Kevin Courey

Prof. Sabine Bergler

COMP 479

November 24, 2025

Project 2 Report

The purpose of this project is to implement a web crawler that will search for Master's and PhD theses, as well as other .pdf documents within the spectrum.library.concordia.ca domain. The text from these documents is then extracted, normalized, and stored within an inverted index. A query processor is also implemented and provides users with the ability to enter conjunctive (AND) queries, which will return a list of the downloaded documents that satisfy each query. Lastly, the scikit-learn library is utilized for the purpose of clustering all downloaded documents. All code was written using Python 3.14.0.

I began the project by refining the inverted index and query processor I had implemented for Project 1. The InvertedIndex class now contains a Dictionary object; the Dictionary contains a vocabulary (Python's built-in dictionary data type) which maps terms to a PostingsList object; and a PostingsList is essentially a Python dictionary that maps document IDs (which will be referred to as "docIDs" henceforth) to term positions. In brief, my new InvertedIndex now acts as a positional inverted index. The reasoning behind this design choice was to ensure that the time-complexity for term-lookup and docID-lookup (i.e. the most frequently executed operations) within the inverted index is O(1). Furthermore, I decided to store term positions as opposed to term frequency within the postings lists, since this information can be used to support phrase queries (which I may want at some point in the future) as well as determine the term frequency of a term-docID pair.

The QueryProcessor module has been simplified to run in the following manner: prompt the user to enter their query, retrieve all docIDs that satisfy said query (using the intersect() method), display these docIDs to the user, then add them to a set called “My-collection”, whose contents will be stored in a text file labelled “My-collection.txt” as specified in the project guidelines. A Query is a class that contains three attributes: the string representation of the user’s query, the normalized form of this query, and an inverted index. The tokens from the normalized query are extracted and stored within this inverted index, so that the term frequency for each query term may be determined (which is essential, if I ever want to determine their tf-idf weights).

The Normalization module is also a simplified version of what I had written for Project 1, with the only major differences being that I am now utilizing the Strategy (GoF) design pattern to structure this module and have removed the code that was previously responsible for calculating statistics related to the effects of the compression techniques on the dictionary and postings for a given inverted index. It should also be noted that while tokens were normalized only *after* the inverted index was fully populated in Project 1, I have modified the program so that now tokens are normalized *before* they are added to the dictionary, for the sake of removing the overhead of resizing the dictionary as well as rehashing its contents.

The Strategies module also utilizes the Strategy (GoF) design pattern, and although it serves no useful purpose for the sake of this project (besides containing the logic for my SPIMI-inspired index construction algorithm), I have decided to leave it in the event that I wish to implement BSBI and perhaps more advanced algorithms for constructing an inverted index for large document collections that cannot fit entirely in memory.

The WebCrawler module contains all the logic regarding scraping .pdf files from within the spectrum.library.concordia.ca and storing the contents of these files into an inverted index. The web crawler itself extends the CrawlSpider class from the Scrapy framework, and follows URL links based on a set of rules that I have defined. Once a .pdf file is encountered, its contents are processed by a PdfReader object, which I have imported from the pypdf library. Using this PdfReader, I can extract the text from the .pdf file, tokenize it using the nltk library, and store each token within my inverted index.

Lastly, the Driver module contains the main() method, which does the following: constructs an inverted index, which in turn, activates the WebCrawler and commences the process of locating .pdf files and storing its contents into the aforementioned index. Once this process is complete, the index is sorted (firstly the vocabulary is sorted by term, and then each postings list is sorted by docID, then term positions). Now, a QueryProcessor is constructed, the user is prompted to enter their query, and the terms within this query are compared with the terms within the inverted index to determine which documents "satisfy" the user's query. Once this is complete, the clusterDocs() function is invoked, whose sole purpose is to cluster the .pdf files using K-Means and save the results to: \COMP479_Project2\Results\clusterInfo.txt.

All results collected throughout the user's session are stored within the \COMP479_Project2\Results folder and all .pdf links and their contents are stored within the \COMP479_Project2\Downloads folder.

Now, I would like to go over some of the results that were collected after running my program. I chose to limit the Web Crawler to download fifty .pdf documents. The time it took to locate these documents, extract their text, and feed it one token at a time to the inverted index took approx 402 seconds (this is including the one second wait time per link crawled), meaning

that on average, 1 document was downloaded every 0.82 seconds. The time it took to sort the 48,901 terms and their respective postings lists within this inverted index was 0.1587 seconds. The query “sustainability” resulted in 19 documents being retrieved, and the query “waste” resulted in 11 documents being retrieved. The intersection of these retrievals consisted of 7 documents and thus, their union consisted of (19 + 11 - 7) = 23 documents. The three clusterings (k = 2, k =10, k = 20) were performed on the entire inverted index, with each one providing radically different results. With k = 2, the documents were split perfectly evenly. With k = 10, the clusterings seemed to follow a normal distribution along a Cartesian plane (where cluster number is represented by the x-axis, and the number of documents belonging to a cluster is represented by the y-axis). With k = 20, slightly more than half of the clusters contained 1 document each, while the remaining clusters contained on average roughly 5 documents each. Furthermore, I compared the “abstract” portion of each document within a given cluster to test whether or not similar documents had been clustered together, and to my surprise, they were for the most part. While labelling these clusters is not a straightforward matter, I noticed that the first non-punctuation-mark term from the list of top-most weighted terms from within a cluster of documents tended to provide a decent enough label for classifying that cluster.

To conclude this report, I will delve into my experience crawling and scraping web pages within the spectrum.library.concordia.ca domain. To accomplish this task, I was recommended to utilize the CrawlSpider class from the scrapy framework by one of my TAs. After watching tutorials on how to implement a web crawler with this framework, I began by inspecting the spectrum website and establishing the rules that would allow the crawler to navigate the site and effectively locate important pdf documents. This was no easy task to accomplish properly, and required a significant amount of trial and error before I could get the crawler to locate the

Master's and PhD theses within a few seconds of being activated. The next issue I faced was getting my crawler to actually *download* a user-specified amount of documents within a reasonable amount of time. The reason why this was an issue despite my crawler having been able to locate the theses quickly, had to do with the fact that it would process the theses in the order they were published, and the older the publishing date, the more likely the pdf file contained non-extractable text. However, after realizing that theses published from 2010 onwards contained extractable text, all that needed to be done was to make a simple adjustment to an existing rule, and suddenly my crawler was able to both locate and download documents at a significantly improved rate. The last main issue I faced involved the tokens that were extracted from these documents. After running my program and analyzing the contents of my inverted index, I had noticed that many lines of extracted text spanning multiple pdf documents had been incorrectly tokenized with multiple words being strung together and considered as a single token, leading to the inverted index containing extremely long terms and very short postings lists. To combat this, I first tried using different "pdf to text" libraries, but when this did not work, I decided to modify the rule within my web crawler that targets pdf documents published from a certain date onwards, only this time, starting from 2020. Although this solution has not resolved the issue completely, it has significantly reduced the presence of combined tokens within the inverted index, thereby improving the results aimed to satisfy users' queries.