Automatic Code Completion using Language Model and Code Template

Md Masudur Rahman
Department of Computer Science
University of Virginia
masud@virginia.edu

Abstract

Automatic code completion helps the developer to write code efficiently and effectively. Developer spends much time on writing same code repetitively throughout the life cycle of the project development. We can think of such repetitive code snippet as *code template*, which serves a single or some limited number of functionalities. Some modern code editor provides an option for the developer to save code template manually so that they can reuse those code when needed. Maintaining such code template and use it manually is difficult for the developer. The language model is suitable to capture repetitiveness of source code and on the other hand, code template can capture structural information for a big chunk of repetitive code. In this work, we proposed a novel approach for this code completion task leveraging language model and code template. Our preliminary result shows the effectiveness of using Neural Network based language models such as RNN, LSTM, Bidirectional LSTM over n-gram based language model. We also discuss the potential way of using templates to improve the performance of the language models as well as code completion.

1 Introduction

Given source code context our task is to suggest possible next code snippet in the form of tokens or statements for the developer.

Due to the availability of massive source code from different publicly available online repository, such as GitHub, StackOverflow, SourceForge etc., there are huge opportunities to utilize these valuable data properly. Researcher proposed several automatic approach (Allamanis and Sutton, 2014; Jacob and Tairas, 2010) to collect code template given previously written source code. We can also get the code template from the available public repositories. We can collect template from StackOverflow accepted answer's code snippets. Any answer in StackOverflow is accepted by the person who asks the question and thus indicates non-buggy code. The popular vote of such accepted answer indicates how frequent such code snippets are, thus can be considered as code templates. Figure 1 and figure 2 are two such code templates which are very common and repetitive for the Java programmers.

```
try{
        int num1 = reader.nextInt();
        int num2 = reader.nextInt();
        int output = num1/num2;
        System.out.println ("Result = " +output);
}
catch(ArithmeticException e){
        System.out.println ("Divided by zero");
}
```

Figure 1: Example Code Template: Division exception

```
try{
        int n = reader.nextInt();
        System.out.println(array[n])
}
catch(ArrayIndexOutOfBoundsException e){
        System.out.println ("ArrayIndexOutOfBounds");
}
```

Figure 2: Example Code Template: Array index exception

The language models are very successful in predicting natural language text. Researchers showed

that source code also exhibits naturalness (Hindle et al., 2012) similar to natural language. Such observation motivates the application of language model in source code predictions. But unfortunately source code follows some semantic and structural information which is tough for language model to capture. We can achieve better token prediction by leveraging the semantic and structural information from the template code. From the context of the code, we can match the possible template from a collection of code templates. Such choice of code template then guide the language models to predict better token that might serve the need of the programmers. In this project, we considered N-gram, RNN, LSTM, and Bidirectional LSTM as our language model. The experimental results show the better performance of Neural Network Language model, in our case Bidirectional LSTM, to predict next tokens.

2 Methodology

We used statistical N-gram and neural network, RNN, LSTM, Bidirectional LSTM language models to capture to the naturalness of source code. The language model helps to generate possible next tokens. In the fig. 3, after programmer writes first two lines, our model will first generate next three lines, which is *try* block, except the red marked lines. After that our model will have more context including this generated *try* block. So the model will predict the next *catch* block and *finally* block(marked blue). At that moment if a developer writes *int output = num1/num2* inside this try blcok and triggers the tool, the tool will match the added code or changes with corresponding templates from the code template collection. The tool can match this context with the template in fig 1 and thus suggest the red marked *catch* block here. An interesting observation is that, here the change *int output = num1/num2* is not natural to the other context. In another word, this arithmetic operation statement is not very common with the file reading operation in the fig 3. So to suggest the corresponding catch block would be difficult merely using language model.

3 Evaluation

We evaluated our language models on projects source code collected from GitHub. Though our proposed model should work with code template and language model, in this project we evaluated our language models and leave the augmentation of the models with code template as our future work.

```
File file = new File("file.txt");
BufferedReader reader = null;
try {
   reader = new BufferedReader(new
FileReader(file));
   ..........................................
   int output = num1/num2;
} catch (IOException e) {
   e.printStackTrace();
}
catch(ArithmeticException e){
      System.out.println ("Divided by zero");
}
finally {
   try {
      if (reader != null) {
         reader.close();
      }
   } catch (IOException e) {
   }
}
```

Figure 3: Applying template to complete code

Data set & Setup

We collected two large GitHub Java projects : Maven and Lucene, which were used before in source code language modeling. StackOverflow questions answer's code snippet is a great source of human written code template source. We have collected this code snippet using StackOverflow data dump. Different automatic template generation approaches, which discussed before, can also be used to generate the template from a given source code repository.

As we are interested only in source code, we merged all the *.java* files into a single file for each project. Before feed to the language model we prepossessed the data and removed token with term frequency less than 5 (this can be varied depending on vocabulary size) and replaced them by *unknown* keywords. For each project, we took 90% for training the model and rest of the 10% for testing in terms of the line number.

Model Implementation & Training

For N-gram language model we estimate the log likelihood of tokens on training data and evaluate the performance on test data. For the Neural

Network based language model (RNN, LSTM, Bidirectional LSTM), we used Keras(Chollet, 2015) to build each model. For all the neural network models we used one hidden layer of *64* dimension. The output of each model is the size of vocabulary which is *4679* in the *Maven* dataset. We used *softmax* as activation function for each model. From the Keras (Chollet, 2015) library, we used *categorical-cross-entropy* as our loss function that the model need to optimize.

We trained those models on training data and saved model after each iteration. Each iteration consists of 10 epochs and we trained the models up to 10 iterations and reported the best performing models in term of perplexity, which is the standard language model evaluation metric. For the *Maven* projects we tested the models on 7k tokens with a total vocabulary size of 4679.

Preliminary Results

We reported language models (N-gram(n=1), RNN, LSTM, and Bidirectional LSTM) perplexity comparison in Figure 4 on *Maven* project. With the limited training time (iteration and epoch), LSTM and Bidirectional LSTM outperform N-gram(n=1) based language model. Surprisingly , RNN model performs worst than N-gram (n=1) (perplexity score 75.55). This is because of lack of training time (both iteration and epoch). RNN achieved this 94.54 perplexity in the third iteration which best among its other iteration. On the other hand, LSTM achieves its best perplexity 45.4 at iteration 10 and we believe the increase in iteration will lead to even better perplexity value. Source code exhibits long-term dependency (one file's code might be similar what are written in other files). This dependency is hard to capture by both N-gram and even RNN. We got even more interesting results with our Bidirectional LSTM model, which achieved 33.79 perplexity score hence the best model on this dataset. As structural information is very common in the source code, most cases one lines of code or one token depend on both the previous context and next content. For example, in the case of a statement inside a *loop*, may depend on the other statement after that and also the condition on the for loop before that targeted statement. This scenario is well captured by Bidirectional LSTM model.

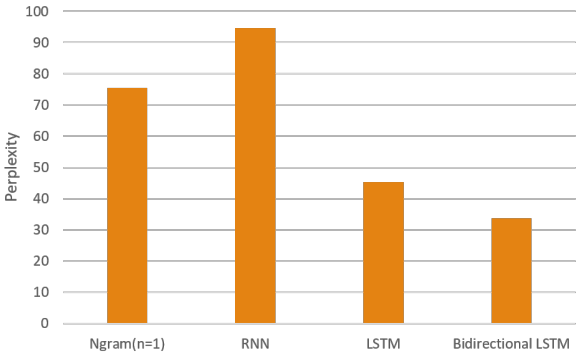

Figure 4: Language Model Comparison on Maven project source code

4 Related Work

There are many research works (Allamanis and Sutton, 2014; Jacob and Tairas, 2010) focus on automatic template collection from source code repository; though code template generation not necessarily mean code completion. (Hindle et al., 2012) used statistical N-gram model for code completion which was improved by (Tu et al., 2014) by incorporating localness information of source code. (Raychev et al., 2014) proposed N-gram and RNN based language model for code suggestion and showed that the combination of this two models probabilities performed better in code completion task. Code syntax and semantics in the form of Abstract Systext Tree (AST), Control Flow Graph (CFG) and Program Dependency Graph (PDG) are used (Nguyen and Nguyen, 2015) to suggest possible code completion. (Nguyen et al., 2016) proposed API code completion using statistical learning from fine-grained changes. They used code changes pattern from development history and code context to learn the likelihood of code completion candidates. (Raychev et al., 2014) used statistical language models (N-gram and RNN) to recommend missing code. None of the previous works use the structural information of code such as code template to augment the code completion accuracy of language models.

5 Conclusion and Future Works

In conclusion, we proposed a novel approach to automatically suggest code completion during programming for the developer. In our method, we leveraged statistical language models and code template information for better code completion.

We would like to run the experiments on other large projects datasets that we collected. We would also like to tune the neural network models' parameter settings for the better prediction results. As discussed in the evaluation section, we will augment the language model by including template structural information in the language models. In addition to that, we would also like to explore this hybrid model in other NLP applications such as automatic email reply, real-time question answering etc.

Acknowledgments

I would like to thank Professor Kai-Wei Chang for introducing me to different NLP techniques and encourage me to do research on NLP. I am grateful to him for his continuous support and guidelines throughout the project. I also thank Professor Baishakhi Ray for her thoughtful comments and encouragement to do this project.

References

Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM.

François Chollet. 2015. Keras. `https://github.com/fchollet/keras`.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE.

Ferosh Jacob and Robert Tairas. 2010. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, page 104. ACM.

Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 858–868. IEEE Press.

Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522. ACM.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM.