TCSS 456 Assignment 1

NOTE: Be sure to adhere to the University's **Policy on Academic Integrity** as discussed in class. Programming assignments are to be written individually and submitted programs must be the result of your own efforts. Any suspicion of academic integrity violation will be dealt with accordingly

Assignment Details:

For this assignment, we will develop n-gram models using given set of documents.

Step 1: Create a Unigram Model

A ***unigram model*** of English consists of a single probability distribution ***P(W)*** over the set of all words.

1. **Creating the word dictionary** [Coding only: save code as `problem1.py` or `problem1.java`]

   The first step in building an n-gram model is to create a dictionary that maps words to java map or python dictionary (which we'll use to access the elements corresponding to that word in a vector or matrix of counts or probabilities). You'll create this dictionary from the given data files (Select 80% of the sentences for training) for all unique words. You'll need to split the sentences into a list of words and convert each word to lowercase, before storing it the dictionary.

2. **Building an MLE unigram model** [Update `problem1.py` or `problem1.java`]

   Now you'll build a simple MLE unigram model. For each of the word in your dictionary assign a counter and initialize it to zero. Iterate through the sentences and increment counts for each of the words they contain.
   Finally, to normalize your counts vector and create probabilities, you can simply divide the word count by sum of all counts like so:

   ```
   Prob(word) = Count(word) / Count(all words)
   ```

   Write your new probabilities to a file called `unigram_probs.txt`.

Step 2: Create a Bigram Model

A ***bigram model*** of English consists of two probability distributions: $P(W_0)$ and $P(W_i \mid W_{i-1})$. The first distribution is just the probability of the first word in a document. The second distribution is the probability of seeing word W_i given that the previous word was W_{i-1}

3. **Building an MLE bigram model** [Coding only: save code as `problem2.py` or `problem2.java`]
   Now, you'll create an MLE bigram model, in much the same way as you created an MLE unigram model (use the same 80% training sentences as Unigram MLE). I recommend writing the code again from scratch, however (except for the code initializing the mapping dictionary), so that you can test things as you go. The main differences between coding an MLE bigram model and a unigram model are:
   - Select an appropriate data structure to store bigrams.
   - You'll increment counts for a combination of word and previous word. This means you'll need to keep track of what the previous word was.
   - You will compute the probability of the current word based on the previous word count.

   $$\text{Prob}(\text{word}_i \mid \text{word}_{i-1}) = \text{Count}(\text{word}_{i-1}\ \text{word}_i) \ / \ \text{Count}(\text{word}_{i-1})$$

   Consider we observed the following word sequences:

   ```
   finger remarked
   finger on
   finger on
   finger in
   finger .
   ```

   Notice that "finger on " was observed twice. Also notice that the period is treated as a separate word. Given the information in this data structure, we can compute the probability $p(on \mid finger)$ as 2/5 = 0.4. Similarly, we can compute the probability $p(. \mid finger)$ as 1/5 = 0.2.

   When complete, add code to write all bigram probabilities to `bigram_probs.txt`, one per line
   - $p(\text{on} \mid \text{finger}) = 0.4$
   - $p(. \mid \text{finger}) = 0.2$

4. **Add-δ smoothing the bigram model** [Coding and written answer: save code as `problem3.py` or `problem3.java`]
   This time, copy **`problem2`** to **`problem3`**. We'll just be making a very small modification to the program to add smoothing. In class, we discussed smoothing in detail:

add-δ smoothing, in which some amount δ is added to every bigram count.

You should modify your problem to use add-δ smoothing with δ=0.1, i.e., pretending that we saw an extra one-tenth of an instance of each bigram.

```
counts += 0.1
```

Now change the program to write the same bigram probabilities as before (i.e from file `bigram_probs.txt`) to a file called `smooth_probs.txt`.

Step 3: Using n-grams model

5. **Calculating sentence probabilities**

   For this problem, you will use each of the three models you've constructed in problems 1–3 to evaluate the probability of the other data file for the all test sentences (remaining 20% sentences).

   First, you'll edit `problem1`, and add code at the bottom of the script to iterate through each sentence in the test sentence and calculate the joint probability of all the words in the sentence under the unigram model. Then write the probability of each sentence to a file `unigram_eval.txt`, formatted to have one probability for each line of the output file. To do this, you'll be updating the joint probability of the sentence (multiplying the probabilities of each word together). One easy way to do this is to initialize `sentprob = 1` prior to looping through the words in the sentence, and then just update `sentprob *= wordprob` with the probability of each word. At the end of the loop, `sentprob` will contain the total joint probability of the whole sentence.

   Now, you'll do the same thing for your other two models. Add code to `problem2` to calculate the probability of each sentence from the testing data and write that to a file `bigram_eval.txt`. Similarly, add code to `problem3` and write the sentence probability of each sentence from the testing data under the smoothed model to `smoothed_eval.txt`.

6. Next, you will calculate the perplexity of each model (`unigram`, `bigram`, and `smoothed_bigram`) for the testing data corpus and report them in the `Readme.txt` file. You can calculate the perplexity as follows:

$$\text{Perplexity(C)} = \sqrt[N]{\frac{1}{P(s_1, s_2, \ldots, s_m)}}$$

where,

C = Corpus (our testing set of sentences)

N = Total number of sentences in the testing set

$P(S_1, S_2, \ldots S_m) = P(S_1) * P(S_2) * \ldots * P(S_1)$

(consider the sentence probabilities from the `unigram_eval.txt`, `bigram_eval.txt` and `smoothed_eval.txt` files).

Data Files:

The following data file have already been given to you.

- A Scandal in Bohemia (`doyle_Bohemia.txt`)

Note:

- Before you can start working on any of the n-gram models, you need to clean the text file (i.e. text processing) by splitting the text into sentences and removing punctuations (*see Tutorial 1: Text Processing with NLTK*).
- Split sentences into ***Training* (80%)** & ***Testing (20%)*** set. Example, if there are overall 1000 sentences in the text file, then use 800 sentences for ***training*** *(i.e use these 800 sentences to create MLE of your 3 models)* and rest 200 sentences for ***testing*** *(i.e use these 200 sentences to calculate probabilities and then perplexity of each model).*
- When you read in the files, please convert all upper case to lower case.

Submission Guidelines:

Submit your files on Canvas using the Programming Assignment 1 submission Link. **You will submit a zip file containing:**

- Turn in your 3 models: problem1, problem2, and problem3
- The following output files:
    - `unigram_probs.txt`
    - `bigram_probs.txt`
    - `smooth_probs.txt`
    - `unigram_eval.txt`
    - `bigram_eval.txt`
    - `smoothed_eval.txt`

- `Readme.txt` with the perplexity for `unigram model`, `bigram model` and the `smoothed bigram model`. Also answer the following 2 questions in your readme

  Compare the perplexities of the testing set under all three models.
    - *Which model performed worst and why might you have expected that model to have performed worst?*

    - *Did smoothing help or hurt the model's 'performance' when evaluated on this corpus? Why might that be?*