Translation of Natural Language to Code Using Deep Learning Transformers

Jaideep Nuvvula nuvvula@usc.edu
Shenaz Narsidani narsidan@usc.edu
Apoorva Veerelli veerelli@usc.edu
Zixin Ding zixindin@usc.edu
Samhitha Gundam gundam@usc.edu

Abstract

Code generation transformers that are pre-trained on large corpora of programs have shown great success in translating natural language to code. Although these models do not explicitly incorporate program semantics during training, they are able to create valid code snippets for many problems. In this paper, we experiment on the translation potential of transformers and pre-trained transformers (CodeT5) in converting human languages to python source code and gather results using BLEU evaluation. An analysis is presented to depict the outcomes of both methods. In conclusion, we observe a huge performance difference between the two models, with CodeT5 outperforming the base transformer and conventional recurrent based architectures by a BLEU score of 49.06

1 Introduction

Deep learning has pushed the boundaries in achieving a significant performance of code generation from the text. Many approaches were carried out using rule-based mechanisms, encoder-decoder recurrent neural networks, sequence-to-tree models with attention and so on. The current state-of-the-art approach is the use of transformers to obtain an executable code snippet. Transformers (Vaswani et al.,, 2017) are powerful owing to their ability to parallel process the sequential data with the help of positional encoding coupled with an attention mechanism.

In the code generation field, the semantic parsing of text to a machine-understandable representation is crucial in order to achieve notable results. The sub-problem we choose to explore is to generate python code snippets from the text. For this project, we carry out the following experiments to tackle the problem steadily:

1. Address the translation of natural language to code using various transformer architectures.

2. Annotate the code snippets in terms of programming language semantics (operators, function names, literals etc.)

We evaluate the model performance using the BLEU score and compare the results with other published models. Through this project, we also take an attempt at the CMU CoNaLa, the Code/Natural Language Challenge to secure a place on the leader board.

Using Natural Language Processing facilitates the semantic representation of text in logical form. Also, we can naturally train the syntactic nature of the programming languages using unsupervised learning. Furthermore, any ambiguity present in the intent can be resolved and made comprehensible. Since English, the lingua franca of all programming languages acts as a medium of communication between humans and computers, applying natural language processing is an ideal solution for addressing the problem.

The accomplishment of the project would lead to solve repetitive code problems that can't be tackled by writing an abstraction. Other benefits include: mitigation of code errors, fast go-to-market turnaround for simple apps, stable underlying architecture, elimination of the need to learn programming languages and increased scope of users job market.

2 Related Work

There has been a lot of exploration of systems for natural language to code generation. Some of the works (Allamanis et al., 2015; Yin et al., 2017) have leveraged the grammar of code to extract features like the Abstract Syntax Tree for modelling (treating code and natural language as separate modalities) or use neural network models for the same while some treat PYTHON and its docstrings as fundamentally no different than other ‘natural’ languages, representing both source code and natural language docstrings as sequences of tokens sharing the same vocabulary. One such method is using PYMT5 (Clement et al., 2020), an encoder-decoder transformer that can both predict whole methods from natural language documentation strings (docstrings) and summarize code into docstrings of any common style.

Another notable related work is Codex (Chen et al., 2021), a language model developed by researchers at OpenAI, which is used to study how well it can write code in Python specifically. This is the model used to power GitHub copilot. The researchers fine-tuned the model on publicly available code (correctly implemented standalone functions) obtained from GitHub. They then evaluated the correctness of code samples generated by Codex over HumanEval, a new handwritten evaluation set which assesses language comprehension, algorithms, and simple mathematics.

In an attempt to improve the model performance at the CoNaLa challenge, an experiment was conducted to show the advantage of a self-attention-based transformer over recurrent layers in the text-to-code task. (Kusupati et al., 2022) From the paper in our course syllabus, the tradeoff between self-attention layers and recurrent layers for sequence transduction problems comes down to the dominant term between sequence length n and representation dimensionality d. (Vaswani et al.,, 2017) The analysis and empirical results are important factors to consider for our model design.

3 Experiment

3.1 Dataset

We have trained the transformer models on the CoNaLa datase, which was created for the CMU CoNaLa (Code/Natural Language) challenge. CoNaLa is a manually curated dataset crawled from Stack Overflow. It consists of 2,739 training and 500 test examples. Each example consists of a question ID (ID of the Stack Overflow question), intent (the natural language intent), rewritten intent (incorporating function and variable names into the intent to better reflect the full meaning of the code), and a snippet (desired output).

3.2 Transformers

Transformer models use encoder and decoder architecture adopting self-attention and cross-attention mechanism with positional word embeddings(Vaswani et al.,, 2017) which allows it to enhance its performance over other sequence-to-sequence models. The encoder as well as the decoder consist of stacks of layers which are identical. Each layer generally consists of a multi-head attention along with a feed-forward neural network. Through our implementation, we alter the number of attention heads and the layers in the transformer network to obtain better performance on the data

3.3 Pretrained Transformers

Most of current methods, follow the conventional NLP pre-training techniques on source code by treating it as a sequence of tokens like natural language. They ignore the rich structural formal in programming language, which is vital to fully comprehend the semantics. CodeT5 [1] builds on the architecture similar to that of T5, additionally incorporates code-specific knowledge to the model It was pre-trained on CodeSearchNet data by optimizing the following objectives.

- Masked Span Prediction (MSP) randomly masks spans of arbitrary lengths and the decoder recovers the original input. It captures the syntactic information input and learns cross-lingual representations.

- Identifier Tagging (IT) It is applied only to the encoder. It differentiates identifier tokens (variable or function name) from the rest of the tokens.

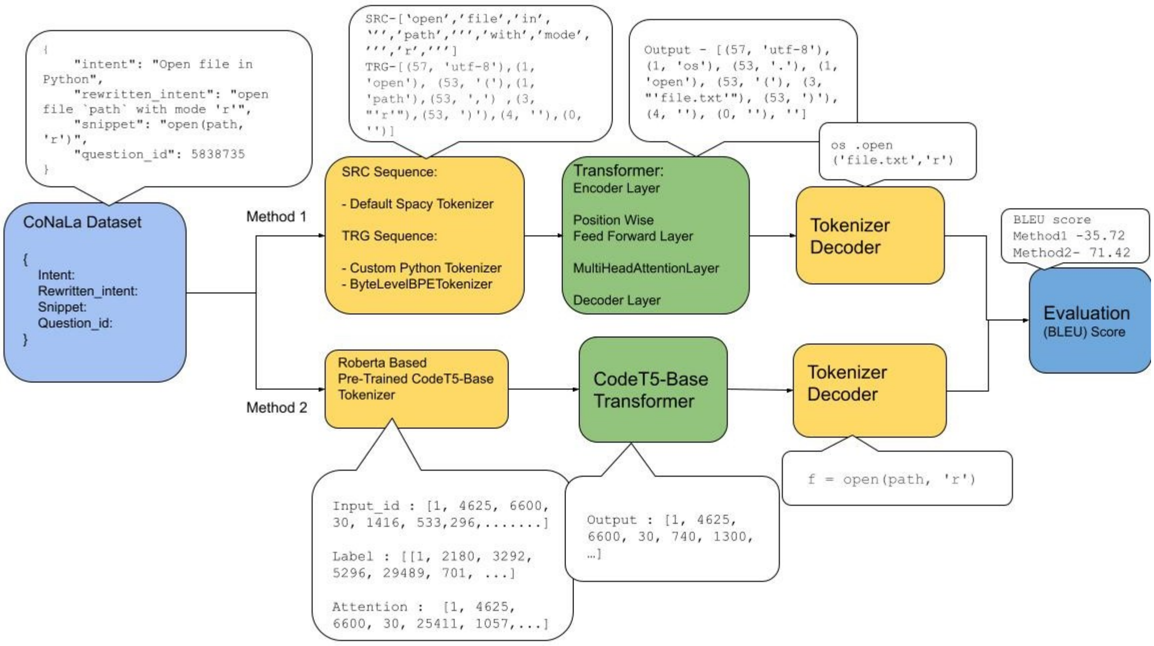
- Masked Identifier Prediction (MIP) - It only masks identifiers and employs the same mask of one unique identifier across the code. It comprehends the semantics based on the obscure code.

- Bimodal Dual Generation (dual-gen) - It optimizes the conversion from code to its comments and vice versa.

3.4 Evaluation Metrics

The results of the model are evaluated according to BLEU score after tokenization. BLEU has been frequently reported to correlate well with human judgement, and thus remains a benchmark for the assessment of any machine-translated text. Scores

[1] https://blog.salesforceairesearch.com/codet5/



```
{
    "intent": "Open file in
Python",
    "rewritten_intent": "open
file `path` with mode 'r'",
    "snippet": "open(path,
'r')",
    "question_id": 5838735
}
```

```
SRC-['open','file','in',
'`','path','`','with','mode',
''','r',''']
TRG-[(57, 'utf-8'),(1,
'open'), (53, '('),(1,
'path'),(53, ',') ,(3,
"'r'"),(53, ')'),(4, ''),(0,
'')]
```

```
Output - [(57, 'utf-8'),
(1, 'os'), (53, '.'), (1,
'open'), (53, '('), (3,
"'file.txt'"), (53, ')'),
(4, ''), (0, ''), '']
```

```
os .open
('file.txt','r')
```

```
BLEU score
Method1 -35.72
Method2- 71.42
```

```
Input_id : [1, 4625, 6600,
30, 1416, 533,296,.......]

Label : [[1, 2180, 3292,
5296, 29489, 701, ...]

Attention :  [1, 4625,
6600, 30, 25411, 1057,...]
```

```
Output : [1, 4625,
6600, 30, 740, 1300,
…]
```

```
f = open(path, 'r')
```

Figure 1: Implementation

are calculated for individual translated code segments by comparing them with a set of executable code snippets. These scores are then averaged over the whole corpus to reach an estimate of the translation's overall quality. Since BLEU assessment does not account for intelligibility or grammatical correctness, the metric is suitable for verifying code correctness. In addition, the CoNaLa challenge only accepts BLEU score results, hence the major focus was targeted towards one criterion.

4 Implementation

You can find our implementation of the Code through our Github Repository. [2]

4.1 Method 1 - Transformers

We initially preprocess the data to convert the data into the numerical form using different tokenization methods. We tokenize the intent in the data using a tokenizer provided by SpaCy[3]. SpaCy first tokenizes the text, i.e. segments it into words, punctuation and so on. This is done by applying rules specific to each language. Similarly, we tokenize the code snippet in python using a custom python tokenizer using Python tokenize[4] library that is specifically built for Python source code. Since our dataset is small containing only 2379 samples, we added a step for data augmentation to increase the size of the dataset. While tokenizing the python code, we mask the names of certain variables randomly(with 'var1, 'var2' etc) to ensure that the model that we train does not merely fixate on the way the variables are named and actually tries to understand the inherent logic and syntax of the python code.

This data is then used to train the transformer architecture. We add label smoothening to cross-entropy in our loss function so that our model does not become too confident in predicting some of our variables that can be replaced via augmentations. Label Smoothing is a regularization technique that introduces noise to the labels. This accounts for the fact that datasets may have mistakes in them, so maximizing the likelihood of $logp(y/x)$ directly as cross-entropy does, can be harmful. We set the dropout rate to 0.1, using an adam optimizer with learning rate 0.005 by default for our transformer. To understand the performance of the transformer better we tried changing the setting for the number of layers and attention heads to obtain the best performance metric. We feed our data to the transformers and train the model using backpropagation. The output of the transformer is expected to be the tokenized code output which is untokenised using Python untokenize function from the aforementioned tokenize

[2] https://github.com/samhithasamg/CSCI-544-Project-Text-to-python-code-converter

[3] https://spacy.io/usage/spacy-101

[4] https://docs.python.org/3/library/tokenize.html

Intent	Ground Truth	Transformer Generated Code	CodeT5 Generated Code
open file ‘path‘ with mode ’r’	open(path, ’r’)	os .open (’file.txt’,’r’)	f = open(path, ’r’)
get a list of all keys in Cassandra database ‘cf‘ with pycassa	list(cf.get_range().get_keys())	[var_1 for var_1 in list (var_2 .items ())for var_1 in var_2]	[k for k in cf.keys()]

Table 1: Outputs

#attention heads	#layers	BLEU score
4	3	22.89
8	3	24.05
16	**3**	**31.16**
16	6	13.84
CodeT5-base		**49.06**
reference model		**16.36**

Table 2: Results

library. This produces the expected translation of the given intent into a python code. We evaluate the performance of these models using BLEU score. The results can be seen in Table 2

4.2 Method 2 - CodeT5-base Transformer

CodeT5 uses a code-specific BPE (Byte-Pair Encoding) tokenizer trained using the HuggingFace Tokenizers library. We pre-processed the intent and the code snippet for the model using RobertaTokenizer[5], with the files codeT5 tokenizers. In our experiment, we used codeT5-base[6] version for the model and the tokenizer with maximum length for text as 48 and maximum length for target i.e. code as 128 with padding and truncation enabled. The codeT5 tokenizer is used to tokenize both the annotated intent and the snippet resulting in tokens referred as input id and labels respectively and additionally attention values. These resulting values are used to train the transformer for our use case with a learning rate of 0.0003. The tokenizer is used to encode the resulting outcome of the codeT5 transformer to code

5 Results and Discussion

The implemented models with Spacy and Python tokenize library gave significantly better results when compared to the referred base model(Kusupati et al., 2022)(BLEU score of 16.36) which used SentencePiece tokenizer on text and code. We experimented with our model by varying the heads and layers. The model that gave the best score was the one with 16 heads and 3 layers yielding a BLEU score of 31.16. CodeT5, being a pre-trained model gave a BLUE score of 49.06. From the outputs generated in Table 1, we observe that although the model is not generating accurate output, it is learning the semantics of the programming language.

6 Future Work

While our implementation has improved the performance of the model over our base paper(Kusupati et al., 2022), we realise certain additional methods can be implemented to further try to improve the performance. Similar to our usage of the python tokenize library, one can use multiple other tokenizer models such as Roberta-tokenizer, and Sentence Piece for tokenizing the data to understand how this affects the performance. Similarly, through our project we use the codeT5-base pretrained transformer, but implementations of different versions of code-t5 like code-t5-small, code-t5-large, code-t5-large-ntp-py can be used for the experiment. Lastly, we used BLEU metric for evaluating the performance of our model but this metric does not account for syntax errors, and for the possiblilty of multiple code snippets producing the expected output as ground truth. Hence, one can implement and evaluate the models using a new evaluation metric that combines BLEU score and the syntactic and semantic correctness of the generated code.

References

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefine-

[5] https://huggingface.co/docs/transformers/model$_{doc/roberta}$

[6] https://huggingface.co/Salesforce/codet5-base

ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *In Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.*

Miltiadis Allamanis and Daniel Tarlow and Andrew D. Gordon and Yi Wei 2015 Bimodal Modelling of Source Code and Natural Language, ICML

Yin, Pengcheng and Neubig,Graham A Syntactic Neural Model for General-Purpose Code Generation 2017 https://doi.org/10.48550/arxiv.1704.01696

Clement, Colin and Drain, Dawn and Timcheck, Jonathan and Svyatkovskiy, Alexey and Sundaresan, Neel PyMT5: multi-mode translation of natural language and Python code with transformers 2020

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I. and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. *[online] arXiv:2107.03374* .

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, Sida I. Wang 2022. Natural Language to Code Translation with Execution *arXiv:2204.11454v1 [cs.CL].*

Kusupati, Uday and Ailavarapu, Venkata Ravi Teja 2022. Natural Language to Code Using Transformers *arXiv.2202.00367.*

https://www.kaggle.com/code/rhtsingh/text-to-code-generation-with-tensorflow-mbpp

https://towardsdatascience.com/building-a-python-code-generator-4b476eec5804

Division of Labor

Samhitha Gundam: Experimental Research of different models, Implementation of the transformer model and Data pre-processing, Scope of Improvement, Documentation

Shenaz Narsidani: Experimental Research on various models, Contributed to leveraging CodeT5 Transformer, pre-processing data for CodeT5 model and identifying the scope of improvement, Documentation

Jaideep Nuvvula: Proposal of the reasonable approaches for machine-translation of text to code. Focus on participation and results submission to CoNaLa code challenge by CMU. Investigation of a fitting evaluation metric to capture the quality of text to code generation.

Apoorva Veerelli: Experimental research of tokenizers for the pre-processing of data, Researching and evaluating the different available datasets to use, Documentation

Zixin Ding: Experimented with customizing transformer tokenizer targeting Python programming language, researching self-attention transformer model architecture and evaluation methods.