Introduction to Machine Learning (CS 135)
Project 02 (80 points)

We are building a word representation (i.e., TF-IDF) and looking for similar documents. This will be completed in its script, which will be autograded, while the notebook is submitted for manual grading.

Also, include the collaborator's file. First, ***preprocess*** your data, removing punctuation, non-English and non-text characters, and unifying the case (i.e., setting everything to upper- or lower-case). The aim is to generate ***feature representations*** of the text (i.e., the words, meaning, semantics): represent the strings of words in a sentence as feature vectors $\mathbf{x}_n$ of common length n. We will start by implementing functions to encode (i.e., represent our text data). Then, we use these feature representations to compare documents.

Part Zero: Collaborators file (5 points)

Provide the usual file containing your name, the amount of time you worked on the assignment, and any resources **Name the file *COLLABRATORS.txt* for credit: iNo points otherwise.**

1 Part I: Feature Representation

(80 Points) Let's explore a classic technique for representing and comparing text documents (i.e., determining a relevance score for a pair or set of documents). Work through exercise step-by-step to work towards building a program capable of discriminating between contents of text documents using a simple yet powerful approach. The sonnet collection of Shakespeare will be used. Our goal is to be able to find relevant documents automatically that are similar (i.e., imagine an online ebook store, what we are about to build could be used to make recommendations based on a user's history or personal ratings, whether as is or an extended version). Okay, let's begin...

1.1 Project Summary and Rubric

For this part, do the following:

a. (*5 pts.*) Read about argparse, and look at its implementation in the Python Script. Express your understanding in the notebook.

b. (*5 pts.*) The script already reads and cleans the text– Look over the code provided to understand what is done. Especially identify and understand the i/o operation(i.e., reading in of text file), as having to read or write from the disc is typical/ fundamental.

c. (*20 pts.*) Complete tf in text_analyzer.py so that it counts the number of word occurrences and stores the result as a list of tuples (word, count)– have each element of the list corresponds to a word w paired with its TF in a tuple. Each element of the list (i.e., each tuple) is a unique word (i.e., key) alongside its corresponding frequency (i.e., value). We can view each tuple as a sort of key-value pair. Is there another built-in Python type that we could have used instead? Sort the list of tuples according to frequency value, going from highest to lowest count.

d. (*20 pts.*) Fill in function idf to compute the above. Print and report the top 20 for the IDF of the corpus. Observe and briefly comment on the difference in top 20 lists (comparing TF of corpus vs its IDF).

e. (*20 pts.*) Fill in the script's function tfidf. Determine the TF-IDF of '1.txt' and print the 20 words with the top values. What is different with this list than just using TF? Include the printout and explanation in the notebook.

f. (*10 pts.*) Compare all documents. In other words, provided N documents, you should have $N-1$ scores for each. Visualize as a heatmap. Are there different ways to sort the classes before generating a heatmap that helps show insight?

1.2 Argparse

Open the notebook, and execute text_analyzer.py as we would from the terminal. Also, open file and check out its contents (particularly argparse).

a. Use help, Google, or other online resources to understand the argparse module. Write a short paragraph describing how argparse is used in text_analyzer.py.

b. Run the following command from the command prompt:

```
python text_analyzer.py --help
```

What does this print? Which lines of code in text_analyzer.py cause this to be printed?

```
python text_analyzer.py -i <input file or directory>
```

This can be done in the notebook by adding '¡ at the beginning of the command. For the remainder of the assignment, this is omitted (i.e., assumed you are doing this).

1.3 Term frequency (TF)

The term frequency (TF) of a word in a document F is defined as:

$$TF(w, F) = \text{number of times the word } w \text{ appears in document } F.$$

Modify text_analyzer.py to calculate the TF for words in any text document(s). The program should run by executing the following command from a terminal (or notebook):

```
python text_analyzer.py -i <input file or directory>
```

Call function on 1.txt via

```
python text_analyzer.py # since default input is set to sonnet 1 (i.e., 1.txt)
```

Print the top 20 words, and include them in the report. Does there appear to be any 'noise'? If so, where? If not, it should be clear by the end of the assignment.
The TF of a word in a *corpus* of documents $C = F_1, F_2, ..., F_k$ is defined as

$$TF(w, C) = \text{number of times the word } w \text{ appears in the corpus } C$$

$$\sum_{i=1}^{k} TF(w, F_k),$$

where k is the number of documents (i.e., corpus size).
Modify text_analyzer.py to compute the TF of all words from the entire corpus. The program should work whether called on single or multiple documents without modification (HINT: check out where a single file or multiple files get read in without modification to the code)! Now we want to pass the directory path via command-line argument as follows:

```
python text_analyzer.py -i data/text/shakespeare_sonnets/
```

Print and report the 20 most frequent words (and the corresponding frequency values) in the entire *corpus*.
Do you believe the most frequent words would discriminate between documents well? Why or why not? Just take a minute or two to think about it (no need to respond), as this is addressed in the following question. After pondering, any thoughts on how we can improve this representation?
Let's consider how we are representing a document– we can represent the collection of sonnets in such a way that summarizes the contents of files and allows us to determine a relevant score between two or more documents. However, we treat all words the same (i.e., we introduce no means of normalizing or introducing weights to words of seemingly greater importance). In fact, what makes a word important when comparing documents for a relevance score? We will next cover a means of measuring a word's importance such that it is proportional to its TF in a particular document & inversely proportionate to its TF across the entire corpus.

A closer look When comparing an entire *corpus*, the TF alone carries little discriminating power when used to measure the similarity of different text documents (in our case, sonnets). To improve our current program, let's normalize the counts such that it takes into account the commonality of each word w.r.t. to the entire *corpus*: common words (e.g., 'the,' 'a', 'to,' etc.) are, in this use-case, 'meaningless.' Thus, we want to adjust our representation of documents so that 'meaningless' words do not dominate with top scores (i.e., highest frequencies).

There are many ways to add weight; however, following a simple yet classic approach, we can adjust values so the rare words have a higher score. Thus, we will put more weight on rare attributes/features (i.e., words), yielding a more robust text relevance scoring program. For instance, if only 2 documents have the word 'baseball,' and it is mentioned in both documents a few times, then that is a better indicator of relevance than a dozen or more 'the' tokens common in all documents, such as 'meaningless' words like (e.g., 'that,' 'the,' 'for,' etc.), in essence, tells us little to nothing. This concept will be reoccurring in various topics and forms throughout this course: the discriminate power of universally common features is low; the more unique/rare features provide a more powerful representation when looking for relevance (i.e., similarities) in data.

If this is not clear, please bring it up in class and/or PIAZZA, as this concept, in the general sense, is vital for efficient data analysis. Also, check online, as there are a ton of short tutorials and write-ups on the topic– if you learn a topic of relevance that differs from these contents and/or peaks your interest, then do share. Anyways.... back to the problem at hand (i.e., let's normalize values of TF so that rarity carries more weight).

1.4 Inverse document frequency (IDF)

The inverse document frequency (IDF) is used to represent text document's word counts better and, thus, better discriminates between different documents. IDF takes into account the importance of each word, while TF considers all words to be equally important. The IDF value calculated for the entire *corpus* will then be used to adjust the values of the TF represented as an unnormalized histogram. IDF, which, again, puts more emphasis on rarer words and nearly ignores the more common words, is computed as follows:

Given a corpus of documents $C = F_1, F_2, ..., F_k$, the of a word w is given by

$$\begin{aligned} IDF(w, C) &= log \frac{\text{Number of documents in corpus } C}{\text{Number of documents in } C \text{ containing word } w} \\ &= log \frac{|C|}{|F \in C : w \in F|} \end{aligned}$$

where $|A|$ denotes the size of set A. It is important to see that IDF is a *corpus-level* representation, which we will now use to adjust our representation for a document.

1.5 Term frequency–inverse document frequency (TF-IDF)

Finally, the term frequency–inverse document frequency (TF-IDF) score can be calculated with the functions implemented up to this point. The TF-IDF is a numerical value for measuring the importance of words in a document based on its TF (i.e., in the respective document) and IDF (i.e., an entire collection of documents aka *corpus*). The intuition here is that the most frequent words in a document may or may not be important w.r.t. to other documents in a collection. Thus, if a word appears in too many other documents, it will likely be useless (i.e., what is the difference if common to all). Thus, we can now assign lower and higher scores to common and rare words.

Mathematically speaking, the TF-IDF of word w in a document $F \in C$ is given by

$$TFIDF(x, F, C) = TF(w, F) \cdot IDF(w, C).$$

The program should calculate this value when passed in '–tfidf' flag as follows (call like so to determine TF-IDF of sonnet '1.txt'): Modify text_analyzer.py

```
python text_analyzer.py -c data/text/shakespeare_sonnets/ --tfidf
```

The **cosine similarity** between two documents F_A and F_B in corpus C is defined as

$$\cos(F_A, F_B) = \frac{\sum_{w \in F_A \cup F_B} TFIDF(w, F_A, C) \cdot TFIDF(w, F_B, C)}{\sqrt{\sum_{w \in F_A} (TFIDF(w, F_A, C))^2 \cdot \sum_{w \in F_B} (TFIDF(w, F_B, C))^2}}.$$

Create a function that accepts two TF-IDF of two different documents and returns the cosine similarity score. Hint: numpy could be your friend here.

2 Submission

There are ***three*** Gradescope submission links for this assignment. You will upload the following: (1) **Collaborators information**: Submit `COLLABORATORS.txt` see specifications above; (2) **PDF** containing all the required parts for manual grading; (3) **Code** for autograder.