CSCI 140 Project 3 Part 2

Due Friday, October 16 2020 by 1700 (5:00 PM EST)

For Part 2 of this project, you will write 2 functions and a short main program to process and analyze the data. You will also make use of a function that has already been written for you. You are provided with three text files, as well a Python skeleton file (Project_3_Part2.py). Your functions AND your main program (2 data files and a small test file) will go in this file, but they must be placed in specific locations using specific syntax. Read the instructions carefully.

BE AWARE: You must use NLTK where indicated. Solutions that do not use NLTK will not receive credit.

You will use your functions to analyze a Twitter data set obtained via the Twitter API from August 9 through August 31. The data was downloaded by searching for tweets with two keywords: tortoise and umbrella and by excluding re-tweets (see below for a definition). This data was approved by Twitter for educational use. **Please be aware that this is real, raw data downloaded from the actual Twitter platform and thus contains information that is not under the control of the course instructors. If you discover content you find objectionable, please alert the instructors.** Web links have been removed from the tweets in order to reduce the amount of unfiltered content.

About Twitter data and Tweets

Tweets are submission to the social media platform Twitter. They are 140 characters in length or less. Tweets often contain hashtags which are words or phrases with the prefix #, for example, #avocadotoast. Tweets may reference other Twitter users, as indicated by the @, for example, @realDonaldTrump. Users may re-post a tweet posted by another user – this is referred to as re-tweeting, and is signified by RT followed by the user who originally posted it, for example, RT @POTUS. The tweets you will be examining for the project will contain links which have the prefix https or http. They will also contain emojis, for example: 🤔.

Some of the tweets in this project obtained via the Twitter API are truncated, i.e., the whole tweet is not shown. These tweets will have an ellipsis (an ellipsis is ...) at then end of the text and may or may not have a link after, for example:

I'm sad you are leaving. I thought you were the wisest, clearest and most honest voice on the Comm…

Setup for Working with NLTK

The instructions to install and set up NLTK are in the Module 3 Main Page. Install and set up NLTK first if you have not already done so.

Keep these instructions open while you are working. As you read through them take notes – write down the steps and the important points. Most questions can be answered and common mistakes avoided by reading through the instructions thoroughly.

All of your code will be submitted in the Project_3_Part2.py file ABOVE the line that says if __name__ == '__main__':. You will fill in your functions under the definition lines.

This shows you what a complete file looks like. Filled in code for Part 1 is shown in bold, and filled in code for Part 2 is shown in italics. Your actual code should not be in bold or italics. Your file should look something like this when you submit it. The print lines included are for demonstration purposes ONLY. They are there because we cannot fill in the actual code for you. **DO NOT PUT THESE PRINT LINES OR THE FAKE FUNCTION CALL IN YOUR CODE.**

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import TweetTokenizer


def make_tokens(text): #do not change
    tt = TweetTokenizer(preserve_case=False) #do not change
    tokens = tt.tokenize(text) #do not change
    return tokens #do not change


def prune_tweet_tokens(tokens, stops = []): #do not change
    print('This is where your code for this function goes.')


def analyze(filename, analysis = 'frequency', stops = [], number = 10, word = ''):
    print('This is where your code for this function goes.')


if __name__ == '__main__':
    print('This is where you paste in your main program. An example function call:')
    fake_call_result = make_tokens('fake_file.txt')
```

Be sure to ask if you do not understand. An incorrectly constructed file will mean that your code will cannot be tested.

Task 1: Functions

Function already written for you

One function has already been provided for you. The function **make_tokens(text)** takes a single argument, **text**, which is a string of text. It returns a list of tokens produced by NLTK's TweetTokenizer. You are required to use this function in your **analysis** function (see below). Test this function to see what it does. You may want to review the readings on TweetTokenizer.

Functions for you to write

Task 1: *prune_tweet_tokens(tokens, stops = [])*

This function may seem strange because it eliminates all of the features we were so interested in in Part 1 of this project! The purpose of this function is to prepare our list of words for analysis such as word frequencies and word usage patterns. This function does not require the use of NLTK – if you are using NLTK functions in it, you might be doing it wrong.

prune_tweet_tokens takes one required argument, **tokens**, which is a list of strings. We will assume that the strings in **tokens** are the output from using a TweetTokenizer – that is, the input to this function will always be something in the same format as what TweetTokenizer (and the **make_tokens** function) produces. Keep that in mind. Your function should do the following:

- Remove emojis
- Remove hashtags
- Remove mentions
- Remove punctuation, i.e. any of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ to start with
- Remove stopwords provided in the optional argument **stops,** which can be a list or set

Notice that the default value of **stops** is an empty list, which means by default no stopwords are removed. When you write your main program (see below) you will need to create a set of stopwords using the NLTK english stopwords as a starting point.

Here are some examples of how **prune_tweet_tokens** works. This is not actual code, and words like "Input tokens" should not print when your function runs.

Input tokens: `['#turtles', '#tortoise', 'rainbow', '🌈', 'the', 'therapy', 'tortoise', '🐆', 'we', 'took', 'rainbow', 'the', 'leopard', 'tortoise', 'to', 'visit', 'granny', 'today', '.']`

stops is `{'the', 'we', 'to'}`

prune_tweet_tokens returns: `['rainbow', 'therapy', 'tortoise', 'took', 'rainbow', 'leopard', 'tortoise', 'visit', 'granny', 'today']`

Input tokens: `['sen', '.', 'sanders', 'proposes', 'one-time', 'tax', 'that', 'would', 'cost', 'bezos', '$', '42.8', 'billion', ',', 'musk', '$', '27.5', 'billion']`

Default value of **stops**

prune_tweet_tokens returns: `['sen', 'sanders', 'proposes', 'one-time', 'tax', 'that', 'would', 'cost', 'bezos', '42.8', 'billion', 'musk', '27.5', 'billion']`

You may be confused by the last example. Remember how Tweet_Tokenizer works and how it separates out actual punctuation marks. When we prune tweet tokens, are we editing strings or are we editing a list?

Write the whole thing in one line with list comprehension for extra credit!

Task 2: *analyze(filename, analysis = 'frequency', stops = [], number = 10, word = '')*

The function **analyze** will explore the characteristics of our text. The user can choose to get word frequency information, word context, or collocations (words that appear together).

analyze takes one required input, **filename,** which is a string that represents the name of a file. The optional argument **analysis** is a string that determines what analysis the function will perform and what it either returns or prints. The argument **analysis** will always be either 'frequency' , 'concordance', or 'collocations' - let's look at the possible cases:

Case 1: **analysis** is *'frequency'*

The function will perform a word frequency analysis. The text stored in the file represented by **filename** will be processed into tokens, and then stopwords, punctuation, mentions, hashtags and emojis should be removed. The stopwords to remove are indicated by the optional argument **stops**. The function will then compute the **number** most common words, and return a list of tuples containing these words and their frequencies. The optional argument **number** determines how many words to return. Here are two examples:

- **stops** are the NLTK English stopwords, **number** is 5, **filename** is 'test_file.txt' (test file provided with this project – you should open it and look at it to better understand what this function does); **analyze** returns:

```
[('tortoise', 5), ('via', 2), ('r', 2), ('rainbow', 2), ('today', 2)]
```

(The r is coming from a reddit mention)

- **stops** is the function default value (empty list), **number** is 10, **filename** is 'test_file.txt' (test file provided with this project); **analyze** returns:

```
[('tortoise', 5), ('the', 3), ('via', 2), ('r', 2), ('rainbow', 2), ('today', 2),
('must', 2), ('tootles', 1), ('otis', 1), ('and', 1)]
```

Don't get hung up on the list of tuples. Review NLTK. This should be automatic if you are using the correct objects/functions from NLTK.

Case 2: **analysis** is *'concordance'*

This function finds context (the text surrounding a word) for a word input in the optional argument **word**. The number of concordances (contexts) to find and print is determined by the optional argument **number**. In this case the function prints the concordances but returns nothing. There should not be a line that starts with **return** in this part of the function. Here is an example:

- **word** is 'tortoise' , **number** is 5, **filename** is 'test_file.txt' (test file provided with this project); **analyze** returns None, **analyze** prints:

```
Displaying 5 of 5 matches:
ay everyone make it a fiesta via r tortoise rainbow the therapy tortoise we to
via r tortoise rainbow the therapy tortoise we took rainbow the leopard tortoi
rtoise we took rainbow the leopard tortoise to visit granny today how do you f
granny today how do you feel today tortoise or hare mark must must be the tort
oise or hare mark must must be the tortoise of readers
```

Note that you should NOT have a line of code that says return None anywhere in this function, this project, or anything you do for this class. Remember also that when we analyze context we don't remove stopwords. We will remove punctuation, emojis, mentions and hashtags. (HINT: this is so that you can use one of your other functions for processing.)

Case 3: **analysis** is *'collocations'*

This function finds the collocations, aka bigrams, i.e. word pairs that occur together in the text. The number of collocations to print is determined by the argument **number**. Here is an example:

- **filename** is 'tortoise_tweets.txt', **number** is 5; **analyze** returns None (but you should not have a line of code that says this), **analyze** prints a list of collocations – this may look slightly different depending on your version of NLTK. Your output should look like this:

Newer NLTK:

```
[('acquisition', 'corp'), ('tortoise', 'shell'), ('looks', 'like'), ('body', 'reusable'),
('reusable', 'wipes')]
```

Older NLTK:

```
['acquisition corp', 'tortoise shell', 'looks like', 'body reusable', 'reusable wipes']
```

Please note that if you have slight differences, like you have 4 of these collocations but the fifth is different, it is likely not an issue. Just make sure that you calling collocations correctly.

These should print as strings in a list. Why? Because that's easier for you – if you are using NLTK correctly, it will print just like this with no extra effort. If you are writing code to try to turn things into a list, something has gone wrong.

Note that you should NOT have a line of code that says return None anywhere in this function, this project, or anything you do for this class. Remember also that when we analyze collocations we don't remove stopwords. We will remove punctuation, emojis, mentions and hashtags. (HINT: this is so that you can use one of your other functions for processing.)

Key points:

- You MUST use NLTK for the analysis. This will help you....creating a list of tuples sounds really difficult, but NLTK does that automatically. There is a function called concordance and one called collocation_list that you should use. Refer back to the readings.
- You MUST use your other functions, **make_tokens** and **prune_tweet_tokens**, to complete this function. You should not repeat code from these functions here.
- For all cases, remove punctuation, emojis, hashtags and mentions. The *frequency* case is the ONLY case for which you should remove stopwords.
- Notice that in the *frequency* case, the **word** argument is not used at all. In the *concordance* case, the **stops** argument should not be used at all. In the *collocations* case, **stops** and **word** should not be used at all.
- The only case that returns something is the *frequency* case. The other two cases print – they should print the output, but not print the word None.
- Make your code efficient. See what you can combine from the various cases instead of repeating the same code. Don't include optional arguments if you are using the default values.
- The function only returns something for the *frequency* case. The other two cases just print something. If the word "None" prints in the *concordance* and *collocations* cases, something is wrong.

Taks Two: Main Program

For this part of the assignment you will use your functions to analyze the data in the tortoise_tweets.txt and umbrella_tweets.txt files. You will be graded on style. This means not using default arguments when possible (i.e. not specifying them in your code if you are using the default value), minimizing extraneous variables, and not having the word None print out.

- Create a list or set that contains all of the English stopwords from NLTK. This should be done using NLTK OR if you are unable to use NLTK stopwords, by reading in from the text file stopwords.txt. Use these stopwords below as needed.
- Generate and print a list of tuples containing the top 15 most frequently used words and the number of times they occur, excluding stopwords and any type of punctuation, from the umbrella_tweets.txt file. You will need to refine your stopwords or your definition of punctuation to make this happen. Part of the requirements for this part of the assignment involve this refinement so you should include the code you used to do this.
- Generate and print a list of tuples containing the top 5 most frequently used words and the number of times they occur, excluding stopwords and any type of punctuation, from the tortoise_tweets.txt file. You will need to refine your stopwords or your definition of punctuation to make this happen. Part of the requirements for this part of the assignment involve this refinement so you should include the code you used to do this.
- Generate and print 5 collocations for tortoise_tweets.txt
- Generate and print 4 concordances (words in context) for the word **watch** in umbrella_tweets.txt

SUBMISSION EXPECTATIONS

`Project_3_Part_2.py` Your implementations of the functions for Task 1 in the appropriate location. You implementation of the main program in the appropriate location. If your functions and/or main program are not in the appropriate parts of the file, your code may fail testing.

`Project_3_Part_2.pdf` A PDF document containing your reflections on the project including any extra credit opportunities you chose to pursue. Also, explain the choices you made in the main program to refine your frequency distribution. **You must also cite any sources you use. Please be aware that you can consult sources, but all code written must be your own. Programs copied in part or wholesale from the web or other sources will result in reporting of an Honor Code violation.**

POINT VALUES AND GRADING RUBRIC

This part of the project is worth 10 points.

Part 1:

- prune_tokens (3.25 pts)
- analysis (4.75 pts)

Part 2: Main Program (1.75 pts)

Writeup – 0.25 pts

Grade Level	Execution	Correctness and Algorithms	Formatting	Style and References
A	Code executes without errors, requires no manual intervention	Code correctly implements all algorithms, passes all test cases (with possible exception of extremely rare case as applicable), and meets all required functionality	Inputs and outputs formatted as per the specification both in terms of content and type, may be minimal formatting issues (spacing, punctuation, etc.); correct file formats and names	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; any references cited in write-up
B	Code executes without errors, may require minimal manual intervention such as change of an import line	Meets all required functionality with exception of minor/ rare test cases (as applicable)	Inputs and outputs formatted as per the specification both in terms of content and type, some minor formatting issues may be present (e.g. incorrect string text), correct file formats and names	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; any references cited in write-up
C	Code does not execute without minor manual intervention such as correcting indentation or terminating parentheses or a single syntax error	Code contains logical errors and fails subset of both rare and common test cases, overall algorithmic structure reflects required functionality but implementation does not meet standards	Inputs and outputs have major formatting issues, files incorrectly named but file formats correct	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; references cited in write-up
D	Code does not execute without major manual intervention such as changing variable names, correcting multiple syntax errors, algorithmic changes	Code contains major logical errors and fails majority of test cases, lack of cohesive algorithmic structure, minimal functionality	Inputs and outputs have major formatting issues, files incorrectly named but file formats correct	Code uses some structures and modules from class, but otherwise largely taken from outside sources with citations; variable names and code structure hard to decipher
F	Code requires extensive manual intervention to execute	Code fails all or nearly all test cases and has no functionality to solve the proposed problem	Inputs and outputs have major formatting issues, files in incorrect formats which cannot be graded	Code uses structures and modules from outside sources only with no citations; variable names and code structure hard to decipher; code written in other programming languages or Python 2.7