Recommending code tokens via N-gram models

Students: ***Mostafa Ahmed, Fahmida Hossain, and Johora Polin***

Introduction

Our report for HW1 aims to mine GitHub repositories to extract a reasonable number of methods (around 25k). Thus, we can apply some necessary preprocessing before creating our corpus. Then, we implement an n-gram model with Kneser-Ney smoothing and backoff, and thus train it on 70% of the corpus for the sake of recommending the code tokens.

Dataset

We used *Seart* tool (https://seart-ghs.si.usi.ch) and applied some filters such as minimum number of stars and having an MIT license to identify many Java projects on GitHub. The search results were exported as a JSON file containing repository metadata.

A simple Python script `clone.py` was enough to automate the cloning of resulted repositories in JSON file. Since the volume of repositories was big enough, we decided to take only a sample of the projects [18 projects] to meet the requirement of the task, approximately 25,000 methods. Specifically, we extracted 27459 java methods from those projects

Data Preprocessing and Corpus

After deciding the projects, we had to apply some necessary preprocessing before preparing the corpus, and then splitting it into training, validation, and testing. The script, 'parse_split.py', performs the following tasks: traversing the Java files in the cloned repositories, removing comments, extracting the methods from each file using abstract syntax tree parsing (AST), tokenizing the code using `*javalang*` library, shuffling the extracted methods, and finally splitting the corpus into training (70%), validation (10%), and testing (20%) sets.

```
Total methods: 27459
Training data: 19221 methods
Validation data: 2745 methods
Testing data: 5493 methods
(myenv) mostafa@Mostafas-MacBook-Pro Homework 1  %
```

Training N-gram Model

We relied on an N-gram model with backoff (n-gram.py). This allows to count occurrences of N-grams and their contexts in the corpus. Additionally, we used Kneser-Ney smoothing to handle the unseen N-gram. Since results with different window sizes were not promising, we used the backoff, which could deal with the lower-order N-grams when higher-order ones are not available. For the intrinsic evaluation, we calculated the perplexity of the model on the validation and test sets, where our model was generating N-grams of orders from 1 to 6. The model is trained on the prepared corpus with the following parameters:

- Maximum N-gram order: 6 - Smoothing discount factor: 0.75

```
(myenv) mostafa@Mostafas-MacBook-Pro Homework 1  % python3 n-gram.py
Generating 1-grams: 100%|
Generating 2-grams: 100%|
Generating 3-grams: 100%|
Generating 4-grams: 100%|
Generating 5-grams: 100%|
Generating 6-grams: 100%|
callculating perplexity: 100%|
Validation Perplexity: 10.849118189912803
callculating perplexity: 100%|
Test Perplexity: 11.215555312891219
```

The shown perplexity scores indicate that the model has learned to predict Java code tokens with a reasonable accuracy given the fairly large number of java methods. Here is an example for the model generating code completions for a sample context:

Context: Get (playerData . getDimension

Predicted next tokens: [('(', 0.9062520627062707), (')', 0.10776872101821673), ('.', 0.10466053640689096), (';', 0.06141069259477078), (',', 0.03749333567844168)]

For a better result, we may explore increasing the maximum N-gram order, while we are conservative about that as it may overfit and not generalize. Also, fine-tuning the smoothing parameters, or even implementing a more advanced language model will potentially lead to a better performance.