ETD Search Engine Report

Kristen Jodie Basson **Fardoza Tohab** **Mosamat Sabiha Shaikh**

BSSKRI003 **TBHFAR002** **SHKMOS004**

Introduction

In this report we document the development and evaluation of a targeted search engine for electronic theses and dissertations. Data is sourced from the Networked Digital Library of Theses and Dissertations (NDLTD). NDLTD maintains a collection of more than 6 million metadata records which is available for public use. Apache Solr, an open-source search platform, is used for searching and retrieving relevant results. The search engine was designed to be accessible through a web-interface.

Dataset

A perl harvesting file named “harvey.pl” was used to collect 529276 metadata records from the NDLTD ETD metadata collection. Using the OAI protocol for metadata harvesting, we managed to acquire a stream of XML-encoded records.

Collection Processing

The format of the XML files we obtained was not suitable for indexing in Solr. As a result, we created a python file to take in XML files and transform the records into a format that is compatible with Solr’s indexing requirements. The transformed metadata records were indexed by Solr creating indexes that facilitate search operations.

Queries

We conducted an experimental evaluation of the default search engine using title and description with 8 unique search queries. Each member assessed the retrieved document list for relevance and the average of each was taken. We implemented field boosting to give a higher weighting to the description using Solr’s TF-IDF scoring process. The experiment was repeated with the same set of queries. The two sets of experiments will be referred to as *before* and *after* field boosting implementation.

The Queries we used:

1. Climate Change
2. Mathematical Modelling
3. Mental Health
4. Cloud Computing
5. Cancer Prevention
6. Greenhouse Gas Emissions
7. Black Holes
8. World War II

Algorithms Employed

- **Tf -Idf scoring**

To improve the ranking of the search results returned, we boosted Solr's tf-idf scoring. To implement this, the query in the *perform_solr_search* function in the app.py file was constructed with the following parameters relevant to tf-idf scoring:

- defType is set to 'edismax', which specifies the query parser type as Extended Dismax.
- qf (Query Fields) is set to ' description^2 title', which assigns a higher weight (2) to the description field compared to the title field.
- pf (Phrase Fields) is set to 'title^2 description', which gives a higher weight (2) to the title field in phrase queries.
- The score field is included in the returned results using fl (Field List) parameter: 'fl': '*,score'.

These parameters affect how the search query is analysed and scored by Solr, and the resulting documents are ranked based on tf-idf scoring. The higher weights assigned to the title field indicate its greater importance in the ranking.

Additionally, the code retrieves the relevance score for each document in the *extracted_result* dictionary which is later used to sort the search results in descending order by relevance.

- **ndcg.py and ndcg2.py**

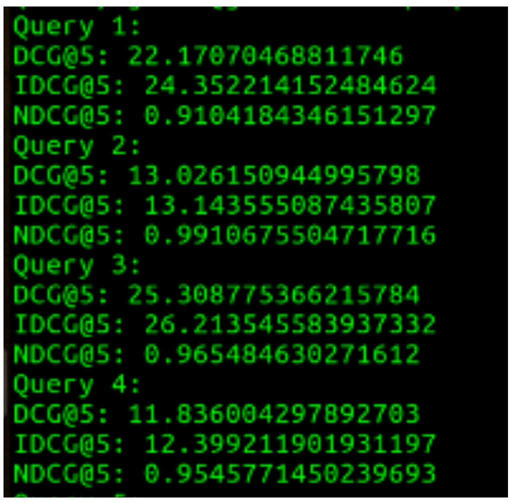
This python file was used to calculate the ndcg values of the average relevance scores given to each query. The file uses the sklearn and numpy library to perform the calculations.

- **Converting XML**

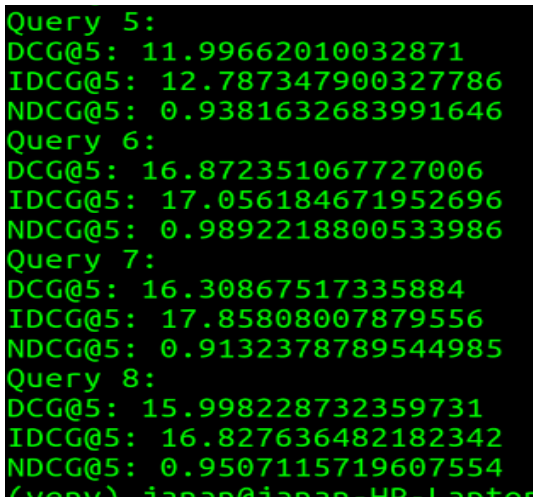
The python file that converts the collected metadata into metadata that can be indexed by Solr made use of the Beautiful Soup library. This library is used for parsing and manipulating XML documents. Going through all the XML files, the records in each of the original XML files is found and the fields in each record are extracted such as "title", "description", "creator", etc. If a field exists it is extracted and added to the new records and output in an XML file, if it does not exist it is excluded from the new XML file. These files are then in the format to be indexed by Solr.

Evaluation of Results

The "Relevance" Excel spreadsheet contains the queries and relevance judgement from each member as well as the calculated averages of the judgements and the precision calculations. The figures below show the ndcg values of the results produced by Solr before implementing field boosting.

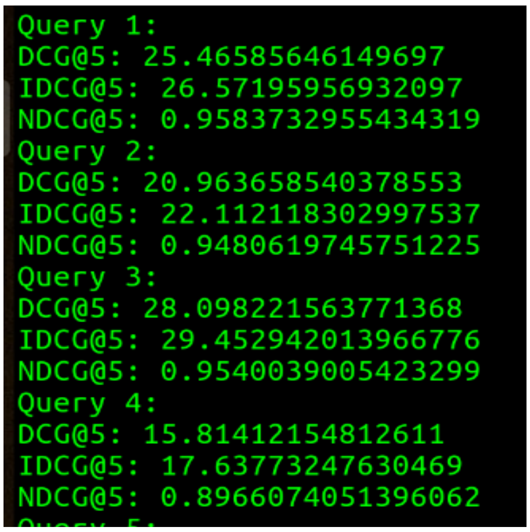


```
Query 1:
DCG@5: 22.17070468811746
IDCG@5: 24.352214152484624
NDCG@5: 0.9104184346151297
Query 2:
DCG@5: 13.026150944995798
IDCG@5: 13.143555087435807
NDCG@5: 0.9910675504717716
Query 3:
DCG@5: 25.308775366215784
IDCG@5: 26.213545583937332
NDCG@5: 0.965484630271612
Query 4:
DCG@5: 11.836004297892703
IDCG@5: 12.399211901931197
NDCG@5: 0.9545771450239693
```

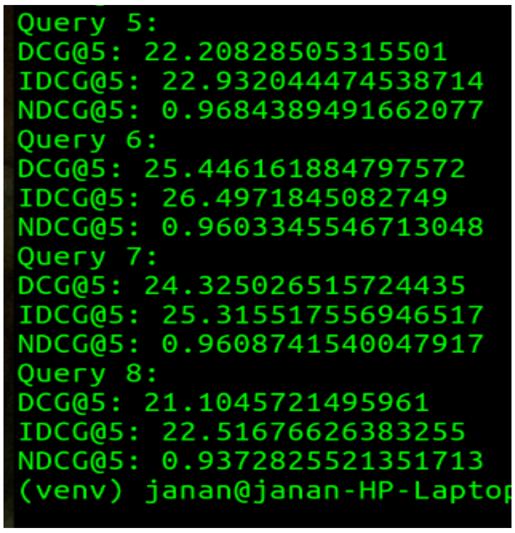


```
Query 5:
DCG@5: 11.99662010032871
IDCG@5: 12.787347900327786
NDCG@5: 0.9381632683991646
Query 6:
DCG@5: 16.872351067727006
IDCG@5: 17.056184671952696
NDCG@5: 0.9892218800533986
Query 7:
DCG@5: 16.30867517335884
IDCG@5: 17.85808007879556
NDCG@5: 0.9132378789544985
Query 8:
DCG@5: 15.998228732359731
IDCG@5: 16.827636482182342
NDCG@5: 0.9507115719607554
```

The figures below depict the ndcg values of the results produced after implementing field boosting:



```
Query 1:
DCG@5: 25.46585646149697
IDCG@5: 26.57195956932097
NDCG@5: 0.9583732955434319
Query 2:
DCG@5: 20.963658540378553
IDCG@5: 22.112118302997537
NDCG@5: 0.9480619745751225
Query 3:
DCG@5: 28.098221563771368
IDCG@5: 29.452942013966776
NDCG@5: 0.9540039005423299
Query 4:
DCG@5: 15.81412154812611
IDCG@5: 17.63773247630469
NDCG@5: 0.8966074051396062
```



```
Query 5:
DCG@5: 22.20828505315501
IDCG@5: 22.932044474538714
NDCG@5: 0.9684389491662077
Query 6:
DCG@5: 25.446161884797572
IDCG@5: 26.4971845082749
NDCG@5: 0.9603345546713048
Query 7:
DCG@5: 24.325026515724435
IDCG@5: 25.315517556946517
NDCG@5: 0.9608741540047917
Query 8:
DCG@5: 21.1045721495961
IDCG@5: 22.51676626383255
NDCG@5: 0.9372825521351713
(venv) janan@janan-HP-Laptop
```

As shown in the above figures, the quality of ranking was good in both the before and after cases. With high ndcg values, we can deduce that the search algorithms effectively managed to capture the relevance of the retrieved list items. Using even just the default scoring, the ranking algorithms successfully identifies and places the relevant results at the top. The values used were averages of 3 independent relevance judgements making the results more concrete and less biased. We can deduce that users of the system will find relevant and desired papers at the top thus improving efficiency and user experience.

Table 1: Relevance judgements before tf-idf boost

Query 7:	Black Holes		
ResultNo:	Kristen	Fardoza	Sabiha
1	0	0	0
2	5	5	5
3	4	4	4
4	4	3	3
5	0	1	2
6	4	5	4
7	3	3	5
8	3	2	4
9	4	4	3
10	3	3	3
11	2	2	4
12	2	3	5
13			
14			
15			
16			
17			
18			
19			
20			

Table 2: Relevance judgements after tf-idf boost

Query 7:	Black Holes		
ResultNo:	Kristen	Fardoza	Sabiha
1	0	0	0
2	4	4	4
3	5	5	5
4	5	5	5
5	5	5	5
6	5	5	5
7	5	5	5
8	5	5	4
9	5	5	4
10	5	4	3
11	0	2	1
12	4	4	3
13	0	0	0
14	5	5	3
15	0	0	0
16	0	0	0
17	0	0	0
18	4	5	3
19	0	0	0
20	4	5	3

A major difference noted by the weight change is that now the retrieved results list length is not limited as can be seen in the above tables. The default scoring restricts the retrieved results as seen in the graph below. The query for “Black Holes” only matched and returned 12 documents before. 20 results always get displayed after implementing field-boosting (The displayed results were capped to 20). The orange line on the graph below depicts that there may or may not be relevant documents returned in later results. These are the documents the original algorithm did not determine as relevant but is considered relevant by the user which get displayed. Furthermore, higher relevant results were seen to show up in the “after” situation. This was a trend noticed from all the queries.

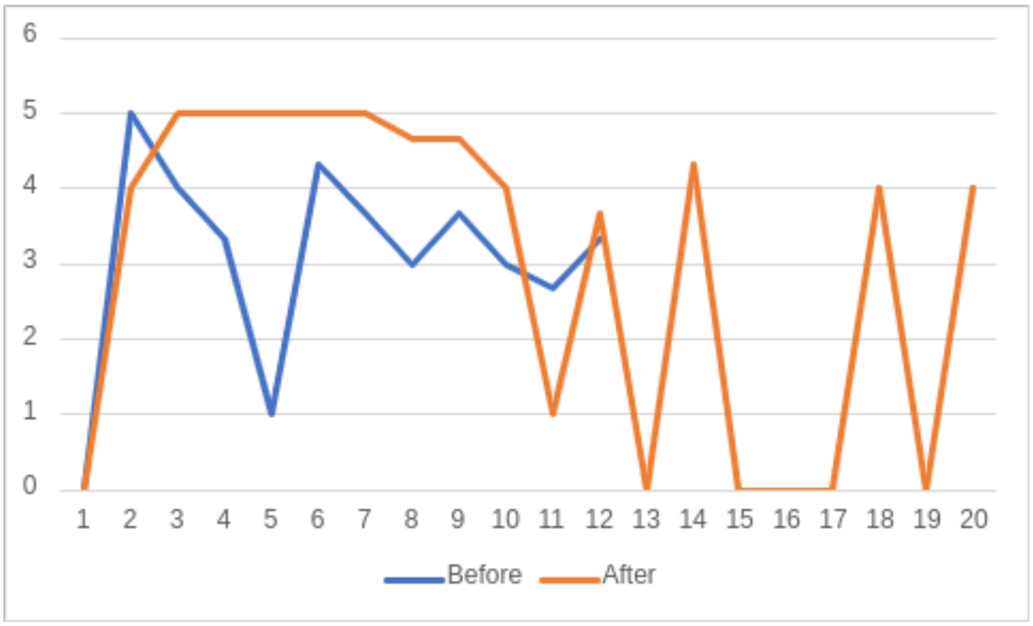


Figure 1: A graph that depicts the relevance judgements for each result displayed from a scale 0 -5 with 5 being the most relevant

There were outliers where relevant documents appeared later in the results list in the after algorithm. Playing around with the weights given to features such as title and description gave us different results. In our case a higher weight was given to description for query fields and a higher weight to title for phrase fields. Further experimentation should be conducted to determine the optimal setting in the new algorithm. However, the current algorithm returns satisfactory results.

MAP

The MAP (Mean Average Precision) was calculated for both the before and after results. The relevance values from 0 to 2 for the MAP calculations were taken as irrelevant while the values 3 to 5 were taken as relevant. The MAP for the before results was 0.89 and the after results was 0.87. Considering there were more documents recalled and present in the MAP calculations after from the boosting of the tf-idf scoring, the MAP only decreased by 0.2.

Additional Features

- **Spell checker**

A spell checker was added to the system to improve the user experience. Users that make typical typing errors or misspell words in their search queries will benefit from the usage of a spell checker. The search engine will deliver more accurate and relevant search results and improve user experience by automatically correcting spellings.

The Python pyspellchecker library was used to correct misspelled queries and add a spell checker. A SpellChecker class instance is generated as *spell* after importing the pyspellchecker library. The query provided by the user is broken up into words by the *correct_spellings* function, which then spell checks each word individually. The individual words are then joined to create a corrected query string.

When the /search endpoint is called, the *correct_spellings* function is called to correct the spellings of the query before performing the search with the corrected query.

- **Highlighting**

Highlighting is beneficial because the search engine directs the user's focus to the precise sections of the text that are most relevant to their search query by putting emphasize on these terms within the search results. This visual focus allows users to easily recognize and evaluate the importance of the search results. Highlighting also enables users to check the relevance of a search result without having to read the entire document. Additionally, it immediately gives users context about why a specific result is being displayed. Users can easily check if the material satisfies their information needs by scanning the highlighted sections.

To highlight the matching phrases or terms in the search results, the *displayResults* function was modified. A function called *highlightMatches* is added that takes a user query and an element to highlight. It performs a global case-insensitive match for the query using a regular expression with the 'gi' flags. The replace method is used to wrap the matching phrases or terms in a span element with a highlight class, which is styled using CSS to provide visual highlighting.

- **Multilanguage search**

By providing Multilanguage search, the Apache Solr search engine can evaluate and index ETD information in several languages. This means that the search engine will retrieve results in the user's query language. The search engine may provide more accurate search results with increased precision and recall by understanding the language context of the query and the indexed information. Multilanguage search also improves user convenience by enabling users to search for ETD metadata using search queries in their native languages and eliminates language barriers. Users can express their information demands more accurately and easily locate relevant content without needing to translate their queries into a specific language. The languages supported by the system currently are English, French, and Spanish.

To implement Multilanguage search, a dropdown to choose the language is added to the front.html file, and the form is changed to include the language parameter in the query string. The *perform_solr_search* function in the app.py file was modified to include the language parameter in the query. The *performSearch* function was also modified to get the language from the dropdown and include it in the query string.

- **Displaying top news headlines**

By displaying top news articles relevant to the user’s query, the search engine provides additional context and relevance to the user's information needs. This allows users to keep up with the most recent developments by viewing recent news items or updates relevant to their query. This also enables users to explore interdisciplinary connections and gain a broader understanding of their research area as well as provide a convenient way to access timely information without having to rely on separate news portals or sources. By incorporating top news headlines, the search engine enhances user engagement, encouraging users to spend more time exploring the search results. This broadens the information perspective for users, exposing them to a variety of sources, including academic research and real-time news, fostering a more comprehensive understanding of their topic of interest.

To display the top news headlines, News API is used. It is a straightforward Rest API that can be used to get live content from any website. We may retrieve a nation's top headlines from a specific source, such as the Times of India, Hindustan Times, BBC News, and many more. Additionally, we may get articles related to a particular topic.

To implement this, the Python NewsApiClient class was imported from the newsapi module, and an instance of this class is created and initialized with a NewsAPI key. A new route is then created to the Flask App to handle the news headlines request. A *get_top_headlines* function is then added to retrieve the top news headlines using NewsAPI. The index function is then modified to pass the news headlines to the render_template function. Lastly, the front.html and styles.css is modified to display the news headlines.

- **Voice Search**

Voice search provides a more intuitive and natural way for users to interact with the search engine, making it more accessible and convenient. It enables users to effectively articulate their queries more quickly than typing, leading to quicker search results. It also provides accessibility for users who struggle with typing due to disabilities or conditions that make typing challenging. It is also useful for users who are on the move or in situations where typing may not be convenient. Voice-based navigation enhances interactivity and user control, offering a more engaging and immersive search experience.

To implement voice search, a microphone button is added next to the search button which activates the *webkitSpeechRecognition* object when clicked and starts listening for speech. When speech is detected, it executes the *performSearch* function with the transcript as the query argument and sets the value of the search input to the recognized transcript. An AJAX request is sent by the *performSearch* function to the server to perform the search and then calls the *displayResults* function to display the search results.

- **Sort**

A sorting feature was added to enhance the usability and efficiency of the search engine by enabling data to be organized according to the users' preferences. Searching for papers is made simpler creating a more natural and user-friendly experience.

The sorting feature was implemented by extracting the retrieved results list as an array. Built in Python sorting methods were used to sort the results to the desired outcomes with each being their own method. The options are displayed in a menu on the web interface and depending on how the user chooses to sort the results, the appropriate method will be called. The results can be sorted by either date or title in either ascending or descending order.

The four sort methods are *sortResultsByName*, *sortResultsByNameDesc*, *sortResultsByYear* and *sortResultsByYearDesc* in the JavaScript file. An arrow function is used with a *localeCompare* method inside it to make the appropriate comparisons.

- **Filter**

The ability to filter a large volume of results bases on predefined criteria's is a crucial web interface component. Users can refine and narrow down the retrieved results hence enabling efficient data exploration and an overall pleasant user experience. Desired information is displayed quickly and easily.

To enable filtering, a user can enter optional parameters to the search engine. If these extra details are entered, the search engine will append them to the query and perform a search on that basis. The results returned adhere to the chosen filter. The user can choose to filter the results using the creators name or the year they wish to obtain results from.

The JavaScript file takes in the values for the query, year, creator, and language. The year and creator are initialised as *None* as default values in the python file. If a user fills them in those values will replace the default values. The *perform_solr_search* function will append the year and/or creator to the search query if the values are given.

- **Autocomplete**

Autocomplete predicts and suggests possible search queries from the titles of the dataset. Autocomplete reduces errors and enables discoverability thus improving the user experience. It streamlines the search entry process making it more convenient for the user.

The *jQuery UI library*, which has the *autocomplete* functionality was added to the html file to aid with adding this functionality. The function is called in the JavaScript file which connects to the */autocomplete* url in the python file. The *autocomplete* and *get_auto_complete_suggestions* functions in the python file extract titles and matches it with the user's query input.