

Natural Language to Python Source Code using Transformers

Meet Shah
Information Technology
Sardar Patel Institute of Technology
Mumbai, India
meet8june@gmail.com

Rajat Shenoy
Information Technology
Sardar Patel Institute of Technology
Mumbai, India
rajatshenoy@gmail.com

Radha Shankarmani
Information Technology
Sardar Patel Institute of Technology
Mumbai, India
radha_shankarmani@spit.ac.in

Abstract—Writing code using natural language is a very exciting application of Neural Machine Translation. To achieve a small part of such an application, in this paper, we try to generate python source code snippets from natural English language descriptions using the Django dataset. We trained the self-attention based transformer architecture on the snippets from the dataset. We achieved a BLEU score of 64.29

Keywords—Transformers, tokenizers, Django, English, Python

I. INTRODUCTION

Programming is a vast field of engineering. More and more people are joining this profession and the skill is in high demand. But most of the programmers face this problem that the field is too fast and there is too much adaptation required if we want to stay in this field for a long time. Writing source code in different languages is very tough for programmers. They need to first refer to the documentation of those programming languages, understand the syntax and then write the source code for their program.

We strongly believe that programming should be based on the application of logic and problem-solving skills more as compared to having the knowledge of a particular programming language. That is why we propose this solution where we translate natural language (English) to a programming language (Python). This will allow programmers to write programs in natural language and not worry about the syntax in various languages and the silly syntax errors that keep on occurring all the time.

Semantic parsing is the problem of converting natural language constructs into logical constructs. One of the applications of semantic parsing is machine translation. Machine Translation uses software to convert one language to another language. We are working on a small part of this problem by converting natural English language constructs into python programming language syntax. We will be using the Django dataset for this task.

In the future, maybe when this proposed system is developed enough, even non-programmers will be able to write programs with the help of just natural language if they understand the logic to solve the problem. This paper is in the primitive stages of the research topic and is subject to further improvement and research.

II. LITERATURE REVIEW

Many papers have been published that follows a deep learning approach for natural language to source code conversion. [1] and [2] use attention-based encoder-decoder architectures. They use one or more than one LSTM layers with appropriate attention mechanisms.

Encoder/decoder based transformers introduced in [3] are now replacing LSTM based architectures. Transformer based architectures are now outperforming LSTM based architectures. State-of-the-art performance has been achieved in machine translation tasks[4], language modelling tasks, classification tasks [5], grammar correction tasks [6], summarization tasks[7], entity recognition, dialogue generation tasks [8], and other NLP tasks as well using the Transformer architecture.

Programming languages are of course not natural but many NLP approaches are being applied to them. The programming languages have grammar and syntax but with relatively lesser vocabulary than natural languages. There also are relationships between the vocabulary. Therefore there are many opportunities to model these languages using NLP models and tasks.

Such NLP tasks applied to programming languages definitely have some uses in software development. Examples of which include summarization/ translation of source code to generate documentation or summaries[9][10], using natural language query for code search [11][12], patching and detecting bugs using translation and error correction[13], code completion using language modelling or generation[14][15]

We will be using the Django Dataset [16] in this paper. The Django Dataset provides a dataset for annotated code snippets. For example, the dataset will contain the natural language intent 'call the params.get method with string 'KEY_PREFIX' and an empty string as arguments, substitute the result for self._key_prefix.' mapped to the python code 'self._key_prefix = params.get('KEY_PREFIX', '')'. The Django dataset contains 18805 such pairs of natural language intents mapped to python code. We explore how encoder/decoder based transformer architecture performs on these datasets.

III. METHODOLOGY

The main methodology behind translating natural language to code is machine translation. The basic idea is that a programming language is just like a normal natural language with much less vocabulary.

There exist mature solutions to translate one natural language to another. Some products like Google Translate do it really well. If programming languages are treated like natural languages and the machine translation solutions are applied to translate from natural language to programming language, we will reach a good solution at some point in the future.

So for machine translation, the solutions are recurrent neural networks (LSTMs) and Transformers. Most of the literature we reviewed used RNNs as a solution. But transformers are faster and more robust as compared to LSTMs and hence we chose to use transformers as a solution.

The transformers cannot process English or python directly. They need to be converted to numerical form to be able to use them. That is where tokenizers come in handy and we have built our custom tokenizer based on BertTokenizer and we have built our custom transformer which inherits from the TensorFlow model. The vocabulary generated from the tokenizer is used to train the transformer.

IV. IMPLEMENTATION

A. User Interface

The implementation consists of 3 parts. The first one is the user interface. It is built in React. React is a pretty popular javascript library used to build AJAX based frontend of a web app. The frontend consists of 3 user input types. The user can give the input via a file in which they can type in multiple lines of natural language intents. They can also type single line text intents. They can give audio inputs by speaking in English. They will get a corresponding python snippet as an output. We have also included a Code-editor for the user, where a person can edit the predicted code as well and send the corrected code to us that we can see in Fig. 1

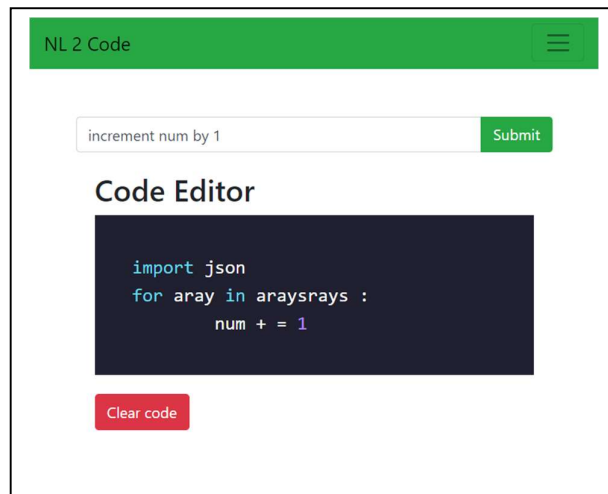


Fig. 1 – UI of our implementation

B. Server

The second part is the server side part. This part is made in Django. Django is a versatile server side python framework that has many tools for the convenience of web developers. Django consists of the basic routing of the web app and that is where the saved ML model is used to translate the English input received from the client side to python and send it back to the client.

C. Model

1) The Dataset

The third and the main part of the project is the model itself that does all the work behind the scenes. The dataset used is the ase15-django-dataset [16]. This dataset consists of 18805 records of English and corresponding python snippets. The data is distributed across code snippets of various libraries like Django, celery, JSON, etc. show in Fig. 2

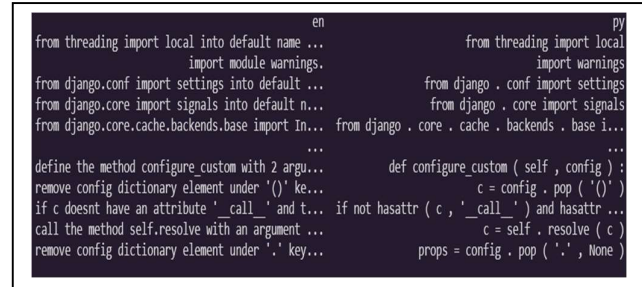


Fig. 2 – The Django Dataset [16]

2) Pre-processing

This dataset is split into train and test with train dataset consisting of 15000 records and test dataset consisting of the remaining 3805 records. Some preprocessing is done on the data first and then it is converted into the TensorFlow dataset format for better processing with the TensorFlow library. Tensorflow is an open-source library in python by Google and it is used for ML and DL operations. It has a vast number of libraries built into it and is popular due to its versatility. The model training is done in a Google Colab GPU environment. The TensorFlow version was 2.4.1.

3) Tokenizer

Then the bert_vocab_from_dataset library was loaded from tensorflow_text for the purpose of vocabulary generation from English and python. The vocabulary size for the bert vocab was set to 4000. The vocabulary for python and English was created and stored in a text file. Two Bert tokenizer models were trained using these txt files. One corresponds to English and the other corresponds to python.

A tokenizer is a tool that converts string input from the vocabulary into a numerical format. Tokenization is performed on the input before feeding it to the transformer model so that the model can understand the input. Also, the numerical output received from the transformer model is converted to a string or a human-readable format by

detokenizing using the tokenizer. Then a custom tokenizer is built and saved using TensorFlow's save model function.

4) Transformer

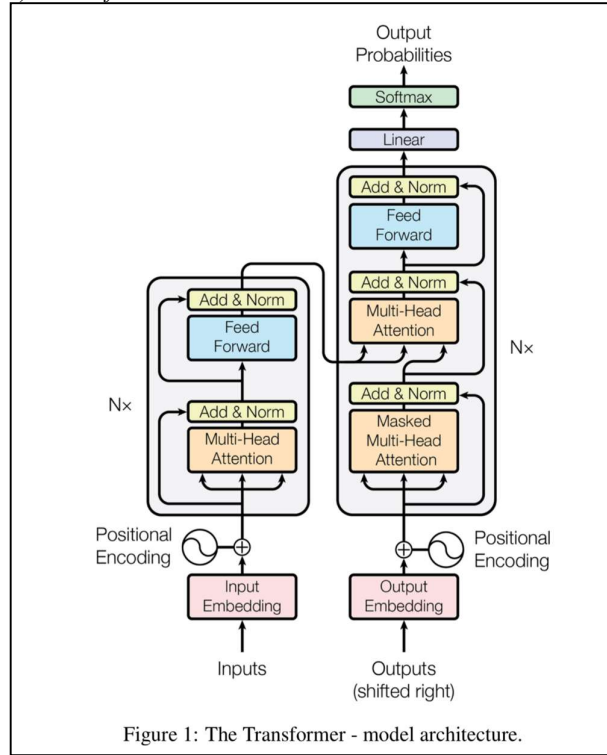


Fig. 3: Transformer Architecture [3]

The tokenized pairs of English and python are divided into batches with a batch size of 64 and buffer size of 20000. Then positional encoding is done. RNNs inherently know the position of a word in a sentence because it is a sequential model. A single word input is accepted sequentially by RNN. Whereas in transformers, the model itself does not have any idea about the position of the word in the sentence because of the inherent nature of taking and processing all the words simultaneously. Hence, we need to add an extra piece of information along with the word which contains details about the position of the word. This extra piece of information is known as positional encoding. Then we create a custom transformer model.

This transformer [3] consists of an encoder, a decoder and a final linear layer. The encoder first performs input embedding, then does positional encoding and then sends the data through multiple encoding layers. An encoding layer consists of a multi-head attention sublayer and a pointwise feed-forward network sublayer. The multi-head attention operation splits a linear layer into heads, then performs scaled dot-product attention, and then concatenates the heads into a final linear layer. The decoder performs an output embedding, then it performs positional encoding and then the data goes through multiple decoder layers. A decoder layer first performs masked multi-head attention, then performs multi-head attention with padding mask and then the data goes through pointwise feed-forward networks. We can see this architecture in Fig. 3.

We then set our hyper parameters. We have used the Adam Optimizer with a customized learning rate scheduler in accordance with the formula described in [3].

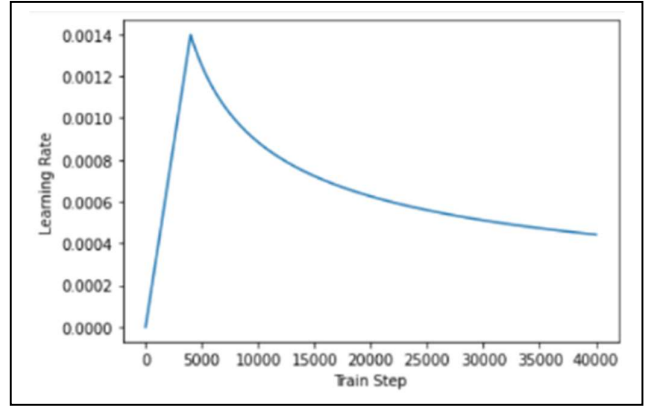


Fig.4: Learning Rate Schedule

We then defined our loss function and our accuracy metrics. We used checkpointing while training our model. We had got access to a single Testa T4 GPU for training. We then train the model, running 50 epochs on the training dataset. After this, we evaluate the model, save the weights to use them in the server.

V. RESULT

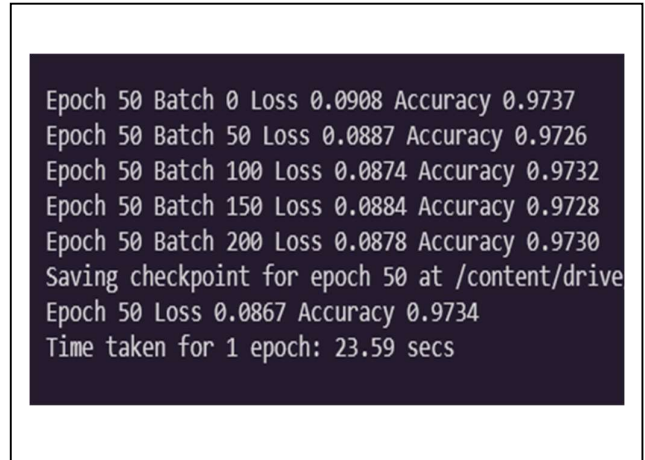


Fig.5: Accuracy and Loss Metrics

We trained the transformer on 15000 pairs of intents and snippets. We obtained accuracy of 0.973 and a loss of about 0.0867 on the 50th epoch with each epoch taking on average

We also calculated the BLEU Score of our model. To evaluate the quality of machine translated natural language from another language BLEU (bilingual evaluation understudy) is used. It used to determine how close is the machine translated text compared to the professional human translation. It is one of the most popular and inexpensive metrics. We have shown some of our BLEU scores and our average BLEU score in Fig. 6

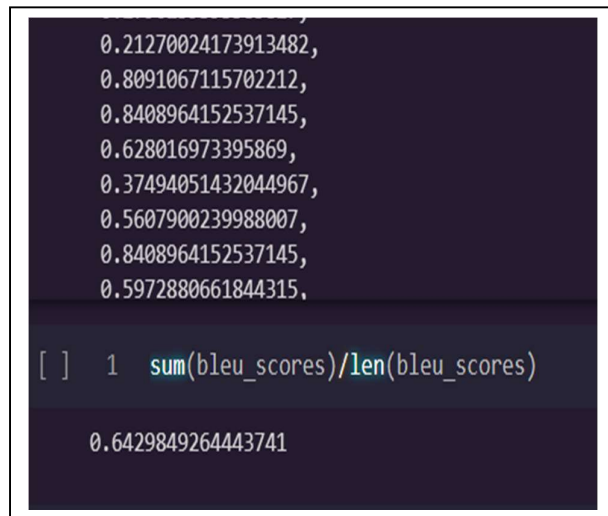


Fig.6: Our Average BLEU Score

We also tried giving new test cases to the model, some were successfully translated, some were not. The model struggles with variable names but fairly accurately predicts the syntax of Python programming language. We can see in figure 7 that the model gave the correct prediction for the English intent. But in figure 8 we can see that the model understood the syntax of classes but struggles with the variables a bit.

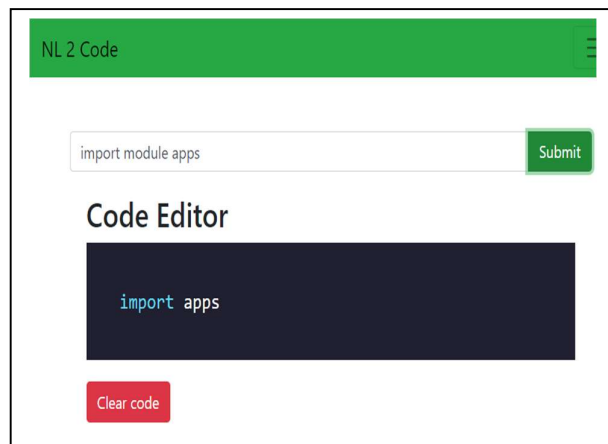


Fig.7: Correct prediction

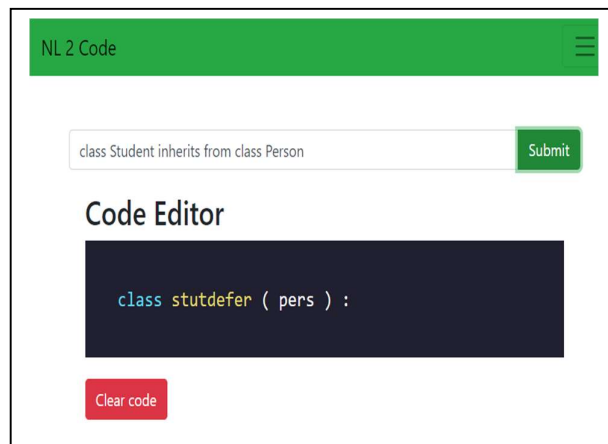


Fig.8: Incorrect prediction

VI. CONCLUSION

In this paper, we presented a way to translate natural language (English) to a programming language (python). We aimed to solve the “programming language barrier” in programming. We used the transformer model on the Django Dataset and achieved a BLEU Score of 64.29. Our model fairly understands the syntax of the Python Programming Language but struggles with variable names.

This topic is in primitive stages of research and needs more improvement. In future when this technology is mature enough, any person who does not know a programming language, will be able to write programs in that programming language.

VII. FUTURE SCOPE

The prototype we presented in this paper can be converted into a full system by adding functionalities like login, voice authentication, history of programs written, etc. Also, creating an extension for all major IDEs and text editors would be a great way for programmers to adapt to this technology. By this, programmers will be able to easily write code and edit it if required in their favourite editor. The model can be improved by training the model on a dataset with more number of records. Also experimentation can be done by using different tokenizers to improve the model even further.

REFERENCES

- [1] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696
- [2] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. arXiv preprint arXiv:1704.07535 .
- [3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems. pages 5998–6008.
- [4] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683
- [5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805
- [6] Christopher Bryant, Mariano Felice, and Edward Briscoe. 2017. Automatic annotation and evaluation of error types for grammatical error correction. Association for Computational Linguistics.
- [7] Yang Liu and Mirella Lapata. 2019. Text summarization with pre trained encoders. arXiv preprint arXiv:1908.08345
- [8] Paweł Budzianowski and Ivan Vulic. 2019. Hello, ‘ it’s gpt-2-how can i help you? towards the use of pre trained language models for task oriented dialogue systems. arXiv preprint arXiv:1907.05774.
- [9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400.
- [10] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 397–407.

- [11] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 933–944, New York, NY, USA. Association for Computing Machinery.
- [12] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.
- [13] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2019. Cpc: automatically classifying and propagating natural language comments via program analysis.
- [14] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735.
- [15] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformers. *arXiv preprint arXiv:2005.08025*.
- [16] Y. Oda *et al.*, "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation," *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, 2015, pp. 574-584, doi: 10.1109/ASE.2015.36.