CSCI680: AI4SE

Assignment 1: N-Gram Language Model

Khai-Nguyen Nguyen

I. Introduction

N-gram is a statistical language modeling approach that helps predict the next token (a word, subword, or character) by calculating the probability of that token appearing conditioned on the last N-1 tokens.

Supposed we have the input sequence of "The cat sits over ___", a 3-gram (or tri-gram) would return the probability distribution of ***p(__ | sits over)*** from which, depending on the sampling strategy, we use to predict the next word in the blank.

As N-gram can be used to predict the next token, a natural application of this model is code completion, wherein given the sequence of code tokens from previous time steps $[token_{t-1} \ldots token_{t-(N-1)}]$ of the user, the model predicts the token at time step t.

II. Methodology

Packages To implement the N-gram model, we use the **nltk** package as it has a predefined "ngram" function which we could use to build the model. We further use the **transformers** package to obtain the tokenizer from CodeLLama for text processing. We use the package **datasets** to obtain clean, ready-to-use dataset in Java language.

Dataset As mentioned above, we use the **datasets** package to load the Java subset of **google/code_x_glue_ct_code_to_text** with 165k rows of training data and 11k rows of testing data. The rows contain Java functions collected and cleaned from Github. Particularly, each row contains the **preprocessed** code tokens, which are sequences of tokens (e.g. *["public", "static", "boolean", "check", "(", "String", "passwd", ",", "String", "hashed",..]*). **The final test set used in this project contains 100 randomly sampled rows of the test set**.

Implementation Our implementation of the N-gram language model is a collection of 4-gram, 3-gram, 2-gram and 1-gram models. Particularly, given an input sequence of code tokens, we input it to 4-gram, and if it does not work, then on 3-gram, and this process repeats all the way up to the 1-gram. In subsequent sections, we will call it the [4,3,2,1]-gram language model.

Split	Num samples
Train	165k
Test	100

Table 1. Dataset statistics. The model was trained on the preprocessed train set of 165k samples and evaluated on 100 samples of the test set. The test set contains both preprocessed data and non-processed data (simple Java code), which we will use CodeLLama to process in our experimental settings.

III. Experimental Setup and Results

Preprocessed Tokens In this experiment, we directly run our [4,3,2,1]-gram language model on the 100 test samples of preprocessed tokens. This serves as the ideal scenario.

CodeLLama Tokens In this experiment, we run our [4,3,2,1]-gram language model on the 100 test samples of the non-processed tokens. This means that they are Java code needed to be tokenized. This serves as the real world scenario where the code is uncleaned.

Setup We train two separate models, [4,3,2,1]-gram$_{\text{Pre}}$ and [4,3,2,1]-gram$_{\text{CodeLLama}}$ which were trained on the **Preprocessed Tokens** and the **CodeLLama Tokens** train set respectively. We then evaluated them on both subsets in the test data.

Results The models' performance were evaluated on accuracy, defined as the total number of exact matches (predicted token == ground truth token) over the total number of tokens. We report the performance of our models in Table 2. Experimental results show that (1) performing inference on data tokenized by the same tokenizer yields better results in both scenarios (0.4552 vs 0.2436 for [4,3,2,1]-gram$_{\text{Pre}}$ and 0.2998 vs 0.1765 for [4,3,2,1]-gram$_{\text{CodeLLama}}$) and (2) Preprocessed Tokens are much better than CodeLLama Tokens as training data as [4,3,2,1]-gram$_{\text{Pre}}$ performs much better than [4,3,2,1]-gram$_{\text{CodeLLama}}$ when evaluated on same-tokenizer (0.4552 vs 0.2998) and different-tokenizer (0.2436 vs 0.1765) scenarios.

Model	Preprocessed Tokens	CodeLLama Tokens
[4,3,2,1]-gram$_{\text{Pre}}$	**0.4552**	0.2436
[4,3,2,1]-gram$_{\text{CodeLLama}}$	0.1765	**0.2998**

Table 2. Experimental Results of [4,3,2,1]-gram$_{Pre}$ and [4,3,2,1]-gram$_{CodeLLama}$ on Preprocessed Tokens and CodeLLama Tokens. We define same-tokenizer results as results obtained from testing [4,3,2,1]-gram$_{Pre}$ on Preprocessed Tokens and [4,3,2,1]-gram$_{CodeLLama}$ on CodeLLama Tokens. The other case is called different-tokenizer.

Free-text Generation We also provide the user with the option to generate free text, wherein the user needs to provide the initial seed code tokens as a starting point for code generation.

IV. Conclusion

In this project, we implemented a [4,3,2,1]-gram language model for code completion, evaluating its performance on both pre-processed and unprocessed Java code tokens (where it will then be processed by CodeLLama tokenizer). Our results show that models trained and tested on the same-tokenizer data achieve higher accuracy, with the preprocessed token model performing better overall than the CodeLLama token model.