Homework 4: Information Retrieval

L5-Group-3

CS 201 Data Structures II
Spring 2023

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 and a corpus, find completion matches for the query from the corpus. For example, see Figure 1a.
2. Given a query and a corpus, 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 in the corpus. You will correspondingly write and implement 2 classes: `Trie`, and `InvertedIndex`.

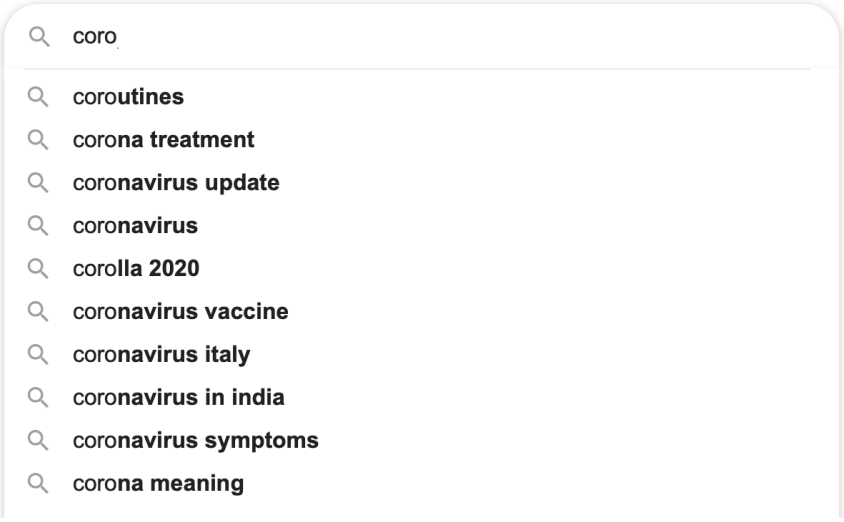
1 Class Details

`Corpus` This class encapsulates a `Trie` instance and an `InvertedIndex` instance in order to support completion and search queries on a corpus as described above by delegating to the appropriate member structure. A `Corpus` instance is initiated with the path to a ZIP file containing the documents to be processed. The documents are text files which may or may not have a `.txt` extension. The unzipped directory may or may not contain sub-directries. The text files may be at the root level of the unzipped directory or in sub-directories. The corpus must be able to find and process all contained documents regardless. The ID of each document is its path relative to the unzipped directory. Some example corpora are listed in Section 3 for your testing. The class offers `prefix_complete()`, `query()`, `and_query()`, and `or_query()` methods by delegating to the appropriate member. The details of these are given below.

`Document` A representation of a document in the corpus. It processes a text file and offers it in a manner suitable for the other structures. Each `Document` instance also stores an ID in order to uniquely identify the document from which it derives.

`Trie` This class represents a trie (standard or compressed, your choice). Specifically, an instance of this class is used by `Corpus` to implement the `prefix_complete()` method. This class offers a method of the same name which behaves as follows. It accepts a `string` argument which is the `prefix` for which completions from the corpus are sought. It returns a `dict` in which each key is a completion from the corpus and the corresponding value is a `list` of 3-`tuple`s representing the *location* information of the completion. That is, it contains the ID of the document that contains the completion and the starting and ending indexes of the completion in the document. Indexes start from 0.

`InvertedIndex` This class represents an inverted index. Specifically, an instance of this class is used by `Corpus` to implement the `query()`, `and_query()`, and `or_query()` methods. This class offers methods of the same name which behave as follows. The `query()` method accepts a `string` and an `int` argument representing the `query` term and the number of desired results. Note that query may be a space separated list of multiple query terms in which case all of the contains terms form the query. It returns a sorted `list` of 2-`tuple`s (or *pairs*) representing the ranked list of documents. That is, each pair contains the rank and corresponding document ID. Ranking is according to relevance of the document with the query. The most relevant document is ranked 1, the next most relevant is ranked 2, and so on. Relevance is to be computed using TF-IDF scores. The result list includes the top-`k` results only. The `and_query()` method accepts two `string`s and an `int` arguments which represents `query1`, `query2` terms and the number of desired results. It returns the intersection of the ranked list of documents for `query1` and `query2`. The `or_query()` method accepts two `string`s and an `int` arguments which represents `query1`, `query2` terms and the number of desired results. It returns the union of the ranked list of documents for `query1` and `query2`.



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

(b) Not this Moogle!

2 Tasks and Implementation

The `Corpus` and `Document` classes have already been implemented for you in `src/corpus.py` and `src/document.py`. You have to implement the classes `InvertedIndex` and `Trie` in `src/index.py` and `src/trie.py`.

In the `InvertedIndex` class, you have to implement the following methods:

- `query(self, terms:str, k: int) -> List[Tuple[int,str]]`
- `and_query(self, query1:str, query2:str, k:int) -> List[Tuple[int,str]]`
- `or_query(self, query1:str, query2:str, k:int) -> List[Tuple[int,str]]`

In the `Trie` class, you have to implement the following methods:

- `prefix_complete(self, prefix:str, node:TrieNode = None, word: str = '') -> List[Tuple[int,int,str]]`

You may implement any helper functions that you want but there names should begin with an underscore.

2.1 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. You can use this operation to break a document into terms.

2.2 Testing

Once you have successfully implemented your classes, you can test your code by applying it to the sample corpora listed below. You may create some smaller copora of your own for initial testing. For grading purposes, your submission will be tested automatically on GitHub using `pytest test_index.py test_trie.py` . The test files will import `src/corpus.py`. Optimize your code so as to meet the `pytest` limit of 1 minute. A timed out test is a failed test.

2.3 Allowed modules

As you have found out, `pytest` on GitHub fails if your code `import`s arbitrary modules. The allowed modules for this assignment are `pathlib` (doc, RP), `zipfile` (doc, RP), and `nltk` (doc, RP). Modules that are part of python by default, e.g. `math`, can also be used.

3 Corpora

You are free to use any corpus of your choice. Google Dataset Search and Kaggle are excellent resources for datasets. You may create your own corpus as well. Below are details of some specific datasets.

1. "The *20 Newsgroups* data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups." More details including a download link are available here.

2. The *StackSample* dataset contains text from 10% of Stack Overflow questions and answers on programming topics. Further details including a download link are available here.

4 Some Information Retrieval Rambling

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.

This section lists some refinements to make Moogle even more awesome! The tasks in this section are **not required and do not carry marks**. They are listed as suggestions for your own tinkering pleasure!

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. pre-processing the documents to an appropriate form. This usually involves the following.

Stop Words and Punctuation How should your system handle stop words and punctuation? The usual practice is to leave them out.

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.

Others How about case sensitivity, words with apostrophe, e.g. "don't", how to handle quotation marks, and initials, e.g. "George W. Bush"?

4.2 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.

`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.

Credits

This homework and related files are due in part to Muhammad Qasim Pasta and Unaiza Ahsan.