Crawler implementation:

Crawler takes three parameters. First is the website that crawling will start, second one is how many webpages will be crawled and the third one is how many threads will be used. Crawler is implemented of three functions. One function is find_links who makes requests in order to take the text of a website. Second is the file handler who handles the documents that have been crawled or the ones that are in waiting. The crawler.py is the last file for crawler. That one is the file that run all the procedure of the crawler. It gives order to gather links, read the text that is in the websites and send them to Index. Also it calls the function csv_hanler the one that makes the csv file that saves the dictionary. The crawling procedure uses a hybrid algorithm who combines bfs and dfs and reads all the pages from the file queue.txt. Pgaes that crawled are saved to the file crawled.txt and erase from the file queue.txt. More information for some files is given.

file_handler.py: It use the os library. This is the file that handles folders and txt. It contains the following functions:

1η) create_proj_directory(directory): Here we create a folder that will contain the files queue.txt and crawled.txt.

2η) create_files(project,first_url):Here we create the two txt files. In queue.txt is written the first link that the crawler takes.

3η) write_to_file(path,data): It creates a new folder and writes in it.

4η) delete_file_inside(path): Deletes the content of a folder.

5η) file_to_set(file_name): It reads a file and it converts every line of it to set items.

6η) set_to_file(links,file_name): It writes every item of a set in a line of a file.

find_links.py:

It uses the html.parser library from HTMLParser and from urllib the parse. There is the class **FindLinks(HTMLParser).** It opens a url and it search for the rest urls under some conditions and it saves them in a set.

.The class contains the following functions:

1η) handle_starttag(self,tag,attrs): This function belongs to HTMLParser and we make override. When the function feed() is called from HTMLParser and meet the tag <a> this function called too. If it called and the tag is <a> we search if the attribute is href. The href declares the destination of the link. If it is href then we make an absolute url and we add it to a set with links.

2η) get_links(self): It returns the links that have been found.

3η) error(self,message): When it finds an error it ignores it.

crawler.py:

Here is the crawler class.We take the urllib.request and more specific the urlopen. Also from bs4 we use the BeautifulSoup. All the functions except the constructor are static. The functions are the following:

1η) __init__(self,project,base_url): It starts with the project name and the homepage. It creates the files queue.txt and crawled.txt. It calls the functions startup() and crawling of the class.

2η) start_up(): When the files of the project the crawler starts.

3η) crawling(self,page_url): It reads a url that is given as an argument to the collect_links and all this to the add_links. The url is deleted from the queue and it is added to the crawl.

4η) collect_links(self,page_url): It uses the **find_links.py** file. With the library urlopen it asks for permission to read the html of a website and sends the url to the save_text who will keep the text. If it cant open the url we print the error that is found. In the end it returns the rest links that found with the help of get_links() from the **FindsLinks().**

5η) save_text(page_url): Its reads the text and make some preprocessing before it passes it as an argument to the index with the url. write_a_csv(Index(text, page_url)) In this line it called the function write_a_csv from the csv_handler with argument one object of Index who has as an argument the processed text and the page that found. We make overwrite the dictionary after we update the inverted index.

6η) add_links(links): All the links that are found in other links and have not yet found or crawled is add it to the queue.

7η) update_files(): It uses the function set_to_file() from the file **file_handler.py**.

Indexer Implementation:

Here the inverted index is created and processed. Reading the text of each webpage we update it. Indexer will use two files.**Index.py** and **csv_handler.py.**

Index.py:

Here we have the Index class.It uses the libraries re,string and from nltk.corpus, nltk.stem, nltk.tokenize the stopwords, WordNetLemmatizer and word_tokenize respectively. We have the variable the_dict in which we store the whole inverted index who updated for every new webpage that comes. All the functions except the constructor are static.
The functions are the following:

1η) __init__(self,text,url): In constructor the function preprocess is called for the specific object and the the_dictionary.

2η) __iter__(self): This allow to make iterate the text.

3η) preprocess(text): Here we preprocess the text of every webpage deleting symbols, stopwords, numbers and punctuation. We keep only latin letters and we convert the letters to lowercase. Also we convert every word to the root(stemming). Finally we sort alphabeticaly and we keep in a variable the whole size of the text. It returns the preprocessed text.

4η) docs_dictionary(text): Here it is created a small dictionary who stores the information of the text for every webpage that comes. Specifically the words and their frequency.

5η) the_dictionary(text,page_url): Here update for every webpage the whole inverted index. The call of docs_dictionary is made here who return the information it found. For every word of the specific url we check if it is already in the dictionary. If it doesn't exist we add to the inverted index a new line with the word word and also the frequency of this word in this url, the url and the total words that url conatins. These three information are a list. If it already exist a word to the inverted index then these three elements are added to a list who contains as many sublists as the word has been found on different sites.

csv_handler.py:

The pandas and os libraries are used. Save the inverted index to a csv file. This is implemented here. The functions are as follows:

1η) write_a_csv(obj): Here we create the csv file if it does not exist. If there is, overwrite the inverted index. So every time we crawl a new page the csv file is updated. For implementation we use the pandas library.

2η) read_a_csv(csv_name): We read a csv.

3η) delete_a_csv(csv_name): We delete a csv.

Call of **Crawler** and **Indexer**:

Crawler_Indexer_run.py: The threading, Queue and shutil libraries are used. Here is the main function that starts the crawler. First we ask the user to give the homepage, the number of pages he wants to be crawled, and the number of threads that will be used. Unfortunately with the way we save the inverted index (csv) we do not manage to keep the inverted index from the previous crawl. So if we crawl again from scratch we delete the inverted index and make it from scratch. Run the crawler manufacturer and then the main. The main is implemented with multithreading. First create_threads is called and then crawl. It has the following 4 functions:

1η) create_threads(): We implement the threads provided by the user who will exist as long as main runs. The multithreading process begins by calling the function nextjob().

2η) next_job(): Implements the new process that is pending. It reads a url from the queue and calls the crawling function of the crawler class by passing this url as an argument.

3η) new_jobs(): The new_jobs with crawl call each other and read the urls one by one of the queue to do the next 'work'.

4η) crawl(): It calles the new_jobs as long as there are pages in the queue.

Query Processor Implementation:

The query processor also connects to our server (localhost) where the user over there gives the query he wants to search and the amounts of top-k documents he wants to be returned. Depending on the user's query we calculate the similarity of the documents with the query using tf-idf and cosine similarity and show the top-k documents to localhost. The server.py, Similarity.py, QueryRun.py and index.html files are used in the query processor.

Index.html: Here we create the interface of our localhost. In this file we make 3 inputs. One for the user query, one for the top-k, and one button to do the search. The css language is used to configure localhost by putting colors, shadows, shapes, and to center the output.

server.py: Here we build a simple http server. The following libraries are used: urllib, logging, SimpleHTTPRequestHandler and HTTPServer from http.server, and BeautifulSoup from bs4. We have a class S that accepts SimpleHTTPRequestHandler. We have the following functions:

1st) _set_response (self): Sets the response 200 and sends a header.

2nd) do_GET (self): The way we read what the user has given is from the url. That is, we take the path and retrieve the information. By doing split and replace we read the piece we want from the path and then isolate the query and the top-k. If the user does not put amounts of documents they want to be returned we return up to 10 results. The QueryRun and Similarity files will be called here. We preprocess through QueryRun the user query and calculate the tf of the query. Then we create an object of the Similarity class in which we give the user query his tf and top-k. This will return the results-urls. For each url we read its title and print it together with the url in localhost. There is also the run function (server_class = HTTPServer, handler_class = S, port = 8080) which is out of class. In this function we give a port to run the http server and connect it to the class S mentioned above. As the saying goes Crawler-Indexer is a different process from QueryProcessor. This is why server.py has a separate main. In this main we use the sys library and call the run function to implement the server.

QueryRun.py: In this file preprocess is done in the user query and the TF is calculated for this query. As in the Index.py file, the following libraries are used for preprocess: re, string and from nltk.corpus, nltk.stem, nltk.tokenize stopwords, WordNetLemmatizer and word_tokenize respectively. In the file there is a global variable, the users_tf_list which is a list with tf for each word of the user. We use the QueryRun class, where we have the following functions (All functions except the constructor are static):

1st) preprocess (text): Here we process the text of each page by deleting symbols, stopwords, numbers and punctuation. We keep only Latin characters and convert the letters to lowercase. We also turn every word into its root. Finally we sort alphabetically and hold on to one. variable the overall text size. To do all this, all the libraries mentioned above are used. Returns edited text. Also the words before we put them in a list, we put them in a set, so that the duplicates are removed.

2nd) users_tf (size_of_search): Here we calculate the tf for each word processed in the user query after they are preprocessed. The tf of each word is equal to 1 / size_of_search as there are no duplicates.

Similarity.py: Here we calculate the similarity of the user query with the documents that contain at least one word from those searched by the user. For these documents we calculate the cosine similarity, with the weights equal to TFxIDF and return the topK websites. The file contains the Similarity class and the pandas, csv and math libraries are used. This class is used in server.py where an object of the class is created for each user query. The class uses 5 variables, which are: - idf: a list that holds idf for each URL that contains at least one word from the query
-tf: a list that holds tf for each URL that contains at least one word from the query
-urls: a list that holds all URLs that contain at least one word from the query
-weight_of_url: a list that holds weights (TFxIDF) for each URL that contains at least one word from the query
-results_urls: The URLs that will be returned to the user at the end.
All functions except the manufacturer are static. Its functions are as follows:
1st) __init __ (self, search, query_tf, topK): There are 3 arguments here. The search is the query words that the user searched for but after they are pre-processed in the QueryRun class. Query_tf is the tf of the query which is also calculated in QueryRun. Finally it reads the topK documents that the user wants to be returned. For each query the lists idf, tf, urls, weight_of_url become clear, so that they can be calculated from the beginning.

2nd) copy_index (): We copy our csv file with the inverted index to a new csv, so that if at the moment of crawling a query is searched, we have the previous inverted index, until the crawling is over.

3rd) tf_idf (list_search): Here the weight is calculated for all pages that contain at least one of the query words. We read the contents of the copied inverted index. In the beginning, we initialize the lists that will be used. Because a word can exist in many URLs, the tf, urls, weight_of_url lists are as many sublists as the words in the query, where each sublist contains the results for one word. It becomes a double for, which for each line in the index, checks if every word in the query is there. For each query word in the index we keep all the information, that is, the frequency of the word, the URL found and the number of distinct words in the URL. This information is reserved for all URLs where the word was found. Then we calculate how many URLs contain the word and read from the file "numOfSites.txt" created in Crawler_Indexer_run.py how many pages we crawled. The idf of the word is then calculated using the follow type:

$$idf(w) = log(\frac{N}{df_t})$$

, where N is the number of all URLS and dft is the number of URLs containing the word. To calculate tf one more for loop is needed to scan all URLs containing the word. The formula we use is as follows:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

, where the numerator is how many times the word appears in the URL, and the denominator is the number of words contained in the URL. Finally we save in the urls list, the url in exactly the same places of the list as the tf / idf. That is, in position 0 of each of the three lists there is information about the same word.

4th) cosine_similarity (query_tf, top_k): We make a list, the distinct_urls, that contains each URL, only once. Then in a double for we calculate the weight W = TFxIDF of each URL. Then if the weight W is not 0 we add it to the list weight_of_url and in the same place we add the URL to distinct_urls. Then we calculate the cosine similarity of each URL (distinct_urls) with the query. We use the following formula:

$$S_{cosine}(q,d) = \cos(\theta) = \frac{\vec{q} \bullet \vec{d}}{|\vec{q}| \cdot |\vec{d}|} = \frac{\sum_{i=1}^{M} w_{t_i,q} \cdot w_{t_i,d}}{\sqrt{\sum_{i=1}^{M} w_{t_i,q}^2} \cdot \sqrt{\sum_{i=1}^{M} w_{t_i,d}^2}}$$

. The numerator is the sum for all pages, of the product of the weight of each page with the weight of the query, which is its tf, since idf is 1. The denominator is the product of 2 meters, the weights of the URLs and of query weights.

Now that we have calculated the similarity of each page with the query we calculate the topk URLs that will be returned. Calculate the max cosine similarity, save it first in a list, delete it from the list of cosines and distinct urls and continue the loop until fill in the topK number or until we have another page to return to. We return the results to server.py.