model.

Keywords: Code clone detection, transformer model, code embeddings

Contents

1 Introduction 1

1.1 Background and Motivation 1

1.2 Research Methods 2

1.3 Research Problem and Questions 3

1.4 Thesis Structure 4

2 Code Clone Taxonomy and Detection Approaches 5

2.1 Code Clone Types 6

2.1.1 Type-1 7

2.1.2 Type-2 7

2.1.3 Type-3 7

2.1.4 Type-4 8

2.2 Detection Methods 8

2.2.1 Text-based Approaches 9

2.2.2 Token-based Approaches 9

2.2.3 Tree-based Approaches 11

2.2.4 Graph-based Approaches 12

2.2.5 Deep-learning-based Approaches 13

2.2.6 Metric-based Approaches 14

3 Comparison of Three Transformer-based Models for Code Clone

Detection **16**

3.1 Models 17

3.1.1 Llama 3.2 17

3.1.2 CodeT5+ 18

3.1.3 GraphCodeBERT 19

3.2 Datasets 20

3.2.1 PoolC Dataset 20

3.2.2 CodeXGLUE-BigClone Dataset 21

3.2.3 BigCloneBench Toma Subset 22

3.3 System Architecture and Experimental Setup 22

3.3.1 Fine-tuning the Models 24

3.3.2 Generating Tokens and Embeddings from Source Code 28

3.3.3 Combining the Labeled Embedding Pairs and Training an XG-Boost Classifier 33

3.3.4 Comparing Performance Metrics between Models 35

4 Evaluation of Code Clone Detection Methods **38**

4.1 Impact of Fine-Tuning on the Performance of Code-Specific Variants 38

4.1.1 GraphCodeBERT Fine-Tuning 38

4.1.2 CodeT5+ Fine-Tuning 41

4.1.3 Comparison of Fine-Tuning Effectiveness for GraphCodeBERT and CodeT5+ 43

4.2 Comparative Performance Analysis of CodeT5+, GraphCodeBERT, and Llama 3.2 1B 45

4.2.1 Receiver Operating Characteristic Analysis 46

4.2.2 Comparison of Model Performance 49

5 Conclusions **55**

5.1 Future Work 56

References **58**

Appendices

A Code Clone Type Examples **A-1**

B Use of Generative AI **B-1**

List of Figures

2.1 Abstract syntax tree of the pseudo code presented in Algorithm 1 . . 12
2.2 Program dependence graph of the pseudo code presented in Algorithm 1 13

3.1 The steps of the pipeline from fine-tuning to model evaluation 23
3.2 The process for transforming two code fragments from a code pair into a single feature vector 34

4.1 Receiver Operating Characteristic (ROC) curves for clone detection on the Java dataset using all three models. 46
4.2 Receiver Operating Characteristic (ROC) curves for clone detection on the Python dataset using all three models. 47
4.3 Receiver Operating Characteristic (ROC) curves for clone detection on the Java dataset including only weakly type-3 and type-4 clones using all three models. 48
4.4 Confusion matrices of each model in the Java Dataset 50
4.5 Confusion matrices of each model in the Python Dataset 52
4.6 Confusion matrices of each model in the Java Dataset with weakly type-3 and type-4 clones 54

List of Tables

3.1 Overview of CodeT5+ model variants [30] 19

3.2 Fine-tuned code clone models and their base models 25

3.3 CodeXGLUE-BigClone - Token Counts (raw vs whitespace removed)[39] 29

3.4 PoolC - Token Counts (raw vs whitespace removed)[39] 30

4.1 Comparison of GraphCodeBERT base, CLS pooling fine-tuned, and mean pooling fine-tuned variants for code clone classification on the Java dataset. The percentage illustrates the best fine-tuned variant's performance against the base model. 39

4.2 Comparison of GraphCodeBERT base, CLS pooling fine-tuned, and mean pooling fine-tuned variants for code clone classification on the Python dataset. The percentage illustrates the best fine-tuned variant's performance against the base model. 40

4.3 Comparison of `codet5p-110m-embedding` and its fine-tuned variant for code clone classification on the Java dataset 41

4.4 Comparison of `codet5p-110m-embedding` and its fine-tuned variant for code clone classification on the Python dataset 42

4.5 Average cosine distances for non-clone and clone code embedding pairs for Java and Python. Percentage change shows the relative effect of fine-tuning compared to the base model, evaluated on 25,000 clone and non-clone pairs. 44

4.6 Comparison of classification performance metrics for clone detection on the Java dataset using fine-tuned CodeT5+, GraphCodeBERT fine-tuned with CLS pooling, and Llama 3.2 1B. 49
4.7 Comparison of classification performance metrics for clone detection on the Python dataset using fine-tuned CodeT5+, GraphCodeBERT, and Llama 3.2 1B. 51
4.8 Comparison of classification performance metrics on Java dataset with weakly type-3 and type-4 clones using fine-tuned CodeT5+, GraphCodeBERT fine-tuned with CLS pooling, and Llama 3.2 1B. . . 53

1 Introduction

1.1 Background and Motivation

Cloning source code is widely accepted as a bad coding practice. Cloned code itself is not the problem but it can multiply time spent debugging defects if the same fragment of faulty code is cloned in several locations. Cloning bug free code fragments is not safe either since while the code works somewhere else, it might not work as planned in another context [1]. Code clones can appear on multiple levels of granularity: statements, functions, classes, files, subsystems, and systems [2]. This thesis concentrates mainly on function level clones. Cloning code can seem like an easy way to minimize the time spent refactoring the code. However, it is usually a clear tell tale sign of a code with poor maintainability [1].

Code clones are generally labeled into four different types based on their similarity. These types are called type-1, type-2, type-3, and type-4 clones. Clone types can be thought to locate on a line with type-1 on its one end and type-4 on its other end. The similarities in type-1 clones are purely syntactic on a text level while similarities in type-4 clones are close to purely semantic [3].

Automatic code clone detection is an active research area in software engineering. Its findings can be utilized in tasks, such as code refactoring, bug finding, and code searching. Current automatic detection methods perform well in detecting type-1, type-2, and type-3 clones in small to large-scale code bases [4]. However, the

challenges arise when semantic, type-4 clones are tried to detect in large-scale code bases with multiple million lines of code [5].

1.2 Research Methods

Two research methods are used in this thesis: a literature review and method development. The literature review is used to gather a theoretical foundation on code clone types and detection methods. Method development focuses on comparing the code clone detection capabilities of three transformer-based models. During the method development phase, a pipeline was constructed that fine-tuned two of the transformer-based models, generated code embeddings from labeled code clone and non-clone pairs with the fine-tuned models and base models. An XGBoost classifier was trained with these high-dimensional feature vectors derived from the code embeddings and its performance was measured with multiple metrics.

The literature was searched mainly using ACM Digital library as it was found to contain the most relevant information about prior literature related to code clone detection methods. Google Scholar was also used to search highly specific, single studies. Literature searches were conducted using keywords including *code clone detection approaches*, *code clones*, *automatic code clone detection*, *token-based code clone detection*, *learning-based code clone detection*, and *graph-based code clone detection*.

The embedding generation capabilities of three transformer-based models for capturing similarities in Python and Java code are evaluated using a unified pipeline. The selected transformer-based models are CodeT5+, GraphCodeBERT, and Llama 3.2 1B. In addition to the base models, also multiple fine-tuned variants for CodeT5+ and GraphCodeBERT are used in the embedding generation. The pipeline is implemented and run locally using standalone Python scripts. The embeddings generated by the models are furthermore used to train a XGBoost classifier, which is used to

classify code pairs as clones or non-clones. Each model's performance is captured and analyzed using multiple different metrics. The proposed pipeline is not intended to be a complete code clone detection tool by itself, but rather to contribute meaningful insights into how the selected transformer-based models are suited for generating embeddings for code clone detection tasks. To use the proposed approach as a complete code clone detector, a parser would need to be implemented to extract code fragments from the source code at the desired level of granularity. Also, a proper clone reporter service that visualizes the cloned code fragments would need to be implemented.

1.3 Research Problem and Questions

The rapid development of code-specific transformer-based models has opened new possibilities for automatic code understanding and semantic code analysis. However, it remains unclear how generation of code embeddings for code clone detection tasks differ between smaller code-specific models and larger general-purpose language models when fine-tuning is applied to fully leverage the capabilities of the code-specific models.

The objective of this thesis can be summarized in three research questions:

- **RQ1:** What types of code clones exist?
- **RQ2:** What different approaches are presented in the literature for code clone detection?
- **RQ3:** How do smaller code-specific transformer models and their fine-tuned variants compare to larger general-purpose models in generating embeddings for code clone detection?

1.4 Thesis Structure

In Chapter 2, the basic concepts and background about code clones, including code clone types, common clone detection tools, and detection methods are reviewed.

Chapter 3 provides a review of the three transformer-based models utilized in the pipeline, an overview of the two code clone datasets used in the pipeline, and a detailed description of each of the pipeline's steps.

Chapter 4 evaluates the findings discovered through the test setting defined in Chapter 3. It provides a comprehensive analysis of how the three different transformer-based models and their fine-tuned variants perform in generating embedding representations from code fragments. The chapter also includes a surface-level analysis of how fine-tuning affected GraphCodeBERT's and CodeT5+'s performance. Chapter 4 discusses the key findings uncovered in the method development phase. Chapter 5 answers the research questions and suggests directions for future research.

2 Code Clone Taxonomy and Detection Approaches

The act of copying a piece of code from one location in a code base and pasting it to another place to perform another task, is called code cloning [6]. The code can be pasted with or without making modifications to the original piece of code. However, code cloning can occur also through other mechanisms. Roy et al. introduced six examples of how code clones can be produced: copying and pasting code, forking, design, functional, and logic reuse, merging of two similar systems, system development with generative programming approach, and delaying restructuring the code [7]. These examples are not mutually exclusive and may overlap. In forking, a similar solution from the system is reused with the hope that it will be greatly modified in the future development of the system. Design, functional, and logic reuse can occur when a system has multiple similar subsystems or tasks and a developer reuses the high level approach to solve both cases. In these situations the solutions on the code level can vary a lot. As the name suggests, merging of two similar systems means combining two different systems into one. This can create code clones as the different systems may have been developed by different teams and the developers may have not had clear visibility to the implementations in the other system. System development with generative programming approach means using tools that generate, for example, ready-to-use templates for common tasks,

like CRUD operations. Lately, studies have researched how the use of generative AI tools may cause code clones in software systems [8]. Delay in restructuring occurs when developer plans to refactor a piece of code in the future which can also lead to occurrence of code clones when the code is used as reference later.

Cloning code is a major reason behind maintainability challenges in software engineering. Cloning code can enhance the development speed significantly, as developers copy already existing code and make some modifications to it so that it solves another task. At times, this can seem more convenient and faster solution than modifying already existing code to perform multiple tasks. However, the achieved performance enhancements can vanish when a code base with a large amount of cloned code enters the maintenance and further development phase [1]. It has been estimated that this phase can consume up to 80% of the total cost of the project [1].

2.1 Code Clone Types

Code clones are commonly classified into four different types based on their similarity. These types are separated into syntactical (types 1-3) and semantical (type-4) classes based on syntactic and semantic similarities [9]. Syntactic similarity denotes that the code fragments have similarities on the text level, while semantic similarity denotes that the code fragments are designed to perform a similar functionality but on textual level they can differ a lot. Code clones can be divided into four types: type-1, type-2, type-3, and type-4. Clone types 1 to 3 are considered to belong to the class of syntactic similarities and clone type-4 belongs to the class of semantic similarities [9]. The classification of the clones into these four types is not always straightforward and it might not be trivial in which type a clone should be classified in. Especially, the division between type-3 and type-4 clones is challenging. Clones in this area are often referred to as clones in the Twilight zone [3]. Examples of each

code clone type can be found in Appendix A.

2.1.1 Type-1

Type-1 clones are code fragments that are copied with possible modifications only made to white spaces or comments. Type-1 clones are the simplest type of clones and they can be effectively detected using hybrid approach based on tree-based techniques and text based techniques [6].

2.1.2 Type-2

Type-2 clones are code fragments that are structurally nearly identical with each other. In addition to possible modifications on white spaces and comments, there might be differences in identifier names, data types, variables, and literals [3]. Studies show that type-2 clones can be effectively detected using hybrid approach of token based and abstract syntax tree (AST) based approaches [6].

2.1.3 Type-3

Type-3 clones are syntactically similar as well as types 1 and 2. However, in type-3 clones there can be more variation between the clones since cloned code fragments can have added or removed statements. Clones locating in the so called Twilight zone between type-3 and type-4 clones can furthermore be divided into 3 subtypes based on their syntactic similarity [3]:

- Strongly type-3: 70–90% similarity
- Moderately type-3: 50–70% similarity
- Weakly type-3: below 50% similarity

Large-Variance Clones are also a subtype of type-3 clones [4]. Large-Variance Clones are caused by deleting or inserting a large number of statements in a single

or various locations in a code fragment. Large-Gap Clones are a subtype of Large-variance clones and they are caused by deleting or inserting a large number of statements in a single location in a code fragment [4].

2.1.4 Type-4

The similarities between type-4 clones are mostly or purely semantic. Type-4 clones can differ drastically from each others both on textual syntactic level and on the code fragment's complexity level. Type-4 clones have emphasis on the functional objective of the code fragment instead of the syntax which is used to achieve functional task. [3] Among these four clone types, automatic detection of types 3 and especially 4 clones is currently the most challenging task [3].

2.2 Detection Methods

Code clone detection has been an active research area for the past two decades and researchers have proposed many different detection methods [10]. Clone detection methods can be classified into multiple different categories based on their underlying techniques of analyzing the code fragments. This thesis focuses on the six approaches that are discussed extensively in prior literature. These approaches are text-based approaches, token-based approaches, tree-based approaches, graph-based approaches, deep-learning-based approaches, and metric-based approaches. The most suitable clone detection approach depends greatly on the clone detection problem that is aimed to be solved. For example, a method that performs well on detecting type-1 clones can be powerless in detecting other types of clones. The rapid recent development of Large Language Models (LLMs) has also accelerated the development and enhanced the performance of deep learning -based clone detection approaches [11].

2.2.1 Text-based Approaches

As the name suggests, text-based clone detection methods compare the text level similarities of code fragments. These methods are usually relatively simple. Usually they don't need any pre-processing of the code fragments since only purely textual methods are used. Text-based methods are most effective in detecting type-1 and type-2 clones [12]. Text-based methods simply compare code fragments line by line. The lightweightness and simplicity makes text-based methods scalable and they can be used also in very large code bases [10]. For example, NiCad is a text-based tool used in detecting syntactical code clones. NiCad locates code clones using longest common subsequence method [6].

2.2.2 Token-based Approaches

Token-based clone detectors transform code fragments into a sequence of tokens and compare these sequences with each other to detect similarities. Token-based approaches are considered low-complexity and highly scalable. These approaches ignore the semantic meaning of the code fragment, which limits them to detecting only types 1, 2, and 3 clones [13]. Different token-based detection tools have some variation in their phases but in general they have two high-level phases: lexical analysis and clone detection [12]. The steps of these phases can vary greatly between different approaches and tools. Additionally, the names used for different phases and their steps vary across the literature meaning they are not well-established.

Lexical analysis is the initial phase of token-based approaches. The main purpose of this phase is to transform the source code into a sequence of tokens for further analysis. Lexical analysis can contain steps like pre-processing, filtering and cleaning, code normalization, and tokenization. In pre-processing, the source code is parsed into code fragments on the desired granularity level. For example, well known token-based clone detection tool SourcererCC, enables analysis on multiple

granularity levels such as files, methods, statements, and blocks [14]. In filtering and cleaning, unnecessary elements like comments and white spaces can be removed from the code fragments. Code normalization is a step that aims to reduce the impact of varying identifier names, constant values, and other surface-level variations on the clone detection. In this step, for example, identifier names can be unified and constant values can be abstracted or replaced with placeholders [15]. The design decisions made while implementing this stage can have major impact on the outcome of the detection process. For example, the NIL detection technique does not perform identifier normalization, such as renaming variables or function names. This leads to a lower false positive rate. However, it also reduces the ability to detect type-2 clones in cases where identifier names have been modified [4]. Tokenization is a phase where the extracted code fragments are transformed into individual tokens, which can be relatively resource-heavy task [16].

Clone detection is the phase where similar code fragments are identified as clones. There are many different approaches on detecting identical or nearly identical subsequences of tokens. For example, suffix-tree-based token-by-token comparison is a well-researched detection method in literature [6]. Suffix tree is a data structure that stores all the suffixes of a string. In the context of token-based clone detection approaches streams of tokens are translated into suffix trees. In suffix trees the textual value of the suffix is initialized as a key and the position in the token as its corresponding value. Methods like longest common subsequence can be utilized with suffix trees to detect similar sequences in the token streams. Extraction of clone matches using suffix trees can be performed in time complexity of $O(N)$ making it a flexible method also for larger code bases [17].

2.2.3 Tree-based Approaches

Tree-based approaches utilize abstract syntax trees (AST) or parse trees in detecting code clones. In these approaches code fragments are transformed into tree structures with a lexical analyzer or a parser [12]. They perform the best in detecting type-3 clones but have challenges in scaling to large-scale code bases [6]. Tree-based approaches have two main phases: transforming code into a tree structure and using tree matching algorithms to detect similar subtrees [10]. The main task of detecting similar subtrees is simple. However, when large code bases are investigated, comparing every subtree with every other subtree becomes computationally very expensive and unfeasible to implement in practise. If no optimization is done in comparing subtrees, the time complexity of detecting similar subtrees is $O(N^2)$ or even worse [18].

Parse tree or concrete syntax tree is a data structure that stores the structure of a code fragment in a tree structure. The nodes of a abstract syntax tree represent the syntactic construct of the code and edges represent the relationships between the constructs. It corresponds directly to the syntactic representation of the code fragment. It contains every syntactic detail of the code fragment [19].

Abstract syntax tree is a data structure that stores the structure of a code fragment also in a tree structure. The difference to a parse tree is that abstract syntax tree represents the structure of the code in more abstract way, for example, by abstracting syntactic details out [19]. Abstract syntax trees can be used in varying areas in computer science and code clone detection is one of them [20].

Algorithm 1 Pseudo code of a function that returns the square of the parameter with the larger absolute value

```
function maxSquare(x, y):
    if x * x > y * y:
        result := x * x
    else:
        result := y * y
    return result
end function
```

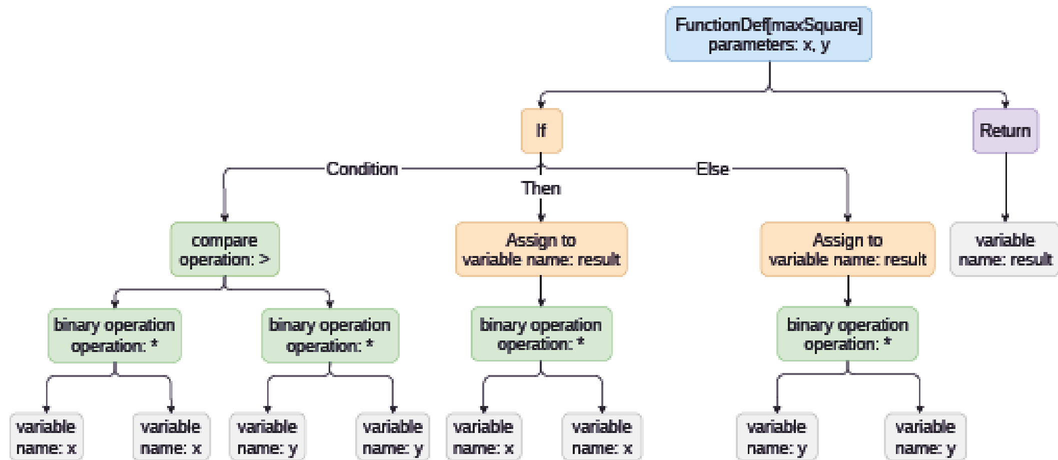

Figure 2.1: Abstract syntax tree of the pseudo code presented in Algorithm 1

2.2.4 Graph-based Approaches

In graph-based code clone detection approaches, the code fragments are represented in a graph structure and analyzed for further clone detection [20]. Graph-based approaches usually utilize the capabilities of program dependence graphs (PDG) to show the code fragment's or program's control flow and dataflow dependencies [17]. Once the PFGs are formed from the code fragments, matching algorithm is used to

detect structurally similar subgraphs. Graph-based approaches are used especially in detecting type-3 and semantic type-4 code clones since they can capture the semantics of the code fragment [17]. However, using program dependence graphs in large-scale code bases is challenging since generating them is a time consuming task [12].

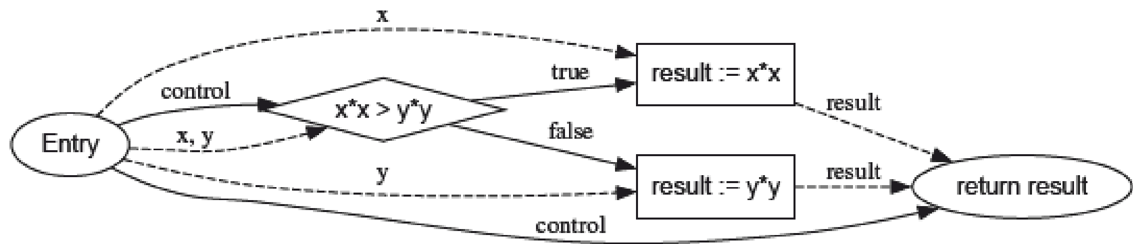

Figure 2.2: Program dependence graph of the pseudo code presented in Algorithm 1

2.2.5 Deep-learning-based Approaches

Deep-learning-based approaches include a vast number of different techniques of detecting code clones. Multiple different techniques have been proposed that are utilizing Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), and Graph Neural Networks (GNN). Lately, approaches utilizing Large Language Models have also been proposed [21]. Neural networks are very commonly utilized in deep-learning-based approaches. These methods usually have two main steps: neural network is used to calculate the embeddings for each code fragment and then the similarity of these fragments are calculated [22]. In recent years, the development of multiple Large Language Models (LLMs) has driven growing interest in LLM-based code clone detection. Prior literature has proposed methods like simple-prompt, zero-shot prompting, and chain-of-thought approaches. LLM-based techniques utilize the ability to prompt the model using natural language [21]. An example prompt for zero-shot approach could be:

```
Analyze the following two code functions. Decide if they perform the
same functionality. In other words, analyze if the functions are
semantic clones. Ignore the syntactical differences, like formatting,
identifier names, and control structure differences.

Respond with only:
CLONE or NOT_CLONE.

Function A:
<CODE_A>

Function B:
<CODE_B>
```

LLMs can also be employed as embedding generators instead of querying the LLM with natural language prompts and placing code fragments in the queries [21]. In embedding-based approaches the code fragments are transformed into multi-dimensional feature vectors. The feature vectors of transformed code fragments are furthermore compared using different similarity tests, for example, with cosine similarity. Modern large-scale software systems often use multiple programming languages. Comparing feature vectors enables cross-lingual code clone detection as code fragments implemented in different programming languages can be transformed into a same feature space [23].

2.2.6 Metric-based Approaches

Metric-based approaches calculate hand-selected metrics for code fragments in the source code and compare these code fragments based on the calculated metrics. Examples of commonly compared metrics include fan-out, cyclomatic complexity,

and the ratio between input and output variables [24]. Metric-based approaches have also high scalability as they rely on relatively simple calculations. However, this comes with the cost of low portability, meaning that these approaches cannot be used in other environments or programming languages without significant modifications [24].

3 Comparison of Three Transformer-based Models for Code Clone Detection

In this chapter a clone detection pipeline is proposed in order to compare the performances of three different transformer-based models. The pipeline contains an embedding-based code clone detector which utilizes XGBoost to classify code clones. This detector is used to compare code clone detecting capabilities of three transformer-based models. The models selected for comparison are CodeT5+, Llama 3.2 1B, and GraphCodeBERT. CodeT5+ is a set of Large Language Models that are specifically pretrained for code-related tasks [25]. In contrast, Llama 3.2 1B is a general-purpose LLM based on transformer architecture [26]. GraphCodeBERT is a pretrained BERT-style model that is specifically trained for code-related tasks.

These models are selected in order to compare how relatively compact models that are trained specifically for code logic perform against drastically larger general-purpose LLM. These models are compared in an unified pipeline that consist of three to four steps, which are designed to identify which aspects of the models' architectures, fine-tuning strategies, and feature vector generation have the greatest impact on the pipeline's code clone detection performance.

The pipeline is not designed to be a stand-alone code clone detection tool by

itself. However, transforming it into a fully functional Minimum Viable Product (MVP) of a code clone detecting tool would require relatively small modifications and additions to the source code. A functional MVP would need a parser that extracts the code fragments at a desired granularity level from the source code and a clone reporter service that visualizes the found code clones.

3.1 Models

The models compared in the pipeline are based on a transformer architecture which currently represents the state-of-the-art for both Natural Language Processing (NLP) and software engineering tasks [27].

3.1.1 Llama 3.2

Llama 3.2 is a family of decoder-only LLMs developed by Meta. The family includes four different sized models: Llama 3.2 1B, Llama 3.2 3B, Llama 3.2 11B, and Llama 3.2 90B. These models were trained with approximately 1, 3, 11, and 90 billion parameters, respectively.

Llama 3.2 1B is the most compact model of the family containing approximately one billion parameters. It is a text-only model, meaning it accepts text as an input and produces text as output [28]. As well as other Llama 3.2 models, the 1B model also has a context window of 128 thousand tokens. Context window describes the size of the space that a model can process at the same time. There is no general-purpose formula for converting lines of code into tokens, as different models use different tokenization strategies, programming languages have varying levels of syntactical complexity, and lines of code can differ significantly in length. However, it has been estimated that one hundred lines of Python code convert into approximately 1,000 tokens [29]. Using this conversion, the Llama 3.2 1B model

used in the clone detection approach proposed in this thesis has a context window of approximately 12,800 lines of code. In the proposed approach, the context window is populated with one fragment of Python or Java code at a time. This reserves only a small fragment of the full context window. Although utilizing the whole context window or even a large portion of it is not realistic in this test setting due to computational limitations, the large context window allows processing longer code fragments in the dataset without truncation and losing information of the code fragment. Llama 3.2 1B supports generating embedding vectors up to size of 2048. However, this comes at the cost of increased computational time.

In prior literature, Llama 3 models have been used in code clone detection mainly by applying prompt-based approaches, including zero-shot, one-shot, and few-shot methods, as discussed in section 2.2.5. In these test settings Llama 3 models have performed well in detecting syntactic clones but lacked in performance of detecting semantic clones [21]. Llama has been proved to perform better on LLM-generated datasets, which indicates a potential bias [21].

3.1.2 CodeT5+

CodeT5+ is a family of encoder-decoder code-specific LLMs developed by Salesforce [25]. It adapts Google's T5 (Text-to-Text Transfer Transformer) encoder-decoder architecture. Compared to its predecessor, CodeT5, it is pretrained with more diverse training data [30]. The family contains five different sizes of base models with 220 million, 770 million, 2 billion, 6 billion, and 16 billion parameters. For most of the models, there are also variants that are specialized on certain tasks or data domains. The additional variants contain bimodal, Python-specified, embedding, and instruction-tuned models. The bimodal model has been specifically trained for tasks which require handling code and natural language simultaneously, Python model is trained specifically with python code, and instruction-tuned model

is trained to accurately follow natural language instructions. The embedding model is tuned specifically to generate embeddings from code. [30]. In this, the embedding-specific `codet5p-110m-embedding` is used in generating embeddings from Java and Python code. The context window of CodeT5+ models is 512 tokens, which means that longer functions need to be truncated or skipped completely. The selected CodeT5+ variant extracts 256-dimensional vectors from code fragments.

Model type	**Model name**
110M embedding model	`codet5p-110m-embedding`
220M bimodal model	`codet5p-220m-bimodal`
220M base model	`codet5p-220m`
770M base model	`codet5p-770m`
220M Python-tuned model	`codet5p-220m-py`
770M Python-tuned model	`codet5p-770m-py`
2B base model	`codet5p-2b`
6B base model	`codet5p-6b`
16B base model	`codet5p-16b`
16B instruction-tuned model	`instructcodet5p-16b`

Table 3.1: Overview of CodeT5+ model variants [30]

CodeT5+ models are pretrained but can furthermore be fine-tuned with custom data. In prior literature CodeT5+ has been used in clone detection mainly by training it with code clone datasets and optimizing the encoder to produce embeddings that capture semantic similarity between code fragments [25]. With this approach, Codet5+ has reported precision of 94,1% and recall of 96,4% with the 220-million-parameter model and precision of 93,5% and recall of 96,7% with the 770-million-parameter model when using a dataset curated from BigCloneBench dataset [25].

3.1.3 GraphCodeBERT

GraphCodeBERT is a pretrained encoder-only model developed by Microsoft that is based on Transformer architecture for programming language. It aims to understand code semantics with data flow modeling in pre-training stage instead of

syntactic AST-approach [31]. GraphCodeBERT is a transformer-based model with a BERT-style encoder architecture. It has a context window of maximum of 512 tokens. The size of the context window forces us to skip relatively large number of code pairs is both datasets in the proposed clone detection approach. The embeddings generated with GraphCodeBERT encoder are 768-dimensional. BERT-style models are trained with masked language modeling. GraphCodeBERT is pretrained with CodeSearchNet dataset [32], which contains paired natural language documents and source code. The dataset contains code fragments in six different programming languages: Go, Java, JavaScript, PHP, Python, and Ruby, which gives GraphCodeBERT the code understanding capabilities primarily in these languages [32]. In the context of GraphCodeBERT masked language modeling means that parts of the input code tokens are randomly masked during pretraining and the model is trained to predict the masked tokens based on the surrounding code context. This masking process is illustrated in the following simplified example.

Original Code (W)	**Masked Input ($\tilde{W}$)**
`def add(a, b):`	`def add(a, b):`
`  return a + b`	`  return a + [MASK]`

3.2 Datasets

3.2.1 PoolC Dataset

A subset of PoolC [33] dataset of Python code clones is used in this experiment. The dataset contains 6,712,982 Python code fragment pairs that are labeled as non-clone or clone. There are approximately 600,000 unique code fragments and the

dataset is already divided into five different folds for training, testing, and validating models. Each fold is divided into a training split and validation split with approximately 80% of the data used for training and 20% for validation. Only the `PoolC/5-fold-clone-detection-600k-5fold` dataset is used in this experiment. Each row in the dataset contains seven columns: `code1`, `code2`, `similar`, `pair_id`, `question_pair_id`, `code1_group`, and `code2_group`. Columns `code1` and `code2` contain a string type formatting of two Python code fragments. These code fragments vary from short functions for mathematical calculations to larger code blocks which contain multiple function definitions and package imports. Column `similar` contains a binary value that indicates if these two code fragments are clones of each other. Other columns are not needed in the experiment. The dataset contains all types, type-1, type-2, type-3, and type-4 clones, which makes it a suitable dataset for evaluating and benchmarking code clone detection with multiple different models across a wide range of code clone scenarios [34].

3.2.2 CodeXGLUE-BigClone Dataset

BigCloneBench is the most popular benchmark used in code clone detection. It contains 43 different functionalities that are implemented with Java code [35]. The data to BigCloneBench is gathered from IJaDataset 2.0, which is a raw code repository that contains Java code from over 25,000 open source Java projects. In this thesis CodeXGLUE dataset [36], which is mined as well from IJaDataset 2.0, is used to test different transformer-based models' capabilities in generating code embeddings from Java code. CodeXGLUE-BigClone Dataset can be thought as a machine-learning-ready dataset with standard splits. The dataset contains a total of 1,731,860 rows, which are split into approximately 52% training data, 24% validation data, and 24% test data.

In CodeXGLUE-BigClone dataset, each row contains six columns: `id`, `id1`, `id2`,

`func1`, `func2`, and `label`. Column `id` is the identifier of the row, `id1` is the identifier of the first Java function, `id2` is the identifier of the second Java function, `func1` is the first Java function, `func2` is the second Java function, and `label` is the indicator if the two functions are clones. In this thesis only columns `func1`, `func2`, and `label` are used.

3.2.3 BigCloneBench Toma Subset

Feng et al. manually constructed five subsets from the BigCloneBench dataset containing Type-1 to Type-4 code clones for their token-based approach [37]. Subsets 1 and 2 include type-1 and type-2 code clone pairs correspondly. In addition, separate datasets were created for strongly and moderately type-3 clones. The subset used in this thesis include weakly type-3 and type-4 clones. This contains 109,914 clone pairs defined in a CSV file. Each row of the CSV file specifies the identifiers of the two code fragments in the clone pair, while the corresponding Java functions are stored separately as raw Java files in a different directory. Using this dataset requires more preprocessing than the PoolC and CodeXGLUE-BigClone datasets, which are provided as ready-to-use Hugging Face datasets.

3.3 System Architecture and Experimental Setup

An unified pipeline is used to compare code clone detecting capabilities of three transformer-based models' embedding generation for gradient boosting classifier. The selected granularity level of clone detection is the block level, meaning that the pipeline detects similar chunks of code.

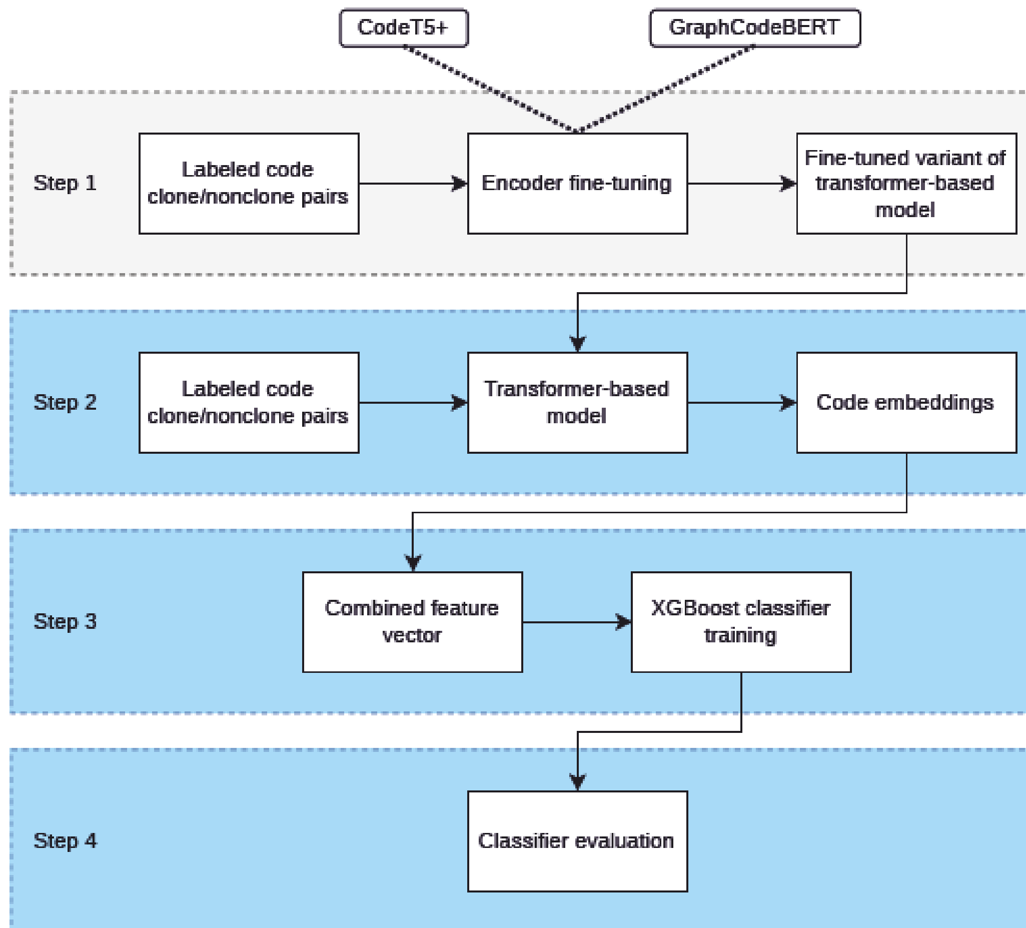

Figure 3.1: The steps of the pipeline from fine-tuning to model evaluation

The figure 3.1 illustrates the four steps of the clone detection embedding generation pipeline. The pipeline consists of three to four high-level steps. Step 1 in the figure is shown in a different color than the later three steps, as the first step is skipped when the pipeline is run using base models without further fine-tuning. At a high level, the pipeline consists of four main steps:

- Step 1: Fine-tuning the models
- Step 2: Generating embeddings from source code

- Step 3: Combining the labeled embedding pairs and training an XGBoost classifier
- Step 4: Comparing performance metrics between models

3.3.1 Fine-tuning the Models

The pipeline is executed in two configurations. In the first configuration, the transformer models generate embeddings from the code fragments without prior fine-tuning. In the second configuration, fine-tuning is applied to GraphCodeBERT and CodeT5+ for generating embeddings for the code clone detection task. This experimental design supports answering RQ3 because the smaller, code-specialized models are supported with appropriate task-specific data, allowing them to reach their optimal performance. This enables a relevant comparison against the larger Llama 3.2 1B model, which is evaluated with frozen parameters. From the pre-trained `graphcodebert-base` model, four fine-tuned variants are derived to produce language-specific code clone embeddings: Python-specific and Java-specific models using both CLS and mean pooling strategies. Models labeled with `cls` use CLS pooling during fine-tuning, while those labeled with `mean` use mean pooling. Two additional fine-tuned language-specific models for Java and Python are derived from `codet5p-110m-embedding` to generate code clone embeddings. The fine-tuning is executed in order to enhance the models' performance primarily in code clone detection in the specific languages. Fine-tuning's secondary objective is to enhance the models' capabilities in interpreting these languages. However, the dataset of 4,000 clone and non-clone pairs is relatively small compared to the massive corpora used in models' pre-training. Because the secondary objective likely has only little effect on the models' code clone detection, the primary and secondary objectives are not measured separately. All fine-tuned models used in this thesis are trained using a batch size of 4 with gradient accumulation over 2 steps, resulting in an ef-

fective batch size of 8. The models are fine-tuned for 4 epochs with a learning rate of 2×10^{-5}, using a tuning dataset of 4,000 samples, containing labeled clone and non-clone code pairs.

Fine-tuned Model	**Base Model**	**Target Language**
graphcodebert-clone-java-mean	graphcodebert-base	Java
graphcodebert-clone-java-cls	graphcodebert-base	Java
graphcodebert-clone-python-mean	graphcodebert-base	Python
graphcodebert-clone-python-cls	graphcodebert-base	Python
codet5p-clone-java	codet5p-110m-embedding	Java
codet5p-clone-python	codet5p-110m-embedding	Python

Table 3.2: Fine-tuned code clone models and their base models

Fine-tuned CodeT5+ Models

The fine-tuning of both CodeT5+ models was conducted in bi-encoder configuration. CodeT5+ is an encoder-decoder model meaning that it uses the encoder to process input code and the decoder to generate output sequences [25]. However, in the proposed code clone detection approach, decoder is not utilized in generating the embeddings for code pairs. Because of this, only the encoder is fine-tuned. The fine-tuning is performed with code pairs in `PoolC` and `CodeXGLUE-BigClone` datasets. Bi-encoder configuration means that both code fragments in a clone pair is passed separately through the encoder, producing independent embeddings for both fragments.

The fine-tuning process begins by passing two code fragments to the encoder. The encoder generates 256-dimensional embeddings for both of the code fragments. After this, the two embeddings are concatenated by calculating their element-wise absolute difference and dot product. The embedding generation process during fine-tuning is identical to the process used in the final proposed code clone detection approach and is described in more detail in subsection 3.3.2. After the encoder's generated embeddings have been concatenated, the embedding is passed through a

feed-forward classifier to predict whether the code pair is a clone or not.

Fine-tuned GraphCodeBERT Models

GraphCodeBERT doesn't have any language-specific base models. In this approach fine-tuning is performed by fine-tuning the same GraphCodeBERT four times. The Python-specific `PoolC` and Java-specific `CodeXGLUE-BigClone` datasets are used with two pooling strategies: `[CLS]` token pooling, native to GraphCodeBERT, and mean pooling, which may capture richer sequence-level representations. The effectiveness of both fine-tuning strategies are compared separately.

$[CLS]$ **token pooling** based fine-tuning of GraphCodeBERT includes five main stages. The stages are preprocessing, input sequence construction, Siamese forward pass, heuristic feature fusion, and optimization & backpropagation. In **preprocessing** stage, comments and docstrings are removed from the code fragments. Also, ASTs are generated from the cleaned code fragments. GraphCodeBERT adopts a graph-based approach on representing the code instead of AST-based approach, but ASTs are still needed to generate Data Flow Graphs (DFG) from the code fragments. After constructing the ASTs, the corresponding DFGs are generated for each code fragment. In **input sequence construction** stage, the code fragment is tokenized and concatenated with the corresponding DFG nodes, with the `[SEP]` token separating the two components. The final input sequence is constructed as follows: `[CLS]` + Code tokens + `[SEP]` + DFG nodes. Also, padding and masking is applied to reach a fixed length of 512 tokens. This stage also includes the selection of code pairs from the dataset. Code pairs in which one or both code fragments exceed the maximum input length of the GraphCodeBERT encoder are excluded. Only code pairs in which both code fragments fully fit within the maximum sequence length of 512 token are kept. This prevents the fine-tuning process from being affected by truncated inputs, which could otherwise lead to the loss

of critical syntactic and semantic information and degrade model performance. In **Siamese forward pass** stage, both processed input sequences in a code pair are passed through a shared GraphCodeBERT encoder. Same encoder is used in order to ensure that both code fragments are projected into the same vector space using a consistent set of learned weights. In this stage the `[CLS]` token absorbs information about all the other tokens through attention mechanism. In **heuristic feature fusion** stage the two individual vectors generated in the forward pass are combined into one high-dimensional feature vector. The resulting feature vector is a 2305-dimensional vector, formed by concatenating the two embeddings produced in the forward pass along with their element-wise absolute difference and a scalar dot product. While the original research paper of GraphCodeBERT relied only on the scalar dot product of the $[CLS]$ embeddings for fine-tuning, the approach adopted in this thesis uses a high-dimensional feature fusion strategy [31]. By concatenating the individual code fragment embedding representations alongside their element-wise absolute difference and the dot product similarity, the model can effectively capture both individual semantic meanings of the code fragment and measures of dissimilarity and similarity within a single 2305-dimensional feature vector. **Optimization & backpropagation** is the last stage of the fine-tuning process. In this stage cross-entropy loss is used to calculate how far off the model's current understanding is from reality. Once the loss is calculated, it triggers the actual learning process, backpropagation, by calculating the gradients of the loss with respect to all trainable parameters in the Siamese network. After backpropagation has discovered the necessary changes, the optimizer applies them to the model's weights. This loop is repeated for every 10,000 batches across four epochs.

Mean pooling based fine-tuning of GraphCodeBERT includes all the same steps as $[CLS]$ token pooling based fine-tuning. The only difference lies in the heuristic feature fusion stage. In mean pooling based approach, instead of using the

$[CLS]$ token to generate the 2305-dimensional vector, the model calculates the arithmetic average of all token embeddings within the last hidden state of the encoder. Mean pooling is described in more detail in subsection 3.3.2.

DFG nodes in the end of the input sequence capture the semantic structures of the code fragment [31]. As $[CLS]$ token aggregates information also from the DFG nodes in the end of the input sequence, the assumption is that $[CLS]$ token pooling based fine-tuning is more effective in detecting semantic type-4 code clones compared to mean pooling based fine-tuning.

3.3.2 Generating Tokens and Embeddings from Source Code

In this step of the pipeline the desired dataset is fetched and the model generates embeddings from the fetched code fragments. The embeddings are generated for all three datasets: the `PoolC` dataset, the `CodeXGLUE-BigClone` dataset, and the `BigCloneBench Toma subset`. To capture the absolute embedding generation performance and enable a fair comparison for each model, only code fragments that fits the model's context window are used for each model. For GraphCodeBERT and CodeT5+ this means 512 tokens. Llama 3.2 1B can theoretically handle code fragments up to 128,000 tokens but in the test setup the maximum token count is limited to 2048 to make the computations more efficient. Even though the GraphCodeBERT and CodeT5+ can handle the same amount of tokens, this doesn't automatically mean that they can handle as long code fragments due to differences in their tokenization methods. Before embedding generation, each code fragment is preprocessed to exclude comments. This is performed to reduce syntactic noise, as comments don't affect the code's functionality. Removing comments also allows the models to handle longer code fragments.

Tokenization

CodeT5+ models `codet5p-220m-py` and `codet5p-220m` employed in this thesis, use the same Byte-level Byte-Pair Encoding (BPE) tokenizer as its predecessor CodeT5. The tokenizer has a vocabulary size of 32,100 tokens [38]. The vocabulary contains special tokens, such as `<pad>`, `<s>`, `</s>`, `<unk>`, `<mask>`, and 100 additional mask tokens `<extra_id_0>`–`<extra_id_99>`.

GraphCodeBERT has a RobertaTokenizerFast tokenizer based on byte-level BPE scheme [39]. The tokenizer has five different special tokens: `<pad>`, `<s>`, `</s>`, `<unk>`, `<mask>`. It has a vocabulary of size 50,265 tokens.

Llama 3.2 1B utilizes a SentencePiece tokenizer, which is based on BPE [40]. It has a vocabulary size of 128,000 tokens. The large vocabulary enables the tokenizer to represent source code in a fine-grained way, as it contains specific tokens for a wider range of subword sequences.

The tokenizer plays a crucial role in how long and complex code fragments the model can handle. The more compact form the tokenizer can tokenize the source code, the longer and more complex code fragments the model can handle. The number of tokens per word ratio the tokenizer produces is called the tokenizer's fertility. The closer the fertility is to one, the more efficient the tokenizer is [41]. In the clone detection approach proposed in this thesis the bigger fertility the tokenizer has, the shorter and less complex code fragments it can analyze.

Model	Avg (Raw)	Avg (WS removed)	Avg% Change
Llama 3.2 1B	309.9	251.5	-18.84%
CodeT5+	388.9	324.0	-16.70%
GraphCodeBERT	779.8	351.2	-54.96%

Table 3.3: CodeXGLUE-BigClone - Token Counts (raw vs whitespace removed)[39]

The Tables 3.3 and 3.4 represent the average token count when 1,000 code fragments were tokenized from both PoolC and CodeXGLUE-BigClone dataset. The table 3.3 illustrates how the models differ in tokenizing raw Java code into different

Model	Avg (Raw)	Avg (WS removed)	Avg% Change
Llama 3.2 1B	140.1	112.4	-19.79%
CodeT5+	180.1	151.6	-15.83%
GraphCodeBERT	217.7	146.2	-32.85%

Table 3.4: PoolC - Token Counts (raw vs whitespace removed)[39]

amounts of tokens. Llama 3.2 1B produces the most compact representations, with an average of 309.9 tokens per fragment. CodeT5+ produces representations a bit more loosely, with an average of 388.9 tokens per fragment. In contrast, GraphCodeBERT generates substantially more tokens from the same code fragments, producing an average of 779.8 tokens per code fragment. As a result, GraphCodeBERT can handle significantly smaller code fragments if no preprocessing or truncation is applied. As Table 3.2 shows, GraphCodeBERT's tokenizer is affected heavily by the amount of white spaces in the code fragment. The average token count decreases by 54.96% when whitespaces are removed. The number of tokens decreases much less for Llama 3.2 1B and CodeT5+, by 18.84% and 16.70%, respectively. Even after removing whitespace, GraphCodeBERT still produces slightly more tokens than CodeT5+. Llama 3.2 1B clearly produces the fewest number of tokens. In contrast, when comparing tokenization among the models with Python code, the Table 3.4 shows that GraphCodeBERT still produces more tokens than the other models, but the relative difference is clearly smaller. When the whitespaces are removed, GraphCodeBERT produces even less tokens than CodeT5+.

In conclusion, the model's tokenizer's fertility directly limits the length of code fragments the model can analyze without truncating or preprocessing the code. In the context of the clone detection approach proposed in this thesis, the higher the tokenizer's fertility is, the shorter and less complex clones and non-clones it can detect.

Embedding Generation

After the tokenizer has generated token sequence for a code fragment, the sequence is passed to the transformer model. As CodeT5+ and GraphCodeBERT have an encoder in their architectural design, the encoder is used to generate embeddings. Llama 3.2 1B is a decoder-only model, which makes its embedding generation differ from that of the other two models. Instead of passing the tokenized code through the encoders like with CodeT5+ and GraphCodeBERT, with Llama 3.2 1B the embeddings are generated by passing the input through a series of decoder blocks and the hidden states from the final layer are extracted [40]. In this phase, each token from a token sequence is transformed into an embedding representation. This causes different length token sequences to transform into different number of vector representations. In order to perform reliably comparable embeddings from the code fragments, the embeddings must be fixed into same length. Each model has a hidden size that determines the length of the embeddings it transforms each token into. GraphCodeBERT transforms each token from the sequence into a 768-dimensional embedding, the used CodeT5+ checkpoint transforms each token into a 256-dimensional embedding, and Llama 3.2 1B transforms each token into a 2048-dimensional embedding. These token-level embeddings are transformed into a single fixed-length embedding using mean pooling or CLS pooling, depending on the model. **Mean pooling** is used with Llama 3.2 1B and CodeT5+ to compress the token-level embeddings into a single fixed-length vector that represents the whole code fragment. Mean pooling is a strategy that reduces the sequence of token-level embeddings into a single fixed-size embedding by calculating the average of hidden states over all token-level embeddings. Depending on the transformer model, the pipeline uses attention mask to ignore padding tokens while performing mean pooling. Padding tokens are special tokens that are added to token sequences to unify all sequences in a batch to a unified length. In designing the pooling phase, the length of the output vectors can

be fixed to a desired length. The pipeline in the proposed approach transforms code fragments into embeddings whose dimensionality is determined by the hidden size of the underlying transformer model, meaning the selected `codet5p-110m-embedding` model produces 256-dimensional embeddings while Llama 3.2 1B produces 2048-dimensional embeddings. Equation 3.1 below represents how the final embedding **E** is derived from the hidden states of all tokens. In the equation h_i represents a token embedding and m_i corresponding value in the attention mask.

$$\mathbf{E} = \frac{\sum_{i=1}^{L} h_i \odot m_i}{\sum_{i=1}^{L} m_i} \tag{3.1}$$

With GraphCodeBERT, **CLS pooling** strategy is used instead of mean pooling to extract a single fixed-length embedding from the hidden states. BERT models usually use $[CLS]$ token to summarize the whole sequence [42]. The `[CLS]` (classification) token is placed at the first position in sequence, and its last hidden-state representation captures the summarized representation of the whole sequence making it suit well for classification and similarity-based tasks. In CLS pooling, only the $[CLS]$ token is used to represent the entire input sequence, which makes mean pooling unnecessary.

Transformer models differ in their objectives, architecture, and training data, causing them to represent the same code fragment in different ways. On average, similar code fragments should convert into similar embeddings in order to classify code clones reliably. This property is important when implementing clustering or distance-based clone detection approaches.

The final set of labeled embedding pairs is divided into a training set and a testing set, with the testing set sized 20% of the training set size. For the PoolC dataset, the validation split is used to construct the testing set, as there is no dedicated test split provided for the purpose. Due to the absence of three separate splits for training, fine-tuning, and testing in the dataset, the same training dataset is used for both

training and fine-tuning. The validation set used during fine-tuning is constructed from the training data in a manner that no samples appear in both splits. This separation helps preventing overfitting to the training data, which could cause worse generalization to unseen data. With CodeXGLUE–BigClone dataset, the predefined validation, training, and testing splits are used for validation, training, and testing. The BigCloneBench Toma subset is used only as a training set as the same trained model is wanted to be tested with all kinds of code clone types.

3.3.3 Combining the Labeled Embedding Pairs and Training an XGBoost Classifier

In the third step of the pipeline, embeddings generated from the code pairs by the transformer-based models are combined into feature vectors and an XGBoost classifier is trained for clone detection with the produced labeled feature vectors.

After step 2, 256-dimensional, 768-dimensional or 2048-dimensional embeddings remain depending on the model, one for each code fragment in the pair. These two embeddings are concatenated into a single feature vector that can be used in training a binary classifier. The concatenation is identical to the embedding concatenation in fine-tuning stage.

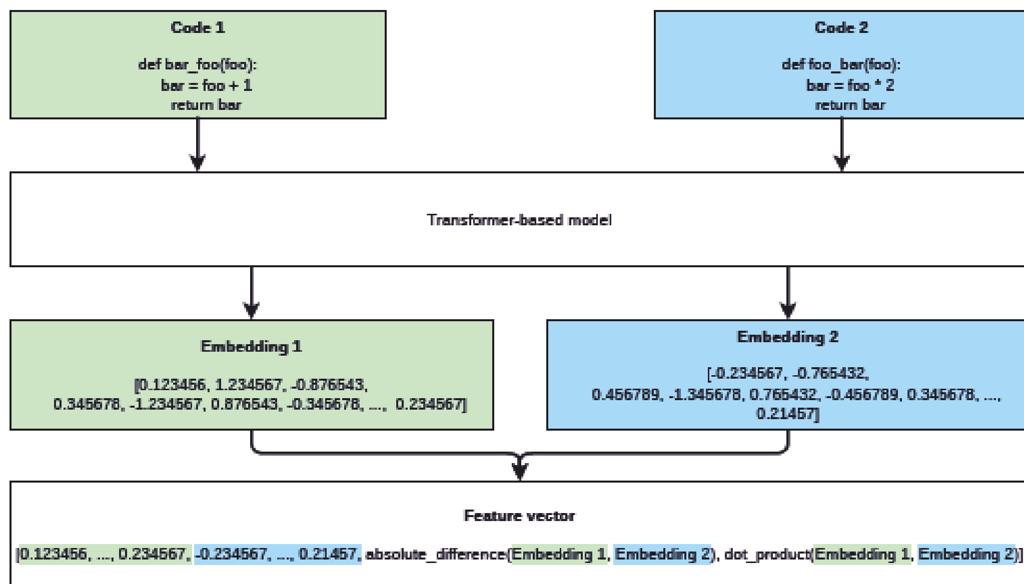

Figure 3.2: The process for transforming two code fragments from a code pair into a single feature vector

The process illustrated in Figure 3.2 produces feature vectors whose dimensionality depends on the base embeddings. For CodeT5+ this means 769-dimensional feature vector, 2305-dimensional feature vector for GraphCodeBERT, and 6145-dimensional feature vector for Llama 3.2 1B.

After the generation of single feature vectors, an **XGBoost** (eXtreme Gradient Boosting) classifier is trained with the labeled feature vectors. XGBoost is an efficient boosting algorithm, which combines multiple decision trees to perform classification and regression tasks. In the proposed approach, XGBoost is used as a classifier for code clone detection, where it is trained to classify fixed-length feature vectors as clones or non-clones. XGBoost is selected primarily as it is highly modifiable, it has a high accuracy on tabular data and it is efficient also with larger datasets. Randomized search with cross-validation is used in training the XGBoost classifier to find the best hyper parameters for embeddings generated by each trans-

former based model. Randomized search cross-validation is chosen over grid search to keep the hyperparameter optimization computationally more lightweight, as training the classifier with a dataset of 50,000 samples is computationally heavy when the samples are vectors with a dimensionality of 769 to 6145. The search evaluates the number of trees at values of 100 and 200, different learning rates of 0.05 and 0.1 and the maximum depth of each tree using values of 4, 6, and 8. Five different combinations of these hyperparameters are randomly selected and 2-fold cross-validation is performed with them. In total, 10 fits are performed to find the best configuration for CodeT5+ and GraphCodeBERT. For Llama 3.2 1B, only two different combinations of hyperparameters are selected for both folds to reduce the computational load.

3.3.4 Comparing Performance Metrics between Models

The final step of the pipeline evaluates the performance of the XGBoost classifier trained on embeddings produced by different models. For both Java and Python, six embedding variants are compared: GraphCodeBERT-base, GraphCodeBERT fine-tuned with CLS pooling, GraphCodeBERT fine-tuned with mean pooling, CodeT5+, CodeT5+ fine-tuned, and Llama 3.2 1B. The evaluated performance metrics include precision, recall, accuracy, F1-score, ROC AUC, Matthews Correlation Coefficient, and inference latency. Precision, recall, and F1-score are reported as macro-averages to weight both classes equally. While analyzing the results, it is noteworthy that the macro-averaged F1-score might not fall between macro-recall and macro-precision. By using macro-averaged precision and recall, the performance within both classes (clone and non-clone) can be summarized into single metrics.

Precision is calculated as the number of true positives divided by the total number of instances predicted as positive. In the context of the proposed clone detection approach, number of true positives mean the number of feature vectors correctly pre-

dicted as clones. This number is then divided by the total number of feature vectors predicted as clones to calculate the precision score. The higher the precision score is, the fewer non-clones the classifier identifies as clones. Within a code clone detection tool, high precision is important when false positives are costly, for example, when the code fragments are very complex and developers would waste time investigating code fragments incorrectly identified as clones. However, relying only on high precision is not enough as the classifier may still fail to detect many actual clones.

Recall is calculated by dividing the number of true positives by all the positives in the dataset. In the context of the proposed clone detection approach, the number of feature vectors correctly labeled as clones by the total number of clones in the dataset. Recall measures how well the classifier can classify all clones in the dataset. High recall is important in clone code detection tools when the cost of missing a true clone in the code base is costly, for example, when undetected clones could lead to hidden bugs or maintenance difficulties.

Accuracy measures the ratio between correctly, as clone or non-clone, classified feature vectors and the total number of the vectors. Accuracy can be used to measure the balanced overall performance of the classifier, where both false positives and false negatives should be minimized. Accuracy measures how correct the classifier is overall. However, alone it is not as expressive as precision or recall. High accuracy is important for example, in systems where automated decision making is done based on the classifier outputs.

F1-score is used to calculate the balance between recall and precision. It doesn't focus on true negative values, but only emphasizes how well the classifier identifies true values in the dataset. High F1 score indicates an effective overall performance of the classifier and is an important metric on its own in the proposed code clone detection approach. In the context of code clone detection, F1 score is important especially when both missed clones and false detections must be minimized.

Receiver Operating Characteristic Area Under the Curve (ROC AUC) is used to measure how well the classifier distinguishes non-clone pairs from clone code pairs across different decision thresholds. In the context of code clone detection, a high ROC AUC means that the detector can reliably distinguish clone from non-clone code pairs by consistently giving higher similarity scores to true clones across a wide range of decision thresholds.

Matthews Correlation Coefficient (MCC) is an effective metric to measure the performance of a binary classifier. MCC uses its full potential when it is used with imbalanced dataset. The dataset used in the proposed approach has balanced labels with 50% clones and 50% non-clones. However, MCC is still an efficient metric as it takes into account true positives, false positives, true negatives, and false negatives, covering all measures in the confusion matrix. The closer the MCC score is to one, the more consistently it classifies samples correctly. If the score is close to negative one, the classifier consistently classifies the samples incorrectly. If the score is close to zero, it indicates that the classifier gives no better results than random guessing.

Inference latency is measured to demonstrate how fast classifier can make predictions. It gives insights on how realistic it would be to use the proposed kind of code clone detection approach in a real-world scenario. When a codebase requiring clone detection grows in scale, the time the classifier spends on individual classifications can accumulate into long time periods. Inference latency is used to evaluate whether the approach remains practical and scalable as the size of the codebase increases. The measured inference latency times are not exact, as they are influenced by factors such as hardware constraints, background processes, and runtime variability. The exact obtained results should not be used to directly compare different models with each other. The magnitude of the results is more important within this metric.

4 Evaluation of Code Clone Detection Methods

4.1 Impact of Fine-Tuning on the Performance of Code-Specific Variants

For the code-specific models GraphCodeBERT and CodeT5+, their performance in embedding generation for clone detection was evaluated in a setting where multiple variants were fine-tuned on top of the base model. The variants were fine-tuned with labeled code clone datasets to adapt them to the downstream task of embedding generation for code clone detection.

The accuracies, precisions, recalls, and F1-scores among the models presented in the tables of this section and section 4.2 exhibit many identical or very similar values. This happens mainly because datasets are balanced with 5,000 samples per class. In the cases of CodeT5+ and GraphCodeBERT in Tables 4.6 and 4.7, the classifiers produce very few false positives and false negatives, which causes the metrics to be identical or very close to each other.

4.1.1 GraphCodeBERT Fine-Tuning

For both datasets, `PoolC` and `CodeXGLUE-BigClone`, GraphCodeBERT was fine-tuned under two settings. In the first setting, the `[CLS]` token was used in pooling

to transform the token-level embeddings into single fixed-length embeddings for both code fragments in a pair. In the second setting, mean pooling was applied by calculating the element-wise mean over all token-level embeddings to obtain a single embedding for both code fragments in the pair.

CLS Pooling vs. Mean Pooling in Fine-Tuning

CLS pooling is a native process for BERT-models to obtain a single embedding from token-level embeddings. Mean pooling is commonly used with models that do not include a `[CLS]` token, which motivates this comparison. Both pooling strategies are compared to identify the most effective approach across different scenarios.

Metric	Base model	CLS	Mean	Change
Accuracy	0.9462	0.9488	0.9483	+0.27%
Precision	0.9833	0.9830	0.9849	+0.16%
Recall	0.9078	0.9134	0.9106	+0.62%
F1-Score	0.9441	0.9469	0.9463	+0.30%
ROC AUC	0.9932	0.9935	0.9951	+0.19%
MCC	0.8950	0.8999	0.8992	+0.55%
Inference Latency (ms)	0.0055	0.0062	0.0055	0.00%

Table 4.1: Comparison of GraphCodeBERT base, CLS pooling fine-tuned, and mean pooling fine-tuned variants for code clone classification on the Java dataset. The percentage illustrates the best fine-tuned variant's performance against the base model.

The Table 4.1 indicates that the performance differences between the two fine-tuned variants and the base model are very minimal. There are no relevant differences in the performances of the models. The fine-tuned variants perform slightly better than the base model. The CLS pooling fine-tuned variant achieves the best score in accuracy, recall, F1-score, and MCC. Mean pooling fine-tuned variant on

the other hand achieves the highest score in in precision and ROC-AUC.

The fine-tuned variant using CLS pooling is selected for further analysis with other transformer-based models, as it performs slightly better than the other variants in four measured metrics.

Metric	**Base model**	**CLS**	**Mean**	**Change**
Accuracy	0.8275	0.8324	0.8133	+0.59%
Precision	0.8373	0.8320	0.8098	-0.63%
Recall	0.8130	0.8330	0.8190	+2.46%
F1-Score	0.8250	0.8325	0.8144	+0.91%
ROC AUC	0.9128	0.9149	0.9009	+0.23%
MCC	0.6553	0.6648	0.6266	+1.45%
Inference Latency (ms)	0.0096	0.0062	0.0059	-38.54%

Table 4.2: Comparison of GraphCodeBERT base, CLS pooling fine-tuned, and mean pooling fine-tuned variants for code clone classification on the Python dataset. The percentage illustrates the best fine-tuned variant's performance against the base model.

The Table 4.2 indicates that the performance differences between the two fine-tuned variants and the base model are also very minimal within Python-specific dataset. The variant fine-tuned with CLS pooling performed the best. It was slightly better than the base model in every other metric than precision. However, a bit surprisingly the variant fine-tuned with mean pooling performed the worst.

As the CLS pooled variant performs the best, it is selected for further analysis with other transformer-based models.

Variants' performance in clone detection within Java and Python code

The Tables 4.1 and 4.2 clearly indicate that all the GraphCodeBERT variants perform better within Java dataset than Python dataset. This can be caused by multiple different factors. Possible reasons can be the static typing of Java and dynamic typing of Python, possible large number of type-3 and type-4 clones in the Python dataset, and more diverse Java-related training data in GraphCodeBERT's pre-training phase.

4.1.2 CodeT5+ Fine-Tuning

For both datasets, `PoolC` and `CodeXGLUE-BigClone`, checkpoint `codet5p-110m-embedding` of CodeT5+ was fine-tuned once using mean pooling to obtain single embeddings for both code fragments in the pair.

Metric	Base model	Tuned	Change
Accuracy	0.8747	0.9857	+12.69%
Precision	0.9322	0.9874	+5.92%
Recall	0.8082	0.9840	+21.75%
F1-Score	0.8658	0.9857	+13.85%
ROC AUC	0.9486	0.9969	+5.09%
MCC	0.7561	0.9714	+28.47%
Inference Latency (ms)	0.0010	0.0011	+10.00%

Table 4.3: Comparison of `codet5p-110m-embedding` and its fine-tuned variant for code clone classification on the Java dataset

Table 4.3 indicates that fine-tuning of CodeT5+ enhances its performance significantly. Accuracy, precision, recall, and F1-score all increased consistently by approximately 6-22% each. MCC score increased by promising 28.47% due to fine-tuning, meaning the fine-tuned variant achieves a substantially better balance between true

and false predictions. Also, ROC AUC increased by 5.09% to 0.9969, approaching the theoretical maximum. The inference latency remained approximately the same. However, without more precise and controlled testing, the only reliable conclusion is the relative magnitude of the latency between these two variants are approximately the same.

Metric	Base	Fine-tuned	Change
Accuracy	0.8562	0.9420	+10.02%
Precision	0.8837	0.9463	+7.08%
Recall	0.8204	0.9372	+14.24%
F1-Score	0.8509	0.9417	+10.67%
ROC AUC	0.9353	0.9839	+5.20%
MCC	0.7142	0.8840	+23.77%
Inference Latency (ms)	0.0012	0.0011	-8.33%

Table 4.4: Comparison of `codet5p-110m-embedding` and its fine-tuned variant for code clone classification on the Python dataset

As well as with Java dataset, the Table 4.4 shows that the model's performance increased significantly after fine-tuning also with Python dataset. Accuracy, precision, recall, and F1-score all gained consistent improvements, with gains ranging from 7% to 14%. ROC AUC increased by 5,20% to 0,9839, which indicates variant's solid ability to separate clone and non-clone classes.

Variants' performance in clone detection within Java and Python code

Both the base model and the fine-tuned variants perform better on the Java dataset than on the Python dataset. However, fine-tuning noticeably improves the performance in both datasets. The performance improvements are moderately larger for Java dataset, but clear enhancements are also gained for Python dataset. Again, possible reasons for the better performance on Java include the language's typing

system, variations in clone type proportions in the datasets, and the differences within these languages during the model's pre-training data. The performance gap between the two languages is still relatively small.

4.1.3 Comparison of Fine-Tuning Effectiveness for GraphCodeBERT and CodeT5+

It is clear that fine-tuning enhanced CodeT5+ model's performance significantly more than the fine-tuning of GraphCodeBERT. GraphCodeBERT already achieved a decently strong level of performance on the Java dataset with the base model, leaving limited room for further improvement for fine-tuning. However, on the Python dataset, the base model performed poorly leaving more room for enhancements for the fine-tuned variant. Regardless of the large room for performance enhancements, the CLS pooling fine-tuned model performed only marginally better than the base model. The mean pooling fine-tuned model performed worse than the base model. This suggests that fine-tuning can only slightly enhance the performance of GraphCodeBERT or even degrade it in the proposed clone code detection approach.

Language	Model	Type	Base	Fine-tuned	Change
Java	CodeT5+	Non-clone	0.6366	0.9876	+55.12%
		Clone	0.5176	0.0092	-98.23%
	GraphCodeBERT (Mean)	Non-clone	0.1159	0.0790	-31.85%
		Clone	0.0736	0.0502	-31.78%
	GraphCodeBERT (CLS)	Non-clone	0.1159	0.0933	-19.53%
		Clone	0.0736	0.0584	-20.61%
Python	CodeT5+	Non-clone	0.3964	0.7920	+99.80%
		Clone	0.2386	0.0423	-82.30%
	GraphCodeBERT (Mean)	Non-clone	0.1677	0.1107	-34.00%
		Clone	0.1142	0.0757	-33.70%
	GraphCodeBERT (CLS)	Non-clone	0.1677	0.1368	-18.40%
		Clone	0.1142	0.0923	-19.20%

Table 4.5: Average cosine distances for non-clone and clone code embedding pairs for Java and Python. Percentage change shows the relative effect of fine-tuning compared to the base model, evaluated on 25,000 clone and non-clone pairs.

In the proposed code clone detection approach, if the fine-tuned model is successfully fine-tuned, it should generate embedding such that the distance between embeddings of clone code pairs is smaller than the distance produced by the base model. For non-clone code pairs, the fine-tuned model should generate embeddings with larger distances compared to the base model. Table 4.5 shows that fine-tuning CodeT5+ model behaves in this way. In Java dataset, clone pairs' distance decreased by 98,23% and the distance between non-clone pairs increased by 55,12%. For GraphCodeBERT, fine-tuning decreased the distance between clone pairs, but it also decreased the distance between non-clone pairs to a similar degree, or even more. In the Python dataset, the fine-tuned CodeT5+ model increased the distance between non-clone pairs even more than in the Java dataset, by 99.8%. The distance between clone pairs did not decrease as much proportionally as in the Java dataset,

but still achieved a significant reduction of 82.3%. GraphCodeBERT faced the same problems in the Python dataset as in Java dataset.

There are numerous possible reasons behind the poor effectiveness of fine-tuning on GraphCodeBERT's embedding generation capabilities for code clone detection. To get more meaningful insights on fine-tuning's effects on the model's performance in the proposed clone detection approach more advanced, refined, and controlled experiments should be conducted. However, possible reasons for fine-tunings little effect include overfitting or catastrophic forgetting caused by task-specific fine-tuning, a mismatch between the fine-tuning objective and GraphCodeBERT's pre-training objectives, the models emphasis on the dataflow of the code fragments through data flow graphs.

To better understand why fine-tuning enhances CodeT5+'s performance but not GraphCodeBERT's, the architectures and modeling approaches of the two models should be examined. Due to its text-to-text architecture, CodeT5+ may be more receptive to fine-tuning than GraphCodeBERT's DFG-based structure-aware approach.

4.2 Comparative Performance Analysis of CodeT5+, GraphCodeBERT, and Llama 3.2 1B

In prior subsections, the most effective variants were identified through a comparison of base and fine-tuned models. The models that were found the most effective are selected for comparison in this section. For CodeT5+, the fine-tuned model clearly performed best on both the Java and Python datasets. In contrast, for GraphCodeBERT, the fine-tuned model using CLS pooling achieved the best performance on the Java dataset, although the improvement was marginal. However, on the Python dataset, the base GraphCodeBERT model had slightly the best performance.

The test results presented in this section is produced by XGBoost classifier trained with 25,000 clone and 25,000 non-clone pairs for each model. The test results are produced by predicting the class of 5,000 clone and 5,000 non-clone pairs.

4.2.1 Receiver Operating Characteristic Analysis

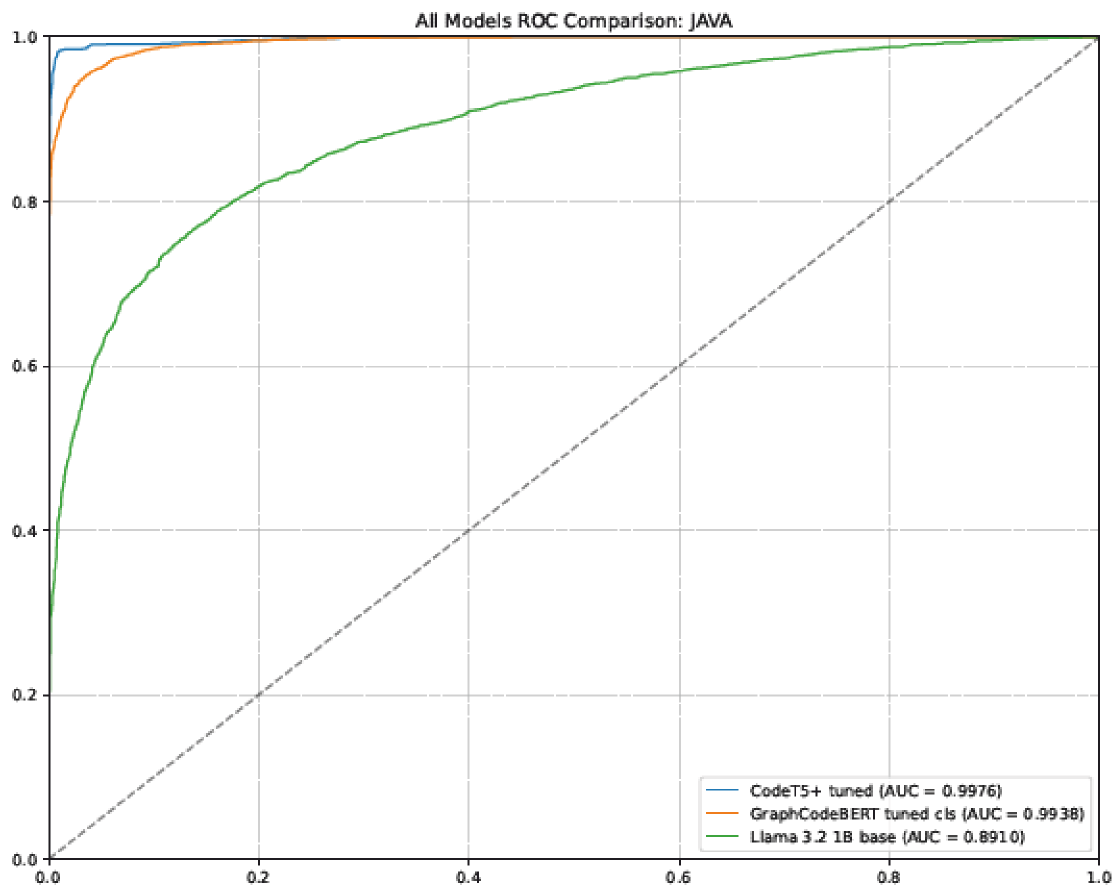

Figure 4.1: Receiver Operating Characteristic (ROC) curves for clone detection on the Java dataset using all three models.

Figure 4.1 visualizes clearly that the fine-tuned code specific models CodeT5+ and GraphCodeBERT, outperform the general-pupose LLM Llama 3.2 1B in distinguishing between clone and non-clone classes on the Java dataset. While the ROC curves of CodeT5+ and GraphCodeBERT reach high true positive rates almost immediately, the ROC curve of Llama 3.2 1B increases much slower. The ROC AUC

of CodeT5+ is exceptionally high (0.9976), which indicates nearly perfect discrimination performance. GraphCodeBERT also achieves excellent ROC AUC of 0.993, although it is slightly lower than that of CodeT5+. For Llama 3.2 1B, ROC AUC remains fairly decent at 0,8910, but is still clearly lower than that achieved by the two code-specific models.

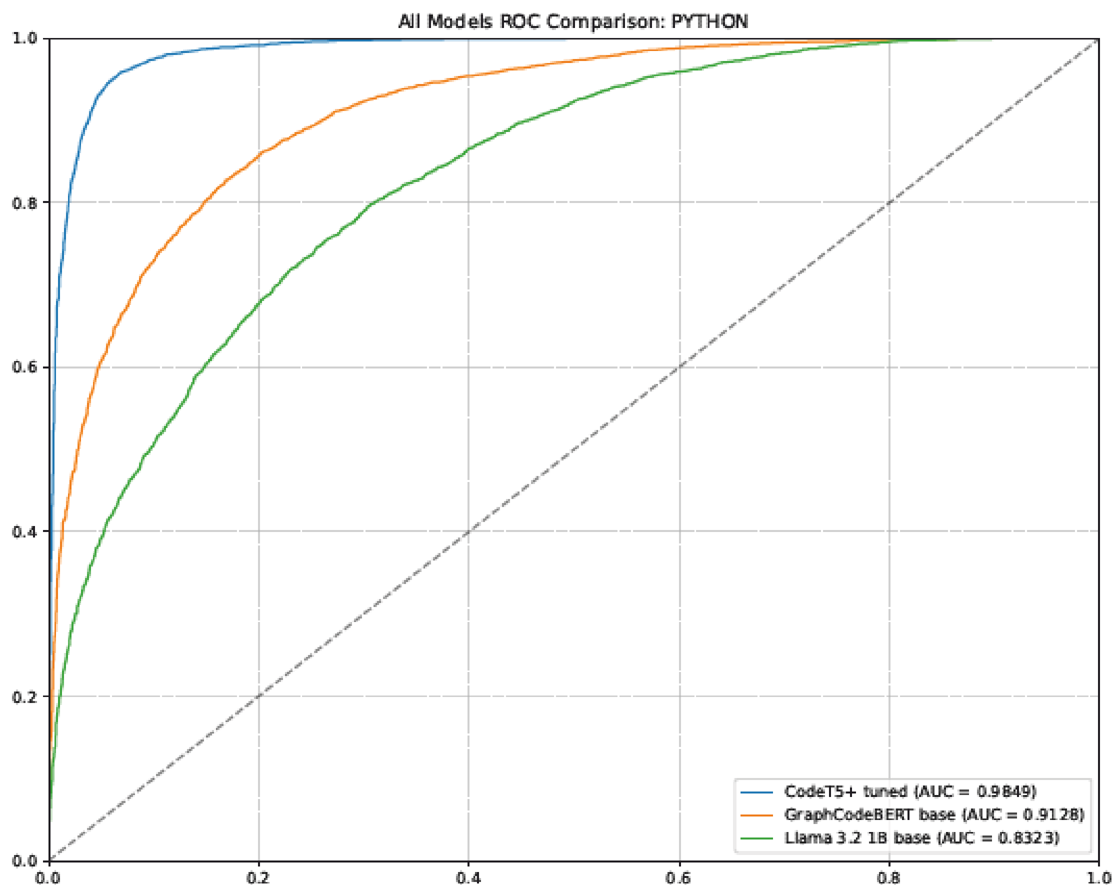

Figure 4.2: Receiver Operating Characteristic (ROC) curves for clone detection on the Python dataset using all three models.

In the Python dataset, Llama 3.2 1B also showed the weakest performance. However, the performance gap to the code-specific models was notably smaller than in the Java dataset, especially to GraphCodeBERT. The ROC AUC of each model decreased compared to the Java dataset. The ROC AUC decreased the least for

CodeT5+ resulting it to have clearly the highest value of 0,9849 compared to GraphCodeBERT's 0,9128 and Llama 3.2 1B's 0,8323. GraphCodeBERT's value still remains on a reasonably strong level, whereas Llama 3.2 1B's value indicates clear challenges in distinguishing the clone and non-clone classes from each others.

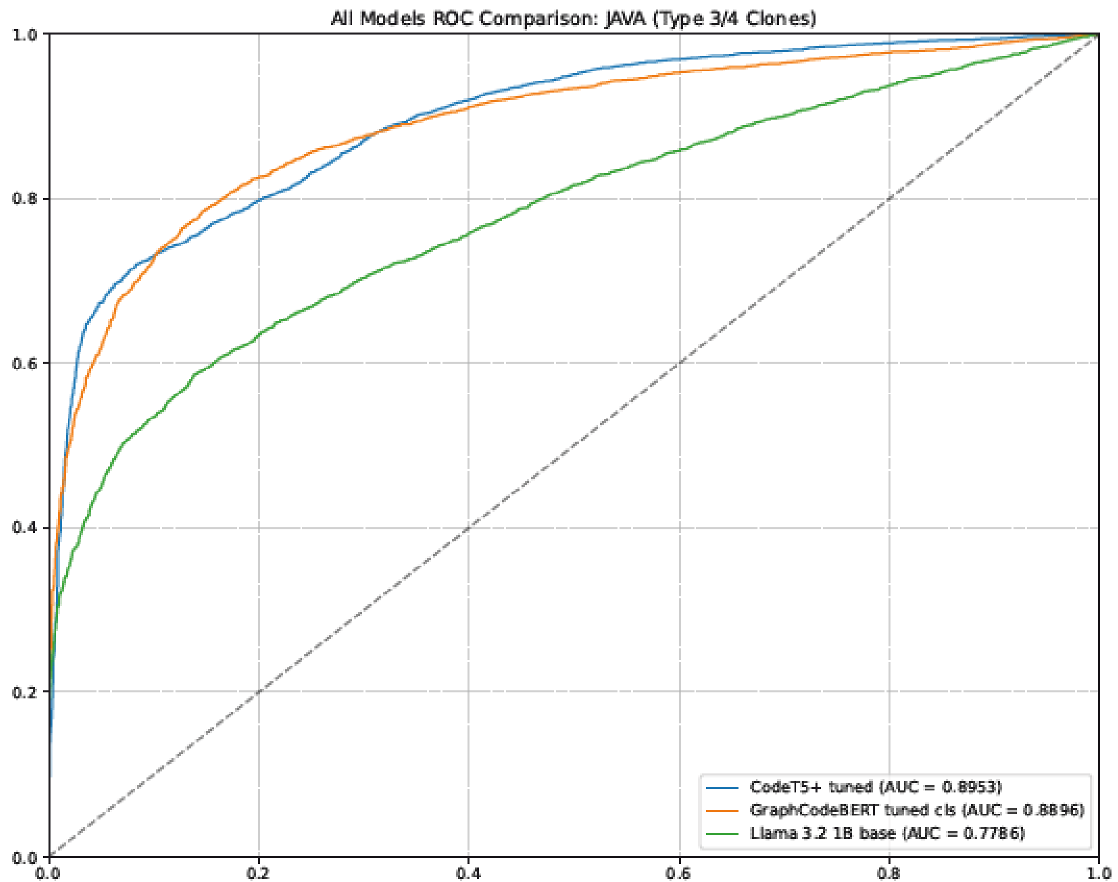

Figure 4.3: Receiver Operating Characteristic (ROC) curves for clone detection on the Java dataset including only weakly type-3 and type-4 clones using all three models.

As expected, all three models experience more difficulties in a dataset which contains weakly type-3 and type-4 Java clones. Even though CodeT5+ achieves a slightly higher ROC AUC value (0.8953) than GraphCodeBERT (0.8896) and a clearly higher value than Llama 3.2 1B (0.7786), it does not uniformly per-

form better than the other models across all classification thresholds. The ROC curves of CodeT5+ and GraphCodeBERT overlap at two thresholds, indicating that CodeT5+ performs better than GraphCodeBERT at low and high false positive rates, while GraphCodeBERT achieves higher true positive rates at between these two areas.

4.2.2 Comparison of Model Performance

Metrics with CodeXGLUE-BigClone dataset

Metric	**CodeT5+**	**GraphCodeBERT**	**Llama 3.2 1B**
Accuracy	0.9862	0.9517	0.8135
Precision	0.9883	0.9815	0.8611
Recall	0.9840	0.9208	0.7476
F1-score	0.9862	0.9502	0.8003
ROC AUC	0.9976	0.9938	0.8910
MCC	0.9724	0.9051	0.6325
Inference latency (ms)	0.0009	0.0053	0.0038

Table 4.6: Comparison of classification performance metrics for clone detection on the Java dataset using fine-tuned CodeT5+, GraphCodeBERT fine-tuned with CLS pooling, and Llama 3.2 1B.

As indicated by the ROC curves, Table 4.6 shows that CodeT5+ achieves consistently better performance among all relevant measured metrics than GraphCodeBERT and Llama 3.2 1B in the Java dataset. Both CodeT5+ and GraphCodeBERT reach metric levels that are expected for a well-performing code clone detector. In contrast, Llama 3.2 1B shows noticeably weaker performance, which are not high enough for practical code clone detection. Somewhat surprisingly, the inference latency is of the same order of magnitude across all models and would not cause any of them to be impractical for real-world use.

It is notable that CodeT5+'s metrics are exceptionally high compared to prior code clone detection research. The original study reported a recall of 96.8%, a precision of 94.1%, and an F1-score of 95.2% using 220-million- and 770-million-parameter models [22]. In contrast, the proposed clone detection approach achieves a recall of 98,4%, a precision of 98,8%, and an F1-score of 98.6%. GraphCodeBERT performs better than previously reported results in precision, achieving 98.2%. However, recall and F1-score slightly decreased to 92.1% and 95.0%, compared to 94.8%, 95.0%, and 95.2% in prior literature.

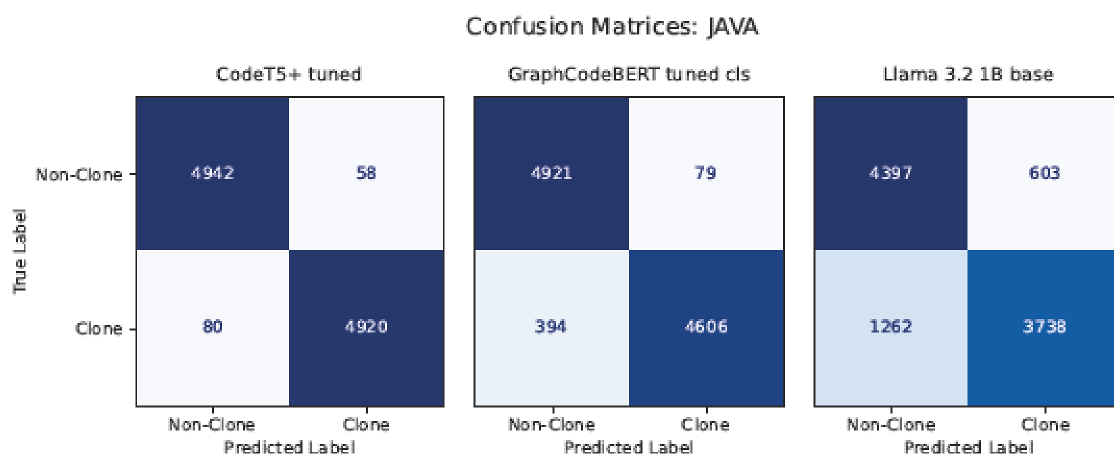

Figure 4.4: Confusion matrices of each model in the Java Dataset

Figure 4.3 visualizes well that CodeT5+ and GraphCodeBERT detect non-clones in the Java dataset on almost identical accuracy. However, larger differences appear when detecting actual clones, GraphCodeBERT's accuracy drops noticeably. GraphCodeBERT misclassifies a significantly higher number of clones as non-clones compared to CodeT5+, which explains the relatively large difference in their MCC values shown in Table 4.6. Llama 3.2 1B faces also the same challenge and tends to misclassify clones as non-clones, but it also incorrectly classifies a relatively high number of non-clones as clones.

Metrics with PoolC dataset

Metric	**CodeT5+**	**GraphCodeBERT**	**Llama 3.2 1B**
Accuracy	0.9437	0.8348	0.7452
Precision	0.9465	0.8409	0.7421
Recall	0.9406	0.8258	0.7516
F1-score	0.9435	0.8333	0.7468
ROC AUC	0.9849	0.9168	0.8323
MCC	0.8874	0.6697	0.4904
Inference latency (ms)	0.0019	0.0071	0.0036

Table 4.7: Comparison of classification performance metrics for clone detection on the Python dataset using fine-tuned CodeT5+, GraphCodeBERT, and Llama 3.2 1B.

Table 4.7 summarizes that CodeT5+ maintained its high performance also in the Python dataset. Its performance decreased slightly, but the decrease was much bigger within GraphCodeBERT and Llama 3.2 1B. Whereas GraphCodeBERT was a strong model in Java dataset, in Python dataset it fails to achieve the performance level that is required by a practical code clone detection tool. The results suggest that GraphCodeBERT is missing certain characteristics required for effective adaptation to the Python language. Even after fine-tuning, GraphCodeBERT was unable to successfully adapt to code clone detection in Python. Llama 3.2 1B's performance also degraded and fell far behind the best-performing model. Inference latencies didn't experience any significant changes.

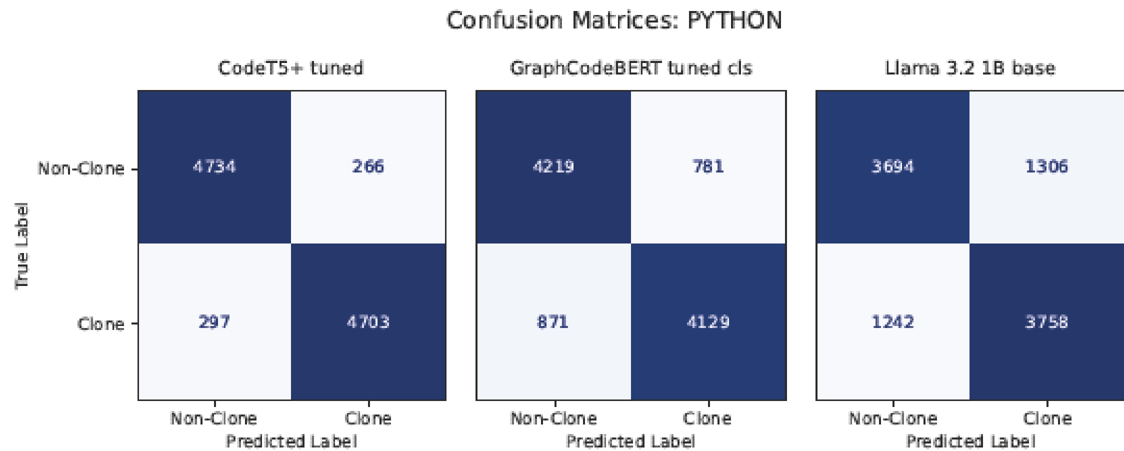

Figure 4.5: Confusion matrices of each model in the Python Dataset

Whereas all three models produced clearly more false negatives than false positives in the Java dataset, in the Python dataset the models predicted approximately equal numbers of false positives and false negatives. However, for CodeT5+ and GraphCodeBERT, the numbers of both false positives and false negatives increased substantially. In contrast, Llama 3.2 1B marginally reduced the number of false positives, but the number of false negatives more than doubled. Also, Llama 3.2 1B managed marginally to increase the number of true positives.

Metrics with BigCloneBench Toma subset

Metric	CodeT5+	GraphCodeBERT	Llama 3.2 1B
Accuracy	0.7831	0.7828	0.6997
Precision	0.7365	0.7361	0.6955
Recall	0.8816	0.8818	0.7104
F1-score	0.8025	0.8024	0.7029
ROC AUC	0.8953	0.8896	0.7786
MCC	0.5775	0.5770	0.3995
Inference latency (ms)	0.0009	0.0050	0.0022

Table 4.8: Comparison of classification performance metrics on Java dataset with weakly type-3 and type-4 clones using fine-tuned CodeT5+, GraphCodeBERT fine-tuned with CLS pooling, and Llama 3.2 1B.

Whereas the difference between the performances of CodeT5+ and GraphCodeBERT was very small in CodeXGLUE-BigClone dataset, the differences are even smaller in the BigCloneBench Toma subset. Even though CodeT5+ is slightly better in every other metric than recall compared to GraphCodeBERT, the differences are so small that the models's performances can be considered almost equal. In other words, GraphCodeBERT's performance degrades less than CodeT5+'s performance when the dataset contains only weakly type-3 and type-4 clones. As in other experiments, Llama 3.2 1B showed the worst performance across all metrics other than inference latency.

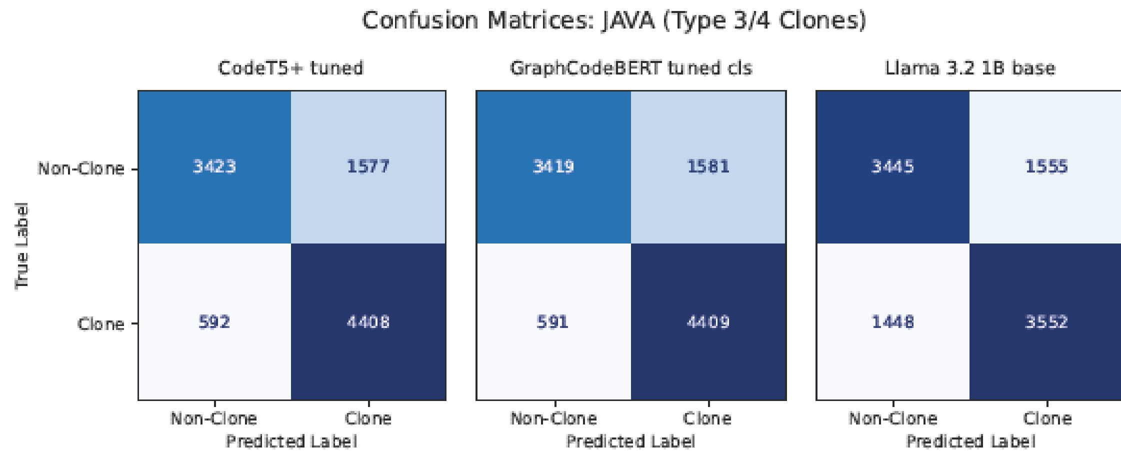

Figure 4.6: Confusion matrices of each model in the Java Dataset with weakly type-3 and type-4 clones

In the Java dataset containing all four types of clones, CodeT5+ was able to generate very few false positives and false negatives. GraphCodeBERT successfully minimized the amount of false positives, but had a weakness of predicting relatively more false negatives. In the BigCloneBench Toma subset containing weakly type-3 and type-4 Java clones, CodeT5+ and GraphCodeBERT show an increased tendency to predict false negatives, as shown in Figure 4.6.

5 Conclusions

The objective of this thesis was to examine different kinds of code clones and review current approaches in detecting them. In addition, the thesis aimed to compare three transformer-based models, GraphCodeBERT, CodeT5+, and Llama 3.2 1B, and their fine-tuned variants in terms of their effectiveness in generating embeddings for code clone detection tasks.

RQ1 (*What types of code clones exist?*) provided foundations for understanding RQ2 and RQ3. It helps to clarify the need for different kinds of code clone detection approaches.

The literature review showed that there are four main types of code clones: type-1, type-2, type-3, and type-4. Type-3 clones are at times separated into weakly, moderately, and strongly type-3 clones based on their level of similarity. In addition, type-3 clones included also a subset of Large-Variance Clones.

RQ2 (*What different approaches are presented in the literature for code clone detection?*) aimed to clarify what the current state-of-the-art code clone detection approaches are and how different code clone types require different approaches. It also helped to clarify the relevance of RQ3 in providing a foundational understanding for learning-based code clone detection approaches.

The literature review clarified that there is no clear taxonomy of different code clone detection approaches and many current tools use overlapping approaches. However, six of the most common approaches were identified in the prior litera-

ture: text-based, token-based, tree-based, graph-based, deep-learning-based, and metric-based approaches.

RQ3 (*How do smaller code-specific transformer models compare to a larger general-purpose model in generating embeddings for code clone detection?*) focused on the learning-based approaches reviewed in the literature review. It analyzed how well more compact, code-specific models perform in generating meaningful code embeddings compared to a larger general-purpose LLM.

The conducted tests clearly indicated that the code-specific models outperformed the general-purpose LLM in every relevant measured metric. As expected, the models' performance declined when detecting weakly type-3 and type-4 clones instead of all clone types simultaneously. Notably, for real-world usage, the tested general-purpose LLM (Llama 3.2 1B) generated embeddings for code fragments significantly slower, approximately 0.75 embeddings per second, compared to 13.78 embeddings per second for GraphCodeBERT and 9.19 embeddings per second for CodeT5+. In addition to Llama 3.2 1B's significantly poorer performance in detecting code clones, its much slower embedding generation pace further confirmed that code-specific transformer-based models outperform the general-purpose LLM in every aspect within the test setting defined in this thesis.

5.1 Future Work

The pipeline presented in this thesis can be run with multiple different combinations of settings. Only a few of these were explored in the work. Key components of these settings include the fine-tuning dataset, the strategy for combining embeddings, and the choice of classifier, including its decision threshold and training data. Also, the effect of fine-tuning the LLM can give insights, especially when detecting type-4 clones. A simple randomized search cross-validation was used to identify effective hyperparameters for the XGBoost classifier for each transformer-based model, as

it is less computationally and memory-intensive than full grid search. As future work, grid search should be conducted to identify even more optimal hyperparameter configurations for the classifier.

The pipeline in this work evaluates the models' performance in detecting code clones across two datasets: one containing type-1 to type-4 clones, and another containing weakly type-3 and type-4 clones. Additionally, the models' performance can be evaluated individually for each code clone type to find out if differences arise among the models within different types.

During the fine-tuning, it was found that CodeT5+ enhanced its performance significantly through fine-tuning while GraphCodeBERT didn't show any significant performance improvement. CodeT5+ showed a steep improvement at the beginning of fine-tuning, which slowed down quickly when more data was used to further fine-tune the model. Fine-tuning is a relatively resource-intensive task, and investigating the trade-offs associated with the size of the fine-tuning dataset can provide valuable insights for implementing more efficient fine-tuning strategies.

References

[1] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?", *7th International Working Conference on Mining Software Repositories*, pp. 503–530, 2010. DOI: 10.1109/MSR.2010.5463343.

[2] Y. Golubev and T. Bryksin, "On the nature of code cloning in open-source java projects", *IEEE 15th International Workshop on Software Clones*, pp. 22–28, 2021. DOI: 10.5281/zenodo.3608211.

[3] F. H. Quradaa, S. Shahzad, and R. S. Almoqbily, "A systematic literature review on the applications of recurrent neural networks in code clone research", *PLoS ONE*, vol. 19, pp. 1–40, 2024. DOI: 10.1371/journal.pone.0296858.

[4] T. Nakagawa, Y. Higo, and S. Kusumoto, "Nil: Large-scale detection of large-variance clones", *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 830–841, 2021. DOI: 10.1145/3468264.3468564.

[5] J. Svacina, J. Simmons, and T. Cerny, "Semantic code clone detection for enterprise applications", *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 129–131, 2020. DOI: 10.1145/3341105.3374117.

[6] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review", *Information and Software Technology*, pp. 1165–1199, 2013. DOI: https://doi.org/10.1016/j.infsof.2013.01.008.

[7] C. Roy and J. Cordy, “A survey on software clone detection research”, *School of Computing TR 2007-541*, pp. 1–115, 2007. [Online]. Available: https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf.

[8] W. Wu et al., “An empirical study of code clones from commercial ai code generators”, *Proceedings of the ACM on Software Engineering*, vol. 2, pp. 2874–2896, 2025. DOI: 10.1145/3729397.

[9] S. B. Ankali and L. Parthiban, “Detection and classification of cross-language code clone types by filtering the nodes of antlr-generated parse tree”, *I.J. Intelligent Systems and Applications*, vol. 11, no. 4, pp. 43–65, 2021.

[10] Y. Wang, Y. Ye, Y. Wu, W. Zhang, Y. Xue1, and Y. Liu, “Comparison and evaluation of clone detection techniques with different code representations”, *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 332–344, 2023. DOI: 10.1109/ICSE48619.2023.00039.

[11] M. Khajezade, F. H. Fard, and M. S. Shehata, “Evaluating few-shot and contrastive learning methods for code clone detection”, *Empirical Software Engineering*, pp. 1–9, 2024. DOI: 10.1007/s10664-024-10441-z.

[12] F. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection”, *IEEE Access*, pp. 86 121–86 144, 2019. DOI: 10.1109/ACCESS.2019.2918202.

[13] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, “Machine learning is all you need: A simple token-based approach for effective code clone detection”, *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–123, 2024. DOI: 10.1145/3597503.3639114.

[14] Mondego, *Mondego/sourcerercc: Sourcerer’s code clone project*, https://github.com/Mondego/SourcererCC, Accessed 08/01/2026, 2016.

[15] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?", *Proc. 31st International Conference on Software Engineering*, pp. 485–495, 2017. DOI: 10.1109/ICSE.2009.5070547.

[16] H. Kaushik and D. K. D. Gupta, "Code clone detection: An empirical study of techniques for software engineering practice", *Lampyrid*, pp. 1–23, 2023. DOI: 10.1145/3729397.

[17] H. Kaur and R. Maini, "Performance evaluation and comparative analysis of code-clone-detection techniques and tools", *International Journal of Software Engineering and Its Applications*, pp. 31–50, 2017. DOI: 10.14257/ijseia.2017.11.3.04.

[18] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees", *Proceedings of ICSM'98*, pp. 368–377, 1998. DOI: 10.1109/ICSM.1998.738528.

[19] C. Dubach, "Comp 520 compiler design: Lecture 7 – parse tree and abstract syntax", 2024, Lecture notes, McGill University, School of Computer Science.

[20] H. Murakami, *Fast and precise token-based code clone detection*, Graduate School of Information Science and Technology, Osaka University, Technical report, available online, Accessed: 03/01/2026, Jan. 2016. [Online]. Available: https://scispace.com/pdf/fast-and-precise-token-based-code-clone-detection-5cpmz8vvsn.pdf.

[21] Z. Zhang and T. Saber, "Exploring the boundaries between llm code clone detection and code similarity assessment on human and ai-generated code", *Big Data and Cognitive Computing*, pp. 1–19, 2025. DOI: https://doi.org/10.3390/bdcc9020041.

[22] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree", *2020 IEEE 27th*

International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 261–271, 2020. DOI: 10.1109/SANER48275.2020.9054857.

[23] M. B. Moumoula, A. K. Kaboré, J. Klein, and T. F. Bissyandé, "The struggles of llms in cross-lingual code clone detection", *Proc. ACM Softw. Eng.*, pp. 1023–1045, 2025. DOI: https://doi.org/10.1145/3715764.

[24] A. Kumar, R. Yadav, and K. Kumar, "A systematic review of semantic clone detection techniques in software systems", *IOP Conference Series: Materials Science and Engineering*, vol. 1022, pp. 1–11, 2021. DOI: 10.1088/1757-899X/1022/1/012074.

[25] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation", *CoRR*, pp. 1–26, 2023. DOI: 10.48550/arXiv.2305.07922.

[26] Meta AI, *Meta llama 3.2-1b model*, https://huggingface.co/meta-llama/Llama-3.2-1B, Accessed: 06/01/2026, 2024.

[27] A. Vaswani et al., *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[28] A. A. Abdullah et al., "Evolution of meta's llama models and parameter-efficient fine-tuning of large language models: A survey", *CoRR*, pp. 1–37, 2025. DOI: 10.48550/arXiv.2510.12178.

[29] Z. Liang. "Code to tokens conversion: A developer's guide". Accessed: 01/14/2026. [Online]. Available: https://prompt.16x.engineer/blog/code-to-tokens-conversion.

[30] Salesforce Research. "Codet5 and codet5+ released models". Accessed: 01/15/2026. [Online]. Available: https://github.com/salesforce/CodeT5/tree/main/CodeT5%5C#released-models.

[31] D. Guo et al., *Graphcodebert: Pre-training code representations with data flow*, 2021. DOI: 10.48550/arXiv.2009.08366.

[32] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search", *arXiv preprint arXiv:1909.09436*, pp. 1–6, 2019. DOI: 10.48550/arXiv.1909.09436.

[33] Hugging Face Datasets, *Poolc/1-fold-clone-detection-600k-5fold dataset*, https://huggingface.co/datasets/PoolC/1-fold-clone-detection-600k-5fold, Accessed: 10/01/2026, 2025.

[34] J. Martinez-Gil, "Evaluating small-scale code models for code clone detection", *CoRR*, pp. 1–20, 2025. DOI: 10.48550/arXiv.2506.10995.

[35] K. Kitsios, F. Sovrano, E. Barr, and A. Bacchelli, "Detecting semantic clones of unseen functionality", in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2025, pp. 1312–1324. DOI: 10.48550/arXiv.2510.04143.

[36] Google Research and Hugging Face, *Codexglue cc clone detection big clone benchmark*, https://huggingface.co/datasets/google/code_x_glue_cc_clone_detection_big_clone_bench, Accessed: 19/01/2026, 2021.

[37] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "Machine learning is all you need: A simple token-based approach for effective code clone detection", in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639114.

[38] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, *Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation*, 2021. DOI: 10.18653/v1/2021.emnlp-main.685.

[39] L. Talvitie, *Code-clone-detection-transformer-analysis*, https://github.com/Larppuli/code-clone-detection-transformer-analysis, GitHub repository, 2026.

[40] H. Touvron et al., *Llama: Open and efficient foundation language models*, 2023. DOI: 10.48550/arXiv.2302.13971.

[41] B. B. Lambruschini, P. Becerra-Sanchez, M. Brorsson, and M. Zurad, "Reducing tokenizer's tokens per word ratio in financial domain with TMuFin BERT tokenizer", in *Proceedings of the Fifth Workshop on Financial Technology and Natural Language Processing and the Second Multimodal AI For Financial Forecasting*, 2023, pp. 94–103. DOI: 10.18653/v1/2023.finnlp-1.9.

[42] A. Raj, *Understanding the [CLS] token in bert: A comprehensive guide*, https://aditya007.medium.com/understanding-the-cls-token-in-bert-a-comprehensive-guide-a62b3b94a941, Accessed 12/01/2026, 2024.

Appendix A Code Clone Type Examples

This appendix presents example implementations of the four code clone types discussed in Chapter 2.1 using Java functions. All example functions return the smallest common divisor of the given parameters.

Algorithm 1 Original code fragment used as the basis for all clone examples.

```
public static int smallestCommonDivisor(int a, int b){
    int limit = (a < b) ? a : b;
    for(int i = 2; i <= limit; i++){
        if(a % i == 0 && b % i == 0) return i;
    }
    return 1;
}
```

Algorithm 2 Type-1 code clone: differs from the original only in comments and white spaces.

```
public static int smallestCommonDivisor(int a, int b){
    int limit = (a < b) ? a : b; // choose smaller
    for(int i = 2; i <= limit; i++){
        if(a % i == 0 && b % i == 0) return i;
    }
    return 1;
}
```

Algorithm 3 Type-2 code clone: identifiers and data types have been modified.

```
public static int smallestCommonDivisor(int x, int y){
    float smallest = (x < y) ? x : y;
    for(int i = 2; i <= smallest; i++){
        if(x % i == 0 && y % i == 0) return i;
    }
    return 1;
}
```

Algorithm 4 Type-3 code clone: the computation logic is preserved, but the control structure has been modified.

```
public static int smallestCommonDivisor(int a, int b){
    int smallest;
    if(a < b) smallest = a;
    else smallest = b;

    for(int i = 2; i <= smallest; i++){
        if(a % i == 0 && b % i == 0) return i;
    }
    return 1;
}
```

Algorithm 5 Type-4 code clone: syntactically different but semantically equivalent to the original.

```
public static int smallestCommonDivisor(int a, int b){
    for(int i = 2; ; i++){
        boolean dividesA = (a / i) * i == a;
        boolean dividesB = (b / i) * i == b;
        if(dividesA && dividesB) return i;
        if(i > a && i > b) break;
    }
    return 1;
}
```

Appendix B Use of Generative AI

Generative artificial intelligence was used mainly in three use cases in this thesis: in code generation, in the design phase of the code clone detection pipeline, and for detecting grammar errors in text.

In code generation, generative AI was used to quickly build prototypes of independent functionalities in the code clone detection pipeline. After validating the ideas, the functionalities were developed more carefully and refined to perform exactly the desired task.

In the design phase of the pipeline, generative AI was used to explore potential structures for the pipeline. After the initial exploration, prior literature and existing implementations were reviewed to refine the final design of the pipeline.

Generative AI was also used to detect obvious grammar errors in the text, and occasionally to improve sentence structure to enhance clarity.