Michael Ing September 22, 2020

CS 2731 – Hwk 1

**Program Breakdown**

The program is contained primarily in one file, “NGram.java”, with a designated executable file, “NGramRunner.java”. I wrote this program with a big NGram class that has a subclass (NGramNode) where the bigram and trigram counts are stored. Each NGram has an array of nodes, with each node corresponding to a single character (or unigram). I used characters based on the ASCII table from 32 (space) to 255, to include all possible typed characters. Each NGram also keeps track of the number of active nodes (dictionarySize), the total number of characters (numChars), the configuration (mode), k and lambda values for different configurations, and a devset (represented by an array of ints). At first, I denoted uppercase and lowercase letters as different symbols, but after some testing found that with limited data it made more sense to convert all letters to uppercase, for better accuracy and prediction. I also replaced all new line characters with space characters and ignored starting and ending stream characters as there were too few to be of any use.

Other than the constructor and some basic getter methods, most of the NGram functions are private. The main interaction from the runner program runs through addSequence(), configure(), and getPerplexity(), getNextLetter(), and printToFile(). The configure() method simply changes the mode (which in turn affects how probabilities are calculated) and runs tuning functions as necessary. The getPerplexity() method calculates perplexity values for each n-gram (based on the assigned mode) and returns an array of doubles with the three different perplexity values. getNextLetter() and toString() are used for printing output, with getNextLetter() finding the highest probability for the next character and returning it, and printToFile() converting the model with unigram, bigram, and trigram probabilities into an easily printable string that is then printed to the appropriate file. AddSequence() directly interacts with the NGramNode class, which is described below.

The addSequence() method takes three integer parameters, c1, c2, and c3. It then checks if the node corresponding to c1 exists, i.e. if that character has been seen before. If not this new node is added, and entered. Inside this node, there are two arrays, one for bigram counts and one for trigram counts. The corresponding bigram and trigram counts for (c1, c2) and (c1, c2, c3) respectively are then incremented, and a unigram count is also tracked with appropriate getters. During the entire method numChars is incremented only once, so the user must read the file character by character, e.g. call addSequence(c1, c2, c3), then addSequence(c2, c3, c4), then addSequence(c3, c4, c5), and so on until the end of the file is reached.

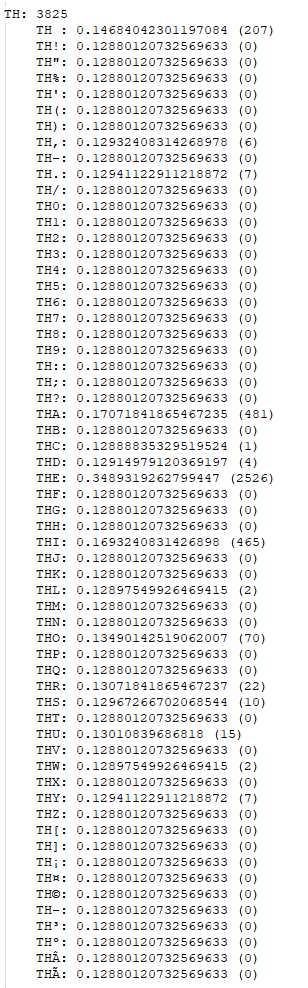
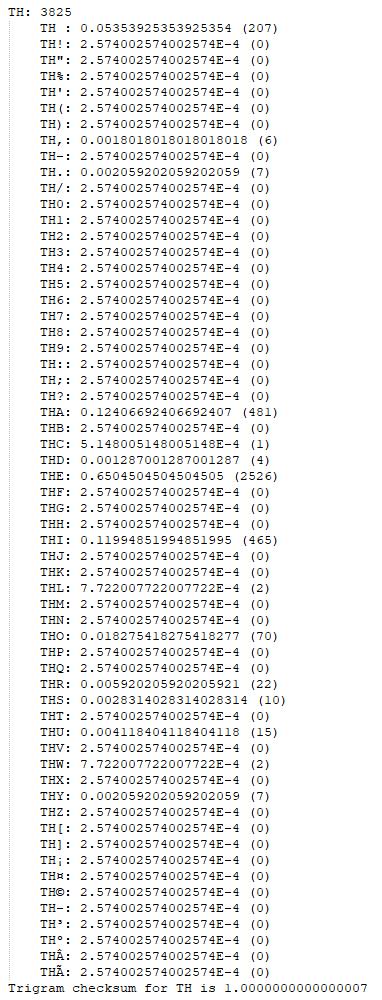
The devset is handled via the addToDevset() method, which takes one integer parameter. In the runner class, a random devset starting point is generated, with the devset taking the size of 5% of the total file size (this value was tuned according to generated accuracies). Once this devset window is generated, the entire file is read, and based on the configuration and the file location, either addToDevset() or addSequence() is called, which populates the devset when needed. A boolean value based on the mode is used to determine if the devset is needed or not, which allows this devset step to be ignored completely.

The same devset is used to generate both the tuned k values and the tuned lambda values. This tuning is done by starting at some set value (0.25-0.75 for k, 0.5 for bigram lambdas, and 0.33 for trigram lambdas). Then, over a set of tuning iterations, probabilities are compared for the current tuned values, the tuned values stepped up, and the tuned values stepped down. This is repeated with smaller and smaller step sizes for the set number of iterations (6 iterations was deemed appropriate and efficient) to allow the variables to settle on some ideal value. For trigram lambdas, this tuning occurred twice, once with the first variable, and then again with a second variable, to allow all three variables to be tuned.

The NGramRunner file has a few static methods associated with each task, including readFile(), writeModel(), readTestFile(), and generateText(). While mostly self-explanatory, readFile() reads in the file and builds the NGram model based on the desired configuration, writeModel() writes the NGram data to a text file in the “outputs” folder, readTestFile() reads the test file and returns the appropriate perplexities (which are stored in a 2-d array for each mode), and generateText() takes one or two characters as parameters and returns the generated sentence using the NGram model. There is a keyboard interface included in the runner program that allows for user input and interaction, which includes repeated text generation and retraining of the model, as well as perplexity tracking. It is also possible to input text generation characters from the command line, by typing one or two characters.

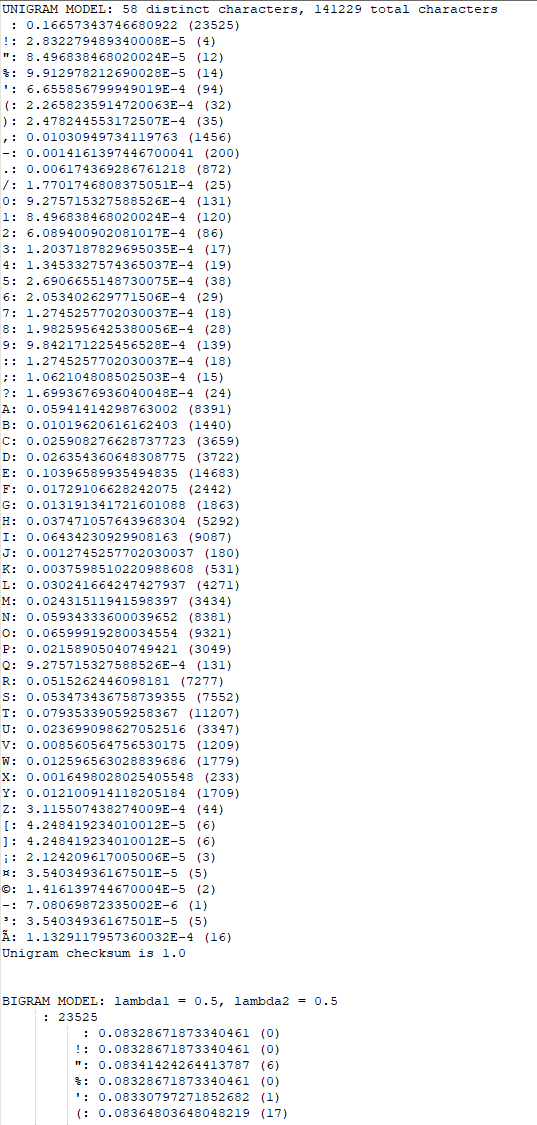
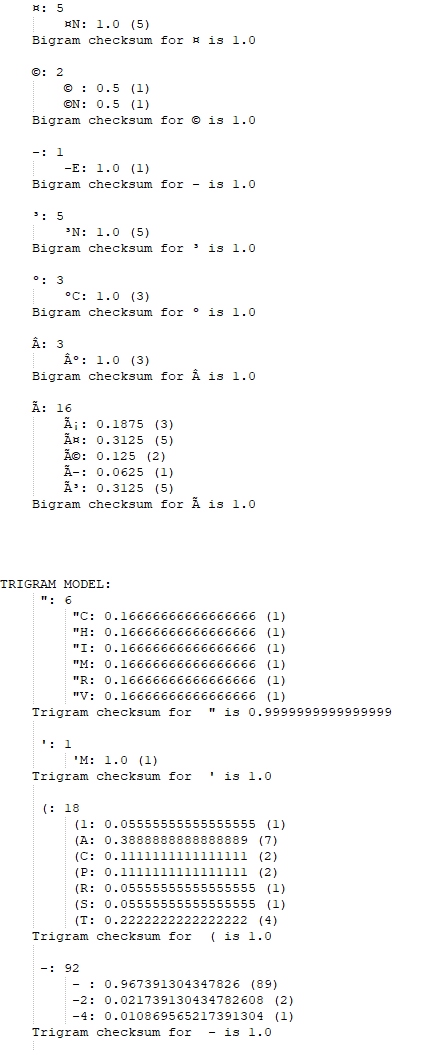
**TH Character History**

Below is an excerpt of my English trigram models for “th”, with add-1 on the left and linear interpolation on the right:

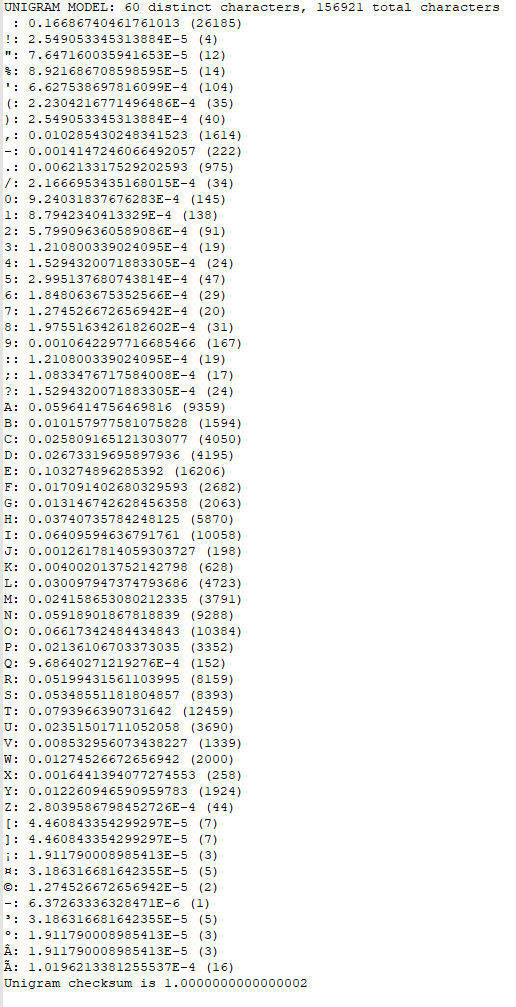
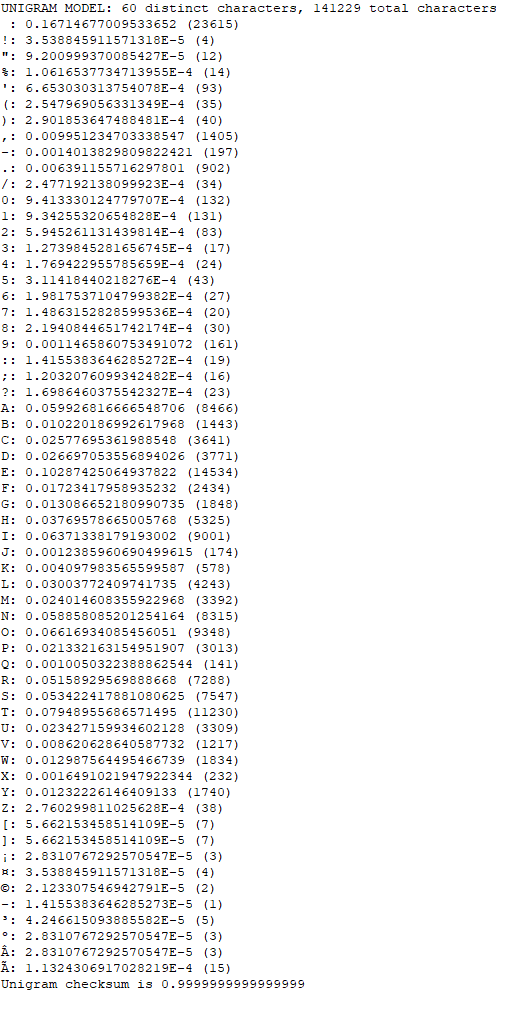


**Validity of Probability Distributions**

For proof that my probability distributions are valid, the printToFile() method has checksums that are printed in output files (one for each unigram, bigram, and trigram). Except for linear interpolation, all of these checksums successfully add to 1.0, as can be seen by the screenshot below. For linear interpolation, lambdas are used (and printed in output files) that always add to 1.0 as well.



**No-Overlap Devset Tuning**

Devset validation can be seen by directly comparing the unigrams of a mode that does use a devset and one that does not use a devset. As can be seen below, looking at “normal” mode vs. “add-k” mode, the unigram counts (as well as total character counts) are different for the same “training.en” document, which shows that the data used for the devset is taken from the training set and not the test set:

**Perplexity Scores**

For the sake of space, all perplexity scores are listed in appropriate text files under “outputs/PP\_MODE.txt”, where MODE is the corresponding mode. Perplexity scores per sentence for the English model and average perplexity scores are all listed in those text files.

**Generated Text Output**

Text generation output can also be found in the outputs folder under “outputs/gentext\_MODE.txt”, where MODE is the corresponding configuration. Bigram and trigram text generations for all 26 letters can be found in each text file for the English model.

**Analysis of Language Identification Results**

The perplexity score is inversely related to probability, and is a combination based on the existing n-gram model of likelihood that the given word, sentence or line could occur. Perplexity is normalized by the number of words (or characters) by taking the nth root of the resulting probability, which makes the value more readable. It translates to language identification because if strings of characters are present in a language more often, the inverse probability of a given sequence of characters will be lower (probability of seeing this string is higher, and thus inverse is lower). When extended over multiple sentences in a document and averaged, lower aggregate perplexity scores indicate that a group of sentences is more likely to occur and is therefore more likely written in the corresponding language.

The perplexity scores for my unsmoothed “normal” mode model are extremely high to the point of being incalculable by the model, compared to my smoothed “add-k” and “tuned-lambda” models, which have measurable and reasonable amounts. This tells me that tuning is important for allowing for different possibilities, that an unsmoothed model would otherwise completely ignore. There is surprisingly little difference in perplexity scores between tuned and untuned models however, which tells me that this tuning process was perhaps not as effective as expected.

Comparing the unigram, bigram and trigram perplexity scores show drastic differences across models. For the “normal” mode, only unigram perplexities are useful, which makes sense since many character strings are unaccounted for and would therefore be unhelpful in finding probabilities. The “add-1” and “add-k” modes are consistent with one another, showing marked improvement from unigram to bigram and then again from bigram to trigram, although less so. This indicates that increased complexity (when smoothed) improves the results, providing for more likelihoods and thus proving to be a better indicator. Finally, the “equal lambda” mode also follows this trend, but the “tuned lambda” mode has an incalculable perplexity for trigram perplexity, which seems to point to an issue with the tuning process. With a proper tuning method, it seems that this mode would also follow suit with the other smoothed methods. Overall, this analysis demonstrates that smoothing is the most important aspect of getting reasonable perplexity, and that linear interpolation has a small but mostly negligible advantage over add-k smoothing.

**Analysis of Generation Results**

The language generation by the different English models showed limited success, and mostly a lot of “THE ” strings. This makes sense for bigrams, as the “T” character was most common after a space, and thus as soon as a space occurred “THE” would just repeat for the remainder of the sentence. The trigram model showed only limited improvement, generating a few more sensical words but ultimately returning to the “THE” pattern. This is likely since “THE” is the most common word in the English language, so it makes sense that it would be repeated once found once. It is likely that a higher n-gram model would have more variation and potentially result in some real English being generated, due to the increased variability in text strings found.

There was little variation (in bigrams and trigrams) in text generation between models, mostly because of the method for finding the next character. Since it was entirely based on the most likely character to appear next, this rarely changed between unsmoothed and smoothed models in both bigrams and trigrams, resulting in almost identical text being generated. Even with variation of characters, bigram generators never got past a single word before entering the “THE” pattern, while trigram generators never got past three words. Nevertheless, I am optimistic that the use of higher n-gram models would give exponentially better results, and thus some real English text.

A better solution would involve having a few text generation choices (ideally among the top 3-5 most likely characters) and then randomly choosing one, or even randomly choosing from all possible characters based on their probability. This would introduce the randomness necessary to generate more probable English text, and not fall into a pattern of repeating characters continuously.