Semantic Code Search

John Nguyen and Uzair Inamdar

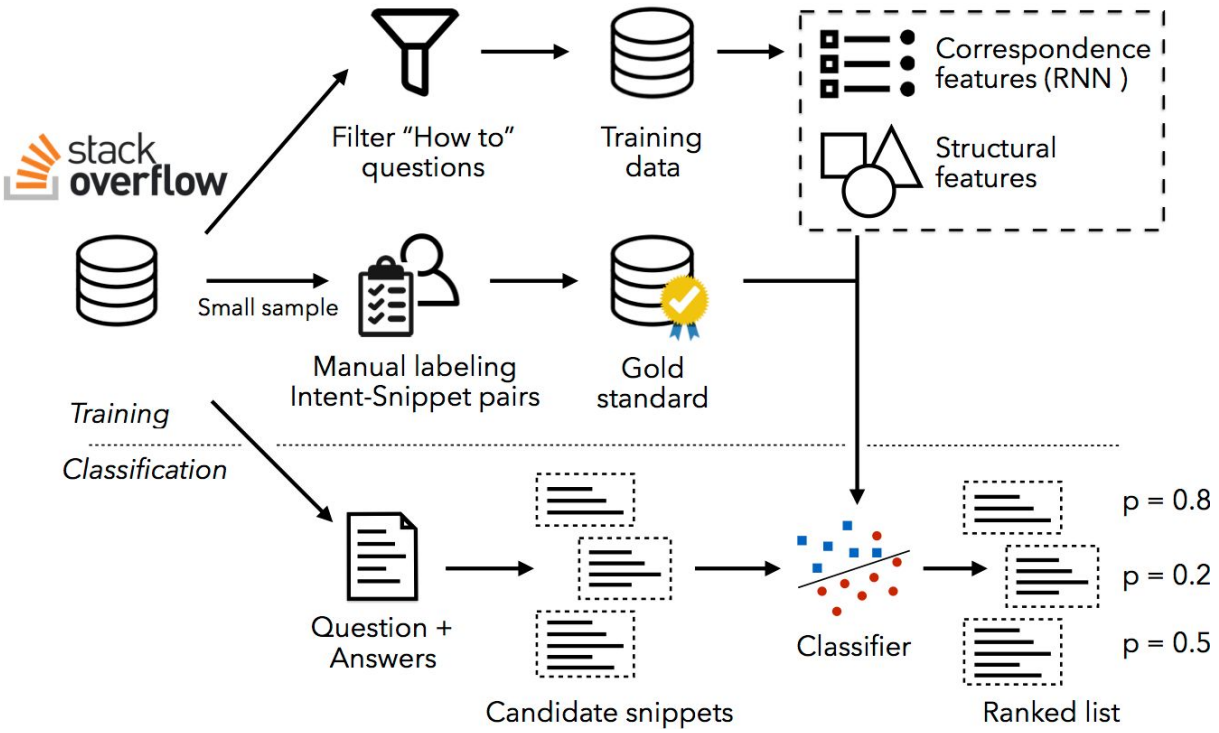
1. Motivation

StackOverflow (SO) is by far the most used service by developers who look up code. It serves as the center place at which professionals and amateur programmers look to the solutions for their code problems. According to the 2019 SO developer survey, 60% of the developers spend time daily on SO, 50% of which do so multiple times of the day. What's striking is that 97% of people who visit SO, visit it to find answers to code problems. This proves that SO as a service is very robust and is extremely effective in getting the desired solutions. The survey results not only shows the importance of such a service but also the need for other such similar services. The appeal that draws so many people to SO is understood by us and several other researchers in this space to convert a given natural language intent to code. This is often perceived as a translation task due to its text-to-text nature, but can be approached in many different ways.

2. Prior Work

Most prior works focus on jointly embed code snippets and natural language descriptions into a high-dimensional vector space, in such a way that code snippet and its corresponding description have similar vectors. CODEnn (Code-Description Embedding Neural Network), the current state-of-the-art bi-directional LSTM, learns to unified vector representation of both source code and natural language queries so that code snippets semantically. Our work takes a different approach to semantic search. We focus on translation, a task to learn a mapping from natural language vector space to code vector space. We drew inspiration from the Latent Predictor Network (LPN), a state-of-the-art sequence-to-sequence code generation model (Ling et al., 2016), and NL2Code, a syntax aware NMT which leverages ASTs and grammar rules (Yin et al. 2017). Our work uses the state-of-the-art NMT, Transformer model (Vaswani et al. 2017), to simply treating code as a natural language as one would for translating from English to German.

3. Data



We were directed to by the professor to look at CoNaLa dataset that is used for the CoNaLa challenge. The dataset is created by scraping SO questions relating to code answers. The dataset contains a total of 2,379 manually annotated intent and snippet pairs, and around 600k automatically mined intent-code pairs. One important thing to note is that the automatically mined dataset has multiple entries of the same question, but with different answers. Each of these different answers has a probability of it being the right answer attached to it. We use all of these pairs to train our model.


```
{
   "intent": "Prepend the same string to all items in a list",
   "rewritten_intent": "prepend string 'hello' to all items in list 'a'",
   "snippet": "['hello{0}'.format(i) for i in a]",
   "question_id": 13331419
},
```


Annotated


```
{
   "parent_answer_post_id": 14694669,
   "prob": 0.8237010308958093,
   "snippet": "soup.get_text().replace('\\n', '\\n\\n')",
   "intent": "Converting html to text with Python",
   "id": "14694482_14694669_0",
   "question_id": 14694482
}
```


Mined

4. The Model and Codebase

Overview

We make use of Tensor2Tensor's (T2T) transformer model for our project. It consists of 6 hidden layers, and we feed batched data of size 1024. The T2T is modular and easily extensible. Different applications of T2T models are called as problems and each problem has its own class. People can either use pre-existing problem classes with their own data, inherit the broad problem classes into their own problem classes, or build a problem class entirely from scratch.

Our code resides in the **usr_dir** directory, but makes use of *Text2Text* problem class from the **text_problems.py** file inside the **tensor2tensor/data-generators** folder. This is important because one might need to view or change the default methods that are inherited from the parent class. The pre-existing classes have good documentation about each class member is used. 2 methods however, are the most important that we inherit and overwrite in our problem classes -

1. *vocab_type()*: This is responsible for automatically generating the vocabulary for the given data. There are 3 default options: character, subword, and token. While the first two are used to automatically generate vocab, the last one is available to those who wish to feed their own vocab file.

   The type of vocab you use can have a great effect on the performance of the model and how it is trained. As we discuss later, switching just changing the vocab type of our model made our trained model better at outputting code.

2. *generate_samples()*: This function is responsible for generating samples that are fed to the model during training. The format for the sample that is to be fed must be in the form of : *{"inputs":* <an intent>*, "targets"*: <a code snippet>*}*. We clean and preprocess the data before we dispatch each line in this format to the model for training.

Explored Techniques

The first technique we used is to just use the regular word tokens and feed it through the transformer model. This did not perform very well. Even though it generated valid code tokens, the sequence/structure was not right, and it was missing punctuation symbols like brackets. Apart from missing few optimizations, the primary reason the model failed was because it lacked a lot of the semantic information of the code and only relied on lexical information.

The second technique we used is using BPE for generating our vocabulary. Our latest functional model makes use of this technique, and is able to achieve a much better result

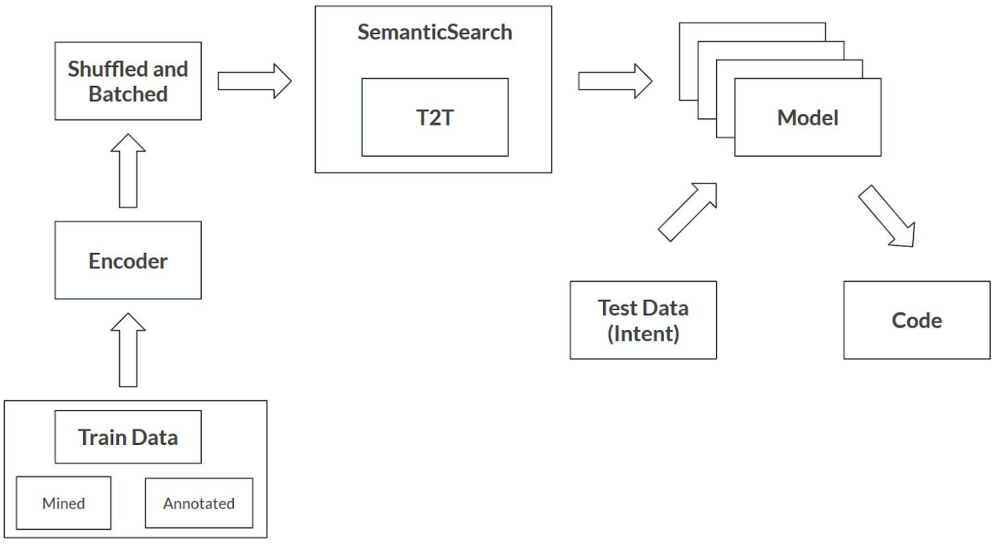
than by just using regular word tokens. The way BPE works is that learns the commonly sequences of characters as subtokens and adds them to the vocabulary. This improves the handling the out of vocabulary words, especially the ones that are compositional in nature consisting in-vocab subtokens.

The third technique that we were partially successful in implementing was using AST. This technique is housed inside the *SemanticSearchAst* problem class. We believe that using AST will give our model a significant boost in performance. Using AST will provide the structural information that can improve the prediction of the model. The main problem we were running into was deserialize the AST string to code. This was essential for code generation. Since we ran out of time and resources, this remains in the list of future work that we would like to implement.

Training

As mentioned earlier our code resides inside the **usr_dir** directory, and consists of the semantic_search.py file that houses 3 semantic search problems. These 3 problems use different techniques to tackle the problem of semantic code search. To make use of our model, you must install the extended T2T framework using the *pip* installer with the "-e" flag. The path to the installed needs to point to the path where the setup.py exists.

Once the installation is complete, the first step is to use the *t2t-datagen* command by specifying the path of the data among other things. We then run the *t2t-trainer* to train the model by specifying transformer model as the model of your choice, the problem class you want to use, and specifying the hyperparameters and location of the data. We test our trained model using the *t2t-decoder* by passing it an input file containing multiple test intents.



Results

Model	BLEU
Baseline Seq2Seq	**14.26**
Transformer - BPE	**6.01**
Transformer	**16.41**

Generating an exact match for complex code structures is nontrivial, we follow Ling et al. (2016), and use BLEU as our evaluation metric. Evaluation results for Python code are listed above. The baseline Seq2Seq model is provided by the authors of the Conala dataset. Transformer - BPE is our transformer model without BPE tokenization. Finally, transformer is the best performing model. Utilized Tensor2tensor built in sub-word tokenizer, we were able to achieve significant improvement over the baseline model, placing ourselves in the top 5 on the leaderboard.

Future Work and Conclusion

The semantic Code Search project has been a great learning experience that has opened doors for countless paths to explore. Little has been achieved in terms of the ideas that we thought of due to time and resource limitations and we plan on continuing to work on this, adding more problem classes, or improving the existing ones.

As mentioned before AST is something that we were partially successful in completing, and hope would improve the accuracy of our model. But there are 2 more interesting techniques we hope to implement - **code2vec** and **Variational Autoencoders**. These 2 methods provide embedding of a lot of information in a compact form that can be used for number of applications. These 2 techniques can open doors for better code summarization techniques as well, that are a pleasant side-effect of our work.

References:

[1] Yin, Pengcheng and Deng, Bowen and Chen, Edgar and Vasilescu, Bogdan and Neubig, Graham. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. MSR 2018.

[2] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomas Kocisky, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In Proceedings of ACL.

[3] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and
[5] I. Polosukhin. Attention is all you need. In Neural Information Processing Systems (NIPS), 2017.