Algorithmic Methods of Data Mining

Homework 3

Due: 3/12/2017, 23:59.

Instructions

You must hand in the homeworks electronically and before the due date and time.

The third homework will be performed in **groups**. If you do not know who are your group partners, email Adriano.

Handing in: You must hand in the homeworks by the due date and time by an email to `fazzone@diag.uniroma1.it` that will contain as attachment **(not links to some file-uploading server!)** a .zip file with your answers. The filename of the attachment should be
`AMD_Homework_3__StudentID_StudentName_StudentSurname.zip`;
for example:
`AMD_Homework_3__1235711_Robert_Anthony_De_Niro.zip`.
The email subject should be
`[Algorithmic Methods for Data Mining] Homework_3 StudentID StudentName StudentSurname`;
For example:
`[Algorithmic Methods for Data Mining] Homework_3 1235711 Robert Anthony De Niro`.
After you submit, you will receive an acknowledgement email that your project has been received and at what date and time. If you have not received an acknowledgement email within 2 days after you submit then contact Adriano.

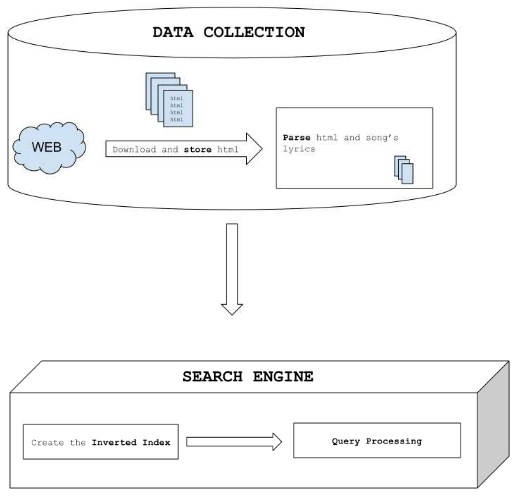
The solutions for the theoretical exercises must contain your answers either typed up or hand written clearly and scanned.

For information about collaboration, and about being late check the web page.

The homework consists in analyzing the text of songs and building a search engine for songs. Here a little image, which summarizes the steps you should perform for the search engine.

Data collection.

For this step you need to write a program called collect.py,which performs a data fetching from the web. The collect program downloads songs from the AZLyrics website. Here a little sketch, which summarizes the steps you should perform for the search engine.



1. Download the web pages of the songs in page `https://www.azlyrics.com`. Start by the letter A and download as many songs as possible, but at least 30K. Keep the pages locally at your disk, in this way you should not have to download them again, if needed. Also they will be required for displaying the results.

   Remark. It is very important to wait for a random period of 1–2 seconds between downloading two pages, otherwise the website may block you (hint: take a look at `time.sleep()`).

2. Parse the downloaded pages and for each extract the lyrics, artist, title, and the url of the song.

3. Store the parsed songs as documents in a MongoDB database, one document per song.

Song statistics.

First we will perform some basic statistics to understand the songs.

1. **Identify Artist with most songs and create a histogram of the number of songs per Artist.**

   Looking at the data can you assume or infer something intuitively? For instance, has X written more songs the Y, because X has been composing music since 1996? This kind of information does not exist in the dataset you have, but if you find something interesting looking at the data you can do extra research online to see whether certain trends might describe phenomena that, in some way, affect your data.

2. **Identify the 20 most popular words (exclude stopwords) and comment** (hint: if the most used words are love, fun and great, what does it mean?)

3. **Identify the 10 most common singer names (e.g, "Alice," "Bob," "Frank") and see whether singers whose name is the same tend to publish more songs than others** (hint: can you visualize it?)

4. **Create a histogram of song lengths** (total words per song—include stopwords).

Search engine. Here we create two programs: *Index* and *Search*. These two allow to build up a Search Engine that answers Boolean queries and using as scoring function the (tf–idf cosine similarity). In particular, it returns the top-10 documents.

The *Index* program builds an inverted index over the downloaded documents. Documents should be pre-processed to remove stopwords, punctuation, to normalize the vocabulary, and stem. For this purpose, you can use the `nltk` library).

Create a collection index containing one document (the vocabulary) with all the terms (format: `term_id: term`) and for each term as a separate document in the following way:

```
{term_id_1:[(document1, tf_{term,document1}), (document2, tf_{term,document2}),]}
{term_id_2:[(document1, tf_{term,document1}), (document2, tf_{term,document2}),]}
```

where `document1` is the id of the first document that contains the term corresponding to `terma_id_i` in its lyrics, and `tf_{term, document}` is the term frequency of the term in the document. Of course, the document frequency corresponds to the length of the posting list.

The *Search* program receives as an argument a set of words (e.g. love sun). Then, it should read the index in memory and ask the user to perform two types of queries. For both types, to perform the queries, you should use the algorithm that we performed in class.

Type 1. Return the top-10 most relevant documents according to tF-idf-cosine scoring (or all the documents with a nonzero score, if the results are fewer than 10). Use a heap data structure (you can use Python libraries) for maintaining the top-10 documents.

Type 2. Here we want conjunctive queries (AND), in which case it should return all the songs that contain all the query terms. However, we want to post process the results as some search engines do (e.g., `yippy.com`), and return *clusterized* results. It means that when you enter the query, the returned documents are represented in clusters. Since you want the *best ever* search engine, you decide to implement this capability. How to do it?

Once you obtain the matching songs you give the number of results to the user, and you ask her for a value of k. Subsequently, you clusterize the results into k clusters, and you give them as output for the query. You can choose the clustering methodology Each document can be represented as a normalized tf-idf vector (such that its length is 1) in the space of the terms of the vocabulary. Then, you can use as metric to define the distance among documents using the Euclidean distance. For each cluster display a word cloud of the most common terms in the cluster, and a list of the artists and songs.

Note: You have some freedom about how to store the songs to make your clustering efficient.