Homework 4: Information Retrieval
Moogle – My Awesome Google

CS 201 Data Structures II
Habib University
Spring 2020

Due: 1830h on Monday, 6 April

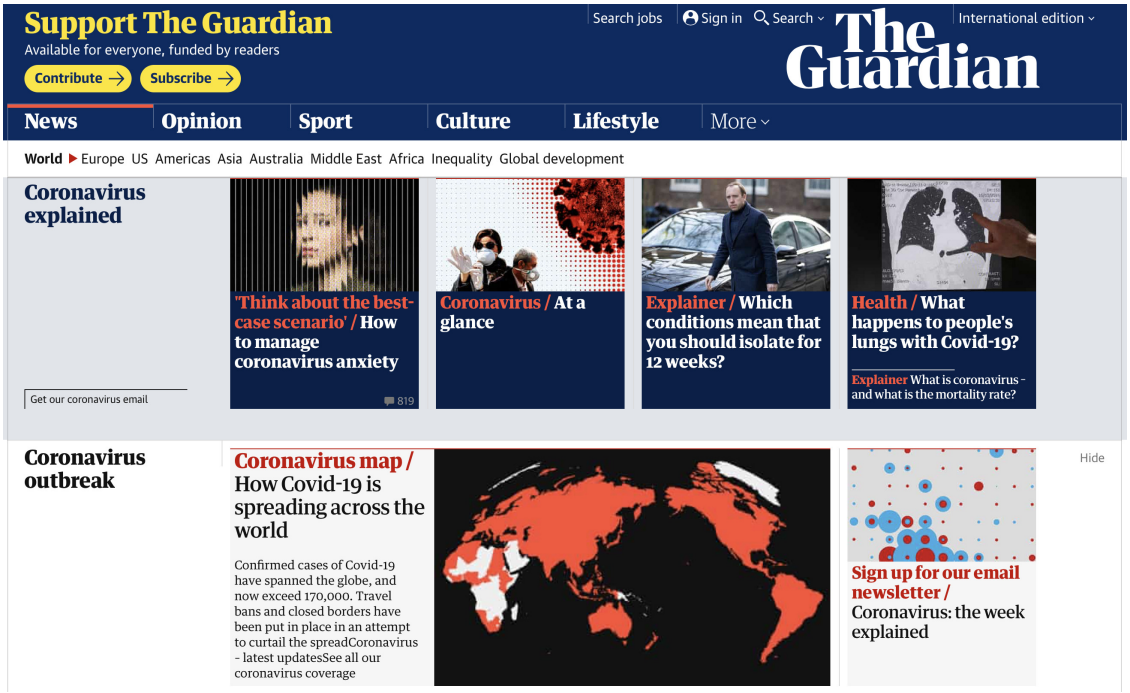


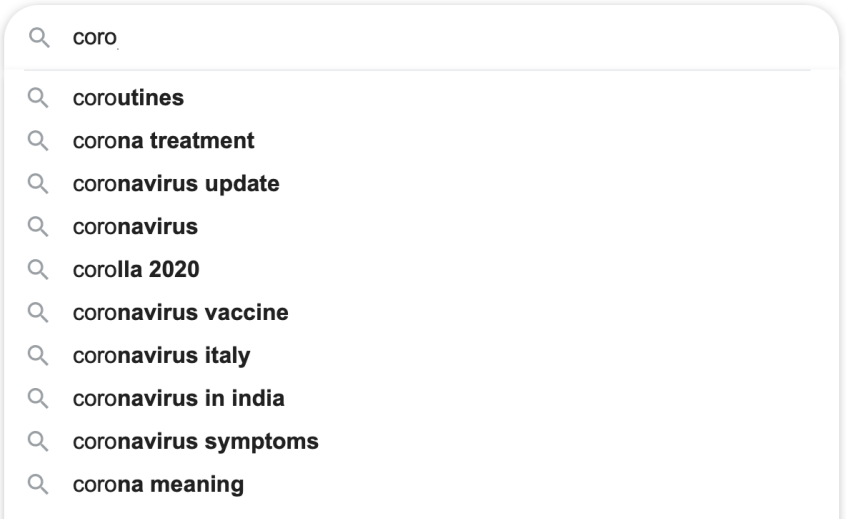
Figure 1: Coronovirus Outbreak | The Guardian, accessed Sunday, 22 March, 2020.

In this assignment you will build Moogle (My Google), a system to perform information retrieval tasks on a corpus. Specifically, Moogle will perform 2 tasks.

1. Given a query, find completion matches for it from the corpus. For example, see Figure 2a.
2. Given a query, retrieve a list of documents from the corpus ranked according to their relevance to the query.

The first task is supported by building a trie with all the words in the corpus. The second is supported by an inverted index built from all the documents from the corpus. You will write the necessary functionality without building the interface.

The corpus consists of 3028 news articles published between 1 January, 2020 and 21 March, 2020 matching the keyword, "coronavirus", and retrieved through the API of The Guardian.



(a) An example of auto-complete suggestions from `https://www.goole.com`.

(b) Not this Moogle!

Figure 2: What we want and not necessarily want from Moogle.

1 Tokenization

An important operation in this context is tokenization which breaks a long string into smaller strings or *tokens* which are more appropriate for the application. There is no *correct* or *standard* tokenization, rather different applications require the string to be tokenized differently. This operation features repeatedly in our implementation.

2 Implementation

The accompanying implementation provides the following classes and supporting helpers.

`Corpus` It encapsulates a `Trie` and an `Index` and supports completion and search queries as described above by delegating to the appropriate member structure. A `Corpus` instance is initiated with the path to a directory containing the documents to be processed. The documents must be in `.txt` format.

`Document` A representation of a document in the corpus. It processes a `.txt` file and offers it in a manner suitable for the other structures. It uses the helper, `document_tokenize`, to tokenize the content of the document. You may leave the helper unchanged initially. Each `Document` instance also stores an ID in order to uniquely identify the document from which it derives.

`Location` Represents the location of a word in the corpus. A word may appear in multiple locations in the same document and in multiple documents. The `Location` class stores document ID and the start and stop indexes (as per python slicing convention). Many of the functions in the provided implementation that deal with words also expect an accompanying list of `Location` objects.

`TrieNode` A node in the trie. It simply contains a dictionary (hash table) that maps letters in the alphabet to children. Note that the node does not store a label.

`Trie` A `Trie` contains its root node and 2 methods. It uses `^` as the string terminating character. Labels are stored not in the nodes but in the edges, or more specifically as the keys in the `children` dictionary

at each node. The node corresponding to the last letter of a word has an entry in its dictionary with the terminator as key and the list of *Location*s as value.

- The `add_doc` method adds a document to the trie. Each word from the document is first preprocessed using a helper, `trie_preprocess`, which you can initially leave unchanged, and the result is then added to the trie using another helper, `add_word`, which you have to implement.
- The `complete` method returns matches from the trie to given prefixes which are passed in as a single space separated string. It uses the helper, `prefix_tokenize`, to tokenize the received string into individual prefixes. You can initially leave this helper unchanged. The actual matching uses another helper, `match`, which you have to implement.

`Index` The inverted index supports TF-IDF matching of a query to the stored documents.

- The `add_doc` method adds a `Document` to the index and has to be implemented by you. Before addition to the index, each word from the document should be pre-processed by the helper, `index_preprocess`, which you can initially leave unchanged.
- The `query` method takes a single space separated query string to return a ranked list of document IDs. It should be implemented by you. The query string should first be tokenized using the helper, `query_tokenize`, which you may initially leave unchanged.

3 Tasks

You have to implement the following functions.

- the `add_word` helper to `Trie`
- the `match` helper to `Trie`
- the `__init__` method of `Index`
- the `add_doc` method of `Index`
- the `query` method of `Index`

For better results (see Section 4), you may also have to modify the following helpers.

- `document_tokenize`
- `trie_preprocess`
- `prefix_tokenize`
- `index_preprocess`
- `query_tokenize`

There should not be a need to modify any of the other code.

4 Fine Tuning

Congratulations, you have implemented your (very first) search engine! Be proud and play around with Moogle. Go over some of the documents, perform some searches, verify them, try out some completion results, and so on.

In so doing, you will begin to realize some quirks. You may come across strange characters (these are due to unhandled Unicode characters in the original documents). Stop words will pop up. Punctuation is

not correctly handled. Some of the original documents are also strange–they contain little to no content, more strange characters. All of this is common in information retrieval.

In this section, you will refine your search engine to become even more awesome!

Make a new branch in your repository called `awesome` to include your refinements. The `master` branch will contain code to meet basic functionality and pass Travis tests. Add a section called *Refinements* at the end of the `README` in the `awesome` branch where you will list all the refinements that you incorporate. There are no specific instructions for these refinements and you are free to implement what you like. Some refinements are a matter of personal taste, so for each refinement include a line or two to explain why you think that refinement improves your search engine.

4.1 Document Cleaning (Garbage In Garbage Out)

Your results are only as good as your input and the quirks mentioned above are typical problems faced in Information Retrieval. That is why significant effort is spent on *document cleaning*, i.e. reading or processing the documents to a form appropriate for further processing. In our setup, this will be handled by the `document_tokenize` helper. Make appropriate refinements to `document_tokenize` so that documents are better handled. Make sure to include each incorporated refinement in the `README`.

Stop Words and Punctuation How should your system handle stop words and punctuation? The usual practice is to leave them out. You may implement this functionality in `document_tokenize` or in the individual tokenization functions of `Trie` or `Index`.

Stemming Should documents containing the word "doctors" match a query for "doctor"? How about "isolate" and "isolation"? Should "driving" appear as a completion for "drive"? The usual answer is "yes". These pairs of words are said to have the same *stem* and reducing a word to its stem is called *stemming*. You can best decide at what level to perform stemming–at the document level, for the trie, or for the index–and modify the corresponding functions. Make sure to justify your choice in your *Refinements* section in the `README`.

Others How about case sensitivity, words with apostrophe, e.g. "don't", how to handle quotation marks, and initials, e.g. "George W. Bush"? Feel free to incorporate related refinements and explain them in your `README`.

Even More The next level of search is "semantic search" where matching takes into account not only keywords but also their *meaning*, e.g. the system can distinguish between "who" and "WHO", between "pen", the writing instrument, and "pen", the holding area for animals. Such pairs of words are called *homonyms* and are one of the many exciting challenges that Information Retrieval deals with. There is no expectation from you to address them in this assignment.

`nltk` As we see above, Information Retrieval has strong overlaps with Natural Language Processing (NLP). As such you may find the *Natural Language Toolkit (ntlk)* in python to be especially useful as you refine Moogle.

Word that should match may not match. Some obvious results may seem missing.

Credits

This homework and related files are courtesy of Muhammad Qasim Pasta and Abdullah Zafar