School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Simplified Transformers in Code Completion: An Exploration of Performance Dynamics

Michael David Wiciak

Submitted in accordance with the requirements for the degree of
BSc Computer Science with AI

2023/2024

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (07/05/2024)
Link to online code repository	URL	Sent to supervisor and assessor (07/05/2024)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Michael Wiciak

Summary

This paper delves into the realm of code completion, specifically focusing on Masked Language Modelling (MLM) tasks within the context of software development. Code completion, a pivotal feature in enhancing developer productivity, involves predicting and suggesting code snippets based on the existing context, thus reducing manual typing efforts. The study scrutinizes various models' performance dynamics in this domain, with a particular emphasis on Transformers, which are renowned for their prowess in natural language processing tasks. Notably, the investigation underscores the significance of pre-trained models fine-tuned for MLM objectives, where models like CodeBERT-base exhibit superior accuracy in code completion tasks across diverse programming languages, showcasing the efficacy of supervised learning approaches.

The findings also shed light on the nuanced performance nuances across different models. While CodeComments, leveraging CodeBERT-base, emerges as a standout performer in overall accuracy, ASTComments, powered by UniXCoder, showcases commendable speed in token generation, particularly excelling in C++ contexts. Conversely, the study reveals the limitations of simpler models like the RoBERTa-based Code model, which consistently lags behind in performance metrics. Thus, the paper underscores the critical importance of selecting models tailored to specific project requirements, advocating for domain-specific fine-tuning to address the unique intricacies and challenges posed by individual software projects.

Acknowledgements

I would like to like to highlight the support my supervisor, Professor Zheng Wang, has provided throughout the project. Without his support, this project would not be possible.

I would also like to thank my assessor, Dr Rafael Kuffner Dos Anjos, for his valuable feedback and guidance that helped steer this project in the right direction.

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

In the last decade, innovations in Natural Language Processing have paved the way for the development of pre-trained models, such as BERT and RoBERTa. These models, being trained on large corpora of unlabelled text data, allowed research to develop state-of- the-art models for various downstream tasks, including code completion. By default, these pre-trained models showed their capability for code-related tasks, having been trained on data that also included code, but their performance can be significantly enhanced by fine-tuning them to a task. The methods, architectures and additional complexities of pre-trained models like CodeBERT and UniXcoder, promise better performance but pose challenges for developers. This paper aims to provide a detailed guide on how to fine-tune three pre-trained models, RoBERTa, CodeBERT and UniXcoder for code completion. Additionally, it aims to evaluate the performance of the models on a common dataset, CodeSearchNet, and attempts to answer the question of which model is most effective for domain-specific code completion.

1.2 Background Research

The cause behind the creation of pre-trained models as BERT [8] and RoBERTa [22], is the invention of Multi-layer Transformer Architecture [31]. This architecture allowed models to process large amounts of data in parallel and more efficiently than before, and as a result, capture the context and meanings of data beyond the surface-level patterns. It revolutionised the field and is the building block of Large Language Models (LLM). Pre-trained models work by training a neural network on a large corpus of unlabeled text data to learn general patterns and representations of the data. unlabeled data refers to data that lacks explicit annotations or labels indicating the desired output or meaning. BERT was trained on BooksCorpus dataset, which contains over 11,000 books, and the English Wikipedia. RoBERTA was trained on the same data, with the addition of Common Crawl data, which contained, at the time, billions of web pages [7]. As such, both BERT and RoBERTa were trained on unlabelled text data, so they were self-supervised models. Even after the models process all their respective data, they cannot be used for any specific task, as by default all they can do is feature extraction, which is representing the supplied input as a matrix of weights once it passes through the model. As such, they require fine-tuning on task-specific labelled data. This process involves updating the parameters of the pre-trained model using a smaller labelled dataset and, as such, the model is specified to a task. These tasks include for example text classification, named entity recognition, or, in the case of this study, code completion.

1.2.1 Code Completion

This paper focuses on code completion, specifically for Masked Language Modelling (MLM) tasks. Code Completion refers to the process of completing code snippets based on the context of the code, with the context being the code that has been written before the snippet [25] [26]. Tools based on Code Completion enhance the productivity of the developer by reducing the need for manual typing and by providing suggestions based on context, language syntax, and previously written code. Approaches to Code Completion vary depending on how much the model predicts at once. Seq2Seq models predict the entire code snippet at once, whereas MLM models predict one token at a time [30]. This paper focuses on MLM models, as it is much easier to judge whether the model has given the correct answer or not. Pre-trained models can be fine-tuned to perform code completion tasks by focusing on an objective like MLM, where the model is trained to predict a masked token in the code snippet.

1.2.2 RoBERTa for Code Completion

MLM is a generic objective that is not unique to code completion tasks. It is used in many Natural Language Processing tasks that require the prediction of a masked token in a sentence. The Pre-trained RoBERTa was fine-tuned for MLM based on the data they were provided with to answer queries like "The capital of France is [MASK]", where the model would predict "Paris". Since the RoBERTa was trained on also contained some code snippets, it indirectly learned some code representations and as such, was able to perform code completion tasks to some extent. However, the model was not specifically trained on code data, so their performance on code completion tasks was not optimal. It might be argued that because the models were trained on redundant NL data that wasn't useful for code completion tasks and this generality of the model made it difficult to predict code accurately. On the other hand, training on unrelated NL might have allowed it to gain a deeper understanding of NL, which code inheritably contains in the form of comments and documentation.

1.2.3 Fine-tuning for Code Completion

As such, the ideal circumstance would be to use the pre-existing weights of RoBERTa but build upon them by fine-tuning them for code completion. That is exactly what CodeBERT [10] did, by building upon RoBERTa's language understanding capabilities and pre-training the model on code-specific data, in that case, it was CodeSearchNet which is described more in **??**. By pre-training RoBERTa on code data, CodeBERT attempted to make the model code specific, rather than the general text understanding that RoBERTa achieved. This approach captures the benefits of both approaches, the default RoBERTa weights help with general natural language understanding, while the weights changed to incorporate intricacies of programming language syntax and semantics. This allowed the model to consistently outperform RoBERTA on code-related tasks.

1.2.4 Bimodal Models

CodeBERT is a bimodal model, as such it uses pairs of Natural Language (NL) and Programming Language (PL) data to improve the model's performance. This is because code is

often written with comments and docstrings that provide context and explanation for the code. However, there are other representations of input data that could be used to provide better results when applied to general-purpose code completion tasks. General-purpose means that the model can generate code snippets for any programming language, not just the one it was trained on. One such representation is by transforming code snippets into Abstract Syntax Trees (AST), which was utilised by UniXcoder [11]. The reasoning behind this is that ASTs capture the structure and semantics of code more effectively than raw code snippets and as such, UniXcoder outperforms was created as a pilot study of the viability of such an approach. Another representation could be done using Dataflow analysis, which captures the flow of data throughout a program. This was used by GraphCodeBERT [12] as an attempt to improve upon UniXcoder. Dataflow analysis can provide additional context for code snippets, like variable dependencies and data transformations, which are crucial for understanding the behaviour of a program and how data changes through its execution. Additionally, GraphCodeBERT was created as a follow-up to UniXcoder because computing Data Flow is a less computationally expensive way to incorporate the structure of the code into the model. Whereas calculating ASTs particularly expensive in the preprocessing of the dataset. All three models are bimodal as they use pairs of NL and PL but in the more complex models, UniXcoder and GraphCodeBERT, process the PL data further to provide additional context for the model.

1.2.5 Challenges of Existing Models

All of these models outperformed RoBERTa for code-related tasks, but are computationally expensive to train and require more complex data representations. There is also a simpler alternative; fine-tune a RoBERTa model on code data. Regardless of the differences in data representation, all of those models were multi-lingual, which means they received input data in a variety of programming languages, adding complexity to the training process and increasing the amount of time it takes to train a model to high accuracy. For general-purpose code completion tasks, like for the query ‘write bubble sort in C”, such models will be able to provide high-accuracy answers. However, if you are a developer working on a specific large software project, with your task being for example building a new feature or fixing a bug in a codebase with thousands of files, the context surrounding these tasks increases in importance. One example of such a project would be the Linux kernel, of which version 5.15 contains 27 million lines of code [29]. In such scenarios, general code completion might not be sufficient as the task requires a deep understanding of the project’s codebase. Understanding the underlying architecture, design patterns, and interactions between different modules becomes critical. In such cases, the limitations of traditional code completion models become obvious. While models like CodeBERT, UniXcoder, and GraphCodeBERT excel in generating syntactically correct code snippets, they may struggle to provide solutions that align perfectly with a project’s architecture or adhere to specific coding conventions followed by developers. Most importantly, the effectiveness of these models heavily relies on the quality and diversity of the training data. For niche or domain-specific languages, frameworks or tasks, the availability of training data might be limited, leading to suboptimal performance regardless of the model used. To address these challenges, one can fine-tune a model for code completion on the specific codebase of the project, this is called domain-specific fine-tuning [5]. By training the model on

the project's codebase, it becomes aware of the unique patterns, conventions, and intricacies of that software project.

1.2.6 Research Gap

However, if one were to fine-tune such a model, there is a question of which model to use. Although more complex models like UniXcoder and GraphCodeBERT outperform CodeBERT on code completion tasks, that is only true for general-purpose code completion tasks as that is what it was evaluated on. It could be the case that even the simplest models like RoBERTA could outperform the more complex models Also, those models were trained in multiple languages, which might not be necessary for most software projects. As such, you could potentially train the model on less data and in less time, which would save computational resources and time. To achieve the results that for example CodeBERT did, it had to train 16 interconnected NVIDIA Tesla V100 with 32GB memory for 120 hours [10]. This is a significant amount of computational resources and time, which might not be feasible for most developers or companies. As such, there is a need to explore the effectiveness of simpler models for code completion tasks, as it could allow developers to have their own tailored code completion tools that they can realistically train on either their hardware or on cloud hardware that is more affordable than 16 interconnected NVIDIA Tesla V100s.

1.2.7 Research Objective

As such, the goal of this paper is to provide a guide for developers on how to fine-tune a model on a codebase and to evaluate the effectiveness of simpler models for code completion tasks. This will be achieved by fine-tuning all three models on the same codebase (CodeSearchNet) for MLM and evaluating their performance on code completion tasks. Meaning I will be fine-tuning RoBERTA, CodeBERT and UniXcoder while adapting their designs in a way that an arbitrary developer could do it.

1.3 Significance of Research

Code Completion tools have become a common occurrence in most Integrated Development Environments (IDEs) and text editors [33]. At the moment, these tools help developers by speeding up mundane tasks such as typing out long variable names or function calls, completing well known algorithms or providing suggestions for what the developer might want to write next. Their benefit is unquestionable; efficiency, automation of repetitive tasks and reduction in syntax errors that sometimes take developers a long time to find and fix[28]. Tools such as GitHub Copilot [24] have millions of paying users, clearly showing there is a demand for such tools. However, the current tools are not perfect and can sometimes provide incorrect suggestions or not provide any suggestions at all. If the code that the developer is writing is using an obscure library or an API that the tool is not aware of, the tool will not be able to provide any suggestions. As such, one significance of this research is to explore if developers could train their models on their codebase to provide more accurate suggestions for their specific tasks and explore what the process looks like.

The other significance is that there is a push in the research field to find more complex and interesting ways of representing the inputs and modifications to the original multi-layer transformer architecture. This is because it seems to be very hard to train models to be generally good for every common programming language and still be useful for niche uses. However, from available research, there is no push for developers even in large software companies to try to train their models on their codebase. The benefit of such a tool was described above, but it could be that the costs outweigh the benefits. Typically, the training of such models is done by large companies like Microsoft, Google and Facebook, who have the resources to train such models. The specific hardware they use, and the energy costs associated with training such general-purpose models, are not disclosed, but it is likely very expensive. That's why most developers don't do it. This research will explore if the developers don't try to research cutting-edge models, by liming the number of languages it works on, training it for fewer steps, fine-tuning rather than training from scratch and using a redesigned model that will speed up training, it still be useful for them. If it is useful, this could save developers and software companies time and money. If not, this research could show there such an approach is lacking and hopefully future research will explore this further and solve the potential problems that this paper might find.

1.4 Analisys of existing fine-tuned models on domain codebases

The area of personalised code completion tools is a relatively new topic of research, with There exist companies that have proprietary tools which attempt to create such tools for enterprises, like Codeium [18] and TabNine [9]. There are guides available which mention how to personalise such models for domain-specific tasks [23]. Therefore, there are commerce applications of such tools, but there is no research that explores which models are best for such tasks if one is to train it on their codebase. General-purpose tools like GitHub Copilot improved over time by scanning the whole file that you are writing to and any previous files it has already suggested for you in that repository. This is a form of personalisation, but it is not the same as training a model on your codebase, as the performance of such models goes down there more files it has to scan [17]. This is because of the limited context size of the model, which is the number of tokens it can remember from the previous code it has seen. From the research GitHub Copilot provides, the model will take into account the whole file you are currently working on and the neighbouring tabs that are currently open in the IDE. As such, it is unrealistic to expect the programmer to have all the files open in the IDE that they are working on and also, for such models that utilise transformer architecture, there is a limited context size of the model. Although the number for the GitHub copilot isn't known, it is not infinite and in the case of a large codebase, it would need an exceptionally large context size to be able to provide accurate suggestions. The current LLM that has the largest context size is Google Gemini Ultra, which has a context size of 1 Million tokens [6]. Although this is significant, it is still not enough to scan the whole codebase of a large software project which has millions of lines of code, not tokens.
As figure **??** demonstrates, the number of tokens in a single codebase can be quite significant. That is why the objective of this paper is to look into alternatives.

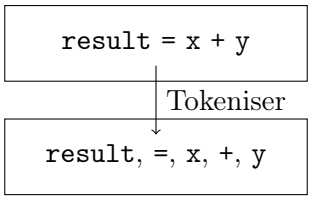


Figure 1.1: Example of even a single line of code can produce many tokens.

1.5 Motivation behind this

The approach described was inspired by a Google DeepMind[21], where they concluded that large datasets require large and complex models. However, smaller datasets require smaller and less complex models instead to achieve optimal performance. This means that if one wants to train a general-purpose model, one would need a complex model, much like CodeBERT did. However, if the mode is designed for optimal performance on a singular codebase, a less complex model would be ideal. Although this was applied in the context of image generation, the same concept could be applied to code completion tasks.

1.6 Masked Language Modelling

This study will focus on Masked Language Modelling (MLM) code generation methods as they are commonly used by popular tools like GitHub Copilot [1] and because of the ease to judge whether the model has given the correct prediction or not. The approach that will paper will

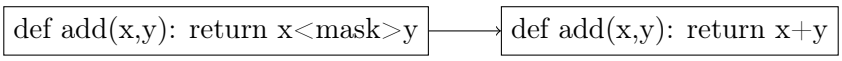


Figure 1.2: What should happen if a simple Python code gets passed to a code completion model trained for MLM

be using was introduced by the Google AI language team as a follow-up to BERT, trying to find the optimal way to fine-tune BERT for a specific task, in that case, Question answer pairs. Their approach was improving BERT by leveraging bidirectional pre-training, which randomly masks tokens in the input text and tasks the model with predicting the original vocabulary ID based on context. Unlike previous unidirectional models, BERT's MLM objective enables the fusion of both left and right contexts, allowing for deep bidirectional representations. When applied to figure 1.2, the model improves more if it's able to see both the right and left context of the masked token. CodeBERT recommends using this approach when detailing how to fine-tune it for various tasks [10]. Of course, when applied to code completion tasks, there needs to be a multitude of ways of evaluating it, not just testing it on predicting the replaced masked token in the middle of a code snippet, but also when applied to predicting the next token, which will be discussed in the Results section.

1.7 Review of models

This paper will focus on fine-tuning these three base models as they provide a good representation of the shift of complexity and performance of the models. The base models of

Roberta, CodeBERT and UniXcoder are available on HuggingFace. All 3 of the models are based on RoBERTa which was based on BERT, which was based on the multi-layer transformer. As such, their layers and sizes are all the same, yet their input data representation and objectives are much different. As such, they share the same architecture from Table 1.1. When fine-tuning such models on specific objectives, the requirements of the model have to be

Table 1.1: RobertaModel Parameters and Sizes

Component	**Size/Description**
Word Embeddings	50265 vocabulary size, 768 embedding size
Position Embeddings	514 positions, 768 embedding size
Token Type Embeddings	1 token type, 768 embedding size
Layer Normalization	768-dimensional vectors
Dropout	10% dropout rate
Encoder Layers	12 layers
Attention Mechanism	768-dimensional queries, keys, and values
Intermediate Dense Layer	3072 output features
Output Dense Layer	768 output features
Pooler	768 input and output features
Activation Function	Tanh activation
Number of Parameters	124,645,632

satisfied. Despite all three models sharing the same architecture, RoBERTa is called a simpler model, as all RoBERTa needs is a dataset of text passing it to the model, and masking it beforehand. CodeBERT needs a dataset of NL and PL pairs of function code, and UniXcoder needs a dataset of AST and comment pairs. For CodeBERT, if the chosen dataset doesn't contain a matching NP for PL, it requires that NP be filled in for all function code that one wishes to supply to the model, which takes time. Alternatively, excluding such code is a possibility, but it leaves less code to train on. This is an especially important point if your codebase is not large. For UniXcoder, computing the AST representations of data is a computationally expensive task. Obtaining these inputs requires a lot of data manipulation and preprocessing, which is not required for RoBERTA. Developers could go through the process of calculating the required data and scraping their codebase, but there are questions as to the usefulness of such models. Especially since developers can optimise the models for fine-tuning on a specific codebase without having to go through these steps if one picks the simpler approach of fine-tuning RoBERTa.

1.7.1 RoBERTa

BERT (and by extension, Roberta) operates through a two-step framework: pre-training and fine-tuning. During pre-training, the model learns from unlabeled data via tasks like Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). The input representation is designed to handle various downstream tasks, accommodating both single sentences and sentence pairs in a unified token sequence. This representation incorporates WordPiece embeddings with a 30,000-token vocabulary, where the first token is always a special classification token ([CLS]). Sentence pairs are packed together with a special separator token ([SEP]), and embeddings indicating sentence A or B membership are added to each token.

Through dynamic masking during pre-training and a self-attention mechanism during fine-tuning, BERT (and Roberta) effectively handles diverse natural language tasks with a unified architecture, achieving state-of-the-art results.

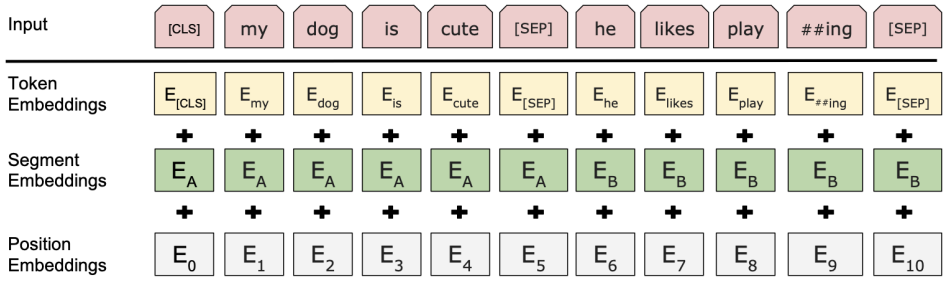


Figure 1.3: The input representation for RoBERTa consists of the combined token, segmentation, and position embeddings [8].

1.7.2 CodeBERT

During pre-training, it concatenates two segments with a special separator token: one segment for natural language text and the other for code. This concatenated input includes a special token ([CLS]) at the beginning and an end-of-sequence token ([EOS]) at the end. Input representations are processed using WordPiece for natural language and tokenization for code. The output includes contextual vector representations for each token and an aggregated sequence representation from the [CLS] token. In contrast to BERT's input representation handling single sentences or sentence pairs, CodeBERT deals with bimodal data pairs consisting of natural language-code pairs or unimodal data like code without paired natural language text. Pre-training involves objectives like Masked Language Modeling (MLM) and Replaced Token Detection (RTD), leveraging both bimodal and unimodal data. In fine-tuning, CodeBERT is applied to various downstream tasks such as natural language code search and code-to-text generation, adapting the model's representations accordingly. Compared to BERT's single or paired sentence input representation, CodeBERT handles the integration of natural language and code, reflecting the unique nature of the tasks it addresses.

1.7.3 UniXcoder

UniXcoder integrates code comments and Abstract Syntax Trees (AST) to enhance code representation in pre-trained models. It employs a Transformer architecture with mask attention matrices and prefix adapters to control model behaviour. The input representation involves transforming ASTs into token sequences while retaining structural information, achieved through a one-to-one mapping function. For each code snippet, the comment and flattened AST token sequence are concatenated with a prefix indicating the model's mode (e.g., encoder, decoder). During pre-training, UniXcoder applies N transformer layers over the input, producing hidden states for downstream tasks. Compared to CodeBERT and BERT/Roberta, UniXcoder uniquely incorporates both code comments and ASTs, offering richer semantic and syntactic information for code representation. Additionally, its input representation differs in that it directly integrates ASTs alongside code comments, providing comprehensive knowledge for code understanding.

Chapter 2

Methods

2.1 Version control and project management

2.1.1 Development Tools

The project utilised GitHub for version control, enabling collaborative development and tracking of code changes. Python served as the primary programming language due to its seamless integration with the Transformer library from PyTorch, essential for model implementation. Jupyter Notebook facilitated data comparisons and visualization through graphs, enhancing interpretability and ease of experimentation. Weight and Biases were employed for monitoring and tracking the training progress of the models, ensuring efficient optimization.
PyTorch was selected for model implementation and training, aligning with the methodology of the original authors as documented in their publication [11]. Hugging Face was utilized for model architecture and tokenization, following the approach established by the original authors.
Visual Studio Code (VS Code) was chosen as the Integrated Development Environment (IDE) for its lightweight nature and compatibility with running Jupyter Notebooks, streamlining code development. Leveraging the notebook environment addressed the challenge of dataset loading time, which amounted to approximately two minutes per iteration due to its considerable size. By structuring the code into executable cells, Jupyter Notebook allowed for selective execution, reducing the necessity to reload the dataset frequently.
It also uses Python 3.12 and heavily utilises the Transformer library, version 4.39.
Given the substantial size of the dataset (2GB) and individual model files (approximately 1GB each), Google Drive served as the storage solution due to its ample storage capacity, surpassing the limitations of GitHub. The GitHub repository facilitated version tracking and collaboration, while Google Drive accommodated the storage demands of large files, preventing upload constraints posed by GitHub's 100MB file size limit.

2.1.2 Project Management Methodology

The project was organised into three distinct sprints, each focusing on specific tasks to ensure a systematic and efficient development process: The division of the project into sprints was driven by the need for effective project management and timely completion. This approach ensured that the project remained manageable, allowing for focused attention on each phase of development. Given the project's inherently modular nature, the subdivision into sprints provided a logical framework for progress tracking and milestone achievement.
The intervals between sprints served multiple purposes, including testing the implementation of preceding sprints to validate functionality (writing test scripts to test the correctness of preprocessing for example), refining code as necessary, and allocating time for report writing and research in preparation for subsequent sprints. This iterative process facilitated continuous

improvement and refinement of the project's components, ultimately contributing to its overall success.
I aimed for 1-month sprints to give the project the time to fix any unexpected issues that might arise.

2.1.3 Sprint 1: Data Collection and Preprocessing

During this initial sprint, the primary focus was on acquiring the CodeSearchNet dataset, conducting necessary preprocessing steps, and organizing the data to facilitate subsequent model training tasks.

2.1.4 Sprint 2: Model Implementation and Training

The second sprint concentrated on the implementation of various models, their training utilizing the preprocessed CodeSearchNet dataset, and conducting preliminary evaluations to ensure the functionality and efficacy of the models.

2.1.5 Sprint 3: Model Evaluation and Analysis

The final sprint involved a comprehensive evaluation of the trained models on the validation set, accompanied by in-depth analysis and comparisons.

Figure 2.1: Gantt Diagram for the project, showcasing the time taken for each project phase.

2.2 Data collection and preparation

CodeSearchNet [14] corpus is a dataset of over 2 million functions from open-source code spanning six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). The dataset includes function-level documentation, code, and metadata, making it a valuable resource for training and evaluating code completion models. The data was scrapped from GitHub back in 2018. The dataset is unique because it provided NL and PL pairs, allowing for the creation of bimodal models, like CodeBERT. Each entry in the dataset is represented by a JSONL object, which is a format for storing structured data where each line of a text file represents a separate JSON object. There are numerous datasets available, including those based on single codebases, but using CodeSearchNet seems ideal as it allows for easy comparisons between CodeBERT and UniXcoder as they also used this dataset. Additionally,

this project introduced many improvements to the original models, so it is important to the same data to allow for those improvements could be fairly evaluated.

```
{
    "repo": "kgori/treeCl",
    "path": "treeCl/bootstrap.py",
    "func_name": "hessian",
    "original_string": "def hessian(x, a):\n    \"\"\" J'.J \"\"\"\n    j =
        jac(x, a)\n    return j.T.dot(j)",
    "language": "python",
    "code": "def hessian(x, a):\n    \"\"\" J'.J \"\"\"\n    j = jac(x, a)\n
            return j.T.dot(j)",
    "code_tokens": ["def", "hessian", "(", "x", ",", "a", ")", ":", "j",
        "=", "jac", "(", "x", ",", "a", ")", "return", "j", ".", "T", ".", "
        dot", "(", "j", ")"],
    "docstring": "J'.J",
    "docstring_tokens": ["J", ".", "J"],
    "sha": "fed624b3db1c19cc07175ca04e3eda6905a8d305",
    "url": "https://github.com/kgori/treeCl/blob/
        fed624b3db1c19cc07175ca04e3eda6905a8d305/treeCl/bootstrap.py#L78-L81
        ",
    "partition": "train"
}
```

Listing 2.1: The smallest object in the train.jsonl of the Python CodeSearchNet with the size of the original string being 75 length.

Table 2.1: The data fields in the CodeSearchNet dataset. Each JSONL object contains the following fields.

Field	**Description**
ID	Arbitrary number
Repository Name	Name of the GitHub repository
Function Path in Repository	Path to the file which holds the function in the repository
Function Name	Name of the function in the file
Whole Function String	Code + documentation of the function
Language	Programming language in which the function is written
Function Code String	Function code
Function Code Tokens	Tokens yielded by Treesitter
Function Documentation String	Function documentation
Function Documentation String Tokens	Tokens yielded by Treesitter
Split Name	Name of the split to which the example belongs (one of train, test or valid)
Function Code URL	URL to the function code on GitHub

To make the data fit the models, it will require severe modifications. The most important data are the code tokens and documentation string tokens, as these are required to be passed to the models. This project solely focuses on Python and JavaScript (JS) data. The reasoning behind

this stems from the approach that less but high-quality data could outperform the high data approach[21]. There are twice as many JSONL objects for Python than JS in CodeSearchNet. However, the JS code mostly focuses on web development tasks, whereas the Python code in the dataset focuses on a wide range of applications. This could be an interesting comparison to check if the approach from Scaling (Down) CLIP could also be applied for code completion for the RoBERTa transformer architecture. Additionally, the number of languages in a codebase varies dramatically, so it's difficult to predict how many languages should this project focus on. As such, the project assumes that at minimum a large codebase requires one frontend and one backend programming language, and JS-Python combinations seem to be very common [16].

Dataset	Train	Test	Valid	Overall
Python	251820	14918	13914	280652
JavaScript	93889	6483	8253	108625

Table 2.2: Number of JSONL objects in each dataset split for Python and JavaScript.

Before starting with preprocessing, it is important to understand the data the models will be working with and since code tokens are a crucial part of it, visualisation of them hints at what the models will be trained to predict.

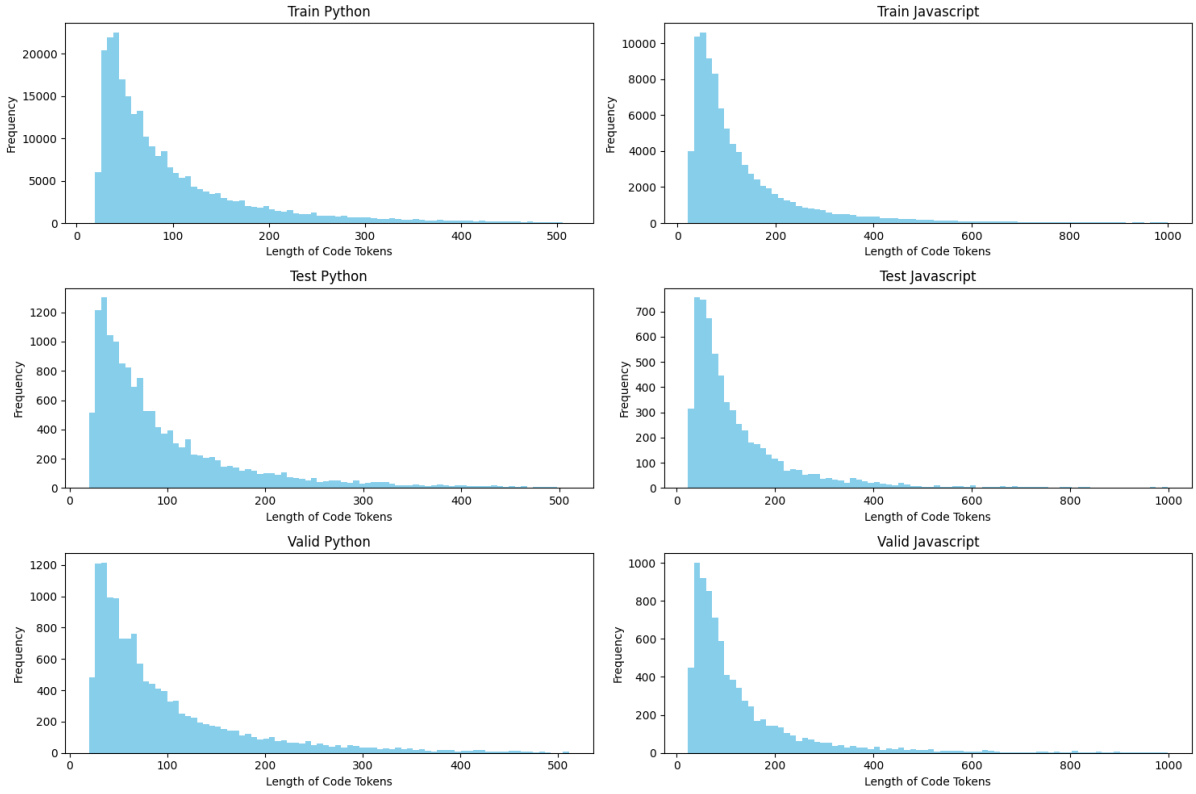


Figure 2.2: Histogram of code token lengths for Python and JavaScript data from CodeSearchNet.

The histogram shows that the majority of the tokens for Python are concentrated in the 50-token range, whereas for JS, it sits in the 100 range. This is important as it could explain if there are ranges where each model performs better than. Also, there are outliers where the data exceeds 1000 tokens, but only for JS code. The largest single code snippet in the dataset is over 10,000 tokens long. However, any code snippets that exceed 512 tokens will have to be

```
Number of outliers in python/train.jsonl: 0
These are where the code_tokens are too large (>1000).
[]

Number of outliers in javascript/train.jsonl: 1099
These are where the code_tokens are too large (>1000).
[1010, 1140, 1370, 1672, 1036, ...]
```

Figure 2.3: Outliers calculated by CodeSearchNet/tables.ipynb. The numbers in the [] represent the length of the code tokens that exceed 1000

discarded from the dataset as the RoBERTa architecture has a maximum token limit of 512.

2.2.1 Preprocessing

Given that the dataset's raw format doesn't align with the architecture's input requirements, preprocessing is necessary to transform the data into a suitable format before feeding it into the model which in this case will be tokenisation, masking, padding and features distinct for each model.
As seen in table 2.1, the code tokens are already tokenised. However, each of the models has its distinct tokeniser, so this step will have to be repeated. The next necessary step is to remove redundant fields, such as Repository Name, Function Path and others, until only the code tokens and docstring tokens remain.

2.2.2 Distinct datasets

It is required to create distinct datasets for each model. This is because the models have different requirements for the input data, so the data has to be processed differently and stored for training, validation and testing purposes. For RoBERTA, all that will be provided to it will be the code tokens with all the comments removed. As RoBERTa accepts just text for fine-tuning, this tests the hypothesis of whether code tokens as input representation are enough. For CodeBERT, code tokens and docstring tokens are required as the model is bimodal. For UniXcoder, the computation of ASTs for each code snippet is required and stored alongside the relevant docstrings.
To remove comments in CodeSearchNet, look for a token in `code_tokens` that starts with \" and ends with \", and remove every instance of them from the object.

2.2.3 Masking

The masking design implemented in the project involves randomly selecting a token from the code tokens within each JSONL object and replacing it with a special token <mask>, while preserving the original token as the desired output. To ensure the integrity of the code structure, tokens identified as comments are excluded from the masking process.
Despite only masking one token in a sequence, this approach is sufficient for code generation tasks as the model will still be able to predict the next token multiple times. The reasoning behind masking just one token lies in the model's inherent ability to capture contextual

information from the surrounding tokens in the sequence. By masking only one token at a time, the model is forced to rely on the contextual cues provided by the unmasked tokens to predict the missing token. This encourages the model to learn rich representations of the code syntax and semantics, enabling it to generate accurate and coherent code sequences. Furthermore, limiting the number of masked tokens reduces the computational complexity of the training process while still allowing the model to effectively learn and generalise from the data [27].

2.3 Software Architecture

2.3.1 Supervised Learning and Importance of it

For CodeBERT, even when the authors attempted to fine-tune the model for any code-specific task, have continued with their approach of utilising self-supervised learning. Meaning, for both pre-training and fine-tuning, they have used unlabelled data. This approach works, however requires a large amount of data and extra computing, meaning the model has to train for longer. However, since a dedicated codebase won't have as much data as CodeBERT used, using self-supervised learning might not yield good results. As such, this project utilises supervised learning as an approach to test if training the models on fewer data, but higher quality data allows the models to achieve similar accuracy as detailed in [?]. This change of methodology requires the input data to also contain an output field, which indicates that the model should predict 2.3.

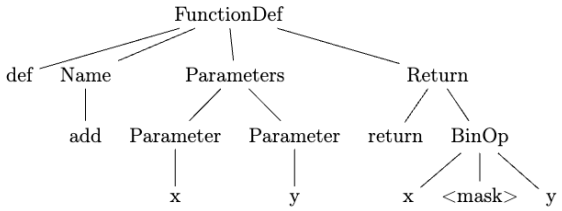
Models	First Input	Second Input
RoBERTa	`def add(x, y): return x <mask> y`	n/a
CodeBERT	`def add(x, y): return x <mask> y`	"Add x and Y and return the result"
UniXcoder		"Add x and Y and return the result"

Table 2.3: Where the desired output is always `def add(x, y): return x + y`

2.3.2 Tokenisation

The tokenisation process involves converting raw input data, comprising code snippets and associated annotations, into structured sequences of token IDs, which are then fed into the models for training.

For code snippets, subword tokenization effectively handles out-of-vocabulary words and captures fine-grained syntactic and semantic information present in the code. This approach not only enhances the model's ability to understand complex code structures but also ensures robustness to variations in coding conventions and style. The reasoning behind using this technique is the commonality for Neural Machine Translation and other ML tasks [19].

Similarly, for abstract syntax trees (ASTs), we employ a recursive traversal method to map each node in the AST to its corresponding token IDs. This technique, demonstrated in the

`python_ast_to_ids` and `javascript_ast_to_ids` functions, encapsulates the hierarchical structure of the code, preserving both syntactic and semantic relationships between different components of the code [32].

2.3.3 Padding

In the design of our system, we employ a customized collate function, denoted as `collate_fn`, to handle the variability in token lengths across input sequences during batch processing. This function is integral to the training pipeline as it facilitates the efficient handling of variable-length inputs, ensuring compatibility with neural network architectures that require fixed-length input tensors.
The `collate_fn` function operates on a batch of input samples, where each sample consists of input IDs, attention masks, and corresponding labels. The primary objective of the function is to pad the input sequences to a uniform length, thereby enabling batch processing without the need for dynamic computation graphs.
To achieve this, the function dynamically determines the maximum sequence length within the batch and pads each input sequence accordingly. Specifically, it pads the input IDs and attention masks using PyTorch's `pad_sequence` function, ensuring that all sequences within the batch are of uniform length. Additionally, it stacks the labels into a single tensor, ready for use in the training process.
By incorporating this custom collate function into our training pipeline, we effectively mitigate the challenges posed by variable-length inputs, thus optimizing the efficiency and scalability of our multi-task learning system. This ensures consistent and reliable model performance across diverse input data, ultimately enhancing the system's ability to generalize and produce accurate predictions.

2.3.4 Hyperparameters and Optimisation

Tested on a subset of data with varying learning rates, epochs, and batch sizes, identifying the optimal parameters; employed early stopping to prevent overfitting and a learning rate scheduler for dynamic adjustments during training. Specifically, experiments conducted on 10% showed the following parameters yielded the best results. A learning rate of 2e-5, 4 epochs, batch size of 26, epsilon of 1e-8 for the AdamW optimiser, and a gamma value of 0.1.

2.3.5 Evaluation Metrics

The reasoning behind the evaluation metrics chosen in the Results Chapter emphasises the significance of time and accuracy in Code Completion, which this project assumes would be more important for developers than the standard benchmarks used in NLP tasks.
The Algorithm 1 provides an overview of the fine-tuning process for the models. While slight variations exist among models due to differences in inputs, such as varying data loaders and tokenisers, the training loop remains consistent.
Early stopping is implemented in the fine-tuning algorithm to prevent overfitting and optimizing model performance. By monitoring the average loss during training and comparing it to the best loss seen so far, the the algorithm determines whether the model's performance is

Algorithm 1 Fine-tuning Algorithm for CodeBERT, RoBERTa and UniXcoder

```
Load the pre-trained model
Initialize optimizer, learning rate scheduler and define the loss function
for each epoch in the range of total epochs do
    for each batch in the training dataloader do
        Retrieve input_ids, attention_mask, and labels from the batch
        Reshape the labels to match the shape of the inputs
        Zero the gradients of the optimizer
        Forward pass: Compute model outputs (including loss)
        Calculate loss and accumulate it
        Backward pass: Compute gradients and update model parameters
        Apply gradient clipping if necessary
        Update optimizer and learning rate scheduler
    end for
    Calculate the average loss for the epoch
    Check for early stopping:
    if the current average loss is better than the best loss seen so far then
        Update the best loss and Reset patience counter to 0
    else
        Increment patience counter
    end if
    if the patience counter exceeds the specified patience then
        Break out of the training loop and Save Model
    end if
end for
Save the trained model and tokenizer
```

improving or deteriorating. If the current average loss improves upon the best loss observed previously, indicating better generalization, the best loss is updated, and the patience counter, which tracks the number of epochs without improvement is reset. However, if the model's performance fails to improve for a specified number of epochs (patience), the algorithm terminates training early to prevent further overfitting and save the model with the best performance. This adaptive stopping criterion ensures that the model is trained for an the optimal number of epochs, striking a balance between maximizing performance and avoiding overfitting, meaning it saves time for developers as no points training it for longer than necessary.

2.3.6 Lora

LoRA (Low-Rank Adaptation) presents a novel technique for accelerating the fine-tuning of large pre-trained language models while minimizing memory consumption [3]. By decomposing weight updates into two smaller matrices A and B through low-rank decomposition, LoRA efficiently represents the adaptation process mathematically as $\Delta W = A \cdot B^T$. This approach drastically reduces the number of trainable parameters while enabling efficient adaptation to new data. LoRA's advantages lie in its parameter efficiency, allowing for the creation of lightweight and portable models with frozen pre-trained weights. Moreover, its compatibility with other parameter-efficient methods and seamless integration with Transformer models make it a versatile tool for various downstream tasks. The resulting LoRA models exhibit

comparable performance to fully fine-tuned models without introducing any inference latency, as adapter weights can be merged with the base model. Overall, LoRA stands as a promising approach to optimize fine-tuning processes, offering efficient adaptation with minimal memory overhead and mathematical rigor [13].

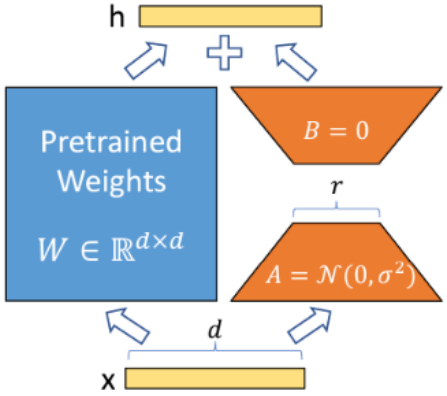


Figure 2.4: LoRA's low-rank adaptation process, illustrating the decomposition of weight updates into smaller matrices A and B.

2.3.7 Pipeline

The aim is to offer guidance on replicating this process for use with any model compatible with the input representation if needed. As depicted in the pipeline diagram in 2.5, CodeSearchNet can be substituted with alternative datasets, and the model being trained can be replaced accordingly. However, the core structure remains consistent, with additional preprocessing steps required for different input representations. This standardized pipeline was employed across the project to ensure uniform training procedures, enabling reliable comparisons and evaluations.

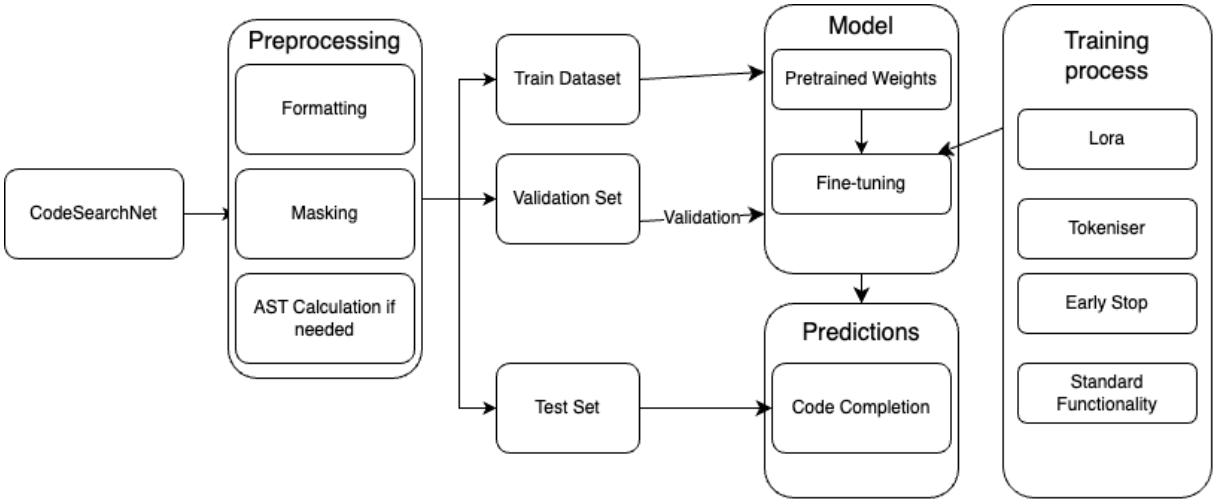


Figure 2.5: The pipeline for training any model.

Chapter 3

Results

This chapter focuses on the results of the model trained model and very detailed analysis to try to identify how the models behave in different situations and what their performance depends on, so that a potential user can make an informed decision about which model is best for their use case.
The models were trained on ARC3 [2], utilising batched jobs on six nodes equipped with 24 cores, 256GB of memory, and 4 x NVIDIA P100 GPUs each, alongside a local storage capacity of 800GB per node.

3.1 Testing of Models

Model	Language	Correct Predictions	Total Predictions	Accuracy
Code	JavaScript	4895	6843	71.54%
CodeComments	JavaScript	5566	6843	81.34%
ASTComments	JavaScript	4963	6843	72.54%
Code	Python	9298	14918	62.33%
CodeComments	Python	11381	14918	76.29%
ASTComments	Python	9179	14918	61.53%

Table 3.1: Combined Results for Models across Python and JavaScript. All the models were tested on their respective test sets for Python and JavaScript.

Although the performance of UniXcode (ASTComments Model) in the context of MLM remains unknown, insights from CodeBERT and Roberta, as reported by the AI team at Microsoft, indicate promising benchmarks: 80% and 60% accuracy, respectively, following fine-tuning for code completion tasks across six languages in CodeSearchNet. In this study, the **Code** model, leveraging Roberta-base, achieved an overall test set accuracy of 66.8%, while **CodeComments**, based on codebert-base, attained 78.5%. These results, obtained with significantly fewer data and computational resources, closely approach or surpass the reported baselines. Notably, these findings suggest that extensive fine-tuning on large datasets may not be requisite for achieving comparable performance levels, underscoring the validity of the concept.

The performance analysis reveals that the **ASTComments** model did not surpass **CodeComments** in either language and demonstrated comparable results to RoBERTa, which was unexpected. Disparities in AST representations across languages may contribute to this discrepancy, necessitating further exploration with additional data and language variations. The decision to train UniXcoder on all languages in **CodeSearchNet** suggests the potential for improved performance in such comprehensive settings. Additionally, the utilization of ASTs as input, rather than code tokens, may impact the model's ability to predict code-level tokens

effectively. It remains possible that UniXcoder developers have implemented strategies to address these differences, potentially leading to improved performance in certain scenarios, although these intricacies remain undisclosed.

The comparison between JavaScript (JS) and Python reveals notable performance differentials despite disparities in dataset sizes. Notably, JS outperforms Python across various metrics, even though the Python training dataset contains 2.68 times more data than its JS counterpart, as seen in Table 2.2. This discrepancy suggests that factors beyond mere dataset size influence model performance. One potential explanation for JS's superior performance could lie in its inherent language characteristics, such as syntactic nuances and program structure, which may render it more conducive to effective model training and prediction. Additionally, differences in the distribution and complexity of tasks within each language's dataset may also contribute to JS's comparative advantage. As detailed in Section 2.2, JS is typically focused on specific tasks, such as web development, leading to a narrower range of expressions and patterns in the data compared to Python, which is employed across a broader spectrum of applications. This narrower focus may simplify the learning task for JS models, enabling them to achieve higher accuracy with less data. However, for Python models to achieve similar performance levels, it may be necessary to either increase the volume of training data or extend the duration of training to capture the diverse range of language usage scenarios.

Even though JS had less data, it had longer code tokens as seen in Table 2.2, which might be another reason why it performed better than Python for all models. This seems to indicate that the quality of the data is more important than the quantity of the data, suggesting that the approach of [21] detailed in **??**sec:motivation can be indeed applied to code completion tasks in Roberta-based models.

3.2 Simple Example

To illustrate the input needed for using the model, it can be observed that the docstring is not needed in the input for CodeComments to give a prediction. For fine-tuning, it is needed, but not a requirement for testing and this principle was used throughout this chapter to assess the quality of the models.

```
Code: ['def', 'func', '(', 'self', ')', ':', 'if', 'self', '.', '_func',
    'is', 'None', ':', 'self', '.', '_func', '=', 'NNTreeNodeFunc', '(',
    'self', ')', '<mask>', 'self', '.', '_func']
Predicted: return
Expected: return
```

Figure 3.1: Example of the output when curring CodeComments on this code. It gives the correct output as the predicted value matches the expected.

3.2.1 Time generation per token

The motivation for investigating time generation per token stems from the recognition that while some models may exhibit superior performance metrics, the efficiency of their predictions is equally crucial. If a model requires significant time to generate individual tokens, its utility for programmers diminishes, as manual coding might be more expedient.

Model	Language	Total Models	Total Instances	Average Time for code completion
Code	JavaScript	4895	6843	0.1329s
CodeComments	JavaScript	5566	6843	0.1299s
ASTComments	JavaScript	4963	6843	0.0896s
Code	Python	9298	14918	0.1403s
CodeComments	Python	11381	14918	0.1408s
ASTComments	Python	9179	14918	0.0901s

Table 3.2: Average Time per Model

As shown in Table 3.2, the averages for each model denote relatively minimal durations, suggesting comparable performance across all models concerning code generation time. However, it seems that the ASTComments model performs an average of 36% faster for Python and 38% for Javascript. This enhanced performance could potentially be attributed to the implementation strategy of UniXcoder, particularly the integration of pooling layers over hidden states. Such architectural choices may facilitate more effective encoding of ASTs into the model, increasing the speed of predictions.

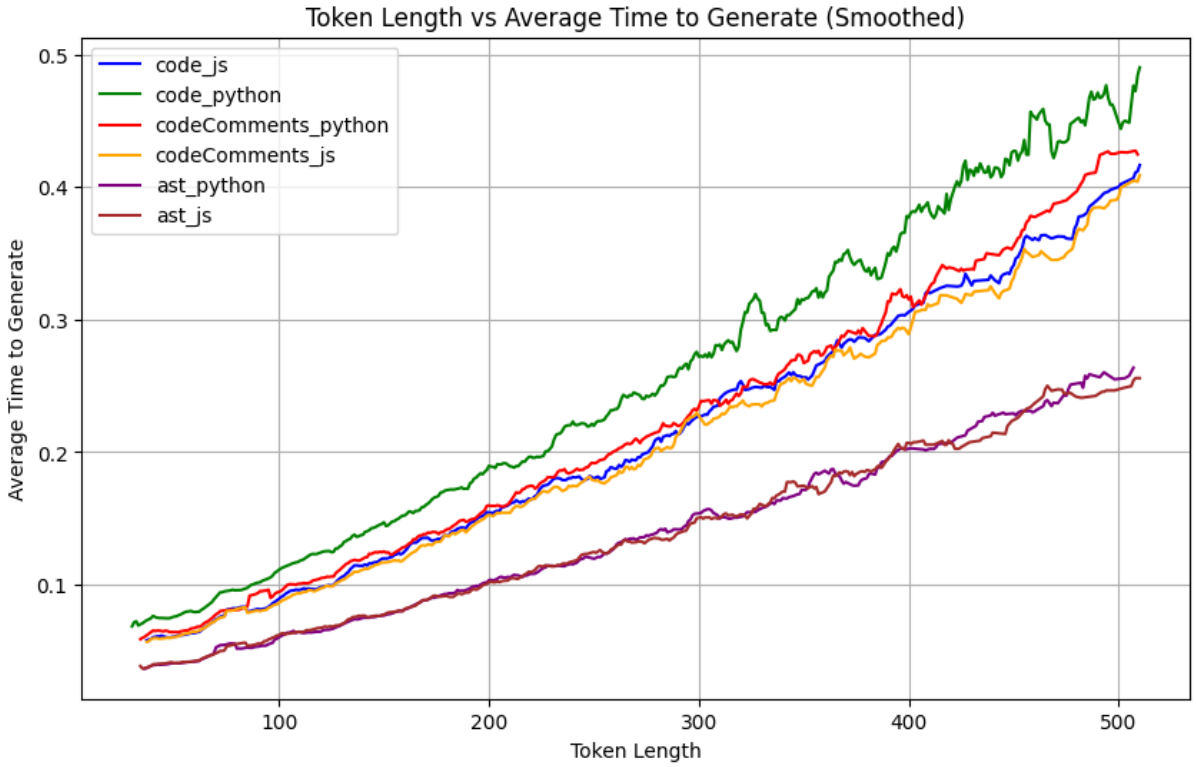


Figure 3.2: Time taken to generate code for each model on the test set.

The analysis reveals a discernible correlation between code generation time and token length,

as illustrated in Figure 3.2. The duration increases proportionally with token length, a phenomenon anticipated given the inherently greater computational demands of lengthier code excerpts. Notably, each model exhibits a distinct correlation, as evidenced by the gradients computed for *code_js* (0.0007407), *code_python* (0.0008817), *codeComments_python* (0.0007601), *codeComments_js* (0.0007178), *ast_python* (0.0004803), and *ast_js* (0.0004683). Significantly lower gradients observed for AST models suggest heightened efficiency, particularly when handling longer code segments at a time. Consequently, for programmers worried about scalability as they have code tokens that are long in their codebase, the AST model emerges as a promising choice. Although the correlation isn't that much different between the models, it might not be a deciding factor for the model choice.

3.2.2 Other languages testing

The reasoning behind this test is to indicate what happens to such models when a third language enters the field that the model has not been trained on. In this case, checking the performance of the models on C++ code snippets from IBM's CodeNN dataset [15]. Developers might be aware that in the future, another language could be introduced to the codebase so choosing a model that best would deal with this might be the ideal option for them.

Model	Correct Predictions	Total
Code	377	672
CodeComments	411	672
ASTComments	469	672

Table 3.3: Prediction Results of all models on C++ code snippets.

As depicted in Table 3.3, the models demonstrate a discernible decrease in efficacy when confronted with C++ code snippets, with the ASTComments model achieved the highest accuracy rate of 69.7%. While this accuracy appears promising, it is imperative to acknowledge the contextual factors surrounding the evaluation. The CodeNN dataset subset utilized for testing comprises notably shorter and simpler code snippets compared to the comprehensive CodeSearchNet dataset. Moreover, the absence of intricate constructs such as object code or complex functions within the CodeNN subset underscores the relative simplicity of the data. Despite these simplifications, the attained performance fell short of anticipated levels. This shortfall underscores the inherent difficulty in extrapolating model performance to new languages. Should the developer seek to integrate additional languages into their codebase, retraining would be imperative. However, without a substantial corpus of legacy code in the new language, retraining poses a formidable challenge, potentially delaying the adoption of the new language until a sufficient volume of language-specific data is amassed. Notably, the superior performance of the AST model is attributed to its enhanced ability to grasp the underlying structural intricacies of the ode, a capability bolstered by its inherent understanding of code syntax. This advantage conferred by the AST property underscores its utility in navigating language-specific nuances and subsequently enhancing model performance.

Listing 3.1: Example where all code models failed. This is the prediction of the CodeComments model.

```
for ( <mask> i = 1 ; i < 10 ; i++ )
    for ( uint8_t j = 1 ; j < 10 ; j++ )
         printf ( "%ux%u=%u\n" , i , j , i * j );
return 0;
Predicted: int
Expected: uint8_t
```

As illustrated by the example showcased in Listing 3.1, the predictive capabilities of the models were challenged when determining the appropriate data type for the variable *i*. This challenge primarily stems from a lack of familiarity with certain data types, such as uint8_t, which are not prevalent in either Python or JS training datasets due to their absence from the vocabulary of these languages. This underscores the inherent limitations of employing models trained on specific languages for code completion tasks in diverse language environments. Potentially, if you trained these models in more languages like CodeBERT did, you could achieve good results in different languages too. However, this was not explored by the original authors.

3.2.3 Mask size testing

The rationale behind conducting this test lies in simulating most code completion tasks, where the model must anticipate the next token based on preceding tokens without knowledge of what lies ahead. This prediction process typically employs repetitive MLM, which predicts one token at a time. The objective of this test is to evaluate the suitability of the models for code completion by assessing their ability to accurately predict tokens, with the challenge increasing as the size of the masked context grows.

The test operates by progressively masking the last x tokens, where x ranges from 2 to 10, for each token to be predicted. Multiple predictions are then made sequentially until the final token is determined. If an incorrect prediction occurs at any point, the prediction process is halted, and the incorrect prediction is tallied. Subsequent predictions with a higher masking level for the same input are automatically considered incorrect since they would yield the same result. The correctness ratio is defined as the proportion of correct predictions made by a model out of the total number of predictions it made. Mathematically, it is calculated as:

$$\text{Correctness Ratio} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{3.1}$$

In this formula, the numerator represents the number of predictions that were correct, while the denominator represents the total number of predictions made by the model.

As depicted in Figure 3.3, the models behave wildly differently. The Code model, based on RoBERTa, exhibits a consistent decline in performance it was only able to predict 12.5% of the tokens correctly when the last 10 tokens were masked. the next token given it predicted the first token correctly. Although it improves a bit for predicting the 3rd token (17.5% accuracy), it remains significantly lower than the other models and fails from 4 tokens onwards. This decline in performance is attributed to the model's inability to grasp bimodal data that is often

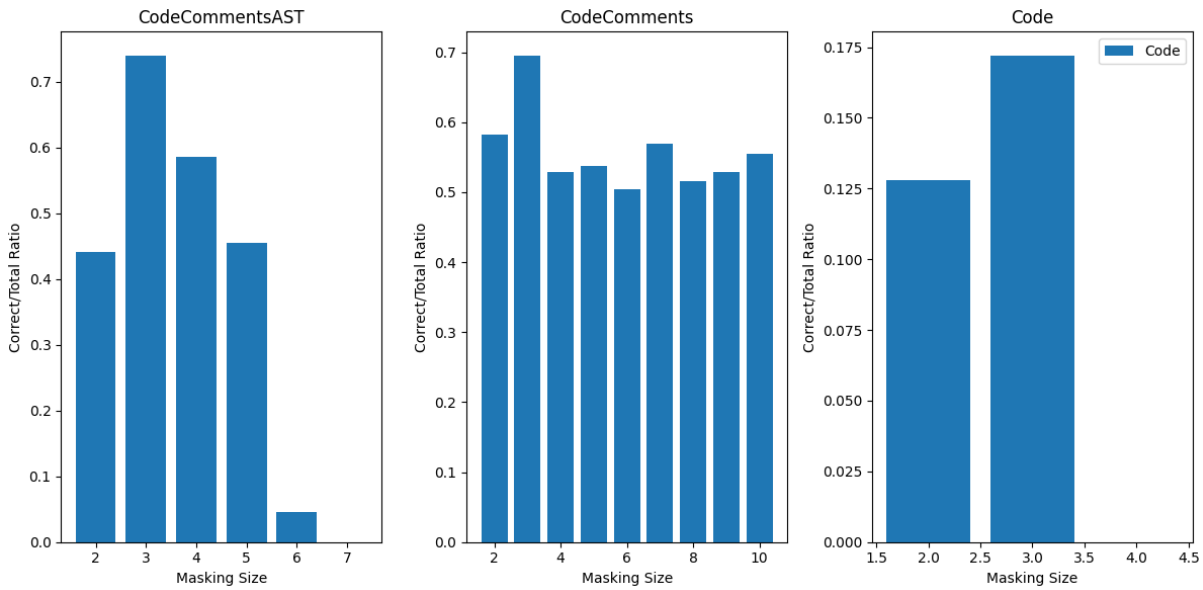


Figure 3.3: Masking Size Ratio for each model.

present in code snippets, which could give clues to the next token.

Conversely, the CodeComments model, based on CodeBERT, demonstrates a more stable performance across all masking levels, especially since it maintained its correctness ratio of 50% for all 10 tokens it was tested on. This is certainly impressive and suggests that the model is capable of predicting the next token with a high degree of accuracy, even when the context is significantly masked. This performance stability is attributed to the model's extensive training in code snippets, which enables it to effectively predict the next token based on the preceding context.

The ASTComments model performed in the middle of the two models, however, it achieved a correctness ratio of 0.7 for a mask size of 3, which is the highest out of all models. This might reflect that at that point, AST becomes the most useful metric for the model to predict the next token. However, if more of the code is masked, the ASTComments model quickly renders useless and the performance drops significantly, swiftly declining from mask 3 to mask 6 until it disappears at mask 7. This might indicate that the model is overly reliant on the AST representation, which is hard to predict when most of the code is masked.
As illustrated in Figure 3.4, the difference in performance becomes more apparent. It becomes clear that for a programmer to use these models for code completion by repeating MLM for each token, the CodeComments model would be the best choice by far, as it is the only one that reliably can predict the next token, even when the context is masked. This is a significant advantage for the model.

3.2.4 Accuracy of CodeBERT vs DocString size

After evaluating several models, CodeBERT emerged as the most promising model from the tests so far. To further investigate its performance dependencies, analysis was conducted to ascertain whether its performance is influenced by the size of the docstring. The rationale

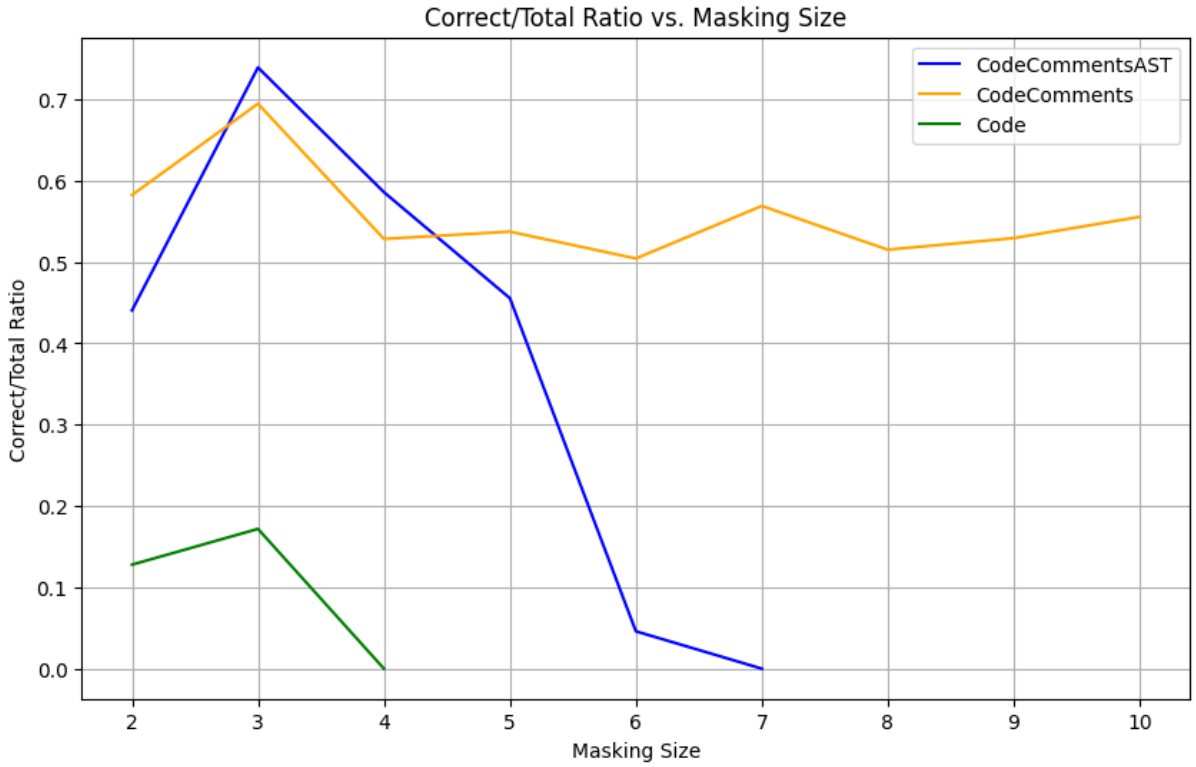


Figure 3.4: Correctness Ratio for each model.

behind this investigation was to provide insights to potential users regarding the importance of docstring size and to determine the optimal size for maximizing model performance.
The relationship between model correctness and docstring length was examined using linear regression analysis. The regression line equation, $y = mx + b$, was used to quantify this relationship, where y represents the correctness of the model, x denotes the length of the docstring, m signifies the slope of the regression line, and b represents the intercept.

The calculated slope of the regression line, $m = -0.02$, reveals a negative relationship between docstring length and model correctness. This suggests that for every unit increase in docstring length, the model correctness decreases by 0.02 units on average. The intercept of the regression line, $b = 0.78$, represents the model correctness when the docstring length is at its smallest, which was 26 in the subset used for testing.

Figure 3.5 visually depicts the results of the regression analysis, presenting the regression line alongside scatter plot data points. Surprisingly, the results indicate that the model doesn't favour larger docstrings for predicting the next token correctly; instead, it performs better with shorter docstrings, with the shortest observed being 26 characters. This observation raises intriguing questions worthy of further investigation. However, the limited dataset availability for smaller docstrings, as illustrated in Figure 2.2, poses a challenge.

It's worth noting that an interesting trend emerges after the docstring length surpasses 200 characters: the model tends to make either all correct predictions or all incorrect predictions for docstrings of that length. This variability is unusual and may be attributed to various

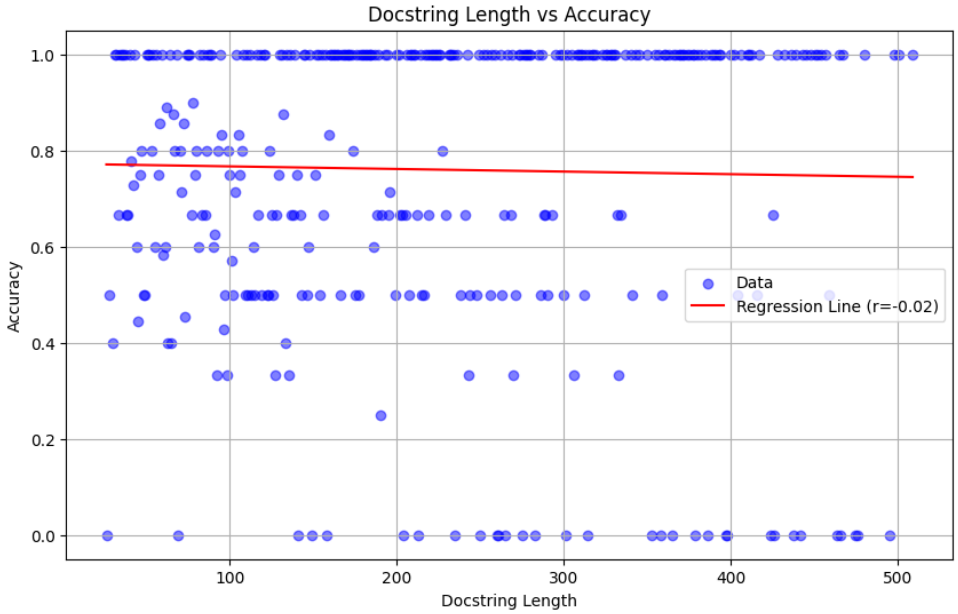


Figure 3.5: Relationship between Docstring Length and Model Correctness

factors, such as the complexity of the code tokens or the presence of specific keywords that the model struggles to predict. Additionally, the limited size of the dataset may contribute to this phenomenon, raising the possibility that it could be a coincidence.

It's important to note that the gradient of the regression line isn't particularly steep, indicating a weak correlation. Consequently, it's plausible that with a more extensive dataset, the results might differ.

3.2.5 Does large docstring slow down codeBERT

The preceding analysis suggests that the utility of larger docstrings reaches a plateau beyond a certain threshold. However, an intriguing question arises: does the model experience a slowdown in performance with larger docstrings? Such an observation could prompt developers to consider optimizing docstring lengths, perhaps by generating concise summaries, particularly if model speed is a priority.
The analysis of model performance concerning docstring length yielded intriguing insights into the relationship between these two factors. The linear regression analysis revealed a strong correlation (correlation coefficient $r = 0.93$) between docstring length and average time taken for model inference. This correlation indicates that as the length of the docstring increases, there is a corresponding increase in the time taken for the model to generate predictions.

At smaller docstring lengths (e.g., token 26), the average time taken was approximately 0.05 units, suggesting relatively quick model inference. However, as the docstring length increased, the time taken also increased at a relatively consistent rate. Notably, there were instances of outliers where disproportionately longer times were observed for larger docstring lengths, such as a time of 0.43 being predicted for a docstring size of 260. While such outliers could be attributed to several factors, including the presence of large code tokens within the example or

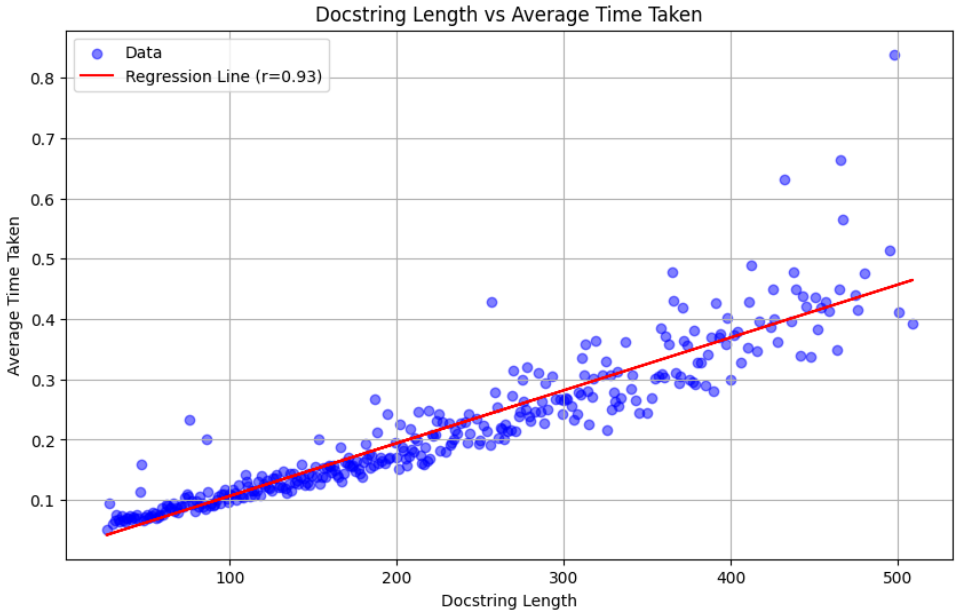


Figure 3.6: Time taken to generate code for each model on the test set with different docstring sizes.

hardware variability during experimentation, they underscore the potential impact of docstring length on model inference time.

The observed trend suggests that developers may benefit from optimizing docstring lengths to expedite model inference. However, it's essential to acknowledge that this observation may not be conclusive due to the potential correlation between docstring size and code token length. Larger functions are likely to have longer docstrings, and consequently, longer inference times, thereby confounding the relationship between docstring length and model performance.

3.2.6 Error analysis

Looking for similarities between incorrect results across models per language helps identify if the models make similar mistakes. If they do, future improvements could be addressed all at once. Conversely, if the models make different mistakes, it suggests their approaches lead to varied errors. Understanding these errors aids in future improvements. Additionally, comparing across two languages helps assess variability in these errors. When two models make a wrong prediction on the same input, it's important to determine whether their predictions align or differ.

Models Compared	Same Errors	Different Errors	Similarity (%)
C vs CC vs AST	211	165	56.12
C vs CC	240	136	63.83
C vs AST	269	107	71.54
CC vs AST	211	31	87.19

Table 3.4: Overall Similarity Between Models in Python

It appears evident from Table 3.4 that, regardless of programming language, all three models

Models Compared	Same Errors	Different Errors	Similarity (%)
C vs CC vs AST	168	128	56.76
C vs CC	191	105	64.53
C vs AST	233	63	78.72
CC vs AST	172	28	86.00

Table 3.5: Overall Similarity Between Models in JavaScript

Models Compared	Same Errors	Different Errors	Similarity (%)
C vs CC	3	239	0.80
C vs AST	14	355	3.72
CC vs AST	1	241	0.41
C vs CC	0	200	0.00
C vs AST	7	289	2.36
CC vs AST	1	199	0.50

Table 3.6: Frequency of Same Wrong Predictions Between Models

demonstrate a similar propensity to make approximately 56% of the same errors. This consistency across architectures suggests a systematic failure within the models, wherein they consistently struggle with specific code snippets. Such consistency in error patterns despite variations in input representations implies that the models' shortcomings may not solely stem from syntactic or semantic differences between the languages. Instead, it could be indicative of deeper underlying challenges, such as a lack of robustness in understanding certain coding constructs or nuances within the programming paradigms themselves. This persistent occurrence of shared errors across models underscores the need for more nuanced training data or enhanced model architectures to address these recurrent challenges effectively.

An intriguing observation from Table 3.4 is that the CodeComments and AST models' similarities for making the same errors 87.19% of the time, suggesting a shared interpretation of the training data. This alignment in error patterns despite disparate training methodologies—ASTs with NL pairs for AST and NL-PL pairs for CC, hints at a level of convergence in the models' understanding of code syntax and semantics.

All discrepancies in similarity between languages exhibit striking uniformity, with minimal deviations observed. However, the notable 7.18% disparity between the Code and AST models in both Python and JS stands out as a significant deviation. This discrepancy suggests a potential limitation in the utility of the AST model for JavaScript, as it leans more towards the RoBERTa model. Such divergence implies that ASTs may be less effective for interpreting JavaScript code snippets compared to Python. This could be attributed to inherent complexities within JavaScript's syntax and semantics, which may pose challenges for AST-based modelling approaches. Additionally, variations in the JS's nuances might require a more complex approach for tailoring an AST model for it.

It is evident from Table 3.6 that although the models frequently encounter errors on the same

code snippets across languages and architectures, they do not consistently make the same incorrect predictions. This observation is noteworthy as it underscores the distinctiveness in the types of errors each model tends to produce, irrespective of the specific architecture. The consistency in differing mistakes across models suggests that variations in input representations have a discernible impact on the models' predictive outcomes, which could reflect the nuanced differences in how each model learns and generalizes from the training data,

It's worth noting, as highlighted in Table 3.6, that only the comparison between the Code and AST models in both languages demonstrated identical predicted tokens in identical code snippets. This finding suggests that models trained solely on code input representations and those incorporating ASTs with docstring representations can yield similar predictions for the same tokens, albeit making different mistakes overall. This observation underscores the potential influence of input representations on error patterns, implying that the structural information captured by ASTs, combined with natural language context, may play a role in converging error predictions. However, it's crucial to acknowledge the limitations of these findings due to the small sample size. Further analysis and exploration are warranted to better understand the underlying reasons behind these observed patterns.

Chapter 4

Discussion

4.1 Outcome of the study

The study yielded promising results by providing a comprehensive evaluation of each model under consistent conditions, offering insights into their respective strengths and weaknesses.

4.1.1 CodeComments

It's evident that the CodeComments model, leveraging CodeBERT-base, emerged as the standout performer, surpassing its counterparts in overall accuracy across the test dataset. Particularly notable was its proficiency in actual code completion tasks, showcasing superior performance even as the size of the mask varied. This dominance underscores the efficacy of supervised learning approaches, with CodeBERT-base fine-tuned for Masked Language Modeling (MLM), using labeled data, proving to be the preferred choice for scenarios requiring effective training on limited data.

4.1.2 ASTComments

The ASTComments model, based on UniXCoder, demonstrated notable potential in terms of token generation speed, emerging as the fastest model across both languages. Additionally, its performance stood out particularly in tests involving C++, a language not included in the models' training data. This versatility suggests that the ASTComments model holds promise for projects prioritizing prediction speed over prediction accuracy. Moreover, it presents an attractive option for projects anticipating the introduction of additional languages in their codebase, as it eliminates the need for retraining—unlike models such as CodeBERT-base and RoBERTa-base, which require retraining for each language addition due to its poor accuracy on the C++ test dataset.

4.1.3 Code

The Code model, based on RoBERTa-base, consistently exhibited the poorest performance across all evaluated metrics, indicating its unsuitability for code completion tasks, even when fine-tuned on code examples. This finding contradicts the initial study's hypothesis suggesting that simpler models may suffice for such tasks; clearly, this is not the case. Instead, more complex models like UniXCoder appear to offer faster performance, while models like CodeBERT exhibit superior accuracy. As a result, it seems that under most circumstances, the simpler RoBERTa model is not the optimal choice, and instead, more nuanced models like UniXCoder or CodeBERT should be preferred for code completion tasks.

4.2 Future Work Ideas

This section outlines potential avenues for future research, including exploring alternative model architectures, investigating the impact of training processes, and considering the use of dedicated codebases for model evaluation.

4.2.1 Input types

An intriguing avenue for future exploration lies in GraphBERT, another model based on the RoBERTa architecture, as discussed in the introduction. Initial indications suggest that GraphBERT offers notable speed advantages over the AST model, raising questions regarding its performance under similar circumstances to those evaluated in this study. This is because GraphBERT is designed to process graph-structured data, which is a variant of ASTs that is much quicker to calculate. (graphbert paper)

4.2.2 Training process

Another potential area for future exploration lies in the training process of models like CodeBERT-base. While the models in this study were fine-tuned on 4 epochs, there is room to investigate whether increasing the number of epochs could lead to further performance improvements. Although previous findings from the fine-tuning process suggest that training on a smaller number of epochs may be more beneficial for tasks such as CodeSearch and Code Summarization (code bert), it remains unclear whether this holds true for code completion tasks. Therefore, conducting experiments to assess the impact of varying the number of epochs on the performance of code completion models could provide valuable insights into the optimal training strategy for such tasks.

4.2.3 Other models

Another intriguing avenue for future investigation involves exploring alternative architectures such as GPT for code completion tasks. While RoBERTa, utilized in models like CodeBERT-base, employs bidirectional training with masked language modeling, allowing it to capture context from both directions, GPT employs a unidirectional left-to-right language modeling approach, predicting the next token based solely on preceding tokens (https://arxiv.org/pdf/2305.10435). Each approach offers distinct advantages, but it remains unclear which is best suited for code completion tasks. Therefore, further exploration is warranted to compare the effectiveness of different architectures, such as GPT, in the context of code completion.

4.2.4 Dedicated Codebase

An intriguing avenue for future research involves using a dedicated codebase, such as the Linux Kernel, instead of CodeSearchNet. This would provide valuable insights into model performance under different circumstances. However, challenges such as unknown variables and the lack of fine-tuning on unseen codebases need to be addressed.

References

[1] T. Aidan. A deep dive into github copilot, 2021. Accessed on 2024-05-07.

[2] ARC. Usage, 2024. Accessed on 2024-05-07.

[3] J. Bhavin. Understanding lora — low rank adaptation for finetuning large models, 2023. Accessed on 2024-05-07.

[4] N. Brown, A. Williamson, T. Anderson, and L. Lawrence. Efficient transformer knowledge distillation: A performance review, 2023.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[6] G. Chaim. What is a long context window?, 2024. Accessed on 2024-05-07.

[7] C. Crawl. Common crawl archive. https://commoncrawl.org. Accessed on 2024-05-07.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[9] S. Eric. Generative ai coding startup tabnine raises $25m, 2023. Accessed on 2024-05-07.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

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

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

[13] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models, 2021.

[14] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In K. Erk and N. A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, Aug. 2016. Association for Computational Linguistics.

[16] JetBrains. The state of developer ecosystem 2021, 2021. Accessed on 2024-05-07.

[17] R. Johan. How github copilot is getting better at understanding your code, 2023. Accessed on 2024-05-07.

[18] H. Kevin. Fine-tuning with codeium for enterprise: Personalizing ai tools to your code, 2023. Accessed on 2024-05-07.

[19] T. Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018.

[20] W. Kwon, S. Kim, M. W. Mahoney, J. Hassoun, K. Keutzer, and A. Gholami. A fast post-training pruning framework for transformers. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[21] Z. Li, C. Xie, and E. D. Cubuk. Scaling (down) clip: A comprehensive analysis of data, architecture, and training strategies, 2024.

[22] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[23] S. Paul. Personal copilot: Train your own coding assistant, 2023. Accessed on 2024-05-07.

[24] T. Ray. Microsoft has over a million paying github copilot users: Ceo nadella, 2023. Accessed on 2024-05-07.

[25] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, 2014. Association for Computing Machinery.

[26] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6), jun 2014.

[27] K. Sinha, R. Jia, D. Hupkes, J. Pineau, A. Williams, and D. Kiela. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little, 2021.

[28] W. Takerngsaksiri, C. Warusavitarne, C. Yaacoub, M. H. K. Hou, and C. Tantithamthavorn. Students' perspective on ai code completion: Benefits and challenges, 2023.

[29] L. Torvalds. Linux kernel repository, 2024. Accessed on 2024-05-07.

[30] E. van Scharrenburg. Code completion with recurrent neural networks, 2018.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023.

[32] Y. Wang, J. Liu, D. Zhang, and X. Qiu. Reasoning about recursive tree traversals, 2019.

[33] A. Williams. Top 5 code completion services, 2023. Accessed on 2024-05-07.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

A.1.1 Limitations of AST model

In hindsight, the limited amount of training data provided for the AST model may have contributed to its lower accuracy in predicting code completion. AST-based models typically rely on extensive data to effectively learn complex patterns inherent in code structures. However, the small dataset used in this project may have constrained the model's ability to generalize and accurately predict code snippets. This limitation highlights the importance of ensuring ample training data for AST models to achieve optimal performance in code completion tasks. Future endeavors could benefit from employing larger and more diverse datasets to provide AST models with the necessary information to better understand and predict code syntax and semantics.

A.1.2 Code Conventions

During testing, the observed consistency in the program's style preferences, such as function name conventions, suggests a potential limitation in the diversity of coding styles evaluated. While the program demonstrated proficiency in adhering to specific stylistic conventions, such as consistent function naming, it may have lacked exposure to variations in coding styles, such as camel case or snake case. Exploring alternative testing methodologies that encompass a broader range of coding styles and enforce specific style conventions could provide a more comprehensive evaluation of the models' adaptability and effectiveness across different coding paradigms. This approach would allow for a more nuanced assessment of the models' capabilities and their ability to accommodate diverse coding styles commonly encountered in real-world software development projects.

A.1.3 Model Size

Despite efforts to fine-tune the models more efficiently, they still retain considerable size, with each model requiring around 500MB of storage space. This size poses challenges in terms of deployment and scalability, particularly in resource-constrained environments. While reducing model size without sacrificing performance is desirable, achieving this balance remains a significant challenge. Future iterations could explore techniques such as model distillation [4] or pruning [20] to reduce the size of the models while preserving their accuracy and effectiveness in code completion tasks. By addressing the issue of model size, the accessibility and scalability of code completion models could be significantly enhanced, facilitating their integration into a wider range of applications and platforms.

A.1.4 Cross-validation

Implementing cross-validation during training could have provided a more robust evaluation of model performance (https://medium.com/swlh/k-fold-as-cross-validation-with-a-bert-text-classification-example-4017f76a863a), especially considering the limited dataset size. Cross-validation entails partitioning the dataset into multiple subsets, training the model on different combinations of these subsets, and evaluating its performance on the remaining data. While this approach can yield more reliable performance metrics and help mitigate issues such as overfitting, it also incurs computational expenses and time constraints. Despite these challenges, incorporating cross-validation into future training procedures could offer valuable insights into the generalization capabilities of code completion models, particularly when working with smaller datasets. This could lead to more reliable and accurate models by ensuring that they are capable of effectively handling diverse coding scenarios and patterns.

A.1.5 Evaluating Lora

Exploring optimizations like the use of Lora for faster fine-tuning could have been beneficial, but the unreliable speed of Lora on the ARC cluster made it challenging to assess its effectiveness accurately. Lora, a technique for efficient distributed training, has the potential to accelerate the fine-tuning process, reducing the time and computational resources required to train models. However, due to the variability in Lora's performance on the ARC cluster, it was difficult to ascertain whether it offered a significant speedup in fine-tuning tasks.

A.2 Personal reflection and lessons learned

A.2.1 Project Management

Delving deeply into a project like this proved to be a significant commitment of time and effort, given its inherent complexity and the potential for additional layers of intricacy. It became evident that exercising restraint and proceeding methodically, one step at a time, was essential to manage the project effectively. This approach allowed for a more manageable workload and facilitated a clearer focus on incremental progress, rather than attempting to tackle all aspects simultaneously. In navigating the project's complexities, prioritizing tasks and breaking them down into manageable chunks proved instrumental in maintaining momentum and avoiding overwhelm.

A.2.2 Resource Utilisation

One crucial takeaway from this project was the realisation of the importance of leveraging available documentation and resources provided by the creators of tools like CodeBERT. Initially unaware of the comprehensive resources at my disposal, I spent considerable time on trial and error, attempting to represent data effectively. Accessing these resources from the outset could have significantly streamlined the process and enabled more efficient data representation. It became evident that familiarity with the available documentation not only saves time but also enhances the overall efficiency and effectiveness of the project. Moving

forward, prioritizing thorough exploration of existing resources and documentation will be essential for optimizing project workflows and achieving more robust outcomes.

A.2.3 Codebase

Training on dedicated codebases could potentially yield more intriguing results; however, the project's dynamic nature and constant changes would have complicated comparisons and drawn conclusions. Introducing a new codebase could introduce confounding variables, making it challenging to attribute observed differences solely to the system's design. Thus, maintaining consistency in the dataset and methodology was crucial for ensuring meaningful and interpretable results throughout the project. While training on a dedicated codebase may offer valuable insights into model performance under different circumstances, it would also require careful consideration of factors such as dataset diversity, model generalization, and experimental reproducibility.

A.3 Legal, social, ethical and professional issues

The issues that arise from this project typically revolve around either the data or how will programmers use this technology. The questions around data question its locality, the quality of the data and the privacy of the data. Programmers, one they have this tool, could in ideal circumstances use it for what it was designed for, helping them be more productive and minfdul of their time. However, this can be easily abused and it is not clear how to guard against this.

A.3.1 Legal issues

CodeSearchNet is a dateset that derived its content from open-source repositories, as mentioned in (section). The problem is that the dataset doesn't contains a licence field, which would specify for each piece of code, whether the original creations have given explicit permission for it to be used for commercial reasons. It most cases, it could be assumed it is but if this model would be used for commercial reasons, there needs to be explicit confirmation that is it legal to do so rather than assuming. The codeSearchNet is put under the MIT licence, so it is free to be used for commercial reasons but the code inside it might not be. Another problem is that the code itself often uses various libraries and other code that might not be under the MIT licence, so it is important to understand the licences of the code that is being used. This gap leaves the door open for potential legal issues if the model is used for commercial reasons. The same can be applied to copyright law and intelectual property rights. Since its infesable to scan the dataset one code snippet at a time to decide if its fine or not, there is no real way to mitigate this issue. An exception would be running the same project but with a different dataset, such as the GitHub Copilot dataset which specifices the licences of each code snippet and declares that it never used copyrighted code.

A.3.2 Social issues

impact of code completion tools on developer productivity and software quality since it will just predict, there is a risk of it being used to just copy and paste code without understanding

it There is a risk that developers will use this tool to just copy and paste code without understanding it, which could lead to poor quality code and potential security vulnerabilities. So despite saving the programmer time, it would actually cost them more time in the long run and might promote bad coding practices that could have raminfications on the whole sociaty. For example, One way to mitigate this issue is the adoption of AI generated code detectors, that can detect if the code was generated by an AI model and flag it for review by a human developer. This would help ensure programmers don't push working bad code to production and allow the reviews of the code to be more thorough when it detects AI generated code.

A.3.3 Ethical issues

Since most data is from various libraries and usages, the api calls performs badly on since its so rare There are biases in the dataset due to the nature of its contstuction. Some of the code is object code so there is a bias to often predict self as the first parameter of a function as detailed in section []. Other biases include arise from type of code in the dataset. Typically, code that involves obscure API calls or rare libraries will perform worse than code that uses common libraries and functions, just by the nature of the fact that there is less data on them. This is a classic example of dataset imbalance, where where the prevalence of certain classes overwhelms the minority classes, leading to biased predictions favoring the majority classes. As a result, the model prioritizes accuracy metrics at the expense of fairness and inclusivity, perpetuating existing biases present in the training data. As such, if a developer that uses a rare library might be excluded from being able to use this tool. One way to mitigate this is Data Augmentation where I could generate more data for the rare libraries and api calls to make the model more robust to them.

A.3.4 Professional issues

In deploying AI models for predicting code, ensuring transparency, accountability, and responsible use of technology is paramount. It's crucial to recognize and comprehend the limitations of the model, rather than treating it as a black box solution. This is particularly significant given that the predicted code may be indistinguishable from code written by developers. When considering accountability for the generated code, questions arise about who holds responsibility: the developer, the model itself, or even the developer whose code was included in the dataset used to train the model. Moreover, ethical dilemmas emerge when considering whether individuals can be held accountable for intentionally providing false comments to mislead the model, raising concerns about the integrity and trustworthiness of the generated code. Addressing these professional issues requires clear guidelines, ethical standards, and collaborative efforts among developers, researchers, and organizations to ensure the responsible deployment and usage of AI technologies. This is still an active research area and there is no clear answer to this problem.

Appendix B

External Material

CodeSearchNet, CodeNN Dataset was used for this project.

Appendix C

Additional Results

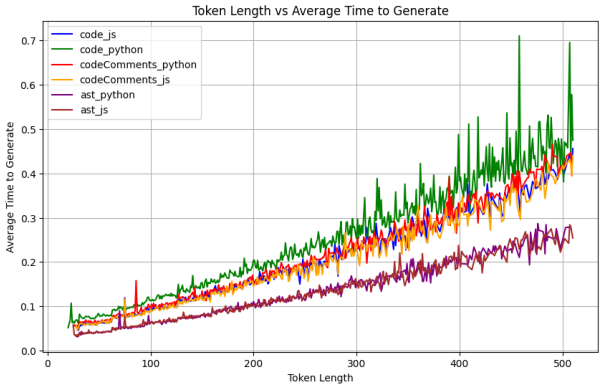


Figure C.1: Not Smoothed Time