Task 1 – Vanilla Language Modelling Report

By Valerija Holomjova

The models take approximately 27s to load for a 30k word corpus. To run the code, run the runner.py python file in any way preferred such as on any IDE (I used PyCharm) or Linux terminal.

Storage

The N-gram models and their frequencies are stored as global arrays. They were made global so that each function could have access to them. I was considering storing the N-Grams as an array of dictionaries or objects where each object would have the actual word, grammar type and frequency, but I have decided to use arrays to reduce run time. The data stored based on the words in the N-Gram arrays and the N-gram frequency array are parallel – meaning the position of a word in the N-Gram array will be identical to its frequency position in the N-gram frequency array.

```
# ------- Global Variables -------
word_count = 0 #Stores the count of words from the training data corpus
#Stores the unigram,bigram and trigram and their probabilities.
unigram, bigram, trigram = [], [], []
unigram_freq, bigram_freq, trigram_freq = [], [], []
```

I have also made a global integer that will store the number of words in the training set, so it can be used to calculate the probabilities of the N-grams which consists only of words from the training set. The rest of the data such as the N-Gram probabilities is obtained through functions.

Reading The File/Storing The Sentences

This part of the assignment is completed in the 'main()' function. For this program I have only chosen one file from the **BNC corpus**[1] ('A6U.xml'), so that the run time of the program wouldn't be as slow since the file has around 30k words. Otherwise, I would try to iterate through a folder of files and merge the sentences of each file together. This program can be used for any other files in the BNC corpus.

The program works by extracting the sentence objects <s></s> from the XML file which are then iterated through so the actual words could be obtained. The 'getWords()' function was created in order to loop through each of the DOM sentence objects and return arrays of words. It was made recursive since the XML file had occurrences of tags such as <hi></hi> (indicating a highlighted part of a sentence), which would cause errors since it would have the word elements within them. Hence the recursive function loops through such tags and obtains the words contained within them and merges it with the outer sentence so that one entire sentence can be returned.

An example of the I/O of the 'getWords()' function is shown in the table below.

Input (Sentence Object)	Output (Array)
<s><w>This</w><hi><w>Is</w><w>An</w></hi><w>Example</w></s>	["This", "Is", "An", "Example"]

The output of each DOM object sentence is appended to an array called "final_sentences". This array is then shuffled and 80% of the sentences is stored for training (in an array called

[1] http://ota.ox.ac.uk/desc/2553

"training_sentences") and the remaining 20% is stored for future use (in an array called "evaluation_sentences". The training sentences are then used to build the N-Gram models.

Building the N-Gram Models

The N-Gram models were built in the 'buildNGrams()' function. It works by looping through each word in the training sentences and building the N-Gram models simultaneously:

1. First, it adds the word to a unigram array or increment its frequency (in the unigram frequency array) if it already exists.
2. Next, it checks if the word has another word after it. If this is the case, it either adds both words (as an array) to the bigram array or increments the frequency of the bigram word (in the bigram frequency array) if it already exists. If it has no word after it, it continues.
3. Finally, it checks if two words exists after the word. If this is the case, it adds all three words (as an array) to the trigram array of increments the frequency of the trigram word (in the trigram frequency array) if it already exists. If it doesn't have two words after it, it continues.

This process is repeated for each word. The number of words in the unigram is stored in the global word count variable which will be used to calculate the probabilities of individual words.

Calculation of the N-Gram Models

The probabilities of the N-Grams were calculated in the 'calculateProb()' function. The function works by looping through each of the N-Gram arrays and calculating the probability of the word or words for each N-Gram using an appropriate formula.

- For the unigram, the probability of each word is calculated by dividing the frequency of the word by the word count.
- For the bigram, the probability of the words is calculated by dividing the frequency of each bigram word by the frequency of the first word in the bigram word.
- For the trigram, the probability of the words is calculated by diving the frequency of each trigram word by the frequency of the first two words in the trigram word.

The N-Gram probabilities are in parallel to the N-Grams, so the unigram probability array will include the probabilities for each of the words in the unigram array.

Testing

The program was frequently during implementation using the dummy data and corpus data. A function called 'printTest()' was left to demonstrate the probabilities of one of the N-Grams – the trigram. This function will print out the probability of each and is solely made for demonstration and testing purposes. Below is a screenshot of the output from the 'printTest()' function.

```
P(and |Latin America )=0.1111111111111111
P(Europe |America and )=1.0
P(is |and Europe )=1.0
P(not |Europe is )=1.0
P(only |is not )=0.034482758620689655
P(one |not only )=0.14285714285714285
P(of |only one )=0.5
P(cultural |one of )=0.07142857142857142
P(of |cultural domination )=0.5
P(former |of the )=0.003257328990228013
P(by |the former )=0.2
```