University of
St Andrews

RepoSim

Leveraging Pretrained Language Models for Semantic Repository Comparison

by
Zihao Li

Supervisor: Dr. Rosa Filgueira Vicente

Matriculation Number: 200007779

University of St Andrews

Abstract

In recent years, the rapid development of natural language processing (NLP) techniques has spurred their application to a wide range of fields, including software engineering. One such application is the analysis of source code for semantic similarity, which can facilitate tasks like code clone detection and repository comparison. This thesis investigates the effectiveness of state-of-the-art pretrained language models, GraphCodeBERT and UniXCoder, in performing pair-wise clone detection tasks and comparing the semantic similarity of repositories.

We fine-tuned GraphCodeBERT and UniXCoder on a dataset of Python code pairs for clone detection tasks and evaluated their performance on the C4 dataset. Furthermore, we developed RepoSim, an approach to efficiently compare the semantic similarities of repositories categorized by their related topics, and examined the performance of various language models on this task. Our evaluation results indicate that GraphCodeBERT achieved the best performance on Python clone detection tasks, while UniXCoder fine-tuned on code search tasks excelled at identifying semantically similar repositories based on their code embeddings.

The thesis provides valuable insights into the capabilities of pretrained language models in handling code-related tasks and contributes to the understanding of how these techniques can be applied to capture the semantic information of repositories. This work not only advances research in this area but also paves the way for future exploration of cutting-edge language models, such as ChatGPT, LLaMa, and Bard, in the context of repository similarity comparison and other related tasks.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 7261 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Rosa Filgueira, for providing me with the opportunity to delve into the intriguing field of machine learning through this project. Her constant guidance and support have been invaluable in helping me navigate the complexities of this research. I also want to express my appreciation to my family and friends for their unwavering support, both financially and emotionally, throughout this year. Your encouragement has been essential in helping me achieve my academic goals.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development has seen significant advancements in recent years, with code repositories growing in size and complexity. As a result, understanding the semantic similarities between different codebases has become an essential task for developers, researchers, and organizations. This not only aids in code reuse but also helps to identify potential code clones, which can lead to more efficient software maintenance and higher software quality. Machine Learning (ML) and Natural Language Processing (NLP) techniques, such as Recurrent Neural Networks and Transformers, have emerged as powerful tools to address these challenges and have been employed in various code-related tasks, such as code completion, code summarization, and code search.

In this thesis, we explore the use of ML techniques for code clone detection and repository similarity comparison. We fine-tune two pre-trained models, GraphCodeBERT and UniXCoder, on pair-wise clone detection tasks and evaluate their performance. Additionally, we develop an approach to efficiently compare the semantic similarities of repositories with different topics and test the performance of various language models' embeddings on this task. Our evaluations show that the GraphCodeBERT model excels at Python clone detection tasks, while the UniXCoder model fine-tuned on code search performs best at identifying semantically similar repositories based on their code embeddings.

The thesis is organized as follows: Chapter 2 lists the objectives set for this project and

the completion status. Chapter 3 provides a comprehensive background on ML and NLP techniques for code clone detection. Chapter 4 presents the related works and tools used for comparing repository similarities. Chapter 5 delves into the implementation of the models and the experimental setup. Chapter 6 presents the evaluation results, highlighting the performance of the models on clone detection tasks and repository similarity comparison. Chapter 7 focuses on the reproducibility of our work, detailing the steps to fine-tune the models and reproduce the evaluation results. Finally, Chapter 8 concludes the thesis, summarizing the findings and outlining future research directions.

Our work aims to contribute to the understanding of how advanced ML and NLP techniques can be applied to code-related tasks and their effectiveness in capturing the semantic information embedded within source code. By exploring the capabilities of GraphCodeBERT and UniXCoder in clone detection and repository similarity comparison, we hope to shed light on the potential of these models in various code-related tasks and inspire further research in this area.

Chapter 2

Objectives

2.1 Primary Objective

1. Develop a comprehensive survey of various machine learning approaches for code-to-code similarity:

   - Based on Abstract Syntax Trees-based models
   - Based on Python code embeddings in token-based models:
     - LSTM
     - Pre-trained Transformers
     - Sequences of Elements
   - Based on metadata and docstrings similarity
   - Based on call graphs, data flow, control flow graphs, and repository structure
   - Based on pre-trained transformer models for code understanding

   In this survey, we will discuss how these techniques work and compare their performance.

2. Select a code-to-code similarity technique and develop Python scripts to evaluate its performance when comparing Python codes.

3. Fine-tune the selected model using a public Python code dataset to investigate potential improvements in its performance.

4. Enhance `inspect4py` to capture additional metadata information from a given repository, such as Readme, Description, Topics, Tags, Forks, Stars, etc. Personal information that identifies users will not be extracted during this process, and we will use repositories with open source licenses only.

5. Validate the correctness of the extracted metadata from a selection of public repositories, ensuring that the metadata matches the purpose of the source code. A selection of repositories will be manually annotated and compared to the metadata extracted by `inspect4py`. We aim to validate the metadata extracted using over 400 open-source repositories.

2.2 Secondary Objectives

1. Train or fine-tune a model for evaluating metadata similarity between repositories. The dataset used for training will come from metadata extracted by the extended `inspect4py` developed in the primary objectives.

2. Utilize this model to evaluate the similarity of open-source repositories based on their metadata.

2.3 Tertiary Objectives

1. Train or fine-tune a model for evaluating similarity between repositories based on their structure. The structure of each repository will also be extracted by `inspect4py`.

2. Use this model to evaluate the similarity of open-source repositories based on their structure. Again, we plan to use the same collection of repositories specified in the Repo2Vec paper[1] to evaluate the performance of our model.

[1]Md Omar Faruk Rokon et al. *Repo2Vec: A Comprehensive Embedding Approach for Determining*

3. Train a model that evaluates similarity based on each repository's metadata, structure, and code similarity, and compare its performance with the previous models.

2.4 Objective Completion Status

In this thesis, we have covered all aspects of the Primary and Secondary Objectives, except for the context survey which is submitted as a separate document, and the validation of the correctness of the extracted metadata from repositories. The metadata validation was not implemented, as we are using the official GitHub API to perform metadata extraction, there is little need for validating the correctness of the content.

As for the Tertiary Objectives, we have decided to discard the use of repository structure and focus solely on metadata and source code of the Python repositories. Later evaluations show that we have achieved high performance on comparing repository similarities based on these two components.

Repository Similarity. 2021. DOI: 10.48550/ARXIV.2107.05112. URL: https://arxiv.org/abs/2107.05112.

Chapter 3

Foundation

Detecting and analyzing code similarities is becoming increasingly important in the field of software engineering, as it can have a significant impact on code quality, maintainability, and reliability. In recent years, various code similarity tools have been developed, utilizing different code representations like the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and deep learning techniques like Recurrent Neural Networks (RNN) and Transformers.

In this chapter, we will provide an in-depth understanding of the different techniques used to determine code similarities. We will begin by defining the types of clones and introducing the common code representations. Next, we will introduce widely used neural network models for processing various types of sequences, such as sentences, paragraphs, and even code snippets. We will also explore the evolution of these models over time, from the initial development of Recurrent Neural Networks for natural language processing tasks to the recent advancements in language model architectures like BERT, RoBERTa, and CodeBERT. Finally, we will introduce state-of-the-art models used in this project, and explore techniques for optimizing their performance.

3.1 Types of Clone

As proposed in a survey[1], we can divide types of code clones into four categories:

- Type I: Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.
- Type II: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.
- Type III: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.
- Type IV: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

In this project, our aims is to detect Type IV clones, i.e. code pairs that are semantically similar but may be syntactically different.

3.2 Abstract Syntax Tree

Abstract Syntax Tree or AST in short is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text[2]. It contains semantics and syntactic information of the program, therefore is commonly used to analyze the similarity between source codes.

[1] Chanchal Roy and James Cordy. "A Survey on Software Clone Detection Research". In: *School of Computing TR 2007-541* (Jan. 2007).

[2] Wikipedia contributors. *Abstract syntax tree — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1103626323`. [Online; accessed 26-October-2022]. 2022.

3.3 Control Flow Graph

Control flow graph or CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution[3]. It captures more comprehensive semantic information such as branch, loop and calling relationshops(represented by call graph) in the code, but much coarser syntactic information compared to AST[4]. It can be combined with AST to better represent the source code.

3.4 Embedding

In the context of machine learning, embedding is the process of representing high-dimensional data, such as words, images, or user preferences, in a lower-dimensional vector space. This technique is commonly used in natural language processing tasks, where words or phrases are represented as vectors. By using neural network models like Word2Vec or BERT to learned from large amounts of text data, we can encode texts into vectors such that similar texts are close to each other in the vector space, whereas dissimilar texts are far apart.

Programming languages share many similarities with natural language, Therefore, some techniques used in natural language processing, including embedding, can also be applied to programming languages for measuring code similarity.

3.5 Cosine Similarity

Cosine similarity is a measure of similarity between two non-zero vectors in a multidimensional space. It measures the cosine of the angle between the two vectors, which ranges from -1 to 1, where 1 indicates that the vectors are identical, 0 indicates that they are orthogonal (perpendicular), and -1 indicates that they are diametrically opposite.

[3] Wikipedia contributors. *Control-flow graph — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=1115609094`. [Online; accessed 25-October-2022]. 2022.

[4] Chunrong Fang et al. "Functional Code Clone Detection with Syntax and Semantics Fusion Learning". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 516–527. ISBN: 9781450380089. DOI: 10.1145/3395363.3397362. URL: https://doi.org/10.1145/3395363.3397362.

The formula for cosine similarity between two n-dimensional vectors A and B is given by[5]:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Cosine similarity can serve as a metric for measuring semantic similarity by encoding the meaning of words or sentences as vectors in a high-dimensional space based on their occurrence or co-occurrence statistics in a corpus of text, and each dimension in the vector space may represent a feature or aspect of the meaning.

3.6 Recurrent Neural Network

Recurrent Neural Network or RNN in short is a type of neural network that is designed to handle sequential data. Unlike most traditional feedforward neural networks, which process inputs one by one and produce an output, RNNs have a recurrent structure where connections between nodes can create a cycle, allowing output from some nodes to affect subsequent input to the same nodes[6]. This helps the network to maintain an internal state, or memory, that captures information about the sequence it has seen so far, which enables it to handle sequential data or time series data more effectively.

Before transformers came out, RNN, especially Long Short-Term Memory (LSTM) networks were the primary workhorse for natural language processing (NLP) tasks. However, RNNs have several limitations, including difficulties in capturing long-term dependencies, the vanishing gradient problem, and the sequential nature of computations leading to slow training and inference times. Since RNNs depend on previous output to compute the next output during training, they cannot take advantage of parallel computing, which slows down the training process.

[5]Wikipedia contributors. *Cosine similarity — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Cosine_similarity&oldid=1141784488`. [Online; accessed 25-March-2023]. 2023.

[6]Wikipedia contributors. *Recurrent neural network — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Recurrent_neural_network&oldid=1117752117`. [Online; accessed 28-October-2022]. 2022.

3.7 Transformers

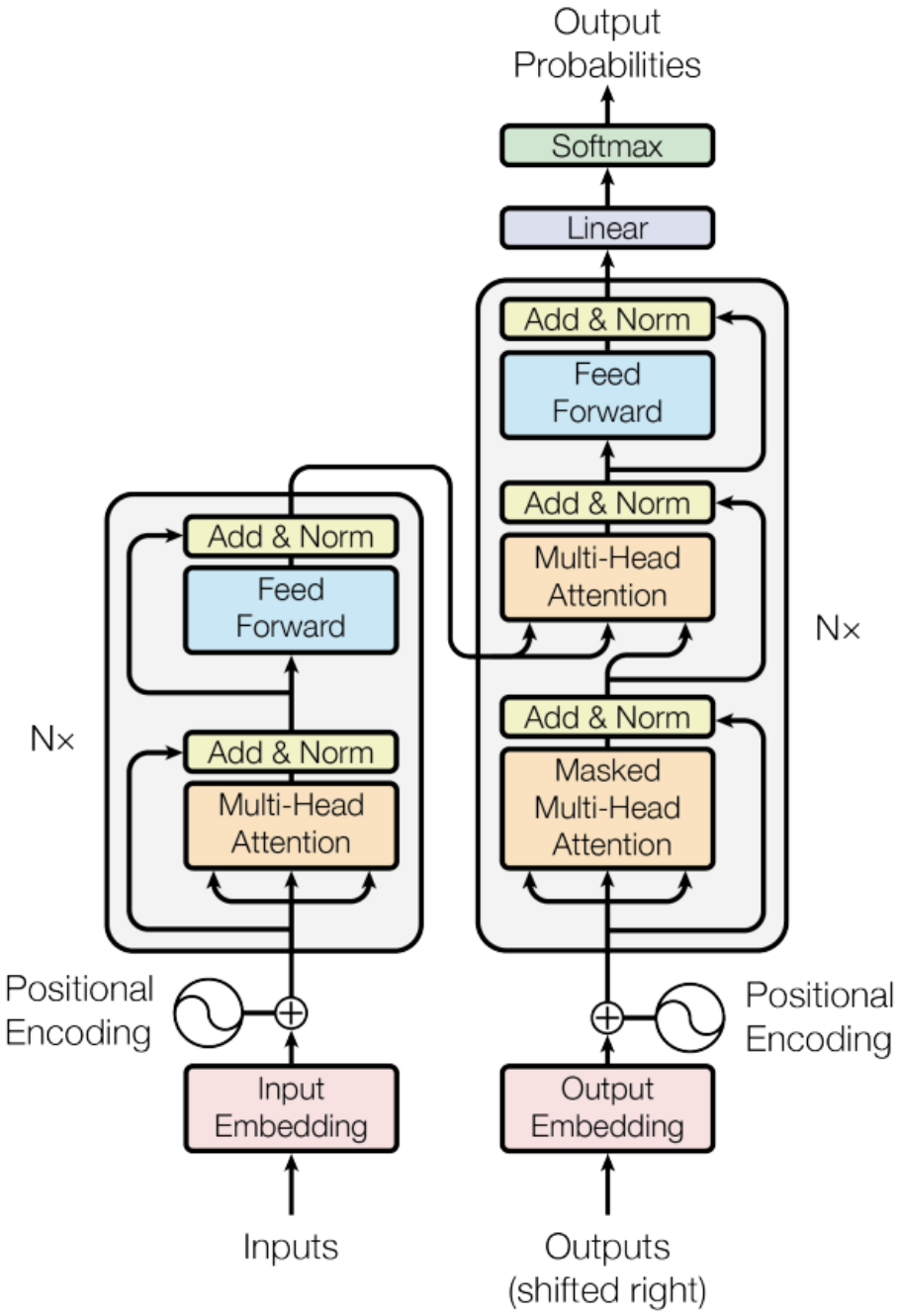

Figure 3.1: Transformer Architecture[7]

[7] [8]

A transformer is a deep learning model that leverages the self-attention mechanism to differentially weight the significance of each part of the input data[8]. It is also designed to process sequential data, but unlike RNNs which process input sequence word by word, it can process input data in parallel, which takes advantage of the modern GPU's parallel computing power and makes the training process more efficient. It has two essential mechanism:

- **Positional encoding** This involves adding a vector representing the positional information of each element (e.g., word, token) in the sequence to its corresponding embedding, so that the resultant embeddings are order-aware. By using positional encoding, transformers can process sequences in parallel without losing information about the sequence order.

- **Self-attention** This enables transformers to effectively model relationships between elements in a sequence, by dynamically weighting the importance of each element based on its relevance to other elements within the same sequence. It also significantly reduces the computational complexity per layer and increases the amount of computation that can be parallelized[9].

Nowadays transformers are increasingly the model of choice for NLP problems, pretrained transformer models, such as BERT and GPT, have played a significant role in advancing the field of natural language processing.

3.8 BERT

Bidirectional Encoder Representations from Transformers or BERT in short is a transformer-based machine learning technique for natural language processing(NLP) pre-training de-

[8]Wikipedia contributors. *Transformer (machine learning model) — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Transformer_(machine_learning_model)&oldid=1118251522`. [Online; accessed 28-October-2022]. 2022.

[9]Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: `10.48550/ARXIV.1706.03762`. URL: `https://arxiv.org/abs/1706.03762`.

veloped by Google[10]. It has two pre-training objectives: masked language model and next sentence prediction. The masked language model randomly masks some words in the input sequence and the model is trained to predict the masked words. The next sentence prediction task is to predict if the second sentence is the continuation of the first sentence. Both tasks can be trained in unsupervised fashion which enables BERT to be pre-trained on large datasets like BooksCorpus and English Wikipedia[11]. The pre-trained BERT model is able to have certain level of understanding in the language and can be fine-tuned for specific downstream tasks.

3.9 RoBERTa

RoBERTa (Robustly Optimized BERT pre-training Approach)[12] is a modified version of BERT developed by Facebook in 2019. RoBERTa is trained using a larger corpus and for a longer period of time compared to BERT, which leads to improved performance on various NLP tasks.

3.10 CodeBERT

CodeBERT[13] is a pre-trained model for programming language that uses the same architecture as RoBERTa. It is a multi-programming-lingual model pre-trained on NL-PL pairs in six programming languages, including Python, Java, JavaScript, PHP, Ruby, and Go. During the pre-training phase, CodeBERT is trained on both bimodal and unimodal data, where bimodal data refers to natural language-code pairs and unimodal data refers to natural language or code only. CodeBERT has two objectives, Masked Language Modeling

[10]Wikipedia contributors. *BERT (language model) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=BERT_(language_model)&oldid=1107671243. [Online; accessed 31-October-2022]. 2022.

[11]Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[12]Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. DOI: 10.48550/ARXIV.1907.11692. URL: https://arxiv.org/abs/1907.11692.

[13]Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. DOI: 10.48550/ARXIV.2002.08155. URL: https://arxiv.org/abs/2002.08155.

(as described in the previous section) and Replaced Token Detection. In the Replaced Token Detection objective, some input tokens are replaced with plausible alternatives generated from a generator, and the goal is to train a model that can determine whether each token in the corrupted input was replaced by a generator sample or not[14]. After pre-training and fine-tuning, CodeBERT achieves state-of-the-art performance on both natural language code search and code documentation generation.

3.11 GraphCodeBERT

GraphCodeBERT[15] is a pre-trained language model developed by Microsoft Research Asia for source code processing. It is an extension of the original CodeBERT model.

GraphCodeBERT incorporates a graph neural network (GNN) architecture to encode both the syntactic and semantic information in the source code, which allows it to better capture the structural relationships between code elements. Specifically, GraphCodeBERT leverages abstract syntax trees (ASTs) to represent the code and builds a graph structure over the ASTs. The nodes of the graph represent code tokens (e.g., variables, functions, operators), and the edges represent the syntactic relationships between these tokens (e.g., parent-child relationships in the AST).

GraphCodeBERT is pre-trained on a large-scale corpus of source code and can be fine-tuned on a variety of downstream tasks, including code classification, code completion, and code retrieval. The pre-training task for GraphCodeBERT are the masked language modeling (MLM) task as described before and the Edge Prediction task. In the Edge Prediction task, the model learns representations from data flow by training to predict edges between masked variables in the data flow.

At the time of its publication, GraphCodeBERT had achieved state-of-the-art performance on several code-related downstream tasks including code search, clone detection,

[14]Kevin Clark et al. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020. DOI: 10.48550/ARXIV.2003.10555. URL: https://arxiv.org/abs/2003.10555.

[15]Daya Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. 2020. DOI: 10.48550/ARXIV.2009.08366. URL: https://arxiv.org/abs/2009.08366.

code translation and code refinement. For this project, a modified GraphCodeBERT model was fine-tuned and employed as a clone detection tool.

3.12 R-Drop: Regularized Dropout for Neural Networks

The regularization strategy known as **R-Drop**[16] is an extension of the dropout technique. It has been shown to yield substantial improvements when applied to fine-tune large-scale pre-trained models such as BERT and RoBERTa.

In this project, we utilized a custom clone detection model[17] based on GraphCodeBERT that had been optimized with the R-Drop technique. The model was originally developed by the **snoop2head** team and had demonstrated outstanding performance in a Python code similarity competition, where it ranked in the top three on the leaderboard[18].

3.13 Contrastive learning

Contrastive learning is a machine learning technique that enhances the quality of data representations by contrasting positive examples (similar data points) against negative examples (dissimilar data points) in a latent space. By pulling semantically close neighbors together and pushing non-neighbors apart, contrastive learning enables language models to learn better semantic representations[19].

This technique is also effectively used in programming language models, such as C4[20], SYNCOBERT[21] and UniXCoder[22], to extract code semantics into embeddings. These

[16] Xiaobo Liang et al. *R-Drop: Regularized Dropout for Neural Networks*. 2021. DOI: 10.48550/ARXIV.2106.14448. URL: https://arxiv.org/abs/2106.14448.

[17] Park SangHa. *sangHa0411/CloneDetection*. https://github.com/sangHa0411/CloneDetection. 2022.

[18] *Monthly Dacon Code Similarity Judgment AI Contest*. May 2022. URL: https://dacon.io/competitions/official/235900/leaderboard.

[19] Tianyu Gao, Xingcheng Yao, and Danqi Chen. *SimCSE: Simple Contrastive Learning of Sentence Embeddings*. 2021. DOI: 10.48550/ARXIV.2104.08821. URL: https://arxiv.org/abs/2104.08821.

[20] Chenning Tao et al. "C4: Contrastive Cross-Language Code Clone Detection". In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 2022, pp. 413–424. DOI: 10.1145/3524610.3527911.

[21] Xin Wang et al. *SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation*. 2021. DOI: 10.48550/ARXIV.2108.04556. URL: https://arxiv.org/abs/2108.04556.

[22] Daya Guo et al. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. 2022. DOI:

models had strong performance in tasks involving semantic similarity comparison between various programming languages and natural language, such as code search and clone detection, as indicated by their evaluations.

3.14 UniXCoder

UnixCoder is another transformer-based pre-trained model which leverages cross-modal contents including AST and code comment to enhance code representation[23].

To better represent semantics of code fragments by embeddings, the authors introduced two pre-training tasks: multi-modal contrastive learning(MCL) that utilizes AST to acquire more comprehensive code semantic representations and cross-modal generation(CMG) that utilizes code comment to align embeddings among programming languages.

UniXcoder surpasses previous state-of-the-art models, including CodeBERT, GraphCodeBERT, SYNCOBERT, and CodeT5, in clone detection and code search tasks, as demonstrated by experiments in the paper. The authors also proposed a new task called zero-shot code-to-code search to evaluate the performance of code fragment embeddings. In this task, a source code is used as a query to retrieve codes with the same semantics from a collection of candidates in a zero-shot setting. UniXcoder achieved state-of-the-art performance and outperformed GraphCodeBERT with an approximately 11-point improvement in the overall score. The paper's ablation studies indicate that contrastive learning and the use of multi-modality contributed the most to the model's improvement.

Given UniXCoder's strong performance in extracting code semantic information and its applicability across various programming languages, we will employ it in this project to extract code embeddings and conduct cosine similarity comparisons between them.

10.48550/ARXIV.2203.03850. URL: https://arxiv.org/abs/2203.03850.

[23]Daya Guo et al. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. 2022. DOI: 10.48550/ARXIV.2203.03850. URL: https://arxiv.org/abs/2203.03850.

Chapter 4

Related works / Tools

4.1 inspect4py

Inspect4py[1] is a static code analysis framework designed to help developers understand and classify Python software repositories, by analyzing and extracting important information such as functions, classes, methods, documentation, dependencies, call graphs, and control flow graphs. It is useful for understanding, managing and maintaining software repositories.

This project includes extending functionalities of **inspect4py** and allow it to extract additional metadata information from a given repository, such as Readme, Description, Topics, Tags, Forks, Stars, etc, which can be helpful in finding repositories that share similar characteristics, such as authorship, topic, description, and more.

The tool's ability to extract function source code and documentation strings is leveraged to convert input repositories into lists of function source code and docstrings. Deep learning models can then be used to calculate embeddings for each code or docstring in these lists.

[1] Rosa Filgueira and Daniel Garijo. "Inspect4py: A Knowledge Extraction Framework for Python Code Repositories". In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 2022, pp. 232–236. DOI: 10.1145/3524842.3528497.

4.2 Repo2Vec

Repo2Vec[2] is an approach proposed to represent a GitHub repository in an embedding vector using information from three types of sources: metadata, directory structure, and source code. The approach follows a pipeline that consists of four steps:

1. **meta2vec** Metadata embedding is computed by mapping the metadata of a repository into an embedding vector using the **doc2vec** model. The metadata includes information such as the repository name, description, and language.

2. **struct2vec** Directory structure embedding is computed by following three steps: First, the directory structure is transformed into a tree representation. Second, vector of each node is generated using the **node2vec** model. Lastly, node vectors are synthesized into a single structure vector using column-wise aggregation methods to represent directory structure.

3. **source2vec** Source code embedding is computed also by following three steps: First, generate vector of each method using the **code2vec** model. Second, aggregate all method code vectors in each file to get file vectors. Lastly, aggregates all file vectors to get the overall code vectors.

4. **repo2vec** Finally, repository embedding is computed by concatenating the embeddings obtained from the previous steps. The weights of each embedding type are determined by the user and can be adjusted to give more or less importance to certain information sources.

The evaluation of Repo2Vec demonstrates the effectiveness of leveraging information from multiple sources to effectively represent a repository, it also highlights the importance of including metadata embeddings for the overall performance. In line with this approach, our project aims to incorporate both code embedding similarity and documentation embedding similarity when comparing two repositories. By considering multiple sources of

[2]Md Omar Faruk Rokon et al. *Repo2Vec: A Comprehensive Embedding Approach for Determining Repository Similarity*. 2021. DOI: 10.48550/ARXIV.2107.05112. URL: https://arxiv.org/abs/2107.05112.

information, we can gain a more comprehensive understanding of a repository's content and ultimately improve our ability to accurately compare similar repositories.

Chapter 5

Implementation

In this section, we present our implementation process for comparing semantic similarity between Python source codes and repositories. First, we discuss two approaches: a cross-encoder technique using a modified GraphCodeBERT model, and a bi-encoder method employing a UniXCoder model. Next, we outline the fine-tuning process of these models using specific datasets and provide a detailed analysis of their strengths and weaknesses. Subsequently, we describe our methods for comparing repository similarity, emphasizing the advantages of the bi-encoder approach in terms of efficiency. Finally, we explain the selection of the best performing docstring embeddings model based on our evaluation of pre-trained sentence transformer models.

5.1 Code Similarity Comparison

In this project, we utilized two pre-trained language models to compare the semantic similarity between Python source codes. Specifically, we employed a modified GraphCodeBERT model that was fine-tuned on a clone detection task, as well as a UniXCoder model that was fine-tuned on a natural language code search task. These models produce similarity results using two different paradigms: cross-encoder and bi-encoder, both of which are detailed in the following subsection.

5.1.1 Cross-Encoder and Bi-Encoder

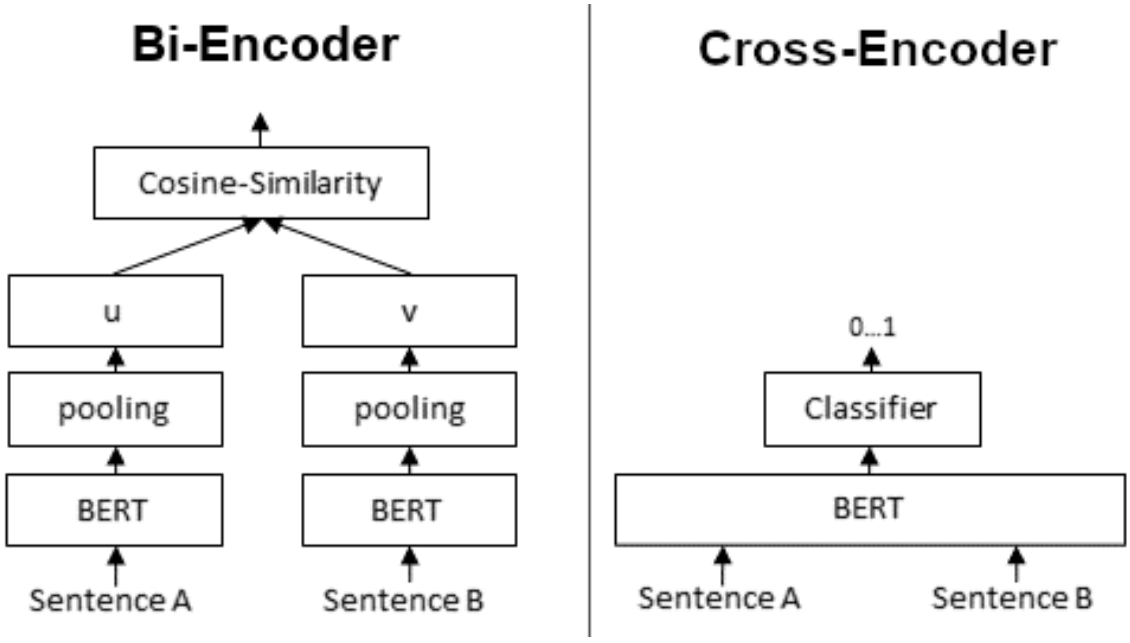

Figure 5.1: Bi-Encoder vs. Cross-Encoder[1]

When comparing the semantic similarity of sentences, two types of architectures are commonly used: Bi-Encoders and Cross-Encoders. Each architecture has its own strengths and weaknesses.

Bi-Encoders encode each sentence independently and produce sentence embeddings that can later be compared using a similarity function like cosine similarity. As they only need to encode each sentence once, they are well-suited for tasks involving large numbers of sentence comparisons, such as clustering and information retrieval.

Cross-encoders, on the other hand, take in pairs of sentences and concatenate them before encoding. In clone detection tasks, the output embeddings from the encoder are classified using a linear classifier to determine whether the sentence pair is a clone or not. This architecture typically performs better than bi-encoder for semantic similarity comparisons, as it can capture the global context of both input sentences during the encoding process, while bi-encoder only considers the local context as they encode each

[1][24]

sentence separately.

5.1.2 Approach 1: GraphCodeBERT as a Cross-Encoder

Unlike Java or C, which already have existing clone detection benchmarks (datasets with confirmed clone pairs) such as BigCloneBench[2] and POJ-104[3], Type IV clone datasets of Python code are scarce. We eventually managed to find two datasets that contain labeled code pairs:

1. **5fold dataset**: We initially discovered a code similarity contest hosted by **Dacon**[4], an AI hackathon platform in Korea. This contest aimed to find the model with the best performance in detecting semantically similar Python codes. The **snoop2head** team ranked among the top three in this contest, with a final score of 0.98061. For reproducibility, they uploaded their models[5] to GitHub and the fine-tuning dataset[6] to Hugging Face Hub. The dataset contains $5,388,622$ labeled code pairs in the training set and $1,324,360$ in the testing set.

2. **C4 dataset**: We also found another smaller dataset that contains semantically similar clone pairs for multiple programming languages. This dataset was originally used in the research paper **C4: Contrastive Cross-Language Code Clone Detection**[7] for detecting clones between different programming languages. We filtered the dataset so that it only contains Python clone pairs, and then generated negative

[2] Jeffrey Svajlenko et al. "Towards a Big Data Curated Benchmark of Inter-Project Code Clones". In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME '14. USA: IEEE Computer Society, 2014, pp. 476–480. ISBN: 9781479961467. DOI: 10.1109/ICSME.2014.77. URL: https://doi.org/10.1109/ICSME.2014.77.

[3] Lili Mou et al. "Convolutional neural networks over tree structures for programming language processing". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 2016, pp. 1287–1293.

[4] *Monthly Dacon Code Similarity Judgment AI Contest*. May 2022. URL: https://dacon.io/competitions/official/235900/leaderboard.

[5] Park SangHa. *sangHa0411/CloneDetection*. https://github.com/sangHa0411/CloneDetection. 2022.

[6] Young Jin Ahn. *Clone-Detection-600k-5fold*. Hugging Face Datasets. Contact email: young_ahn@yonsei.ac.kr. 2022. URL: https://huggingface.co/datasets/PoolC/1-fold-clone-detection-600k-5fold.

[7] Chenning Tao et al. "C4: Contrastive Cross-Language Code Clone Detection". In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 2022, pp. 413–424. DOI: 10.1145/3524610.3527911.

samples by randomly selecting non-clone codes for the first code in each positive clone pair. This resulted in a balanced dataset with $6,852$ positive samples and an equal number of negative samples.

Fine-tuning on the 5fold dataset We utilized the modified GraphCodeBERT model and the dataset uploaded by the **snoop2head** team in an effort to reproduce their work. Due to limited time and resources, we only fine-tuned the model on the first fold of the 5fold dataset using an A40 GPU server for approximately 11.5 hours. We then selected the checkpoint with the highest evaluation score for subsequent experiments.

5.1.3 Approach 2: UniXCoder as a Bi-Encoder

Although the GraphCodeBERT model demonstrated effectiveness in later clone-detection evaluations, we found that employing it for comparing large repositories with numerous functions was impractical.

To address this issue, we opted for a bi-encoder approach. While our fine-tuned GraphCodeBERT model is adept at determining whether an input sentence pair is a clone or not, its performance is not as strong when used as a bi-encoder—i.e., encoding each code separately and comparing the cosine similarity of their output embeddings. Therefore, UniXCoder's ability to generate better embeddings containing semantic information through contrastive learning makes it a more preferable option over the previous GraphCodeBERT model.

UniXCoder - Fine-tuned on Clone Detection Task

Since the fine-tuning script provided by the author of UniXCoder is less efficient than the one used for the custom GraphCodeBERT model, a smaller number of code pairs were used in fine-tuning. As a result, we fine-tuned UniXCode on approximately 13,000 balanced code pairs from the 5fold dataset for around 10 hours on an A40 server.

Although the model is fine-tuned using the cross-encoder paradigm, it still demonstrates promising performance when scoring semantically similar code pairs from the C4

dataset. However, when deploying it to compare overall function semantics in repositories, it is less effective than the model introduced next.

UniXCoder - Fine-tuned on Code Search Task

The previously mentioned models are both trained on the clone detection task, which fine-tunes the model using a cross-encoder paradigm. To investigate whether a model fine-tuned in a bi-encoder paradigm is better at generating individual code embeddings, we then fine-tuned UniXCoder on a code search task, which aims to find semantically similar codes given a natural language query.

We fine-tuned UniXCoder using the provided fine-tuning script[8] on the AdvTest[9] dataset, which is sourced from CodeSearchNet[10]. This process took around 3 hours on an A40 server for one epoch.

We then compared the UniXCoder fine-tuned on the clone detection task with this model. Both models exhibited good performance when comparing code pairs using a bi-encoder approach, but the latter model produced significantly better embeddings when comparing repository similarity. More details are provided in the Evaluation section.

5.2 Comparing Similarity Between Repositories

Another important task in this project is to compare and identify semantically similar repositories. By combining the results of source code semantic comparison and metadata comparison, we can represent the semantic information of repositories as embeddings and compare their similarity, as demonstrated in **Repo2Vec**[11]. Here, we used documentation strings extracted by **inspect4py** as the metadata component of each repository. We

[8]Daya Guo et al. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. 2022. DOI: 10.48550/ARXIV.2203.03850. URL: https://arxiv.org/abs/2203.03850.

[9]Shuai Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *CoRR* abs/2102.04664 (2021).

[10]Hamel Husain et al. "Codesearchnet challenge: Evaluating the state of semantic code search". In: *arXiv preprint arXiv:1909.09436* (2019).

[11]Md Omar Faruk Rokon et al. *Repo2Vec: A Comprehensive Embedding Approach for Determining Repository Similarity*. 2021. DOI: 10.48550/ARXIV.2107.05112. URL: https://arxiv.org/abs/2107.05112.

adopted a straightforward approach by extracting all functions and docstrings into one-dimensional sets, respectively, and comparing function sets and docstring sets between repositories in some way.

5.2.1 Cross-Encoder and Hungarian Algorithm

The Hungarian algorithm was originally proposed as a method to efficiently solve the assignment problem[12]. In our case, it is a useful tool for finding a one-side-perfect matching between two sets. This means that each element from the smaller set is assigned to a unique element in the larger set, such that the total similarity score of the assignment is maximized, based on the given similarity scores for each matching between the two sets.

We used our GraphCodeBERT model to calculate similarity scores for every unique pair in the Cartesian product of two function sets. Then, we applied the Hungarian algorithm to find the assignment with the maximum total similarity score.

We tested this method on two GitHub repositories that are both collections of various algorithm implementations in Python. The results showed that although this method successfully identified many code pairs that implemented the same algorithm, there are some inevitable issues that cannot be ignored:

1. **Inefficiency** The greatest issue with this approach is its inefficiency. For two repositories with m and n unique functions, respectively, it would require computing similarity scores for $m \cdot n$ function pairs. Usually, the GraphCodeBERT model takes more than 0.7 seconds to predict one code pair on an NVIDIA T4 Tensor Core GPU. In this case, if we have two repositories, each with 100 unique functions, scoring all pairs will take nearly 2 hours. This means the approach is impractical to use, as repositories with more than 100 functions are common in practice.

2. **Poor estimation of probability** As the GraphCodeBERT model uses a linear classifier to perform binary classification on the output embeddings, the similarity

[12]Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

score produced by applying the softmax function on its output logits tends to have either very high scores (e.g., 0.9997) for positive cases, or very low scores (e.g., 0.01) for negative cases. This not only results in unreliable uncertainty estimations but also shows high confidence even when the prediction is incorrect. At the time of writing this report, we have considered using probability calibration[13] to see if it improves the model's probability estimations, but due to time constraints, this will be left for future study.

3. **Some optimal matches are sacrificed** In order to find an assignment with the maximum similarity score, the Hungarian algorithm may sacrifice some locally optimal matches in favor of a higher global match score. As a result, some algorithms may be matched with completely dissimilar algorithms.

5.2.2 Bi-Encoder Approach: RepoSim

As the cross-encoder approach is not scalable, we propose RepoSim: an approach using the bi-encoder paradigm, which involves the following steps:

1. Extract all functions and docstrings of a repository into a function set and a docstring set respectively, using inspect4py.

2. Use UniXCoder to obtain each function's code embeddings and average them to compute the mean code embeddings.

3. Use a docstring embedding model to acquire each docstring's sentence embeddings and compute their mean embeddings.

4. Determine the code similarity score and docstring similarity score between two repositories by calculating the cosine similarity of their mean code embeddings and their mean docstring embeddings.

[13] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: `https://scikit-learn.org/stable/modules/calibration.html#calibration`.

A visualization of converting a repository into its mean code embeddings and mean docstring embeddings is shown in Figure 5.2.

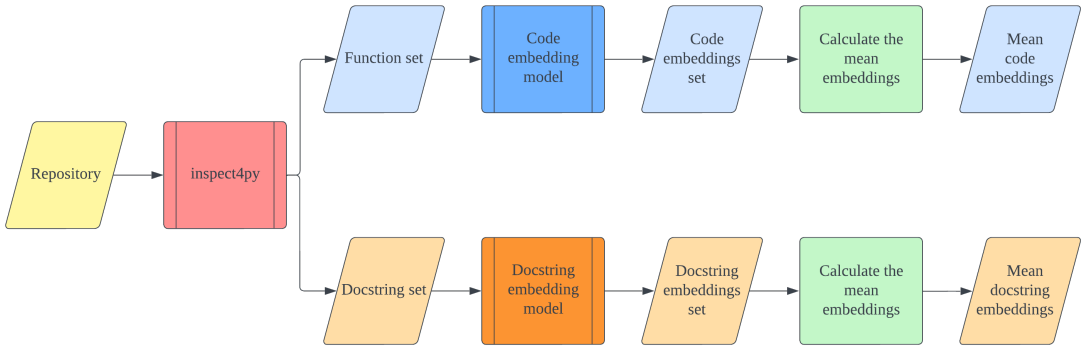

Figure 5.2: Repository to embeddings

This approach is significantly more efficient than the previous one, as each function and each docstring only need to be encoded once, and comparing cosine similarities between embeddings takes far less time than running language models to generate them. This allows us to compare more repositories within a reasonable timeframe. In the later evaluation phase, this method also demonstrated promising results at grouping repositories with various topics and providing reasonable similarity scores.

Choice of docstring embeddings model

We selected a range of well-known and high-performing pre-trained sentence transformer models. By having them generate embeddings for docstrings in each repository and testing their performance with receiver operating characteristic, we determined the best-performing docstring embeddings model to be UniXCoder fine-tuned on code search task. More details can be found in the **Evaluation** section.

Chapter 6

Evaluation

In this evaluation section, we investigate the performance of various models in determining Type IV Python clones and the effectiveness of our proposed approaches, including the Hungarian Algorithm and RepoSim in determining repository similarity. We evaluate models such as custom GraphCodeBERT and UniXCoder on clone detection tasks, and assess different programming language and natural language models for repository similarity using cosine similarity and AUC metric. The results provide insights into the models' performance and the implications of our findings, and also address some limitations and suggesting potential future research directions.

This section focuses on two main aspects:

1. Evaluating the performance of models in determining type IV Python clones.
2. Assessing the effectiveness of our approaches in determining repository similarity.

To quantify the performance in these two tasks, we need to define metrics for each task.

6.1 Metrics used in evaluations

6.1.1 Precision, Recall, and F1 scores

Precision, recall, and F1 scores are common metrics employed to evaluate the performance of classification or ranking models. These metrics are calculated based on the number of

true positives, false positives, true negatives, and false negatives. Here is a definition of these terms:

- True Positive (TP): A positive instance that is correctly classified as positive by the model.
- False Positive (FP): A negative instance that is incorrectly classified as positive by the model.
- True Negative (TN): A negative instance that is correctly classified as negative by the model.
- False Negative (FN): A positive instance that is incorrectly classified as negative by the model.

Precision, recall, and F1 can be calculated using the following formulas:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

In summary, precision is a measure of the model's accuracy in predicting positive instances, recall (also known as sensitivity) assesses how well the model identifies positive instances, and the F1 score is the harmonic mean of precision and recall, which is useful when both precision and recall are important.

The significance of precision and recall depends heavily on the specific tasks. For instance, in code plagiarism detection, we want as few false positives as possible, making precision more important. In this project, we primarily focus on the F1 score when evaluating our model's performance.

6.1.2 ROC and AUC

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a classification model at various classification thresholds[1]. It plots the following two parameters:

- **True Positive Rate (TPR)**: The proportion of positive instances that are correctly classified as positive. It is the same as recall (or sensitivity) defined above:

$$TPR = \frac{TP}{TP + FN}$$

- **False Positive Rate (FPR)**: The proportion of negative instances that are incorrectly classified as positive. It is calculated using the following formula:

$$FPR = \frac{FP}{FP + TN}$$

The number of positive and negative predictions can be varied by changing the classification threshold. The lower the threshold, the more instances are classified as positive, thus increasing both true positive and false positive cases. An example of an ROC curve is shown below:

[1] *Classification: Roc curve and AUC — machine learning — google developers.* July 2022. URL: `https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc`.

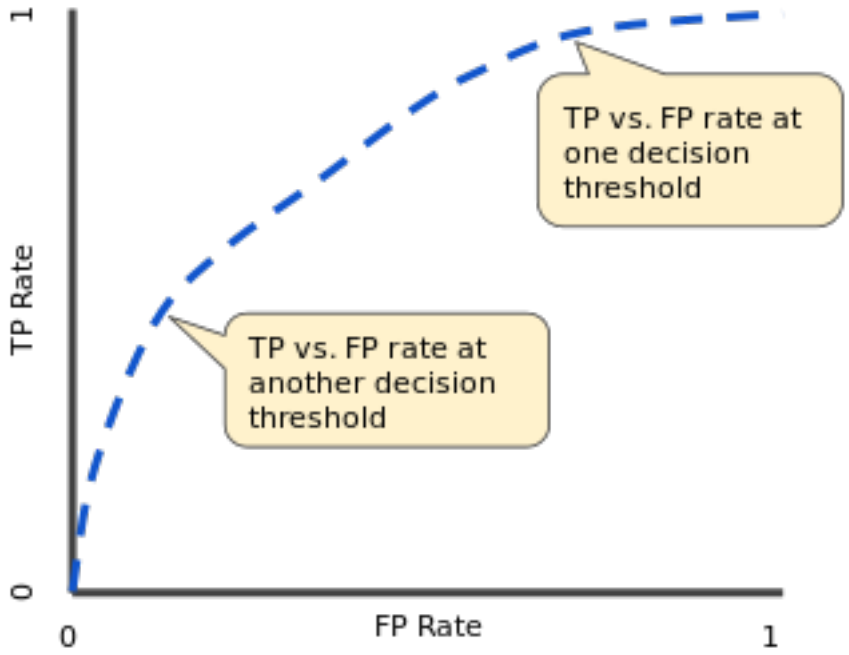

Figure 6.1: ROC curve[2]

The Area Under the ROC curve (AUC), as the name suggests, is the area under an ROC curve ranging from 0 to 1, where 1 represents a perfect classifier and 0.5 represents a random classifier. It can be used to summarize the performance of a binary classification model across all possible classification thresholds. In this project, we will use AUC, as this metric allows us to compare different classifiers and find an optimal threshold for making a binary decision about whether the repositories are similar or not.

6.2 Evaluations on Type IV python clone detection

Model	Fine-tuned task	Clone Detection			
		Recall	Precision	F1-score	AUC
GraphCodeBERT	Clone Detection	96.42	97.56	96.99	—
UniXcoder	Clone Detection	93.64	96.34	94.97	98.5
UniXcoder	Code Search	80.98	85.96	83.4	90.4

Table 6.1: Evaluations on clone detection task with C4 dataset

[2][32]

The most common metric for measuring a model's performance on clone detection tasks is precision, recall, and F1 scores. In this project, we evaluated all of our models, including the custom GraphCodeBERT fine-tuned for clone detection tasks, the UniXCoder fine-tuned for clone detection tasks, and the UniXCoder fine-tuned for code search tasks, on the C4 dataset[3] to test their performance on type IV clone detection tasks.

Our GraphCodeBERT can be used as a binary classifier and generate True/False predictions for each code pair in the C4 dataset. For the two UniXCoder models, we generate predictions by calculating cosine similarity scores for each code pair, then find an optimal threshold with the highest F1 score and use it to classify each code pair by their similarity scores. The evaluation results can be found in Table 6.1.

Since we used a bi-encoder paradigm to perform the clone detection task, the F1 scores of the two UniXCoder models are not as good as the GraphCodeBERT, which is a cross-encoder. However, we can still see they have relatively good performance, especially for the UniXCoder fine-tuned on the clone detection task, whose F1 score is very close to the GraphCodeBERT model. Furthermore, we also drew ROC curves and calculated the AUC for the two UniXCoder models, as seen in Table 6.1 and Figure 6.2. We can consider both classifiers to be good as they both have achieved an AUC higher than 0.9.

[3]Chenning Tao et al. "C4: Contrastive Cross-Language Code Clone Detection". In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 2022, pp. 413–424. DOI: 10.1145/3524610.3527911.

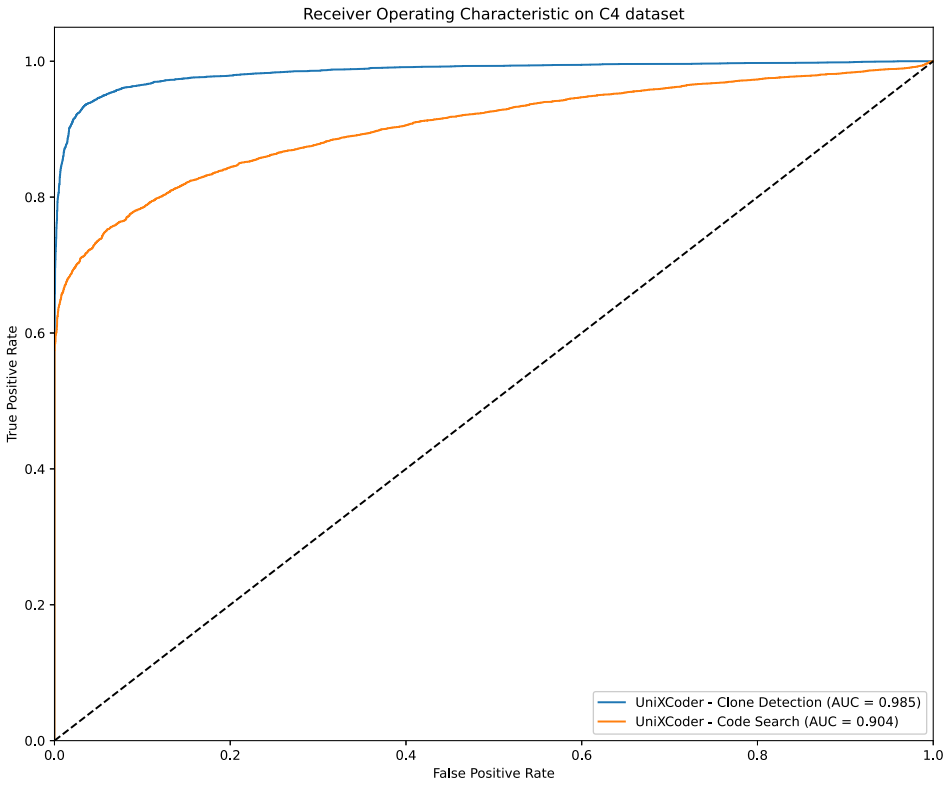

Figure 6.2: ROC curve of each UniXCoder model evaluated on clone detection tasks on C4 dataset

The difference between their cosine similarity score distributions is presented in Appendix B.

6.3 Evaluations on comparing repository similarities

6.3.1 GraphCodeBERT and Hungarian algorithm

We used our GraphCodeBERT model and applied the Hungarian algorithm to compare two GitHub repositories: **keon/algorithms**[4] and **TheAlgorithms/Python**[5]. These

[4]Keon. *Algorithms*. `https://github.com/keon/algorithms`. 2022.
[5]TheAlgorithms. *Python*. `https://github.com/TheAlgorithms/Python`. 2022.

two repositories are especially suitable for semantic comparison, as different implementations of the same algorithm share the same intent or semantics, but their code could look very different.

The resulting assignment, including the similarity score for each match, is stored in a CSV file and is uploaded within the submission. Part of this CSV file is shown in Figure 6.3, and we can see many implementations of the same algorithm are matched together in this assignment.

	name1	name2	docs1
2	gcd	euclidean_gcd	gcd Euclid's Algorithm. gcd{a,b}=gcd{$-a$,b}=gcd{a,$-b$}=gcd{$-a$,$-b$} See proof: https://proofwiki.org/wi
3	num_perfect_squares	perfect_square	num_perfect_squares Returns the smallest number of perfect squares that sum to the specified num
4	_l2_distance	euclidean	_l2_distance Calculate l2 distance from two given vectors.
5	cosine_similarity	cosine_similarity	cosine_similarity Calculate cosine similarity between given two vectors
6	magic_number	is_magic_gon	magic_number Checks if n is a magic number
7	solve_chinese_remainder	chinese_remainder_theorem2	solve_chinese_remainder for a system of equations. The system of equations has the form: x % nums
8	extended_gcd	extended_gcd	extended_gcd Return s, t, g such that num1 * s + num2 * t = GCD(num1, num2) and s and t are co-pri
9	modular_inverse	modular_exponential	modular_inverse a and m must be coprime
10	decimal_to_binary_util	decimal_to_binary	decimal_to_binary_util Convert 8-bit decimal number to binary representation
11	decimal_to_binary_ip	local_binary_value	decimal_to_binary_ip Convert dotted-decimal ip address to binary representation with help of decima
12	euler_totient	euler_modified	euler_totient Time Complexity: O(sqrt(n)).
13	base_to_int	base85_encode	base_to_int Note : You can use int() built-in function instead of this.
14	modular_exponential	naive_cut_rod_recursive	modular_exponential Time complexity - O(log n) Use similar to Python in-built function pow.
15	get_primes	sieve_er	get_primes Using sieve of Eratosthenes.
16	power	combination_sum_iv	power Calculate a ^ n if mod is specified, return the result modulo mod Time Complexity : O(log(n)) S
17	power_recur	_modexpt	power_recur Calculate a ^ n if mod is specified, return the result modulo mod Time Complexity : O(lo
18	lcm	lcm	lcm Computes the lowest common multiple of integers a and b.
19	gcd_bit	get_bitcode	gcd_bit Similar to gcd but uses bitwise operators and less error handling.
20	find_next_square2	check1	find_next_square2 Alternative method, works by evaluating anything non-zero as True (0.000001 -->
21	gen_strobogrammatic	line_length	gen_strobogrammatic Given n, generate all strobogrammatic numbers of length n.
22	recursive_binomial_coefficient	binomial_coefficient	recursive_binomial_coefficient Time complexity is O(k), so can calculate fairly quickly for large values
23	generate_key	get_bit	generate_key k is the number of bits in n
24	is_strobogrammatic2	str_eval	is_strobogrammatic2 Another implementation.
25	prime_check	validate	prime_check Else return False.
26	find_order	gcd	find_order Find order for positive integer n and given integer a that satisfies gcd(a, n) = 1.
27	find_primitive_root	quadratic_roots	find_primitive_root Returns all primitive roots of n.
28	factorial	pollard_rho	factorial If mod is not None, then return (n! % mod) Time Complexity - O(n)
29	factorial_recur	factorial_recursive	factorial_recur If mod is not None, then return (n! % mod) Time Complexity - O(n)
30	find_nth_digit	get_digits	find_nth_digit 1. find the length of the number where the nth digit is from. 2. find the actual number w
31	cycle_product	vol_spheres_union	cycle_product cycle index of a symmetry group), compute the resultant monomial in the cartesian pr

Figure 6.3: Assignment by Hungarian Algorithm

Due to the inefficiency of this method, we do not have enough predictions to provide an accurate evaluation of its performance. Therefore, we will skip this part.

6.3.2 Evaluations of RepoSim using different models

Our repository samples are obtained from the **awesome-python**[6] list. This list categorizes hundreds of Python repositories by their related topics, such as **Algorithms, Audio, Authentication, Job Scheduler, Natural Language Processing, Machine Learning**, and so on. They can be used as ground truth, indicating semantically similar repositories.

We generated the mean embeddings for each repository's functions and docstrings, following the process depicted in Figure 5.2, using different programming language and natural language models. For each possible pair of repositories, we calculated their cosine similarity. Finally, to evaluate the performance of each model, we used the AUC metric.

Unlike other natural language models, UniXCoder is a cross-modal pre-trained model capable of producing both docstring and function embeddings. Figure 6.4 displays the ROC curves for all models, while Table 6.2 summarizes their AUC scores.

[6]Vinta. *Awesome Python*. `https://github.com/vinta/awesome-python`. 2023.

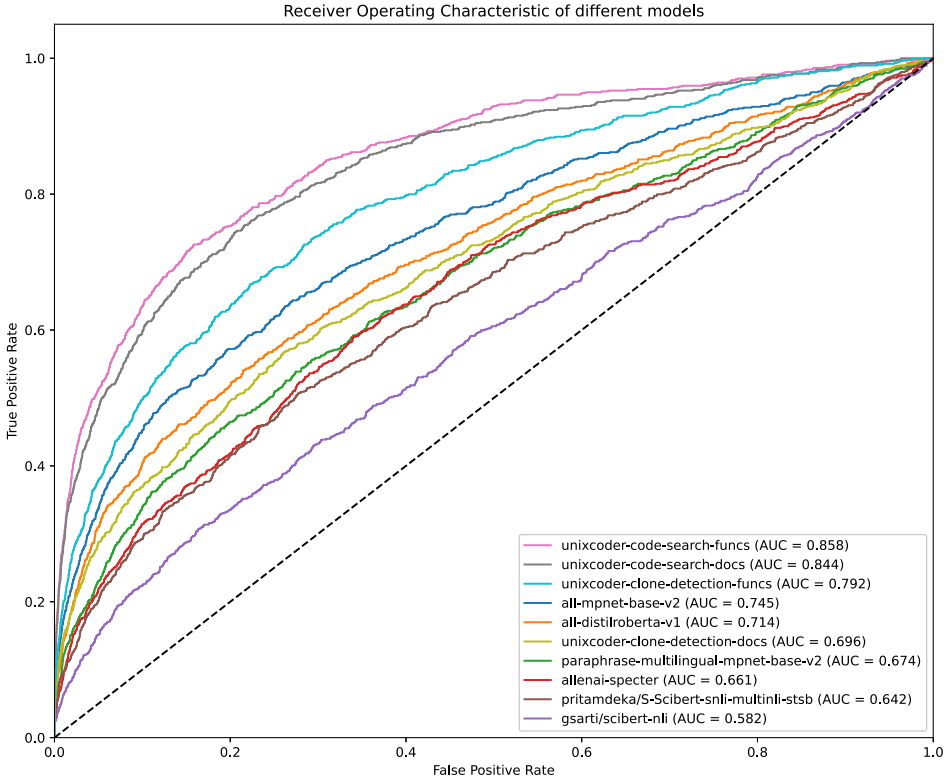

Figure 6.4: ROC curves of different models in repository similarity comparison

Model	AUC	
	Docstring embeddings	Code embeddings
gsarti/scibert-nli	58.2	—
pritamdeka/S-Scibert-snli-multinli-stsb	64.2	—
allenai-specter	66.1	—
paraphrase-multilingual-mpnet-base-v2	67.4	—
all-distilroberta-v1	71.4	—
all-mpnet-base-v2	74.5	—
unixcoder-clone-detection	79.2	69.6
unixcoder-code-search	**84.4**	**85.8**

Table 6.2: AUC of different models in repository embeddings comparison

The evaluation results show that our UniXCoder model, fine-tuned on the code search task, has the best performance in comparing repository similarities.

6.4 Visualization of the best-performing embeddings

Figure 6.5 presents a visualization of code embeddings generated by UniXCoder, fine-tuned on the code search task, after reducing them to two dimensions using a dimension reduction algorithm called t-SNE[7]. In this figure, each scatter represents a repository and is labeled with its name, while the color of the scatters represents a topic.

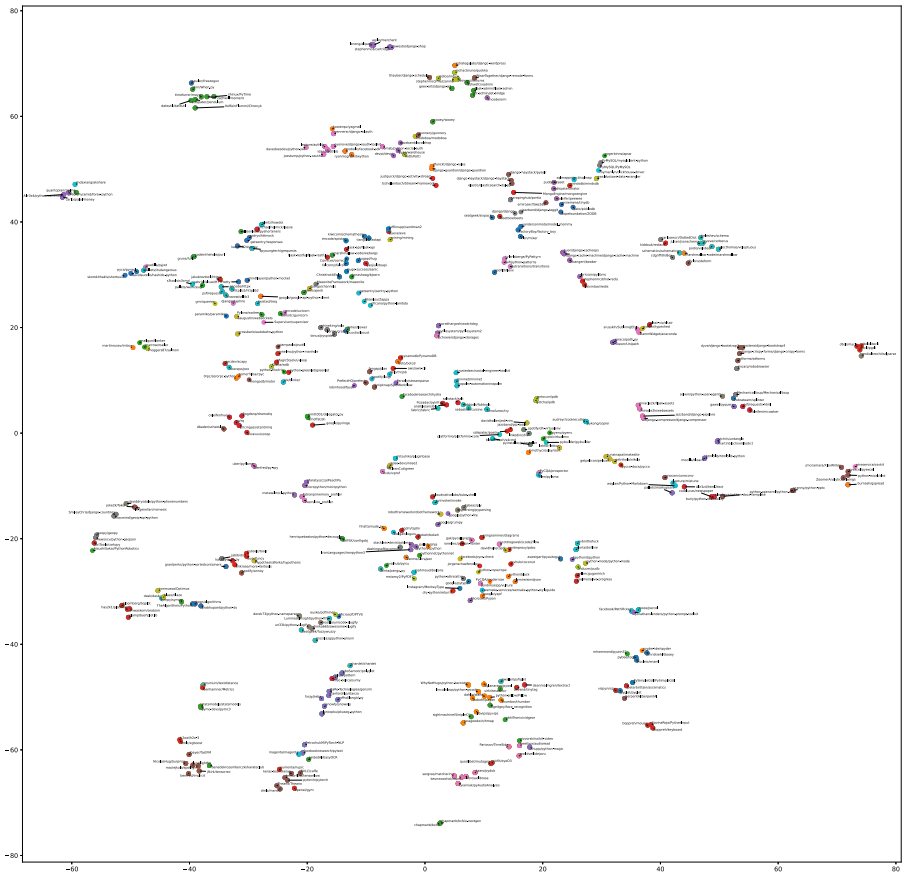

Figure 6.5: T-SNE 2D visualization of repository mean code embeddings

[7]Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". in: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: `http://jmlr.org/papers/v9/vandermaaten08a.html`.

Some clusters of repositories with similar topics from this visualization are included in Appendix A.

Chapter 7

Reproducibility

Our hardware used for fine-tuning models:

Component	Specification
GPU	A40 (48GB) × 1
CPU	15 vCPU AMD EPYC 7543 32-Core Processor

7.1 Fine-tuning GraphCodeBERT model on clone detection

The fine-tuning script for GraphCodeBERT is available in a GitHub repository by the original author: `https://github.com/snoop2head/CloneDetection`. To run the training script, first install the necessary requirements, and create a **wandb**(`https://wandb.ai`) account to track hyperparameters, model predictions, and other related data during fine-tuning. Next, update the `train.py` script as follows:

1. Remove the parameter `use_auth_token=True` on line 44.
2. Assign a `WANDB_AUTH_KEY` obtained from **wandb** either directly in the source code or through the shell environment variable on line 131.
3. Replace the value of the `entity` parameter on line 160 with your entity name created on **wandb**.

The training script will automatically download the 5fold dataset and store it in a cache directory. To specify a cache directory, add the following lines at line 2 of the training script:

```
import os
os.environ['TRANSFORMERS_CACHE'] = <your_cache_dir>
os.environ['HF_HOME'] = <your_cache_dir>
import wandb
```

Figure 7.1: Fine-tuning GraphCodeBERT

As depicted in Figure 7.1, the model was trained on the first fold of the 5fold dataset for approximately 11.5 hours on an A40 server. For later evaluations on the C4 dataset, we selected **checkpoint-22000**, as it achieved the best evaluation accuracy of 0.984 during

fine-tuning.

7.2 Fine-tuning UniXCoder model on clone detection

The UniXCoder author provided a fine-tuning script for clone detection tasks: `https://github.com/microsoft/CodeBERT/tree/master/UniXcoder/downstream-tasks/clone-detection/BCB`. However, we need to prepare our own Python dataset since the original script uses **BigCloneBench**, a Java clone dataset. To fine-tune UniXCoder on clone detection with the 5fold dataset, we have uploaded a Python script that converts it into a **dataset** directory for the fine-tuning script.

Due to the 5fold dataset's size, the fine-tuning script was unable to process all training data without exceeding our GPU RAM, requiring us to reduce the training size to $\frac{1}{10}$. Training took over 10 hours on our A40 GPU server.

7.3 Fine-tuning UniXCoder model on code search

Fine-tuning UniXCoder on code search is more straightforward, as both the fine-tuning script and Python dataset are provided. To reproduce, follow the AdvTest part of the data-download and fine-tune setting sections in the original source: `https://github.com/microsoft/CodeBERT/tree/master/UniXcoder/downstream-tasks/code-search`. Figure 7.2 shows the process of training UniXCoder on our A40 server, which took less than 3 hours to complete.

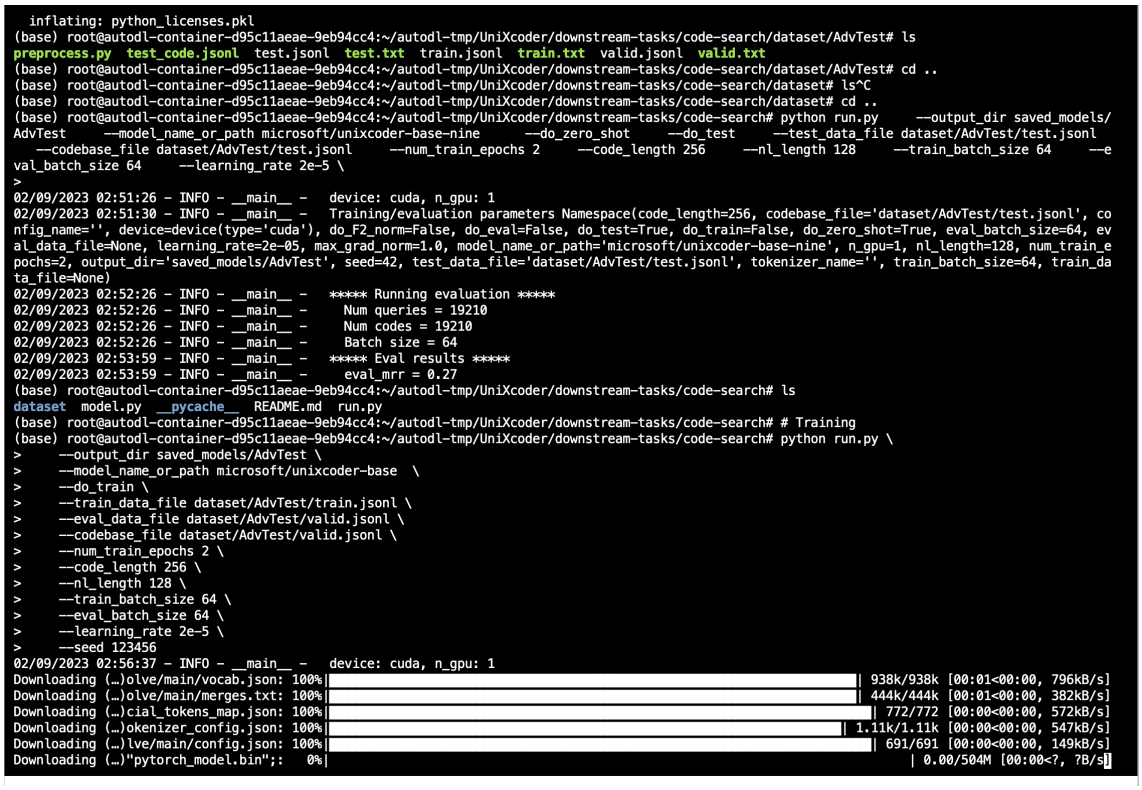

Figure 7.2: Fine-tuning UniXCoder on Code Search task

7.4 Reproducing repository similarity comparisons

We have uploaded our work as Jupyter notebooks to GitHub. The `notebooks` directory contains two sub-directories: `Cross-Encoder` and `Bi-Encoder`, each featuring notebooks for generating and evaluating embeddings using the respective approaches described in the **Implementation** and **Evaluation** sections.

Our evaluation results can be replicated using notebooks that run directly on Google Colab (`https://colab.research.google.com`). For more information about the content of this repository, please refer to the `README.md` file in the submission's root directory.

Chapter 8

Conclusion and Future Works

This thesis has investigated the evolution of NLP models, from Recurrent Neural Networks to the latest advancements in Transformers and their extensions. We fine-tuned two pre-trained programming language models, GraphCodeBERT and UniXCoder, and assessed their performance in pair-wise clone detection tasks. Additionally, we devised an approach to efficiently compare the semantic similarities of repositories with varying topics and evaluated the performance of different language models' embeddings in this context. Our evaluations revealed that the GraphCodeBERT model exhibited the best performance in Python clone detection tasks, while the UniXCoder model fine-tuned on code search tasks excelled at identifying semantically similar repositories based on their code embeddings.

For future work, we plan to further refine our GraphCodeBERT model's similarity scores by applying probability calibration techniques to normalize them. We also intend to explore whether code search tasks are better suited for extracting semantic information from source code compared to clone detection tasks.

Furthermore, the recent emergence of large language models such as ChatGPT, LLaMa, and Bard has revolutionized natural language processing, enabling more powerful understanding and generation capabilities for tasks like code similarity comparison, code completion, code summarization, and more. We are eager to investigate their potential by applying our method to the code embeddings generated by these models and evaluating

their performance in repository semantic similarity comparison. This research could provide valuable insights into the effectiveness of advanced language models for this task and contribute to the progress of the field.

Appendix A

Embeddings clusters

This appendix section presents figures of some clusters in the 2D visualization of all repositories' mean code embeddings. Many repositories within the same cluster share similar intent compared to others:

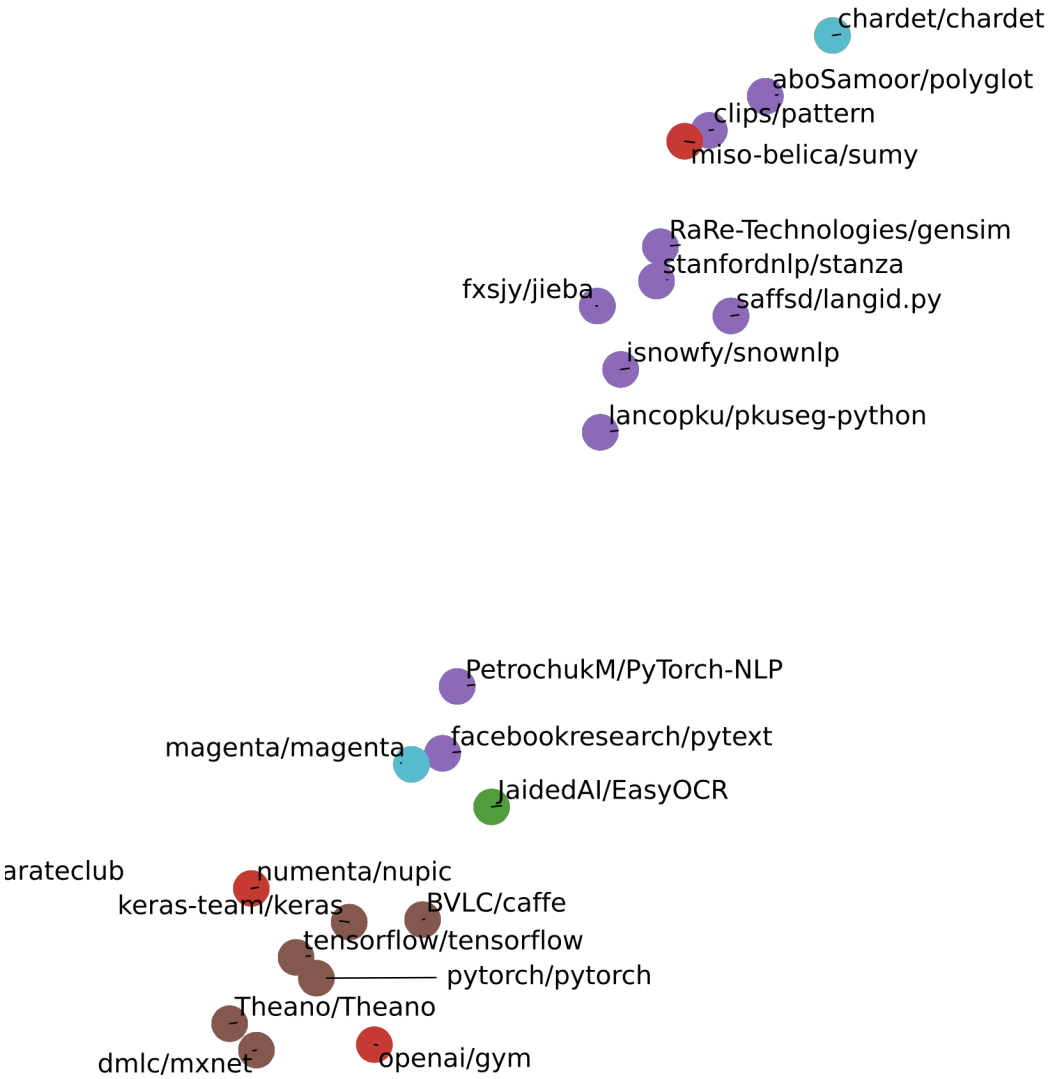

Figure A.1: Machine Learning and NLP repositories cluster

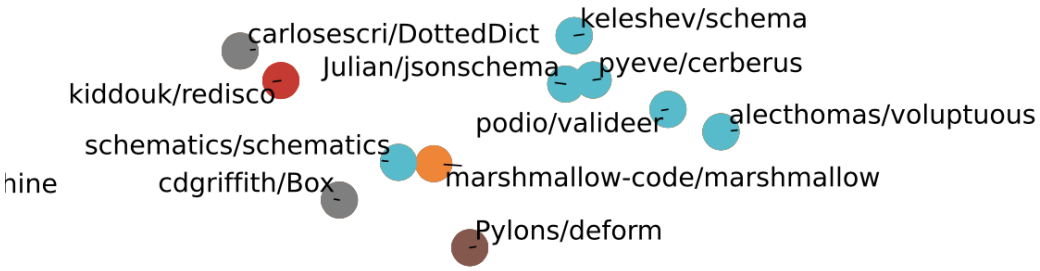

Figure A.2: Validation/schema repositories cluster

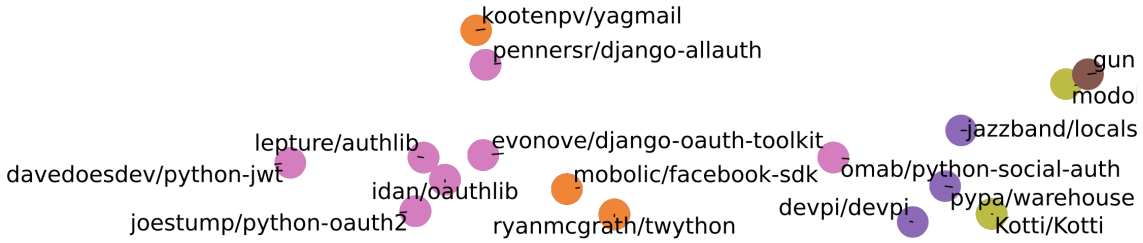

Figure A.3: OAuth repositories cluster

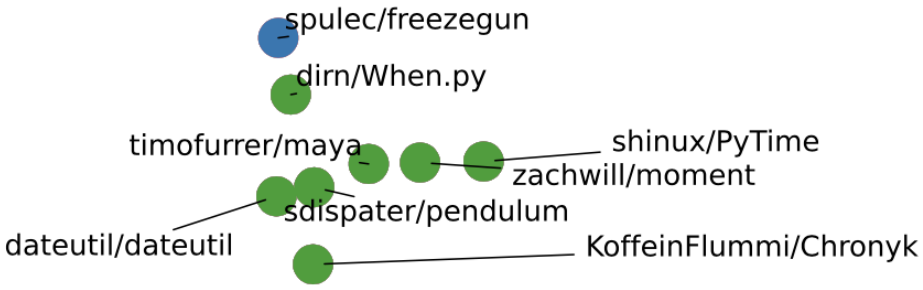

Figure A.4: Date/time repositories cluster

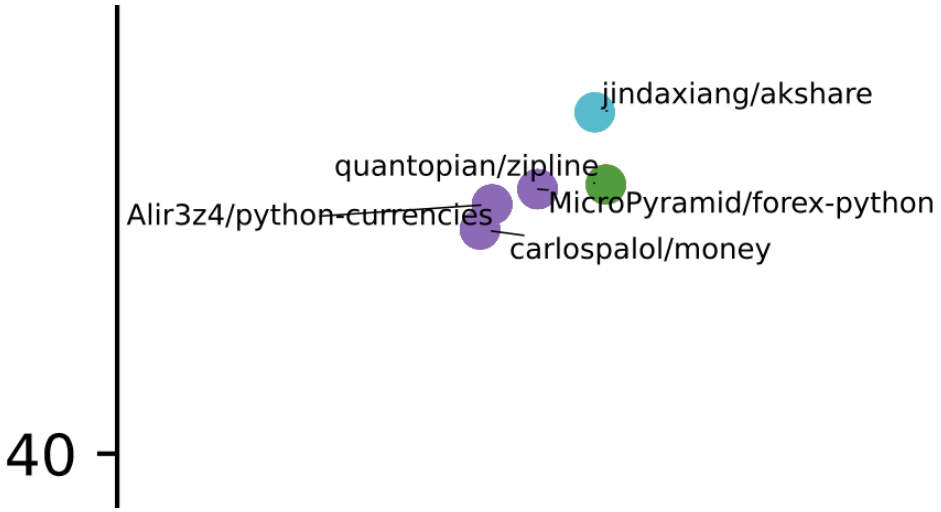

Figure A.5: Currency repositories cluster

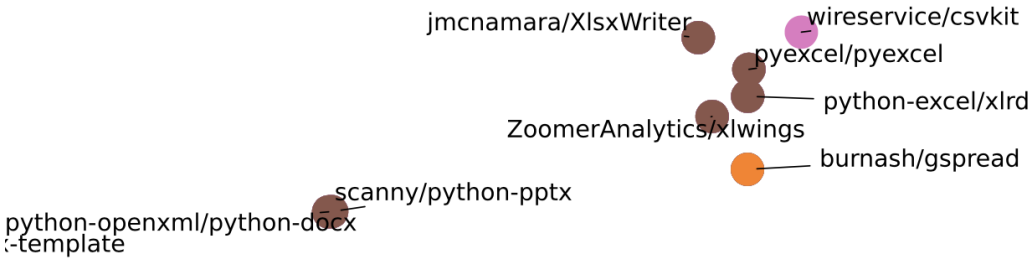

Figure A.6: Spreadsheet repositories cluster

Despite the limitation of assigning only a single topic to each repository, the embedding visualization reveals hidden relationships between repositories belonging to different topics through their clusters. For instance, the repository **wireservice/csvkit** is labeled as belonging to the topic "CSV", while **burnash/gspread** is labeled as belonging to the topic "Third-party APIs". Nevertheless, they are both clustered with repositories that work with Excel documents, which is logical since they can all be used to process spreadsheets.

Appendix B

Cosine Similarity Distribution

Figure B.1 illustrates the cosine similarity scores distribution of the two UniXCoder models. In comparison to the clone detection model, which has a polarized similarity score distribution, the code search model exhibits a more uniform similarity distribution and is less likely to give very low similarity scores.

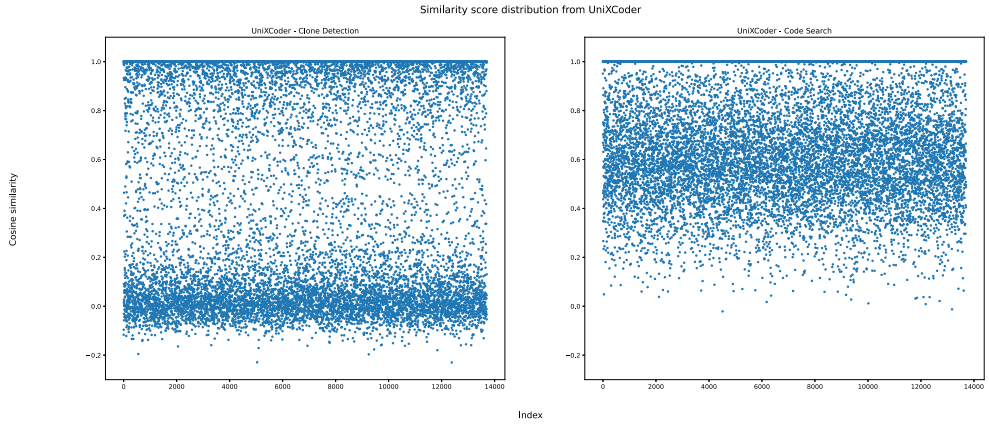

Figure B.1: Cosine similarity Distribution of two UniXCoder models

Bibliography

[1] Md Omar Faruk Rokon et al. *Repo2Vec: A Comprehensive Embedding Approach for Determining Repository Similarity*. 2021. DOI: 10.48550/ARXIV.2107.05112. URL: https://arxiv.org/abs/2107.05112.

[2] Chanchal Roy and James Cordy. "A Survey on Software Clone Detection Research". In: *School of Computing TR 2007-541* (Jan. 2007).

[3] Wikipedia contributors. *Abstract syntax tree — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1103626323. [Online; accessed 26-October-2022]. 2022.

[4] Wikipedia contributors. *Control-flow graph — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=1115609094. [Online; accessed 25-October-2022]. 2022.

[5] Chunrong Fang et al. "Functional Code Clone Detection with Syntax and Semantics Fusion Learning". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 516–527. ISBN: 9781450380089. DOI: 10.1145/3395363.3397362. URL: https://doi.org/10.1145/3395363.3397362.

[6] Wikipedia contributors. *Cosine similarity — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Cosine_similarity&oldid=1141784488. [Online; accessed 25-March-2023]. 2023.

[7] Wikipedia contributors. *Recurrent neural network — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Recurrent_neural_network&oldid=1117752117. [Online; accessed 28-October-2022]. 2022.

[8] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[9] Wikipedia contributors. *Transformer (machine learning model) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Transformer_(machine_learning_model)&oldid=1118251522. [Online; accessed 28-October-2022]. 2022.

[10] Wikipedia contributors. *BERT (language model) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=BERT_(language_model)&oldid=1107671243. [Online; accessed 31-October-2022]. 2022.

[11] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[12] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. DOI: 10.48550/ARXIV.1907.11692. URL: https://arxiv.org/abs/1907.11692.

[13] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. DOI: 10.48550/ARXIV.2002.08155. URL: https://arxiv.org/abs/2002.08155.

[14] Kevin Clark et al. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020. DOI: 10.48550/ARXIV.2003.10555. URL: https://arxiv.org/abs/2003.10555.

[15] Daya Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. 2020. DOI: 10.48550/ARXIV.2009.08366. URL: https://arxiv.org/abs/2009.08366.

[16] Xiaobo Liang et al. *R-Drop: Regularized Dropout for Neural Networks*. 2021. DOI: 10.48550/ARXIV.2106.14448. URL: https://arxiv.org/abs/2106.14448.

[17] Park SangHa. *sangHa0411/CloneDetection*. https://github.com/sangHa0411/CloneDetection. 2022.

[18] *Monthly Dacon Code Similarity Judgment AI Contest*. May 2022. URL: https://dacon.io/competitions/official/235900/leaderboard.

[19] Tianyu Gao, Xingcheng Yao, and Danqi Chen. *SimCSE: Simple Contrastive Learning of Sentence Embeddings*. 2021. DOI: 10.48550/ARXIV.2104.08821. URL: https://arxiv.org/abs/2104.08821.

[20] Chenning Tao et al. "C4: Contrastive Cross-Language Code Clone Detection". In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 2022, pp. 413–424. DOI: 10.1145/3524610.3527911.

[21] Xin Wang et al. *SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation*. 2021. DOI: 10.48550/ARXIV.2108.04556. URL: https://arxiv.org/abs/2108.04556.

[22] Daya Guo et al. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. 2022. DOI: 10.48550/ARXIV.2203.03850. URL: https://arxiv.org/abs/2203.03850.

[23] Rosa Filgueira and Daniel Garijo. "Inspect4py: A Knowledge Extraction Framework for Python Code Repositories". In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 2022, pp. 232–236. DOI: 10.1145/3524842.3528497.

[24] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: https://arxiv.org/abs/1908.10084.

[25] Jeffrey Svajlenko et al. “Towards a Big Data Curated Benchmark of Inter-Project Code Clones”. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME ’14. USA: IEEE Computer Society, 2014, pp. 476–480. ISBN: 9781479961467. DOI: 10.1109/ICSME.2014.77. URL: https://doi.org/10.1109/ICSME.2014.77.

[26] Lili Mou et al. “Convolutional neural networks over tree structures for programming language processing”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 2016, pp. 1287–1293.

[27] Young Jin Ahn. *Clone-Detection-600k-5fold*. Hugging Face Datasets. Contact email: young_ahn@yonsei.ac.kr. 2022. URL: https://huggingface.co/datasets/PoolC/1-fold-clone-detection-600k-5fold.

[28] Shuai Lu et al. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation”. In: *CoRR* abs/2102.04664 (2021).

[29] Hamel Husain et al. “Codesearchnet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).

[30] Harold W. Kuhn. “The Hungarian Method for the Assignment Problem”. In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

[31] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://scikit-learn.org/stable/modules/calibration.html#calibration.

[32] *Classification: Roc curve and AUC — machine learning — google developers*. July 2022. URL: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc.

[33] Keon. *Algorithms*. https://github.com/keon/algorithms. 2022.

[34] TheAlgorithms. *Python*. https://github.com/TheAlgorithms/Python. 2022.

[35] Vinta. *Awesome Python*. https://github.com/vinta/awesome-python. 2023.

[36] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☑ **Undergraduate Project**

Title of project

Exploring ML solutions for calculating Python Code Similarity

Name of researcher(s)

Zihao Li

Name of supervisor (for student research)

Rosa Filgueira

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☑ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

Zihao Li

Print Name

ZIHAO LI

Date

22/09/2022

Signature Lead Researcher or Supervisor

Print Name

Rosa Filgueira

Date

22/09/2022

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Research with secondary datasets

Please check UTREC guidance on secondary datasets (https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/secondary-data/ and https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/confidentiality-data-protection/). Based on the guidance, does your project need ethics approval?

YES ☐ **NO** ☑

** If your research involves secondary datasets, please list them with links in DOER.*

Research with human subjects

Does your research involve collecting personal data on human subjects?

YES ☐ **NO** ☑

If YES, full ethics review required

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☑

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☐

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ **NO** ☑

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/