Complex & Intelligent Systems (2022) 8:4357–4368
https://doi.org/10.1007/s40747-022-00708-7

ORIGINAL ARTICLE

Source code auto-completion using various deep learning models under limited computing resources

Madhab Sharma[1] · Tapas Kumar Mishra[1] · Arun Kumar[1]

Received: 15 September 2021 / Accepted: 27 February 2022 / Published online: 6 April 2022

Abstract

Deep learning models have become state of the art in many language modelling tasks. Among such tasks, source code auto-completion is one of the important areas of research. This paper presents various methodologies for source code auto-completion using different Deep Learning models for Python and CSharp Programming Languages. In a resource-limited environment, it is paramount to reduce various overheads: one way of achieving that is to use the code sequences to train and evaluate rather than using other code structures such as semantics. This paper compares various deep learning architectures like CodeGPT [1] from Microsoft, Roberta [2] from huggingface [3] and GPT2 [4] for source code auto-completion. Different dataset strategies are employed for comparison, 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 them into the models. We consider the task of autocompletion on two different datasets: (1) Python dataset; (2) CSharp dataset. The Python dataset is processed by a fine-tuned CodeGPT with an overall accuracy of 71%. For the CSharp dataset with the GPT2 model, a PPL of 2.14 and 4.082 on the training and evaluation dataset is observed. Considering the results, we discuss the strength and weaknesses of different approaches in their application in the real world programming context.

Keywords Source-code auto completion · CodeGPT · Roberta · GPT2

Introduction

In the software development process, code auto-completion is a requisite tool for any developer as it increases productivity and can save countless hours while writing code [30]. 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 analyzing code sequences, (ii) using neural networks, (iii) NLP or deep learning models to perform the auto-completion. The goal is to provide a rapid and correct suggestion for the following code sequence. These autocomplete models can be integrated into various IDEs and online code editors to increase the efficiency and productivity of developers.

Various methods have already been proposed for code auto-completion, e.g. using RNN(LSTM) [5], using LSTM with attention [6], using transformers [7], etc. Various code structures have been used as input data to these models—some of them focus on the raw source text whereas some are based on token streams. Recently, abstract syntax trees are gaining attraction to capture context and predict the next code tokens or streams. Early IDEs also 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 has 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 imple-

✉ Tapas Kumar Mishra
mishrat@nitrkl.ac.in

Madhab Sharma
219CS1143@nitrkl.ac.in

Arun Kumar
kumararun@nitrkl.ac.in

[1] Department of Computer Science and Engineering, National Institute of Technology Rourkela, Rourkela, Odisha 769008, India

mentation. 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 computing resources. The cost of running such tests on model training experiments used to be enormous: this has been eliminated after the introduction of transfer learning.

In this paper, we discuss the following three approaches for source-code autocompletion:

1. Fine tuning CodeGpt on Python dataset.
2. Code prediction using Roberta on CSharp source code.
3. Code prediction using GPT2 model trained on CSharp source code.

The rest of the paper is organized as follows: In Section 2, we discuss the existing techniques, tools and literature for various source code auto-completion tasks. In Section 3, the dataset, various preprocessing steps and the proposed models are discussed. In Section 4, we list out the results obtained for different models with different datasets and compare the results with existing results.

Related work

Various methods have been proposed for source code auto-completion tasks and we discuss few of them in the section.

In *code2vec: Learning Distributed Representations of Code* [6] paper, the authors has presented a neural network framework to learn code embedding. They have used the semantic structure of code, by feeding serialized ASTs to represent into the network. The network itself contains LSTM layers with paths-attentions. A dataset of almost 10k Java GitHub repositories was used during training and evaluation purpose. The model achieved precision, recall and F1 score of 63.1, 54.4, and 58.4 respectively when evaluated on the test set consisting of 50k files. 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.

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. This model relies on ASTs for code snippets. 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 dataset, respectively. The limitation of this model is that it does not consider long-distance context into account; rather it is limited to surrounding contexts.

In [16], 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. The model shows 18.04 acc@1 and 24.83 acc@5 on the java corpus. For the CSharp corpus, it shows 37.61 acc@1 and 45.51 acc@2. acc@n means average accuracy on top n probabilities for a prediction. Limitation of the model is that it works with only one function or class. And the model performs poorly for complex expressions.

In [17], the authors use a transformer [8] model with self-attention mechanism to capture long-range dependencies during code summarization. Source-code token stream is fed to the network as input samples. The proposed model has BLEU [18] score of 44.58 on Java dataset and BLEU score of 32.52 on Python dataset. Limitation of this model is that it does not consider any structural aspect of the code. Moreover, the model was only tested on Java and Python dataset.

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. 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.

In [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 dataset. 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. With the model size of 366 M, GPT-C scored a PPL score of 1.91 on CSharp corpus 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 374 M model size. However, it does not 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.

The authors in [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 four 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.

In [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 dataset was used for the model(it consists of parsed ASTs trees). For the transformer model, GPT-2 small is used, adapted from pytorch version. The reciprocal rank improvement of TRAVTRANS compared to DEEP3 [26] is from 43.9% to 58.0%. And with CODE2SEQ compared with TRAVTRANS is 43.6% to 58.0%. Though TRAVTRANS outperforms other representation, the structural relation between the nodes is not retained, as cited by examples [25] in the paper by the authors.

Methodology

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

1. Fine-tuning CodeGpt on Python data set.
2. Training Roberta model in four different CSharp datasets.
   (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.

Dataset

For using NLP models on the source code, numerous training–testing-validation samples are required. For the Python corpus, PY150 dataset, 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 dataset 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 chunk for evaluation. Breaking into such chunks enabled to run the model efficiently on the limited available resources.

For the C# dataset 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.

For the GPT2 model, in the third approach, SET 1 used in the Roberta model was used in the naive approach. For the Domain-Specific approach, a new dataset 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 datasets have roughly the same size, around 30k 36K samples. The training hyper-parameters were same for both the datasets. For both the cases, the model was trained for five epochs. For the naive approach, the dataset consists of various top GitHub repositories on CSharp source code all belonging to different domains. In case of Domain-Specific study, source codes related to UNITY3D were used. The corpus in all the cases are built from freely available CSharp code repositories from GitHub.

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 (see Listing 1).

For CSharp corpus, the dataset was pre-processed with four different sets as follows: (1) the first set contains all the source code in a single line; (2) in the second dataset, source code is split line by line such that each line contains a single statement; (3) the third dataset contains the tokenized streams of the first set; (4) the fourth set contains tokenized streams 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 done because Roberta can only handle max token length of 514: tokens larger than that get truncated leading to loss of context from each input sample. Figures 1 and 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 length of input stream to encode; thus saving VRAM memory.

Another reason for splitting of the source code line by line was that as the codes were extracted from popular GitHub repositories, they contained a large amount of import statements (like *Using...*) before the actual code blocks. In addition to this, with the truncation after the first 514 tokens as input, less relevant information was fed to the model if splitting is not performed. Hence, to leverage this issue, the whole code files were broken line by line and each line was now a input to the model. As the splitting of codes into lines was not perfect (since some class definition or function were very large), lines which had five code tokens or less were discarded during the prepossessing step.

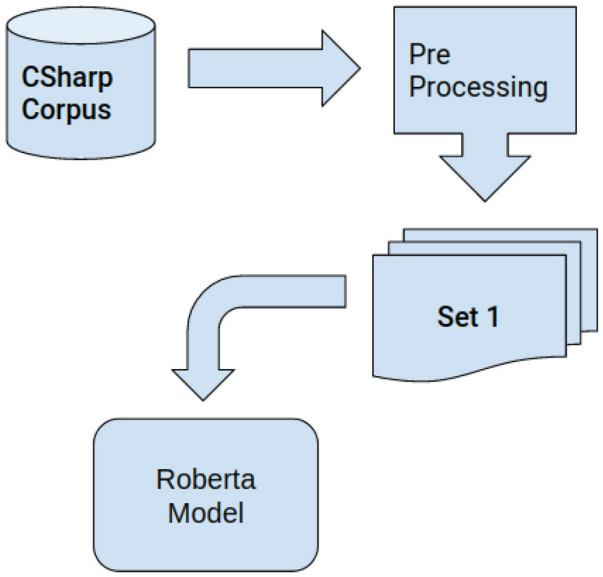

Fig. 1 Pipeline for CSharp codes with Roberta

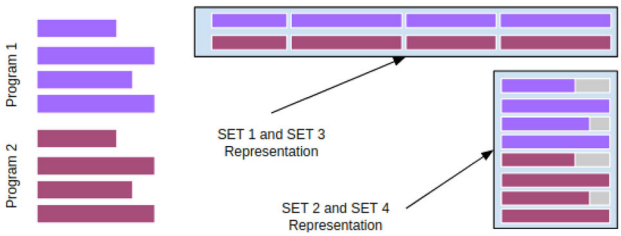

Fig. 2 Pipeline for CSharp codes with Roberta

Yet another issued prevailed: as the variable naming style varied from the source file to file, the model did not perform well as expected on the identifier. To tackle this problem, the tokenized version of the source code was used. Here the source codes of the entire file were tokenized using a tokenized tool [27]. Subsequently, they are spilt into statement wise tokens and fed to the model. The same procedure of discarding lines with tokens less than or equal to 5 was used. Listing 2, 3, 4 and 5 show the SET 1, SET 2, SET 3 and SET 4 representations, respectively.

Listing 1 CSharp Set 1 Sample

```
collection . Add ( name ) ; Assert . Empty ( removeEvent . Event . EventName ) ;
name = "Int32" ; yield return new object [ ] { " Int32 " , " Name " , " Int32 " } ;
```

Listing 2 CSharp Set 2 Sample

```
collection . Add ( name ) ;
Assert . Empty ( removeEvent . Event . EventName ) ;
name = "Int32" ;
yield return new object [ ] { " Int32 " , " Name " } ;
```

Listing 3 CSharp Set 3 Sample

```
ID . ID = ID ; ID . ID ( ID . ID . ID ) ;
ID = STRING_LITERAL ; yield return new object [ ] { STRING_LITERAL , STRING_LITERAL } ;
```

Listing 4 CSharp Set 4 Sample

```
ID . ID = ID ;
ID . ID ( ID . ID . ID ) ;
ID = STRING_LITERAL ;
yield return new object [ ] { STRING_LITERAL , STRING_LITERAL} ;
```

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 dataset of SET 1 was used for training and evaluating. For the domain-specific approach, pre-processing used for SET 1 in Roberta model was used.

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. GPT2 was used for the third methodology. 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 Fig. 3. We are actually using GPT-2 small and Roberta in the implementation. Roberta uses a sequence of 12 encoders: they are not parallel; output of one encoder is passed on to the next. GPT-2 small uses a sequence of 12 decoders.

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 25 h on P100x2 Nvidia cards for Python corpus and 2 h 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 Fig. 4. Each decoder block consists of a masked self-attention layer combined with a feed-forward 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 1. The parameters were unaltered while fine-tuning.

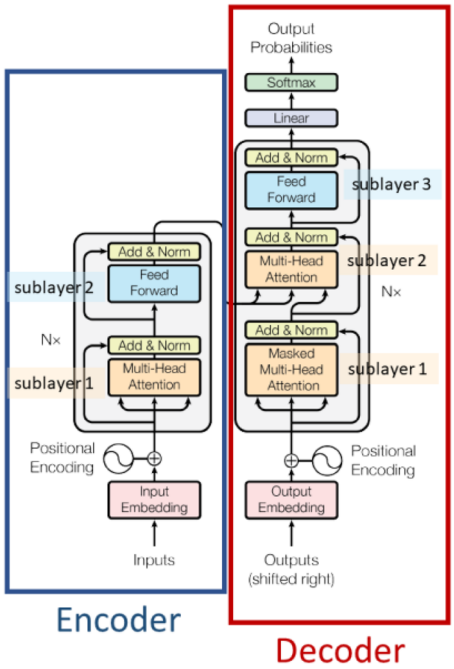

Fig. 3 A transformer model highlighted with encoder and decoder section. Each section has sub-layers, comprising multi-head attention, add & norm along with Feed Forward layers respectively

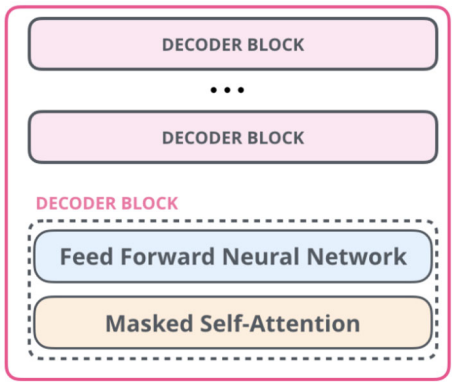

Fig. 4 GPT2 model architecture, where decoders are stacked on top of each other. Each Decoder has Feed Forward Neural Network layer along with Masked Self-Attention layer, both together forms a decoder block in GPT2

Roberta

Roberta is variant of BERT model. Build around BERT's masking strategy, it provides a robust and optimized way to pre-train various NLP tasks. It has around 84 million parameters. BERT or Bidirectional Encoder Representations from

Table 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 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

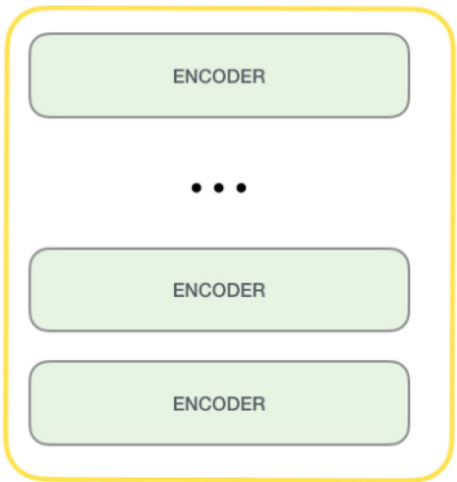

Fig. 5 BERT model architecture, where encoders are stacked on top of each other

Transformers is a decoder only transformer section which was trained for a wide range of NLP tasks like language modelling (see Fig. 5). Roberta performs 2–20% better compared to BERT. Next Sentence Prediction (NSP) was removed from BERT to form Roberta, and dynamic masking method was introduced. Table 2 lists all the hyper-parameters for Roberta model used during training. ByteLevelBPETokenizer is used for creating a tokenized for Roberta over the CSharp dataset, 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.

Table 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

GPT2

Table 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.

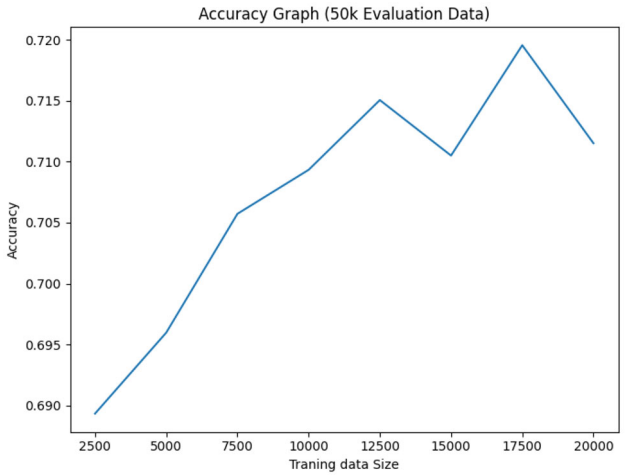

Fig. 6 Accuracy Graph on 50k test samples

The accuracy of the models are computed using the following formula:

$$accuracy = \frac{number\ of\ correctly\ predicted\ tokens}{total\ number\ of\ tokens}.$$

The loss is calculated in the following manner: The loss of a single batch is the cross-entropy loss for that batch. The total loss of a epoch is computed by taking the arithmetic mean of all the batch losses.

Results

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 dataset. The result for testing and training for the CodeGpt on various chucks of Python corpus is summarised in Table 4. Figure 6 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.

The Roberta model for all dataset 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. Roberta-ForMaskLM was the architecture used for the model which masks 15% of random tokens and try to predict the tokens.

For SET 1 and SET 3, the model was trained for 30 epochs as they have 23k training samples. But for SET 2 and

Fig. 7 Training and validation loss plot over 30 epochs, on set 1 CSharp dataset

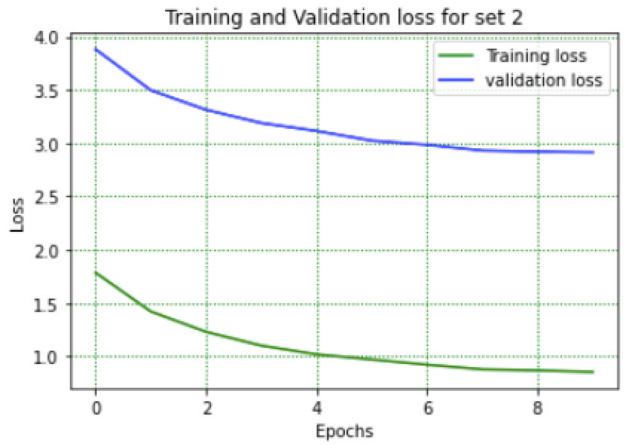

Fig. 8 Training and Validation loss plot over 10 epochs, on set 2 CSharp dataset

SET 4, which are unpacked versions of SET 1 and SET 3, respectively, the total sampled sized increased to over 1.8 M 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 was used and only trained for 10 epochs. Figures 7, 8, 9, 10 show the training and validation loss, respectively, for the model.

Batch size for training and evaluations for each CSharp corpus is summarized in Table 5.

Table 5 summari***zes all the results for testing various CSharp dataset on Roberta model. As seen in Fig. 11, 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 dataset are 0.3757 and 0.3302, respectively, which is a good sign in proving that statement wise token streams perform better while learning the code context compared to the extensive program token stream.

Table 4 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	10,611	0.29235	0.71050
17.5k	12,290	1.03262	0.71956
20k	14,211	0.12262	0.71151

Table 5 Batch Size for training and evaluating, and number of epochs for each set

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 6 Training and Evaluation result for Roberta

Set	Global steps	AvgLoss training	Evaluation Loss
Set 1	5580	0.8306	2.0165
Set 2	53,300	1.1998	2.9193
Set 3	5400	0.5221	0.3757
Set 4	53,300	0.1931	0.3302

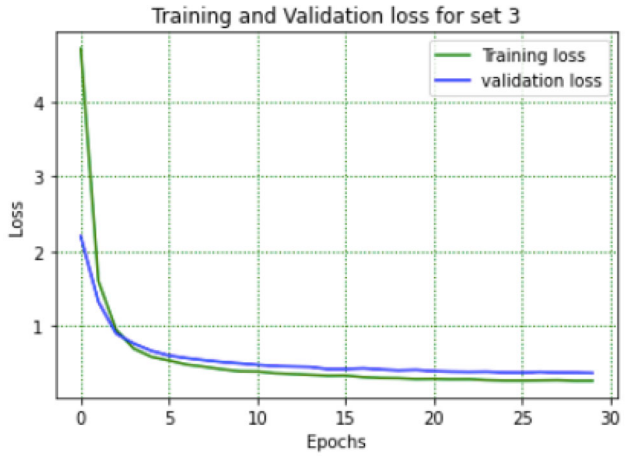

Fig. 9 Training and Validation loss plot over 30 epochs, on set 3 CSharp dataset

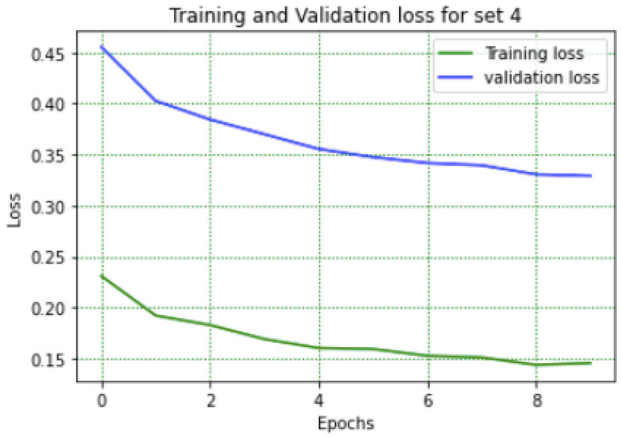

Fig. 10 Training and Validation loss plot over 10 epochs, on set 4 CSharp dataset

From Table 6, 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.

From the results of the third methodology (see Fig. 12), it is found that for the naive approach though the training loss is decreasing with time, the validation loss remains unchanged. This is signifying over-fitting of the model which was expected since the varying code style across the corpus. Figure 13 also shows the same behaviour for the PPL results. Though using GPT2 produced better results than

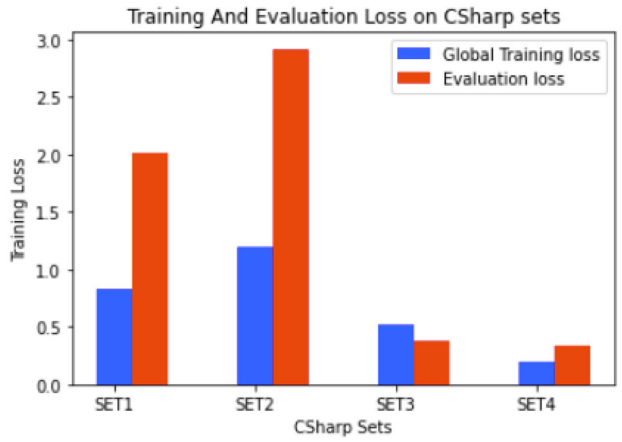

Fig. 11 Avg training and evaluation loss of all the sets

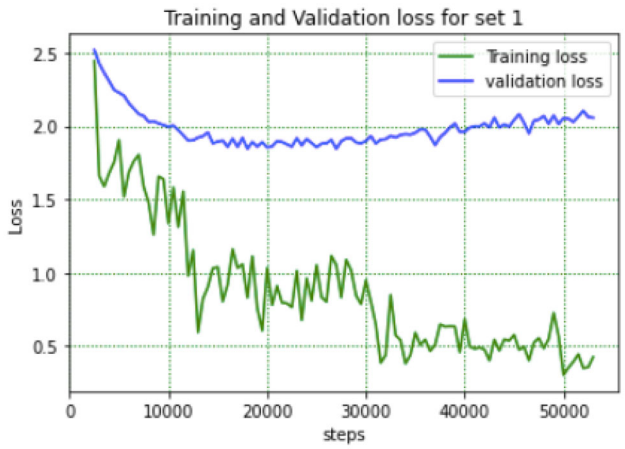

Fig. 12 Training and validation loss for Naive approach

using Roberta Model for the SET 1, the overall performance is still inferior.

For the domain-specific approach, where the coded corpus was created using a single project domain, the UNITY3D scripts. This was to test the performance of the GPT2 model on domain-specific corpus for CSharp. As expected (see Fig. 14), 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. This is in contrast to the naive approach that produced an average evaluation loss of 1.98349 and 9.37218 average evaluation ppl. Similar statistics can be observed for the PPL from Fig. 15. Table 7 lists the average values for all the metrics for both approaches.

The CodeGPT2 model produced the accuracy result of Python dataset 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

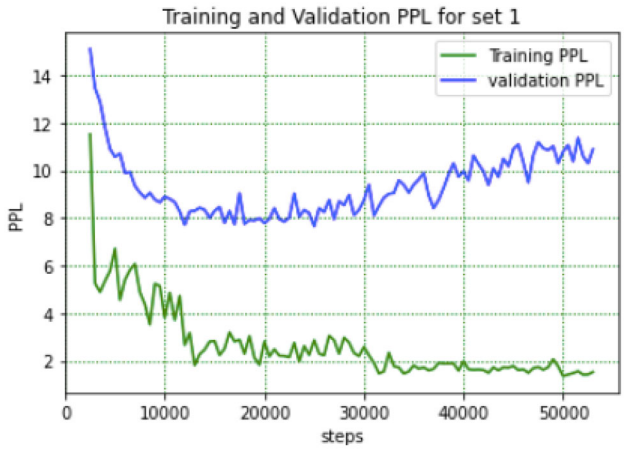

Fig. 13 Training and validation PPL for Naive approach

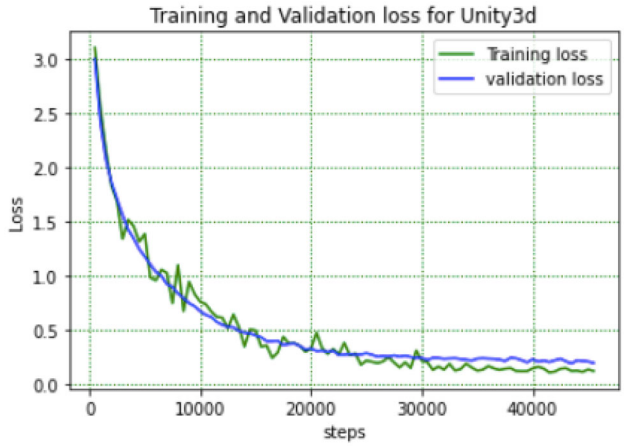

Fig. 14 Training and validation loss for specific domain approach

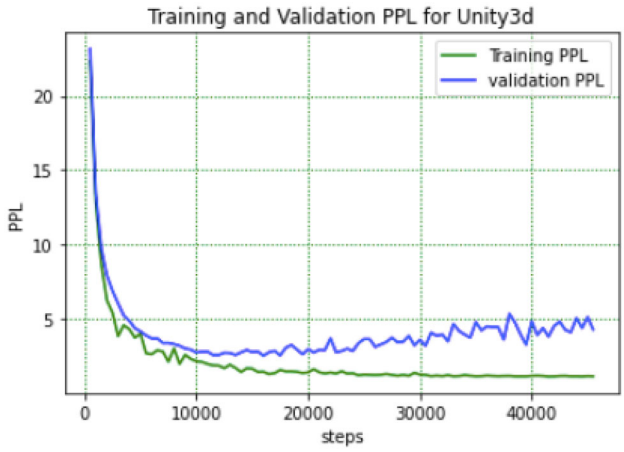

Fig. 15 Training and validation PPL for specific domain approach

Table 7 Training and evaluation result for Gpt2 model for both the 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

maybe due to the small domain size, as less variance in coding style could be interpolated in the corpus.

As for Roberta Model, it did not 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 code 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: GPT2 produced average 1.982349 and Roberta produced 2.0165 training loss. Still, the model still could not overcome the varying code style problem. The domain-specific dataset proved to improve the results: it produced average evaluation loss of 0.53010 and average evaluation ppl of 4.08256 (this is in contrast to the naive approach that produced an average evaluation loss of 1.98349 and 9.37218 average evaluation ppl). The state-of-the-art MultiGPT-C[21] has PPL of 2.01. Note 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.

Time complexity analysis

Each encoder and decoder layer consists of a self-attention layer and a feed forward layer.

Let X be the input to a self-attention layer. Then, X will have shape (n, d) since there are n word-vectors (corresponding to rows) each of dimension d. Computing the output of self-attention requires the following steps (consider single-headed self-attention for simplicity):

1. Linearly transforming the rows of X to compute the query Q, key K, and value V matrices, each of which has shape (n, d). This is accomplished by post-multiplying X with 3 learned matrices of shape (d, d), amounting to a computational complexity of $O(nd^2)$. 2. Computing the layer output as $SoftMax(QK^t/sqrt(d))V$, where the softmax is computed over each row. Computing QK^t has complexity $O(n^2d)$, and post-multiplying the resultant with V has complexity $O(n^2d)$ as well.

Therefore, the total complexity of the self attention layer is $O(n^2d + nd^2)$.

Each block in the encoder and the decoder contains an independent fully connected 2-layer feed-forward network with a ReLU nonlinearity applied separately to each position of the sequence:

$$FFN(Z) = max(0, Z\theta_1 + b1)\theta_2 + b2$$

where Z are the representations passed forward from the attention sublayer, θ_1, θ_2 are two learned independent parameter matrices for each layer and $b1, b2$ are their respective bias vectors.

Therefore, the total complexity of the feed-forward network layer is $O(nd^2)$.

Thus, the total complexity of a single encoder/decoder block is $O(n^2d + nd^2)$. Since there is constant number of such blocks in each transformer, the the total complexity of a transformer model is $O(n^2d + nd^2)$.

Conclusion

This paper demonstrates various experiments with source code auto-completion on Python and CSharp source codes using different deep learning models. We presented multiple strategies used to structure source code datasets to produce significant results in auto-completion tasks. Fine-tuning the CodeGpt model on the Py150k dataset, we achieved an accuracy score of 0.7123 when trained for only five epochs on resource constraint environment. This is comparable to the CodeXGlue [1] result of 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 it requires an additional mapper function. GPT2 model for CShrap domain-specific dataset produced a result of 0.53010 evaluation loss and 4.08256 PPL score, which is better than the naive 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 four layers and four heads and trained for just five epochs on just 1 Google Colab GPU. The MUTLIGPT-C[21], on the other hand, was 24-layer and was trained on 80 GPU workers for 25 epochs. As for future work, we suggest using the abstract syntax tree

and various structural and semantic models of a source code, which may improve the code prediction accuracy. This may also help in generalizing the auto-completion task for multiple languages.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, Li G, Zhou L, Shou L, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu S, Liu S (2021) Codexglue: a machine learning benchmark dataset for code understanding and generation
2. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) Roberta: a robustly optimized BERT pretraining approach. arXiv:1907.11692
3. Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M, Brew J (2019) Huggingface's transformers: State-of-the-art natural language processing. arXiv:1910.03771
4. Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I (2019) Language models are unsupervised multitask learners. OpenAI blog 1(8):9
5. Li J, Wang Y, Lyu MR, King I (2018) Code completion with neural attention and pointer networks. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, pp. 4159–25
6. Alon U, Zilberstein M, Levy O, Yahav E (2019) Code2vec: Learning distributed representations of code. Proceedings of ACM Program. Lang., vol. 3, no. POPL
7. Chirkova N, Troshin S (2020) Empirical study of transformers for source code. arXiv:2010.07987
8. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I (2017) 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., pp 5998–6008
9. Devlin J, Chang M, Lee K, Toutanova K (2018) BERT: pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805
10. Radford A, Narasimhan K, Salimans T, Sutskever I (2018) Improving language understanding by generative pre-training. OpenAI
11. Yang Z, Dai Z, Yang Y, Carbonell J, Salakhutdinov RR, Le QV (2019) 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., pp 5753–5763
12. Ruder S, Peters ME, Swayamdipta S, Wolf T (2019) Transfer learning in natural language processing. In: Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: tutorials, pp 15–18
13. Howard J, Ruder S (Jul. 2018) 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, pp 328–339
14. B-P, Raychev V, Veselin M (2016) Probabilistic model for code with decision trees. ACM SIGPLAN Notice 51(10):731–747
15. Alon U, Brody S, Levy O, Yahav E (2019) 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
16. Alon U, Sadaka R, Levy O, Yahav E (2019) Structural language models for any-code generation. arXiv:1910.00577
17. Ahmad W, Chakraborty S, Ray B, Chang K-W (Jul. 2020) 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, pp 4998–5007
18. Papineni K, Roukos S, Ward T, Zhu W-J (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp 311–318
19. Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N (2019) Pythia: Ai-assisted code completion system. Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining
20. Sabia M, Wang C (2014) *Python Tools for Visual Studio*. Packt Publishing Ltd
21. Svyatkovskiy A, Deng SK, Fu S, Sundaresan N (2020) 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*, pp 1433–1443
22. Jelinek F, Mercer RL, Bahl LR, Baker JK (1977) Perplexity-a measure of the difficulty of speech recognition tasks. J Acoust Soc Am 62(S1):S63–S63
23. Svyatkovskoy A, Lee S, Hadjitofi A, Riechert M, Franco J, Allamanis M (2020) Fast and memory-efficient neural code completion. arXiv:2004.13651
24. Sennrich R, Haddow B, Birch A (Aug. 2016) 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, pp 1715–1725
25. Kim S, Zhao J, Tian Y, Chandra S (2020) Code prediction by feeding trees to transformers. arXiv:2003.13848
26. Rouhani BD, Mirhoseini A, Koushanfar F (2017) Deep3: leveraging three levels of parallelism for efficient deep learning. In: Proceedings of the 54th annual design automation conference, pp 1–6
27. Spinellis D (2018) Tokenize source code into integer vectors, symbols, or discrete tokens. [Online]. Available: https://github.com/dspinellis/tokenizer
28. Hendrycks D, Gimpel K (2016) Bridging nonlinearities and stochastic regularizers with gaussian error linear units. arXiv:1606.08415

29. Mutton A, Dras M, Wan S, Dale R (2007) 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, pp 344–351

30. Hammad Muhammad, Babur Önder, Abdul Basit Hamid, Brand Mark van den (2020) DeepClone: Modeling Clones to Generate Code Predictions. In: Reuse in Emerging Software Engineering Practices, Ben Sassi, Sihem and Ducasse, Stéphane and Mili, Hafedh, Eds., Springer, New York, pp 135–151