C4M Workshops: Medical Document Retrieval using TF-IDF

Introduction and Overview

In this project, you will be implementing a basic document retrieval system. A document retrieval system takes in a query, and finds the most relevant documents 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 all the webpages in the “infectious diseases” category from Wikipedia. The queries will be read in from the keyboard. When the user enters a list of keywords (perhaps 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.

Downloading the Documents

It would be possible to write your program so that each time it runs, it access the documents directly over the internet. But if you run your program many times, that would be a lot of internet traffic. So instead, we have created a zip file that already contains all the html files for the diseases. Download the file `wikipages.zip` and extract it somewhere on your hard drive where you know the path.

Reading in and pre-processing the documents

The starter code `query_handout.py` contains a function for reading in all the texts. Modify the constant `HOMEFOLDER` to be the path where you stored the unzipped the html files and then try running `get_all_texts()` to understand what it returns.

The html documents downloaded from Wikipedia are not ideal for document retrieval; They contain upper- and lowercase letters and punctuation and the string `"Influenza,"` would not be a match to the string `"influenza."` To fix this problem, we would like the individual words to only contain lowercase letters. Complete the helper function `clean_up` that takes a string and returns a new string that is the same as the parameter except converted to lowercase and with all the punctuation replaced by spaces. Test your function by calling it on different input strings.

When you are convinced that your function is correct, modify `get_all_texts` to clean up all the documents by using your new helper function on every text in your dictionary.

Calculating a Score for Each Document

There are a number of different approaches to determining which document is the best match to our query. We will use one called TF-IDF which stands for “term frequency/inverse document frequency.” The basic idea is that a document’s score increases if a keyword occurs frequently in that document but decreases if the keyword is frequent in all the documents. So if one of our keywords is `parotid` (which occurs in the document for Mumps but not in all the documents), the score for Mumps would increase but the keyword `the` shouldn’t affect the score for any particular document, since it occurs in all of them.

The simplified variation of TF-IDF that we will use is easy to implement and will have some fun results, but it could be made a lot more powerful with some other improvements.

Our queries will consist of one or more keywords. The document score will be the sum of the TF-IDF score on each individual keyword. The TF-IDF score for a single keyword k for a specific document d from the full set of documents D is calculated as:

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

Does k appear in d?

Notice that the 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. Before you move on, test your function. How should you do that? You want to call your function on words that you know appear in some documents but not in others. For example, `deafness` occurs in the document for `Mumps` but not the one for `Typhus`, and `bacteria` appears in the document for `Typhus` but not in the one for `Mumps`.

Be careful here. You don't want to report that the text `"Patients could be very itchy."` includes the keyword `"itch"`. There are a number of ways to solve this problem including adding spaces around the keyword or splitting the text into a list of individual tokens. You can use any approach you like, but make sure you test your function before continuing to the next step.

Calculating IDF

Notice that the second case of the `TF-IDF` equation doesn't 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 takes a keyword and the dictionary of documents, and returns:

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

To calculate the logarithm, use the function `math.log`, which is available because we included the line `import math` at the top of the program. Call your function `keyword_found` inside `idf`. Test your function by calculating the `IDF` for various words. Try "deafness" and "fever". Then try the word "the" which appears in all the documents and "parotid" which appears in only four documents.

Try calling your functionon the string `"notfound"` which, as the value suggests, is not found in any of the documents. What happens? To avoid this error, add to your function so that it returns `-1` when the keyword does not occur in any of the documents.

Computing TF-IDF

The next task is to write a function that computes the total TF-IDF score for a given query for every document in the database. It makes sense to store the TF-IDF scores for every document (i.e., every disease) in a dictionary. For example, the final results might look something like

```
scores = {"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. So the strategy will be to use a scores dictionary where the scores are all initially set to zero. Then 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, that score is either 0 (if the keyword isn't in the document) or it is the result from calling our function `idf`.

The starter code includes the function `update_scores(current_scores, keyword, all_texts)` that you should complete. The first parameter is the dictionary mapping each document name to the score. It is this dictionary that gets updated by the function. Before you can test your function, you need a dictionary of this form to pass in as a parameter. Write another helper function `build_empty_scores_dict` that creates (and returns) a dictionary which includes an entry for every document where the key is the name of the document and the value is 0.

First test your `build_empty_scores_dict` and then use the resulting dictionary as the first parameter to test your `update_scores` function. How should you test this function? Explicitly call it on a couple of keywords for which you already calculated the IDF for some specific documents. Then check that after each function call the scores dictionary entries for those documents has been updated correctly.

Being Efficient

You may remember that the `idf` function depends only on the keyword and the full set of documents — not on a specific document. But inside your `update_scores` function, you need the `idf` result many times. It would make sense to call the helper function once inside `update_scores` to calculate this value and then store it and use the stored value each time it is needed. If your current solution to `update_scores` makes multiple calls to `idf`, improve it now.

Putting It All Together

We are finally ready to put this together into a system that reads in a query from the keyboard, and prints the name of the most relevant document. As a first step, prompt the user for a query, clean-up the query to remove punctuation and convert it to lowercase, separate the query into a list of keywords and convert all these keywords to lowercase. To test that you've done this correctly, just print out the list. Then, change your code so that instead of printing out the keywords, you call `update_scores` on each keyword. Remember that before you do this, you will need to create a initial scores dictionary (using the helper function you wrote earlier.) Finally, iterate over the scores dictionary to find the document with the highest score and print out the document name.

Test your code on different queries.

You might want to use your program to process a number of queries and it takes some time (quite a bit of time depending on the speed of your computer) to read in all the texts. So instead of re-starting your program for each new query, put the code that invites 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 enters `quit`. Be careful to reset the scores dictionary to zero between queries.

Test your code yourself and then show it off to your friends.

Bells and Whistles

Here are a few suggestions for making the system perform better.

- Adjust the formula such that if the symptom appears in the file more than once, the TF-IDF increases:

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

- Right now, very rare diseases are as likely to match the keywords as very common diseases. Using the length of the wikipidea page as a proxy, adjust the algorithm to account for how common the disease is
- Have the query return the top N matches rather than just the top match
- Write code to download all the infectious disease pages from `https://en.wikipedia.org/wiki/List_of_infectious_diseases`
- Write code to download all the disease pages from `https://en.wikipedia.org/wiki/Lists_of_diseases`