**CAI Lab 4**

ElasticSearch & Queries

Gerard Comas & Laura García

**Exercise 1**

In the Slow Search, since we will be looking at the 4 documents and at 2 the words in the vocabulary (4), we will be forced to do a total of 8 sums[[1]](#footnote-0), if we use the document-term matrix to perform the computations. If we used a dictionary (term, weight), we would be doing 2 sums.

In the Fast Search, we will only look at those columns of the matrix where the term has weight 1 in the query. Again, if we use the document-term matrix, we are reducing our search to only the two middle columns of the matrix; which is, again, 8 sums. If we used inverted lists, we would only do 2 sums.

In this case, it looks like there is not a great difference between using Slow Search or Fast Search, regarding the number of sums that the program needs to perform. However, we do see a difference in the number of iterations: f.i. in the Slow Search, we are iterating on documents ‘doc1’, ‘doc3’ and ‘doc4’ and we don’t need to; because they do not contain the two terms in the query.

**Exercise 2**

We have implemented the Fast Search version as shown next. The rest of the code implementation will be shown at the end of the report.

| def FastSearch(query, r, PL):  sims = dict()   l2query = np.sqrt(len(query.split())) *# l2 of query as a binary vector*  *# get nr. of docs; just for the progress bar*  ndocs = int(client.cat.count(index='arxiv', format = "json")[0]['count'])   *# scan through words in the query, compute cosine sim between query and each doc*  for w in query.split():  L = PL[w] *# find posting list for the word*  for (docid, weight) in L:  try:  sims[docid] += weight/l2query *# accumulates similarities*  except:  sims[docid] = weight/l2query   *# now sort by cosine similarity*  sorted\_answer = sorted(sims.items(), key=lambda kv: kv[1], reverse=True)  return sorted\_answer[:r] |
| --- |

Here, we report the times needed to execute the Slow and Fast Search algorithms for the query: 'computer magic'. We have made sure that the results are the same for both searches. However, we have noticed some differences in the cosine similarity with respect to the query in one of the documents. This difference is an error of , so we have considered it negligible.

| Slow Search | Fast Search |
| --- | --- |
| 7min 11,8s | 44ms |

Even though these results clearly show that the Fast Search is incredibly faster than the Slow Search; we must also consider creating the Posting Lists. Creating the inverted file (IF) structure took 6min 32,3s. We observe that less time was needed overall when applying the Fast Search than to use the Slow Search. Besides, the more queries we execute with the Fast Search, it makes more sense to invert some time in creating the IF.

| **Rank** | **ElasticSearch** | **Fast Search** |
| --- | --- | --- |
| 1 | 000677 | 000677 |
| 2 | 000650 | 000650 |
| 3 | 001475 | 003482 |
| 4 | 000992 | 001475 |
| 5 | 001477 | 002825 |
| 6 | 013376 | 000265 |
| 7 | 006074 | 000955 |
| 8 | 001652 | 000255 |
| 9 | 001630 | 000896 |
| 10 | 000521 | 000992 |

**Exercise 3**

Here we present the results obtained with the ElasticSearch algorithm and our implementation of the Fast Search.

In the first place, we tried to see the results we would obtain for the same exact query as before: 'computer magic'.

On the one hand, we must mention that the time needed by both algorithms has been practically the same (ES: 66ms and FS: 53ms). If anything, we see that our implementation is slightly faster than ElasticSearch.

On the other hand, the results are given in the table to the right: the rank is the position in which each document was given in the answer for each of the searches (in columns). We see that the results coincide in only 3 documents (‘000677’, ‘000650’, ‘001475’ and ‘000992’) and still they are not displayed in the exact same order. Note, however, that we do not know what the score returned by ElasticSearch is and so we cannot make any assumptions as to why this happens.

For other queries we have obtained the results shown next. We have compared the algorithms for 2 more queries, and they both have led us to very similar conclusions: the time needed by each search was approximately the same and the number of coincidental files is also less than 5 for both cases.

| **Rank** | 'star brown' | | 'planet star' | |
| --- | --- | --- | --- | --- |
| **ElasticSearch** | **Fast Search** | **ElasticSearch** | **Fast Search** |
| 1 | 011045 | 010049 | 001324 | 001324 |
| 2 | 010049 | 011045 | 000152 | 008640 |
| 3 | 009349 | 014209 | 008640 | 011483 |
| 4 | 012590 | 012590 | 009363 | 011154 |
| 5 | 008869 | 001260 | 005483 | 014606 |
| 6 | 012600 | 008304 | 011154 | 006639 |
| 7 | 000616 | 004320 | 002642 | 008808 |
| 8 | 004880 | 000226 | 000197 | 012091 |
| 9 | 004320 | 009349 | 004647 | 002642 |
| 10 | 003245 | 000616 | 005762 | 004647 |

**OTHER CODES**

Creating the vector model with tf-idf

| def encode\_doc(doc):  tv = client.termvectors(index='arxiv', id=doc, fields=['text'], term\_statistics=True, positions=False)  D = tv['term\_vectors']['text']['field\_statistics']['doc\_count']  f\_dict, df\_dict = {}, {}  if 'text' in tv['term\_vectors']:  for word in tv['term\_vectors']['text']['terms']:  f\_dict[word] = tv['term\_vectors']['text']['terms'][word]['term\_freq']  df\_dict[word] = tv['term\_vectors']['text']['terms'][word]['doc\_freq']  maxf = max(f\_dict.values())   doc\_dict = {}  for word in f\_dict.keys():  doc\_dict[word] = (f\_dict[word]/maxf)\*(math.log(D/df\_dict[word], 2))   return doc\_dict |
| --- |

Computing scalar product and norm (to normalize the weights when needed)

| def scalar\_product(doc1, doc2):  prod = 0  for word in doc1.keys():  if word in doc2.keys():  prod += doc1[word]\*doc2[word]   return prod  def my\_norm(doc):  return math.sqrt(scalar\_product(doc, doc)) |
| --- |

Slow Search

| def SlowSearch(query, r):  sims = dict()   l2query = np.sqrt(len(query.split())) *# l2 of query assuming 0-1 vector representation*   *# get nr. of docs; just for the progress bar*  ndocs = int(client.cat.count(index='arxiv', format = "json")[0]['count'])   *# scan through docs, compute cosine sim between query and each doc*  for s in tqdm.tqdm(scan(client, index='arxiv', query={"query" : {"match\_all": {}}}), total=ndocs):  docid = s['\_source']['path'] *# use path as id*  weights = encode\_doc(s['\_id']) *# gets weights as a python dict of term -> weight*  sims[docid] = 0.0  for w in query.split(): *# gets terms as a list*  if w in weights.keys():  sims[docid] += weights[w] *# accumulates if w in current doc*  *# normalize sim*  sims[docid] /= (l2query\*my\_norm(weights))  *# now sort by cosine similarity*  sorted\_answer = sorted(sims.items(), key=lambda kv: kv[1], reverse=True)  return sorted\_answer[:r] |
| --- |

Creating the Inverted File structure

| def inverted\_file():  PLists = dict()  ndocs = int(client.cat.count(index='arxiv', format = "json")[0]['count'])   for s in tqdm.tqdm(scan(client, index='arxiv', query={"query" : {"match\_all": {}}}), total=ndocs):  docid = s['\_source']['path']  weights = encode\_doc(s['\_id'])  n\_weights = my\_norm(weights)   for w in weights.keys():  try:  PLists[w] += [(docid, weights[w]/n\_weights)]  except:  PLists[w] = [(docid, weights[w]/n\_weights)]   return PLists |
| --- |

ElasticSearch

| def ESSearch(my\_query, r):  s = Search(using=client, index='arxiv')   qsplit = my\_query.split()  q = Q('query\_string',query=qsplit[0])  for w in qsplit[1:]:  q = q & Q('query\_string',query=w)   s = s.query(q)   answer = dict()  for a in s[0:r].execute(): *# only returns a specific number of results*  answer[a.path] = a.meta.score   return answer |
| --- |

1. This is because even if the value (doc, t) is 0 in the matrix, the sum will be executed. This does not happen if we use a dictionary as a sparse structure to store the weights for each document. [↑](#footnote-ref-0)