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
Aradhya Dhruv	Dhruv 31.10.2023
Bendale Aneesh Santosh	Aneesh 31.10.2023
Chithra Ramesh Asswin	C R Asswin 31.10.2023
Maheswaran RohinKumar	Rohin 31.10.2023
Sivanandan Vijayakumary Abhilash	Abhilash 31.10.2023

Important note:

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

Text Data Management - Assignment I

Sivanandan Vijayakumary Abhilash
MSAI (G2203094G), NTU
Worked on Data Analysis and Research Trend Explorer
abhilash005@e.ntu.edu.sg

Bendale Aneesh Santosh
MSAI (G2303517J), NTU
Worked on Search Engine and Application
aneeshsa001@e.ntu.edu.sg

Aradhya Dhruv
MSAI (G2303518F), NTU
Worked on Data Analysis and Research Trend Explorer
ar0001uv@e.ntu.edu.sg

Maheswaran RohinKumar
MSAI (G2303513K), NTU
Worked on Data Analysis and Application
rohinkum001@e.ntu.edu.sg

Chithra Ramesh Asswin
MSAI (G2302832A), NTU
Worked on Search Engine and Plots Analysis
asswin001@e.ntu.edu.sg

ABSTRACT

In this assignment, we delved into an array of linguistic methodologies, encompassing tokenization, stemming, sentence segmentation, and POS tagging. Our primary objective was the design and development of a Simple Search Engine tailored for the DBLP dataset. This involved steps such as parsing, indexing, keyword-based querying, and a comprehensive result analysis. Subsequently, we introduced a 'Research Trend Explorer', a tool designed to visually represent the progression of pivotal research topics within specific conferences. This tool also facilitates a comparative study of the evolution of these topics across various conferences over the years.

Keywords Stemming, Lemmatization, POS Tagging, Stopwords, Search Engine, Text Management, Text Processing, Dataset Analysis, Search Engine, NLP (Natural Language Processing)

ACM Reference Format:
Sivanandan Vijayakumary Abhilash, **Bendale Aneesh Santosh**, **Aradhya Dhruv**, **Maheswaran RohinKumar**, and **Chithra Ramesh Asswin**. 2023. Text Data Management - Assignment I. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The realm of text management is vast and intricate, playing a pivotal role in how information is processed, organized, and retrieved. As datasets grow exponentially in the digital age, particularly in specialized domains, it becomes imperative to develop advanced methodologies to harness their potential effectively. In this assignment, we have explored the challenges of dealing with different types of text datasets. We started by collecting data related to specific domains, such as Computer Vision and Travel Blogs and Mobile Application Development(Flutter). Then, we looked at how to break this text into smaller pieces, analyze their meaning, and organize them in a way that a computer can understand. This helped us build a powerful Search Engine to find information in a massive database of research papers. We also explored how research topics have changed over time in conferences. As a final touch, we created a simple application that can find similar research papers based on their titles. Our journey was a mix of theory and hands-on work, and this report tells the story of our exploration in the world of text management.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DATASETS CONSIDERED

In this work, three distinct domains have been selected for domain-specific data analysis. Various vocabulary sets are selected from these domains to get the different terminology used in these domains for documentation. Following is a brief description of each dataset:

- **Computer Vision:** Various published research articles comprise the data set for this domain. We downloaded 10 articles in PDF-format. We sourced our data from top-tier conferences in the field of computer vision, ensuring a diverse and high-quality selection of information.
- **Flutter:** This PDF is generated from data that is parsed from GeeksForGeeks website. This contains a series of web pages documenting how to develop a mobile application using the Flutter tool. For this work, ten web pages are considered.
- **Travel Blog:** This data set is developed by parsing the blog written in a series of web pages, which can be found at link This domain class contains eleven PDFs in our dataset.

3 DOMAIN SPECIFIC DATASET ANALYSIS

3.1 Tokenization and Stemming:

- **Tokenization:** It is the process of breaking up text into text units such as words, sentences, or phrases, symbols, or some other meaningful element. In tokenization, we break the sentences into words and store them as a list of words rather than a continuous sentence. It is the first step in any NLP pipeline, and it has an important effect on the text analysis.

Domain	Total Tokens with Stopwords	Tokens after Stopword removal	Top 1000 Tokens
flutter	14994	11010	[flutter, widget, class, notes, ...
computer vision	109915	77800	[graph, scene, object, 1, visual, 2, 3, image, ...
travelblogs	20843	12666	[road, trip, india, visit, places, best, time, ...

Table 1: The number of tokens extracted for each of the considered document domains

Domain	Total Tokens After Stemming	Top 1000 Tokens
flutter	1432	[cover, indexofitem, vote, basi, 201, opaqu, ...
computer vision	7860	[bracehtipupright, soceiti, 88, drop, subnet, ...
travelblogs	2573	[endless, halt, hydrat, flick, cover, rock, ...

Table 2: The number of tokens extracted for each of the considered document domains after stemming

- **Stemming:** It is the process of reducing a word to its root stem by removing the affixes of the word based on predefined rules. Stemming generates the base word from the inflected word, but it might not always result in semantically meaningful base words.

3.2 Implementation:

In order to implement tokenization and stemming, we started with removing stopwords and special characters from our corpus. The tokenizer was able to identify meaningful tokens from each corpus as shown in (Table: 1). **After tokenization, the document classes, i.e., computer vision, flutter, and travel blogs, contain 77800, 11010 and 12688 tokens, respectively.**

However, it can be further tweaked to make it more domain specific. For stemming we implemented Porterstemmer and snowballstemmer and both of them achieved similar results. **The stemmed document contains only 7860, 1432 and 2573 tokens for computer vision, flutter, and travel blog domains, respectively.** (Table2)

The count of the tokens before and after stemming shows similar distribution, with the most frequent word length being around five characters. However, the number of tokens for each length has significantly reduced in stemmed data. (Figure: 3) and (Figure: 2)

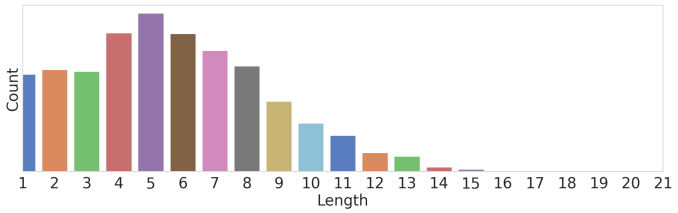

Figure 1: Count of tokens without stemming

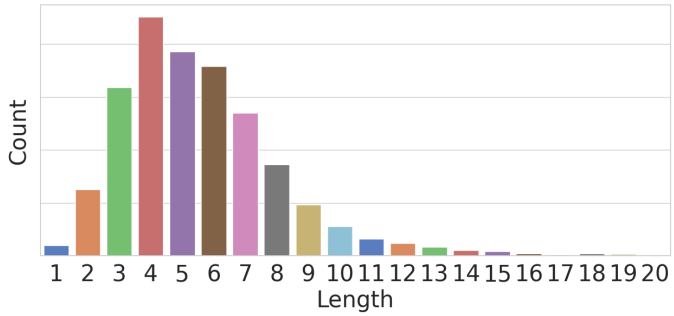

Figure 2: Count of tokens after stemming

3.3 Sentence Segmentation:

Sentence segmentation is the process of breaking down a sentence into its individual words. Figure 3 depicts the sentence length distribution for three domains. The sentence length in the computer vision domain is balanced at around 20 words per sentence. The sentence length in the flutter domain is around 25 words per sentence. The length of the sentences in the travel blog domains soon shows some flat trend, i.e., it contains most sentences in the 15-35 word range. Figure 3

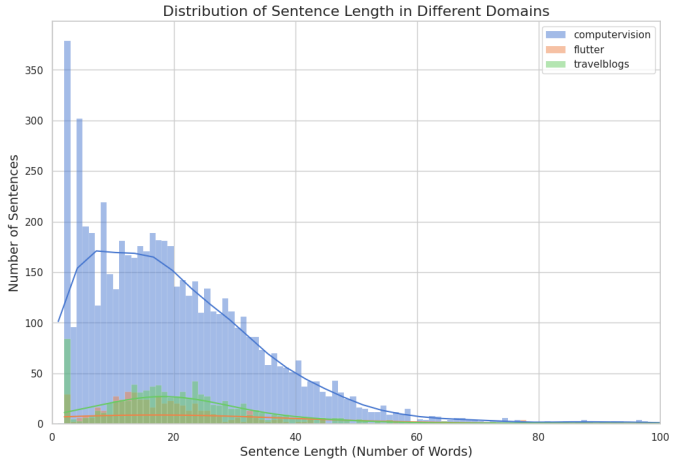

Figure 3: The distribution of the sentence length in the three domains

3.4 POS Tagging:

Part-of-speech (POS) tagging is a process in natural language processing (NLP) that involves labelling words in a sentence with their corresponding POS tags. The POS method of the NLTK library, was utilized for POS tagging and the results obtained were as expected for all the domains.

3.5 Discussion on Tokenization, Stemming and POS Tagging:

Following is a detailed analysis of common pitfalls in our selected domains and how we could potentially improve our analysis to overcome the shortcomings in Tokenization, Stemming and POS Tagging

- **Computer Vision Domain:**
  - **Technical Terms:** Computer Vision documents contain complex terms like "convolutional neural network," which standard tokenization may split into smaller words, impacting readability.
  - **Improvement:** Custom tokenization rules to handle complex terms effectively.
  - **Code and Symbols:** Computer Vision often involves code and math symbols that standard tokenization can mishandle, affecting code analysis.
  - **Improvement:** Pre-processing to protect code and symbols during tokenization.
  - **Multilingual Content:** Some authors mix languages in Computer Vision documents, challenging standard tokenization.
  - **Improvement:** Use language detection to adjust tokenization rules for different languages.
- **Flutter Domain:**
  - **UI Jargon:** Flutter uses UI-specific terms like "StatefulWidget," which standard tokenization may incorrectly separate.
  - **Improvement:** Customize tokenization rules to maintain UI terms' integrity.
  - **Code and Widgets:** Flutter projects include code and widget names like "FlatButton," which standard tokenization might split.
  - **Improvement:** Apply pre-processing to preserve widget names during tokenization.
  - **Context Awareness:** Understanding Flutter widget properties, like "crossAxisAlignment," can be challenging with standard tokenization.
  - **Improvement:** Explore advanced models like BERT for better contextual understanding.
- **Travel Blogs Domain:**
  - **Multilingual Content:** Travel blogs are often written in various languages, and standard tokenization may struggle to handle language shifts.
  - **Improvement:** Use language detection to apply suitable tokenization rules for different language sections.
  - **Special Characters:** Travel blogs contain special characters and symbols that standard tokenization may separate into multiple tokens.
  - **Improvement:** Employ pre-processing techniques to protect and preserve special characters for readability.
  - **Acronyms and Locations:** Travel blogs feature acronyms and location names that standard tokenization may not recognize as complete terms.
  - **Improvement:** Develop specific rules for handling acronyms and location names to enhance user experience.

4 DEVELOPMENT OF A SIMPLE SEARCH ENGINE

The task involves developing a search engine to index and search publications from the DBLP dataset. DBLP provides open bibliographic information on major computer science journals, proceedings, and other forms of publications. For our case, we will be using articles, inproceedings, and proceedings which brings the total dataset to about 6.7 million "documents".

4.1 Parsing the DBLP Dataset

The first step in the process is parsing the DBLP xml dataset. This is done using the *etree.iterparse* function from the *lxml* module, which allows for memory efficient chunk-by-chunk iterative parsing. (Please refer to the code snippet - Figure: 9) The parsed data is then stored in JSON format, with each paper represented as a dictionary containing its title, author(s), publication venue (conference or journal name), and year of publication.

4.2 Indexing

The indexing process was performed using the *Whoosh* library[3]. We used the stopword list from the *NLTK* library. A schema was defined for the index, and documents were added to the index using a writer obtained from the index object. The fields indexed include the title of the paper, author(s) of the paper, publication venue (conference or journal name), and publication year.
Whoosh uses an inverted index method where it creates a mapping of content (such as words or numbers) to its location in a database file or set of files.

The **TEXT** field indexes the text and stores term positions to allow phrase searching. Hence, this field was used to index and store the **Title**, **Author**, and **Venue**. The Title field was also preprocessed which is discussed later in the *Linguistic Processing* section.

The **ID** field type simply indexes the entire value of the field as a single unit (that is, it doesn't break it up into individual terms). This type of field does not store frequency information, so it's quite compact, but not very useful for scoring. Hence, this was used to index the **Year** field.

4.2.1 ***Linguistic Processing.*** The choice of linguistic processing on the words/terms in the chosen fields was as follows:

- ***Stopword Removal***: stopword removal was applied only to the **Title** to enhance search efficiency by reducing data dimensionality. While this can alter phrase meanings, such as changing "to be or not to be" to "be not be", it was deemed beneficial for titles in our setting. **Author**, **Venue**, and **Year** fields were left intact due to their unique information.
- ***No Tokenization***: The text was tokenized into individual words using the default tokenization method provided by

Whoosh. This method splits the text into words at whitespace and punctuation boundaries.
- ***No Stemming***: No stemming was performed on the tokens. This decision was made to preserve the original form of all words, as stemming can sometimes lead to loss of meaning.

Please refer to the code snippet Figure: 10 for the implementation

4.2.2 ***Indexing Analysis.*** The time needed to index every 10% of the documents was collected to analyze the performance of the indexing process. A plot showing the time taken to index each 10% of the data is shown in Figure: 4.

- From the figure it appears that the indexing time increases linearly with the percentage of documents indexed.
- A linear increase in indexing time means that the average time to index each additional document remains constant.
- This suggests that the indexing process scales well with the dataset size, but also indicates that indexing large datasets may be time-consuming.

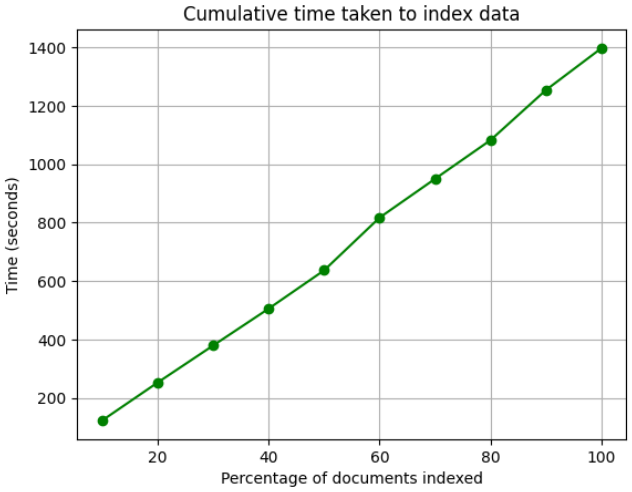

Figure 4: Time Taken to index every 10% of document

4.3 Information Retrieval

For information retrieval, Whoosh uses Okapi BM25 as the default scoring algorithm for ranking the results of a query. Okapi BM25 is a probabilistic model that assigns a score to each document based on the frequency and rarity of the query terms in the document and the collection. It is a variation of TF-IDF, and it has two parameters that control how term frequency and document frequency are normalized.

4.4 Search

- The *open_dir* function from *whoosh.index* is used to open the directory where the index files are stored. This returns an index object.
- The ***Searcher*** object is the main high-level interface for reading the index. The most important method on the *Searcher* object is ***search()***, which takes a *whoosh.query.Query* object and returns a ***Results*** object.
- The ***Results*** object acts like a list of the matched documents. We can use it to access the stored fields of each hit document, to display to the user. The list of result documents is sorted by *score*. (The default scoring algorithm used is BM25)
- The ***MultifieldParser*** class from *whoosh.qparser* is used to parse the query string into a query object. This class allows for searching across multiple fields, in our case, “title”, "author", "venue" and “year”. The parsed query object represents the query in a structured way that can be executed on the index.
- The query used for ***MultifieldParser*** is "Deep Learning AND venue:AAAI AND year:2020" - here, AND, OR, etc are booleans that are used to pass multiple fields in the query as seen in Figure 13 and Figure 14.

The user can input their search query and specify how many results they want to be returned. The search function will then return the top N matching documents from the index. Figure 12 shows the search engine application in action.

From Figure: 13 and Figure: 14, we can observe the results fetched by the search engine based on different scoring algorithms. (We can see the difference in the order of the results fetched in both the algorithms for the same papers.)

5 DEVELOPMENT OF A RESEARCH TREND EXPLORER

In this section of the assignment, we have computed Term Frequency-Inverse Document Frequency (TF-IDF) scores from our document corpus, with the minimum n-gram parameter set to bigrams. Subsequently, we determined the average score for each unique bigram pair within the corpus. Our selection process involved identifying the top five bigram pairs exhibiting the highest TF-IDF scores. These top-ranking bigram pairs serve as indicative markers of prevailing research trends spanning multiple years throughout our comprehensive document corpus. **Underlying Theoretical Concept: TF-IDF**

- TF-IDF is a statistical measure that is used in the fields of information retrieval and machine learning to quantify the importance of string representations such as words, phrases, and lemmas in a corpus [1].
- TF-IDF works by multiplying two metrics: term frequency (TF) and inverse document frequency (IDF). TF measures how often a term appears in a document, while IDF measures the relative rarity of a term in the corpus. Words that are common in every document, such as "this," "what," "if," etc. rank low even though they may appear frequently, since they often don’t represent important information in the document.
- Mathematically, TF-IDF can be expressed as $\textbf{TF-IDF}(t, d) = \textbf{TF}(t, d) \times \textbf{IDF}(t)$

5.1 Explanation of Implementation

- In accordance with our earlier discussions, we have successfully identified and extracted the most prominent keyphrases across various years for all major conferences. To visually represent these keyphrases, we have generated a word cloud. See Figure:5
- Furthermore, we have created a graph to analyze the evolving research trends associated with the keyphrase "Neural Network" across a multitude of conference papers over 15 years. From 2008 to 2013, mentions were minimal. Post-2013, a surge is evident, with ICML peaking at 117 mentions in 2019 and AAAI at 108 in 2023. Other conferences like SIGIR, WSDM, and EMNLP also saw growth, while some, like CIKM, remained moderate. This highlights the rising importance of neural networks in the research community, especially in the last decade. This trend is depicted using the **Figures 6 or 7**
- To demonstrate how TF-IDF scores are being calculated for the entire corpus, the Figure 8 provides a snapshot top 10 keyphrases identified from AAAI conference papers.

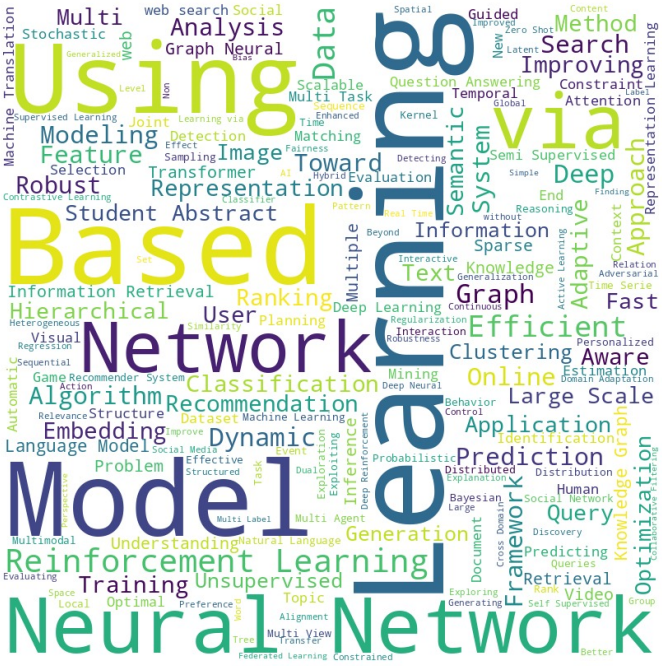

Figure 5: Wordcloud of Key phrases Identified from Corpus (DBLP Dataset)

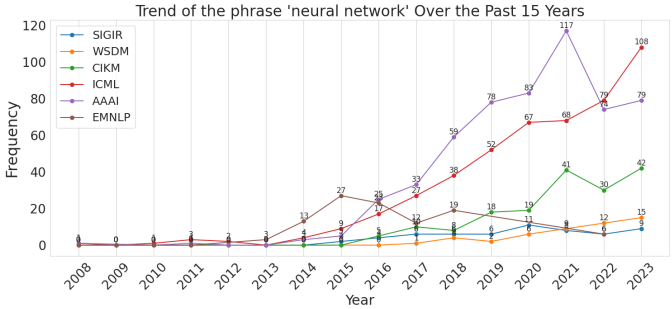

Figure 6: Research Trend for Key Phrase "Neural Network"

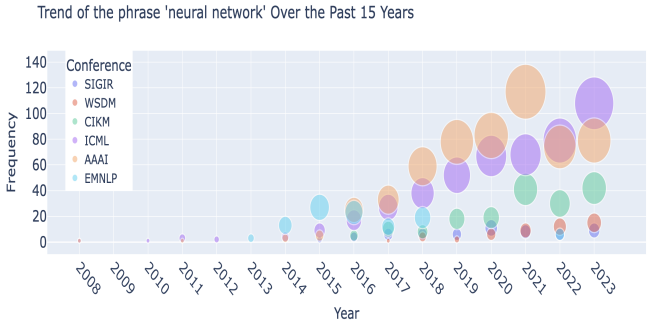

Figure 7: Research Trend for Key Phrase "Neural Network"

score	words
0.001072	deep reinforcement learning
0.001004	graph neural networks
0.000938	deep neural networks
0.000887	neural machine translation
0.000752	convolutional neural networks
0.000672	markov decision processes
0.000603	california usa march
0.000562	aaai spring symposium
0.000552	answer set programming

Figure 8: Top 10 Key phrases identified through Average TF-IDF Score in AAAI Papers

5.2 Discussion of Results

- **Use More Advanced NLP Techniques:** Advanced NLP techniques can include methods like Named Entity Recognition (NER) for identifying entities like names, organizations, and locations, Part-of-Speech tagging to understand the grammatical role of words, and dependency parsing to recognize relationships between words. These techniques can provide richer information for keyphrase extraction by capturing semantic and syntactic patterns.
- **Topic Modeling:** Topic modeling algorithms like Latent Dirichlet Allocation (LDA) can help identify key topics within a corpus. Keyphrases can then be extracted based on the terms associated with these topics.
- **Fine-Tuning Parameters:** Many keyphrase extraction algorithms have parameters that can be adjusted, such as the

length of n-grams, TF-IDF thresholds, or scoring methods. Experimenting with different parameter settings can optimize the extraction process for specific datasets.

- **Using Neural Networks:** Deep neural networks, such as recurrent neural networks (RNNs) and transformer models like BERT, can learn complex language patterns and relationships. They can be used for keyphrase extraction by training on a labeled dataset or fine-tuning on specific tasks.
- **Hybrid Approaches:** Combine multiple methods, such as TF-IDF, TextRank, and rule-based systems, and assign weights to the keyphrases extracted by each method. This can potentially serve as robust way for keyphrase extraction.

6 CONCLUSION

In conclusion, our assignment explored text management, information retrieval, and domain-specific dataset analysis. We used TF-IDF for key phrase extraction, employed the Whoosh library for information retrieval, and conducted in-depth analyses of Flutter, Computer Vision, and Travel Blogs datasets. Our work highlighted the need for custom tokenization rules, symbol protection and hybrid techniques for information retrieval. This practical experiment underscores the importance of adapting techniques specific to a domain so that information retrieval can be more robust.

7 REFERENCES

(1) Simha, A. Machine Learning Understand.
(2) Karabiber, F. *TF-IDF — Term Frequency-Inverse Document Frequency.*
(3) Author, A. *Topic Modeling and Latent Dirichlet Allocation (LDA) using Gensim and Sklearn.*
(4) **Third Party Libraries Used:**
  (a) *Whoosh Documentation*-https://whoosh.readthedocs.io/en/latest/index.html
  (b) *DBLP Dataset*-https://dblp.org/
  (c) *NLTK Documentation*-https://www.nltk.org/
  (d) *lxml Documentation*-https://lxml.de/index.html#documentation
  (e) *PyPDF2 Documentation*-https://pypdf2.readthedocs.io/en/3.0.0/
  (f) *pdfkit Documentation*-https://pypi.org/project/pdfkit/
(5) Course Lectures 7-9

8 APPENDIX

8.1 Code Snippets

```python
def parse_dblp(path, batch_size=60000):
    context = etree.iterparse(path, load_dtd=True, events=('end',), tag=('article', 'inproceedings', 'proceedings'))
    papers = []
    batch_index = 0

    # Creating a directory for the JSON files
    os.makedirs('dataset', exist_ok=True)

    for event, elem in context:
        title = elem.find('title').text
        authors = [author.text for author in elem.findall('author')]
        venue = elem.find('booktitle').text if elem.find('booktitle') is not None else elem.find('journal').text if elem.find('journal') is not None else None
        year = elem.find('year').text if elem.find('year') is not None else None

        papers.append({
            'title': title,
            'authors': authors,
            'venue': venue,
            'year': year
        })

        # calling clear() to free up RAM after fetching the last node
        elem.clear()

        # eliminating now-empty references from the root node to element
        while elem.getprevious() is not None:
            del elem.getparent()[0]

        # If we've reached the batch size, save these papers to a new JSON file
        if len(papers) == batch_size:
            with open(f'dataset/dblp_{batch_index}.json', 'w') as f:
                json.dump(papers, f)
            papers = []
            batch_index += 1

    # Saving any remaining papers to a new JSON file
    if papers:
        with open(f'dataset/dblp_{batch_index}.json', 'w') as f:
            json.dump(papers, f)

parse_dblp('dblp.xml')
```

Figure 9: Parsing DBLP Code Snippet

```python
# Getting the list of stopwords
stoplist = set(stopwords.words('english'))
# Creating a custom analyzer that removes stopwords from the title
analyzer = StandardAnalyzer(stoplist=stoplist)
# Defining the schema using the custom analyzer for the title field only
schema = Schema(title=TEXT(analyzer=analyzer),
                original_title=STORED,
                author=TEXT(stored=True),
                venue=TEXT(stored=True),
                year=ID(stored=True))
# Creating the index
if not os.path.exists("indexdir"):
    os.mkdir("indexdir")
ix = create_in("indexdir", schema)
total_documents = sum([len(json.load(open(os.path.join('dataset', filename)))) for filename in os.listdir('dataset')])
documents_per_10_percent = total_documents // 10
# Initializing a list to store the times and a counter for the documents
times = []
document_counter = 0
# Adding documents to the index
writer = ix.writer()
for filename in os.listdir('dataset'):
    with open(os.path.join('dataset', filename)) as f:
        papers = json.load(f)
        for paper in papers:
            # Recording the start time when starting each 10%
            if document_counter % documents_per_10_percent == 0:
                start_time = time.time()

            writer.add_document(title=paper.get('title', ''),
                                original_title=paper.get('title', ''),
                                author=', '.join(paper.get('authors', [])),
                                venue=paper.get('venue', ''),
                                year=paper.get('year', ''))
            document_counter += 1
            # Recording the end time and calculating the elapsed time after finishing each 10%
            if document_counter % documents_per_10_percent == 0:
                end_time = time.time()
                elapsed_time = end_time - start_time
                times.append(elapsed_time)
                print("Time for this 10% of docs:\t" + str(elapsed_time))
writer.commit()
```

Figure 10: Timed Indexing Code Snippet

```python
# Filter for SIGIR conference
sigir_papers = [paper for paper in data if paper['venue'] == 'SIGIR']
# Combine titles for each year
yearly_texts = {}
for paper in sigir_papers:
    year = paper['year']
    title = paper['title'] if paper['title'] is not None else ''  # Handle None titles
    if year not in yearly_texts:
        yearly_texts[year] = ""
    yearly_texts[year] += title + " "
# Find key phrases using TF-IDF and their frequencies
vectorizer = TfidfVectorizer(ngram_range=(2, 3), max_features=2, stop_words='english')
key_phrases_freq = {}
for year, texts in yearly_texts.items():
    X = vectorizer.fit_transform([texts])
    features = vectorizer.get_feature_names_out()
    frequencies = X.toarray()[0]
    key_phrases_freq[year] = dict(zip(features, frequencies))
```

Figure 11: Research Trend SIGIR

8.2 Search Engine Results

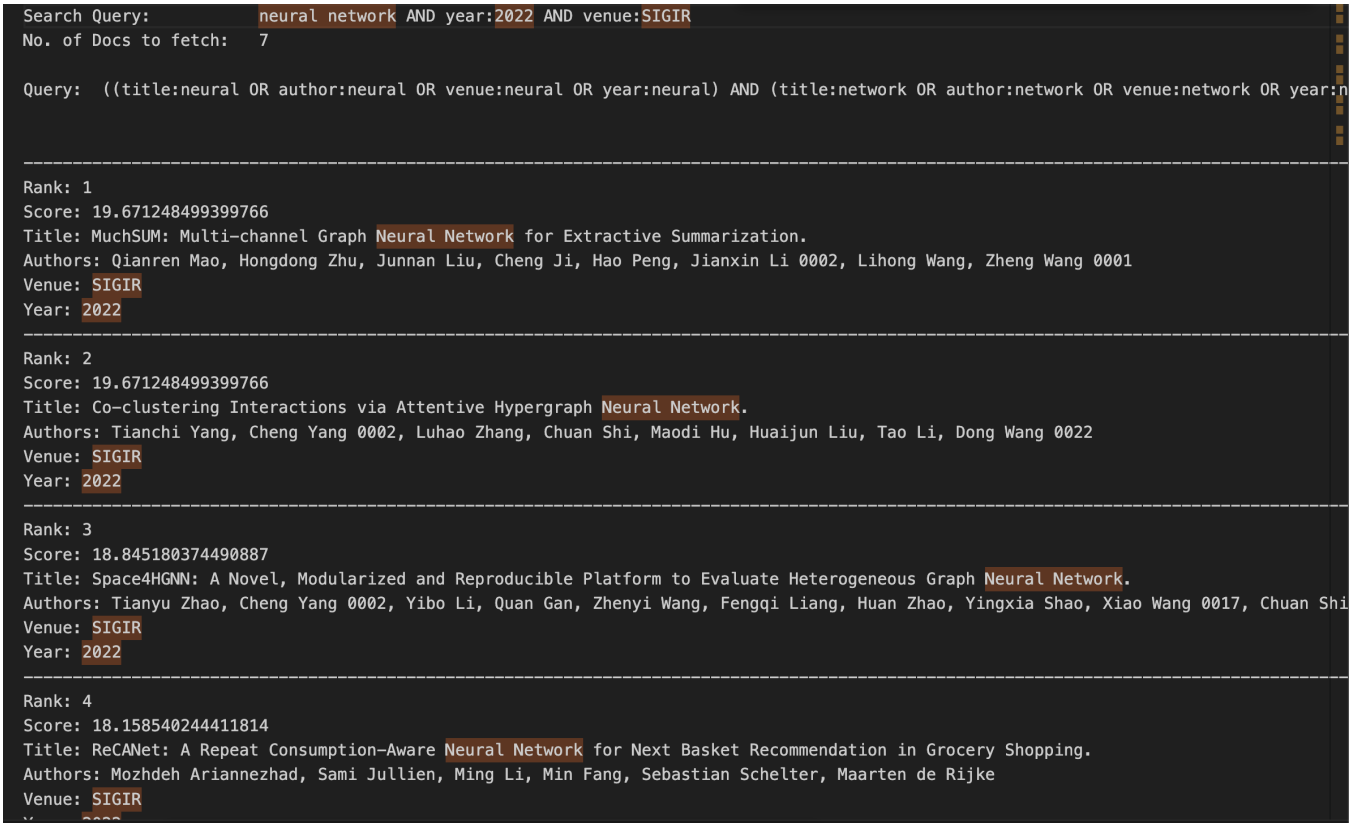

Figure 12: Search Engine Application Results

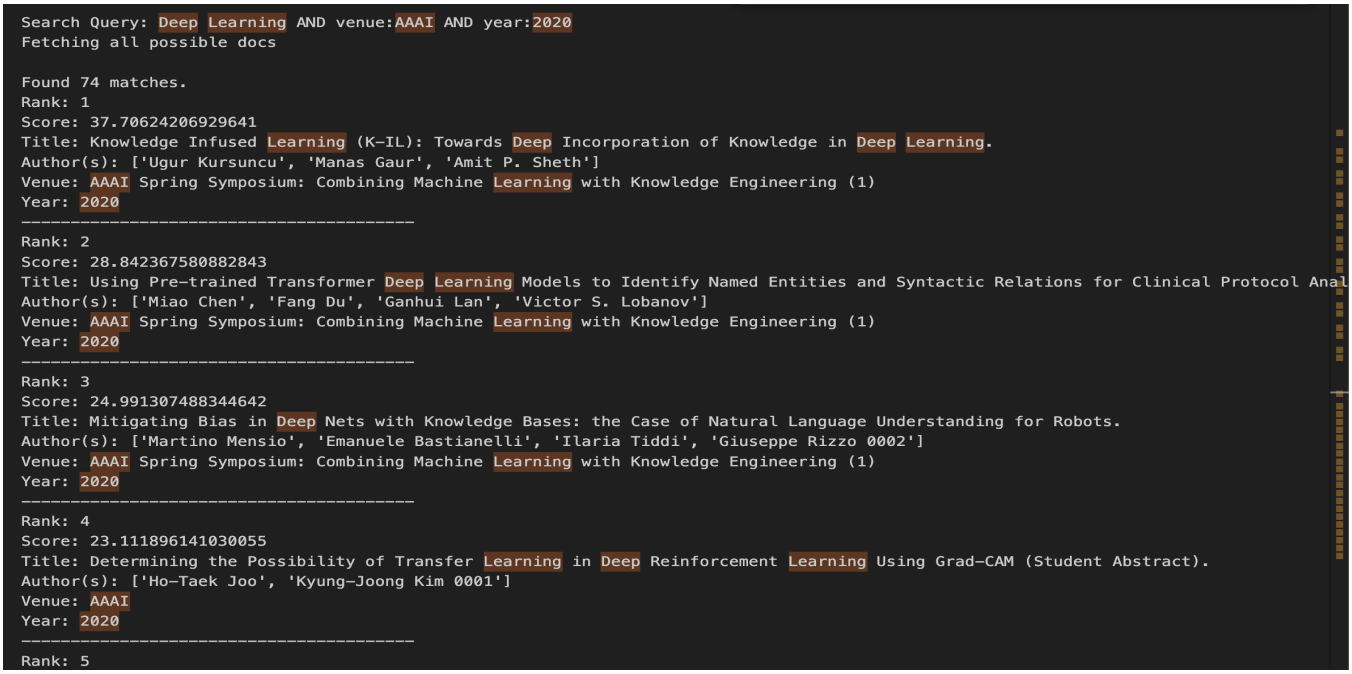

Figure 13: Search Results Using TF-IDF Scoring

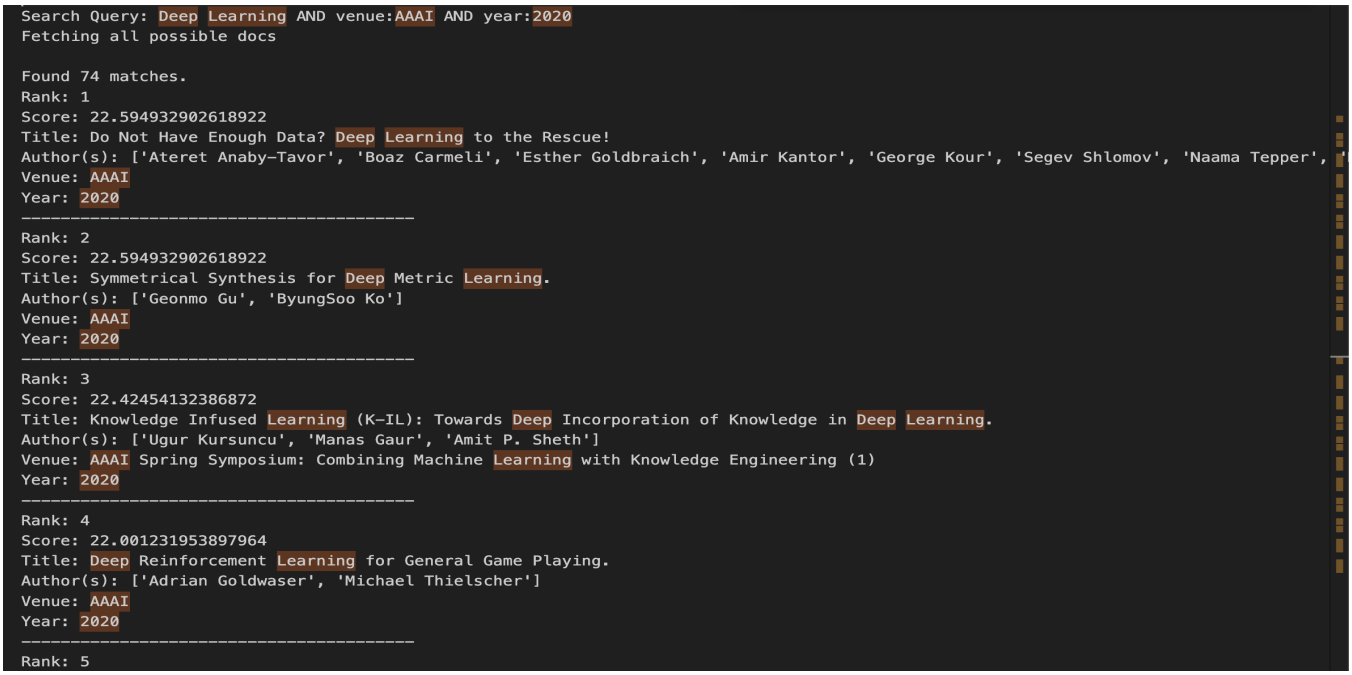

Figure 14: Search Results Using BM25 Scoring