CSCI 4820/5820 (Barbosa)
Project 2: N-Grams

Due: See course calendar for due date. May not be turned in late.

Assignment ID: **proj2**
File(s) to be submitted: **proj2.ipynb, proj2.pdf**

Objective(s): 1) Create an n-gram language model; 2) Generate text passages using n-grams; 3) Analyze the impact of symbol augmentation at the sentence level and at the paragraph level; 4) Compare the generated text from the two training sources used.

Project description:
In this assignment you will have the opportunity to apply what you learned about n-grams and their use in modeling language. Write a small Python Jupyter Notebook that constructs unigrams, bigrams, trigrams, and quadgrams language models from two training corpora (CNN News Stories and Shakespeare Play excerpts) and generates text passages from each, given a starting seed word.

Requirements:

Grading: Where two weights are shown for an item, the first is for CSCI 4820 and the second for CSCI 5820 credit.

1. (**80/65%**) Your program must provide the functionality listed below, in a single file named ***proj2.ipynb***:

- *Input* – The two training data input files consist of news stories from CNN and Shakespeare plays (randomly excerpted). Each of the input files should be processed in turn.
- *Processing* – The processing requirements include:
  - Each line in the file is a paragraph that may contain multiple sentences. The text should be lowercased during processing, augmenting symbols should be added, and n-grams should be extracted in one of two methods: at the **sentence level** and at the **paragraph level**. In no case should n-gram extraction (and thus symbol augmentation) cross over line boundaries.
  - Process each data file in the two manners discussed below, in turn:
    - **Sentence level** - The paragraph should first be sentence tokenized (using NLTK *sent_tokenize*), and all sentences subsequently word tokenized (using NLTK *word_tokenize*). The resulting data should be augmented with start and end symbols.
    - **Paragraph level** - The paragraph should be word tokenized (using *nltk word_tokenize*). The resulting data should be augmented with start and end symbols.
  - The data structure used to hold the tokens/words in each sentence should be augmented with the special start ***<s>*** and end ***</s>*** symbols, according to the n-grams being processed (higher order n-grams require more start symbol augmentation). **Care should be taken that these symbols are added after tokenization**, as the word tokenizer will split the symbols into individual characters, resulting in incorrect processing and output.
  - The extracted n-grams (unigrams, bigrams, trigrams, and quadgrams) should be kept in separate data structures (dictionaries indexed by context tuples work well for this). A parallel data structure should initially hold the counts of the tokens that immediately follow each n-gram context. These counts should be converted to, and stored, as probabilities by dividing by the total count of tokens that appear after the n-gram context.
  - Process both files **first using sentence level**, the followed by paragraph level.

- *Output* – The program should display the following output:
  - (Before producing output, set the NumPy random number generator seed to 0 for testing/debugging)
  - The number of unique unigrams, bigrams, trigrams, and quadgrams extracted (as each file is processed)
  - For each file, choose a random starting word from the unigram tokens (ensure it is not the end symbol). This word (augmented with the start symbol as required) will be the seed for the generated n-gram texts
  - For each of unigrams, bigrams, trigrams, and quadgrams:
    - Using the seed word (prefixed with the start symbol as required), generate text until either 150 tokens have been generated or the end symbol is generated. **In no case should text continue to be generated after the end symbol**.
    - Each next token should be probabilistically selected from those that follow the context (if any) for that n-gram. Use Numpy's *random.choice* to do select these.
    - When working with higher order n-grams, backoff should be used when the context does not produce a token. In that case a token should be generated from the next lower n-gram.
  - Output format – The output for each file should follow below format:

```
Extracted XX unique 1-grams
Extracted XX unique 2-grams
Extracted XX unique 3-grams
Extracted XX unique 4-grams
Seed text: YYYY
Generated 1-gram text of length X
<1-gram text generated>
Generated 2-gram text of length X
<2-gram text generated>
Generated 3-gram text of length X
<3-gram text generated>
Generated 4-gram text of length X
<4-gram text generated>
```

2. (**15%**) Test your program

- Ensure you fully test your program for required functionality.
- Your program must be tested (and must run without error) on your account at https://jupyterhub.cs.mtsu.edu/azuread/hub/login

3. (**5%**) Code comments - Add the following comments to your code:

- Place a Markdown cell at the top of the source file(s) with the following identifying information:

  Your Name
  CSCI 6350-001
  Project #X
  Due: mm/dd/yy

- Add a Markdown cell above each code cell explaining the processing carried out by that code.

4. (**CSCI 5820 students only 15 %**) Analysis – Provide an analysis (in a Markdown cell) of the following questions:

   1. Give a brief description of the quality of the text generated in each n-gram case for each of the two training files. Between training sets, focus on the coherence observed in the generated text in the quadigram cases.
   2. Discuss the differences observed between the two symbol augmentation methods (sentence level and paragraph level) and why each was characterized as it was. Do this for each training set separately.

5. Generate a pdf file of the notebook (from the terminal):

```
$ jupyter nbconvert --to html proj2.ipynb
$ wkhtmltopdf proj2.html proj2.pdf
```

6. Submit required files via the D2L dropbox for this project