The Moogle! search engine

Proyecto de ProgramaciÃ³n I.

Facultad de MatemÃ¡tica y ComputaciÃ³n.

Universidad de La Habana.

Curso 2023

Luis Ernesto Amat CÃ¡rdenas (C-122)

Index

Data Structures

The Ranker class

Overall description

The ranker class defines several static attributes to store the data structures (mainly dictionaries) who will store the data we will extract from the files, the path to their respective cache files, among other things. The cache will be reloaded only during the instantiation.

Most utility methods just store relevant data as it is extracted in their corresponding data structure. For example the "GetWords" method only stores data in the VECTORS Dictionary, the "UpdateWC" method only stores data in the "WC" dictionary, and so on.

You will probably spot a few private methods starting with an underscore ("_") with the same name of other (public) method. Those contain the logic of the method and it was made that way to ease testing. The public method is a wrapper that usually contains I/O and/or other operations used to speed up things but have nothing to do with the main logic (funcional core).

There are some utility methods to persist data, calculate the score, get the snippet, and a couple QOL methods to query the dictionaries.

Obtaining the data

The behaviour changes according to the existance of the cache files, and the last time they were modified.

No cache files

The files are read synchronously (async I/O does not work well on Windows, it seems) and recursively. All the files that match the configured patterns are read, and given to the utility functions that vectorize the documents.

K (configured time, see CACHE_TIMEOUT attribute) minutes have passed

Load the cache and search for new files. Pass new files to the utility functions for vectorization.

else

Just load the cache.

Very short words and dirty words (ex. "demon_slayer42", "d0g") are ignored.

Storing the data

There are several dictionaries with funcional dependencies with each other. This is why we have to do a "rolling update". In other words, if one dict changes we have to modify all the others.

This is specially bad for the dict containing the tf-idf. We could mitigate it dividing it in two dicts but we would have to add another dict and the minimal speed up it's not worth it.

There are other issues. If the system fails in the middle of loading the files the cache will most likely be corrupt because of the "rolling update" mechanism. If the cache for one data structutre was dumped successfully we have no (tivial) way of knowing if the next one (that needed to be updated together with the other) was not updated.

This is mainly originated due to the fact we want to read the files only once.

Persisting the data

We simply dump it using the JSON serializer. There are several methods for handling this because C# is statically typed.

Modifying the data

The main modifications to the raw text we do it splitting it into words, cleaning useless characters like periods or commas, and changing all words to lower-case.

Returning the data

The data we are interested on are the filenames, the words, and a "score" that depends of the words we extracted.

We calculate the score using tf-idf. We also extract a snippet from the text using the GetHighlight method.

The GetHighlight method uses simple regex to match the first line where the first term of the query appears.

TopRanks class

This is basically a key-value dictionary where the key is a string and the value is a list of (string, float) two-tuples. The list is sorted at all times using the float value as a key. Why is this useful for our needs? We only need the most relevant results. In other words, if we only keep the top N files with the highest score for all the words of the documents in a dict we only need to grab them whenever one of these words appears in a query.

It has a few QOL methods to get all the keys, get a value ignoring KeyError, etc.

It also has its own cache. The filename can be defined passing a parameter to the constructor.

As a side note. This class was generic enough to be used by the DiffLib.cs module.

Difflib.cs classes

SequenceMatcher

Same API as the orignal. The original documentation is great so I will just give a quick rundown of the process and the behaviours/implementations details I decided to change.

To calculate the ratio of similarity we call the GetMatchingBlocks method. This is wrapper who calls the recursive _GetMatchingBlocks private method who works repeatedly applying the FindLongestMatch method.

The recursive method calls FindLongestMatch. Depending of the size and location of the match it will call itself to see if there are other matches outside the interval where we found the match. This can be better understood by looking at the code and reading the comments.

The FindLongestMatch method works by iterating the first string and comparing each character with those of the other chain (this is optimized with an indexer dict). When it finishes iterating, we (should) have the coordinates where the match start and the length of the mathch. Then, we perform a lookahead and a lookbehind to see if we missed any common character.

Then we sort the blocks we got from GetMatchingBlocks by block size and use it to calculate the ratio of similarity (the formula it's just [matches * 2/ len(a) + len(b)])

Utils.GetCloseMatches

We simply calculate the ratio of similarity of the word given and a list of possibilities. We use the TopRanks class to only keep track of the N better matches (N is 1 because of the limitations of the front-end).

How the data structures interact with each other (flow of the program)

We can see this is the Moogle.cs file which is the main wrapper. The main class instantiates the Ranker class every time the user submits a query, updating the cache if required. We grab a list of candidates the Ranker class stores in Ranler.TopRanked (if we fail to get any candidate we use Util.GetCloseMatches to return anything resembling the word the user inputed), we get the snippet, and we pass the array of SearchItems we created to the SearchResult class.

The SearchResult class sorts the items and hands them over to the front-end.

Configurations

See Settings.cs. Here you can configure the base directories (it's not a good idea to change this). Regardless, most classes have static parameters that can be safely edited like the filenames for the cache files or how many results we want per word in the query (default is 5), etc.