Øving 2

November 1, 2023

1 Task 1

a) We started with removing the stopwords, then tokenized the text, converted all words to lowercase and removed punctuation. This resulted in the list: `['intelligent', 'behavior', 'people', 'product', 'mind', 'mind', 'like', 'human', 'brain']`.

We then created a dictionary where the key is a word and the value is a list with lists, with the documents the word is in and amount of times the word is in the document. This is our inverted index, and it looks like table 1.

Table 1: Inverted index of text.

brain	[1,1]
behavior	[1,1]
human	[1,1]
like	[1,1]
intelligent	[1,1]
product	[1,1]
people	[1,1]
mind	[1,2]

This one seems a bit strange, as we just have one document, but we didn't find any other definitions of inverted index, so we kinda just assumed this was what we were meant to do.

b) To making an inverted list with block addressing we first split the text in blocks of roughly the same size. We decided to use five blocks, and we didn't remove stop-words or punctuation, as the examples from lecture didn't remove them. The text in blocks looks like this: `['Intelligent behavior in people is', 'a product of the mind', '. But the mind itself', 'is more like what the', 'human brain does .']`

As in task a) we created a dictionary with the word as key and a list of the blocks the word is in as value. This resulted in table 2

Table 2: Inverted index of blocked text.

like	[4]
product	[2]
brain	[5]
people	[1]
human	[5]
intelligent	[1]
behavior	[1]
mind	[2, 3]

c) We struggled a bit with understanding how the suffix tree were to look. In lecture it seemed like we needed to include all substrings of a word, where it in other examples seemed to just use the start of the words, so we didn't really do this one.

d) We tokenized the documents, removed stopwords and punctuation, and converted all words to lowercase. We created a dictionary in the same way as before, which resulted in table 3:

Table 3: Inverted index of collection.

many	[[3, 1]]
brain	[[1, 1]]
together	[[4, 1]]
puzzle	[[3, 1]]
although	[[1, 1]]
mystery	[[3, 1]]
yet	[[4, 1]]
thinking	[[2, 1]]
remains	[[2, 1]]
see	[[3, 1]]
total	[[2, 1]]
understand	[[5, 1]]
big	[[3, 1]]
engages	[[2, 1]]
human	[[1, 1]]
like	[[3, 1]]
put	[[4, 1]]
pieces	[[4, 1]]
jigsaw	[[3, 1]]
know	[[1, 1]]
us	[[4, 1]]
even	[[1, 1]]
years	[[2, 1]]
pretty	[[2, 1]]
much	[[1, 1], [2, 1], [4, 1]]
ago	[[2, 1]]
ten	[[2, 1]]

Task 2

(a) The ELK stack consists of components that makes it easy and safe to take data from a source and analyze, search or visualize it. Some of the components are Elasticsearch, Kibana, Beats and Logstash. Lucene is an open source library that provides indexing and search features, spellchecking, hit highlighting and advanced analysis/tokenization capabilities.

(b) We based our indexer on the indexer we made in assignment 3. We started removing stopwords and punctuation, and decided to stem the documents, mainly just to try it out. We then tokenized the documents, and created an inverted index of the collection.

(d)
- `claim` The matching documents in ranked order with our implementation are: 6 and 2. With ELK the matching document is: 2.
- `claim*` Our implementation: 6 and 2. ELK: 2 and 6 (same score).
- `claims of duty` Our implementation: 6 and 2. ELK: 6 (`match_phrase`), all (`match`, 6 is highest ranked.)

(e)
- In our implementation, the query `claim of duty` results in a match to documents that contain one or both of the terms claim and duty (we removed stopwords). We decided that this was an okay result, given that the documents that contain both terms ranks higher than documents that just contain one of the terms. We also stem the query and the text, which gives us a match on instances of both claims and claim.

  With ELK and matching the entire phrase, we just get the document that contains the entire phrase in correct order. When using match, it responds with all documents containing some of the terms in the phrase, but the one containing all terms ranks highest.
- If we had removed the stopwords in the text we use in ELK, then we still wouldn't get the same results as in our implementation, as we also stem query and text. I assume we would get the same results had we either stemmed the text in both implementations, or not stemmed both.
- Based on our implementation and checking the texts, we think the results matches our expectations.

(f) We used the same setup for indexing the documents as in the previous task, and created a collection with all the emails. For the queries we used `match`, so that the query would match each term in the query.

Query results:

`Norwegian and University and Science and Technology`
We got 10000 results for the query.

`Norwegian University Science Technology`
We got 9744 results for the query.