COMP3074 Human-AI Interaction
Lab 2: Information Retrieval, Representation and Similarity

Jérémie Clos

October 2023

1 Introduction

In this lab, you will build a search engine. It will be quite inefficient, and it will certainly not rival Google, but it will give you insights in the fundamental operations happening anytime you are using a search-based algorithm: search engines, recommender systems (Amazon, Spotify, Netflix), etc.

2 The tools of the trade

2.1 Making sure Numpy and Scipy are installed

Two fundamental Python libraries are typically used to implement linear algebra and scientific computing functions: Numpy (for numerical methods and linear algebra) and Scipy (for scientific computing). If you have installed Anaconda they are already on your system and you just need to import them as such:

```
import numpy
import scipy
```

One of the advantages of numpy is that it gives very fast operations on matrices, which you can access through its array data structure.

2.2 Pickling objects

Python has a handy set of functions to save and load objects called *pickle* and a newer way of doing things through *joblib*. Both work, but joblib is slightly better for large machine objects. Here, we refer to any sort of instantiated data structure as an object. It could be a language model, a machine learning model, or whatever you fancy.

With **pickle**:

```
import pickle
with open("filename.pickle", "wb") as f:
    pickle.dump(someobject, f)
with open("filename.pickle", "rb") as f:
    someobject = pickle.load(f)
```

With **joblib**:

```
from joblib import dump, load
dump(someobject, 'filename.joblib')
loaded_object = load('filename.joblib')
```

This is important if you are building an application that makes use of a complex structure like a classifier or an index because you don't want to have to re-compute it. **This code is not meant to run as is; it is provided as an example of two functions that can help you with your work.**

3 The term document matrix

3.1 Building an index

The fundamental data structure underlying a search engine is the term-document matrix. The term-document matrix is simply the transpose of a document-term matrix: instead of caring about which terms appear in a document, we care about which documents contain a specific term. That being said, it only matters when dealing with some search index optimisation, and both Term-Document and Document-Term matrix are mostly equivalent in the current form, so feel free to flip the axes around in your head if it helps you understand it better. I simply used Term-Document instead of Document-Term because it is an information retrieval usage. One axis of the TD matrix is the vocabulary (the terms we are modelling), while the other axis is the document collection.

The vocabulary is represented in Table 1, and the corresponding term-document matrix is represented in Table 2.

TermId	**Term**
t1	Term1
t2	Term2
t3	Term3
t4	Term4

Table 1: Vocabulary

Term	**Doc1**	**Doc2**	**Doc3**	**Doc4**
t1	frequency(t1,d1)	frequency(t1,d2)	frequency(t1,d3)	frequency(t1,d4)
t2	frequency(t2,d1)	frequency(t2,d2)	frequency(t2,d3)	frequency(t2,d4)
t3	frequency(t3,d1)	frequency(t3,d2)	frequency(t3,d3)	frequency(t3,d4)
t4	frequency(t4,d1)	frequency(t4,d2)	frequency(t4,d3)	frequency(t4,d4)

Table 2: Term-Document matrix

In real life, storing such thing in a fixed-size matrix would be very wasteful because of its sparsity (large number of 0's), and specific data structures are built for sparse matrices. However for the sake of learning, we can handle a bit of waste of resources.

The construction of the vocabulary list goes through the same steps as we have seen in text classification: words need to be tokenized, and then standardised based on a preferred approach (case folding, lemmatisation/stemming). Even more similarly, stop words and low/high frequency words can also be removed. A word that is present in 99% of documents is useless for search, since searching for it should return all documents.

Finally, the frequency of each term in each document is also usually weighted in a very similar way than what we saw in the last lab. The TF-IDF term weighting scheme actually comes from the information retrieval literature.

3.2 The Query

As far as a search engine is concerned, a query is just another document. As such, when a search query is entered by a user, it is simply mapped on the vector space that was induced from the vocabulary of the document collection.

4 Relevance and similarity

The key function of an information retrieval system is the computation of the **relevance** of a document with respect to a query. Because the query is considered to simply be a new document, the relevance of a document with respect of a document can simply be approximated by a **similarity** measure. Of course, in real search engines, there is more at play, but similarity is more than enough for our current use case.

Term	**Doc1**	**Doc2**	**Doc3**	**...**	**Query document**
t1	frequency(t1,d1)	frequency(t1,d2)	frequency(t1,d3)	...	frequency(t1,q)
t2	frequency(t2,d1)	frequency(t2,d2)	frequency(t2,d3)	...	frequency(t2,q)
t3	frequency(t3,d1)	frequency(t3,d2)	frequency(t3,d3)	...	frequency(t3,q)
t4	frequency(t4,d1)	frequency(t4,d2)	frequency(t4,d3)	...	frequency(t4,q)

Table 3: Term-Document matrix with query

The goal of a search engine is therefore to rank all the document by decreasing relevance, where relevance will be defined as the **cosine similarity** between the query document and each document.

4.1 Cosine similarity

The cosine similarity between the query document and a document d is the cosine of the angle between the document vector and the query vector in the vector space induced by the vocabulary, and it is defined as follows:

$$\text{similarity}(q, d) = \frac{q \cdot d}{\|q\| \cdot \|d\|} = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \sqrt{\sum_{i=1}^{n} d_i^2}} \tag{1}$$

In equation 1, q_i and d_i correspond to the i^{th} component of the query and document vector. For example, q_2 corresponds to the frequency of term 2 in the query document. n corresponds to the number of components in the vector, that is, the size of the vocabulary. Terms of the query that are not in the vocabulary are ignored.

You are free to implement the cosine similarity by hand, as it is not very complicated, but some libraries make it a lot easier. For example, Scipy has the cosine distance implemented.

```
from scipy import spatial

query_doc =  [1, 2, 0, 0]
document_1 = [2, 4, 0, 0]
sim_1 = 1 - spatial.distance.cosine(query_doc, document_1) # takes the inverse of
    the distance in order to get similarity
```

Numpy, on the other hand, gives you the tools to implement it yourself much more easily.

```
from numpy import dot
from numpy.linalg import norm

query_doc =  [1, 2, 0, 0]
document_1 = [1, 5, 0, 0]

sim_1 = dot(query_doc, document_1)/(norm(query_doc)*norm(document_1))
```

Note that dot is the dot product that you can see in the middle part of equation (1): $q \cdot d$, while norm represents l_2-norm, symbolised as $\|x\|$.

5 The search

So now that you have the relevance function (cosine similarity), the index (term/document or document/term matrix), you can actually build a search engine! This may seem daunting, so let us review the components involved in this:

5.1 General algorithm

Here is the general algorithm that you should follow for indexing:

```
Compute the vocabulary of the corpus
Use the vocabulary to compute the term-document matrix of the corpus
Apply term weighting technique to that term-document matrix
```

And the general algorithm for search:

```
Apply tokenising, stemming, and other normalisation techniques on search query
Map search query on the document collection-induced vector space
Apply term weightign technique to the tokens in the search query
For each document in the document collection:
  Compute the cosine similarity with the search query
Rank documents by decreasing similarity
```

5.2 Parsing a list of documents to build the index

In Lab 1 we saw how to read a file.

```
with open('document.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(line)  # print the current line
```

And I alluded to using the `os` library (part of the standard library in Python) to parse a folder full of documents.

```
import os

document_path = "data"  # We assume the documents are stored in a folder named
    data, positioned in the folder where your Python code is

corpus = {}
for file in os.listdir(document_path):
    filepath = document_path + os.sep + file
    with open(filepath, encoding='utf8', errors='ignore', mode='r') as document:
        content = document.read()
        document_id = file
        corpus[document_id] = content
```

Now, our documents are all stored in a Python dictionary and corpus and are ready to be analysed. Of course that would be a terrible way to build a large-scale search engine, but it is not really what we are doing now.

As for tokenising, case folding and general text standardisation, you are free to either roll out your own implementation (it is easier than it looks) or reuse some of the ones we saw in lab 1, using NLTK.

Note: you can use pickle and joblib (see section 2.2) to save and load your index so that you do not have to recompute it every time you run your program.

5.3 Asking the user for a query

Short of building a web-based GUI, the Python input() function is the easiest way to ask for direct input for a search query. The following code snippet keeps asking for a new query and stores it in the query variable until the user types STOP.

```
stop = False
while not stop:
    query = input("Enter your query, or STOP to quit, and press return: ")

    if query == "STOP":
        stop = True
    else:
        print(f'You are searching for {query}')
        # your query processing comes here
```

5.4 Confidence threshold and fallback mechanism

Something that I have not touched upon, but is important is the notion of a fallback mechanism. When we are searching for results using similarity, there is always a chance that the query does not match anything in our corpus. This is usually computed by some sort of threshold in our similarity, under which we consider that nothing is relevant. What do we do then? We have a few potential options.

5.4.1 Fallback by warning message

The simplest way to recover from a state like this is simply to display to the user that there is nothing relevant to the query and suggest that they reformulate it or add additional terms to it. This is different from displaying no result at all, which could lead the user to think that the system is not working.

5.4.2 Fallback by query reformulation

Going one step further would be to suggest a query reformulation ourselves. One simple way to do so is to use a language model (e.g., given an existing query q, what is the most likely next term entered by the user?).

6 Evaluating your search results

This leaves us to ask the question: How do you know if your search algorithm is good? There are several very useful metrics used in information retrieval to give you an idea of whether your search engine is successful. We will see 3 of them: Precision, Recall, Normalised Discounted Cumulative Gain (NDCG). But first of all, we need to start by giving ourselves a gold standard to measure the algorithm against.

6.1 Human assessment of relevance

Typically, we measure the quality of a search engine based on a set of reference queries for which we know the expected results. Those results can be either binary, i.e. a document is relevant or irrelevant to a query, or graded, i.e. a document is very relevant, somewhat relevant, or not relevant. In the latter case, relevance is encoded as a number from 0 to 2 while in the former case it is encoded as a binary number 0 or 1.

For example, a sample query would be represented as $q_1 = \{d_1 : 2, d_5 : 2, d_6 : 2, d_7 : 1, d_8 : 1\}$ with the implicit assumption that it is 0 for all other documents. Knowing this, you can then compute several metrics.

6.2 Precision

Precision @ K (precision at K) is concerned with how many of the returned K documents are relevant. It is common to plot the precision at K with multiple K values to see how an algorithm changes when it needs to return more data. Also, it makes intuitive sense: if you can only display K documents on the first result page of your search engine, you don't want it to be polluted with irrelevant documents. Precision is usually evaluated with binary relevance judgments, meaning that a document is either considered relevant or not relevant, with no grading of its relevance.

$$\text{Precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Programming your own Precision score is relatively easy, but if you are looking for a reference implementation, Scikit-Learn has one[1].

6.3 Recall

Recall @ K (recall at K) is concerned with how many of the relevant documents have been retrieved. Like precision, it is common to plot recall at K with multiple K values to see how the results of the algorithms change as more results are returned. Recall makes a lot more sense in small, specialised search engines where only a restricted set of documents is really deemed relevant: medical information retrieval, legal information retrieval, etc. It is used in conjunction with Precision to give a full picture of the capabilities of a search engine.

$$\text{Recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Programming your own Recall score is relatively easy, but if you are looking for a reference implementation, Scikit-Learn has one[2].

6.4 Normalised Discounted Cumulative Gain

Normalised Discounted Cumulative Gain (NDCG) @ p is a way to evaluate search results in a more fine-grained way by considering not only whether relevant documents are returned, but their ordering in the result set. As you can see in the DCG equation below, $log_2 i$ progressively discounts the importance of rel_i so that the results at the beginning of the list of returned documents are judged more important (they are divided by a smaller denominator) than the results at the bottom of the list of returned documents (which are divided by a larger denominator). The DCG is then divided by an ideal DCG, which is the DCG value of a perfect ranking (all highly relevant documents are ranked at the beginning of the list in decreasing order of relevance).

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{log_2 i + 1}$$

$$NDGC_p = \frac{DCG_p}{iDCG_p}$$

Where $IDCG$ is the DCG based on a perfect ranking, i.e. the DCG of the documents ordered by decreasing user-defined relevance. Like the other metrics, building your own NDCG calculator is not difficult and is a good way to learn, but Scikit-Learn provides a reference implementation[3].

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html`
[2] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html`
[3] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ndcg_score.html`

6.5 Evaluation campaigns

Evaluation campaigns are large regular events during which search engines and research groups compete to build the best search algorithm using gigantic datasets. You might be wondering who has the time to go over billion of pages and annotate them as relevant or not for a given query, in order to evaluate those search engines. The truth is that the information retrieval community uses a clever trick to avoid this labelling: **combination**. The process of pooling, which happens for each search engine evaluation campaign:

1. competing search engines run a set of evaluation queries and record the first N results of their search engines
2. a central authority recovers those results and takes the union of all documents returned during that search
3. that union of documents is then split up randomly such that multiple competitors have to label them
4. the competitors receive a subset of the results to label
5. documents are then assigned a relevance based on the aggregated labels from the competitors

In this way, no one has to do all labelling and some notion of fairness is maintained.

7 Tasks

Tasks are much more time-consuming than exercises and allow you to go much deeper into content mastery. We do not expect you to finish all of them in the lab, but they can be good toy problems to practice with.

1. [⋆] Run the cosine similarity (either your own implementation or one of the two shown above) on the following set of test documents, with respect to a query $q = [1, 2, 0, 0, 0, 0]$:
   - $d_1 = [6, 12, 0, 0, 0, 0]$
   - $d_2 = [2, 4, 0, 0, 0, 0]$
   - $d_3 = [3, 6, 0, 0, 0, 0]$
   - $d_4 = [0, 0, 1, 1, 0, 0]$

   Note down $sim(q, d1)$, $sim(q, d2)$, $sim(q, d3)$, and $sim(q, d4)$. What do you observe and how do you explain it?
2. [⋆] Reusing the movie review data from the previous lab, write a program that generates the vocabulary list of all documents in a data structure of your choice, with the following information: term id, canonical form, inverse document frequency. Term id should just be an identifier for the term, e.g. t001. Canonical form can either be stemmed or lemmatised, left for you to choose. IDF is the standard formula shown in this document. Optionally, your program should be able to save that data structure in a file and load it back.
3. [⋆] Write a program to create a term-document or document-term (whichever you prefer, it is not relevant at that stage) matrix with the term frequency values for each document-term pair. The program should have the option of using binary term weights, raw frequency term weights and log-scaled frequency term weights.

4. [⋆] Write a program that takes user input in the console and searches for the 10 most relevant documents based on cosine similarity and your term weighting scheme. You should return not only the document but also some document ID. I suggest using the filename as document ID in order to avoid having to decide on a naming scheme.

5. [⋆] If you can, pair yourself with one or more arbitrary teammates and do some pooled labelling. Agree on a set of 3-5 queries that you will run on your respective search engines, then record a document id to be able to share your results. Then, split those document IDs between each of you and provide an assessment of their relevance. Finally, once you have recorded your ground truth in the form of (query, documentId, relevance) write a program to compute Precision and Recall and compare the performance of your algorithms. If you cannot or do not want to pair with another person, you can do the same thing by simply using multiple variations of your own search engine, e.g. changing term weighting scheme, or changing the size of the vocabulary, or changing the stemming/lemmatisation technique, etc.