Source Code Auto-completion using Deep Learning Models

Madhab Sharma

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Source Code Auto-completion using Deep Learning Models

Dissertation submitted in partial fulfillment

of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering
(*Specialization*: *Computer Science*)

by

Madhab Sharma

(Roll Number: 219CS1143)

based on research carried out

under the supervision of

Prof. Tapas Kumar Mishra

May, 2021

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Department of Computer Science and Engineering
National Institute of Technology Rourkela

May 29, 2021

Certificate of Examination

Roll Number: *219CS1143*
Name: *Madhab Sharma*
Title of Dissertation: *Source Code Auto-completion using Deep Learning Models*

We the below signed, after checking the dissertation mentioned above and the official record book (s) of the student, hereby state our approval of the dissertation submitted in partial fulfillment of the requirements of the degree of *Master of Technology* in *Computer Science and Engineering* at *National Institute of Technology Rourkela*. We are satisfied with the volume, quality, correctness, and originality of the work.

Tapas Kumar Mishra
Principal Supervisor

External Examine

Durga Prasad Mohapatra
Head of the Department

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Prof. Tapas Kumar Mishra
Professor

May 29, 2021

Supervisors' Certificate

This is to certify that the work presented in the dissertation entitled *Source Code Auto-completion using Deep Learning Models* submitted by *Madhab Sharma*, Roll Number 219CS1143, is a record of original research carried out by him under our supervision and guidance in partial fulfillment of the requirements of the degree of *Master of Technology* in *Computer Science and Engineering*. Neither this dissertation nor any part of it has been submitted earlier for any degree or diploma to any institute or university in India or abroad.

Tapas Kumar Mishra
Professor

Dedication

This thesis is dedicated to my mother, father and my sisters.

Madhab Sharma

Declaration of Originality

I, *Madhab Sharma*, Roll Number *219CS1143* hereby declare that this dissertation entitled *Source Code Auto-completion using Deep Learning Models* presents my original work carried out as a postgraduate student of NIT Rourkela and, to the best of my knowledge, contains no material previously published or written by another person, nor any material presented by me for the award of any degree or diploma of NIT Rourkela or any other institution. Any contribution made to this research by others, with whom I have worked at NIT Rourkela or elsewhere, is explicitly acknowledged in the dissertation. Works of other authors cited in this dissertation have been duly acknowledged under the sections "Reference" or "Bibliography". I have also submitted my original research records to the scrutiny committee for evaluation of my dissertation.

I am fully aware that in case of any non-compliance detected in future, the Senate of NIT Rourkela may withdraw the degree awarded to me on the basis of the present dissertation.

May 29, 2021
NIT Rourkela

Madhab Sharma

Acknowledgment

I would like to thank my supervisor Prof. Tapas Kumar Mishra for providing me with constant support and help throughout the duration of the research. I would also like to thank all the faculty members of our department, my friends and family members for their constant support.

May 29, 2021
NIT Rourkela

Madhab Sharma
Roll Number: 219CS1143

Abstract

In the software development process, code auto completion is a requisite tool for any developer, as it increases productivity and it can save countless hours while writing code. This thesis presents literature survey on the domian source code auto completions and it also discusses various methodologies used in various papers along with there results. This thesis also present various methodologies for source code auto-completion using various Deep Learning model, for Python and CSharp Programming Languages. These models use the code sequences to train and evaluate, as other code structures like semantic structures, leads to various overhead, which was not feasible in the resource-limited environment. The models proposed for this task, are CodeGPT [1] from Microsoft, Roberta [2] from hugging face [3] and GPT2 [4].

Various Data-set strategies were also used while training the Roberta Model on CShrap data-set such as (1) treating the whole code file as a single line, (2) using each line as single individual inputs, and (3) tokenizing the codes snippets before feeding into the models. Two types of training approaches were used for the CShrap data-set with GPT2 model, a naive and domain-specific. Domain-specific performed exceptionally well compared to other models on various data-sets. For the Python data-set an overall accuracy of 71% was observed and for the CSharp a PPL of 2.14 and 4.082 on training and evaluation data-set. It was found that domain-specific code corpus performed significantly well compared to other types of the corpus, for any given model.

***Keywords*: *Source-code auto completion*; *CodeGPT*; *Roberta*; *GPT2*;.**

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In the software development process, code auto-completion is a requisite tool for any developer, as it increases productivity and it can save countless hours while writing code. Source-code completion is the task of predicting the code sequences while writing code. This task can be automated using various techniques such as (i) statistically analysing code sequences, (ii) using neural networks, (iii) NLP or deep learning models, to perform the auto-completion. The goal is to provide rapid and correct suggestion for next code sequence. These auto-complete models can be integrated into various IDEs and online code editor to increase efficiency and productivity of developers.

Previously various methods have been proposed for generating code, e.g. RNN(LSTM) [5] based, LSTM with attention [6], transformers [7] etc. Various code structures have been used; as input data to these model some of them focuses on the raw source text, some are based on token streams. Recently abstract syntax trees are gaining attraction, and other structural/semantic meaning of code snippets to capture context and predict the next code tokens or streams. Even various IDEs from early on have implemented some sort of auto-completions. For instance, Eclipse uses type-based auto-complete features for next token suggestions.

With the introduction of transformers [8], which gives state of art performance in the Natural Language Processing domain, code completion task has become more concrete and efficient. Moreover, with the advent of various transformer variant models like BERT [9], GPT [10] and XLNET [11], more progress have been made regarding different source code tasks.

With the introduction of Transfer learning [12], the application of fine-tuning a trained model to the different related domain has become a standard way of research and implementation. Transfer learning is a process, where a model is trained with very large training data over a long period of time on high-end machines and is made available and can be fine-tuned on different tasks. This process of retraining a fully trained model on a downstream task is known as fine-tuning [13]. This process has become the core tool for research work and model implementation for industries and individuals, as it saves countless

hours and compute resources. The cost of running such tests on model training experiments used to be enormous: this could be eliminated after the introduction of transfer learning.

1.2 Various techniques to solve source code auto-completion

Various techniques are used to solve the problem, they are categorised into following categories.

1.2.1 Statistically analysing code sequences.

In statistical analyse technique code token or code sequence are predicted by gathering the statistical information the code patterns. Explicit statistical models are used learn and predict next token. These models are very compact in size and are basically used in many code editors. But they are limited to how much degree of pattern they can learn and predict. The model fails to understand various complex code patterns.

1.2.2 Neural Networks

Neural Networks like CNN are used are used to predict next code token. The code tokens are converted into embedding and fed into the neural network model and model predicts the next token. Its a better approach than statistical analysis model because, better relationship between the code tokens can be learned by the neural network models. But the model size increase substantially. They are preferred in online editors, as the model can be deployed as software service in cloud. With various advancement in CNN, using method like quantisation the models can now be compressed to a very significant amount. In this methods weights are converted from 32bit to 8bit, along with various other optimisation techniques the model size are squeezed to fit into modern IDEs.

1.2.3 NLP deep learning models

Like natural languages like English, programming languages also are build of various grammar syntax. Natural language processing is an area where various techniques are used to understand the complex nature of natural languages. These techniques can also be used in programming languages.various NLP Deep learning models like RNN, LSTM, Transformers etc can also be use to solve various challenges in programming languages. LSTM with attention model gained a lot of popularity in natural language processing as it could work with longer sequence of words and also understand various complex context in a language. Then came the GPT and BERT models which changed the how NLP were used. Also introduction of transfer learning any individual could used these highly efficient trained model in solving

task without need to re trained the model from scratch. The same techniques are used in the case of programming languages tasks to create highly efficient models.

1.2.4 Using various code structures

Using various code structures syntax or semantic, can also help in understanding a piece of code in more details. Code sequences or token sequences and be used to capture the various patterns inside a piece of code. Its fast and easily fits the NLP models for natural languages. But it has a problem, it is not generalized, example if a model trained using code sequences for JAVA cannot be used for Python languages, because the syntax differ.

On the other hand, semantic structure of code like Abstract Syntax Trees, Graph Structures etc can capture the deep patterns of code. And as any language can be converted to ASTs, the model can be generalised to work with different languages. This approach has also have issues, as these models rely on semantic structure to predict the patterns, this makes the model complex to implement and train. The data set required is huge as a single piece of code can have multiple paths to traverse in a single AST.

1.3 Abstract Syntax Trees

Abstract Syntax trees are the most widely used semantic structure to capture the semantic meaning of code snippets, as they can capture context between various code lines . The model train with ASTs can be used with various programming languages, the only thing required would be APIs to convert a programming language code snippet to a general AST style. Each node in a ASts denotes a construct appearing in source code. Figure 1.1 shows the code snippets along with its AST.

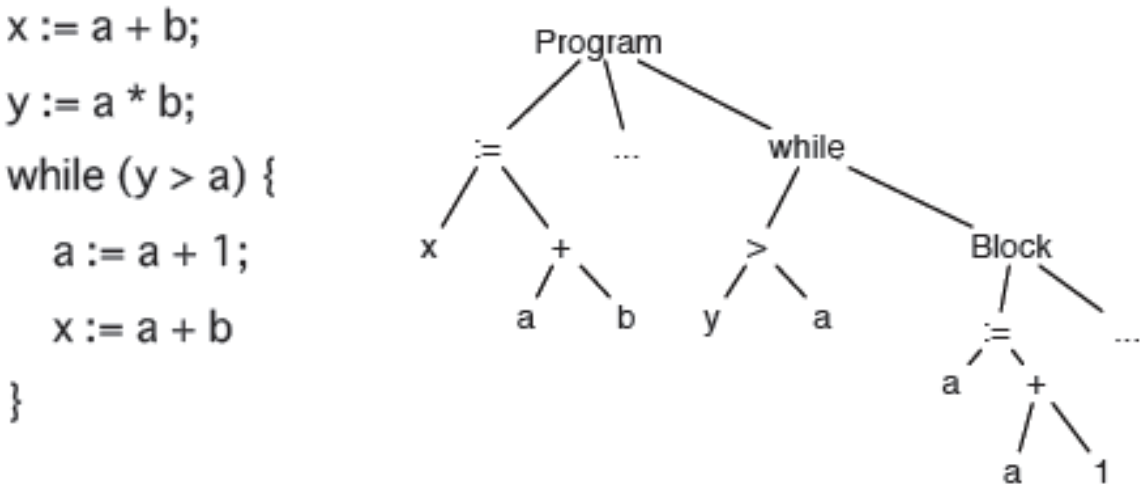

Figure 1.1: Code Snippet(left), Abstract Syntax Tree(Right).

1.3.1 Serializing ASTs

Only having the ASts is not enough we need a way to serialize the ASTs. We can have various way to serialize an ASTs, like depth first search node paths, paths from root parent to each leaf nodes etc.

1.4 Motivation

Source-Code auto completion is a very important and complex challenge. One because of the large diversity in code writing style i.e code in same language can also differ from projects to projects, this makes this task a challenging one. Second, the usefulness of this tools in development has become important because it reduces time and effort wasted in searching for code snippets . Also making the model small and robust helps developers to add this to their development environment with ease. These models can be useful in various tasks like code summarisations, code documentations, auto bug detection, code reviews etc.

1.5 Objective

The first objective is to find out how a different model like CodeGPT from CodeGLUE [1] and Roberta [2] perform on various Python and CSharp data-set, using fine tuning.

The second objective is the find if different structure of source code snippets makes any improvement.

And finally, the last objective is finding out if limiting source data-set to a particular domain i.e taking source code from only a particular project, gives any performance boost.

To summarise following are the objectives:

1. Fine tuning CodeGpt on python data-set.
2. Code prediction using Roberta on CSharp source code.
3. Code prediction using GPT2 model trained on CSharp source code.

1.6 Organization of Thesis

The first chapter in this thesis starts with the introduction of the auto-source code completion, followed by a brief summary of related work on this area and proposed methodologies and results. A short outline of the content of each chapter are mentioned below:-

i. **Introduction**
   This chapter briefly describes about the auto-source code completion, followed by various strategies used in past and currently to solve this challenge. It also outlines the motivation and objective of this work.

ii. **Literature Review**

A brief summary on the relate works are summarized in this chapter. This chapter list results and limitations of various works.

iii. **Framework for Source code autocompletion**

This chapter outlines the proposed methodologies for the source code auto-completion tasks. It describes how various datatset were built and various pre-processing steps used to refine the datasets. It briefly summarises why different approaches are taken in this work, with different model architectures used.

iv. **Results**

This chapter describes the results of the proposed framework with results listed in tables and charts.Also summarises the training and testing results under various models on various datasets. At the end section we compare out results with the various other work to evaluate the performance of the models.

v. **Conclusion and Future Scope**

This chapter presents the conclusion our this work, the limitations and the areas that can be improve in future work.

Chapter 2

Literature Review

2.1 Overview

This chapter contains a brief review of source code auto-completions. Previously various methods have been proposed using various machine learning and deep learning approaches.

2.2 Related Works

Previously various methods have been proposed for source code auto-completion tasks.

2.2.1 code2vec: Learning Distributed Representations of Code

In *code2vec: Learning Distributed Representations of Code [6]* paper, the authors has presented a neural network framework to to learn code embedding. Embedding a code snippet as a vector has a variety of machine-learning based applications, since machine-learning algorithms usually take vectors as their inputs. Some of the application where this learned embeddings can be used are:

1. Automatic code review
2. Retrieval and API discovery

The authors have deliberately picked the difficult task of method name prediction, for which prior results were low as an evaluation benchmark.

They have used the semantic structure of code, by feeding serialized ASTs to represent into the network, to represent a snippet in a way that enables learning across programs. The network itself contains LSTM layers with paths-attentions, address learning which parts in the representation are relevant to prediction of the desired property, and learning what is the order of importance of each part. A data-set of almost 10k Java GitHub repositories, was used during training and evaluation purpose. A total of 1.7M files were used during training and 50k for validation and testing process.

Results

A precision value of 63.1, recall of 54.4 and F1 score of 58.4 was achieved when evaluating the model on the full test set (of size 50k files).

Limitations

The model suffers from Close Labels vocabulary, which means that the model can only predict labels that were present in the vocabulary during training time. Another limitation with the model is a dependency on variable names, as the model was trained on top-stared projects. When given an obfuscated variable names, the model performs poorly.

2.2.2 code2seq: Generating Sequences from Structured Representations of Code

code2seq: Generating Sequences from Structured Representations of Code [15] is an incremental paper of code2vec model which covers the natural language sequence generation from a code snippet. It focuses on Neural Machine Translation(encoder-decoder) architecture to generate texts. It represent a given code snippet as a set of compositional paths over its abstract syntax tree (AST), where each path is compressed to a fixed-length vector using LSTMs . During decoding, CODE2SEQ attends over a different weighted average of the path-vectors to produce each output token, much like NMT models attend over token representations in the source sentence.

The paper shows effectiveness code2seq model over two tasks:

1. code summarization, where it predicts a Java method's name given its body.
2. code captioning, where it predicts a natural language sentence that describes a given C# snippet.

Results

For Code summarization, the LSTM model is trained on three java corpus (small, medium and large) generating F1 score of 50.64, 53.23 and 59.19 for small, medium and large java data-set respectively.

For code captioning task, the model achieves a BLEU score of 23.04, which improves by 2.51 points (12.2% relative) over CodeNN, whose authors introduced this dataset, and over all the other baselines, including BiLSTMs, TreeLSTMs and the Transformer, which achieved slightly lower results than CodeNN.

Limitations

The limitation of this model is that it doesn't consider long-distance context into account; rather it is limited to surrounding contexts.

2.2.3 Structural Language Models of Code

In *Structural Language Models of Code* [16], addresses the problem of any-code completion – generating a missing piece of source code in a given program without any restriction on the vocabulary or structure. The authors have used an Uni-directional LSTM to encode AST paths extracted from source code snippets. A transformer encoder is used to contextualize the encoded paths. It then uses attention over the path contexts. Soft-max is used over the node embedding to predict an AST node. It leverages the uni language model for auto-code completion which are restricted to language-specific vocabulary.

Results

The model shows 18.04 acc@1 and 24.83 acc@5 on the java corpus. For the CSharp corpus, it shows 37.61 @acc1 and 45.51 acc@2. acc@n means average accuracy on top n probabilities for a prediction.

Limitations

Limitation of the model is, it works with only one function or class. And the model performs poorly for complex expressions.

2.2.4 A transformer-based Approach for Source Code Summarization

In *A transformer-based Approach for Source Code Summarization* [17], the authors use a transformer [8] model with self-attention mechanism to capture long-range dependencies for code summarization. Source-code token stream is fed to the network as input samples.

Results

The proposed model has BLEU [18] score of 44.58 on Java data-set and BLEU score of 32.52 on Python data-set. The authors also experimented with respect to relative position and also with copy attention, which didn't perform well compared to the proposed model.

Limitations

Limitation of this model is that it doesn't consider any structural aspect of the code. Moreover, the model was only tested on Java and Python data-set.

2.2.5 Pythia: AI-assisted Code Completion System

Pythia: AI-assisted Code Completion System [19] uses the Abstract syntax trees corresponding to code snippets of python source code for training the model. They have introduced a ranking system for prediction results. It uses PTVS [20] parser from Microsoft to parse AST from source code snippets. It uses LSTM with predicted embedding model

for the code completion task. It also uses *Neural Network Quantization* which reduces the number of bits to store the weights(To 8-bit, integer representation): this leads to the reduction of the model from 152MB to just 38MB.

Results

It achieve acc@1 of 0.71 and acc@5 of 0.92 with MRR of 0.814.

Limitations

Limitation of the quantization process is that it reduces the top-5 accuracy from 92% to 89% though substantially reducing the model size. Currently, it only works work python source code.

2.2.6 IntelliCode Compose: Code Generation Using Transformer

In *IntelliCode Compose: Code Generation Using Transformer* [21] the authors use a pre-trained, multi-layer transformer model of code: GPT-C. GPT-C is a variant of GPT-2, trained from scratch on a large data-set. It uses the source code data as a sequence of tokens, the output of a lexical analyzer, as input samples. It also introduces the multilingual model by extracting a shared sub-token vocabulary from various programming languages.

Results

With the model size of 366M, GPT-C on CSharp scored a PPL [22] score of 1.91 and a PPL score of 1.82 on python corpus. On MultiGPT-C task (C#, Python, JS, TS) with PPL score of 2.01 in 374M model size.

Limitations

However, it doesn't consider any structural aspect of the code, like Abstract Syntax Trees or Concrete syntax tree: according to the authors, it introduces additional overhead and dependencies which reduces the efficiency of the code completion system.

2.2.7 Fast and Memory-Efficient Neural Code Completion

The authors in *Fast and Memory-Efficient Neural Code Completion* [23] suggest a fast and small neural code model, which is not memory hungry like neural model usually are. They achieve this state of the art by utilizing candidate suggestions produced by the static analyser. It reduces the need for maintaining memory-hungry vocabulary and embedding matrix. It is based on 4 modules, working together. The modules are (a) Token Encoder, (b) Context Encoder, (c) Candidate provider and (d) Completion ranker. The authors have experimented with Token, Subtoken, BPE [24] and Char type encoders for Token Encoders. GNU and LSTM were mostly preferred as they have less memory footprint than CNN and transformer

context encoders for context encoding. For candidate provider, STAN(static analysis-based) was used which performed better than vocabulary candidate provider. The model was trained and tested on python source code extracted from GitHub repositories, and to pre-process the data PTVS was used.

Results

The best model 3MB and 50MB (BPE, GRU, STAN) was models of choice which achieve 66% (resp. 70%) recall@1 with about 6ms (resp. 8ms) of average suggestion time.

Limitations

2.2.8 Code Prediction by Feeding Trees to Transformers

In *Code Prediction by Feeding Trees to Transformers* [25], the authors investigate various ways of using Transformer architecture to produce good accuracy for source-code prediction. The authors of this paper have limited their investigation on techniques that revolve around ASTs. They have proposed two ways to serialising ASTs to capture the partial structure rather than just jamming raw ASTs into transformers. The two ways of serialization techniques used are as follows.

1. Tree traversal order.
2. Decomposing trees into paths.

In tree traversal order serialization method, TRAVTRANS is proposed, which is depth-first-search order or pre-order traversal over the AST. In Decomposing trees into paths serialization, PATHTRANS is proposed, which is based on creating paths from the root node to terminal nodes. In this approach, the path from the node the parent node of the leaf is encoded with LSTM block and the leaf token embedding are concatenated with the root-path block and are fed to the transformer network. They found that TRAVTRANS outperformed various other code-auto completion techniques. PY150 data-set was used for the model(it consists of parsed ASTs trees). For the transformer model, GPT-2 small is used, adapted from pytorch version.

Results

The reciprocal rank improvement of TRAVTRANS compared to DEEP3 [26] is from 43.9% to 58.0%. And with CODE2SEQ compared with TRAVTTRANS is 43.6% to 58.0%.

Limitations

Though TRAVTRANS outperforms other representation, the structural relation between the nodes is still not retained, as cited by examples [25] in the paper by the authors.

Table 2.1, summarises the overall architecture and findings from the related works.

Related Workds	Architecture	Results
code2vec	ASTs + LSTM + attensions	precision : 63.1 , recall: 54.4 , F1 : 58.4
code2seq	ASTs + LSTM + attensions	F1 : 59.19, BLEU : 23.04
Structural Language Models of Code	ASTs + LSTM + attensions	acc@1 : 18.04 , acc@5: 24.83
A transformer-based Approach for Source Code Summarization	Code sequence + Transformers	BLEU : 44.58(java) and 32.52(python)
Pythia	ASTs + LSTM + Ranking + NN Quantization	acc@1: 0.71 , acc@5 : 0.92
IntelliCode Compose	Code Sequence + GPT-C	ppl : 1.91(c#), 1.82 (python), 2.01 (c#,py,js,ts)
Fast and Memory-Efficient Neural Code Completion	Code Sequence + GRU + STAN	acc@5: 0.90, recall@1: 66%
Code Prediction by Feeding Trees to Transformers	ASTs + path traversal	recipocal rank : 58%

Table 2.1: All the models with their architecture and results.

Chapter 3

Framework for Source code auto-completion

3.1 Overview

This section describes the various proposed methodologies and approaches: extraction and pre-processing of the data-set, and model details along with parameters used to train them.

3.2 Methodologies

Three methodologies are proposed,

1. Fine-tuning CodeGpt on python data set.
2. Training Roberta model in three different CSharp data-sets.
    (a) Set 1, on code streams.
    (b) Set 2, on code streams broken into statements in each line.
    (c) Set 3, tokenized version of Set 1.
    (d) Set 4, tokenized version of Set 2.
3. Training GPT2 model on two CSharp domains.
    (a) Naive approach.
    (b) Domain Specific

Training the transformers models requires a large amount of computing power and an enormous amount of time. The Roberta, CodeGpt and GPT2 model were all trained in google Colab, with 16GB of VRAM and 10GB of RAM. The machines had CUDA enabled.

3.2.1 Data-set

For using NLP models on the source code, numerous training-testing-validation samples are required. For the python data-set, PY150 data-set, originally used in [14], is used. This large python corpus contains around 100k training and 50k testing samples, respectively. However, using 100k samples requires a huge amount of computing resources and training on such large corpus was not feasible on a limited resource, like Google Colab. So, the large data-set is broken down into chunks of 2.5k, 5k, 7.5k, 10k, 12.5k, 15k, 17.5k and 20k, for training purpose. 50k testing samples were used for each data chuck for evaluation. Breaking into such chunks enabled to run the model efficiently on the limited resources available.

For the C# data-set in Roberta Model, top 25 CSharp source code GitHub repositories are used. The source code filenames with .cs extension were extracted. Each file was initially represented as a single line in the main corpus file. The CSharp corpus contains around 23k source codes for training, 7k for testing respectively. After the codes were collected, a pre-processing step was performed on each source code, which is discussed in the next section.

Firstly, comments were removed from each source files.Then they were further pre-processed to form four types of data-sets,SET 1, SET 2, SET 3 and SET 4. In first set, all the statements were combined to form a single big line of code. In the second approach, each single statement in each file was considered as single input sample. Third and fourth approach was to change the code text streams to code token streams, for first and second set. A comprehensive explanation of pre-processing steps is discussed in methodology section. The sample size used for training the Roberta is 23k samples, with 7k as a test and validation data-set.

For the GPT2 model, in the third approach, SET 1 used in the Roberta model was used in the naive approach. And for the Domain-Specific approach, a new data-set was created from a particular CSharp project domain. Code repositories consisting of codes related to UNITY3D (a game engine based on C#) was used as domain. Both the data-set have roughly the same size, around 30k 36K, samples. The training hyper-parameters were same for both the data-set. Both the time the model was trained for 5 epochs. As mentioned earlier for the Naive Approach, the data-set consists of various top GitHub repositories on CSharp source code, all belonging to different domains.

Two types of source corpus were used in this approach. First, the SET 1 data-set, which consists of all the refined CSharp code snippets from various GitHub repositories (same as the one used in the Roberta Model).This approach is termed as naive approach in this paper. Second, a whole new data-set was created from CSharp source code library of a specific domain: UNITY3D repositories. This was to check if the domain-specific corpus helps in more accurate predictions.

The motivation for the third approach was to check the performance of GPT2 on sequence

prediction on CSharp data-set since GPT2 is good for language modelling tasks. Moreover, it is interesting to study how the same model would perform when trained on same domain code-base. As it was found out, the naive approach, where source codes are from various not related GitHub repositories, though performed very well on training sets performed poorly on evaluation data-set. The main reason for this behaviour was the fast-changing style of code between GitHub repositories. As, the naming convention for the variables, functions and classes changed from repositories to repositories, the model under-performed. The domain specific study was done to validate this hypothesis.

In case of Domain-Specific study, source codes related to UNITY3D were used. All freely available CSharp code repositories from GitHub, were used to build the corpus.

3.2.2 Pre-processing

For CodeGpt, the pre-processing steps for python includes, writing each python code within <s>and </s>tags. <EOL>tags are used to mark the end of the line, as the whole file code is represented in a single line in the python corpus, as shown in Listing 3.1.

```
1. <s>import numpy as np <EOL> def incmatrix(genl1,genl2):... </>

2. <s>import torch <EOL> from torch.utils.data import Dataset <EOL>...</>

...
```

Listing 3.1: Python Sample

For CSharp corpus, the data-set was pre-processed with four different sets: (1) one set contains all the source code as it is in a single line; (2) in the second data-set, source code is split line by line meaning each line contains a single statement; (3) the third data-set contains the tokenized streams of the first set; (4) the fourth set contains tokenized version of second set. From each set, comments were removed. The splitting of the source code in a file line by line and considering each single line as single input sample, rather than the whole source file as the single input, was taken because, Roberta can only handle, max token length of 514, larger than that get truncated, which meant a loss of context from each input sample. Figure 3.1 and Figure 3.2 shows the prepossessing pipeline and set representation for Csharp corpus, respectively. Truncating samples to only 514 size was done, to prevent CUDA memory error because the Roberta only uses 514, of input stream to encode, thus saving VRAM memory.

Another reason for split the source code line by line was because the codes were extracted from popular GitHub repositories, they contained a large amount of import statements eg *Using* statements, before the actual code blocks. Addition to this with the truncation nature of the model i.e excepting only 514 tokens as input, meant that less relevant information was fed to the model. Hence, to leverage this issue, the whole file codes were broken line by line, and each line was now a input to the model. Also, as the splitting of codes into lines wasn't perfect, as some class definition or function were very

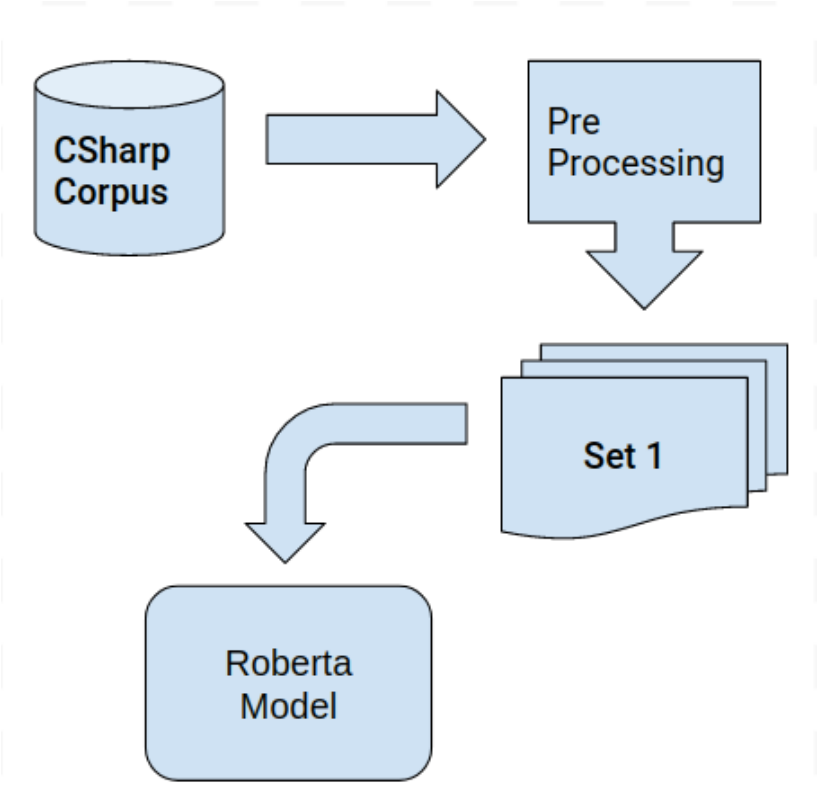

Figure 3.1: Pipeline for CSharp codes with Roberta

large. Lines which had 5 code tokens or less were discarded, during the prepossessing step. Yet, another issued prevailed, as the variable naming style varied from the source file to file. The model didn't perform well as expected on the identifier. To tackle this problem, the tokenized version of the source code was used. Where the source codes of the entire file were tokenized using a tokenized tool [27]. And were then spilt into statement wise tokens and then fed to the model. The same procedure of discarding lines with tokens less than or equal to 3 was used. Listing 3.2, 3.3, 3.4 and 3.5 show the SET 1, SET 2, SET 3 and SET 4 representations, respectively.

```
1
2 collection . Add ( name ) ; Assert . Empty ( removeEvent . EventName ) ;
3
4 name = "Int32" ; yield return new object [ ] { " Int32 " , " Name " } ;
5
6 ...
```

Listing 3.2: CSharp Set 1 Sample

```
1
2 collection . Add ( name ) ;
3
4 Assert . Empty ( removeEvent . Event . EventName ) ;
5
6 name = "Int32" ;
7
8 yield return new object [ ] { " Int32 " , " Name " } ;
```

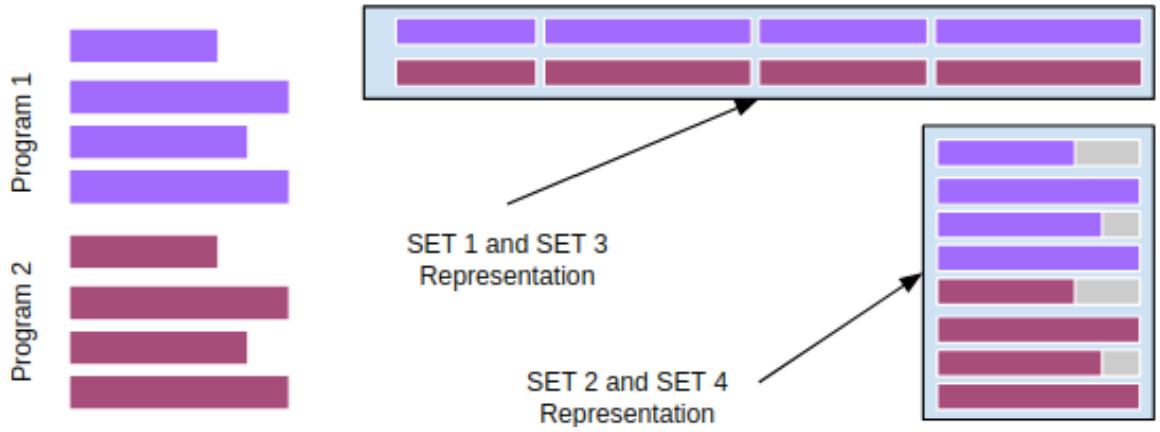

Figure 3.2: Pipeline for CSharp codes with Roberta

```

...
```

Listing 3.3: CSharp Set 2 Sample

```

ID . ID = ID ; ID . ID ( ID . ID . ID ) ;

ID = STRING_LITERAL ; yield return new object [ ] { STRING_LITERAL ...} ;

...
```

Listing 3.4: CSharp Set 3 Sample

```

ID . ID = ID ;

ID . ID ( ID . ID . ID ) ;

ID = STRING_LITERAL ;

yield return new object [ ] { STRING_LITERAL , STRING_LITERAL} ;

...
```

Listing 3.5: CSharp Set 4 Sample

SET 1, SET 2, SET 3 and SET 4 are used for training and evaluation of Roberta model in second methodology. For the naïve approach, in CSharp with GPT2 model,which is the third methodology proposed, same pre-processed data-set, SET 1 was used for training and evaluating. For the domain specific approach, pre-processing used for SET 1 in Roberta model was used.

3.2.3 Models

This section describes the various deep learning models used in this paper. The CodeGpt, a pre-trained model, was used for the Python corpus in first methodology.The Roberta model was used for second methodology .And GPT2 was used for the third. Both GPT and Bert

are generalized transformer models. GPT being the decoder section of the transformer and BERT being the encoder section as seen in Figure 3.3.

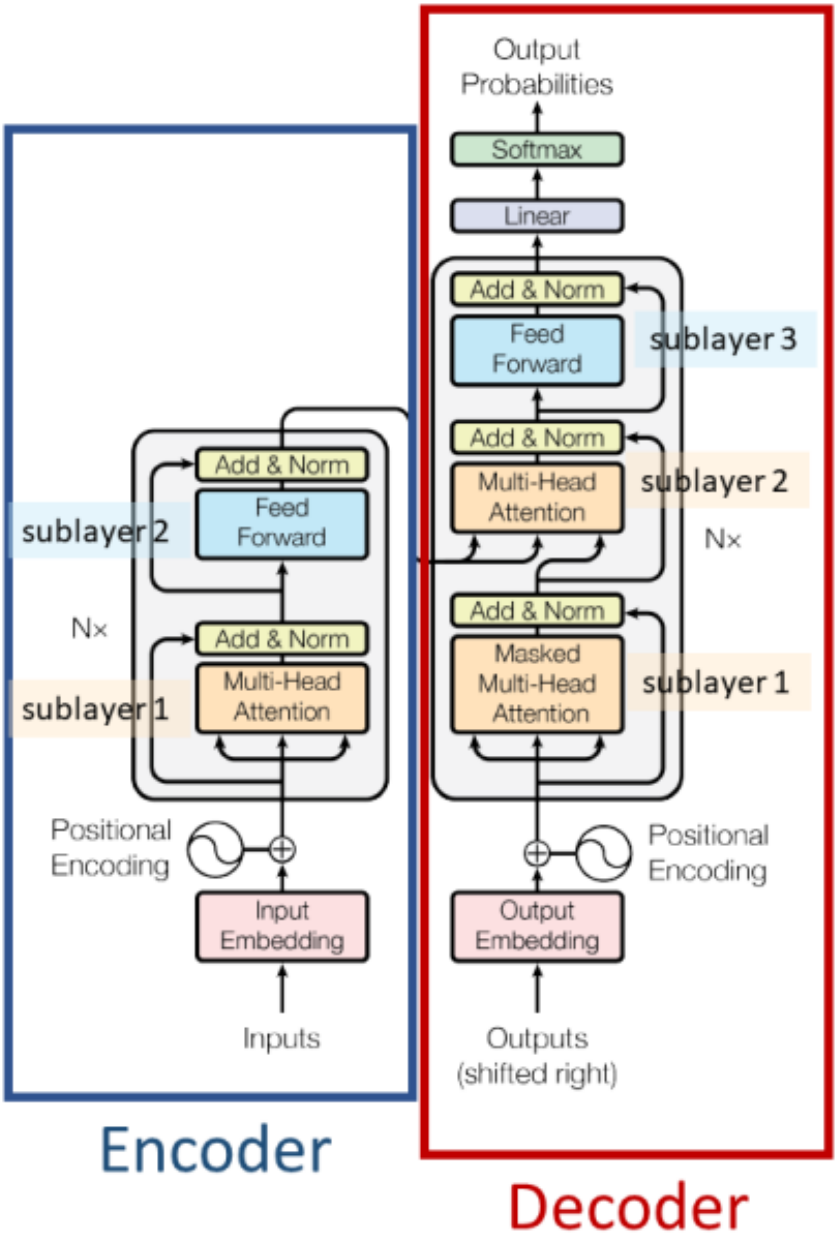

Figure 3.3: A transformer model highlighted with encoder and decoder section. Each section has sub-layers, comprising multi-head attention, add&norm along with Feed Forward layers respectively.

CodeGpt

CodeGpt is based on pre-trained GPT2 model. The model was fine-tuned on a downstream task over java and python corpus, by teams of Microsoft. For training, it took them 25hrs

on P100x2 Nvidia cards for python corpus and 2 hrs on P100x2 cards for Java corpus, respectively. GPT2's architecture is the same as the decoder only transformer. It comprises of only decoder blocks stack on top of each other as seen in Figure 3.4. Each decoder has masked-self attentions and feed-forward neural network. Masked-self attention prevents or blocks a position to speak to its tokens in the right. We have used GPT2-small, which has only 12 layers with model dimensionality of 768. CodeGpt also supports code generation, which is used in the text to code generation tasks. Hyper-parameters used while training the CodeGPT models are listed in Table 3.1. The parameters were unaltered while fine-tuning.

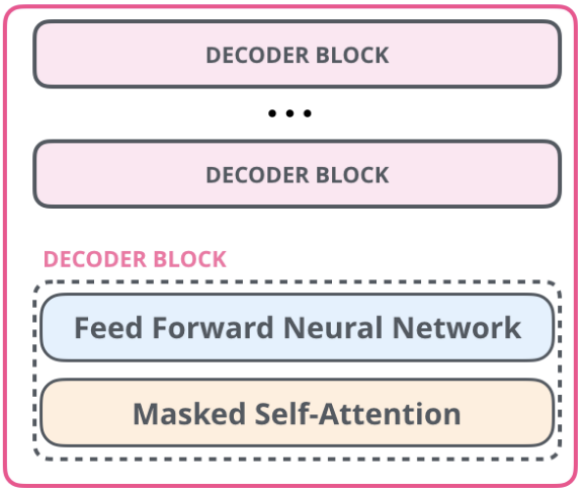

Figure 3.4: GPT2 model architecture, where decoders are stacked on top of each other. Each Decoder has Feed Forward Neural Network layer along with Masked Self-Attention layer, both together forms a decoder block in GPT2.

Roberta

Roberta is variant of BERT model. Build around BERT's masking strategy, it provides a robosty and optimized way to pre-train various NLP tasks. It has around 84Million parameters. BERT or Bidirectional Encoder Representations from Transformers is a decoder only transformer section which was trained for a wide range of NLP tasks, like language modelling, Figure 3.5. Roberta which is an improvement over BERT performs 2-20% better compared to it, parent model. Next Sentence Prediction (NSP) was removed from BERT to form Roberta, and dynamic masking method was introduced. Table 3.2 lists all the hyper-parameters for Roberta model used during training. ByteLevelBPETokenizer is used for creating a tokenized for Roberta over the CSharp data-set, to create a token stream. Trained with a batch size of only 2, because creating a larger batch size was resulting in CUDA out of memory error.

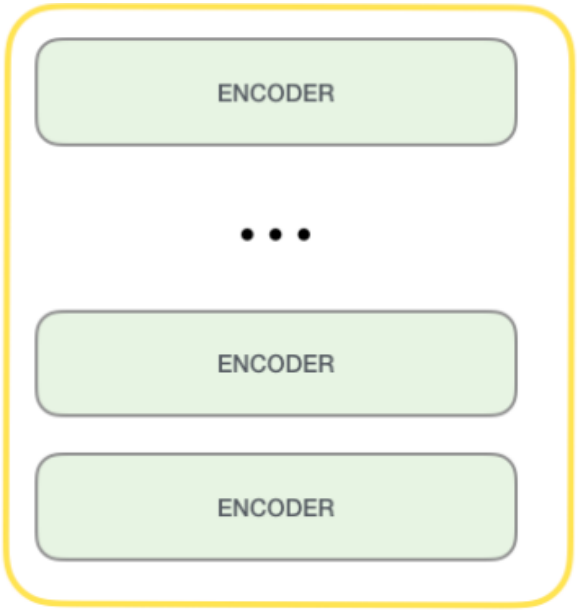

Figure 3.5: BERT model architecture, where encoders are stacked on top of each other.

GPT2

Table 3.3 summarises the hyper-parameters while training and testing the GPT2 model. Same hyper-parameters were used for the third approach in both the strategies. Embedding size of 512 was used with 4 heads and 4 internal layers to reduce the model, that can be trained in colab environment. A tensor flow version of GPT2 was used. Figure 3.6, show the model architecture.

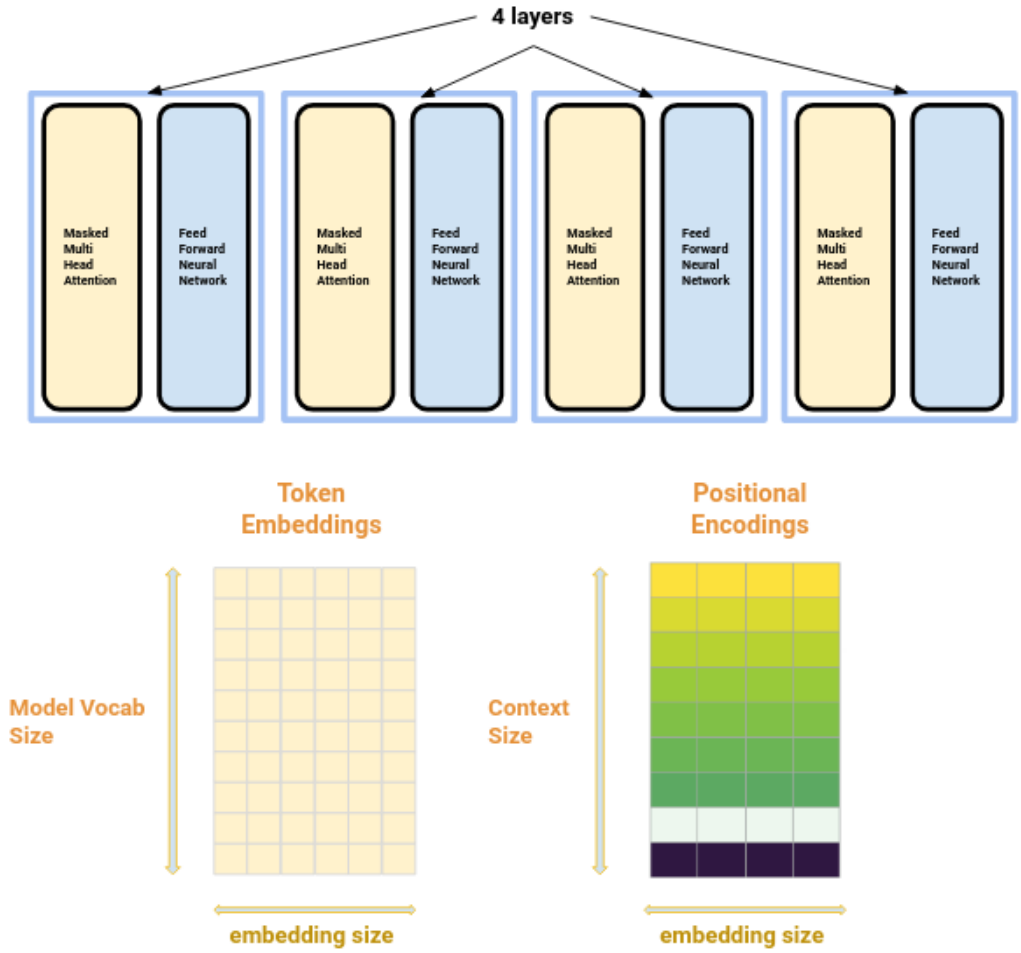

Figure 3.6: The model has 4 layers, and only one token embedding matrix and one Positional Encoding matrix.

Table 3.1: Hyper Param for CodeGpt

CodeGpt Hyper Param	
Activation function	gelu_new
architecture	GPT2LMHeadModel
attn_pdrop	0.1
bos_token_id	50256
embd_pdrop	0.1
eos_token_id	50256
gradient checkpointing	false
initializer_range	0.02
layer_norm_epsilon	1e-05
model_type	gpt2
n_ctx	1024
n_embd	768
n_head	12
n_inner	null
n_layer	12
n_positions	1024
resid_pdrop	0.1
vocab_size	50260

Table 3.2: Hyper Param for Roberta

Roberta Hyper Param	
architecture	RobertaForMaskLM
attn_pdrops_dropot_prob	0.1
bos_token_id	0
eos_token_id	2
gradient checkpointing	false
hidden_act	gelu
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermdeiate_size	3072
layer_norm_epsilon	1e-12
max_position_embeddings	1024
model_type	roberta
num_attention_heads	12
num_hidden_layers	6
pad_token_id	1
type_token_size	1
vocab_size	52000

Table 3.3: Hyper Param for GPT2

GPT2 Param	
Activation function	gelu_new
architecture	GPT2
num_layers	4
num_heads	4
diff	3072
max_seq_len	512
learning_rate	5e-05
optimizer_t	adam
mirrored_strategy	None
grad_clip	False
clip_value	1.0
embedding_size	512
ctx_size	512
vocab_size	50000

Chapter 4

Results

4.1 Auto-source code completion using CodeGpt

The CodeGpt model was trained with a context size of 1024, embedding dimension of 768, positional encoding size of 1024. Attention drop of 0.1. Gelu_new [28] was used as the activation function. The model was trained for 5 epochs for each chunk of data-set. The result for testing and training for the CodeGpt on various chucks of python corpus is summarised in Table 4.1. Figure 4.1 summarises the accuracy result for all chunks of python corpus. As it is found that as the training sample increases, the accuracy on the testing sample is quite the same and increases ever so slightly. This behaviour might be due to the fact that the model was trained for only 5 epochs because of limited computing resources.

Table 4.1: Training and Evaluation result for CodeGpt

Training Size	Global Steps	AvgLoss (Training)	Acc (Testing)
10k	7061	0.68457	0.70933
12.5k	9160	1.04693	0.71506
15k	10611	0.29235	0.71050
17.5k	12290	1.03262	0.71956
20k	14211	0.12262	0.71151

The Roberta model for all data-set for CSharp was trained using the weight decay rate of 0.01, attention drops out the probability of 0.1, maximum positioning embedding of 1024, 12 number of attention heads and 6 hidden layers. And Gelu [29] was used as the hidden layer activation function. RobertaForMaskLM was the architecture used for the model which masks 15% of random tokens and try to predict the tokens.

4.2 Auto-source code completion using Roberta

For SET 1 and SET 3, the model was trained for 30 epochs as they have 23k training samples. But for SET 2 and SET 4, which are unpacked versions of SET 1 and SET 3, respectively, the total sampled sized increased to over 1.8M samples, on which training the model would take a large amount of time. So, a sample size of 1364k samples for training and 364k for testing

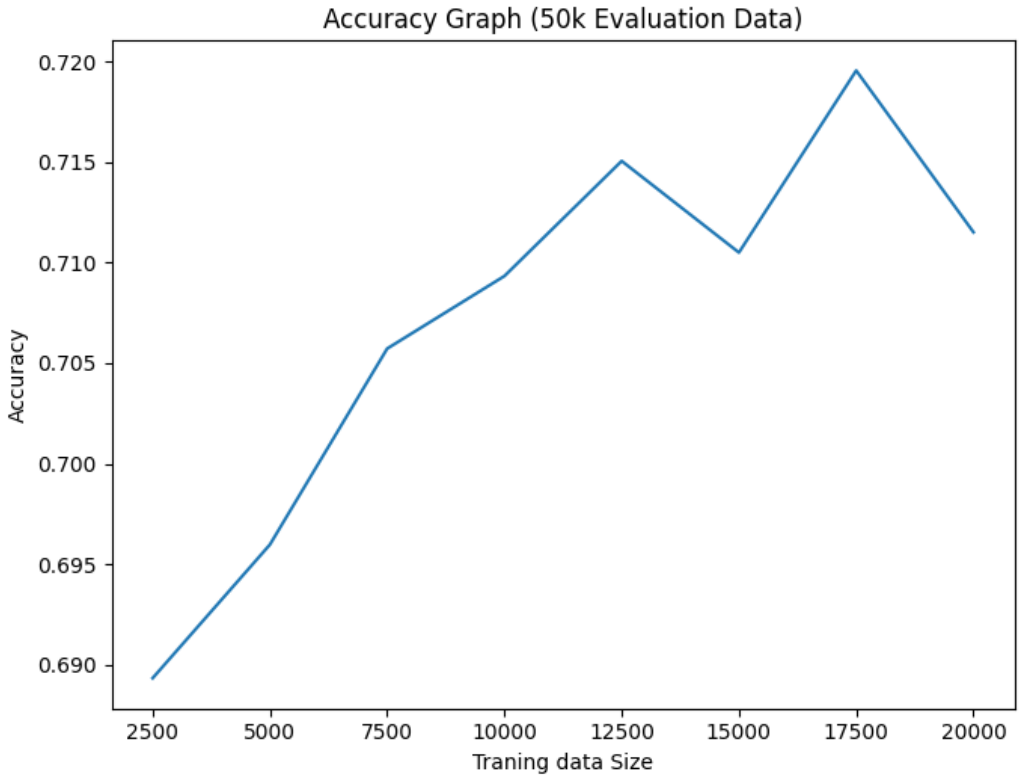

Figure 4.1: Accuracy Graph on 50k test samples

was used and only trained for 10 epochs. Figure 4.2, 4.3, 4.4 and 4.5 show the training and validation loss respectively for the model.

Batch size for training and evaluations for each CSharp corpus is summarized in Table 4.2.

Set	**Train batch size**	**Eval batch Size**	**No. Epochs**
1	128	128	30
2	256	256	10
3	128	128	30
4	256	256	10

Table 4.2: Batch Size for training and evaluating, and number of epochs for each set

Table 4.2 summarizes all the results for testing various CSharp data-set on Roberta model. As seen in Figure 4.6, the training and the evaluation results reduce in a promising way, as for SET 3 and SET 4, which have a token representation for the source code, because the model has less varying elements in the data corpus to learn the context from. The evaluation loss for both SET 3 and SET 4 data-set are 0.3757 and 0.3302, respectively, which is a good sign in proving that statement wise token streams perform better while learning the

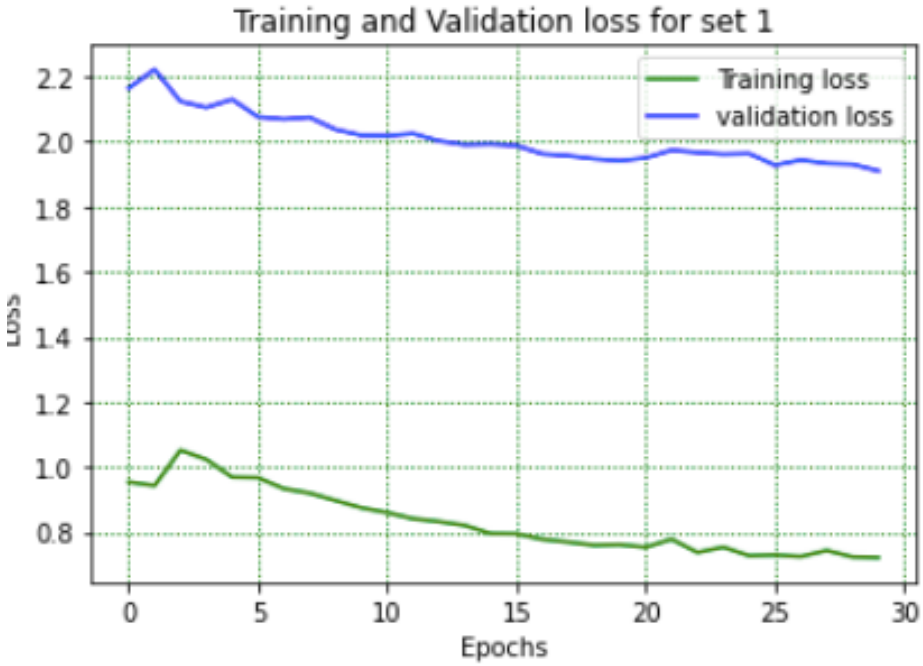

Figure 4.2: Training and Validation loss plot over 30 epochs, on Set 1 CSharp data-set

Set	**Global Steps**	**AvgLoss Training**	**Evaluation Loss**
Set 1	5580	0.8306	2.0165
Set 2	53300	1.1998	2.9193
Set 3	5400	0.5221	0.3757
Set 4	53300	0.1931	0.3302

Table 4.3: Training and Evaluation result for Roberta

code context compared to the extensive program token stream.

From the Table 4.3, which includes all the average training and evaluation loss for Roberta model, it can be also observed that, when using SET 2 instead of SET 1, but the training and validation loss increases. The reason for such behaviour may be because of an increase of naming style in statement representations. And number epoch, being only 10, may have led to such high training and evaluation loss.

4.3 Auto-source code completion using GPT2 on two CSharp domains

From the results of the third methodology, Figure 4.7, it is found that for the naive approach though the training loss is decreasing with time, the validation loss remains unchanged. This signifying over-fitting of the model which was expected since the varying code style across the corpus. Figure 4.8 also shows the same behaviour for the PPL results. Though using

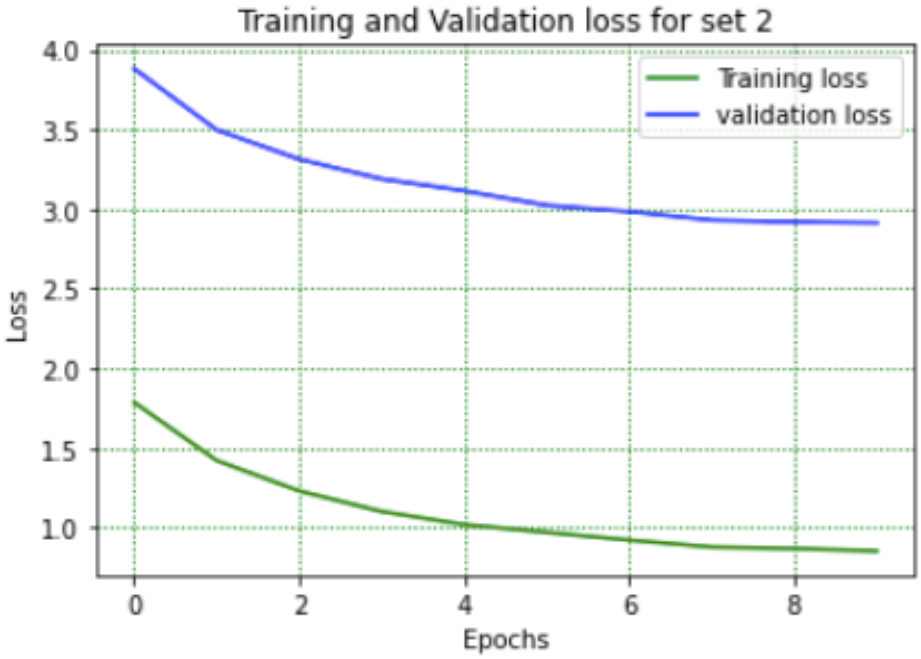

Figure 4.3: Training and Validation loss plot over 10 epochs, on Set 2 CSharp data-set

GPT2 produced better results than using Roberta Model, for the SET 1, but the overall performance is still inferior.

And for the domain-specific approach, where the coded corpus was created using a single project domain, here UNITY3D scripts. This was to test the performance of the GPT2 model on domain-specific corpus for CSharp. And as expected, from Figure 4.9, we can observe that the model performs exceptionally, on a single domain, as produced average evaluation loss of 0.53010 and average evaluation PPL of 4.08256. Where as naive approach produced an average evaluation loss of 1.98349 and 9.37218 average evaluation ppl. Same stats can be observed for the PPL from Figure 4.10. Table 4.4 lists the average values for all the metrics for both approaches.

Set	**Avg Train Loss**	**Avg Eval Loss**	**Avg Train PPL**	**Avg Eval PPL**
Naive Approach	0.86931	1.98349	2.66972	9.37218
Domain Specific	0.50836	0.53010	2.14065	4.08256

Table 4.4: Training and Evaluation result for Gpt2 model for both the approaches.

The CodeGPT2 model produced the accuracy result of Python data-set of 0.7123 approx, for only running the model for 5 epochs, using the training parameters. This is substantially better than the CodeXGlue [1] results, which was found as accurate as 0.7422 approx, trained for 50 epochs. This maybe due to the small domain size, as less variance in coding style could be interpolated in the corpus.

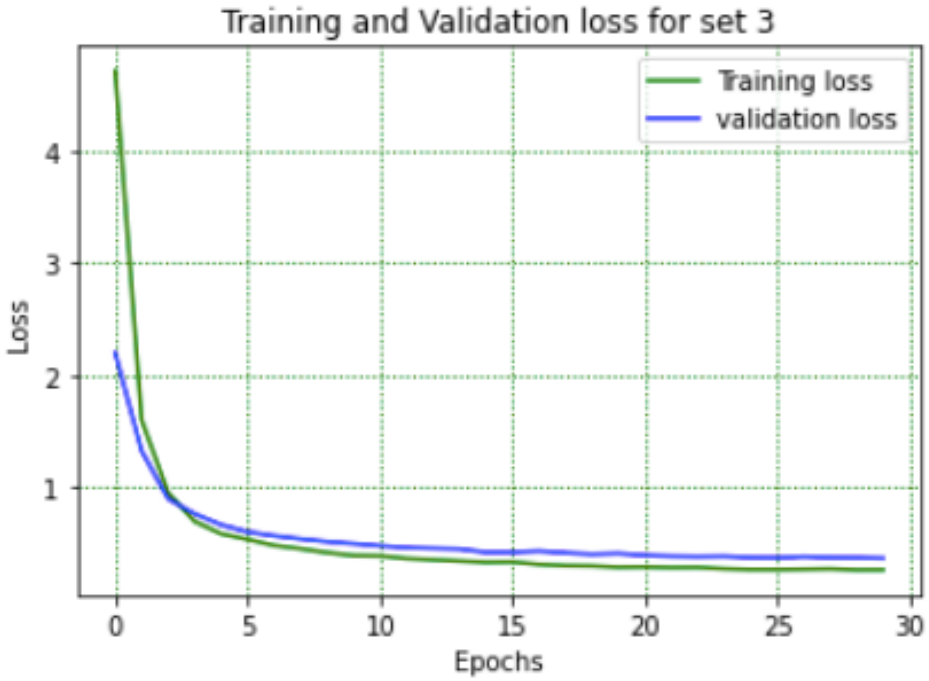

Figure 4.4: Training and Validation loss plot over 30 epochs, on Set 3 CSharp data-set

As for Roberta Model, it didn't perform well, as can be seen from large training and evaluation loss of 0.8306 and 2.0165 for SET 1 and 1.1998 and 2.2193 for SET 2. The reason was the various changing code style in the CShrap corpus. In the SET 3 and 4, the model performed better, as the codes snippets were changed to syntax tokens. But additional mapping strategy is needed to map the tokens to actual words.

The third approach using the GPT2 model performed better than the Roberta model for SET 1, as GPT2 produced average 1.982349 and Roberta produced 2.0165 training loss. Though the model still couldn't overcome the varying code style problem. The domain-specific data-set proved to improve the results, by having average evaluation loss of 0.53010 and avg evaluation ppl of 4.08256 as it didn't has varying code snippets and styles,compared to naive approach which scored around 1.98349 average evaluation loss and average evaluation ppl of 9.37218 . Though the model was trained in small GPT2 size, it could perform much better if trained on a larger size GPT2 model.

4.4 Comparison

Fine-tuning the CodeGpt model on the Py150k dataset, we achieved an accuracy score of 0.7123, when trained for only 5 epoch on resource constraint environment, which is comparable to the CodeXGlue[1] result, 0.7422 accuracy score which was trained for 50 epochs on high-end GPUs. Our GPT2 model performed well compared to Roberta, as evaluation loss was 2.0165 for Roberta and 1.98349 for GPT2 for the SET 1 dataset. Roberta performed well with evaluation loss of 0.3757 and 0.3302 for SET 2 and SET 3 datasets, but

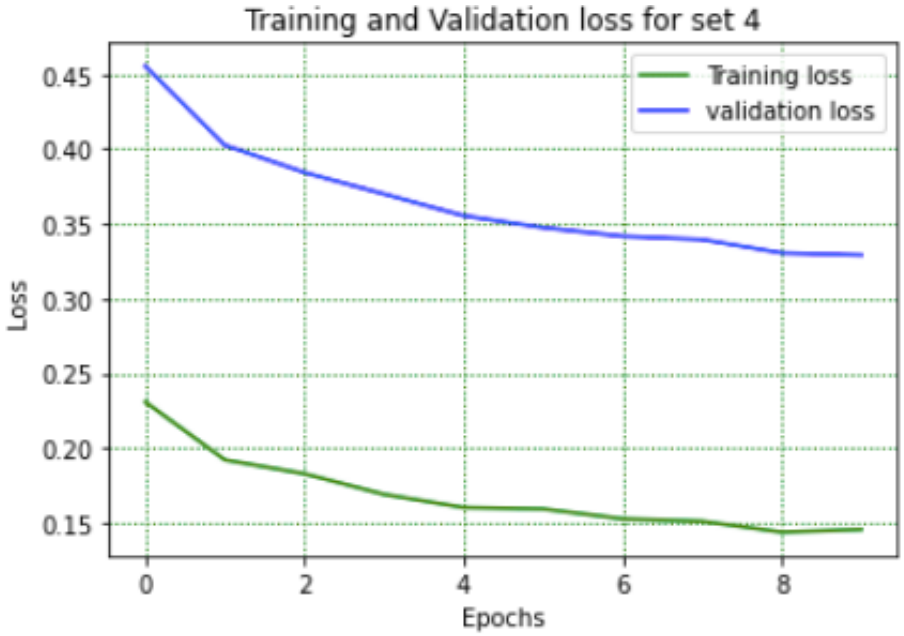

Figure 4.5: Training and Validation loss plot over 10 epochs, on Set 4 CSharp data-set

it requires an additional mapper function. GPT2 model for CShrap domain-specific data-set produced a result of 0.53010 evaluation loss and 4.08256 PPL score, which is better than the nave approach (9.3721 PPL) but is less than the score of MultiGPT-C[21] (PPL of 2.01). One important difference between our approach and MultiGPT-C[21] is that we have trained our model with a small GPT2 of only 4 layers and 4 heads, and trained for just 5 epochs on just 1 Google Colab GPU whereas MUTLIGPT-C[21] was 24-layer and was trained on 80 GPU workers for 25 epochs.

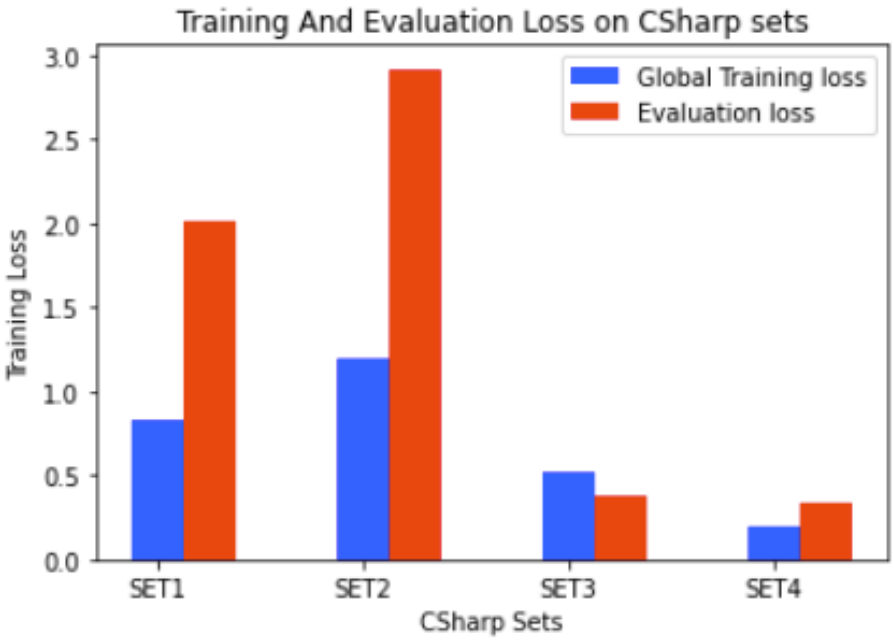

Figure 4.6: Avg Training and Evaluation loss of all the sets.

Figure 4.7: Training and Validation Loss for Naive approach.

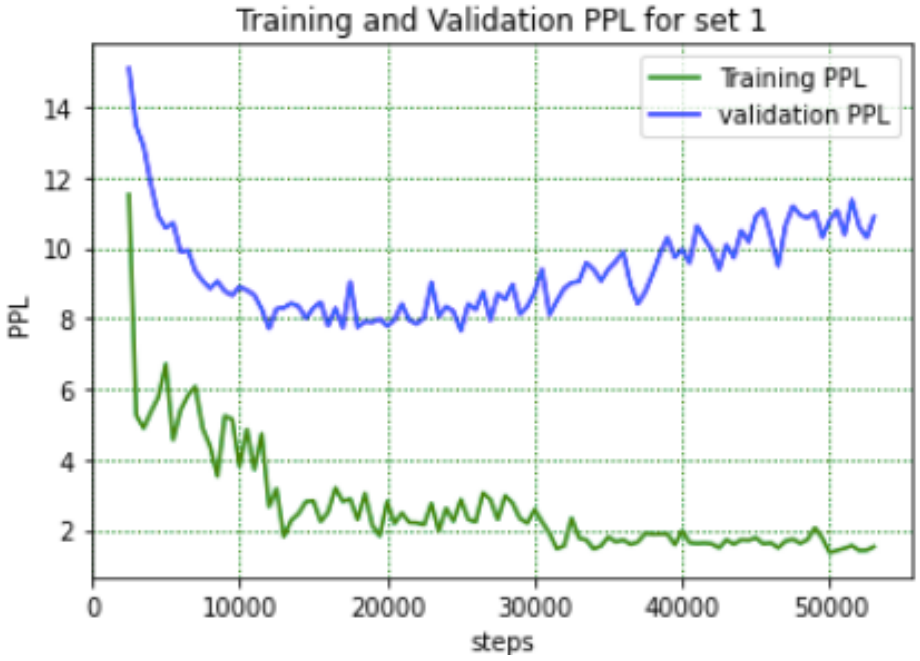

Figure 4.8: Training and Validation PPL for Naive approach.

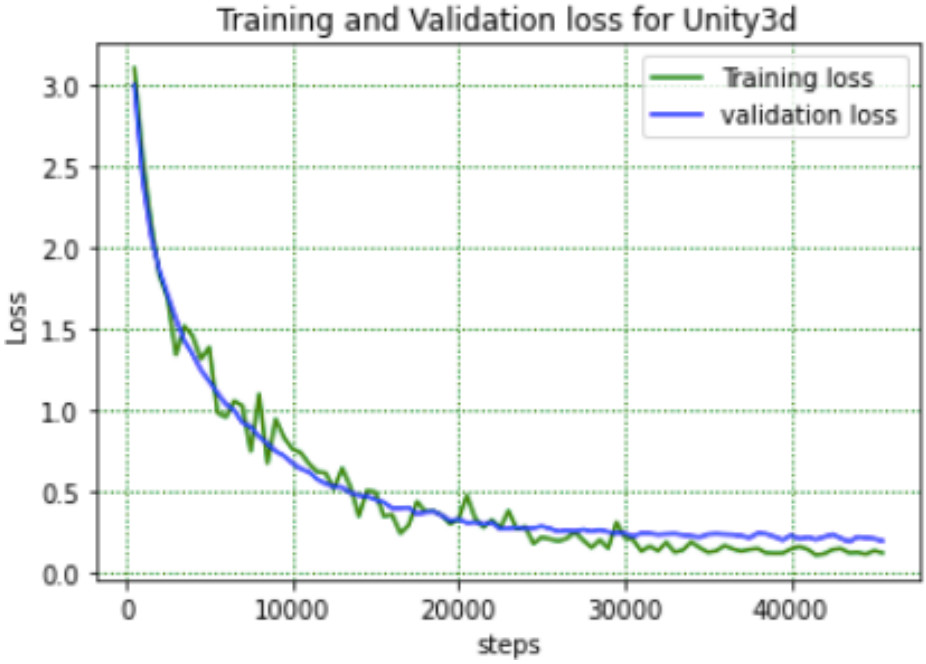

Figure 4.9: Training and Validation Loss for Specific Domain approach.

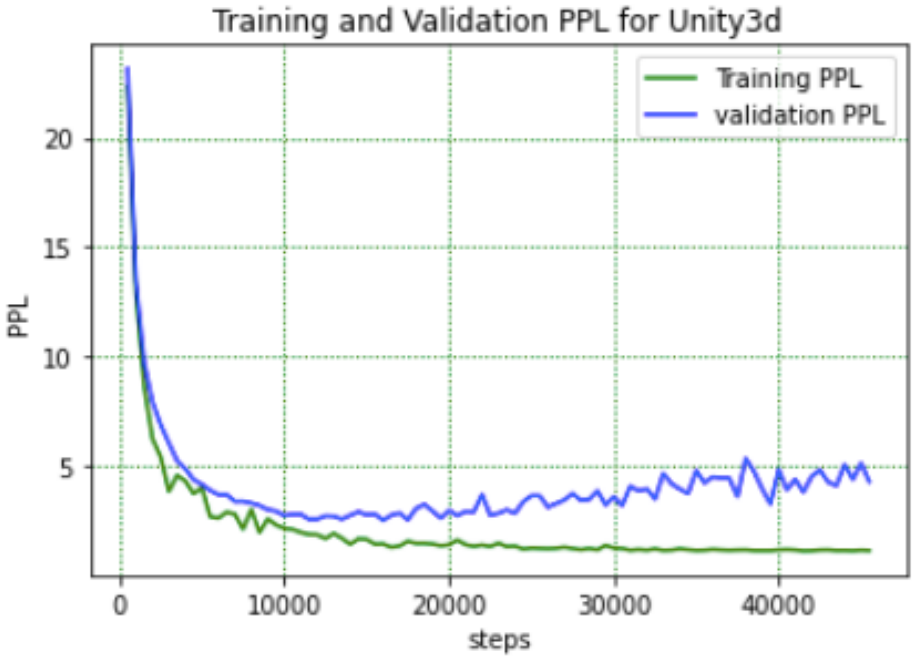

Figure 4.10: Training and Validation PPL for Specific Domain approach.

Chapter 5

Conclusion and Future Scope

In this thesis we demonstrated various experiments with source code auto-completion on Python and CSharp source codes, using various deep learing models. As found from the literature review Abstract syntax trees are widely used to capture the semantic meaning from the code snippets. As using ASTs makes the model generalise to various languages and also makes the model more robust to varying codes. Its is also found that creating a small size model, is useful as they can be integrated in various IDEs. We demonstrated various strategies used to structure source code data-sets to produce signification results in auto-completion tasks.

As for future work, we suggest to use the abstract syntax tree and various structural and semantic model of a source code, which may improve the code prediction accuracy. This may also help in generalizing the auto-completion task for various languages.

References

[1] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021.

[2] 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," *CoRR*, vol. abs/1907.11692, 2019.

[3] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Transformers: State-of-the-Art Natural Language Processing," *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations,* Oct 2020, pp. 38–45

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[5] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 4159–25.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[7] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," *arXiv preprint arXiv:2010.07987*, 2020.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017, pp. 5998–6008.

[9] Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"," *"Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers),June 2019, pp. 4171–4186*

[10] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *OpenAI*, 2018.

[11] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, pp. 5753–5763.

[12] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf, "Transfer learning in natural language processing," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, 2019, pp. 15–18.

[13] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 328–339.

[14] B.-P. V. M. Raychev, Veselin, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, 2016 pp. 731–747.

[15] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[16] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models for any-code generation," *Proceedings of Machine Learning Research*,July 2020, vol. 119, pp. 245–256

[17] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4998–5007.

[18] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[19] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2019.

[20] M. Sabia and C. Wang, *Python Tools for Visual Studio*. Packt Publishing Ltd, 2014.

[21] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[22] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 1977.

[23] A. Svyatkovskiy, et al., "Fast and Memory-Efficient Neural Code Completion," in 2021 2021 IEEE/ACM 18th *International Conference on Mining Software Repositories (MSR) (MSR), undefined, undefined, undefined, 2021 pp. 329-340*.

[24] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725.

[25] S. Kim, J. Zhao, Y. Tian and S. Chandra, "Code Prediction by Feeding Trees to Transformers," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 150-162, doi: 10.1109/ICSE43902.2021.00026*.

[26] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Deep3: Leveraging three levels of parallelism for efficient deep learning," in *Proceedings of the 54th Annual Design Automation Conference*, 2017, pp. 1–6.

[27] D. Spinellis, "Tokenize source code into integer vectors, symbols, or discrete tokens," 2018. [Online]. Available: https://github.com/dspinellis/tokenizer

[28] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with gaussian error linear units," *CoRR*, vol. abs/1606.08415, 2016.

[29] A. Mutton, M. Dras, S. Wan, and R. Dale, "GLEU: Automatic evaluation of sentence-level fluency," in *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 344–351.