# of Data Mining

Homework 2

Due: 13/11/2022, 23:59

Instructions

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

This homework has to be done by each **person individually**.

Handing in: You must hand in the homework by the due date and time by an email to Gianluca (`decarlo.1805894@studenti.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 `DM_Homework_1__StudentID_StudentName_StudentLastname.zip`;
for example:
`DM_Homework_1__1235711_Robert_Anthony_De_Niro.zip`.
The email subject should be
`[Data Mining] Homework_1 StudentID StudentName StudentLastname`;
For example:
`[Data Mining] Homework_1 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 Gianluca.

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.

Problem 1. For this question you have to implement a search engine for the "Top Anime Series" from the list of MyAnimeList. It has several parts that you need to implement. For the linguistic analysis you can use the NLTK Python library.

1. **Get the list of animes**: we start from the list of animes to include in your corpus of documents. In particular, we focus on the top animes ever list. From this list we want to collect the url associated to each anime in the list. The list is long and splitted in many pages. We ask you to retrieve only the urls of the animes listed in the first 20 pages (each page has 50 animes so you will end up with 1000 unique anime urls). The output of this step is a **.txt** file whose single line corresponds to an anime's url.

2. **Crawl animes**: after Once you get all the urls in the first 20 pages of the list, you:

   - Download the html corresponding to each of the collected urls.
   - After you collect a single page, immediately save its **html** in a file. In this way, if your program stops, for any reason, you will not lose the data collected up to the stopping point.
   - Organize the entire set of downloaded **html** pages into folders. Each folder will contain the **htmls** of the animes in page 1, page 2, ... of the list of animes.

3. **Parse downloaded pages**: At this point, you should have all the html documents about the animes of interest and you can start to extract the animes informations. The list of information we desire for each anime and their format is the following:

- Anime Name (to save as animeTitle): String
- Anime Type (to save as animeType): String
- Number of episode (to save as animeNumEpisode): Integer
- Release and End Dates of anime (to save as releaseDate and endDate): Convert both release and end date into datetime format.
- Number of members (to save as animeNumMembers): Integer
- Score (to save as animeScore): Float
- Users (to save as animeUsers): Integer
- Rank (to save as animeRank): Integer
- Popularity (to save as animePopularity): Integer
- Synopsis (to save as animeDescription): String
- Related Anime (to save as animeRelated): Extract all the related animes, but only keep unique values and those that have a hyperlink associated to them. List of strings.
- Characters (to save as animeCharacters): List of strings.
- Voices (to save as animeVoices): List of strings
- Staff (to save as animeStaff): Include the staff name and their responsibility/task in a list of lists.

For each anime, you create an **anime_i.tsv** file of this structure: "animeTitle \t animeType \t ... \t animeStaff". If an information is missing, you just leave it as an empty string.

4. **Search Engine**: the next step is to build a search-engine index. First, you need to build an inverted index, and store it in a file. Build an index that allows to perform proximity queries using the cosine-similarity measure. Then build also a query-processing part, which, given some terms it will bring the most related animes. For this version of the search engine, narrow the interest on the **Synopsis** of each anime. It means that you will evaluate queries only **with respect to the anime's description**. For each most related anime computed by your search engine return its **animeTitle**, **animeDescription**, and **url**. If your search engine returns poor results try by increasing the number of crawled pages. If you want to crawl a large number of pages, to speed up the operation, I suggest you to work in parallel with your group's colleagues or even generate code that works in parallel with all the CPUs available in your computer. In particular, using the same code, each component of the group can be in charge of downloading a subset of pages (e.g., the first 100). PAY ATTENTION: Once obtained all the pages, merge your results into an unique dataset. In fact, the search engine must look up for results in the whole set of documents.

Hand in the code, along with some examples of queries and screenshots of the results. Try short queries, such as a few terms, as well as long queries, which can be other announcements.

Problem 2. Here we are asking to implement nearest-neighbor search for text documents. You have to implement shingling, minwise hashing, and locality-sensitive hashing. We split it into several parts:

1. Implement a class that, given a document, creates its set of character shingles of some length k. Then represent the document as the set of the hashes of the shingles, for some hash function.

2. Implement a class, that given a collection of sets of objects (e.g., strings, or numbers), creates a minwise hashing based signature for each set.

3. Implement a class that implements the locally sensitive hashing (LSH) technique, so that, given a collection of minwise hash signatures of a set of documents, it finds the all the documents pairs that are near each other.

To test the LSH algorithm, also implement a class that given the shingles of each of the documents, finds the nearest neighbors by comparing all the shingle sets with each other.

We will apply the algorithm on the animes previously crawled. We want to find animes that are near duplicates, *i.e.* that have similar descriptions. We will say that two animes are near duplicates if the Jaccard coefficient of their shingle sets is at least 80%. We will use shingles of length 10 characters. Find values for r and b (see Section 3.4 in the book) that can give us the desired behavior. To plot the graph that gives the probability as a function of the similarity for different values of r and b you can use, for example, Wolfram Alpha.

To apply the algorithm you have the following tasks:

1. Find the near-duplicates among all the animes of Problem 1 using LSH. Use the *Synopsis*.

2. Find the near-duplicates among all the animes of Problem 1 by comparing them with each other.

3. Report the number of duplicates found in both cases, and the size of the intersection.

4. Report the time required to compute the near duplicates in either case.

You will need a way to create a family of hash functions. One way is to use a hash function and a code similar to the following.

```
# Implement a family of hash functions. It hashes strings and takes an
# integer to define the member of the family.
# Return a hash function parametrized by i
import hashlib
def hashFamily(i):
  resultSize = 8        # how many bytes we want back
  maxLen = 20           # how long can our i be (in decimal)
  salt = str(i).zfill(maxLen)[-maxLen:]
  def hashMember(x):
    return hashlib.sha1(x + salt).digest()[-resultSize:]
  return hashMember
```

Note that this code is an overkill because we use a cryptographic hash function, which can be very slow, even though it is not needed to be as secure. However, for the necessities of the homework we will use it to avoid having to install some external hash library.

Problem 3. Implement Problem 2 using Apache Spark.