GenAI for Software Development Assignment 1
Krishna Swaminathan

Introduction

For this projec the goal was to predict the next token in Java code. I used n-gram models to do this. This is also very applicable to the real world. There are many progams right now that do the same thing and help with code completion tasks sucj as Github Copilot and Cursor. However instead of predicting entire lines of code the assignment was to predict one token at a time using the n-gram models. This made it a small scale version of what many other programs do.

The way an n-gram model works is by looking at the previous n–1 tokens to guess the next one. For example, a 3-gram model uses the last two tokens to predict the next one. Even though this is a simple compared to neural networks, it works pretty well for small-scale experiments and helps us understand how token probabilities in code can be learned from examples.

For this assignment, I used the Google Guava repository and took out Java files. I extracted all the methods from the Java files. I then cleaned them up. I removed duplicates, short methods, and non-ASCII characters. This is to make the data more uniform and more logical to train with. Then, I split the dataset into training, validation, and test sets so I could train the model and see how well it works on data it hasn't seen before.

To check how good the models are, I used perplexity. Perplexity is used to measure how out of the norm it is for the model when it sees the actual next token. A lower perplexity means the model is better at predicting the tokens. I tried different n-gram sizes (3, 5, 7) and compared their validation perplexities to pick the best one. This is what I did to find the best one.

Finally, I tested the chosen model on both the professor's test set and a self-created test set. I also saved predictions to JSON files so it would be organized and easy to share. This project helped me understand how n-gram models work on code, why smoothing is important, and how dataset size and context length affect performance.

Data collecting and Preprocessing

For this project, I needed a bunch of Java methods to train the n-gram model. I decided to use the Google Guava repository from GitHub because I saw it is one of the more popular Java libraries and has a lot of code to work with. I downloaded the repository as a ZIP file and then unzipped it into my project's data folder. This gave me a total of 3243 Java files to work with.

After that, I extracted all the Java files from the folder. Then I wrote a script to pull out all the method-level code, because I needed to use the individual methods, not entire classes or files. This gave me 42684 methods to work with.

Once I had all the methods, I cleaned them up. I removed methods that were too short, having less than 10 tokends, removed duplicates, and got rid of any non-ASCII characters. All of these cleaning criteria were pretty standard, and nothing was particularly abnormal. After cleaning, I ended up with about 19,853 methods.

Then I split the dataset into different sets for training, validation, and testing. I made three different training sets (T1, T2, T3) with different sizes to see how the model performed with more data. T1 I made have 15,000 methods, T2 had 25,000 methods, and T3 had 35,000 methods. The validation set had 1,000 methods, which I used to check which n-gram size worked best. I also had two test sets the one I got from the professor and also the test set I created. I made the test sets from the same cleaned Guava methods. I did ensure that these methods were separate than the ones used for training.

Finally, I tokenized all the methods into space-separated tokens. That means keywords, identifiers, literals, operators, and punctuation symbols like parentheses, periods, and brackets all became their own tokens. This was important so the model could learn the patterns properly and they were differentiated effectively.

I also built a separate vocabulary for each training set. Tokens that weren't in the training vocabulary got replaced with a set value in the validation and test sets. This is so that the model can handle unknown tokens without having any unexpected behavior or breaking any of the other code.

Training and Collection Data

After cleaning and splitting the dataset, I trained three n-gram models with n = 3, 5, and 7 on the first training set (T1) and evaluated them using the validation set. The results showed that the 3 gram model had the lowest validation perplexity of 167.34 out of all the remaining options. The 5-gram and 7-gram models had much higher perplexities of 984.46 and 2547.65, respectively, therefore making the T1 the best case. These results shows me that using a larger context does not necessarily improve predictions. In this case, the longer n-grams suffered from data sparsity, meaning there were not enough repeated sequences of 5 or 7 tokens in the training set for the model to more reliably estimate probabilities.

I then trained the models on the larger training sets T2 and T3. On T2, the validation perplexity of the 3-gram model slightly increased to 187.38, while the 5-gram and 7-gram models remained higher at 1144.18 and 3028.58. One thing that I did notice is that the T3 results were almost identical to T2, which indicates that adding more training examples beyond a certain point did not significantly effect the performance for the n-gram sizes tested. Based on these results, I selected the 3-gram model trained on T1 as the final model because it consistently had the lowest validation perplexity and was simpler to compute.

Next, I evaluated the final 3-gram model on two test sets. First on the given test set, the model achieved a perplexity of 3438.06, which is a lot higher than the validation perplexity. This large

number is likely due to differences between the Guava training data and the professor's test set, which must contain different coding styles, unique identifiers, or method structures not present in the training data. Despite this, the model performed much better on the self-created test set, achieving a perplexity of 171.75. This tells me there must be a differen style of code or something between the Guava and the teacher's code. The self-test set was drawn from the same cleaned Guava methods as the training set, so it was more similar in structure and vocabulary, which explains the much lower perplexity.

These results do show an important potential observation about n-gram models. They rely seem to heavily on observing repeated patterns in the training data. While they can predict well on similar code, their performance drops quickly when encountering sequences from a different domain or project. This is my takeaway from this specific example, though many tests would be needed to prove this. Nonetheless, the final model successfully demonstrates the assignment requirements, including token-level predictions, handling of out-of-vocabulary tokens, smoothing with add-α, and generation of prediction outputs in JSON format.

Overall, the experiments show that smaller n-grams like like the 3 gram model strike a balance between capturing local context and avoiding sparsity. The JSON outputs also provide a useful way to inspect individual predictions and verify that the model assigns reasonable probabilities to the ground truth tokens.

Conclusion

For this project, I was able to implement n-gram the models to predict the next token in Java methods, using method-level code from the Google Guava repository. The main goal was to explore how different n-gram sizes can affect prediction accuracy and to evaluate model performance using perplexity.

Through working with these models, it became clear that smaller n-grams, like the 3-gram model, performed best on the validation set. Larger n-grams, such as 5-grams and 7-grams, seemed to struggle with data sparsity, which caused much higher perplexity values. This indicates that while longer contexts can theoretically capture more structure, they probably require much more training data to be truly more effective.

The final 3-gram model was evaluated on both the professor's test set and a self-created test set. The model achieved perplexity of 3438 on the professor test set and 171.75 on the self-test set, this shows the difference between familiar and unfamiliar code. This shows that n-gram models can handle code that is similar to the training data well, but generalizing to different projects may not work as well.

Overall, this project provided a cool experience with building a ngram model for code, handling unknown tokens, applying smoothing, and generating structured predictions. While I know n-gram models are simple compared to a lot of the programs that are used today, they are important for understanding the basics of code modeling, probability estimation, and evaluation

metrics like perplexity. I learned a lot here that can be extended to more advanced models in the future, such as neural language models or transformer based code completion systems.

Overview

Used the Google Guava repository as the source of Java code. The repository contained 3,243 Java files, from which we extracted 32,464 methods. After cleaning I 19,853 methods for the dataset.
The dataset was then split as follows:
Validation set: 1,000 methods
Self-created test set: 1,000 methods
Training pool: 17,853 methods
T1: ≤ 15,000 methods
T2: ≤ 25,000 methods
T3: ≤ 35,000 methods

Training 3-gram model on T1
Validation Perplexity (T1, n=3): 167.34165074699644
Training 5-gram model on T1
Validation Perplexity (T1, n=5): 984.4556975600761
Training 7-gram model on T1
Validation Perplexity (T1, n=7): 2547.648797499736
Training 3-gram model on T2
Validation Perplexity (T2, n=3): 187.38464188361172
Training 5-gram model on T2
Validation Perplexity (T2, n=5): 1144.175276203456
Training 7-gram model on T2
Validation Perplexity (T2, n=7): 3028.5762092251325
Training 3-gram model on T3
Validation Perplexity (T3, n=3): 187.38464188361172
Training 5-gram model on T3
Validation Perplexity (T3, n=5): 1144.175276203456
Training 7-gram model on T3
Validation Perplexity (T3, n=7): 3028.5762092251325

Validation Perplexity (n=3): 15.349659348496818
Validation Perplexity (n=5): 15.380650527624233
Validation Perplexity (n=7): 15.382551518659808

Validation Perplexity: 167.34165074699644
Professor Test Perplexity: 3438.0598822850197
Self Test Perplexity: 171.75280942171617