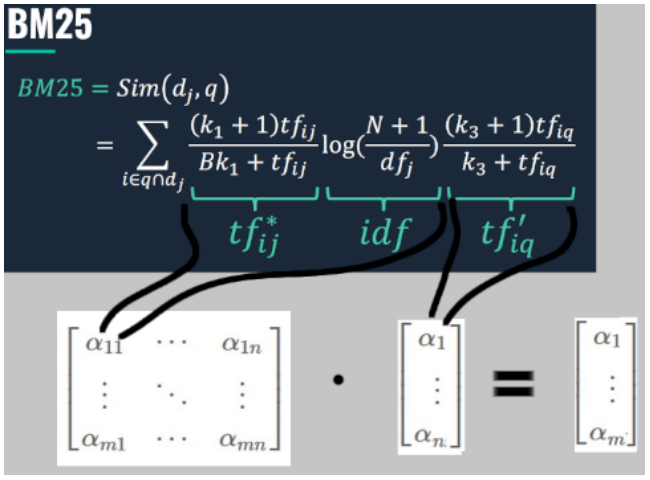
ID: 209236926, 315705111
Email: shavitda@post.bgu.ac.il,
Git-hub: davis0011/IR-Project
Google storage bucket:

The key experiments we ran(we can call then the engines) were as follows:

1. The first version of the engine was heavily reliant on code from the assignments and was not at all optimized to run efficiently. It iterated using the postings_iter function and was not finished in less than 2 minutes on even the smallest searches on the full corpus for TFIDF cosine similarity. At this point we realized we needed to rewrite the word-searching functionality, which we did for the next version.
2. Implemented BM25 according to the equation taught in class.
   We used numpy arrays for the query and documents, after filtering the documents to only the relevant candidates.
   Since the number of candidates is unknown at runtime we would need to create a potentially massive(~500,000 rows) matrix, and after some experiments to check the feasibility of this we found that making one massive matrix was not possible in the time given.
   Our solution was to split the candidates into 1000 buckets, with each bucket containing all candidates with a similar docid%1000.
   Since each bucket contains a smaller matrix than the large matrix we will save a substantial amount of time on rewriting the matrix when its size exceeds the allocated space. Numpy's merging function is ~~gucci~~ very fast so the time it takes to merge the matrices is negligible. The values in the matrix and vector are as follows:



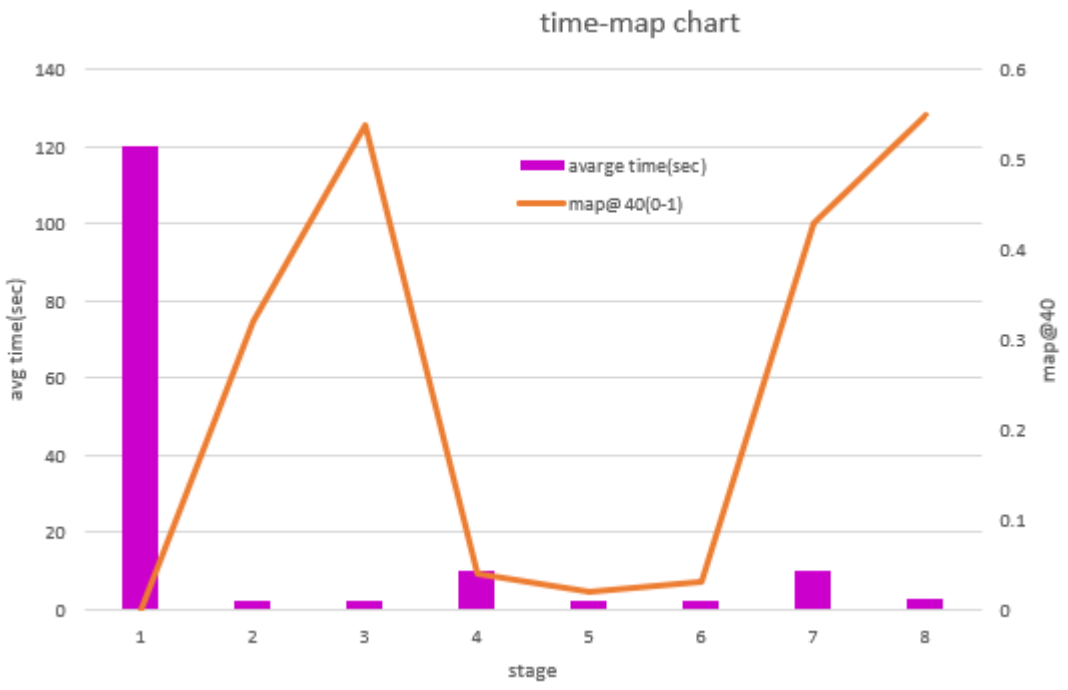
Each row is a doc and each col is a query word, so a cell is the left side of the bm25 formula for a given document and query word. The query has the right part of the formula for each word in the query. Once we perform matrix multiplication between the document matrix and the query to get the final score for each document, then we choose the top n and return them. This method yielded decent results, both fast and above the minimum requirements for MAP@40, but we wanted to improve on it. see Evaluation Chart.

3. We added a second search on the title index to get all relevant documents for that title. Since the BM25 calculation and the title index will not easily be comparable, we decided to normalize the results by the largest value in each set. We then combined the two normalized scores. Results were improved and runtime was not drastically increased.

4. We tried to do query expansion using word2vec but we could not find a way of choosing which word needed to be expanded. We considered ML to figure out this problem but did not have time to implement it. Results were ~~not gucii~~ undesirable. We moved away from this method.
5. We tried to add in a weighting of the results by the pagerank of the candidates. The calculation was initially just a multiplication of the original result by log10 of the pagerank of the result. This dropped the MAP@40 of the engine to ~0.08, and all attempts at changing the weighting system ended in failure.
6. Similar to the last version we tried implementing a calculation using pageviews. This was also a failure in exactly the same way. At this point time was running short so we decided to make this the final version of the engine before selecting the best performing version. This time we tried tweeking the values of the weights and the type of normalization on all previous

parameters, and we even added back the query expansion with its own weighting system. In the end the results were similar to version 4 and not even close to the better(albeit middling) version 2.

7. As a hail-mary we tried using Word2Vec to generate a set of five similar words for each word in the query. For each word in the expanded set we generated the 10 closest words. Having expanded the set of words to many words that should all be close in some way to the “true meaning” of the query, we summed the occurrence of words and looked for a top subset of repeating words. We postulated that these words would be correlated statistically to the query’s meaning. As a final attempt to not stray from the original query we added the words from the original query into the final query. This was by far our best attempt at improving the model, getting us a MAP@40 of about 0.42. Sadly this result was still not as good as model 3, which was simply BM25 on the body and title search.
8. On the last day of the assignment we tried adding the anchor text into the calculation and we were surprised(for no good reason) to find that it helped improve the engine a little. With a few final tweeks to the percentages of the different elements of the score we submitted our BM25 + title + anchor engine. May God have mercy on our souls.



Takeaways and key findings:

- Preprocessing is a godsend. When we needed a static value like the size of a document or its full untokenized title we would have needed to reconstruct or perform a lookup to get it. Insead we made a dictionary of metadata for the documents and loaded it into memory when the engine was turned on.
- Nested data structures are excellent for lookup efficiency. The time it would take to iterate through all words in the Inverted Index to find the right posting lists is massive in comparison to the time it takes to look up the word in the posting_locs dictionary and immediately access the correct bin at the given offset.
- Natural language processing is needed for more robust understanding of queries. Try as we might, we could not get the semantic meaning of words using BM25, Cosine Similarity or even an implementation of Word2Vec(although we will admit it was not a very sophisticated implementation at first, but the second implementation showed some merit).

Good query:
query: “How to make hummus”, time: 3.4963831901550293, ap@40: 1.0
results:

75065, "Hummus"
48876576, "Hummus the Movie"
37785018, "Abu Hummus"
176194, "Make (software)"
2501449, "The Way You Make Me Feel"
2503392, "You Make Me Wanna..."
1826856, "Make It Happen (Mariah Carey song)"
1931027, "Born to Make You Happy"
356928, "Let's Make a Deal"
3557039, "You Make Me Want to Be a Man"
3658961, "Make Your Own Kind of Music (song)"
1533068, "Make Poverty History"
4941998, "You Make Me Sick"
1575040, "I'm Gonna Make You Love Me"

4842561, "Make Me Smile (Come Up and See Me)"

Query gets tokenized and stopwords removed and becomes “Make Hummus”.
All parts of the query are relevant and are not difficult to interpret for BM25 since they have no hidden semantic meaning when placed together. Additionally there seems to be little overlap between Make and Hummus which leads to correct results since neither our engine nor the oracle had to jump through any logical hoops to reach the whole “domain” of the results.

Bad query:
query: ”best marvel movie”, time:7.916390419006348, ap@40: 0.0
22673889, "MTV Movie Award for Best Fight"
22883216, "MTV Movie Award for Best Hero"
50276185, "Pok\u00e9mon the Movie: Volcanion and the Mechanical Marvel"
14321433, "Best Worst Movie"
3674142, "Critics' Choice Movie Award for Best Animated Feature"
10526301, "Best Sports Movie ESPY Award"
3396154, "Critics' Choice Movie Award for Best Actress"
3396197, "Critics' Choice Movie Award for Best Actor"
15721612, "The Best Movie"
3432660, "Critics' Choice Movie Award for Best Screenplay"
3398766, "Critics' Choice Movie Award for Best Director"
3399194, "Critics' Choice Movie Award for Best Supporting Actor"
3405654, "Critics' Choice Movie Award for Best Supporting Actress"
11019770, "Bollywood Movie Award \u2013 Best Film"
11792123, "Bollywood Movie Award \u2013 Best Dialogue"
Query is difficult because after tokenizing it stays the same query which isn't bad in all cases, it would be nice to remove any words not relevant to the results. In this case the words “best” and “movie” get more weight together than the word “marvel”(and as humans we can understand that “marvel” is the important part of that sentence). We can note that “best marvel movie” is a very small subset of “best movie” and so the results that are returned are more relevant to Best Movie than to Marvel. We could solve this problem using LSI or ML or even some non-trivial usage of Word2Vec or Doc2Vec.

indexes/pkl files size:

```
shavitda@instance-1:~$ du -ch *index
997M    anchor_index
6.0G    body_index
160M    title_index
7.1G    total
shavitda@instance-1:~$ du -ch *pkl
85M     cosine_len_dict.pkl
208M    meta_dict.pkl
146M    pagerank.pkl
74M     pageviews.pkl
511M    total
```