Natural language To Code Translation

Guy Yanko (ID. 305737587)
Shahaf Pariente (ID. 204605299)

Submitted as a final project report for Natural Language Processing, The College of Management Academic Studies, 2020

1 Introduction

Our project is based on the "Conala-Challenge". The main idea of that challenge is to create model which will translate a natural language functionality request into a python one-liner code. knowing about very accurate machine translation models, we were intrigued to investigate the accuracy of translate natural language to coding language using some machine translation models.

1.1 Related Works

Now-days, there is no published related works. On the "Conala-Challenge" official website, there are two results submissions (submitted by the challenge organizer) according to BLEU score evaluation.

2 Solution

2.1 General approach

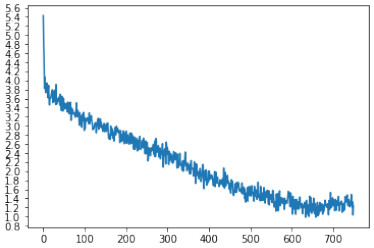
The general approach to solve that translation problem, is seq2seq model, using encoder and decoder with attention. The main idea is to train the model to generate a one-liner python code according to natural language input sentence. During the experiment, we will use 3 different models: 1. Tokenizing the code snippets by special cases - with random initialized embedding weights 2. Tokenizing the code snippets by special cases - with pre-trained embedding weights 3. Code snippet as one token - with random initialized embedding weights Evaluating first model, relative to the second model, will teach us if pre-trained embeddings is better than random weights embedding, when the training set is small for the related translation problem. The third model, is about to solve a translation problem based on the assumption that the data set will act like "python functions vocabulary" - having a natural language sentence, the model will predict the most similar code form the vocabulary (experimental results do not presents in the report).

2.2 Design

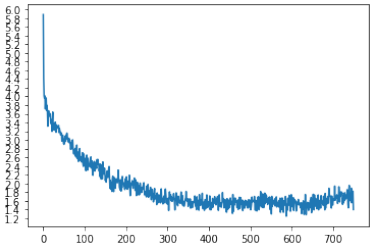
All 3 models we trained, are based on seq2seq model - the encoder contains embedding layer and one GRU layer. The decoder contains embedding layer, attention layer, dropout and GRU layer. Each model is different by the embedding layer - random or pre-trained. Both models uses 300d embedding size and hidden size for the GRU layers. To arrange the data set to fit the model, we've implemented a JSON file reader that create a list of pairs- [intent, snippet]. the intent is the natural language sentence and the snippet is the suitable one-liner code. After organizing the data set, we created two vocabularies. The natural language vocabulary is similar with all three models. The code vocabulary is different between third model and second/first model by the tokenization as described at the general approach section. Training the models took about 1 hour per model. During working on the model, we had a few challenges. first we had to figure out what is the best way to build the code vocabulary - tokenization for the best results. For the training part, we had to find the best pre-trained embedding to avoid from learning bad correlations between natural language and code generation. The final challenge was the evaluation - how we can evaluate the accuracy of the models?

3 Experimental results

Pre-trained embeddings weight model training results:
Model received full match on 352 snippets out to 2300 in training set.
Training time: 44m 57s
Loss plot:



Non-trained embeddings weight model training results:
Model received full match on 100 snippets out to 2300 in training set.
Training time: 66m 59s
Loss plot:



According to "Conala-Challenge", we evaluated the model by BLEU score with 4-gram evaluation.
Pre-trained embeddings weight model BLEU score: 0.135
Non-trained embeddings weight model BLEU score: 0.126

Example for random user intent-snippet pair: ["check if item from list 'b' is in list 'a'","print(x in a for x in b)"]
Pre-trained embedding model result: print('['join()set(a))]
Non-trained embedding model result: any(any(x in a for x in b) for x in a(
As you see, the pre-trained model recognize 'a' as a set. Also uses join as its common use ".join(). The non-trained model trace the generator template in python.

4 Discussion

Our experiment main purpose was to translate natural language to code. After collecting all of the results, we found the translation problem very hard to reach sufficient accuracy level. During our test, we found that using pre-trained embeddings is very useful for reduce training time, but more important - get more accurate results. Also, regarding to translation problems, we found that high drop-out probability could harm the learning of the model. After we took a close look at the results of the test set we found that even though the model accuracy is very low, it produce very good coding patterns. Taking all of our conclusions from the experiment to higher level - Different natural languages have a great common factor that eventually gives us good results in seq2seq translation models. However, our experimental results, indicates that despite human capability to find the common factors between natural language to programming languages, the computational models, find it very hard.

5 Code

Pre-trained model:
https://colab.research.google.com/drive/1N1Pyt4GiutxHf978Q7_eJNmHmG24FhF5
Non-trained model:
https://colab.research.google.com/drive/1YHmWYDkIwCvdxi7PNN-inLx29lO8CAuA
Untokenized model:
https://colab.research.google.com/drive/1UoK9Sc-6vJSqVZtK4cHBx42tog-YCEcW

References

Pytorch documentation:
https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
Hugging-Face documentation:
https://huggingface.co/models
Lecturer presentations: DR. kfir bar