20 Queries in evaluating Search Engine performance *(ranking performance (effectiveness) and in terms of runtime performance (efficiency)).*

Good Queries

Query: **hackathon**
- 0.0151s
- Returned results are all relevant to hackathons

Query: **cristina lopes**
- 0.0011s
- Returned results of Professor Lopes' ICS faculty page, as a featured speaker, etc... which is good top queries are all about her

Query: **machine learning**
- 0.0337s
- All results are machine learning related, ranging from classes, machine learning professors, to publication on machine learning

Query: **master of computer science**
- 0.0568s
- Expecting mcs home page, which it returned the main mcs page along with other relevant searches such as UCI mcs program rankings and research for mcs.

Query: **master of software engineering**
- 0.0761s
- Expecting mswe home page, which it returned the main mswe page along with other relevant searches such as UCI mswe intro classes, and career/jobs within mswe

Query: **research opportunity in summer**
- 0.0449s
- All researches are relevant including honors research opportunities, research summer programs, and students presenting their summer research project

Query: **classification, clustering, and regression**
- 0.0151s
- Mainly returned machine learning and probabilistic learning courses like CS175, CS178, and CS274, and that class does cover a good chunk of classification, clustering, and regression.

Query: **theoretical aspects of bayesian inference**
- 0.0167s
- All results are relevant, including publication, teachings, and papers on bayesian inference

Query: **design and analysis of algorithms for applied cryptography**
- 0.0446s
- Expecting details on COMPSCI 202P. Applied Cryptography course and returned courses offerings, and the professor who teaches it which is good.

Query: **publicly available collection for recommendation systems that records the behavior of customers of the European leader**
- 0.0409s
- Expecting KASANDR dataset as top results and it did return the dataset

Poor Queries

Query: **ramesh**
- 0.0151s
- Returns Professor Ramesh Jain many blog pages, with little content as the top result. Was hoping we could get the main home page
- Solution: We implemented HITS ranking. The same query now instead of subsection of his pages like blogs, returns his home page, research papers, patents, partners, etc..

Query: **camera calibration publication**
- 0.0227s
- Returns a meaningless page with unrelated content as the top result
- Solution: We re-weighted our tf-idf score. The same query now returns a list of Aditi Majumder's publications as the top result, the professor for Computational Cameras and Projectors at UCI.

Query: **UCI computer science research grant**
- 0.2162s
- Returns a Recent News page for the entire Donald Bren school as the top result
- Solution: We re-weighted our tf-idf score. The same query now returns a list of research grants for computer science on Michael J. Carey's page as the top result

Query: **restaurants on campus**
- 0.0217s
- Returns a meaningless page with unrelated content as the top result
- Solution: We re-weighted our tf-idf score. The same query now returns this site; which contains a guide for restaurants on campus as the top site

Query: **multi agent pathfinding**
- 0.0562s
- Returns a meaningless page with barely related content as the top result
- Solution: We re-weighted our tf-idf score. The same query now returns a list of publications related to multi agent pathfinding as the top result.

Query: **semantic segmentation**
- 0.0217s
- Returns a meaningless page with unrelated content as the top result
- Solution: We re-weighted our tf-idf score. The same query now returns a research paper discussing using semantic segmentation for Caenorhabditis elegans detection as the top result

Query: **technic image origin**
- 1.548s
- All 3 tokens have very large postings. This causes the query to take too long.
- Solution: We stopped using `ast.literal_eval()` for loading postings. Search is now down to 0.2878s.

Query: **UCI informational computer science**
- 2.103s
- All 4 tokens have very large postings. This causes the query to take too long.
- Solution: We stopped using `ast.literal_eval()` for loading postings. Search is now down to 0.1721s.

Query: **UCI machine learning**
- 0.9103s
- All 3 tokens have very large postings. This causes the query to take too long.
- Solution: We stopped using a `.json` file and switched to `.pickle`

Query: **application back canon detail equivalent first general home image june know last may next origin previous question research size technical uci view work xcode year zero**
- 7.6940s
- Edge case query, this query contains the most frequently appeared word *(that is not a stop word)* in each alphabet (a-z).
- Solution: We stopped using a `.json` file and switched to `.pickle`

Query: **of next the and up to in technic detail imag for prev size origin taken iso on at is canon eo equiv 10 thi with by 25 from that be inform are comput 100 it uci 50 an as use 20 f2 or d60 ic 70 have crw all not**
- 9.7546s
- Edge case query, this query contains the 50 most frequently appeared words (stop word inclusive).
- Solution: We stopped using a `.json` file and switched to `.pickle`

Explain what you did in your search engine to make them perform better.

For the search engine, these are the general heuristics that we look at for any improvements for our queries/retrieval:

1. **TF-IDF Scoring**: For ranking documents based on their relevance to the query. It helps in identifying documents that are not only frequent in terms of query terms but also unique.
2. **Cosine Similarity**: Angle between the query vector and document vectors, which gives a more accurate representation of relevance compared to just TF-IDF.
3. **Term Match Boost**: Boosting the score of documents if there are more matches of the query terms in the document.
4. **Eliminate Duplicates**: Detecting and removing duplicate or near-duplicate pages definitely improve the quality of our search results.
5. **Hyperlink-Induced Topic Search (HITS)**: Assigns two scores to each page (authority and hub), reflecting the concept of "hubs" that point to many authoritative pages
6. **Page Rank**: Assigns scores to each page where higher scores are awarded for important pages linking to the target page.
7. **Positional Index**: Lastly, we chose to add token positions to our posting. This allows us to add implementation for 2-gram and 3-gram search support at query time.

Initially as our **minimal viable product**, we did a bare bone Search Engine with just ***TF-IDF scoring and Near-Duplicate Detection***. Our search engine performs poorly as expected, averaging **18.0 seconds query time**.

One problem that we had was certain sites like: https://www.ics.uci.edu/~ziv/ooad/intro_to_se/sld027.htm, is just straight up a bad result, though it was appearing on query "***master of computer science***" a lot, even though it has nothing to do with mse. Was considered to be low value and unreadable but that link was the exception. But then we found: https://www.ics.uci.edu/~ziv/ooad/intro_to_se/tsld027.htm, which is a lot better but still doesn't show up alot.

We then added ***Cosine Similarity and Term Match Boost*** to our search engine, which gave us more relevant results.

Another problem we ran into was that certain sites appears as #1 search for non-conventional words. Or encountering a situation where a page with a comprehensive word list is being ranked highly for queries involving rare or unique words. I think this is a prime example of "keyword stuffing" from earlier lectures, this really influences the tf-idf ranking if the word is rare in other documents. Listed below are some sites we saved:

- https://www.ics.uci.edu/~kay/wordlist.txt
- https://ics.uci.edu/~kay/courses/h22/hw/DVD.txt
- https://ics.uci.edu/~kay/courses/h22/hw/wordlist-random.txt

We end up ignoring any .txt files, since they are not HTML and are not relevant to the search.

At this point we are getting ok results but our time is still quite bad. Our first approach was to do index sharding. So we split the index into alphabetical shards: ***(a-c), (d-f), (g-i), (j-l), (m-o), (p-r), (s-u), (v-z) and (misc)***. This improved our query time, lowering it down to an average of **1.0s-3.0s**.

We've noticed a significant time improvement from purely reading the full index vs partial. So we thought maybe it might shave off more time if we partition more. Now we have **36 shards, 26 for each letter of the alphabet (a-z) and 10 for numbers (0-9)**. This improved our query time even more, now averaging **0.4s-1.7s**.

We also decided to add a **cache to our search engine**, so that we can store the results of the most common queries/most recently searched queries. This improved our query time even more, now averaging **0.01s-1.7s**.

Though the problem with this is that the initial query time would still be slow because we have to query and build the cache first. Also we had to put a limit on the cache size, so that we don't end up loading the entire index into the cache on accident. This solution poses some issues, but it is a good trade off for the time improvement.

As for the time being, we were looking at other alternatives for storing the index (data structure wise). And we tested out 3 different formats:

1. **JSON**
2. **CSV**
3. **Pickle**

Utilizing JSON was our format method, it gets the job done but it is not the most efficient as it requires a lot of parsing through into the index. CSV performed worse than JSON, which was surprising. Reading CSV usually is faster than JSON, but we think it is because of the inverted index structure. Reading csv is fast, but then we'd have to parse and convert it into a dictionary which takes too long. Then on to Pickle, which was the fastest out of the 3. Since pickle is in binary format, and is specifically designed for serializing and deserializing Python objects, there's no parsing overhead, which saves a lot of time. From our tests, **Pickle was 2x faster than JSON**:

- 'machine learning' query time: 0.645s -> 0.359s
- 'research' query time: 0.625s -> 0.374s
- 'master of computer science' query time: 1.303s -> 1.143s
- 'master of software engineering' query time: 1.375s -> 0.709s
- 'cristina lopes' query time: 0.713s -> 0.372s
- 'machine learning and its impact on society' query time: 1.753s -> 0.922s

This was great progress. Our "breakthrough" was that we wanted to switch to something else new. Referring to a concept discussed in class for handling inverted indexes, which

was "**peeking**". We used this to access only a very small portion of the index data without loading the entire index into memory. This was a great improvement, as we were able to reduce the query time even more. Now our query time averages **0.1s-0.3s**, which is within the time limit of the project.

Though what we found is that with peek(), the search time is significantly faster than sharding. But only for queries with < 5 terms, the performance falls off if we do a large query. While sharding is consistent and doesn't have a performance drop off. Thus for our final product we employed ensemble programming and combined to use both methods of retrieving data, utilizing the best of both, for search.

We utilize Pickle for storing our index, with another json file as a reference sheet to jump to the point where each term is located. We then peek(term), and if it is a term we have searched before then its cache will return instead.

Query time is now averaging **0.01s-0.2s**!

What was left was we merely played around with weights and tried to optimize our query results, such as implementing **Hyperlink-Induced Topic Search (HITS)** and **PageRank** which improved our queried results slightly. We chose to apply these scores document-wise during the indexing stage, so these implementations did not affect our search time, rather the order in which results were returned.

The last change we made was adding a position field to our posting. This tracks all positions of a given token in a document. This allowed us to implement n-gram support, by simply looking for documents with sequential position fields for tokens in the query. While this improved the accuracy of our search, it did come with some issues. While our search time for smaller queries was unchanged, searches for long queries became much slower. In the future, we could look into implementing an ensemble approach, where shorter queries leverage n grams and longer queries do not.