Assignment 4 — Document Distance

How alike are two texts? Count their words and see

Fakhir Shaheen *(sec - A,B)*

DUE Thu 26 Nov 2026, 11:59 PM WEIGHT 3.375% COVERS Lectures 1–17 FILES 1 CHECKER `check.py`

Objectives

- Introducing the concept of dictionaries and their applications
- Writing and calling helper functions
- Basic file handling

Collaboration

- Students should write up and hand in their assignment separately. Students may not submit the exact same code. There is **NO** reason for multiple students to have **similar looking code**.
- **Students are NOT permitted to look at or copy each other's code or "code structure" / logic.**
- **AI tools (ChatGPT, Claude, Gemini, Copilot, or any other) are allowed ONLY for explanations, NOTHING ELSE. Not for writing code, fixing code or debugging or writing solutions. Do not paste this sheet or your file into any AI tool. If you use one in any way, it is plagiarism and you get 0 in A4.**

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

How this assignment works

- Work top to bottom. Read a task, answer its keys, then write its code. Tasks open one at a time: the next opens when this one passes.
- ✎ means pen on this sheet, Shell closed. Only calculators are allowed. ⌨ means at the machine, in `a4.py`.
- The deliverable is `a4.py` in your `a4` folder. Tasks 1 to 8 must PASS, and each is worth the same.
- `check.py` checks your functions the way you can yourself. After `a4.py` has run, it types calls into the Shell, such as `get_letter_frequencies('hello')`, and compares what comes back. Run it as often as you like, and hand in from its menu.
- This assignment assumes Labs 1 to 9 and uses Lectures 1 to 17. Lecture 17 (Dictionaries) is the one it needs most.

Getting Started

A) File Setup

Before you start: get the files

Download the A4 kit from the course calendar and unzip it: `a4.py`, `check.py`, and various documents of texts and lyrics within the folder `tests/student_tests`. **Keep them together in one folder**, `a4`, and do not rename them.
You will edit ONLY `a4.py`.
If you are using Thonny, open `a4.py` and press F5: it should print nothing and stop without an error. Then open `check.py` and press F5 to check your functions.

B) Document Distance Overview

Given two words *(or even two 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. If they are somewhat similar then they will get a floating value score between 0 and 1.

You can use this to detect plagiarism, find similar documents, or even to recommend similar songs or movies to a user.

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 assignment. All inputs will be lower case.

Important: Do NOT change the name, input parameters, or specifications of any of the provided functions! You may add helper functions, but changing the definitions of the given functions will cause `check.py`'s calls to fail.

C) How `check.py` tests your functions

`check.py` calls each function by itself, from the Shell. So a function must **return** its answer and print nothing. Three things about its calls:

- **A dictionary has no order that matters.** So when a function returns a dictionary, `check.py` sorts its pairs before it compares them:

```
>>> sorted(get_frequencies(['h', 'e', 'l', 'l', 'o']).items())
[('e', 1), ('h', 1), ('l', 2), ('o', 1)]
```

- **The grader has no `tests` folder.** So `check.py` does not read your files from the disk. It first types the text of each file into the Shell (`files = {...}`) and makes `open` read from those texts. Then `load_file` works as usual. So read a file only with `load_file`, and give it the file name exactly as you got it.
- **Floats may differ in their last digits.** `0.30000000000000004` counts as `0.3`.

If your function is correct, its row in the report says PASS:

```
  Task 02  Get Frequencies            PASS
```

1 Text to List (~15 min)

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']
```

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).*

TASK 1 **Text to List** EASY a4.py ~15 min

Every later function works on a list of words, so this one comes first.

Your task: Implement `text_to_list` in `a4.py` as per the given instructions and docstring.

In addition to running `check.py`, you can quickly check your implementation on the provided examples for each problem by uncommenting the relevant lines of code at the bottom of `a4.py`:

```
if __name__ == "__main__" and os.path.isdir("tests"):
    ## Tests Problem 0: Prep Data
    test_directory = "tests/student_tests/"
    hello_world = load_file(test_directory + 'hello_world.txt')
    hello_friend = load_file(test_directory + 'hello_friends.txt')
    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']
```

Why: `hello_friends.txt` holds `hello friends`: two words, so a list of two.

These lines run only when the `tests` folder is next to `a4.py`: on your machine, never on the grader. Leave the `if` line as it is.

Testing: run `check.py`. It covers the old tests `test_prep_hello_world`, `test_prep_1a` and `test_prep_all_whitespace`: a text with two spaces in a row and new lines, and the long text `test1a.txt`.

Pen on this sheet, Shell closed, before you write Task 1 in `a4.py`. Only calculators are allowed.

Key 1.1 A file holds the line `Rain, rain -- go away!` Predict how many elements the list `text_to_list(load_file(...))` should have.

Answer: .....

Key 1.2 A text starts with two spaces, then `hi`, then a blank line, then `there`, then two spaces. Which list should `text_to_list` give for it?
(a) `["  ", 'hi', "  ", 'there', "  "]`
(b) `['hi', 'there']`
(c) `['hi', "  ", 'there']`

Key 1.3 A file holds `Don't stop.` Predict the first element of the list that `text_to_list` gives for it. Write the element without quotes.

Answer: .....

2 Get Frequencies (~20 min)

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}
```

TASK 2 Get Frequencies EASY `a4.py` ~20 min

⌨ Read the docstring: the input may be a list of strings *or a string*. The same loop must work for both.

Your task: Implement `get_frequencies` in `a4.py` using the above instructions and the docstring provided.

In addition to running `check.py`, you can quickly check your implementation by uncommenting the relevant lines of code at the bottom of `a4.py`:

```
if __name__ == "__main__" and os.path.isdir("tests"):
    ## Tests Problem 1: Get Frequencies
    test_directory = "tests/student_tests/"
    hello_world = load_file(test_directory + 'hello_world.txt')
    hello_friend = load_file(test_directory + 'hello_friends.txt')
    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}
```

Why: `hello` is twice in `hello world hello`, and `world` once.

Testing: run `check.py`. It covers the old tests `test_frequency_word_hello`, `test_frequency_hello_world`, `test_frequency_1a` and `test_frequency_1b`, and calls `get_frequencies` on a string.

Pen on this sheet, Shell closed, before you write Task 2 in `a4.py`. Only calculators are allowed.

Key 2.1 Predict how many keys `get_frequencies(['red', 'blue', 'red', 'green', 'blue', 'red'])` should have.

Answer: .....

Key 2.2 Predict the value for the key `'t'` in `get_frequencies('letter')`.

Answer: .....

Key 2.3 Which dictionary should `get_frequencies(['b', 'a', 'b'])` give?
(a) `{'b': 1, 'a': 1}`
(b) `{2: 'b', 1: 'a'}`
(c) `{'b': 2, 'a': 1}`

3 Letter Frequencies (~10 min)

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. `check.py` checks this: a `get_letter_frequencies` that never calls `get_frequencies` does not pass.

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}
```

TASK 3 Letter Frequencies EASY `a4.py` ~10 min

This function is short, because Task 2 does the work.

Your task: Implement `get_letter_frequencies` in `a4.py` using the above instructions and the docstring provided.

In addition to running `check.py`, you can quickly check your implementation by uncommenting the relevant lines of code at the bottom of `a4.py`:

```
if __name__ == "__main__" and os.path.isdir("tests"):
    ## 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}
```

Why: `that` has two t's, one h and one a.

Testing: run `check.py`. It covers the old tests `test_letter_frequency_1` and

`test_letter_frequency_4` (`supercalifragilisticexpialidocious`), and every call must go through `get_frequencies`.

Pen on this sheet, Shell closed, before you write Task 3 in `a4.py`. Only calculators are allowed.

Key 3.1 Predict the value for the key `'o'` in `get_letter_frequencies('cocoa')`.
Answer: .....

Key 3.2 Predict how many keys `get_letter_frequencies('balloon')` should have.
Answer: .....

Key 3.3 Why can `get_letter_frequencies` be a single line that calls `get_frequencies`?
(a) `get_frequencies` first splits the word into letters with `split()`.
(b) A for loop over a string gives its letters one at a time, so `get_frequencies` counts letters the way it counts words.
(c) It cannot: the word must first be turned into a list of words.

4 Similarity (~35 min)

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 elements in L_1 and 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:

- $\delta(e) = |\,\text{count}(e, L_1) - \text{count}(e, L_2)\,|$
- $\sigma(e) = \text{count}(e, L_1) + \text{count}(e, L_2)$

Similarity is defined as:

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

where the sums are taken over all the elements $u_1, u_2, u_3, \cdots$ 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:
  - δ(`'hello'`) $= |2 - 1| = 1$
  - δ(`'world'`) $= |1 - 0| = 1$
  - δ(`'friends'`) $= |0 - 1| = 1$
- The frequency totals $\sigma(u)$ are:

- $\sigma(\texttt{'hello'}) = 2 + 1 = 3$
- $\sigma(\texttt{'world'}) = 1 + 0 = 1$
- $\sigma(\texttt{'friends'}) = 0 + 1 = 1$

- Thus, **similarity** is calculated as:

$$1 - \frac{1+1+1}{3+1+1} = 1 - \frac{3}{5} = 0.40$$

(0.40 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: there the top sum is called DIFF and the bottom sum ALL.

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

TASK 4 Similarity

MEDIUM `a4.py` ~35 min

⌨ The function gets two frequency dictionaries, not two lists: the counts are already done. Walk over the keys of both.

Your task: Implement the function `calculate_similarity_score` in `a4.py` with the given instruction and docstrings.

In addition to running `check.py`, you can quickly check your implementation by uncommenting the relevant lines of code at the bottom of `a4.py`:

```
if __name__ == "__main__" and os.path.isdir("tests"):
    ## Tests Problem 3: Similarity
    test_directory = "tests/student_tests/"
    hello_world = load_file(test_directory + 'hello_world.txt')
    hello_friend = load_file(test_directory + 'hello_friends.txt')
    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
```

Why: `toes` against `that`: only `t` is shared, once against twice. DIFF $= 1+1+1+1+1+1 = 6$ and ALL $= 4+4 = 8$, so $1 - 6/8 = 0.25$.

Testing: run `check.py`. It covers the old tests `test_similarity_words1` to `4` and `test_similarity_letters_1` and `2`, and the four calls above.

Pen on this sheet, Shell closed, before you write Task 4 in `a4.py`. Only calculators are allowed.

Key 4.1 In `freq_dict1`, `'x'` has 3 and `'y'` has 1. In `freq_dict2`, `'x'` has 1 and `'y'` has 1. Predict the score the function should return, rounded to 2 decimal places.

Answer: .....

Key 4.2 The first list is `['to', 'be', 'to']` and the second is `['be', 'me']`. Predict the top sum, DIFF.

Answer: .....

Key 4.3 Two texts share no word at all. What should the score be?
(a) `1.0`
(b) Python stops with a `ZeroDivisionError`.
(c) `0.0`

Key 4.4 For two texts, DIFF is 2 and ALL is 7. Predict the value the function should return.

Answer: .....

5 Most Frequent Word(s) (~35 min)

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']
```

TASK 5 **Most Frequent Words** MEDIUM `a4.py` ~35 min

The answer is always a list, even when one word wins.

Your task: Implement the function `get_most_frequent_words` in `a4.py` as per the given instructions and docstring.

In addition to running `check.py`, you can quickly check your implementation by uncommenting the relevant lines of code at the bottom of `a4.py`:

```
1 if __name__ == "__main__" and os.path.isdir("tests"):
2     ## Tests Problem 4: Most Frequent Word(s)
3     freq_dict1, freq_dict2 = {"hello": 5, "world": 1}, {"hello": 1, "world": 5}
4     most_frequent = get_most_frequent_words(freq_dict1, freq_dict2)
5     print(most_frequent)      # should print ['hello', 'world']
```

Why: Combined, `hello` has $5+1=6$ and `world` has $1+5=6$: a tie, so both, in alphabetical order.

Testing: run `check.py`. It covers the old tests `test_words1` and `test_words2`, a word in one dictionary only, and a three-way tie.

Pen on this sheet, Shell closed, before you write Task 5 in `a4.py`. Only calculators are allowed.

Key 5.1 In `freq_dict1`, `'sun'` has 2 and `'moon'` has 3. In `freq_dict2`, `'sun'` has 2. The list holds one word. Predict it.

Answer: .....

Key 5.2 In `freq_dict1`, `'a'` has 2 and `'b'` has 1. In `freq_dict2`, `'b'` has 1 and `'c'` has 2. Predict how many words the list should hold.

Answer: .....

Key 5.3 In `freq_dict1`, `'zinc'` has 1 and `'iron'` has 2. In `freq_dict2`, `'gold'` has 2 and `'zinc'` has 1. Which list should the function give?
(a) `['zinc', 'iron', 'gold']`
(b) `['gold', 'iron', 'zinc']`
(c) `['gold', 'iron']`

Key 5.4 In `freq_dict1`, `'fog'` has 4. In `freq_dict2`, `'rain'` has 1 and `'hail'` has 3. The list holds one word. Predict it.

Answer: .....

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

The three functions of this section are Tasks 6, 7 and 8, one each.

6.1 `get_tf` (~25 min)

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

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 a TF of 2/3 for "hello" and 1/3
# for "world".
```

TASK 6 **Term Frequency** MEDIUM a4.py ~25 min

⌨ Load the file with `load_file`, and re-use Tasks 1 and 2.

Your task: Implement the function `get_tf` in `a4.py` as per the given instructions and docstring.

Example usage, in the Shell after you run `a4.py`:

```
>>> get_tf("tests/student_tests/hello_friends.txt")
{'hello': 0.5, 'friends': 0.5}
```

Why: `hello friends` has 2 words, each once: 1/2 each.

Testing: run `check.py`. It covers the old tests `test_tf1` and `test_tf2` (`test1a.txt`).

Pen on this sheet, Shell closed, before you write Task 6 in `a4.py`. Only calculators are allowed.

Key 6.1 A document holds `the sun and the moon and the stars`. Predict the TF of `the`.
Answer: .....

Key 6.2 A file holds `Go, go, GO! Stop.` Predict the TF of `go`.
Answer: .....

Key 6.3 Add up all the values in the dictionary `get_tf` gives, for any document. What is the total?
(a) 1
(b) The number of words in the document.
(c) The number of different words in the document.

Key 6.4 Predict how many keys `get_tf` gives for a document holding `a rose is a rose is a rose`.
Answer: .....

6.2 `get_idf` (~35 min)

For the `get_idf` function that you'll implement, you'll be given a list of file names stored in a variable named `file_paths`. 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 `file_paths`. The output should be a dictionary mapping each word to its IDF.

For example:

```
>>> 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 "hello" a IDF of 0.0. "world" and
# "friends" are in only one document each, giving them an IDF of
# 0.3010299956639812.
```

TASK 7 **Inverse Document Frequency** MEDIUM a4.py ~35 min

⌨ Count documents, not words: a document that has a word five times still counts once for it.

Your task: Implement the function `get_idf` in `a4.py` as per the given instructions and docstring.

Example usage, in the Shell after you run `a4.py`:

```
>>> get_idf(["tests/student_tests/hello_world.txt"])
{'hello': 0.0, 'world': 0.0}
```

Why: With one document, every word is in all the documents: $\log_{10}(1/1) = 0$.

Testing: run `check.py`. It covers the old tests `test_idf1` and `test_idf2` (`test1a.txt` and `test1b.txt`).

Pen on this sheet, Shell closed, before you write Task 7 in `a4.py`. Only calculators are allowed.

Key 7.1 A word is in every one of the documents. What is its IDF?
(a) `1.0`
(b) The number of documents.
(c) `0.0`

Key 7.2 There are 1000 documents, and a word is in 10 of them. Predict its IDF, as a whole number. (`math.log10(10)` is 1, `math.log10(100)` is 2 and `math.log10(1000)` is 3.)
Answer: .....

Key 7.3 There are three documents: `a a a b`, `b c` and `c c`. Predict the number of documents with the word `c` in them.
Answer: .....

Key 7.4 Which words get the largest IDF?
(a) The words in only one document.
(b) The words in every document.
(c) The words used most often in one document.

6.3 `get_tfidf` (~45 min)

For the `get_tfidf` function that you'll implement, you'll be given a file name `tf_file_path` and a list of file names `idf_file_paths`. You will need to load the file, prep the data, and determine the TF-IDF of all words in `tf_file_path`. 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"
>>> 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 the TF-IDF values. "hello" has a TF of
# 2/3 and an IDF of 0.0, giving it a TF-IDF of 0.0. "world" has a TF of
# 1/3 and an IDF of 0.3010299956639812, giving it a TF-IDF of
# 0.10034333188799373.
```

TASK 8 TF-IDF

HARD a4.py ~45 min

⌨ Call `get_tf` and `get_idf`, then sort. The order is by score first, and by word when two scores are the same.

Your task: Implement the function `get_tfidf` in `a4.py` as per the given instructions and docstring.

In addition to running `check.py`, you can quickly check `get_tf`, `get_idf` and `get_tfidf` together by uncommenting the relevant lines of code at the bottom of `a4.py`:

```
if __name__ == "__main__" and os.path.isdir("tests"):
    ## 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)]
```

Why: `hello` is in both documents, so its TF-IDF is 0.0 and it comes first.

Testing: run `check.py`. It covers the old tests `test_tfidf_small` and `test_tfidf_big` (`test1a.txt` against five texts), and checks the words of the list as well as the numbers.

Pen on this sheet, Shell closed, before you write Task 8 in `a4.py`. Only calculators are allowed.

Key 8.1 The TF file holds `x y x`. The IDF files are that file and one holding `y z`. Predict the first word in the list `get_tfidf` should give.

Answer: .....

Key 8.2 The TF file holds `pen ink`. The IDF files are that file and one holding `cup`. Predict the first word in the list.

Answer: .....

Key 8.3 The TF file holds `one two two`. The IDF files are that file and one holding `three four five six`. Predict how many tuples the list holds.

Answer: .....

Key 8.4 The TF file holds `the the the moon`. The IDF files are that file and two holding `the sea` and `the sky`. Sana says `the` is last in the list. Bilal says `moon` is last. Who is right?

(a) Bilal: `the` is in every document, so its TF-IDF is 0.
(b) Sana: `the` has the biggest TF.
(c) Neither: `the` and `moon` tie.

Key 8.5 A document has 8 words, and `moon` is 2 of them. There are 100 documents, and 10 of them have `moon`. Predict the TF-IDF of `moon`.

Answer: .....

Marking and Hand-in

Marks. Tasks 1 to 8 are worth 5 marks each, 40 in all: 2 for passing `check.py`, 2 for other texts and dictionaries (the hidden runs), and 1 for the edge cases (an empty text, one word, ties, blank lines, a word twice in one document). Task 3 must call `get_frequencies` to get full credit.

`check.py`'s points are a first mark only: it contains only a subset of the tests that will be run to determine the grade, so passing all of its tasks does not guarantee full credit on Assignment 4. After the deadline, every function is called again on four hidden cases.

Naming files. Save your solutions with the original file name: `a4.py`. **Do not ignore this step or save your file with a different name! `check.py` will not be able to find your file if you do and you will receive no marks.**

Final submission. **Be sure to run `check.py` and make sure all the tasks pass.** Then hand in with `check.py` too, from its menu. Only `a4.py` is sent. You can hand in as often as you like: the best whole hand-in counts. **Due Thu 26 Nov 2026, 11:59 PM.** Late: minus 10% for each day, up to three days. Nothing after that.

Before you hand in

- ☐ `check.py` says PASS for Tasks 1 to 8.
- ☐ The file is still named `a4.py`.
- ☐ `load_file`, and the names and parameters of the nine functions, are as the starter gave them.
- ☐ The last `if` line of `a4.py` is as the starter gave it, so your test lines run only on your machine.
- ☐ Every function returns its answer and prints nothing.
- ☐ `get_letter_frequencies` calls `get_frequencies`.
- ☐ You typed every line yourself, and can explain every line.

Supplemental Reading about Document Similarity

This assignment 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.