Problem Set 3: Document Distance

Pset Buddy

You do not have a buddy assigned for this pset.

Introduction

Objectives

- Introducing the concept of dictionaries in Python
- Writing and calling helper functions in Python

Collaboration

- Students may work together, but each student should write up and hand in their assignment separately. Students may not submit the exact same code.
- Students are not permitted to look at or copy each other's code or code structure.
- Include the names of your collaborators in comment at the start of each file.
- **Please refer to the collaboration policy in the** Course Information **for more details.**

Although this handout is long, the information is here to provide you with context, useful examples, and hints, so be sure to read carefully.

A) File Setup

Download the file **1_ps3.zip** and extract all files to the same directory. The files included are: **document_distance.py**, **test_ps3_student.py**, and various documents of texts and lyrics within the **tests/student_tests** directory. When you are done, make sure you run the tester file **test_ps3_student.py** to check your code against some of our test cases.

You will edit ONLY document_distance.py.

B) Document Distance Overview

Given two words or documents, you will calculate a score between 0 and 1 that will tell you how similar they are. If the words or documents are the same, they will get a score of 1. If the documents are completely different, they will get a score of 0. You will calculate the score in two different ways and observe whether one works better than the other. The first way will use single word frequencies in the two texts. The second will use the TF-IDF (Term Frequency-Inverse Document Frequency) of words in a file.

Note that you do NOT need to worry about case sensitivity throughout this pset. All inputs will be lower case.

1) Text to List

The first step in any data analysis problem is prepping your data. We have provided a function called **`load_file`** to read a text file and output all the text in the file into a string. This function takes in a variable called `filename`, which is a string of the filename you want to load, including the extension. It removes all punctuation, and saves the text as a string. **Do not modify this function.**

Here's an example usage:

```
# hello_world.txt looks like this: 'hello world, hello'
>>> text = load_file("tests/student_tests/hello_world.txt")
>>> text
'hello world hello'
```

You will further prepare the text by taking the string and transforming it into a list representation of the text. Given the example from above, here is what we expect:

```
>>> text_to_list('hello world hello')
['hello', 'world', 'hello']
```

Implement **`text_to_list`** in **document_distance.py** as per the given instructions and docstring. In addition to running the tester file, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of **document_distance.py**:

```
if __name__ == "__main__":
    # Tests Problem 0: Prep Data
    test_directory = "tests/student_tests/"
    hello_world, hello_friend = load_file(test_directory + 'hello_world.txt'), load_file(test_directory + 'h
    world, friend = text_to_list(hello_world), text_to_list(hello_friend)
    print(world)      # should print ['hello', 'world', 'hello']
    print(friend)     # should print ['hello', 'friends']
```

Note: You can assume that the only kinds of white space in the text documents we provide will be new lines or space(s) between words (i.e. there are no tabs).

2) Get Frequencies

Let's start by calculating the frequency of each element in a given list. The goal is to return a dictionary with each unique element as the key, and the number of times the element occurs in the list as the value.

Consider the following examples:

Example 1:

```
>>> get_frequencies(['h', 'e', 'l', 'l', 'o'])
{'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

Example 2:

```
>>> get_frequencies(['hello', 'world', 'hello'])
{'hello': 2, 'world': 1}
```

Implement **`get_frequencies`** in **document_distance.py** using the above instructions and the docstring provided. In addition to running the tester file, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of **document_distance.py**:

```
if __name__ == "__main__":
    # Tests Problem 1: Get Frequencies
    test_directory = "tests/student_tests/"
    hello_world, hello_friend = load_file(test_directory + 'hello_world.txt'), load_file(test_directory + 'h
    world, friend = text_to_list(hello_world), text_to_list(hello_friend)
    world_word_freq = get_frequencies(world)
    friend_word_freq = get_frequencies(friend)
    print(world_word_freq)    # should print {'hello': 2, 'world': 1}
    print(friend_word_freq)   # should print {'hello': 1, 'friends': 1}
```

3) Letter Frequencies

Now, given a word in the form of a string, let's create a dictionary with each letter as the key and how many times each letter occurs in the word as the value. That sounds very similar to **`get_frequencies`...**

You must call `get_frequencies` in your `get_letter_frequencies` to get full credit.

Example 1:

```
>>> get_letter_frequencies('hello')
{'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

Example 2:

```
>>> get_letter_frequencies('that')
{'t': 2, 'h': 1, 'a': 1}
```

Implement **`get_letter_frequencies`** in **document_distance.py** using the above instructions and the docstring provided. In addition to running the tester file, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of **document_distance.py**:

```
if __name__ == "__main__":
    # Tests Problem 2: Get Letter Frequencies
    freq1 = get_letter_frequencies('hello')
    freq2 = get_letter_frequencies('that')
    print(freq1)      #  should print {'h': 1, 'e': 1, 'l': 2, 'o': 1}
    print(freq2)      #  should print {'t': 2, 'h': 1, 'a': 1}
```

4) Similarity

Now it's time to calculate similarity! Complete the function **`calculate_similarity_score`** based on the definition of **similarity** found in the next paragraph. Your function should be able to be used with the outputs of `get_frequencies` or `get_letter_frequencies`.

Consider two lists L_1 and L_2. Let U be a list made up of all the elements in L_1 or L_2, but with no repeats (e.g. if $L_1 =$ **`['a', 'b']`**, $L_2 =$ **`['b', 'c']`**, then $U =$ **`['a', 'b', 'c']`**). For an element e in L_1 or L_2, let

$$\text{count}(e, L_i) = \begin{cases} \text{number of times } e \text{ appears in } L_i & \text{if } e \text{ in } L_i, \\ 0 & \text{if } e \text{ not in } L_i. \end{cases}$$

We can then define:

- $\delta(e) = |\text{count}(e, L_1) - \text{count}(e, L_2)|$ (where the vertical bars denote absolute value), and
- $\sigma(e) = \text{count}(e, L_1) + \text{count}(e, L_2)$.

Similarity is defined as:

$$1 - \frac{\delta(u_1) + \delta(u_2) + \delta(u_3) + \ldots}{\sigma(u_1) + \sigma(u_2) + \sigma(u_3) + \ldots}$$

where the sums are taken over all the elements $u_1, u_2, u_3, \ldots$ of U, and **the result is rounded to two decimal places**.

Example (where elements are words):

- Suppose
  - $L_1 =$ **`['hello', 'world', 'hello']`**, and
  - $L_2 =$ **`['hello', 'friends']`**.
- The list of unique elements U is $U =$ **`['hello', 'world', 'friends']`**.
- The frequency differences $\delta(u)$ are
  - $\delta($**`'hello'`**$) = |2 - 1| = 1$
  - $\delta($**`'world'`**$) = |1 - 0| = 1$
  - $\delta($**`'friends'`**$) = |0 - 1| = 1$
- The frequency totals $\sigma(u)$ are

    - $\delta(\texttt{'hello'}) = 2 + 1 = 3$
    - $\delta(\texttt{'world'}) = 1 + 0 = 1$
    - $\delta(\texttt{'friends'}) = 0 + 1 = 1$
- Thus, **similarity** is $1 - \frac{1+1+1}{3+1+1} = 1 - \frac{3}{5} = \boxed{0.4}$ (0.4 rounded to two decimal places is still 0.4).

The same calculation with an alternate (but equivalent) explanation can be found in the `calculate_similarity_score`'s docstring.

IMPORTANT: Be sure to round your final similarity calculation to 2 decimal places.

Implement the function **`calculate_similarity_score`** in **document_distance.py** with the given instruction and docstrings. In addition to running the tester file, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of **document_distance.py**:

```
if __name__ == "__main__":
    # Tests Problem 3: Similarity
    test_directory = "tests/student_tests/"
    hello_world, hello_friend = load_file(test_directory + 'hello_world.txt'), load_file(test_directory + 'h
    world, friend = text_to_list(hello_world), text_to_list(hello_friend)
    world_word_freq = get_frequencies(world)
    friend_word_freq = get_frequencies(friend)
    word1_freq = get_letter_frequencies('toes')
    word2_freq = get_letter_frequencies('that')
    word3_freq = get_frequencies('nah')
    word_similarity1 = calculate_similarity_score(word1_freq, word1_freq)
    word_similarity2 = calculate_similarity_score(word1_freq, word2_freq)
    word_similarity3 = calculate_similarity_score(word1_freq, word3_freq)
    word_similarity4 = calculate_similarity_score(world_word_freq, friend_word_freq)
    print(word_similarity1)       # should print 1.0
    print(word_similarity2)       # should print 0.25
    print(word_similarity3)       # should print 0.0
    print(word_similarity4)       # should print 0.4
```

5) Most Frequent Word(s)

Next, you will find out which word(s) occurs the most frequently among two dictionaries. You'll count how many times every word occurs **combined across both texts** and return a list of the most frequent word(s). **The most frequent word does not need to appear in both dictionaries**. It is based on the combined word frequencies across both dictionaries. If a word occurs in both dictionaries, consider the sum of frequencies as the combined word frequency. If multiple words are tied (i.e. have the same highest frequency), return an *alphabetically ordered list* of all these words.

For example, consider the following usage:

```
>>> freq1 = {"hello": 5, "world": 1}
>>> freq2 = {"hello": 1, "world": 5}
>>> get_most_frequent_words(freq1, freq2)
["hello", "world"]
```

Implement the function **`get_most_frequent_words`** in **document_distance.py** as per the given instructions and docstring. In addition to running the tester file, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of **document_distance.py**:

```
if __name__ == "__main__":
    # Tests Problem 4: Most Frequent Word(s)
    freq_dict1, freq_dict2 = {"hello": 5, "world": 1}, {"hello": 1, "world": 5}
    most_frequent = get_most_frequent_words(freq_dict1, freq_dict2)
    print(most_frequent)      # should print ["hello", "world"]
```

6) Term Frequency–Inverse Document Frequency (TF-IDF)

In this part, you will calculate the **Term Frequency–Inverse Document Frequency**, which is a numerical measure that signifies the importance of word(s) in a document. You will do so by first calculating the **term frequency** and **inverse document frequency**, then combine the two together to get

the **TF-IDF**.

The term frequency (TF) is calculated as:

$$\text{TF}(w) = \frac{\text{number of times word } w \text{ appears in the document}}{\text{total number of words in the document}}.$$

The inverse document frequency (IDF) is calculated as:

$$\text{IDF}(w) = \log_{10}\left(\frac{\text{total number of documents}}{\text{number of documents with word } w \text{ in it}}\right)$$

where $\log_{10}$ is log base 10 and can be called with `math.log10`.

We can then combine TF and IDF to form $\texttt{TF-IDF}(w) = \text{TF}(w) \times \text{IDF}(w)$, where the higher the value, the rarer the term and vice versa. For this pset, we'll only be working with individual words, but TF-IDF works for larger groupings of words as well (e.g. bigrams, trigrams, etc.).

For the **`get_tf`** function that you'll implement, you'll be given a file name stored in a variable named `text_file`. You will need to load the file, prep the data, and **determine the TF value** of each word that appears in `text_file`. The output should be a dictionary mapping each word to its TF. Think about how you could re-use previous functions.

For the **`get_idf`** function that you'll implement, you'll be given a list of text files stored in a variable named `text_files`. You will need to load each of the files, prep the data, and **determine the IDF values** of all words that appear in any of the documents in `text_files`. The output should be a dictionary mapping each word to its IDF.

For the **`get_tfidf`** function that you'll implement, you'll be given a file name `text_file` and a list of file names `text_files`. You will need to load the file, prep the data, and **determine the TF-IDF** of all words in `text_file`. The output should be a sorted list of tuples (in increasing TF-IDF score), where each tuple is of the form `(word, TF-IDF)`. In case of words with the same TF-IDF, the words should be sorted in increasing alphabetical order.

For example,

```
>>> text_file = "tests/student_tests/hello_world.txt"
>>> get_tf(text_file)
{"hello": 0.6666666666666666, "world": 0.3333333333333333}
# Explanation: There are 3 total words in "hello_world.txt". 2 of the three total words are "hello", giving

>>> text_files = ["tests/student_tests/hello_world.txt", "tests/student_tests/hello_friends.txt"]
>>> get_idf(text_files)
{"hello": 0.0, "world": 0.3010299956639812, "friends": 0.3010299956639812}
# Explanation: There are a total of 2 documents in this example. "hello" is in both documents, giving "hell

>>> text_file = "tests/student_tests/hello_world.txt"
>>> text_files = ["tests/student_tests/hello_world.txt", "tests/student_tests/hello_friends.txt"]
>>> get_tfidf(text_file, text_files)
[('hello', 0.0), ('world', 0.10034333188799373)]
# Explanation: We multiply the corresponding TF and IDF values for each word in "hello_world.txt" and get th
```

Implement the functions **`get_tf`**, **`get_idf`**, and **`get_tfidf`** in **document_distance.py** as per the given instructions. In addition to running the tester file, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of **document_distance.py**:

```
if __name__ == "__main__":
    # Tests Problem 5: Find TF-IDF
    tf_text_file = 'tests/student_tests/hello_world.txt'
    idf_text_files = ['tests/student_tests/hello_world.txt', 'tests/student_tests/hello_friends.txt']
    tf = get_tf(tf_text_file)
    idf = get_idf(idf_text_files)
    tf_idf = get_tfidf(tf_text_file, idf_text_files)
    print(tf)     # should print {'hello': 0.6666666666666666, 'world': 0.3333333333333333}
    print(idf)    # should print {'hello': 0.0, 'world': 0.3010299956639812, 'friends': 0.3010299956639812}
    print(tf_idf) # should print [('hello', 0.0), ('world', 0.10034333188799373)]
```

When you are done, make sure you run the tester file test_ps3_student.py to check your code against our test cases.

7) Hand-in Procedure

7.1) Naming Files

Save your solutions with the original file name: **document_distance.py**. **Do not ignore this step or save your file with a different name!**

7.2) Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of your collaborators. For example:

```
# Problem Set 3
# Name: Jane Lee
# Collaborators: John Doe
```

Please estimate the number of hours you spent on the Problem Set in the question box below.

A Python Error Occurred:

```
Error on line 30 of question tag.ImportError: cannot import name 'Sequence' from 'collections' (/home/cat
```

7.3) Half-way Submission

All students should submit their progress by the half-way due date (1 week before the final due date).
This submission will be worth 1 point out of the problem set grade and will not be graded for correctness. The intention is to make sure that you are making steady progress on the problem set as opposed to working on it in the final days before the due date.

You may upload new versions of each file until Oct 12 at 09:00PM. You cannot use extensions or late days on this submission.

Please refresh the page before submitting a new file. If you do not, your latest submission won't be updated.

Select File No file selected

Submit

You have infinitely many submissions remaining.

7.4) Final Submission

Be sure to run the student tester and make sure all the tests pass. However, the student tester contains only a subset of the tests that will be run to determine the problem set grade. Passing all of the provided test cases does not guarantee full credit on the pset.

You may upload new versions of each file until Nov 02 at 09:00PM, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

When you upload a new file with the same name, your old one will be overwritten.

Please refresh the page before submitting a new file. If you do not, your latest submission won't be updated.

Select File No file selected

Submit

You have infinitely many submissions remaining.

Supplemental Reading about Document Similarity

This pset is a greatly simplified version of a very pertinent problem in Information Retrieval. Applications of document similarity range from retrieving search engine results to comparing genes and proteins to improving machine translation.

More advanced techniques to calculating document distance include transforming the text into a vector space and computing the cosine similarity, Jaccard Index, or some other metric of the vectors.

6.100L Introduction to CS and Programming Using Python

Fall 2022