C4M: Medical Document Retrieval

Overview

In this project, you will be implementing a basic document retrieval system. A document retrieval system takes in a query and finds the documents most relevant to the query. Google is an example of a general-purpose document retrieval system with the set of documents consisting of the entire World Wide Web.

For our set of documents, we will use the webpages in the "infectious diseases" category from Wikipedia. When the user enters a list of keywords (e.g., symptoms), your program will search for those keywords in the contents of the disease webpages, determine which document is the most relevant based on the presence of the keywords, and then print the title of that best match.

Part 1: Loading and Pre-Processing the Data

Your goal for Part 1 is to write functions that will help you read data about flights from a file and build a dictionary with information about direct flights between cities.

Task: Pre-Processing the Data

The `.html` documents downloaded from Wikipedia are not ideal for document retrieval. They contain uppercase letters, lowercase letters, and punctuation, so the string `"Influenza,"` would not be treated the same as the string `"influenza"`. To fix this problem, we would like the individual words to only contain lowercase letters.

Complete the helper function `clean_up()`, which should read in a string and return a new one that replaces all of the uppercase letters with lowercase letters and all of the punctuation with spaces. To help you remove punctuation, we have created a variable called `PUNCTUATION` that includes all of the non-alphanumeric characters that may be included in the strings.

Task: Loading the Data

It would be possible to write your program so that each time it runs, it accesses the documents directly over the internet. But if you run your program many times, that would be a lot of

internet traffic. Instead, we have created a .zip file that already contains all the .html files for the diseases. Download the file `wikipages.zip`, extract its contents, and put it those files into a single folder in your working directory called `wikipages`. In Google Colab, you can find your working directory by clicking the `Files` icon on left side of the screen. Since there are lots of files, dragging-and-dropping the folder may or may not work. Instead, we recommend that you create a new folder (right-click in the window and then select `New Folder` to create a folder called `wikipages`), click `Upload to Session Storage` at the top to bring up a File Dialog, and then selecting all of the files at the same time using shift-click.

The starter code includes a function called `get_documents()` that we have already written for reading, cleaning, and saving the files' contents into a single dictionary. Read through the function to understand how it works. If you have implemented `clean_up()` and properly added the data to your working directory, you should be able to call `get_documents()` without any issues. Keep in mind that this dictionary is massive, so you will not be able to see if the function has worked by printing all of its contents at once. The starter code simply checks that the dictionary is not empty, but you can check the contents by printing the value associated with a given key.

Part 2: Finding the Most Relevant Document

Your goal for Part 2 is to compute how important a word is in a particular document. There are numerous ways of accomplishing this, but we will use a statistic called term frequency–inverse document frequency, or TF-IDF for short.

Calculating a Score for Each Document

For the **term frequency** component, we will simply consider whether the keyword k occurs in the document d[1]. This approach is called Boolean term frequency:

$$\mathrm{TF}(k, d) = \begin{cases} 0 & \text{if } k \text{ does not appear in } d \\ 1 & \text{otherwise} \end{cases}$$

For the **inverse document frequency** component, we will measure how common a keyword k is across all documents D. We will use the logarithmically scaled fraction of the total number of documents over the number of documents that contain the keyword:

$$\mathrm{IDF}(k, D) = \log \frac{\text{No. of docs in } D}{\text{No. of docs in } D \text{ that contain } k}$$

[1] Other variants of term frequency take into account not only whether the keyword occurs in the document, but also the number of times it occurs.

The TF-IDF score is calculated as:

$$\begin{aligned}\text{TF-IDF}(k, d, D) &= TF(t, d) * IDF(k, D) \\ &= \begin{cases} 0 & \text{if } k \text{ does not appear in } d \\ \log \frac{\text{No. of docs in } D}{\text{No. of docs in } D \text{ that contain } k} & \text{otherwise} \end{cases}\end{aligned}$$

The basic idea is that a document's score increases if a keyword occurs in that document but not in many others. For example, the score for Mumps given the keyword `parotid` should be high since `parotid` only appears in 3 other documents; however, the score for the same document given the keyword `the` should be much lower since `the` appears in all of the documents. Since our queries will consist of one or more keywords, each document's score will be the sum of the TF-IDF score on each individual keyword.

Task: Finding a Word in a Document

Recall that the TF-IDF score depends on whether or not the keyword appears in the document. Write a helper function `keyword_found()` that takes three parameters: the keyword, the document name prefix (the part of the name without the `.html`), and the dictionary of all the documents. Your function should return `True` if the keyword appears in the document and `False` otherwise. Your function should not report that the text `"Patients could be very itchy."` includes the keyword `"itch"`. There are a number of ways to solve this problem. For example, you could add spaces around the keyword or split the text into a list of words.

Task: Computing the IDF

Notice that the second case of the `TF-IDF` equation does not depend on a particular document. It uses the keyword k and all the documents D, but it is the same for every value of d. Write a helper function called `idf()` that computes the IDF component of the TF-IDF score based on a keyword and the dictionary of documents as follows:

$$\log \frac{\text{No. of docs in } D}{\text{No. of docs in } D \text{ that contain } k}$$

Here are a few hints:

- You will want to use the helper function `keyword_found()` that you wrote earlier to identify how many documents contain the keyword.
- Notice that if the keyword does not appear in any documents, the IDF score is undefined since the denominator would be zero. To make sure that your program can progress when this happens, your function should return -1 in these situations.

- To calculate the logarithm, you can use a built-in function from `numpy`[2] — one of the most popular Python libraries for numerical analysis and scientific computing. If you are running your program on a local computer, you will need to install the `numpy`; if you are using Google Colab, however, major libraries like `numpy` are already pre-installed. To actually use `numpy` in your program, you will need to include the line `import numpy` in your code (note: we have already included it above the definition of `idf()`. You can compute the logarithm by calling `numpy.log()`.

Task: Computing the TF-IDF Score

The next task is to compute the total TF-IDF score for a given query for every document in the database. We will store the TF-IDF scores for every document in a dictionary. For example, the dictionary might look something like:

```
doc_to_score = {"Typhus": 3.2, "Mumps": 5.2, ...}
```

Remember that the total score for a document on a query is the sum of the scores on each of the keywords in the query. All of the scores should be set to zero initially. For each keyword, we will increase the scores by adding in the score calculated for that keyword on that document. Looking back at our original equation, the score that we will add will either be 0 (if the keyword is not in the document) or the result from calling our function `idf()`.

You should first complete the helper function `build_empty_scores_dict()`. This function should create and return a dictionary that has each document's name as the keys and the initial TF-IDF score of 0 as the values.

Once you have done that, you should complete the function `update_scores()`. Notice that both the type contract and the description in the docstring indicate that you should update the dictionary `doc_to_score` rather than create a new dictionary. It is equally important that your function updates the score in the dictionary by adding the score for the current keyword to whatever score is already in the dictionary; this will allow you to use this function for queries with multiple keywords later on.

Being Efficient

You may remember that the result of `idf()` is only determined by the keyword and the full set of documents — not on a specific document. Depending on how you wrote `update_scores()`, you may be calling `idf()` every time you are computing the updated score for a given keyword-document pair. If your current solution does this, you should improve your code by calling `idf()` once outside of your loop, storing the result, and then using the stored value each time you need it. This is a common technique that programmers use to make

[2] https://numpy.org/

their code more efficient without changing its functionality; such a process is often known as **refactoring**.

Part 3: Putting It All Together

Now that you have written all of the helper functions, it is finally time to create a program that will read in a query from the keyboard and print the name of the most relevant document! The steps of this program are as follows:

1. Prompt the user for a query using `input()`.
2. Clean up the query by removing punctuation and converting it to lowercase using `clean_up()`.
3. Convert your query into a list of keywords. To test that your program is correct up until this point, you should print out the result.
4. Load the data using `get_documents()`.
5. Initialize a dictionary of scores using `build_empty_scores_dict()`.
6. Call `update_scores()` on each keyword to update your dictionary of scores.
7. Iterate over the scores dictionary to find the document with the highest score and print out the document name. If multiple documents tie for the highest score, you can just print one of them.

The part of this program that will take the longest to run is likely to be `get_documents()`. However, even if you want to test your program with multiple consecutive queries, you only need to load the data once. Instead of restarting your program for each new query, put the code that asks the user to enter a query and computes the result inside a loop. Keep asking for a query and returning the best match until the user finally types in the word `quit`. Be careful to reset the TF-IDF scores to 0 between queries.

Part 4: Optional Bells and Whistles

- Have the query return the top 5 matches rather than just the top match.
- Adjust the TF-IDF formula so that documents score higher if the term appears in them more than once:

  $$\text{TF-IDF}(k, d, D) = \begin{cases} 0 & \text{if } k \text{ does not appear in } d \\ (\text{No. of times } k \text{ appears in } d) \log \frac{\text{No. of docs in } D}{\text{No. of docs in } D \text{ that contain } k} & \text{otherwise} \end{cases}$$

- When you think about major search engines like Google or Bing, they usually rank websites not only according to relevance, but also popularity. However, your algorithm

is currently just as likely to suggest rare diseases as it is to suggest common ones. Using the length of the Wikipedia page as a proxy for the prevalence of a disease, adjust your program to account for how common the disease is.

- Rather than loading the document data from files that have been downloaded ahead of time, write code that downloads all of the pages linked in a Wikipedia page:
  - Infectious diseases: https://en.wikipedia.org/wiki/List_of_infectious_diseases
  - All diseases: https://en.wikipedia.org/wiki/Lists_of_diseases