WikiSearch
$< RetrievalRangers >$
CENG 596 Final Report

Berk Güler
berk.guler@metu.edu.tr

Mustafa Barış Emektar
baris.emektar@metu.edu.tr

Abstract

In this project, an ad-hoc information retrieval system using the $\text{ir}_{datasets} : DPRWiki100$ dataset is contructed by utilizing PyTerrier framework. NLTK library is employed for query processing since they include functions such as tokenization and stemming. Initially, basic BM25 ranking scheme is employed the for retrieval. Then, its performance is enhanced by incorporating optimizations. Furthermore, the system has a minimalistic user interface built using the tkinter library, which provides an intuitive experience for users.

Implementation: The code is available here: WikiSearch

1 Introduction

In the vast landscape of digital information, the challenge of efficiently accessing relevant data is increasing. Whether for academic research, professional pursuits or personal curiosity, the ability to quickly access relevant information among the vast sea of available content is crucial. This overarching problem has been the primary motivation for this information retrieval project.

Within this course project, the aim was to address a specific aspect of this grand challenge: optimizing ad-hoc information retrieval systems. Leveraging the rich resources of the DPR Wiki100 dataset and the versatile PyTerrier framework, the effort was focused on building an efficient information retrieval system. The approach involved using the powerful NLTK library for query processing and enabling core functionalities such as tokenization and stemming. Moreover, the inclusion of a minimalistic user interface, facilitated by the **tkinter** library, ensures an intuitive experience for users, further enhancing the practical utility of our system.

At its core, the project attempted to address the small but important problem of improving the performance of the core BM25 ranking schemes. While existing retrieval methods provided a baseline, it was realized that there was a large room for improvement and optimization. It is aimed to improve the effectiveness and efficiency of a simple information retrieval system by making various enhancements and optimizations.

What makes this project particularly interesting is its practical importance. In an age dominated by data, the ability to access relevant information quickly and accurately is invaluable in many fields. Moreover, the fact that it can process such a large amount of data (almost 5 GB) and provide the user with results quickly and accurately makes this project even more valuable.

Moving forward, this report will delve into the methodology employed in constructing our information retrieval system, detailing the various components and techniques utilized. Then, it will present an in-depth analysis of the performance enhancements achieved through our optimizations, supported by empirical results and comparative evaluations. Finally, it will discuss the implications of our findings, highlighting avenues for future research and potential extensions of our work.

2 Related Work

No additional papers are used. However, the sample projects Pyterrier shared in the documentation were useful for the development of the system.

3 System Architecture

The development of the project is done in Python. Pyterrier and NLTK libraries are used. Pyterrier library is used in the indexing the dataset,as well as the ranking and performance evaluation phases. On the other side, NLTK library is used for other operations like query transformation.

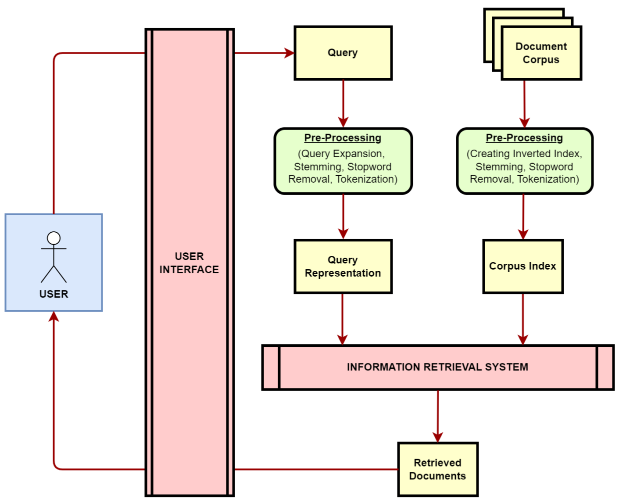

Query
Document Corpus
Pre-Processing
(Query Expansion, Stemming, Stopword Removal, Tokenization)
Pre-Processing
(Creating Inverted Index, Stemming, Stopword Removal, Tokenization)
USER
USER INTERFACE
Query Representation
Corpus Index
INFORMATION RETRIEVAL SYSTEM
Retrieved Documents

Figure 1. System Architecture and Data Flow of The System

3.1 Data Preprocessing

The numerical details of the dataset are shown below:

- Number of documents: 21,015,324
- Number of terms: 4,023,297
- Number of postings: 1,000,890,089
- Number of fields: 2
- Number of tokens: 1,286,954,378

These values are obtained using the *getCollectionStatistics()* method of Pyterrier. This operation is done in *GetStats.py* file.

As it can be seen from the values given, the dpr-wiki100 dataset is a very large dataset which is almost 5GB. This resulted in runtimes of up to hours in stages such as indexing or evaluation. Therefore, at many points in the project, work had to be carried out using smaller datasets. Other datasets used for development purposes in the project are shown below:

- $ir_{datasets} : Vaswani$
- $ir_{datasets} : NFCorpus(NutritionFacts)$

3.2 Indexing component

BM25 was chosen as the information retrieval model for its proven effectiveness across diverse datasets and retrieval tasks, its scalability to handle large document collections efficiently.

Pyterrier's *IterDictIndexer* function was used for dataset indexing. It stores the unique terms (or tokens) present in the corpus along with their corresponding metadata which includes information such as the document frequency (DF) and pointers to the postings list.

The postings included in the index created using *IterDictIndexer* stores the positional or locational information of each term within the documents. This includes the document IDs where each term occurs, along with any additional information such as term frequency, positions, or weights. Essentially, the postings list provides the necessary information for quickly locating documents containing specific terms during the retrieval process.

In this project, two different indices are created. The index generated using the *IterDictIndexer* function in *DataIndexer.py* is the index with stopword removal(TerrierStopwords) stemmer(TerrierStemmer) and tokeniser(TerrierTokeniser) applied. Another index was created in *DataIndexerNonModified.py* without any modification such as stemming and stopword removal. The Figure 2 shows the flow of the indexing process.

In the *SampleQuery.py*, **BM25 model** is applied to two different indices with *BatchRetrieve* function. Then a sample query was run using *.transform* member of the model. The query and the first five results are shown in the Table 1. Note that this example is given only to better explain the indexing phase. More detailed operations are described in the following steps.

Finally, the system does not support additional indices such as meta-data and wild-card querying

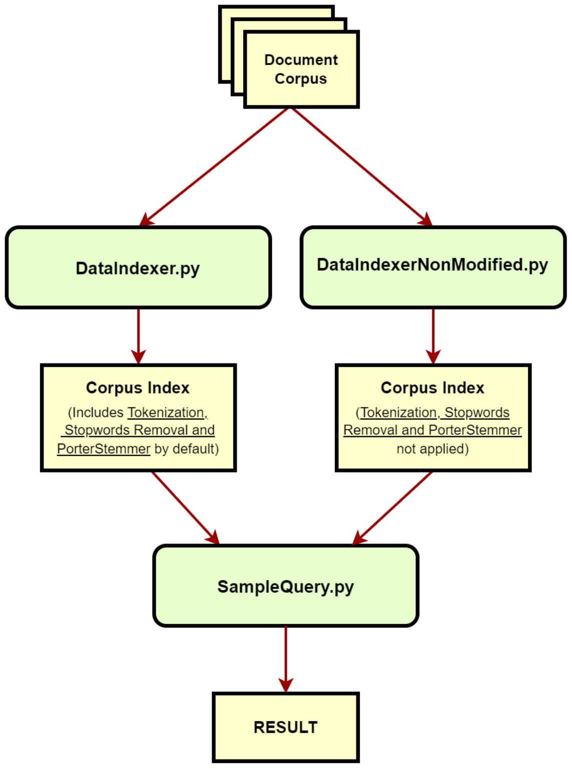

Document Corpus
DataIndexer.py
DataIndexerNonModified.py
Corpus Index
(Includes Tokenization, Stopwords Removal and PorterStemmer by default)
Corpus Index
(Tokenization, Stopwords Removal and PorterStemmer not applied)
SampleQuery.py
RESULT

Figure 2. Flow of The Indexing Process

Query = "Turkey is a predominantly mountainous country, and true lowland is confined to the coastal fringes."

Results:

Title	Text
"Geography of Turkey"	"True lowlands are confined to the "Ergene Ovası" (Ergene Plain) in Thrace, ..."
"Anatolia"	"... converge in the east. True lowland is confined to a few narrow coastal strips along the ..."
"Geology of Turkey"	"Geology of Turkey The geology of Turkey is the product of a wide variety of tectonic ..."
"Alt Camp"	"... is divided into two main topographical areas. The northeastern part is mountainous..."
"Geography of Colombia"	"...the only South American country which borders both the Atlantic and Pacific Oceans. ..."

Table 1. Result of the Query Using Modified Index

3.3 Query Processing/Searching component

After the system receives a query as input, query operations are performed with the help of *QueryTransform.py* class. Query Operations includes the followings:

- **Tokenization**: It breaks down the input query into individual words or tokens.
- **Stopword Removal**: It filters out common words (such as "the," "and," "is") that don't carry significant meaning for information retrieval.
- **Stemming**: It reduces words to their root form (e.g., "running" becomes "run") to improve search accuracy.
- **Punctuation Removal**: Removes punctuation marks from the query to simplify processing.

The **nltk** library was utilized to carry out these operations. The detailed flow of the system is shown in the Figure 3.

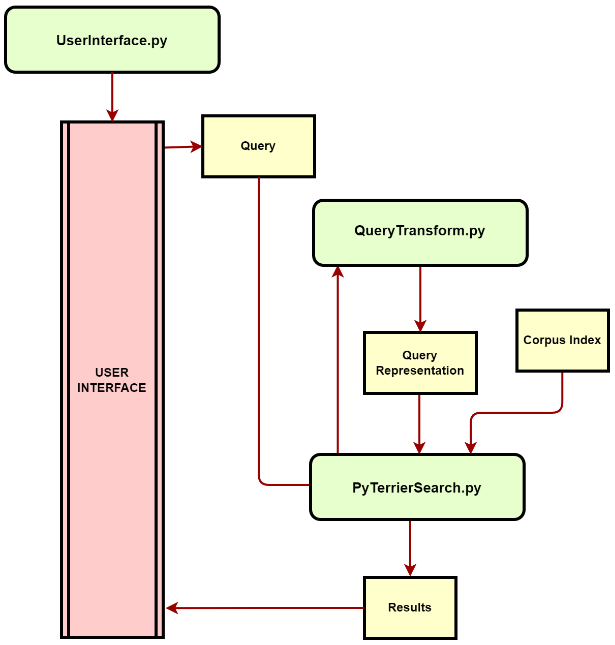


Figure 3. Flow of the System Including User Interface and Query Transform Operations

When retrieving the results for the query, the process is to generate the scores of the documents. In this way, all documents are sorted from the document with the highest score to the document with the lowest score. In fact, the documents shown in the user interface are in this order.

Besides the operations discussed above, query operations also support the **exact phrase query** capability. This capability is discussed in more detail under the User Interface section.

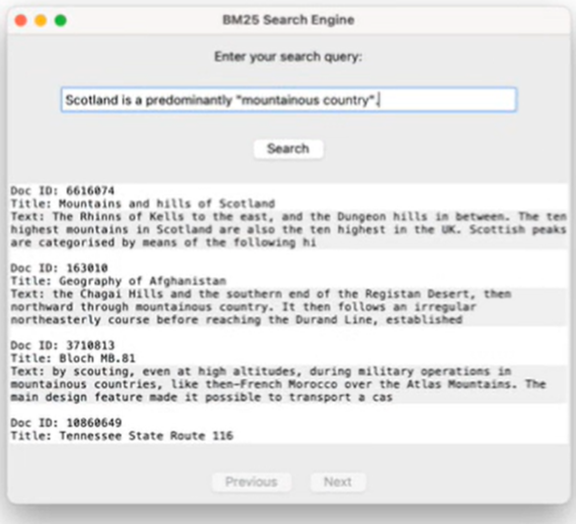
Finally, the system supports the **query expansion**. Pseudo relevance feedback is activated by passing the parameter *controls={"qe" : "on"}* when calling the *BatchRetrieve* function.

3.4 User Interface

One of Python's libraries, **tkinter** , was used to create the user interface. The user interface has a search bar with the search results listed below it. Each document has a name and a snippet from the document. Also, the user interface supports paging. The query results are divided into pages with 10 documents per page and the user can navigate back and forth between pages.

In the user interface, users can use standard queries as well as exact phrase queries. When a query is submitted, the query is first split into two: The part inside the quotation marks and the part outside the quotation marks. While the part outside the quotation marks goes through the operations mentioned under the query processing heading, no operation is performed on the part inside the quotation marks. In this way, exact phrase queries are supported

Finally, the system supports **pseudo relevance feedback**. Detailed information is given under Query Processing section.



```
Doc ID: 6616074
Title: Mountains and hills of Scotland
Text: The Rhinns of Kells to the east, and the Dungeon hills in between. The ten
highest mountains in Scotland are also the ten highest in the UK. Scottish peaks
are categorised by means of the following hi

Doc ID: 163010
Title: Geography of Afghanistan
Text: the Chagai Hills and the southern end of the Registan Desert, then
northward through mountainous country. It then follows an irregular
northeasterly course before reaching the Durand Line, established

Doc ID: 3710813
Title: Bloch MB.81
Text: by scouting, even at high altitudes, during military operations in
mountainous countries, like then-French Morocco over the Atlas Mountains. The
main design feature made it possible to transport a cas

Doc ID: 10860649
Title: Tennessee State Route 116
```

Figure 4. Screenshot of the User Interface

At this point, it is worth mentioning a problem that occurred while developing the user interface. Since name and content queries cannot be made on indexed data; all documents need to be stored in RAM in order to show the documents to the user with their names and contents. Since the compressed version of our dataset is around 4.5 GB, it takes about 2 minutes to get it into RAM. In order to make the user interface more effective, it was necessary to find a solution to this problem with an engineering instinct.

The $\mathrm{ir}_{datasets} : DPRWiki100$ dataset has around 21 million documents and is about 12GB in size without compression. The size of this dataset had to be reduced a bit. For this, we first tried to get the first 300,000 lines of the dataset with a small tool we found on the internet. However, in this case, we realized that since the dataset had a sequential structure, all documents consisted of documents whose titles started with the letter "A". We then decided to create the python files *splitter.py* and *compressor.py* where we first shuffle the dataset (helps to create a more homogeneous dataset), then extract and compress the first 300,000 lines. This allowed the user interface to stand up more quickly.

4 Evaluation

As mentioned earlier, two different indexes were created in the system. One of the indexes is the index with stopword removal and stemming operations applied and the other one is not. A small illustration is shown in Figure 5.

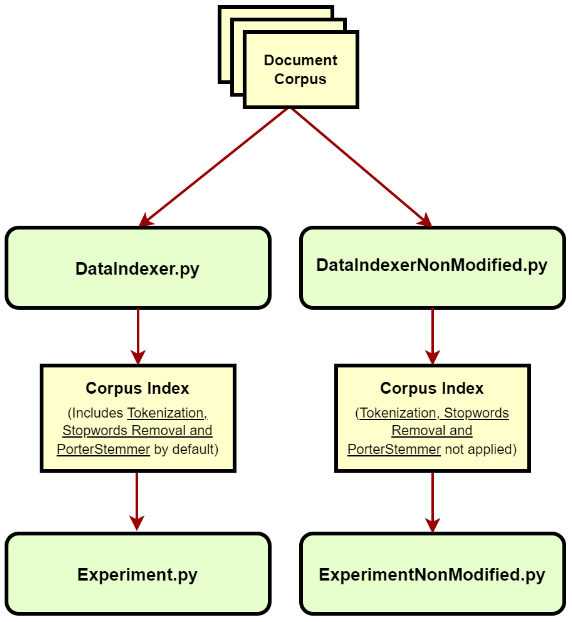

Document Corpus
DataIndexer.py
DataIndexerNonModified.py
Corpus Index
(Includes Tokenization, Stopwords Removal and PorterStemmer by default)
Corpus Index
(Tokenization, Stopwords Removal and PorterStemmer not applied)
Experiment.py
ExperimentNonModified.py

Figure 5. Flow of The Evaluation Process

For these indices, we calculated the performance of the system based on Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG@20) metrics using the *Experiment* function separately.

The effectiveness of the system was evaluated using Mean Average Precision and Normalized Discounted Cumulative Gain.

	AP	**NDCG@20**
BM25 (Non-Modified Index)	0.151913	0.195565
BM25 (Modified Index)	0.208434	0.269578
BM25 (Modified Index with Q.E.)	0.223405	0.281933

Table 2. Result of the Experiments with Two Different Indices

As seen in Table 2, the performance measurements with the modified index (stopword removals and stemming) are almost 1.5 times better.

It can be observed that adding query expansion on top of the modified index improves the performance a bit more.

Also, NDCG@20 performance was slightly better than MAP performance. That means, the system performs well in bringing the most relevant documents to the top 20 positions, it has a lower MAP because the relevant documents are not consistently high across the entire ranking list.

5 Conclusion

To conclude, in this project we created a small search engine using the $\mathrm{ir}_{datasets} : DPRWiki100$ dataset. In the development phase of this project, we used the NLTK, Pyterrier and tkinter libraries in Python, resulting in a small search engine that supports certain capabilities.

We can discuss the things we planned to do in the Proposal report under 4 headings:

- **Data parsing and indexing**: During the dataset indexing phase, we fulfilled all the promises made in the proposal such as Tokenization, Stopword Removal, Stemming, Punctuation Removal.
- **Query processing**: We perform query operations like tokenization, stopword removal, stemming. Also, query operations support the exact phrase query type. Moreover, query expansion is also implemented. On the other side, we did not have time to add the feature of supporting other query types (boolean query, wildcard) specified in the proposal report to our project.
- **Ranking and retrieval**: As stated in the proposal report, the documents were evaluated according to BM25. We also implemented pseudo-relevance feedback along with query expansion. However, we did not implement user based relevance feedback.
- **User Interface**: We made a functional and simple user interface with paging, where the user can navigate between pages consisting of the results of the query sorted by score. It also supports exact phrase query as stated in the proposal report. Although it doesn't require much effort, we didn't have time to add other types of search methods because we had the unforeseen problem mentioned in the User Interface subsection.

As mentioned in the system architecture section, there are a few parts of our project that are open for improvement:

- Support for wildcard and proximity operators for queries can be added. This way the user can make more specific searches
- In further steps, the snippets of the documents resulting from the query can be updated to show the most relevant part of the document.

It was a pleasure to be part of this project. Rediscovering the technologies that underpin the search engines we use every day has reawakened our curiosity.

We have been working on the project with increasing intensity at certain periods. Especially the struggle with the Pyterrier library will never be forgotten. The Pyterrier documentation provides enough foundation to get started. However, since the number of library users is relatively small, the maturity level of the library remains at a certain point. For example, since the two of us were using different operating systems, we got different and varied errors. Since there were not many discussions/examples in the forums, it took us a long time to find solutions to the errors.

We also had major problems with the dataset we chose. The compressed version of the dataset we used is ~4.5 GB and contains ~21 million documents. A dataset of this size is very difficult for an ordinary computer to handle. Therefore, we experienced waiting times of up to hours and had to develop with different smaller datasets.

Finally, we think that the project contributed a lot to the course subjects. We had the chance to experience the practical implementations of the subjects taught. However, we would have liked to do an even more in-depth practice if there had not been technical problems in the project.

References

[1] [n.d.]. NLTK — Natural Language Toolkit. https://www.nltk.org/
[2] [n.d.]. Pyterrier: A Python framework for performing information retrieval experiments. https://github.com/terrier-org/pyterrier
[3] [n.d.]. PyTerrier's documentation 0.10.1 . https://pyterrier.readthedocs.io/en/latest/
[4] [n.d.]. tkinter Python Documentation. https://docs.python.org/3/library/tkinter.html
[5] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. EMNLP 2020. DOI: https://doi.org/10.18653/v1/2020.emnlp-main.550.

[1] [2] [3] [4] [5]