
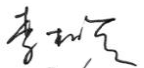

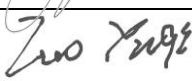


Attn: Dr. Sun Aixin



AI6122 Text Data Management and Processing

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below. We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work. We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work.

Name	Signature / Date
Wang Zijian	 Oct 31, 2023
Li Muhan	 Oct 31, 2023
Bugnot Reinelle Jan Cruz	 Oct 31, 2023
Zuo Yuqi	 Oct 31, 2023

Important note:

Name must **EXACTLY MATCH** the one printed on your Matriculation Card. Any mismatch leads to **THREE (3)** marks deduction.

AI6122 Text Data Management and Processing - Group Project

Li Muhan
Nanyang Technological University
Singapore
G2303579B
limu0006@e.ntu.edu.sg

Zuo Yuqi
Nanyang Technological University
Singapore
G2302844D
zuoy0004@e.ntu.edu.sg

Reinelle Jan Bugnot
Nanyang Technological University
Singapore
G2304329L
bugn0001@e.ntu.edu.sg

Wang Zijian
Nanyang Technological University
Singapore
G2303897K
wang2009@e.ntu.edu.sg

ABSTRACT

This project is a comprehensive exploration of an end-to-end text data management and processing application, structured into four components, including domain-specific dataset analysis, the development of a simple search engine, research trend exploration, and the creation of a practical application interface. The goal of this project is to create a simple search engine command line application for the DBLP computer science publications bibliography dataset, implementing various strategies in text data processing, indexing, information retrieval, and data exploration. The primary programming language we chose to implement the project is Python, in order to take advantage of available open source Python packages for natural language processing tasks. Attached with this paper is the source code representing the final project output.

CCS CONCEPTS

• **Information systems** → **Information retrieval query processing**; **Document representation**; *Content analysis and feature selection*.

KEYWORDS

NLP, dataset analysis, information retrieval, search engine, research trend, python

ACM Reference Format:

Li Muhan, Reinelle Jan Bugnot, Zuo Yuqi, and Wang Zijian. 2023. AI6122 Text Data Management and Processing - Group Project. In *AI6122 (Text Data Management & Processing)*. S.Y. 2023-2024, Nanyang Technological University, Singapore, 10 pages.

1 INTRODUCTION

In this project, we organize this report as follows. We first introduce some domain specific dataset analysis, consisting of tokenization and stemming, sentence segmentation and POS tagging. Next, we introduce how to develop a simple search engine, including dataset

parsing, indexing and information retrieval. Subsequently, we introduce two methods to investigate the research trend: tf-idf and textrank. At last, define the application to be able to search similar research papers by title, based on traditional IR methods and neural network. In our group, Zuo Yuqi is responsible for section 2 and tf-idf method in section 4. Reinelle Jan Bugnot is responsible for section 3. Wang Zijian is responsible for textrank method in section 4. Li Muhan is responsible for Lucene implementation in section 3 and 5.

2 DOMAIN SPECIFIC DATASET ANALYSIS

In this part, we form three datasets and analyse from different perspectives. An overview of the structure of the datasets are as follows:

- (1) **Chemical**: It contains 10 pdf files about chemical paper appendices of experiments. It is mainly collected from ACS website.
- (2) **Jet**: It contains 10 pdf files about Jet patents. It is mainly collected from google patent website.
- (3) **Stackoverflow**: It contains 10 pdf files about questions and answers from Stackoverflow. It is mainly collected from Stackoverflow website.

After forming datasets. We analyse these datasets from three different perspectives, which are shown in the subsections.

2.1 Tokenization and Stemming

Tokenization focuses on dividing text into tokens, which contain words, phrases, symbols and digits. Tokenization helps transform text into a format that can be understood by a computer, making text amenable to computational processing. **Stemming**, on the other hand, focuses on reducing words to their root form. Stemming removes suffixes from words to standardize them to a common form.

In this project, we used the *NLTK* library to do the basic tokenization, and we also used the *PorterStemmer()* module from *NLTK* to do the basic stemming.

- (1) **Chemical** dataset token distribution shown in Fig. 1a.
- (2) **Jet** dataset token distribution shown in Fig. 1b.
- (3) **StackOverflow** dataset token distribution shown in Fig. 1c.

Analysis: from the figures and the data, we can see the tokens distribution follow the rule, which means the token with shorter

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Text Data Management & Processing, October 2023, Singapore
© 2023 Copyright held by the owner/author(s).

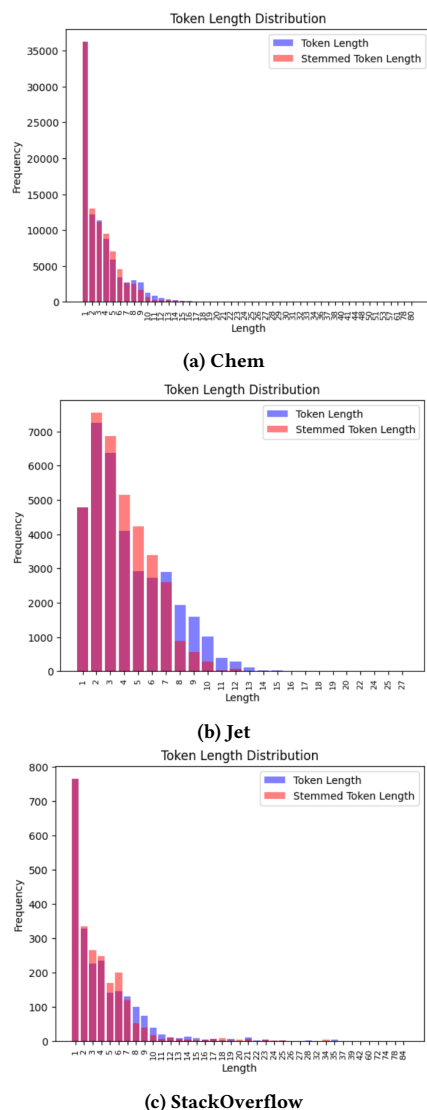


Figure 1: Token Distribution Before and After Stemming

length has higher frequency, and the token with longer length has lower frequency. Most tokens are relatively short so the frequency distribution faces an exponential decrease when length increases. Through the analysis when printing each token, we discovered that the `word_tokenizer()` module of *NLTK* has good performance on domain specific tokenization for our datasets. For example, it successfully tokenizes the RNA Sequences like `'atgggtctctggcg-gagttcaaacagcgtctggcgcgatcaaaacccgt'` for Chemical dataset, and coding libraries like `'Tensorflow'` for Stackoverflow dataset.

However, when we tokenize math equations from the Chemical and Jet datasets, it separates every symbol and digit. For example: `'J', '=', '2.5', 'Hz'`. This kind of tokenization, while excellent in the domains of the previous datasets, may cause language models

(like DeBERTa, LLaMa, etc) to have difficulty in understanding the underlying semantic meaning of mathematical equations. Hence, we may use *MathJax* or *SymPy* to solve this problem.

2.2 Sentence Segmentation

Sentence Segmentation focuses on segmenting tokenized text into sentences. It helps text analysis and text management with fine-grained level. We show our sentence segmentation result in 3 datasets in Fig. 2.

Analysis: From the Fig. 2, we discovered that the lengths of most sentences are lower than 50. We notice that the Stackoverflow is a relatively shorter dataset, the distribution image may not be so clear and salient as other datasets. Most sentences are relatively short so the Number of Sentences distribution faces an exponential decrease when sentence length increases.

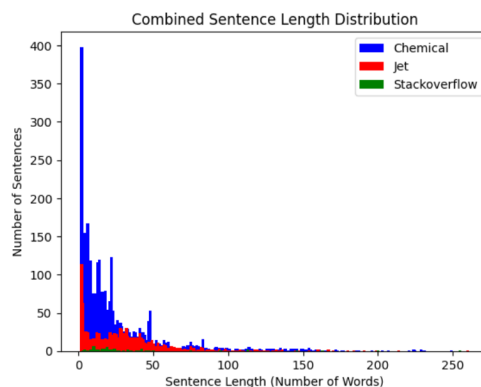


Figure 2: Sentence Segmentation

2.3 POS Tagging

After Sentence Segmentation, we apply POS Tagging on the segmented sentences. We randomly select 3 sentences from each dataset using lemmatization, and apply POS Tagging for each sentences. We then take 1 sentence from each of the sentence list from the 3 datasets to illustrate our result, as shown below:

Chemical Dataset

Sentence: The NMR spectral data agree with the literature report.

POS Tags: (NLTK POS) [('The', 'DT'), ('NMR', 'NNP'), ('spectral', 'JJ'), ('data', 'NNS'), ('agree', 'VBP'), ('with', 'IN'), ('the', 'DT'), ('literature', 'NN'), ('report', 'NN'), (',', ',')]]

Jet Dataset

Sentence: For ordinary operation the motors 42 would furnish all the torque that the propellers could absorb.

POS Tags: (NLTK POS) [('For', 'IN'), ('ordinary', 'JJ'), ('operation', 'NN'), ('the', 'DT'), ('motors', 'NNS'), ('42', 'CD'), ('would', 'MD'), ('furnish', 'VB'), ('all', 'PDT'), ('the', 'DT'), ('torque', 'NN'), ('that', 'IN'), ('the', 'DT'), ('propellers', 'NNS'), ('could', 'MD'), ('absorb', 'VB'), (',', ',')]]

StackOverflow

Sentence: Using NextJS, is there a way to set up multiple websites that looks very similar with a single repository?

POS Tags: (NLTK POS) [('Using', 'VBG'), ('NextJS', 'NNP'), (',', ','), ('is', 'VBZ'), ('there', 'RB'), ('a', 'DT'), ('way', 'NN'), ('to', 'TO'), ('set', 'VB'), ('up', 'RP'), ('multiple', 'JJ'), ('websites', 'NNS'), ('that', 'IN'), ('looks', 'VBZ'), ('very', 'RB'), ('similar', 'JJ'), ('with', 'IN'), ('a', 'DT'), ('single', 'JJ'), ('repository', 'NN'), ('?', '?')]

We can see the *NLTK* POS Tagging has good accuracy in POS Tagging for our datasets. However, when tagging for domain specific sentences, the POS Tags are a bit different with ground-truth POS Tags generated by verification program.

3 DEVELOPING A SIMPLE SEARCH ENGINE

Search Engines arguably play an indispensable role in our digital lives; from acquiring simple answers to day-to-day questions, up until finding complex or nuanced information about a variety of niche topics—all with the key purpose of satisfying information need from a collection of human-accessible knowledge. The fundamental principle behind search engines consists of two parts: (1) an index containing units of information extracted from within all the documents in the search engine's scope, and (2) an information retrieval logic that governs how the index can be used to retrieve documents given a search query [3].

In this project, we developed a simple search engine using these fundamental principles, as outlined in Fig. 3. The data collection used to feed into the search engine index is the DBLP computer science bibliography dataset by <https://dblp.org>. As of October 2023, the dataset contains approximately 6.9M bibliographic records of major and minor computer science publications written by more than 3.3 million authors [2].

3.1 Dataset Parsing

The DBLP dataset is available as one big XML file, approximately 4 GB in size. Hence, in order to access the data, we need to be able to parse the XML file. We performed this by using the *etree* module of the *lxml* python library. Since the source XML file is relatively large, parsing the entire tree in one go will require a significant amount of RAM; which is inefficient and might severely slow down our application. Thus, instead of loading the entire tree at once, we created a generator object that iteratively parses the tree using the `etree.iterparse()` function, and returns the corresponding document.

3.2 Indexing

Indexing, in the context of information retrieval, is the fundamental process of converting text data from a large collection of documents, into a structured and compact representation of accessible information [3]. It is the process used to produce an inverted index, which is a collection of units of information and their locations within a corpus of documents; serving as a bridge between the documents being searched and the users expressing information need through search queries. There are many different types of indexes, each breaking apart information and storing them in slightly different ways. To appreciate the fundamental concept behind inverted indexes, and how a well-constructed index facilitates the retrieval of specific information from a large document collection, in this

project, we created an inverted index (1) through a manually implemented positional index using only low-level python libraries, and (2) through the use of an open source library called *PyLucene*.

3.2.1 Positional Index via Manual Implementation. Creating a simple positional index for a search engine involves maintaining a list (also called *posting*) of all positions in a document a term appears in across all the documents in the corpus. Fig. 4 shows the standard format of a positional index that we used in this project.

The first step in creating a positional index is to tokenize the text from the parsed document provided by our generator object. This means breaking down the text into its individual words or subword units. We achieved this by using readily available tokenizer modules from the *NLTK* python library, specifically, the *WhitespaceTokenizer()* module. Afterwards, text normalization is applied to the tokens in order to reduce the number of unique terms in the corpus (which helps make the index more memory efficient) as well as to store information across terms semantically related tokens into a single term (which helps in the retrieval process). In this manual implementation, we used a simple *PorterStemmer* as the normalization strategy.

The next step is to create the posting list for each term, as specified in the format shown in Fig. 4. A posting list is a list of documents where the term appears, along with the positions of the term in each document. This can be done by iterating over the documents and for each term, storing the document ID and the positions of the term in the document. The positional index can also store the document frequency (the number of documents containing the term) and the total term frequency (the total number of occurrences of the term across all documents).

If a term already exists in the positional index, the document ID and the positions of the term in the document are added to the existing posting list. If the term does not exist in the positional index, a new entry is created with the term, the document ID, and the position of the term in the document. The document frequency and the total term frequency are also subsequently updated.

The pseudocode in algorithm 1 summarizes the logical flow of the steps discussed above, and describes the specific steps that we used to manually implement the positional indexing process for this project.

By recording the positional information of every term in a document corpus, we are able to retrieve documents using phrase or proximity queries of varying complexity, with improved precision, relevance, and flexibility. However, one drawback of using positional indexes is its inherent size. Positional indexes often comprises of around 35-50% of the volume of the original text from which it was generated from [3], and are twice or even four times as large as non-positional indexes. Without implementing complex data compression strategies or any other data engineering techniques that tunes for data flow efficiency, indexing and performing information retrieval on a raw index of up to 6.9M documents is relatively slow and resource-intensive, as illustrated in a simple benchmark we did as illustrated in Fig. 5a. This is especially apparent at the start of the manual indexing process when new terms are still constantly being added to the growing index, as observed in 5b. In terms of information retrieval, our manually implemented index can comfortably be used to retrieve documents from at most 3M documents

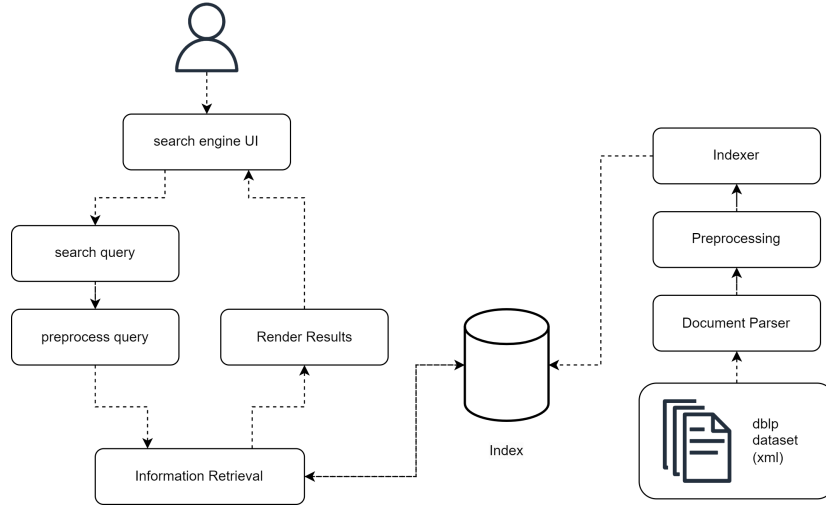


Figure 3: A Simple Search Engine Architecture

```

{ term1 : [doc_freq, {
  docID_1: [posID_1, posID_2, ...],
  docID_2: [posID_1, posID_2, ...],
  docID_3: [posID_1, posID_2, ...] }
]
term2 : [doc_freq, {
  docID_1: [posID_1, posID_2, ...],
  docID_2: [posID_1, posID_2, ...],
  docID_3: [posID_1, posID_2, ...] }
]
...
}

```

Figure 4: Positional index implementation format

on a machine with 16 GB RAM; wherein, anything past that will cause the kernel to crash. Hence, utilizing off-the-shelf information retrieval libraries, such as Java *Lucene*, is necessary to create a robust and efficient search engine.

3.2.2 Index via PyLucene. PyLucene is a Python extension for accessing Java Lucene which embeds a Java VM with Lucene into a Python process. So we mainly refer to the Lucene documentation [10] for the usage of PyLucene.

Indexing using Lucene is quite easy to implement with a few steps. First, we need to define fields and analyzers for the documents. We create an `IndexWriter` with customized analyzers as follows.¹

```

fields = HashMap()
fields.put("author", SimpleAnalyzer())
fields.put("title", StopAnalyzer(stop_words))

```

¹Codes and results are simplified for better demonstration.

Algorithm 1 Positional Index Generation [1]

```

index ← {}
set entry_count
while not exceeding entry_count do
  Generate next sample document
  Tokenize and Normalize sample document
  for term and term_position in document do
    if term not in index then
      Create entry for term
      Set document_frequency of term to 1
      Create a postings under term in index
      postings ← {document_ID : [term_position]}
    else
      Increment document_frequency by 1
      if document_ID in postings then
        postings[document_ID] ← [term_position]
      else
        postings ← {document_ID : [term_position]}
      end if
    end if
  end for
end while

```

```

fields.put("journal", WhitespaceAnalyzer())
fields.put("year", KeywordAnalyzer())
analyzer = PerFieldAnalyzerWrapper(analyzer, fields)

```

```

config = IndexWriterConfig(analyzer)
config.setOpenMode(OpenMode.CREATE)
writer = IndexWriter(store_dir, config)

```

We apply different analyzers for each field. For the author field, we use the `SimpleAnalyzer` which divides the text stream into tokens at non-letters and applies case-folding. This is because the

names are usually spaced by spaces and the case of the letters does not matter. We don't want to change the spelling of the names.

For the title field, in addition to a SimpleAnalyzer, we use the StopAnalyzer which removes stop words based on a given list. We keep a small list of stop words to avoid accidentally removing some important abbreviations.

For the journal field, we use the WhitespaceAnalyzer which only splits the texts at spaces. We are not applying case-folding or any other operations because there are some abbreviations and research field names in the journal names, and it should be kept as it is.

For the year field, we use the KeywordAnalyzer which basically does nothing. So the year as a whole will be indexed as a single token.

Then we can define the field types.

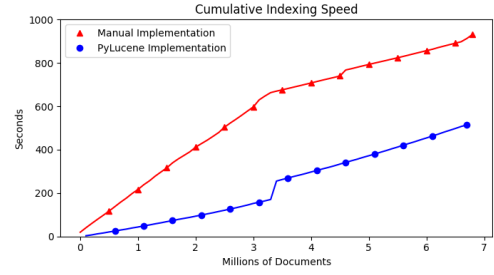
```
ft_author = FieldType()
ft_author.setStored(True)
ft_author.setTokenized(True)
ft_author.setIndexOptions(
    IndexOptions.DOCS_AND_FREQS
)
ft_title = FieldType()
ft_title.setStored(True)
ft_title.setTokenized(True)
ft_title.setIndexOptions(
    IndexOptions.DOCS_AND_FREQS_AND_POSITIONS
)
ft_journal = FieldType()
ft_journal.setStored(True)
ft_journal.setTokenized(True)
ft_journal.setIndexOptions(
    IndexOptions.DOCS_AND_FREQS
)
ft_year = FieldType()
ft_year.setStored(True)
ft_year.setTokenized(True)
ft_year.setIndexOptions(
    IndexOptions.DOCS_AND_FREQS
)
```

The setStored method indicates whether the original text of the field will be stored in the index. We keep everything stored, so we can have everything returned at once when we retrieve the results from searching. It is a trade-off between the storage space and the speed of retrieval. The setTokenized method indicates that the field will be tokenized by our previously defined analyzers, and the setIndexOptions method indicates the indexing options for the field.

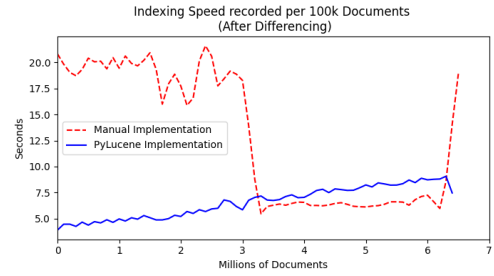
Then we can start indexing the documents by adding them to the IndexWriter.

```
doc = Document()
doc.add(Field("author", author, ft_author))
doc.add(Field("title", title, ft_title))
doc.add(Field("journal", journal, ft_journal))
doc.add(Field("year", year, ft_year))
writer.addDocument(doc)
```

The indexing process is quite fast. On my personal laptop², it takes 9 minutes to index the whole dataset. We get 6,754,140 valid documents in total, which contains at least author, title, and year fields. At the beginning of the indexing, it takes about 5 seconds to index 100k documents. As the indexed size grows, the indexing speed decreases. For the last 10% of the documents, it takes up to 9 seconds to index each 100k documents. The indexing time is shown in Fig. 5b.



(a) Indexing Time Benchmark



(b) Incremental Indexing Time Benchmark

Figure 5: Manual vs PyLucene Benchmarks

3.3 Information Retrieval

3.3.1 BM25 Ranking. The BM25 algorithm introduced by Robertson and Walker takes into account the term frequency, document length, and document frequency. The BM25 score of a document D for a query Q is defined as follows.

$$\text{score}(D, Q) = \sum_{i=1}^n \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avdl}})} \quad (1)$$

where $f(q_i, D)$ is the frequency of term q_i in document D , $|D|$ is the length of document D , and avdl is the average document length in the collection. The parameters k_1 and b are usually set to 1.2 and 0.75 respectively.

3.3.2 Retrieving Documents in PyLucene. After being indexed, each document can be retrieved by its ID as follows.

```
dir = DirectoryReader.open(store_dir)
reader = IndexSearcher(dir)
document = reader.doc(doc_id)
```

²Macbook Pro with M1 Max Processor.

The documents can be searched by indexed terms. We use the `QueryParser` to parse the query string and get a `Query` object. Then we can use the `IndexSearcher` to search the documents and get the results. By default, the ranking of the results is based on the BM25.

```
dir = DirectoryReader.open(store_dir)
searcher = IndexSearcher(dir)
query = QueryParser('title', analyzer).parse(query)
top_docs = searcher.search(query, n)
```

Lucene's `QueryParser` supports rich query syntax, including single keyword queries, phrase queries, boolean queries, wildcard queries, fuzzy queries, and even more.

3.3.3 Test Case 1: Single Keyword Query.³

```
> 2 RET
[QUERY] title:ret - in 0.0671 seconds

Result 1 (215522) <6.91>
[TITLE] Manifolds allowing RET arithmetic.

Result 2 (136327) <6.52>
[TITLE] Fuzzy Recursion, RET's, and Isols.
```

The results look good. As it's using BM25 ranking, the shorter title is preferred.

```
> 2 "structure data"
[QUERY] title:"structure data" - in 0.0300 seconds

Result 1 (101112) <4.30>
[TITLE] The inorganic crystal structure data base.

Result 2 (5038214) <4.30>
[TITLE] Natural proofs for structure, data, and separation.
```

3.3.4 *Test Case 2: Phrase Query.* The results look good. We want exact matches of the phrase, and we do not get phrases like "data structure".

3.3.5 *Test Case 3: Boolean Query.* The terms query is used to search for documents with at least one match between multiple terms. It is equivalent to a boolean query with the operator OR for each term.

```
> 2 database performance
[QUERY] title:database title:performance
      - in 0.0199 seconds

Result 1 (1589718) <5.74>
[TITLE] Database Design and Performance.
[YEAR] 1990

Result 2 (2419046) <5.74>
[TITLE] Database Performance Metrics.
[YEAR] 1989
```

³The format of showcases in this report is: [COMMAND] NUM_RESULTS QUERY → PARSED_QUERY - TIME_COST, RANK (DOC_ID) <SCORE>.

We can also use the boolean query to further search for papers published in certain years, by certain authors, or in certain journals.

```
> 2 database performance AND year:199*
[QUERY] title:database +title:performance +year:199*
      - in 0.0475 seconds
```

```
Result 1 (1589718) <6.74>
[TITLE] Database Design and Performance.
[YEAR] 1990
```

```
Result 2 (4056204) <6.74>
[TITLE] Database Performance Measurement.
[YEAR] 1997
```

3.3.6 *Test Case 4: Failed Cases.* The search engine works mostly as expected. However, there are some failed cases. For example, the following query does not return expected results.

```
> 2 bm25
[QUERY] title:bm - in 0.0430 seconds

Result 1 (584341) <7.59>
[TITLE] On the Number of Monochromatic Solutions of
      ${{\bm x}}+{{\bm y}}={{\bm z}}^{{{\bm 2}}}}$.

Result 2 (1104314) <7.35>
[TITLE] BM25.
```

We identify a few limitations here. First, the query is not parsed correctly. The query parser removes the number 25, because Lucene's `LowercaseTokenizer` treats numerics as boundaries. Second, the ranking is not satisfying. The first result contains irrelevant terms in a formula that matches the query. To solve these problems, we may need to create custom analyzers.

4 DEVELOPING A RESEARCH TREND EXPLORER

In this part, we develop a research trend explorer for conference keywords analysis. We choose ACL conference(without workshop) from the first conference year 1979 to 2023 for the analysis.

4.1 Conference Data Analysis

We processed our data from the original dataset `dblp.xml` by sifting the `<booktitle>` with ACL and then stored the data to the pandas `DataFrame`. However, we noticed that the `DataFrame` has a gap in data in the year 1984. We discovered that the ACL held a joint conference with COLING in 1984, and that `<booktitle>` doesn't contain ACL. We therefore used COLING/ACL joint conference to fill this missing data. After pre-processing, we get the conference data distribution by year as shown in Fig. 6.

We can see the accepted number of papers increases over the years. However, the quantity of papers published before 2005 is too few, making it hard to analyze by only using term frequency. Here, we used two methods for keyword selection: TF-IDF and TextRank.

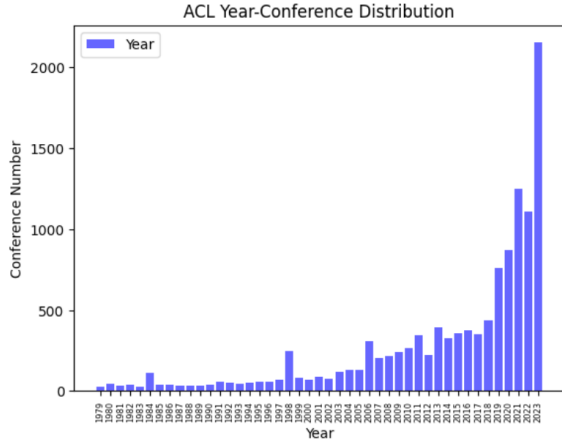


Figure 6: ACL year distribution

4.2 TF-IDF

TF-IDF is a commonly used statistic method for text mining. TF-IDF reflects the importance of a word in the whole corpus. The equation of computing TF-IDF is shown below:

$$TF - IDF(t, d) = TF(t, d) * IDF(t) \quad (2)$$

The *TF* in TF-IDF refers to Term Frequency, which represents the frequency of the term in the whole corpus. The *IDF* part, on the other hand, means Inverse Document Frequency, which indicates the rarity of the term in the whole corpus. By using TF-IDF, we hope to find terms that have a high relevance. To facilitate this, we included a logic that removes common words as included in the stopwords list.

4.3 TextRank

Apart from the above method, we also used **TextRank** in this task [7]. The general steps are similar to TF-IDF while we use textrank to rank the importance of the tokens. TextRank comes from the classic algorithm Pagerank, which is used to solve the value sorting problem of Internet web pages. The score of a web page is given by the equation below:

$$S(V_i) = (1 - d) + d * \sum_{j \in \text{In}(v_i)} \frac{1}{|\text{Out}(V_j)|} S(V_j) \quad (3)$$

where (V_i) is the weight of web page i , d is the damping facator, enabling us to jump to the other web pages randomly in case of no outgoing links, $\text{In}(v_i)$ is the set of inbound links of i and $\text{Out}(V_j)$ is the outgoing links of i .

This mutual jump relationship between web pages can be understood as a "voting" behavior. The connection of web page A to web page B indicates that web page A recognizes web page B, that is, web page A voted for web page B. The more votes (links) you give to web page B, the greater the value of web page B. Besides, the weight is the reciprocal of the number of links from a certain web page. The greater the number, the smaller the weight. In practice, we give each node an initial value, usually $1/N$, in an adjacency matrix, and iterate until convergence.

Extracting keywords by textrank is similar. The nodes in the graph are the tokens instead of the web pages. The weight of the edge is no longer calculated by the inbound and outgoing but a sliding window method.

$$w_1, w_2, w_3, w_4, w_5, \dots, w_n \quad (4)$$

Above is the sentence(titles in our task) and $[w_1, w_2, \dots, w_k]$, $[w_2, w_3, \dots, w_{k+1}]$, $[w_3, w_4, \dots, w_{k+2}]$ are the windows, where the size of window is k . Any two token pairs in the window are considered to have non-directed edges. Compared with the unweighted directed graph in PageRank, what is built here is an unweighted and non-directed graph. In the original paper, the keyword extraction task is mainly an non-directed and unweighted graph, which shows a better performance than a directed graph based on the order of tokens.

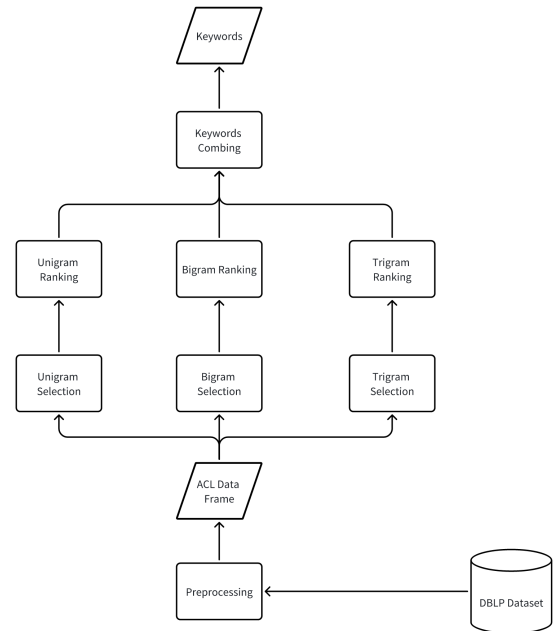


Figure 7: Keywords selection

4.4 Keywords Selection Process

The Keywords Combining is the most tricky part in this section. We find it's really hard to determine the top 3 keywords from keywords list generated from 3 ngram ranking process. We assume the longer phrases convey more important information. However, the assumption faces two problems:

- (1) When the phrases getting longer, the bigram and trigram term frequency will become lower, and the ranking system will lower the score of the phrases.
- (2) The frequency distribution of bigram and trigram in early conference is really sparse, we can't just emphasis phrases with higher gram number.

Considering all the constraints, we add parameters to balance the score after ranking system, we enhance the trigram and bigram score after conference in 2000, and enhance only bigram score before 2000. We also utilize **KeyBERT** to validate our idea. The process of keywords selection is shown in Fig. 7.

Year	Keywords
2023	pretrain language model, large language model, neural machine translation
2022	pretrain language model, neural machine translation, question answering
2021	neural machine translation, pretrain language model, named entity recognition
2020	neural machine translation, question answering, named entity recognition
2019	neural machine translation, question answering, reading comprehension
2018	neural machine translation, question answering, semantic parsing
2017	neural machine translation, dependency parsing, knowledge base
2016	neural machine translation, question answering, language model
2015	convolutional neural network, dependency parsing, statistical machine translation
2014	statistical machine translation, dependency parsing, relation extraction
2013	statistical machine translation, question answering, social media
2012	statistical machine translation, relation extraction, dependency parsing
2011	statistical machine translation, dependency parsing, relation extraction
2010	statistical machine translation, semantic role labeling, dependency parsing
2009	statistical machine translation, dependency parsing, semantic role labeling
2008	statistical machine translation, language model, coreference resolution
2007	statistical machine translation, word sense disambiguation, domain adaptation
2006	statistical machine translation, dependency parsing, conditional random field
2005	statistical machine translation, word sense disambiguation, information extraction
2004	information extraction, machine translation, question answering
2003	machine translation, information extraction, spoken dialogue
2002	named entity recognition, active learning, support vector machine
2001	machine translation, question answering, machine learning
2000	statistical parsing, part-of-speech tagging, morphological analysis
1999	cross-language information, distributional similarity, speech recognition
1998	information retrieval, machine translation, spoken dialogue
1997	machine translation, model statistical, word sense disambiguation
1996	information retrieval, machine translation, optimality theory
1995	sense disambiguation, discourse information, phonological rule
1994	approach automatic, lexical selection, syntax semantic
1993	bilingual corpora, anaphora resolution, grammar
1992	semantic, linear context-free rewriting, discourse
1991	speech recognition, machine translation system, grammar
1990	tree adjoining grammar, combinatory categorial, parsing
1989	discourse, parsing, lexical semantic
1988	parsing, feature structure, categorial
1987	categorial grammar, structure, grammar
1986	machine translation, connectionist, computational complexity
1985	on-line dictionary, parsing, noun phrase
1984	translation, system, parsing
1983	grammar, recognition, parsing
1982	database, representation, understanding
1981	evaluation, semantic, parsing
1980	interactive discourse, communication, parsing
1979	natural language understanding, system, generation

Figure 8: Top 3 Keywords/Key Phrases over the years

4.5 Keywords List

Finally we capture the keywords from the first ACL Conference from 1979 to 2023. We show them in the Table 8.

From Figure 8, we can see the research trend of ACL from its very first version to the last version. From this Figure, we can easily discover the very hot topic in 2023 like Large Language Model (LLM). We can see that Machine Translation is a very dominant topic in ACL conference. The Translation problem dominated the top list for more than 10 years.

4.6 Research Trends Analysis

At the same time, we can mining the relation between hot topics. When a very hot topic occurs, the next few years research will be affected by this hot topic. For example in 2015, the Neural Network was applying to the Machine Translation problem, then the Neural Machine Translation emerging in the next few years compared with the Statistical Machine Translation in the past years.

We show the research trends in the Frequency-Only WordCloud Figure. We adapt 2023 and 2015 WordCloud to illustrate the how research shifting in recent years.

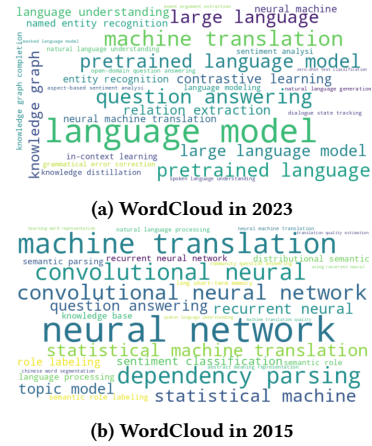


Figure 9: Visualization of Research by WordCloud

5 APPLICATION

Here we define the application to be able to search similar research papers by title. We have built a command line tool to handle all the interactions with the user. The task can be done by a few search strategies discussed below.

5.1 Traditional IR Methods

Lucene provides a built-in `MoreLikeThis` class to find documents that are "like" a given document. It offers options to filter by the frequency and length of the terms and documents.

```
dir = DirectoryReader.open(store_dir)
searcher = IndexSearcher(dir)
mlt = MoreLikeThis(searcher.getIndexReader())
mlt.setAnalyzer(analyzer)
mlt.setFieldNames(['title'])
mlt.setMinTermFreq(1)
mlt.setMinDocFreq(1)
mlt.setMinWordLen(1)
query = mlt.like(doc_id)
top_docs = searcher.search(query, n)
```

A simple example shown above is equivalent to the BM25 ranking. We can verify this by checking the actual query produced by the parser.

```
> mlt 2 100
[QUERY] title:based title:learning title:deep
       title:framework title:prediction title:medical
       title:disease title:cancer title:breast
       title:industries - in 0.1411 seconds

Result 1 (100) <21.42>
[TITLE] Deep learning-based breast cancer disease
       prediction framework for medical industries.

Result 2 (5797220) <12.72>
[TITLE] A deep-learning based diagnostic framework
       for Breast Cancer.
```

Another feature of Lucene is the FuzzyQuery. It searches documents that match a given pattern approximately, based on the edit distance. The query syntax is `term~distance`. The default edit distance is 2.

```
> 3 color~ grey~
[QUERY] title:color~2 title:grey~2 - in 0.1396 seconds

Result 1 (3242120) <6.49>
[TITLE] Color to gray and back: color embedding into
       textured gray images.

Result 3 (4235962) <6.47>
[TITLE] Fast Colour2Grey.
```

5.2 Using Neural Networks

Traditional IR methods like BM25 and TF-IDF process each word as a single token separately. They do not consider the semantic meaning of the words. Matching in term level does not necessarily represent the actual relevance of the documents [11]. Thus, we want to try some neural network based methods.

Doc2vec [5] is a neural network model that learns vector representations of documents, which can be used to find similar documents. We use the gensim library [8] to encode the titles into vectors. Then we can use the cosine similarity to find the relevant titles.

To add a vectorized field to the index in Lucene, we need to define a new field type `KnnFloatVectorField`.

```
doc.add(KnnFloatVectorField(
    "vector", model.infer(title),
    VectorSimilarityFunction.COSINE
))
```

Then we can use the `KnnFloatVectorQuery` to search the documents.

```
dir = DirectoryReader.open(store_dir)
searcher = IndexSearcher(dir)
vector = self.model.infer(query.split(' '))
query = KnnFloatVectorQuery(
    "vector", model.infer(title), n
)
top_docs = searcher.search(query, n)
```

An example of the query is shown below.

```
> knn 2 knn
[QUERY] KnnFloatVectorQuery:vector
       [0.14562196,...][2] - in 0.1521 seconds

Result 1 (5410201) <0.89>
[TITLE] Fast distributed k-nn graph update.

Result 2 (4657100) <0.89>
[TITLE] Wasserstein k-means with sparse simplex
       projection.
```

There are also some downsides to dense vectors. First, it is very slow. It takes tens of hours to train a language model, and the indexing and searching get much longer as well. The time cost of indexing is shown in Fig. 10, which consumes 2 hours compared to the previous 9 minutes. And the search is roughly 10 times slower than the traditional methods, based on my test samples. Second, the results are not always good. As covered in the lecture [4], deep learning methods are not always better than traditional methods, and BM25 is still a good baseline.

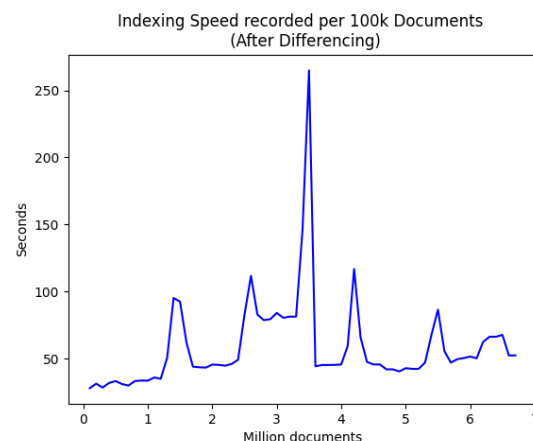


Figure 10: Indexing Time using PyLucene (Vector Field)

To improve the performance, there are some possible solutions. We can boost the embedding quality from large language models. Some techniques to use pre-trained or fine-tuned LLMs have been proposed by Lin et al. and Yoon et al.. It can also be helpful to take advantage of both the traditional methods and the neural network methods. As proposed by Zhu et al., combining TF-IDF and word2vec reached better performance than only TF-IDF or word2vec.

REFERENCES

- [1] 2023. Positional indexes. <https://nlp.stanford.edu/IR-book/html/htmledition/positional-indexes-1.html>
- [2] Sun Aixin. 2023. Assignment PDF, AI6122 Text Data Management Processing.
- [3] Sun Aixin. 2023. Lecture 7 – Boolean and Tolerant Retrieval, Lecture slides, AI6122 Text Data Management Processing.
- [4] Sun Aixin. 2023. Lecture 9.1 – Probabilistic Ranking, Lecture slides, AI6122 Text Data Management Processing.

- [5] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. arXiv:1405.4053 [cs.CL]
- [6] Jimmy Lin, Ronak Pradeep, Tommaso Teofili, and Jasper Xian. 2023. Vector Search with OpenAI Embeddings: Lucene Is All You Need. arXiv:2308.14963 [cs.IR]
- [7] Rada Mihalcea and Paul Tarau. 2004. Texttrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*. 404–411.
- [8] Radim Rehůrek. 2023. *GENSIM 4.3.0 documentation*. Retrieved Oct 30, 2023 from https://radimrehurek.com/gensim/auto_examples/index.html
- [9] Stephen Robertson and Steve Walker. 1994. Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. *Proceedings of the 17th ACM Conference on Research and Development in Information Retrieval (SIGIR)*, Dublin, Ireland, 232–241. https://doi.org/10.1007/978-1-4471-2099-5_24
- [10] The Apache Software Foundation. 2023. *Lucene 9.7.0 queries API*. Retrieved Oct 30, 2023 from https://lucene.apache.org/core/9_7_0/queries/allclasses.html
- [11] Jun Xu, Xiangnan He, and Hang Li. 2018. Deep Learning for Matching in Search and Recommendation. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval* (Ann Arbor, MI, USA) (*SIGIR '18*). Association for Computing Machinery, New York, NY, USA, 1365–1368. <https://doi.org/10.1145/3209978.3210181>
- [12] Jinsung Yoon, Sercan O Arik, Yanfei Chen, and Tomas Pfister. 2023. Search-Adaptor: Text Embedding Customization for Information Retrieval. arXiv:2310.08750 [cs.LG]
- [13] Wei Zhu, Wei Zhang, Guo-Zheng Li, Chong He, and Lei Zhang. 2016. A study of damp-heat syndrome classification using Word2vec and TF-IDF. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 1415–1420. <https://doi.org/10.1109/BIBM.2016.7822730>