Patrick MacEachen - 40209790
COMP479 - Project 2
2024/12/05

Important Classe

In this section I will describe the 3 main classes used and implemented in this project. They are used for the crawling and indexing of PDFs.

Web Crawling

The project began with the development of a web crawler designed to locate master's and doctoral theses on the Concordia Spectrum website. To achieve this, I created a **PdfCrawler** class responsible for navigating through web pages and extracting PDF content from online theses.

The PdfCrawler class is modular, with each step of the web crawling process encapsulated in its own method with a name starting with ***parse_*** . The crawler's entry point is the ***start_requests*** method, which initiates crawling from the Spectrum homepage. Subsequent parsing methods are used to process various web pages, ultimately leading to the page containing all the theses.

A noteworthy feature of the crawler is its prioritization mechanism. During development, I observed that many of the PDFs retrieved were unreadable, significantly increasing the time required to reach the PDF limit, even for small quantities. The process of locating PDFs involves navigating through a page that lists publication years. I leveraged the assumption that more recently published theses are more likely to be readable. Using this assumption, I assigned higher priority to threads associated with recent publication years (handled by the ***parse_publication_year_page*** method). This optimization dramatically improved the crawler's efficiency.

The final step of the process is handled by the pdf_parser method. This method checks if the limit (that can be provided at initialization) for retrieved PDFs has been reached. If the limit is met, the crawler shuts down; otherwise, the method streams each PDF's content page by page, streaming the data in a dictionary with the structure {url: pdf_url, content: pdf_content}.

This structured approach ensured efficient and accurate crawling, while the prioritization mechanism minimized delays and maximized productivity. There is a slight problem with the limit, that I discuss in the reflection section at the end of this report.

Inverted Index

The second step of the project involved constructing an inverted index from the crawled PDFs. I designed an **Index** class to handle the processing of crawled data, the construction of the inverted index, and its subsequent storage.

The Index class has several key attributes:

index: Stores tokens along with their associated document IDs and frequencies in the documents where they appear.
mapper: Maps document IDs to their corresponding URLs, enabling efficient lookup.
mapper_path and index_path: Paths for saving and loading the index and mapper. When an instance of the Index class is created, it checks if the load parameter of the class initialization is set to True, If so, the class loads the saved content, enabling reuse of already built indexes.

The primary functionality of the Index class is to build the inverted index. This is achieved using the ***add*** method, which takes a piece of text and its corresponding URL as input. Inside the method, the text is tokenized, and the tokens are added to the index attribute along with their metadata (corresponding document Id and the frequency).

This design ensures an efficient and modular approach to constructing and managing the inverted index while providing the flexibility to load and store pre-built indices for subsequent use.

Pipeline

To create a stream of data between the PdfCrawler and the Index, I designed a **Pipeline** class. This class initializes the crawling process and configures its settings, such as ensuring compliance with the robots.txt standard.
The Pipeline class has two key methods:
start: Initiates the execution of the crawler. It integrates with Scrapy's dispatcher, allowing the user to specify a method for processing data as it streams in from the crawler.
_process_item: The method responsible for handling streamed data. It processes each streamed item (containing the URL and a page/content from the PDF) by adding it to an instance of the Index class.
An interesting aspect of this implementation is its memory efficiency. Instead of downloading complete PDFs, the pipeline streams data from the crawler, processing and adding it to the index page by page. This approach minimizes the amount of content kept in memory at any given time, ensuring smooth and efficient processing.

This modular and memory-conscious design enables seamless integration between the crawling and indexing processes while maintaining high performance.

Important functions

In this section, I will discuss the two important functions/scripts used in this project.

In the readme.md file in the project, you can find the appropriate commands to run in the terminal and the possible arguments to run any of these scripts/functions,

Building the inverted index

The file called **build_index.py** is used to create the inverted index by crawling the Concordia Spectrum website and indexing the thesis pdfs found in it. This can be useful for only evaluating the crawling and indexing of documents, without having to wait large periods of time to get a large enough dataset. And use the already compiled index for other steps.

It is a small python script that creates an instance of the pipeline class and passes the required parameters such as the limit of pdfs, the index paths, etc. .

Clustering

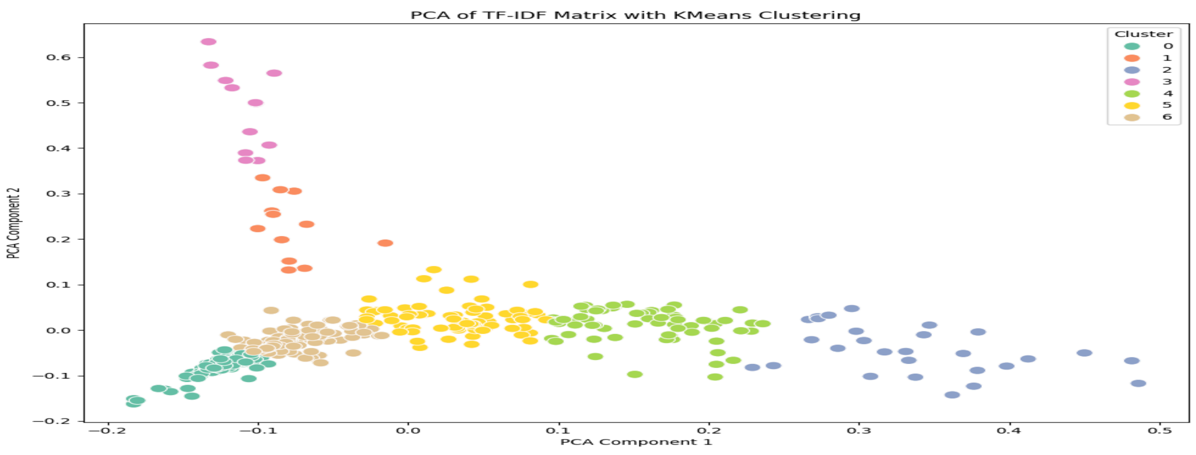
For the clustering process, I developed a simple Python script located in the **clustering.py** file. This script enables clustering on a precompiled index, eliminating the need to wait for a new index compilation before clustering.

Since the clustering involves an inverted index rather than raw documents, the first step is to transform the inverted index into a document–term matrix. In this matrix, each cell represents the frequency of a term t in a document d. To enhance the quality of clustering, I filtered out tokens appearing in 75% or more of the documents, as such tokens are generally uninformative.

The next step involves performing K-Means clustering. I conducted two rounds of clustering: one where the number of clusters matched the number of faculties, and another where the number of clusters corresponded to the number of departments.

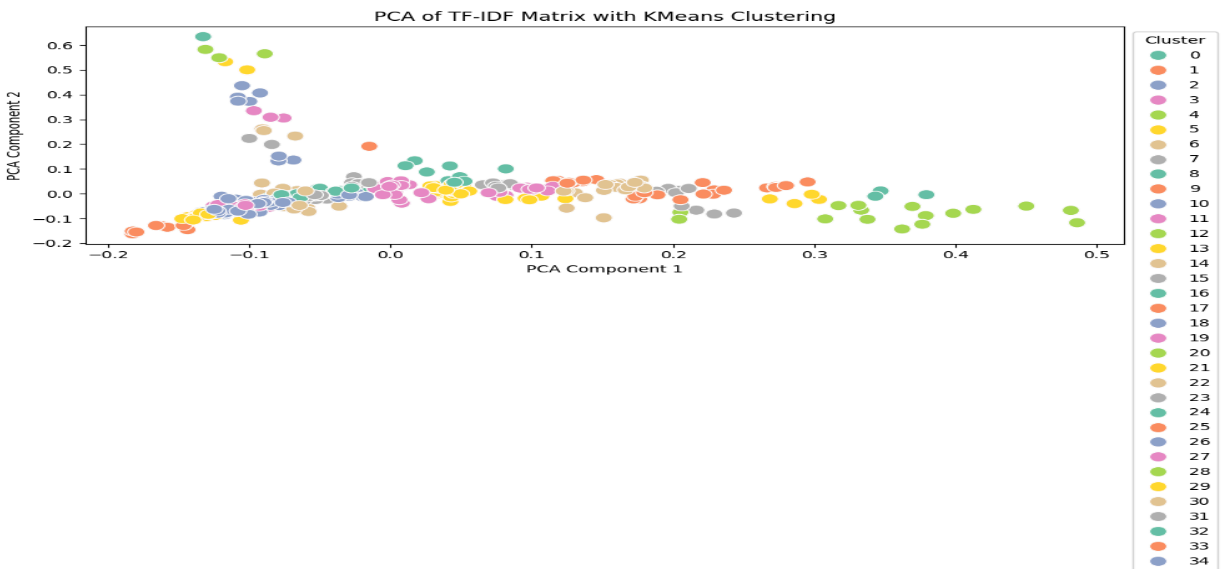
To determine the number of faculties, I analyzed the Faculty/Department pages and counted the hyperlinks starting with "fac=" in their href attributes. This analysis revealed there are 7 faculties.

When reduced to two components for visualization, the resulting plot demonstrates that the clusters are well-defined and show minimal overlap.



For clustering based on the number of departments, I first determined the total number of departments. By analyzing the Faculty/Department pages, I counted the hyperlinks with href attributes starting with "dep=", which amounted to 49 departments.

When visualized in two dimensions, the resulting plot appears less well-defined compared to clustering by faculties. However, it is likely that using more than two components for visualization would result in better-separated clusters



Main

In the **main.py** file, the functions for building and clustering an index are both called sequentially. This is to be able to compile an index from scratch then cluster this new index.

Other directories and files

- Index directory
    - Contains an index.json and mapper.json file. This directory is there so you can build your own index and save it
- Main_index directory
    - Contains the pre-built inverted index and mapper with around 400 documents indexed. This is to avoid having to wait long periods of time to compile a new index and mapper. It is important to not build a new index with this as it`s path. It is best to set load as true to use this.
- Clustering directory

    - This directory contains the saved 3 and 6 cluster Clusterings. In addition, there are the top 20 terms ranked by tf-idf.
- Clustering Faculty and Department directory
    - This directory contains the clusterings for the number of departments and the number of faculties.
    - Includes a 2-d plot of each clustering.

Possible Improvements / Reflexion

Now that I have completed the project, I realize there are several alternative implementations that could have been interesting to explore:

1. **Improving the Crawling-Indexing Pipeline**:
    - Currently, I use the Scrapy library for the crawler but did not create a Scrapy project. Creating a Scrapy project could have simplified my pipeline. Within the built-in pipeline feature of the project, I could have handled document indexing, rather than creating two separate classes for this (Pipeline, Index).
    - Instead of retrieving theses only by year, I could have retrieved them by department (or faculty) and by year. This way, for clustering, I would have at least a few documents for each faculty or department. Additionally, tracking which documents belonged to which faculty or department could have aided the clustering process.
    - I encountered an issue with the upper bound limit for the number of PDFs to index. Once the limit is reached, the crawler is killed, which sometimes interrupts threads that are in the process of retrieving a document. As a result, the total number of indexed PDFs can be slightly lower than the limit. This does not happen every time but does occur occasionally.
2. **Improved Clustering**:
    - Rather than clustering the index after it is created, I could have tried clustering documents as they are streamed into the pipeline.
    - If I had labeled each document with a faculty or department, the clustering process might have yielded more accurate results.

Overall, I aimed to separate different aspects of the project into distinct classes, each with its own responsibility:

- PdfCrawler: For crawling and retrieving the PDFs.
- Index: For building the inverted index.
- Pipeline: For creating the pipeline that streams data from the crawler to the index.

For me, this approach makes the code easier to understand and follow, keeping it clean and organized. While it might seem "overkill" in some areas, it aligns with my preference for clarity and maintainability. For clustering, I opted for a simple function rather than a class, as I felt a class would overcomplicate things.