Group Members: Vi Nguyen, Matt Lucas, James Formato, Hima Barla

**Final Project Report: A Book Retrieval Application**

For our project we built an information retrieval system. The data in which we used for our IR system was from [Project Gutenberg](https://www.gutenberg.org). The language we use was python and we used python libraries such as NLTK and Sci-Kit Learn.Our project starts with scraping data from the Project Gutenberg website. Which stores the Author, Title, Release Date, Language, and Text information into a json file. Next we used a document parser which pre-process the by tokenizing the text, stemming it, and removing undesirable characters. We also calculate the ID and the TF-IDF. This file also takes in the user query and processes it. Then our main program, the book\_term\_manager.py file is the core of the IR project. This program calculates the Cosine Similarity and Euclidean Similarity and compares the query score to that of our data. Ultimately our project consists of different Python programs,that contains comments on what they do, which we will go into further detail in the report. We also hosted the IR system on the webframe Django.

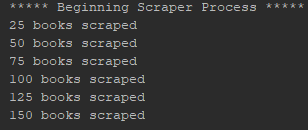
**Scraper\_process.py**

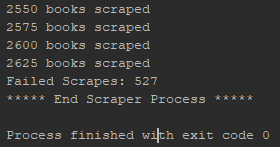
Iterating over the URLs of the eBooks was easy considering the urls had the books IDs in sequential format. In the raw text of each book there was a consistent header and footer; This made scraping the raw text fairly easy.

This file contains 9 functions and the purpose of this program is to gather and scrape and the data from the project. The get\_title function scrapes the book for title which is returned as a string which later will be embedded into the scrape\_book function. The get\_range function gets the pattern range and returns a list of tuples which is later called in the get\_book function. The get\_books function returns the full e-book as a string. The scape\_book function scrapes the book for content and calls the following functions: get\_title, get\_book, get\_author, get\_release\_date, get\_language and the information is returned as a dictionary. The following functions: get\_title, get\_book, get\_author, get\_release\_date, get\_language gathers the information that is listed in the name of the function. The generate\_sample function will be used to generate a smaller sample of the original data, the function allows you to pass a number that will be equivalent to the sample produced. The scraper\_process function will scrape and parse the Project Gutenberg website for book data. The end results should be a books.json file.

Screenshots of test runs:

Fig 1:Scraper





**Document\_parser.py**

This function was to create an initial inverted index and model for the information retrieval system. Initially, the inverted index was created using the entirety of the books for analysis, which resulted in 100,000+ terms. The parser was edited to take only the second bin of 1,000 words within a document. The choice to take the second bin is due to formatting at the beginning of every book (ex. Table of contents, chapters, tributes, etc.). This ensures we are analyzing more of the actual book rather than publishing terms. Another issue arose with strange html formatting. We were seeing terms with “\x80” type characters within them, and there were a lot. A regex matching engine was built to filter out the terms with this formatting after the tokenization.

Lastly, even after the massive reduction in terms there were still very many. We decided to trim the lower half of the terms since we were seeing a lot of terms only appearing a few times. Fig 2 shows the distributions of the terms in our books. We decided on 2.5% trim as the sweet spot, because anything beyond 2.5% was trimming too much of our data. Also, we trimmed the terms which appeared in over 80% of the books. At this point, an initial retrieval system was designed using the vector space retrieval algorithm; However, it proved to be slow and a sample has to be used.

The program contains 9 functions and the purpose of this program is to pre-process the text as well as the query. This program also contains the functions that calculate TF-IDF. The function generate\_histogram generates a histogram of the term frequency distribution.This function was used for exploratory analysis since we wanted to find a way to trim the data. The function get\_stems() is our text pre-processing function which tokenizes the words, stems the words, and removes undesirable characters. The parse\_document function gets the term frequencies and returns a dictionary. The trim\_terms function was another function we used to trim the data, this function trims terms based on how frequently they appear, and it returns a dictionary. Term Frequency – Inverse Document Frequency is one of the most important techniques used for information retrieval to represent how important a specific word or phrase is to a given document.The get\_idf function returns the inverse document frequency. The compute\_tfidf function computes the TF\_IDF and rounds it. The get\_document\_frequencies function parses three documents for term frequencies and returns a dictionary. The process\_query function takes the user query and then does pre-processing such as stemming and removing stop words. The return\_query function takes the user query and dictionary while calling the process\_query function,

Screenshots of test runs:

Fig 2: Histogram of the term frequency distribution

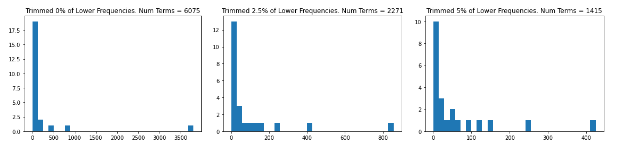


Fig 3:Tf-idf

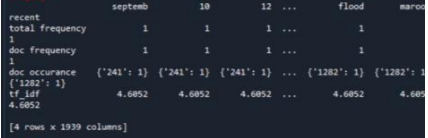


Fig 4: Inverted Index Creation

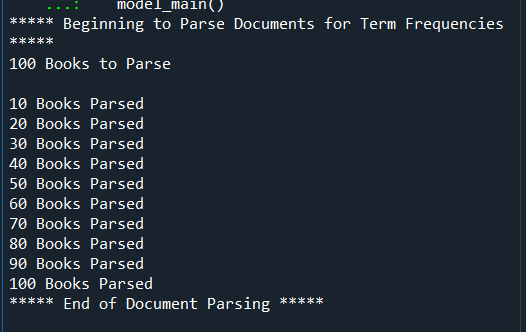
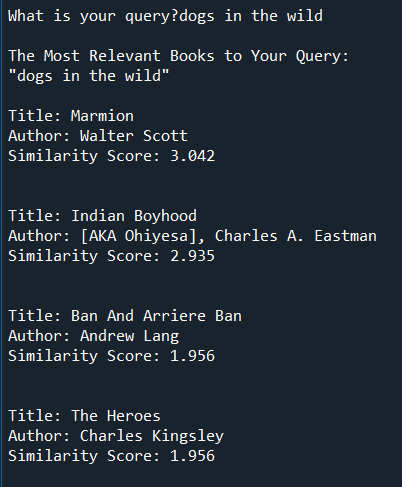


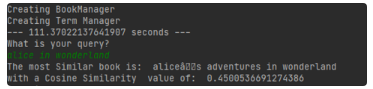
Fig 5: Initial Prototype of Query System: Vector Space Retrieval Model



**Book\_term\_manager.py**

This file is stored in the CSC575\_django folder. This book\_term\_manager.py file contains three classes. The book class has four functions which gather the book and process the text. The BookManager class has 6 functions which mostly involves appending the books. The last function in that file is the query\_similairty function which calculates the similarity metrics, in which we used Cosine Similarity and Euclidean Distance. We use the sci-kit learn libraries that let us call the built in cosine similarity function and euclidean\_distance function in order to calculate these values. The last class in that file is the TermManager class. This class is used to track the terms in all of the books. The process\_books function returns a dictionary of the total word counts called the total\_term\_counts. The process\_books function also returns a dictionary of terms with nested dictionaries of keys which are documents and values which are term count. All these classes come together when you run the book\_term\_manager.py file which when you launch will ask for the user query and say “What is your query?”. Which will take the query and return the most similar book as well as the Cosine Similarity or Euclidean Value as seen in figure 5.This is the basis of our retrieval program. It is then hosted via Django.

Fig 6: Cosine Similarity



**Django**

In our code files, we have a Django folder, which contains some of the previous python files mentioned above which is the logic of our Information Retrieval System as well as code in order to host our application on the web. In the views.py file there are two functions. The home\_page function which returns the webpage. The get\_user\_choice function which renders the query for the user. The first time it will be a get request. The second time it will be a POST request to be filled out by the user. This will bring in the book manager, where there will be pickling happening to speed up processing. Essentially how this file works is that it enables us to host the Information Retrieval System on the web.

Fig.7

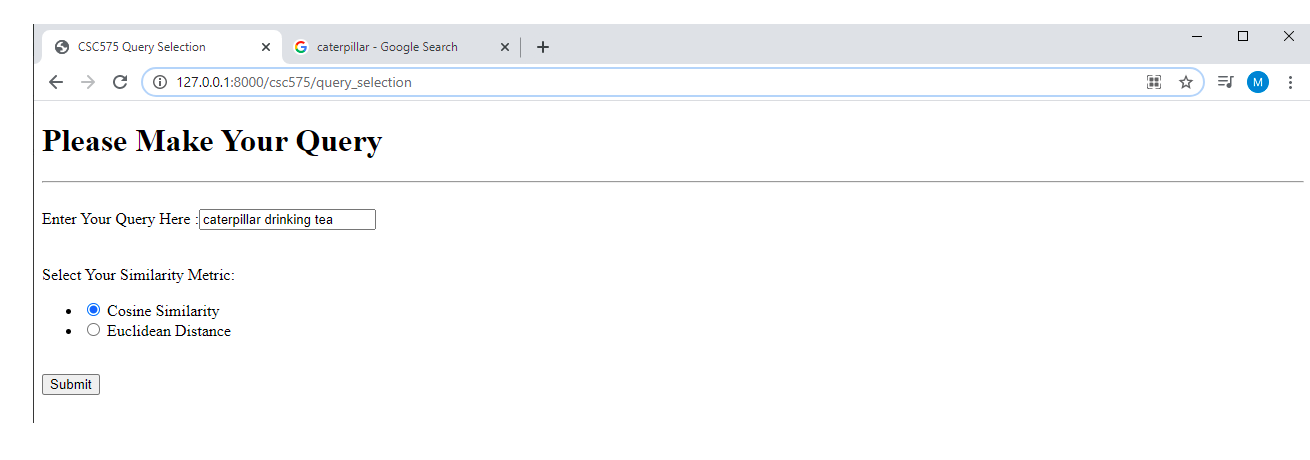
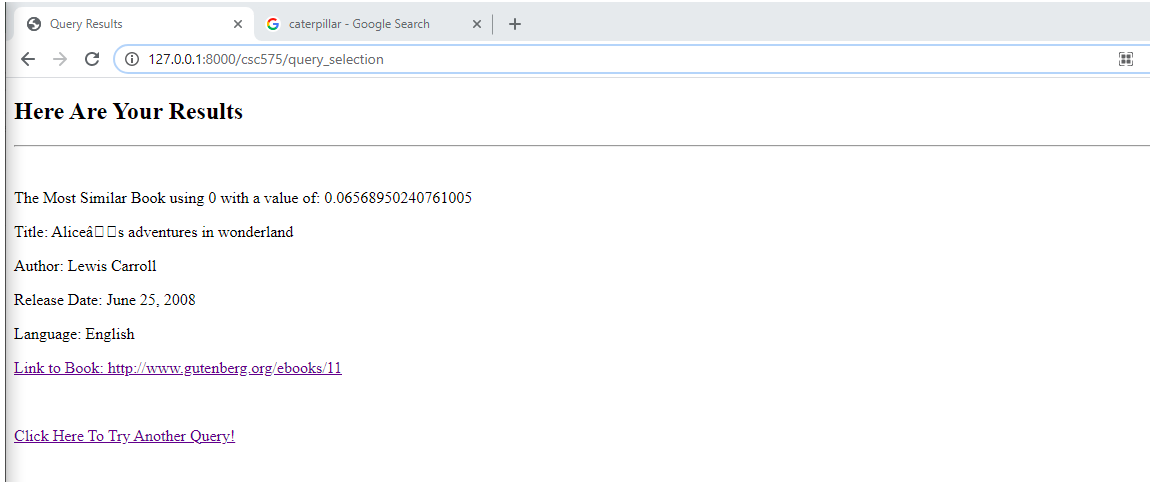
****

Fig 8.



**To use the Information Retrieval program:**

[**https://csc575groupproject.azurewebsites.net/**](https://csc575groupproject.azurewebsites.net/)