Colin Swan

CS1671 Homework 2

February 21st, 2016

Implementing Ngram

**Part I and Part II– Building N-Gram**

This assignment began with the basic implementation of an n-gram model. I chose to use Python as its batteries-included approach seemed like it would be beneficial to this type of project, and because some of its features – such as list comprehensions – seemed suited to the sort of tasks that need to be performed for NLP. I started by creating a generic ngram method that would handle any level of ngram. But when it came time to implementing smoothing it started to become more difficult to keep the methods generic. As a result the script is not guaranteed to work for ngrams greater than 3, though if it does not use smoothing it should work relatively as expected.

For unknown words and infrequent words I start by creating a unigram of the input, and then I replace any words in the input that have a frequency less than or equal to 1.

The program does interpolation while computing per word perplexity. The weights are chosen using the dev file; for bigram models, the weights for unigram and bigram both start at .5 each. The script then runs perplexity calculations 100 times, adjusting the weights of each model by .01 until an equilibrium is reached. A similar approach is take for trigrams; each model starts at .34, and each model is given 33 iterations to try to increase its weight. If entropy increases as a result of the adjustment the iteration for that model is canceled and the iteration for the next model begins. This seems naïve and inefficient, but it seemed to work relatively well and consistently, and was better than hard coding the weights in. I worry this approach may be fragile however, as it seems as though this could end up with one or the other models taking over in certain situations. I observed this in one or two cases in the toy problem where the weights favored either Bigram or Unigram entirely, but it only occurred for two of the cases, and the other weights seemed reasonable, so I left it as it does not seem entirely unreasonable to ignore a model if it is detrimental to perplexity calculations.

**Part III** – **Minor adjustments for larger files**

Part III required only a few changes to get basic functionality working. For example, certain symbols are parsed away in the files so that they do not cause errors with regular expressions, which are used when handling <unk> words. The format for the output was also modified to correspond with the expected output for the accuracy testing script provided in the assignment.

**Results, Concerns, and Conclusions**

There are a number of concerns I have with the script in its current state. I feel it is likely very fragile, as I did not do thorough work on pre-processing any of the inputs. Much of the pre-processing was done to get basic functionality working; the methods for pre-processing are neither efficient nor reliable. I also fear that the adjustments for weights when dealing with smoothed ngrams is inadequate; it seems likely that the weights could reach an equilibrium at a point that is not necessarily optimal in terms of providing best possible entropy results, or that the weights could optimize for an entropy that does not accurately predict the likelihood of a sequence of words.

As far as training with large data sets, the biggest problem for the script seems to be with files over a megabyte in size. A training file around 1MB in size usually takes around 5 minutes for the script to complete. This seems very slow, but is at least workable. Using a training set larger than that quickly results in unfeasible run times, with a 3MB file take more than 10 minutes to run (the script has never been completed with a file over 3MB, that is just how long it was allowed to run).

Another problem that is likely to occur in some files is parsing of special characters, as previously mentioned. Pre-processing was kept to a minimum, so many characters may not be accounted for. This could lead to misrepresented frequencies of words and word pairs, which would affect the resulting ngram models.

When it came to testing actual performance of the ngrams, the results were not what I expected. They are as follows, using WHTCO10.txt (The White Company by Arthur Conan Doyle) as a training set and Holmes.machine\_format.questions.txt as the test set:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Unigram** | **Bigram** | **Bigram Smoothed** | **Trigram** | **Trigram Smoothed** |
| Overall Average % | 18% | 22.12% | 17.12% | 22.12% | 17.02% |
| Dev % | 16% | 19.42% | 18.08% | 19.81% | 17.50% |
| Test % | 20% | 24.81% | 16.15% | 24.42% | 16.54% |
| #Correct (of 1040) | 190 | 230 | 178 | 230 | 177 |

These results are one of the reasons for my concerns with pre-processing and weighting. It seems to me that the smoothed models should perform better than their unsmoothed counterparts. It also is strange that the Trigram model is almost exactly the same as the Bigram model in terms of performance. I’m fairly sure the results are caused by pre-processing and weighting issues.

I think one way of improving this program would be by using a larger data set, as it seemed that many words in the tests were being set as “<unk>” despite using a large training set. This would certainly require better optimization, but at some point it would probably also require other tools, such as distributed computing services. At some point there’s just going to be too much data to be able to quickly access it this way, or at least that appears to be the case based on the difficulties I’ve encountered with “large” files. That said, it seems that ngrams will always have trouble with this type of task as they do not take in to consideration the context of words in a sentence. If the program could consider the possibility of a word given certain sequences of words *elsewhere* in the sentence, and not just around the word itself, then it would probably be better able to predict the likelihood of a word appearing in a certain position in a sentence.