Leyli (Aya) Garryyeva
Professor Antonio Mastropaolo

Assignment 1

CSCI 680
September 26, 2024

Recommending code tokens via N-gram models

In this project, we develop and train a probabilistic N-Gram model for code completion on Java source code. The dataset includes 6, 401 Java classes containing less than 100 code tokens.

The objective of this project is to implement an N-gram model that would recommend code tokens. The model learns the probabilities of token sequences and suggests the next token. We expiriment with context sizes ranging from 1 to 5 and evaluate the models using the preplexity score.

Data Pre-Processing

The original dataset was collected using `https://seart-ghs.si.usi.ch` platform and contains 1, 756, 286 instances of Java source code hosted on Github. We filter out the projects that do not have Apache license and do not contain "Apache" in the repository name.

We remove the comments (multi- and single- line), extract the source code containing only Java classes, remove the newline symbol `\n`. Furthermore, we filter out the instances that contain any URLs, unbalanced quotes, and the number of code tokens exceeding 100.

We obtain the training, validation, and test data which contain 4, 608, 1, 152, and 641 Java classes, respectively.

N-Gram Model

The N-Gram model learns patterns in Java code structure and syntax, allowing it to perform a code completions task. Here are its key features:

- **Tokenization**: We convert the Java source code (classes) into tokens using the `javalang` Python library.
- **N-gram size**: We allow n-gram sizes from 1 to 5 for experimentation with different context lengths.
- **Conditional probability**: We use conditional frequency distributions to predict the next token based on the previous n-1 tokens.
- **Vocabulary**: We create a vocabulary from the training data, handling out-of-vocabulary words with an `<UNK>` token. We add starting (`<s>`) and ending ((`</s>`) tokens to signify start and end of code.
- **Data splitting**: We divide the dataset into training, validation, and test sets.
- **Evaluation**: We calculate perplexity on the validation set to measure model performance.
- **Code completion**: We generate code completions based on a given context for random samples.

Results

We observe from the Table 1 that 3-gram model performs the best, while unigram is the worst followed by 5-gram model. However, upon reviewing the code generated by the N-gram models with context windows between 2 and 4, we notice that majority of the predicted tokens are repeated `Separator` token.

Context size	**Perplexity**
Unigram	18.06
Bigram	6.98
3-gram	6.21
4-gram	8.84
5-gram	16.21

Table 1: Perplexity Score by Context Size

Below is the demo output of the 3-gram model with repeated `Separator` tokens:

```
public class <UNK> extends <UNK>
Separator Separator Separator Separator Separator
Separator Separator Separator Separator Separator
```

Below is the demo output of the 5-gram model:

```
public class <UNK> username region
Val metadata hashCode password BigInteger
roleName GemFireAST
```

We observe from the above examples that 5-gram model manages to predict the next token better despite the higher perplexity score.

Analysis

Overall, our N-Gram model completes the class-level Java code, which meets the requirements of this project. Based on the comparison between the context sizes, we observe that the model performs best when the context size is 5. Yet, the preplexity score for this model is lower than other models that generate non-sensical code.
To improve the model performance we could try to identify if the imbalance within the frequencies impact the performance and address the issue. Additionally, we could re-evaluate some of the pre-processing performed during this study and their impact on the scores. Finally, we could expand our data to include more than 100 code tokens and observe if this improves the model performance.