CSE 143 - Assignment 2

(20 points)

Due Sunday, October 13, at 11:59 PM

1 Regular Expressions

Please create `hw2-part1.py` file for the following steps. Print the results to stdout.

1. Given the following part-of-speech tagged sentence: `All/DT animals/NNS are/VBP equal/JJ ,/, but/CC some/DT animals/NNS are/VBP more/RBR equal/JJ than/IN others/NNS ./.`

   (a) Write a regular expression that matches only the words of each word/pos-tag in the sentence, excluding all the punctuation marks.

   (b) Consider the variable `pos_sent` is equal to the sentence above. Write a python function named `get_words` that takes the `pos_sent` as input and uses regular expression in (1.1.a) to find a list of words. This function should return a sentence including each token separated by whitespace. The correct output of the `pos_sent` is : `All animals are equal but some animals are more equal than others`

   (c) Consider the variable `pos_sent` is equal to the sentence above. Write a python function named `get_pos_tags` that takes the `pos_sent` as input and uses regular expression to find a list of pos tags, excluding punctuation marks. This function should return a list including each tag in their original order. The correct output of the `pos_sent` is : `['DT', 'NNS', 'VBP', 'JJ', 'CC', 'DT', 'NNS', 'VBP', 'RBR', 'JJ', 'IN', 'NNS']`

2. Given a part-of-speech tagged sentence like the one above,

   (a) Write a regular expression that matches a simple noun phrase. A simple noun phrase is a single optional determiner followed by zero or more adjectives ending in one or more nouns.

   (b) Write a python function named `get_noun_phrases` which also takes a variable `pos_sent` as input, and searches for noun phrases in `pos_sent` using the regular expression in (1.2.a). This function should return a list of noun phrases without tags. You can use the function `get_words` created in 1.1 to remove the tags from noun phrases. The correct output of the previous sentence is : `['All animals', 'some animals', 'others']`

3. Given the pos-tag files CORPUS_NAME-pos.txt from HW 1, where the CORPUS_NAME is fables or blogs:

   (a) Write a python function named `most_freq_noun_phrase` which takes the file name as input, and prints out (to the stout) the top 3 noun phrases and their counts for each document. You can reuse the function `get_noun_phrases` in 1.2 to extract the noun phrases, then find the most common `lowercase` noun phrases. A correct output would look like this: `The most freq NP in document[7]: [('the donkey', 6), ('the mule', 3), ('load', 2)]`

   (b) Write a python function named `most_freq_pos_tags` which takes the file name as input, and prints out (to the stout) the top 3 pos tags and their counts for each document. You can reuse the function `get_pos_tags` in 1(c) to extract pos tags, then find the most common tags. A correct output would look like this: `The most freq pos tags in document[7]: [('DT', 28), ('NN', 24), ('IN', 21)]`

   Try to see if you can figure out the topic of each document by looking at these results. Which has stronger indication? Do you see any difference between fables and blogs? (You don't need to include your observations and the answer to this question in what you turn in.)

2 WordNet[1]

WordNet is imported from NLTK like other corpus readers and more details about using WordNet can be found in the NLTK book in Section 5 of Chapter 2. You can browse WordNet online at `http://wordnetweb.princeton.edu/perl/webwn` or you can use the NLTK WordNet browser by opening a command prompt (or terminal) window and typing `python` (or `python3` depending on your computer) to enter the Python environment. Type `import nltk` and `nltk.app.wordnet()`, and NLTK should open a WordNet browse page in your default browser.

In a terminal/commmand line window, for convenience in typing examples, we can shorten `wordnet` to `wn`.

```
>>> from nltk.corpus import wordnet as wn
```

2.1 Synsets and Lemmas

Although WordNet is usually used to investigate words, its unit of analysis is called a synset, representing one sense of a word. For an arbitrary word, i.e. dog, it may have different senses, and we can find its synsets. Note that each synset is given an identifier that includes one of the actual words in the synset, whether it is a noun, verb, adjective or adverb, and a number, which is relative to all the synsets listed for the particular actual word.

While using the `wordnet` functions in the following section, it is useful to also search for the word 'dog' in the online WordNet at `http://wordnetweb.princeton.edu/perl/webwn`.

```
>>> wn.synsets('dog')
```

Once you have a synset, there are functions to find information about that synset. We will start with `lemma_words`, `lemmas`, `definitions`, and `examples`.

For the first synset 'dog.n.01' (which means the first noun sense of 'dog'), we can first find all of its words, or lemma names. These are all the words that are synonyms of this sense of 'dog'.

```
>>> wn.synset('dog.n.01').lemma_names()
```

Given a synset, find all its lemmas, where a lemma is the pairing of a word with a synset:

```
>>> wn.synset('dog.n.01').lemmas()
```

Given a lemma, find its synset:

```
>>> wn.lemma('dog.n.01.domestic_dog').synset()
```

Given a word, find lemmas contained in all synsets it belongs to:

```
>>> for synset in wn.synsets('dog'):
        print(synset, ": ", synset.lemma_names())
```

Given a word, find all lemmas involving the word. Note that these are the synsets of the word 'dog', but also show that 'dog' is one of the words in each of the synsets.

```
>>> wn.lemmas('dog')
```

[1] Modified from: `http://classes.ischool.syr.edu/ist664/NLPFall2014/LabSessionWeek8.10.15.14.pdf`

2.2 Definitions and Examples

Other functions of synsets give additional information like definitions and examples. Find definitions of the synset for the first sense of the word 'dog':

```
>>> wn.synset('dog.n.01').definition()
```

Display an example use of the synset:

```
>>> wn.synset('dog.n.01').examples()
```

We can also show all the synsets and their definitions:

```
>>> for synset in wn.synsets('dog'):
        print(synset, ": ", synset.definition())
```

2.3 Lexical Relations

WordNet contains many relations between synsets. In particular, we quite often explore the hierarchy of WordNet synsets induced by the *hypernym* and *hyponym* relations. These relations are sometimes called "is-a" because they represent abstract levels of what things are. Take a look at the WordNet Hierarchy diagram, Figure 5.1 in Chapter 2, Section 5 of the NLTK book.

Find hypernyms of a synset of 'dog':

```
>>> dog1 = wn.synset('dog.n.01')
>>> dog1.hypernyms()
```

Find hyponyms of the same synset of 'dog':

```
>>> dog1.hyponyms()
```

We can find the most general hypernym as the root hypernym:

```
>>> dog1.root_hypernyms()
```

There are other lexical relations too, such as those about part/whole relations. The components of something are given by *meronymy*. NLTK has two functions for types of meronyms: `part_meronyms` and `substance_meronyms`. NLTK also has a function for things they are contained in: `member_holonyms`. Additionally, NLTK has functions for *antonymy*, or the relation of being opposite in meaning. Antonymy is a relation that holds between lemmas, since words of the same synset may have different antonyms.

```
>>> good1 = wn.synset('good.a.01')
>>> wn.lemmas('good')
>>> good1.lemmas()[0].antonyms()
```

Another type of lexical relation is the entailment of a verb (the meaning of one verb implies the other):

```
>>> wn.synset('walk.v.01').entailments()
```

There are more functions to use hypernyms to explore the WordNet hierarchy. In particular, we may want to use paths through the hierarchy in order to explore word similarity, finding words with similar meanings, or finding how close two words are in meaning.

We can use `hypernym_paths` to find all the paths from the first sense of dog to the root, and list the synset names of all the entities along those two paths.

```
>>> dog1.hypernyms()
>>> paths=dog1.hypernym_paths()
>>> len(paths)
>>> [synset.name() for synset in paths[0]]
>>> [synset.name() for synset in paths[1]]
```

2.4 Exercise

Please type the code for the following steps in `hw2-part2.py` file. Direct the output from these steps to a file called `wordnet.txt`.

1. Pick a word *w1* and show all the synsets of that word and their definitions.
2. Pick one synset of *w1* and show all of its hyponyms and root hypernyms.
3. Show the hypernym path from *w1* to the top of the hierarchy.
4. Chose another word *w2* and pick one synset for it. Then show path similarity between the synsets you choose for *w1* and *w2*.
5. Find the two words that have the highest path similarity from this list *['dog.n.01', 'man.n.01', 'whale.n.01', 'bark.n.01', 'cat.n.01']*. Output all pairs if there is a tie.
6. (Extra 2 credits.) Find the path between two random given synsets. For example *'dog.n.01'* and *'cat.n.01'*.

3 Language Models

3.1 Overview

When working with text we usually want to categorize it in some way. This is a basic form of semantics and allows us to generalize our knowledge about what the document is about. For this assignment, we will look at user generated restaurant reviews from the site we8there[2]. Given the text of a restaurant review, we'd like to know whether it is expressing a positive opinion about a restaurant or a negative opinion.

Usually, text classification requires multiple experts to assign a category from a predefined set of categories to each document we are interested in. This can be very time consuming and expensive. Fortunately, many review sites, like this one, provide a star rating along with the text. We can use this star rating as our annotation without requiring any additional manual labeling.

We can use this annotated data in many different ways. For example, we can:

- learn a classification model to predict the opinion of a new review that has not been given a star rating (for example on another website).
- use it to identify words or phrases that are most associated with positive and negative opinions.
- estimate whether we trust the star rating of an existing review.

In this assignment we will model the reviews using a language model. You will be given 150 restaurant reviews that have been given a positive rating (an overall score > 3 out of 5) and 150 that have been given a negative rating (an overall score < 3), for a total of 300 reviews. These are stored in a file called `restaurant-training.data` and can be downloaded from Canvas.

The file contains a list of user reviews. Each review has the following format:

```
Review: INTEGER
FACET_NAME_1 = VALUE_1
FACET_NAME_2 = VALUE_2
...
FACET_NAME_N = VALUE_N
```

[2] `https://www.we8there.com/`

The *FACET_NAME* is a string. The *VALUE* could be an integer, string, or a label from a closed set (e.g., {*Yes*, *No*}. Not all reviews have all facets. However, for this assignment you will only be interested in the facets *Overall*, which specifies whether the review is positive or negative, and *Text*, which gives the textual review. Each review is separated by a period character (.).

This assignment is also going to give you practice calculating probabilities. The probability of a document is the joint probability of the words in that document:

$$p(\text{document}) = p(w_1, w_2, w_3..., w_n) \tag{1}$$

This probability can be rewritten using the multiplication rule as a series of conditional probabilities:

$$p(w_1, w_2, w_3..., w_n) = p(w_1)p(w_2|w_1)p(w_3|w_2, w_1)...p(w_n|w_{n-1}, w_{n-2}..., w_1) \tag{2}$$

We generally don't have statistics for most of these conditional probabilities. So, an n-gram language model approximates this by assuming the probability of word$_i$ is only dependent on a fixed number of previous words. This is known as the Markov assumption. A bigram model is a special case where $n = 2$:

$$p(w_1, w_2, w_3..., w_n) \approx p(w_1) \prod_{i=2}^{n} p(w_i|w_{i-1}) \tag{3}$$

3.2 Procedure

Please implement `hw2-part3.py` program following these steps:

1. Normalize the data.
   - Lowercase all the words
   - Remove stop words. Get the list of stop words from `nltk.corpus.stopwords.words('english')`. See chapter 2 of NLTK.
   - Remove all tokens that do not have at least one word character. A word character is an alpha ([a-zA-Z]) or numeric ([0-9]) character that may also include an underscore (_). This class of characters is defined as `\w` in Python regular expressions.
2. Create a frequency distribution of the unigrams for each category. Write these frequencies in `descending` order to a file named `CATEGORY-unigram-freq.txt`, where CATEGORY is either *positive* or *negative*. A *positive* review is one that has an *Overall* score > 3 and a *negative* review is one that has an *Overall* score < 3. Each line of the file should contain a word and its frequency separated by whitespace.
3. Create a conditional frequency distribution from the bigrams for each category. Write these frequencies in `descending` order to a file named `CATEGORY-bigram-freq.txt`. Each line of the file should contain the condition word, the word, and the frequency, all separated by whitespace.
4. Find the collocations for each category using the `collocations()` function. Write the output to stdout (this will happen by default).

3.3 Questions

Please write down your answers in `language-model-answers.txt`

1. What are the most frequent 5 unigrams, bigrams, trigrams, 4-grams and 5-grams? And what are their frequencies.
2. What are the collocations that were found for each category?
3. Consider the normalized version of the first sentence of the training data. Given the frequency distributions you created during steps 2 and 3, calculate by hand the probability of that sentence, using a bigram model. Show your work.

4. Consider again the first sentence of the training data, but without stopwords removed. What is the probability of this sentence using a trigram model. You do not need to calculate the number. Just write out the equation with the probabilities you would need to calculate it. What order of Markov Assumption is this model using? What would be order of the Markov assumption for a 4-gram model?

5. Calculate by hand $P(the \cup wine \cup list)$ within the positive domain. Show your work.

6. What happens if you encounter a word that is not in your frequency tables when calculating the probability of an unseen sentence (a sentence that is not in your training data)?

7. A higher order n-gram (4-gram, 5-gram and so on) model is a better language model than a bi-gram or tri-gram model. Would you say this statement is correct? Please state your reasons.

4 What to Turn In

Please submit three `.py` files for Part 1 Regular Expression, Part 2 WordNet, and Part 3 Language Models Precedure.

Please also submit six `.txt` files generated for Part 2 and Part 3:

- `wordnet.txt`
- `language-model-answers.txt`
- `positive-unigram-freq.txt`
- `negative-unigram-freq.txt`
- `positive-bigram-freq.txt`
- `negative-bigram-freq.txt`

All files should be zipped before uploading to Canvas.