**Assignment 1: Recommending code tokens via N-gram models**

This was quite the ordeal but a fun challenge. Thank you for the extension again, I really appreciate the chance. I still believe I could do better, but I won’t ask for another extension after the incredible generosity you have already shown me. I only hope this is along the lines of what you were looking for, even if it doesn’t perform to a high level.

For the data collection process this was a rather straight forward thing for me initially. Given the model is eventually supposed to predict the next java token, the only real java external tokens would be user created names for variables and any libraries/plugins added and even those are generally Java code, I decided the best approach would be to use the Java JDK. Tracking down a clean copy of the source code took a little while surprisingly, but eventually I decided on version 22.0.2 from the archived versions of Java: https://jdk.java.net/archive/. This was a lot of code that had a lot of people check it so I felt confident this would be functioning, consistent java code that would make the process of learning how to tokenize java files easier. Once I had the tar unzipped, I did a search of the folder for all the .java files and moved them into a local corpus folder.

With all the java files in one place, I now needed a way to scan through all of them and tokenize them. I looked into a few options online, things like JavaParser, but these proved to either be too complicated or generated undesirable results. I already knew some basic python so I knew how to make a script that could loop through and open files in a directory, so this part wasn’t too complicated. The function to do this is process\_directory. Originally all it would do is scan through the directory and then print out the title of the file and the first line for every file. Once I knew I had that working, I needed to figure out what tokenizing meant exactly and how to do it.

Conceptually it was easy, take the line in as a string and split it on delimiters. I started by taking a sample line of java code to figure out how to tokenize a line. I began by splitting on empty spaces, simple to do but once the logic is in place it was just a matter of adding more delimiters. This function is: tokenize\_line. To accomplish this I used a regular expression, something I have become familiar with due to my use of javascript in my current job. Originally it was an array of strings that each contained the various different delimiters that I would use by taking a “token” from the empty space split array and combing through the token character by character until it encountered one of these delimiters. This worked somewhat, but was a time consuming process and missed quite a few. Looking for an alternative way I discovered this stack overflow page: <https://stackoverflow.com/questions/4998629/split-string-with-multiple-delimiters-in-python> that outlined Python already had a means to do this. I investigated what the import re was and what all I could do and finally settled on what you see now. The re.compile is where I load in all the many characters I want to split out and the for loop finds them and appends them to the separated\_tokens array. This worked for most of the code, however, I noticed a lot of things were getting through. There is still some things in the training data that is debatable, but the big one was all code inside of double quoted strings were not being separated at all. To fix this I made a second loop through the separated\_tokens array to revaluate all the tokens. I chose this method as opposed to modifying the existing for loop as I anticipated more things needing to be filtered that might be more involved. With this we now had the ability to tokenize a file line by line. With these 2 pieces, I then needed to put the two methods together to loop through the file line by line and tokenize it. I modified process\_directory to, after opening the file, loop through the current file and send every line to tokenize\_line and return the array to be joined together, but I quickly hit the issue of comments. To solve this I split the line by line loop out of process\_directory to make a new function to remove comments and then submit non comment lines to the tokenize function. This function is: remove\_comments\_and\_tokenize. This was a simple function in concept, loop through the file lien by line, scan for comment characters and either remove them, or keep track of multi line comments so none of it is mistakenly sent to be tokenized. To test, I took an old java file from my undergraduate program that I had thoroughly commented and tested it with that.It worked, so I put all the pieces together and it worked. With this now working, it was time to get a proper corpus to fit the assignment. I used the <https://seart-ghs.si.usi.ch> to find git repositories with a lot of java files to meet the 25,000 methods and 5,000 classes requirements. After combing through ~50 repositories I selected a handful that ended up being over 30,000 classes and 100,000 methods. The more data the better, however, given this was written by a lot of different people there would be inconsistencies and potential errors.

When I attempted to run the process\_directory on this new set of files, I hit an error: “UnicodeDecodeError: 'utf8' codec can't decode byte 0xa5 in position 0: invalid start byte**”.** Some of the files were of a different format so after googling what it meant and how to fix it, I added the try/catch block to catch this and encode it as ISO-8859-1. With this, all the files were tokenized and pasted into a output txt file that would be my corpus for the ngram model.

The actual model was the tricky part and has been quite the exciting challenge. I decided then n number for the ngram should come from the user, so I added in the main function the ability to capture the argument from the user. With this, I had the tokenized output and n number to create the model in the function: build\_ngram\_model. As I understood it, I needed to make tuples of every n number of tokens so I would be able to quickly look up combinations to get probabilities. I googled for how to make tuples and discovered defaultdict: <https://docs.python.org/3/library/collections.html#collections.defaultdict>. This would let me make a dictionary of all the tuples I make, so this part was really easy and straightforward.

With the function to build the model, I just now needed to use it and make predictions. I took a line out of the tokenized output as my sample and set to work creating the predict\_next\_tokens function to loop through a line of code and make predictions. This was pretty simple, run the line through the tokenize\_line function and then take n-1 tokens, tuple them, then run it through the model to get all possible options. With all options, calculate probability of each by using the counter function, that was listed in the same link as defaultdict, I could create a collection of all the tuples and then use the most\_common method to get the top 3 probable options to list as recommendations. The issue was, and still is, this method evaluates the line as it was when it was passed in. This means any inserted code is not being evaluated, however, when attempting to evaluate the inserted code I regularly got stuck in infinite loops of it adding infinite code into lines. I struggled with this for a while, but eventually decided I would stick with evaluating the initial line. With this figured out, I now needed to use a sample file with 100 lines. This is where things really became apparent how flawed the logic is. The issues I identified were things not in the corpus, anything shorter than n are left unmodified leaving code that should be modified, and strange recommendations. For the stuff not in the corpus is where you would use some kind of smoothing technique, I set this aside as a thing to do last since it was’t strictly necessary to get this working, but I was already struggling for time to get this working at all so this never came to be. For things shorter than n, I left as I see that as the consequence of using this type of model/that number as n. And for the strange suggestions, this took a lot of tweaking to understand and smooth out some bizarre injections, but ultimately it is working as intended. The odd things can be traced back to the corpus and calculated manually, it is doing what it’s supposed to, it just isn’t very helpful. I spent the last few days trying to find way to smooth this part out so it would be more useful, but I’m out of time and didn’t ultimately get it better than it is now.