**­CS 7800 Information Retrieval**

**Simple Search Engine**

**Abhishek Singh Thakur Nikhil Chavan**

**(U00933835) (U00969241)**

**Usecase:** To implement a simple search engine that uses the Boolean model and Vector model in fetching the matching documents from the 1400 documents provided in Cranfield Dataset. Finally evaluating the Boolean and Vector model results using the NDCG score and Wilcoxon test.

**Setup:** Downloading the Cranfield Dataset, installing python version which supports the NLTK library. Any IDE which is of personal choice I’ve used Visual Studio Code for writing and compiling python code.

**Building Inverted Index:**

* It starts with splitting the cran.all from Cranfield Dataset into JSON object.
* Looping through all the documents (1400 in the set provided), and for each document, we first process the text to remove stopwords, stemming and lowering the text and then we find all the terms in the documents and its positions. If we find a repeating term in multiple documents, we append the doc Id and its position list to the repeating term.
* Finally, we build the index for the cran.all and save the file into index\_file as a JSON object, which later will be used in fetching the Boolean model and Vector Model results for the queries.

**Query Processing:**

* The basic idea in this process is to get the results for a query using Boolean and Vector model. The query is processed to remove stopwords, spell correction, stemming and lowercase since we followed the same for the data documents text as well.
* While calculating Boolean model results for a query, the processed query is split into tokens and each token is searched against the terms in the index\_file to fetch the matching documents for the terms, the process is repeated till we get the results for all the tokens in the query and finally, we find the documents which are present in the array of documents for each token in the query.
* For calculating Vector Model, we use TF-IDF scores and cosine similarity for ranking of the pages. Higher the cosine similarity score higher is the ranking of a document matching the query.
* Calculating TF-IDF scores we build the term frequencies of the terms in the documents which created the inverted index file. We store the TF-IDF scores in the dictionary as

{“docId1”: {“term1”: TF(term1)\*IDF(term1),“term2”:TF(term2)\*IDF(term2)},

“docId2”: {“term1”: TF(term1)\*IDF(term1)…},

“docId3”: {“term1”: TF(term1)\*IDF(term1),……}

.

.

.

“docIdn”: {….}

}

and store it in tfidf file for future use while calculating the cosine similarity for queries and documents. Below are the formulas for calculating the Term frequencies(TF) and Inverse Document Frequency(IDF)

TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).

IDF(t) = 1 + log\_10(Total number of documents / Number of documents with term t in it).

For getting the results for a query with vector model we first pre-process the query, calculated TF(t) and IDF(t) and store the TF(t) \* IDF(t) in the same format as above. To fetch the relevant documents, we find the cosine similarity between the query terms against the documents involving the query terms.

Cosine Similarity (d1, d2) = Dot product(d1, d2) / ||d1|| \* ||d2||

Dot product (d1, d2) = d1[0] \* d2[0] + d1[1] \* d2[1] \* … \* d1[n] \* d2[n]

||d1|| = square root(d1[0]2 + d1[1]2 + ... + d1[n]2)

||d2|| = square root(d2[0]2 + d2[1]2 + ... + d2[n]2)

More the cosine similarity more is the ranking of the page for the document for the query.

* For reference: <https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/>

**Evaluation:**

* This process includes how better the Boolean model and vector model are fetching the proper document results for a sample of queries. The sample of queries is basically found in query.text which is in a certain format, we open the file and get the queryId and its text for all the queries in the file in a JSON object.
* Before we search for a query sample, we process the query and fetch the results for the Boolean and Vector Models. The fetched documents for a query sample are checked against the query actual results stored in qrels.text
* Finally, we compute NDCG\_5 scores for both Boolean and Vector models and use Wilcoxon-test to get a p-value for comparison (i.e. whether one ranking result is statistically significantly better than the other).
* In the use case we are testing the results against 20 sample queries from query.text and then compute NDCG\_5 score and p\_value. We do this 5 times for a better understanding of the results.

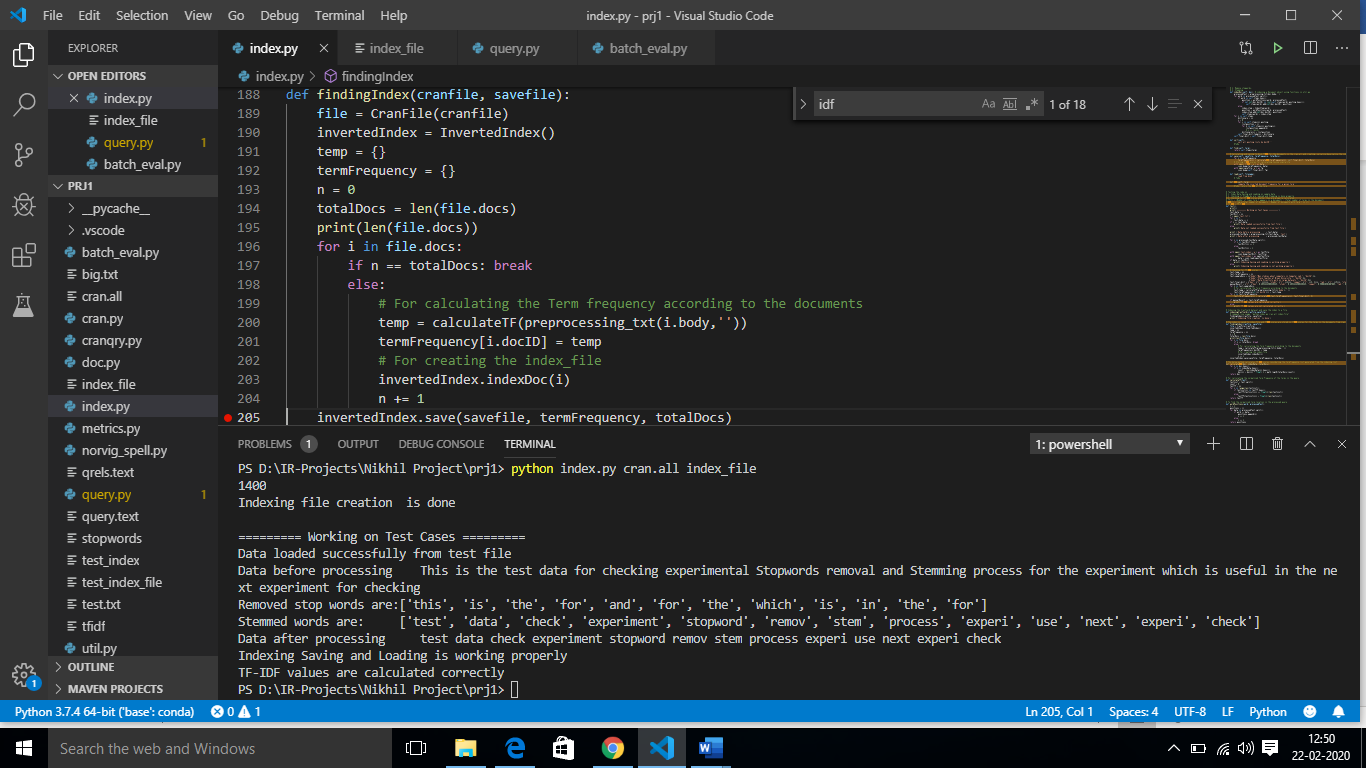
**Test scenarios:**

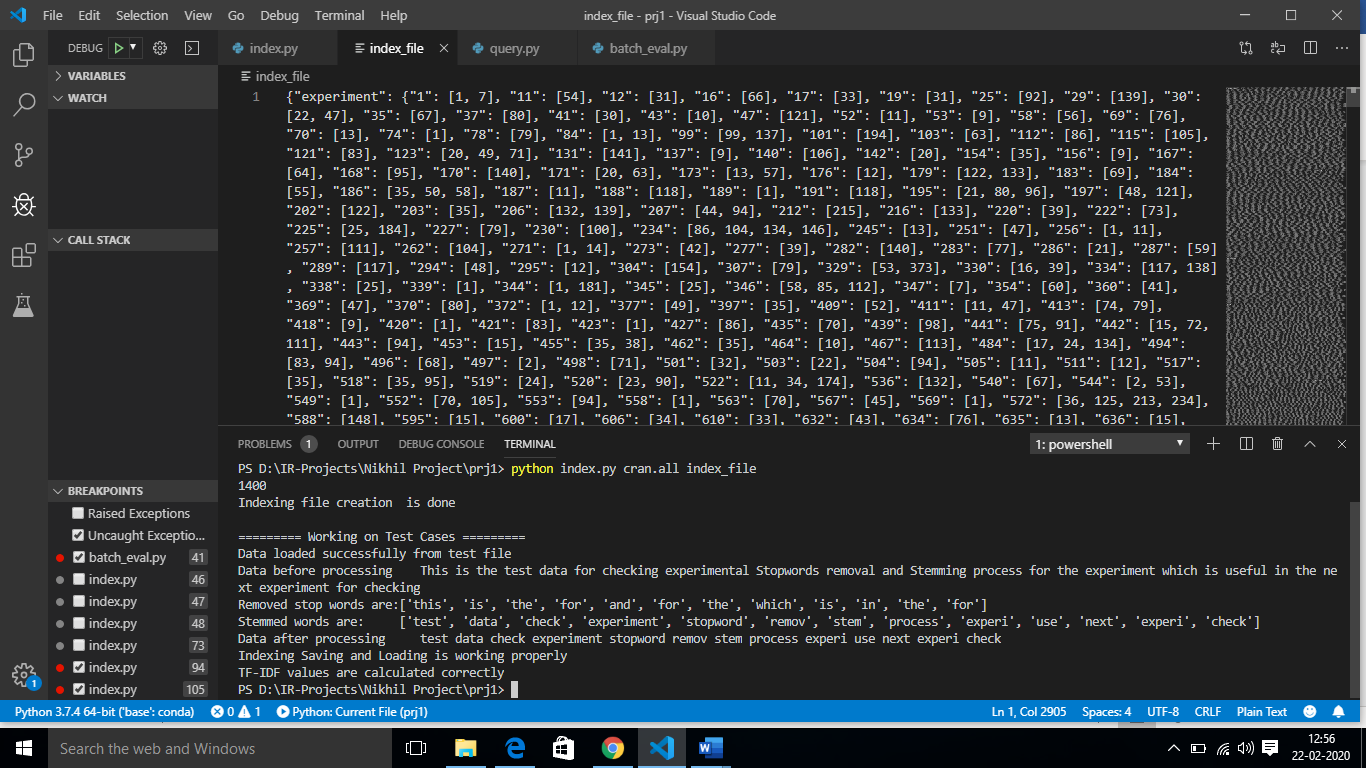
* We make sure that queries and document text are pre-processed i.e. checking for correction, lowercase, stemming and removal of stopwords
* Whether Inverted Index, TFIDF saving and loading working as expected.
* Confirming TFIDF values, cosine similarity is computed correctly or not.
* Whether Sample queries are fetching proper results with Boolean and Vector models, and their respective NDCG\_5 scores with manually calculated results.

**Constructing the Inverted Index and TFIDF:** python index.py cran.all index\_file

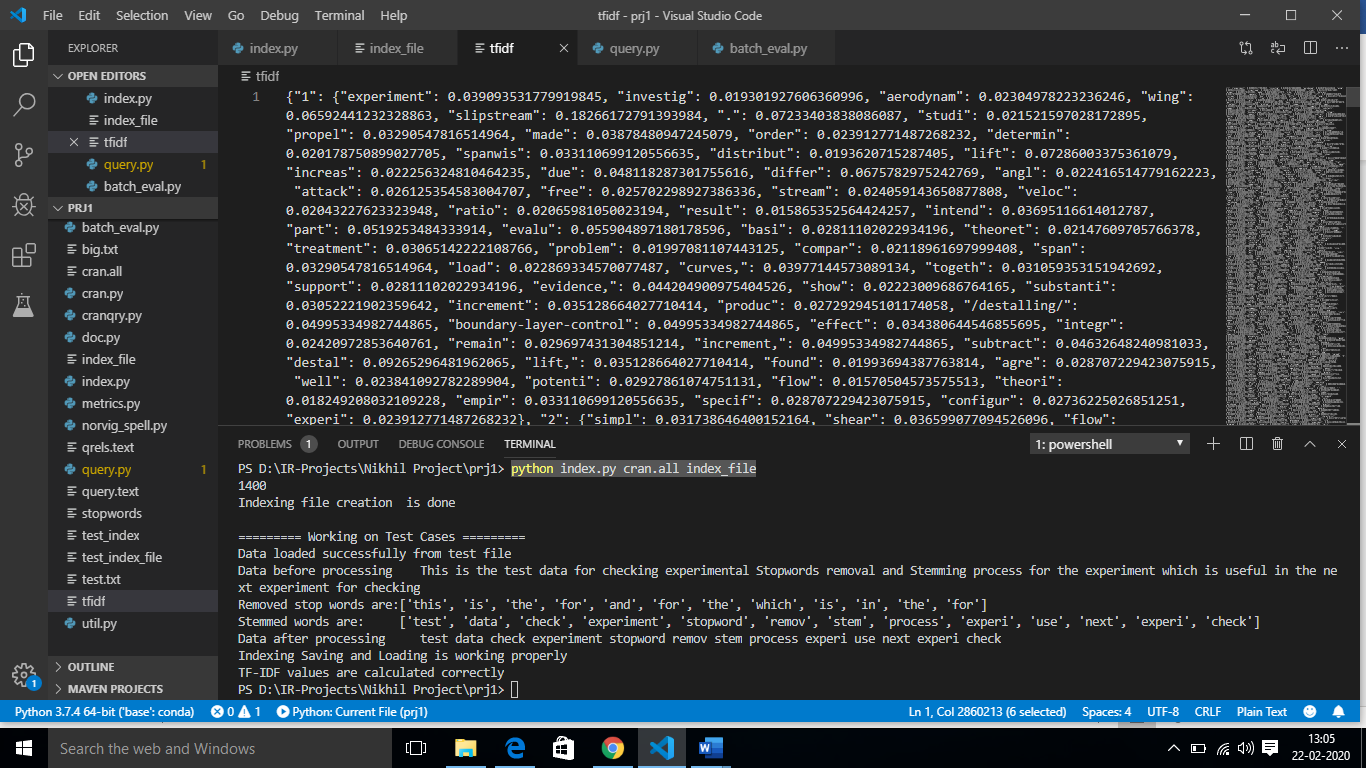
* cran.all – This file contains all the sets of documents upon which indexing is to be created.
* Index\_file – Inverted index is stored in this file.

Results after creating the Inverted index and storing it in index\_file, along with the test cases checking for query and text pre-processing i.e. removal of stopwords, stemming, lowercase and spell correction using Norvig implementation.





The final output i.e. Inverted Index is stored in index \_file. Above is the screenshot of the result in index\_file. Besides, TFIDF scores are stored in tfidf file, below is the screenshot



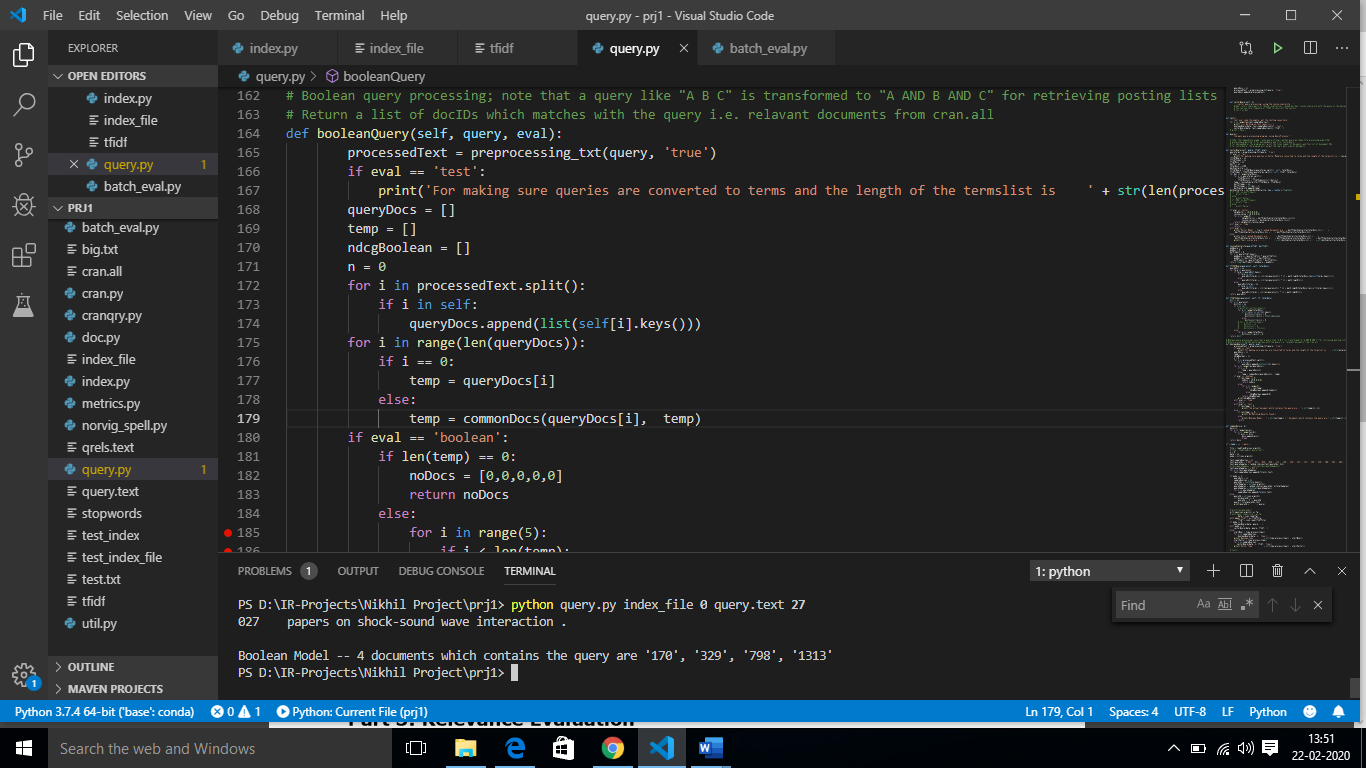
**Query Processing:** python query.py index\_file model-selection query.text qid\_or\_n

* Index\_file: Inverted Index constructed for cran.all documents.
* model\_selection: To know which model to be selected in-order to fetch relevant documents i.e. **‘0’ –** Use Boolean Model, **‘1’ –** Use Vector Model, **‘2’ -** batch evaluation.
* query.text: This file contains the sample queries (included in the Cranfield dataset).
* qid\_or\_n: It is the specific query\_id you choose model\_selection **‘0’** or **‘1’** and in **‘2’** it represents randomly selecting n queries to find the time taken to fetch the Boolean and Vector model results.

**Boolean Model result for query id 27 from query file**: python query.py index\_file 0 query.text 27

027 papers on shock-sound wave interaction.

Boolean Model -- 4 documents which contains the query are '170', '329', '798', '1313'

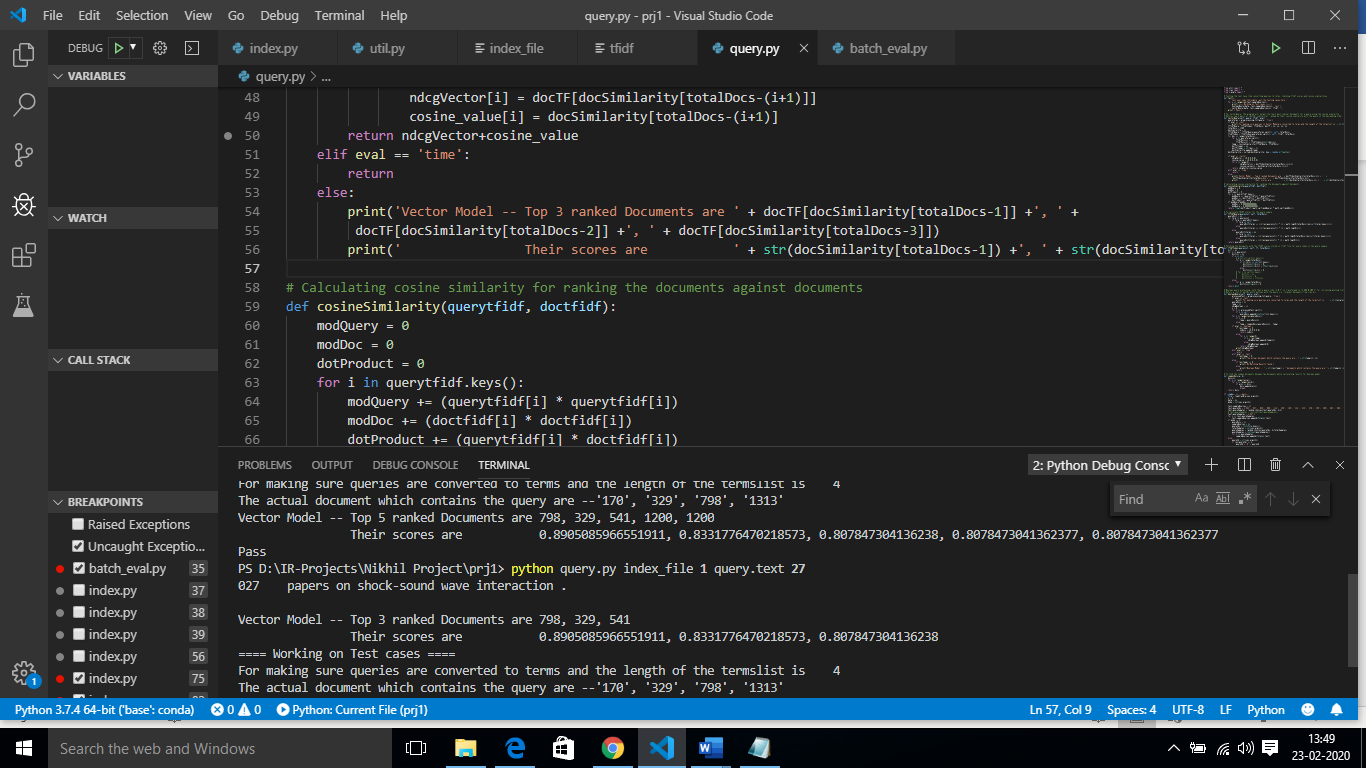


**Vector Model result for query id 27 from query file**: python query.py index\_file 1 query.text 27

027 papers on shock-sound wave interaction.

Vector Model -- Top 3 ranked Documents are 798, 329, 541

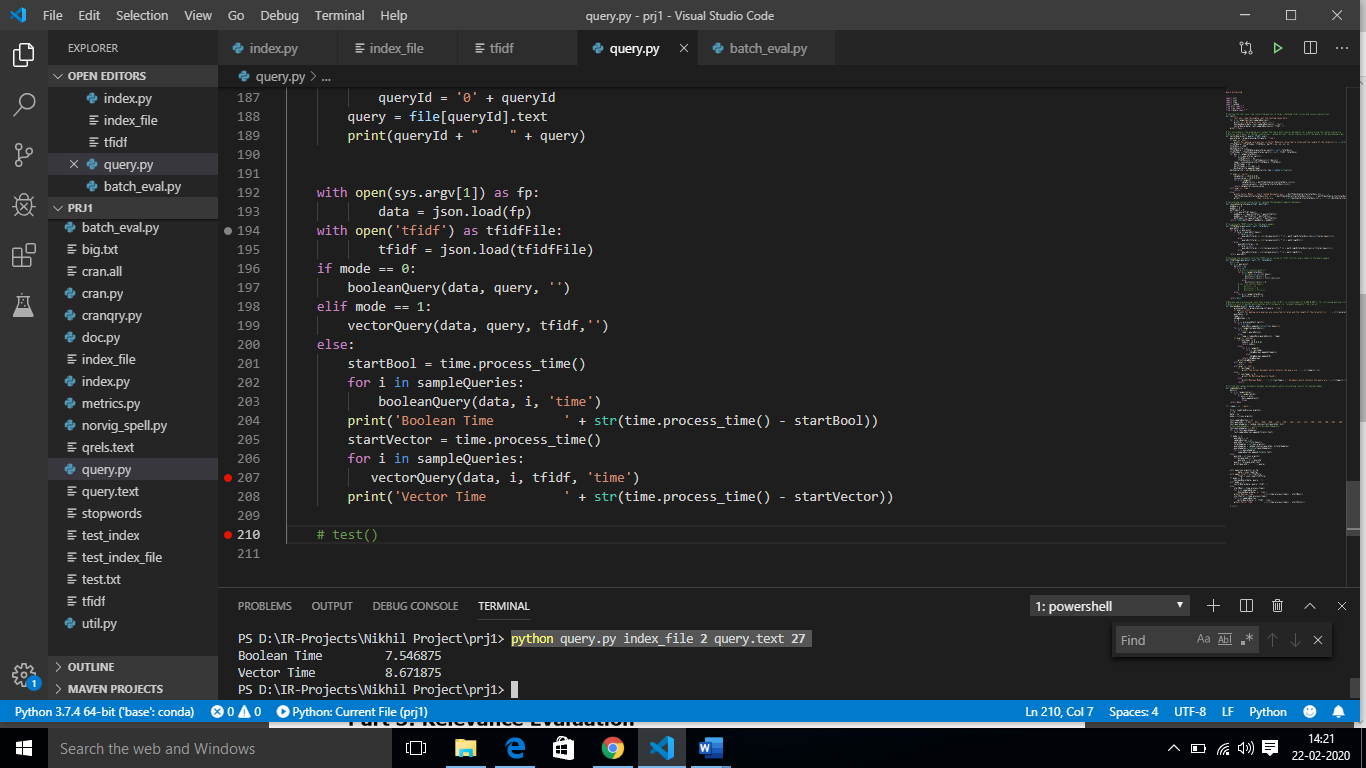
Their scores are 0.8905085966551911, 0.8331776470218573, 0.807847304136238



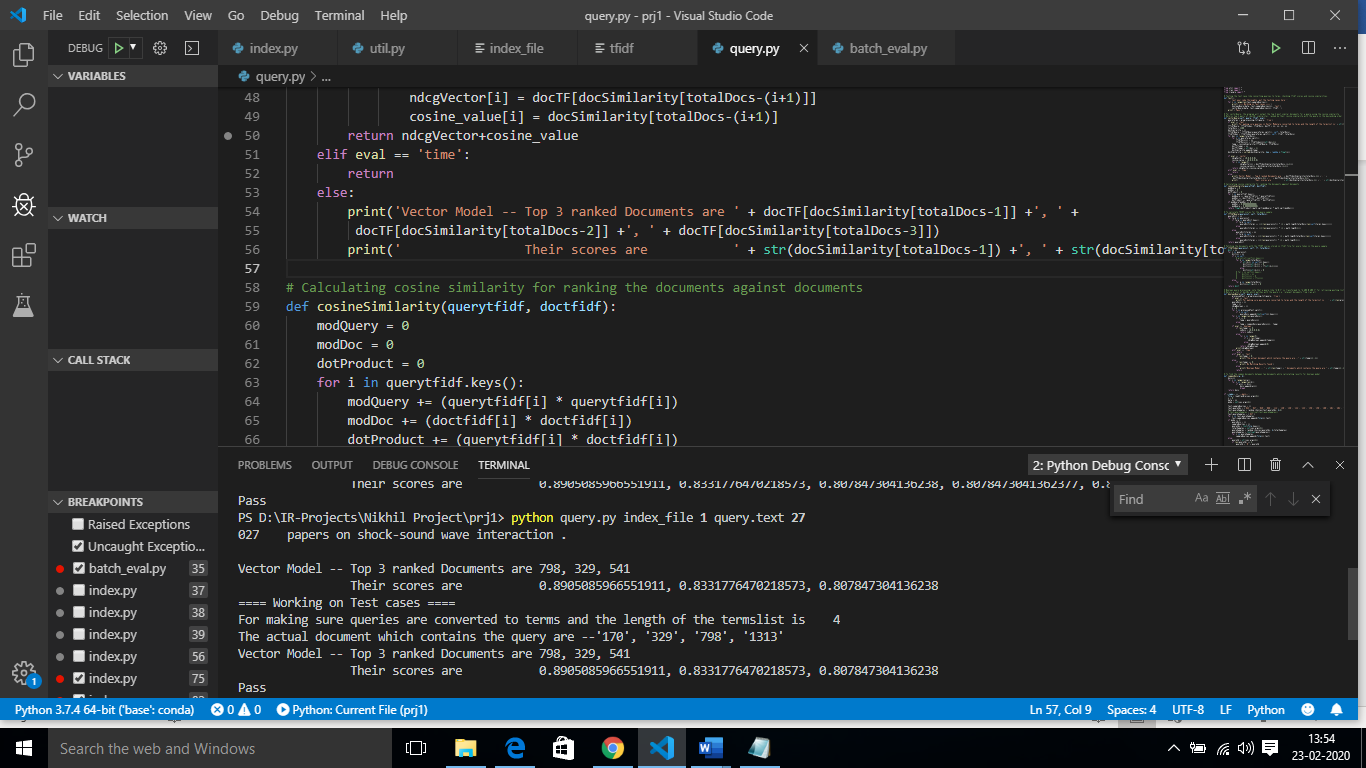
**Time taken in calculating sample queries of size 27:** python query.py index\_file 2 query.text 27

Boolean Time 7.546875

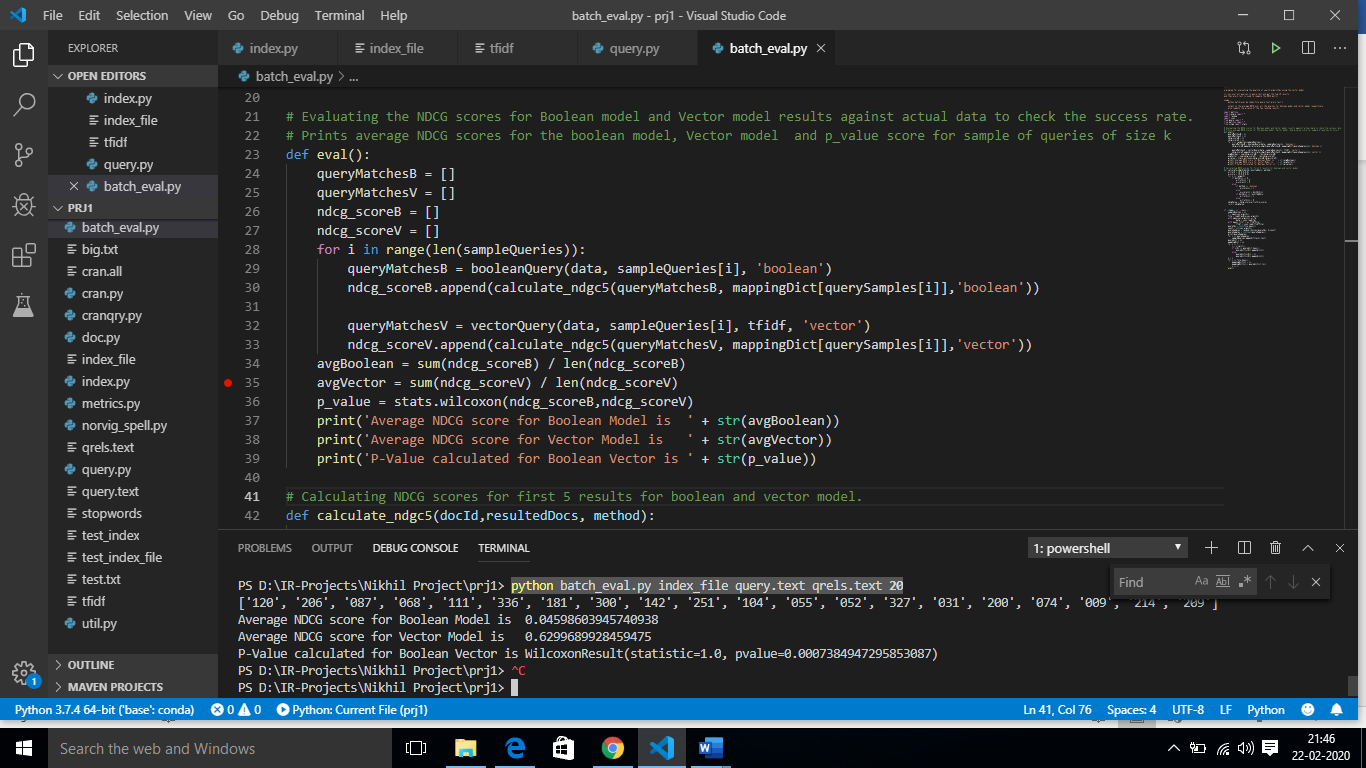
Vector Time 8.671875



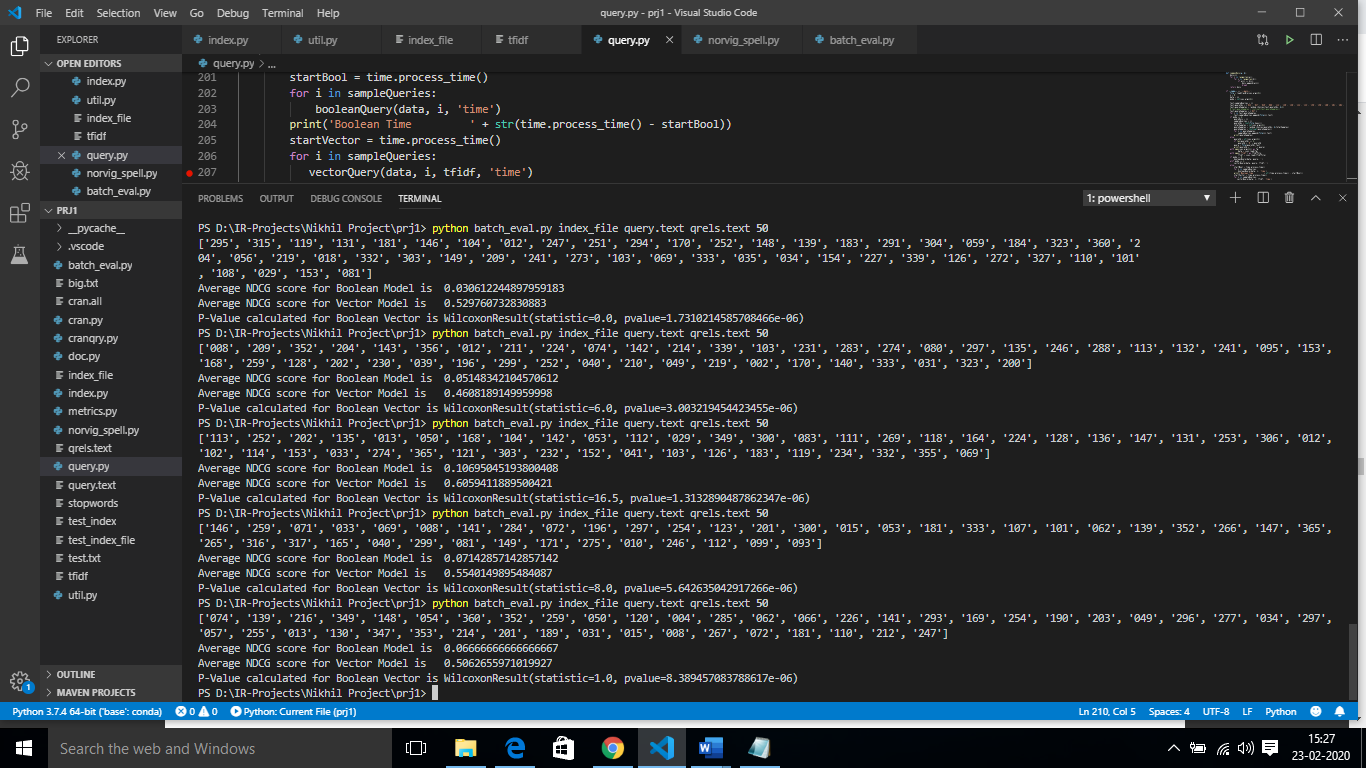
**Testing the Boolean and Vector model results against actual results**



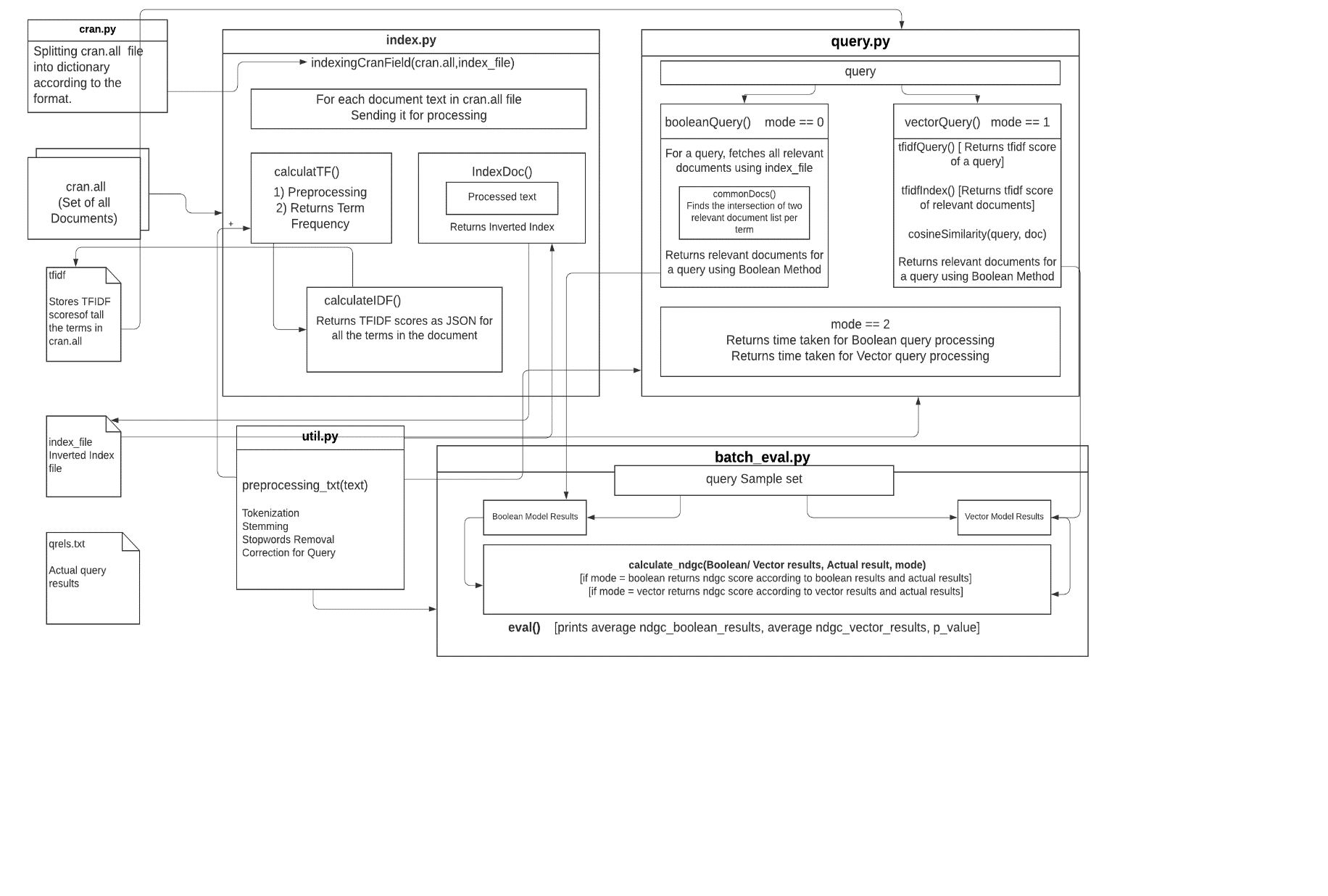
**Evaluating the results with the help of NDGC scores and Wilcoxon-test for better understanding of results:** python batch\_eval.py index\_file query.text qrels.text 20



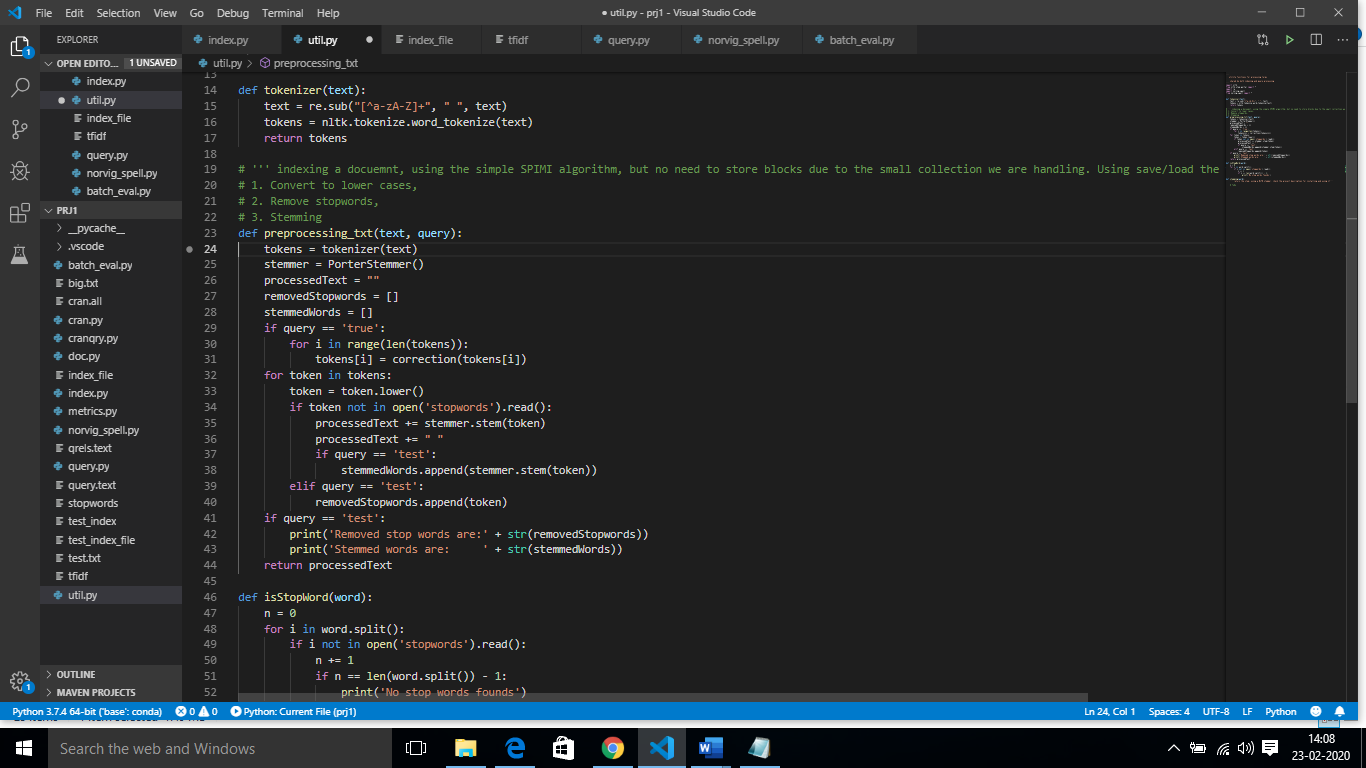
The below screenshot shows the Average NDGC\_5 score for the Boolean and Vector model and the P-value to understand whether one ranking result is statistically-significantly better than the other.



**Design Implementation:**



**Processing the text in util.py**: Text would be processed in order to achieve tokenization, lower case, removal of stopwords and stemming. After tokenization a text using tokenizer(text), preprocessing\_txt(text, query) takes 2 arguments text represents the current text to be processed in achieving the goals mentioned above whereas if a query is true states it queries are to be corrected before processing and when a query is test it prints removed or stemmed words for the test cases. Finally, it returns the processed text as a string.

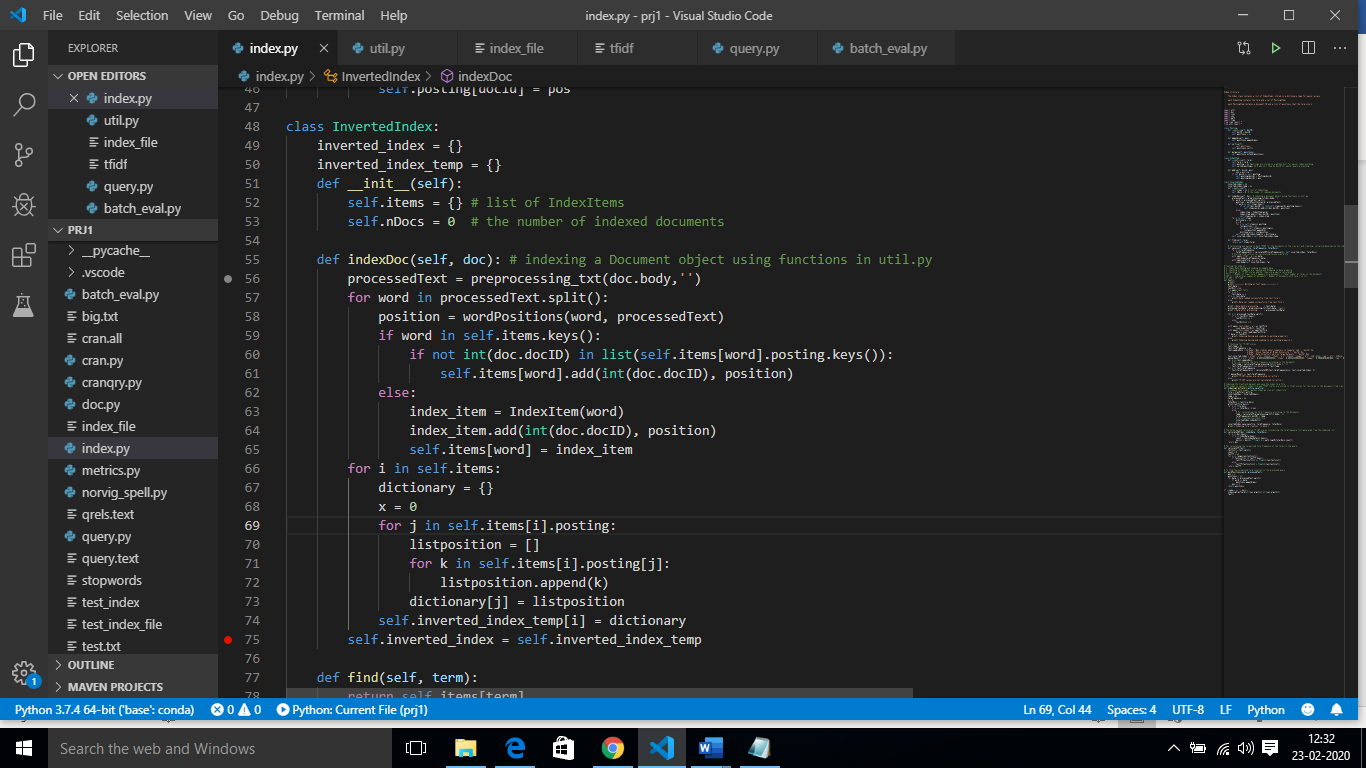
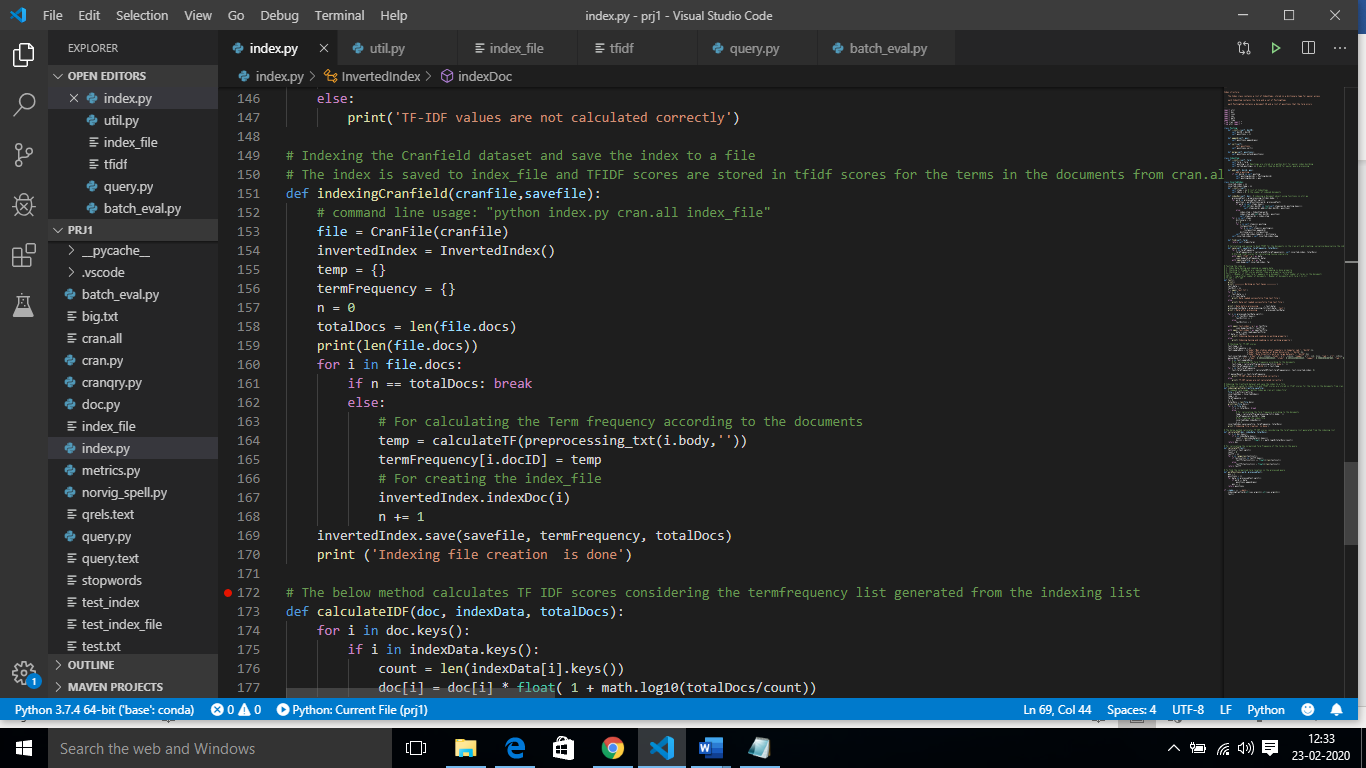


**Inverted Index creation Code implementation:**

It starts with importing the cran.all file and storing the JSON object in a file variable, in indexingCranfield(cranfile, savefile) looping through all the documents in the file for each document we extract its docID and text then we process the text in order to calculate term frequency and create an inverted index for each term in the cran.all file. Finally saving the tfidf values of the terms in the tfidf file and inverted index to index\_file.

While creating the inverted index for a document in indexDoc(self,doc) we consider its docId and text, we pre-process the text first and store it in processedText variable, find all the terms in the document then for each term in the document we find the locations of the term using wordPositions(), we initially generated a dictionary with term with its related document and its positions in the document working on.

Besides, while working with the next document if we find the same term appearing in the previous documents, the docId and the position list are appended to the term in self\_inverted\_index dict. Once all the documents are looped over and terms are being associated with its docId and position list, these are saved as a JSON object in index\_file.



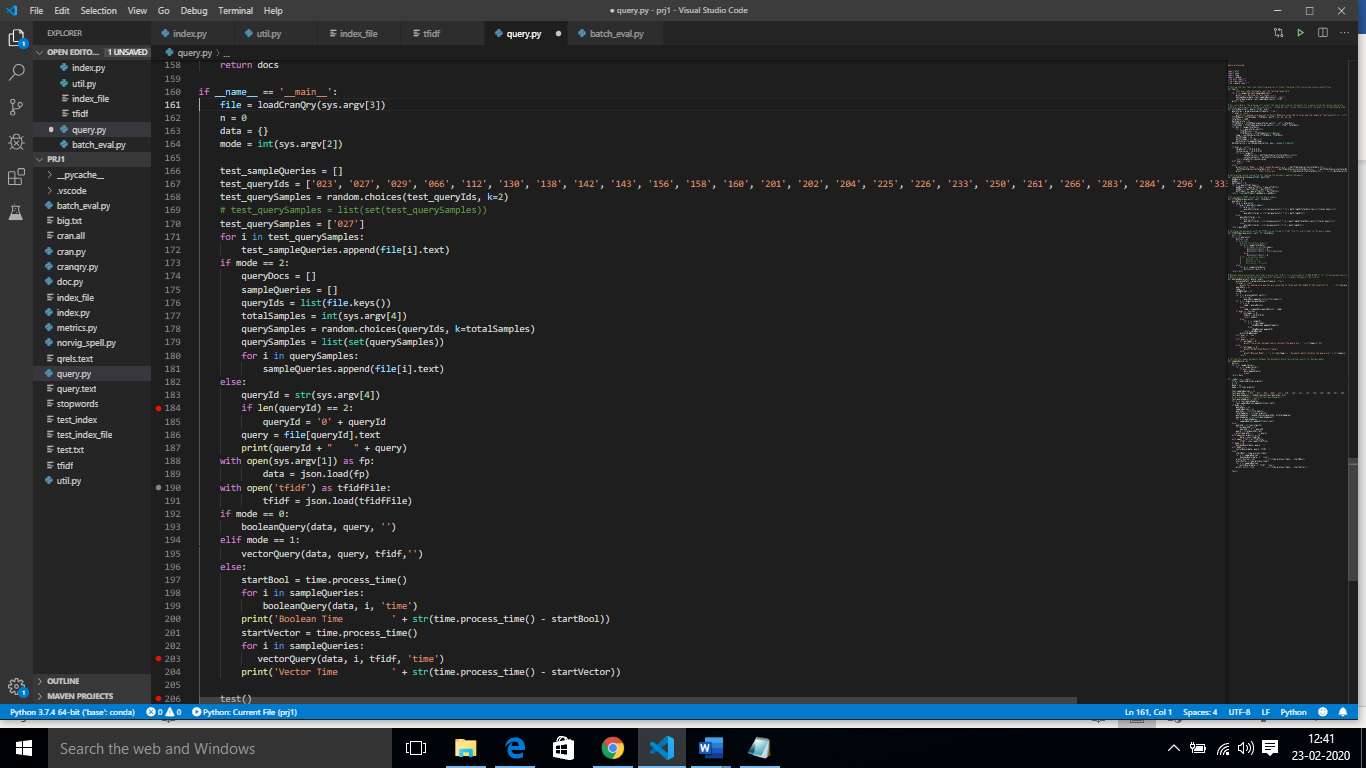
**Calculating TF-IDF for the document set in cran.all**

While iterating through all the documents in above scenario, we also calculate TF for each document text in calculateTF() and is stored as JSON in the format of {docId: {‘term1’: count}}, then once inverted index and Tf is generated we calculate the TF-IDF using calculateIDF() which is TF(t)\*IDF(t), where IDF(t) is calculated in order to find the importance of word in the all the n documents in cran.all, which are later used in the vector space model to fetch the relevant documents for the query sample.



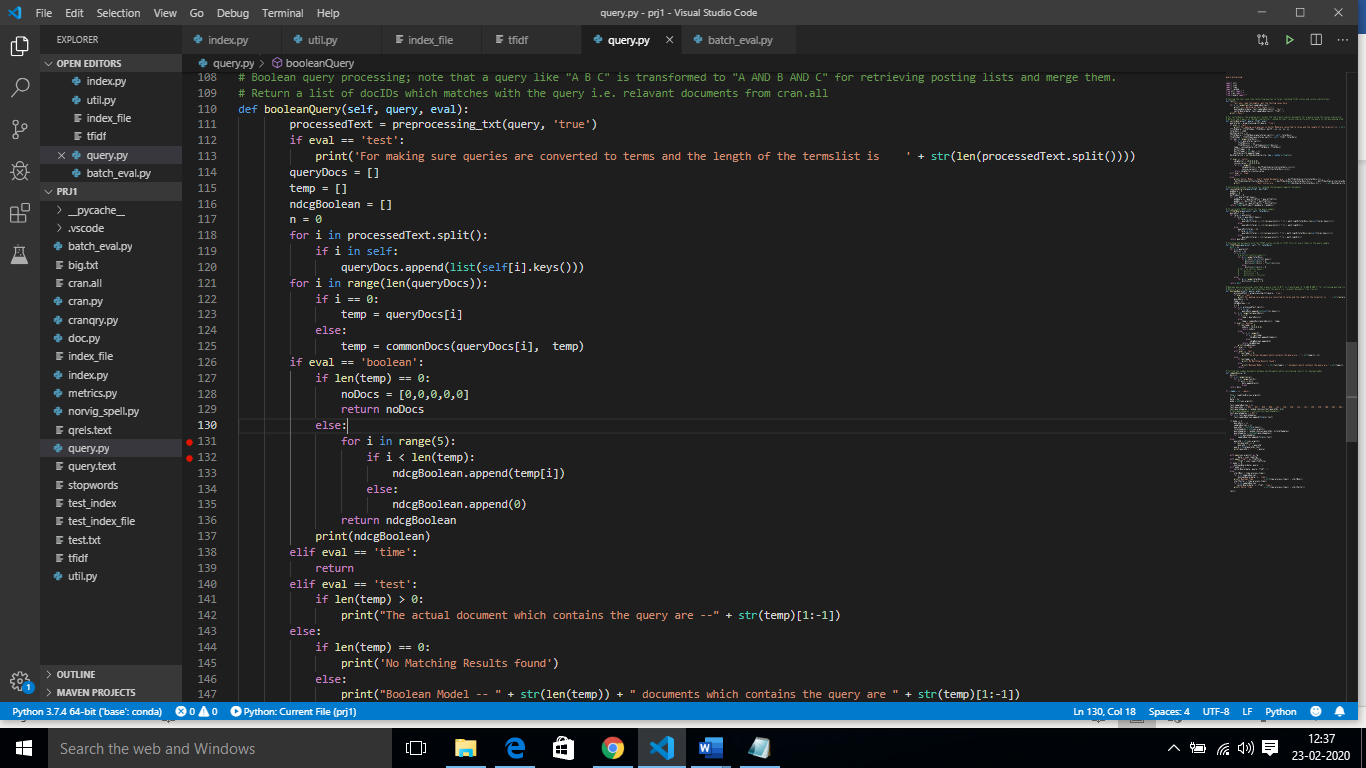
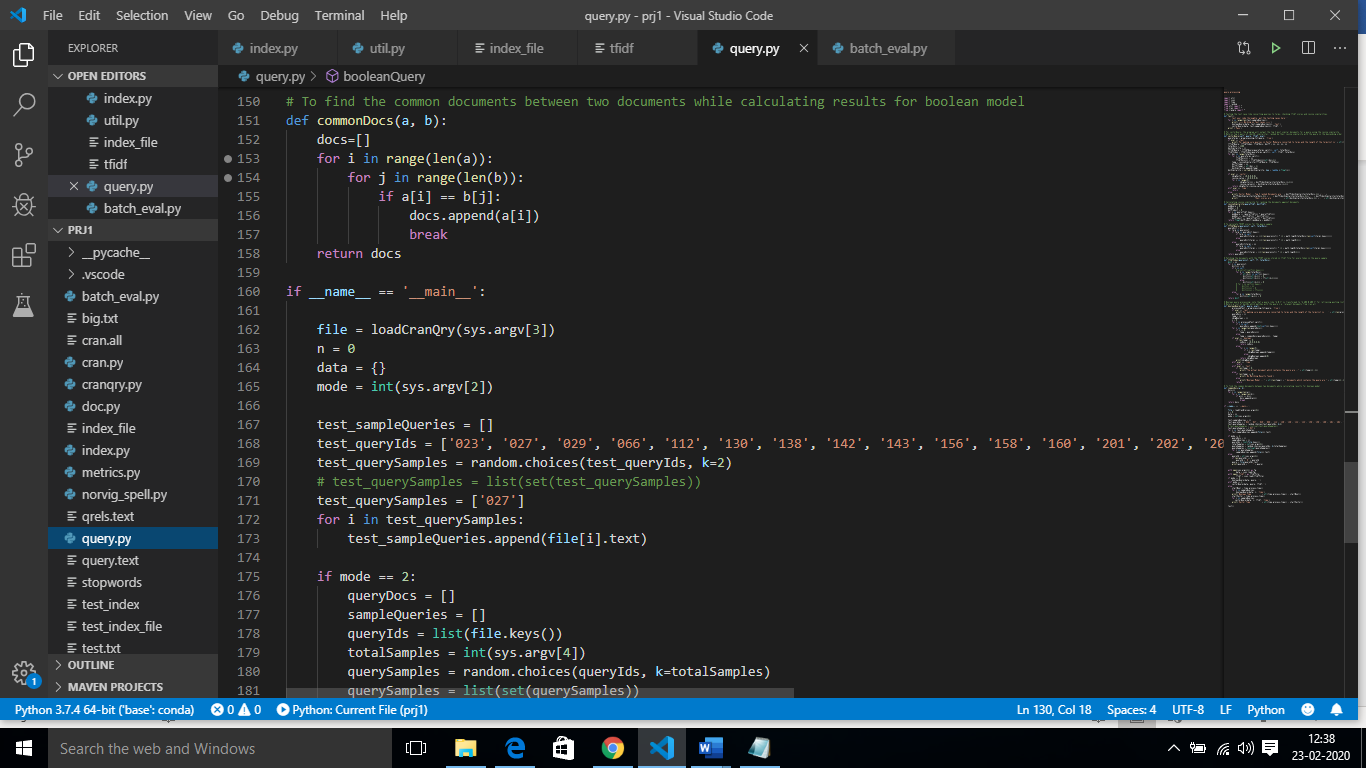
**Selecting of the Modes based on the command line input:**

While running the query.py the query.text which contains all the queries along with its associated id’s are stored in the form of JSON object and looped over each query then for each query object the text is pre-processed where an extra feature of correction is done which is discussed in the above util.py file. When the mode == 0,1 and if the length of the queryid is 2 then 0 is appended since the id’s are stored in the same way as strings. Depending upon the mode = 0 boolean model is selected, 1 for vector model and 2 to find the time taken for the Boolean model and Vector model to fetch the relevant documents for a sample of queries of length queryid.



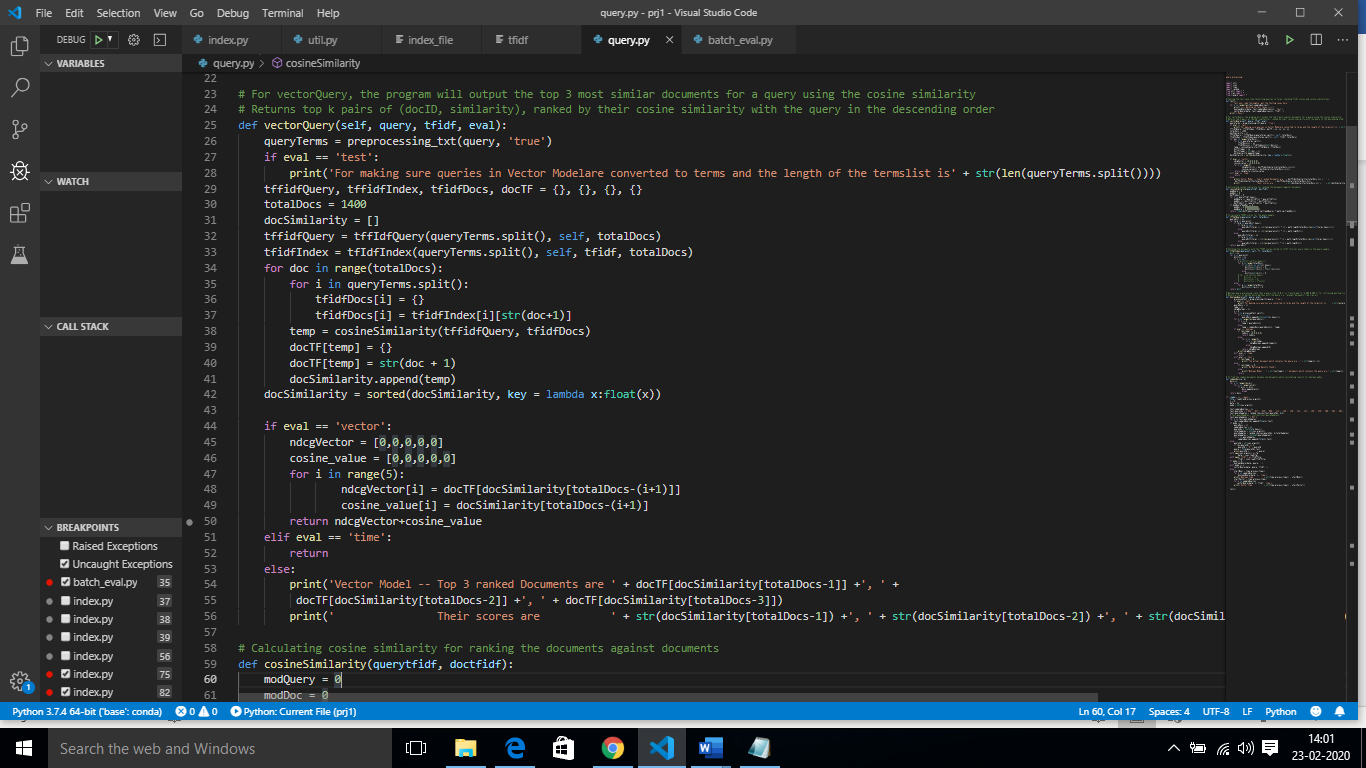
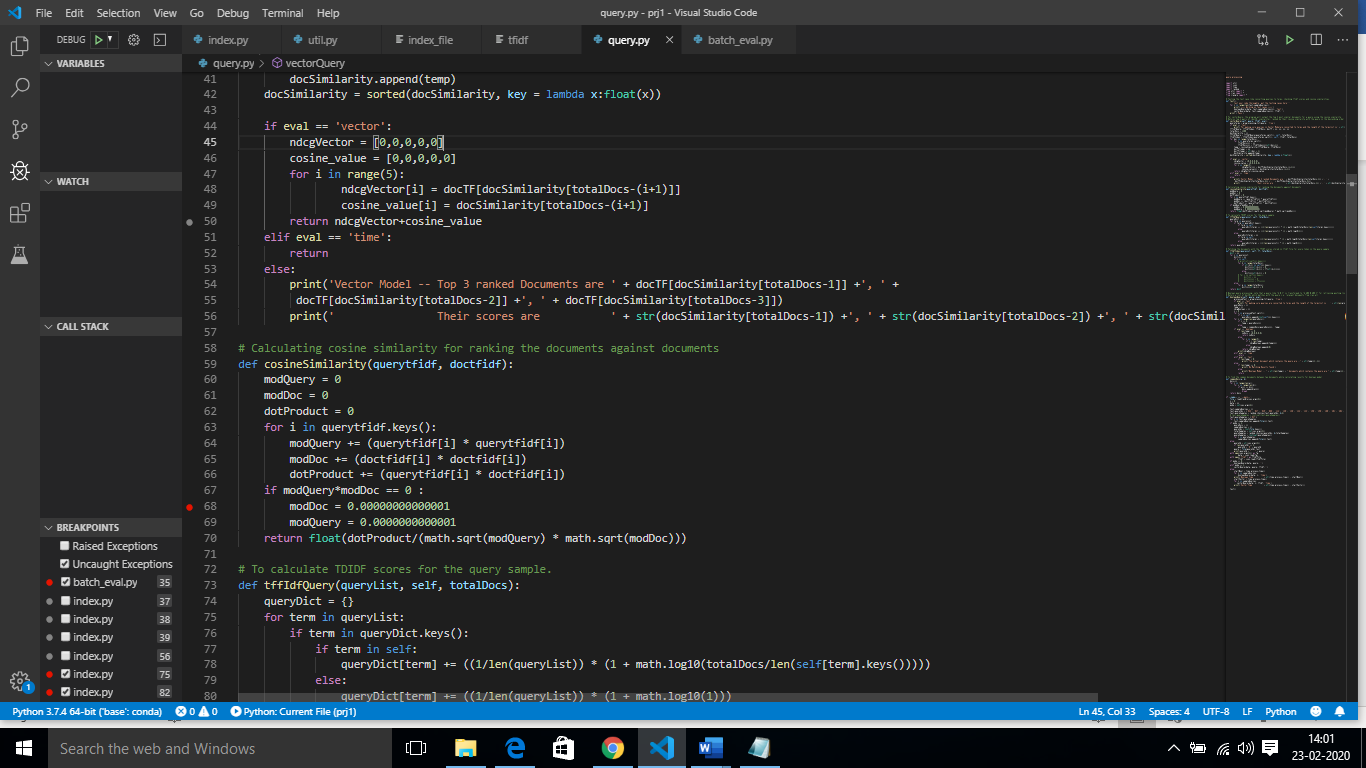
**Boolean Model Code Implementation:**

The booleanQuery() method prints the relavant documents for a query by checking each query term from the processed query in the inverted index stored in the index\_file and stored in the array. commonDocs() fetches all the similar doc id between each term results and prints it on a console the fetched docId’s.

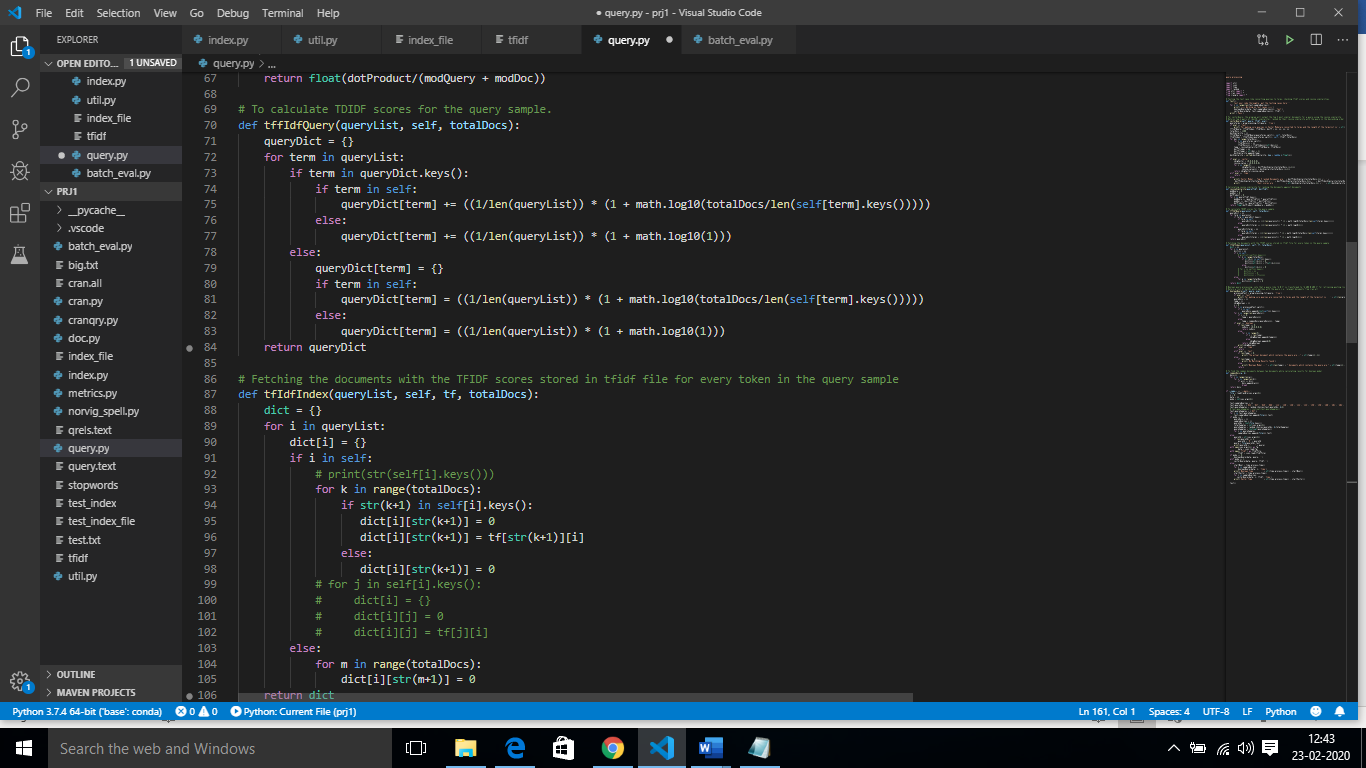
**Vector Model implementation:**

For a query in order to fetch the relevant documents in this model, we find the cosine similarity of the documents for each term in the processed query and this is done in cosineSimilarity() method, where it includes query vector and relevant document vector calculate its Dot product and dividing it with the modulus of the two vectors finally returning the decimal cosine value for each document against a query.



Calculating TFIDF scores for the query sample, fetching the documents with the terms in the query sample:

In order to calculate cosine similarity of a particular query, we send the tfidf scores for every token in the query and also the tfidf scores of the terms in the query in all the documents, so we send an array of tfidf scores of query and also documents involving the query terms. tfIdfQuery() method returns the dictionary with keys as a term in the query sample along with its tfidf scores. tfIdfIndex() return the tfidf scores of the docId’s for every term in the sample query. The results of both the methods are sent to cosineSimilarity function to get the ranking of the pages. The more the cosine similarity score more is the query terms in the document. Finally, we print the top 3 ranked document Id’s along with its cosine similarities.

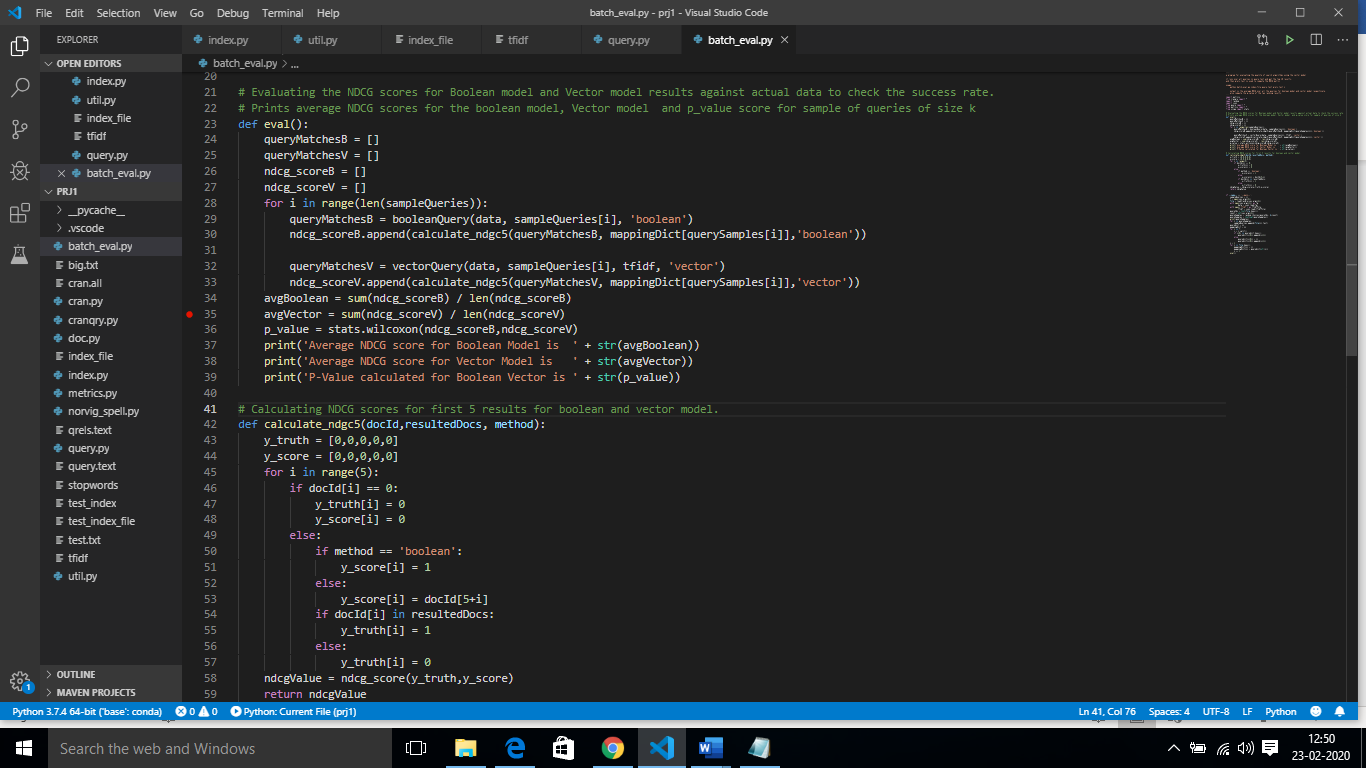
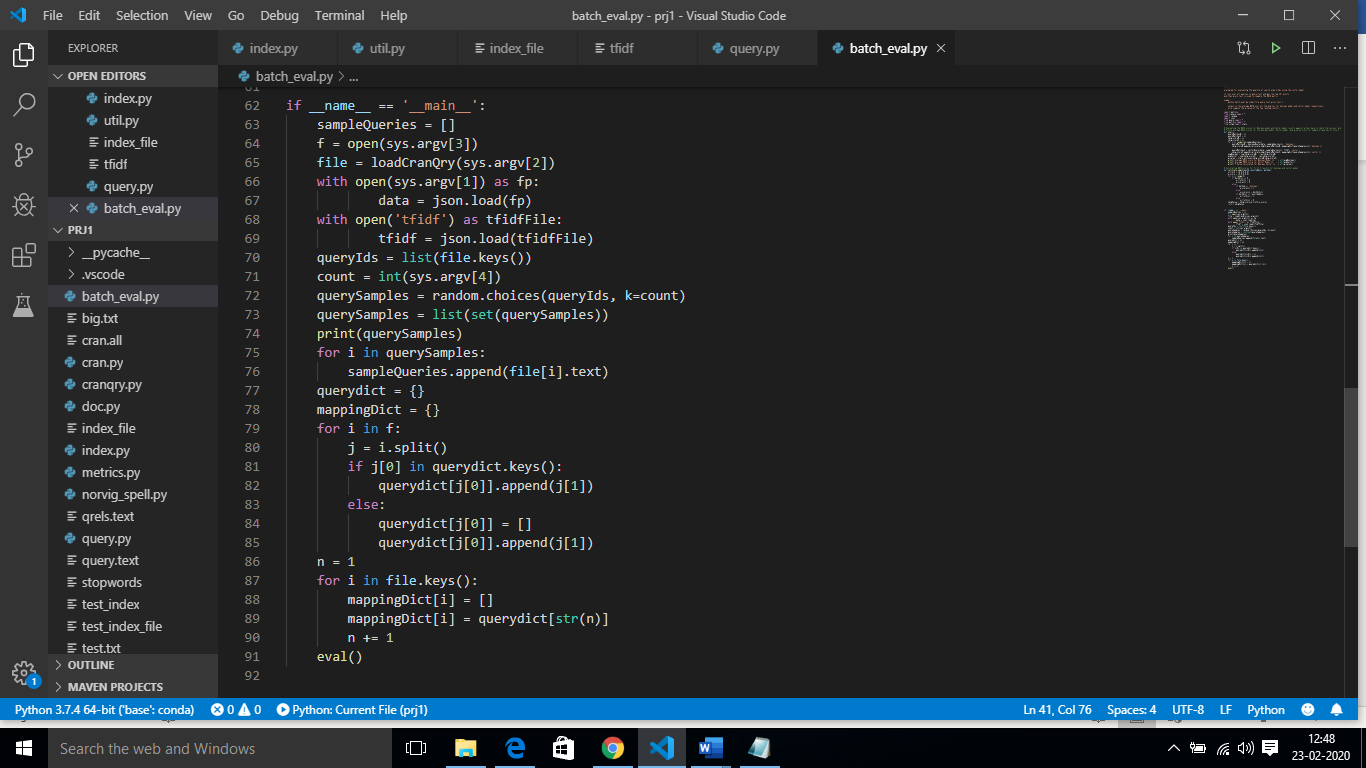


**Batch Evaluation:**

For evaluating the results we load index\_file query.text and qrels.text file (which contains actual results for the query), the first task is mapping the Query samples from query.text with the qrels.text since docID’s are not continues in the query.text for easy evaluation of Boolean and vector model results against the actual results.

Calculating NDGC\_5 scores and the Wilcoxon results.

The eval() method calculates the Boolean and Vector model results for a sample of queries by calculating NDGC\_5 scores for both the models. For a query result, we find its query id in the mapping list and check for the actual results. The calculate\_ndgc5() method takes 2 args where y\_true is actual documents Boolean values i.e. if Boolean model return x, y as docId’s and are present in the actual result list i.e. a,b,d,x,z, then y\_true would be 1,0,0,0,0 and y\_score would be 1,1,0,0,0 and in case of Vector model y\_score would be its cosine similarity values i.e. 0.82, 0.74, 0.71, 0.45, 0.40. Finally the average NDGC scores for Boolean and vector models are printed on console and Wilcoxon test (p\_value) is also printed to check the correctness/ statistics between the 2 average generated NDGC values.

**Test Cases:**

Inverted Index Creation and TF-IDF scores calculation for Testing sample:

Document 1: The game of life is a game of everlasting learning

Document 2: The unexamined life is not worth living

Document 3: Never stop learning

Actual Calculated Indexing after processing the documents:

Manual Result {'game': {'1': [1, 3]}, 'life': {'1': [2], '2': [2]}, 'everlast': {'1': [4]}, 'learn': {'1': [5], '3': [3]}, 'unexamin': {'2': [1]}, 'worth': {'2': [3]}, 'live': {'2': [4]}, 'never': {'3': [1]}, 'stop': {'3': [2]}}

Manual TF-IDF Result {'1': {'game': 0.590848501887865, 'life': **0.2352182518111363**, 'everlast': 0.2954242509439325, 'learn': **0.2352182518111363**}, '2': {'unexamin': 0.3692803136799156, 'life': **0.29402281476392034**, 'worth': 0.3692803136799156, 'live': 0.3692803136799156}, '3': {'never': 0.4923737515732208, 'stop': 0.4923737515732208, 'learn': **0.3920304196852271**}}

Query Processing and Boolean and Vector Model results for **“life and learning”**

**Life = Normalized TF(life in query) \* IDF(life in Manual Result) = 0.5\*1.1760912591 = 0.5880456295**

**Learn = NormalizedTF(learn in query) \* IDF(learn in Manual Result) = 0.5\*1.1760912591 = 0.5880456295**

**Query doc1 doc2 doc3**

**Life 0.5880456295 0.2352182518111363 0.29402281476392034 0**

**Learn 0.5880456295 0.2352182518111363 0 0.3920304196852271**

Cosine Similarity (query, d1) = Dot product(query, d1) / ||query|| \* ||d1||

Dot product (query, d1) = query[0] \* d1[0] + query[1] \* d1[1] \* … \* query[n] \* d1[n]

* (**0.5880456295 \* 0.2352182518111363** ) + (**0.5880456295 \* 0.2352182518111363**) =

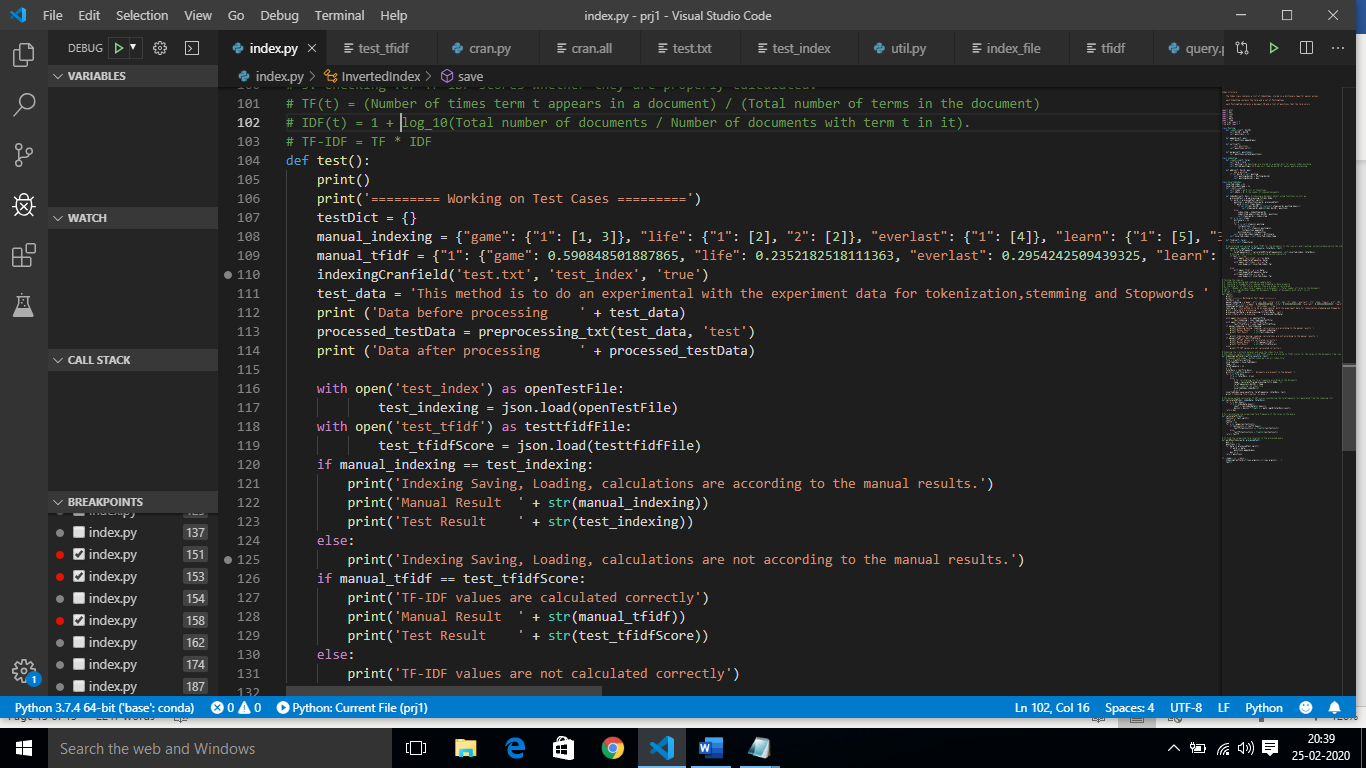
||query|| = square root(query[0]2 + query[1]2 + ... + query[n]2) = square root(**0.5880456295\*\*2+0.5880456295\*\*2) =**  square root(0.6915953248) = **0.8316221046**

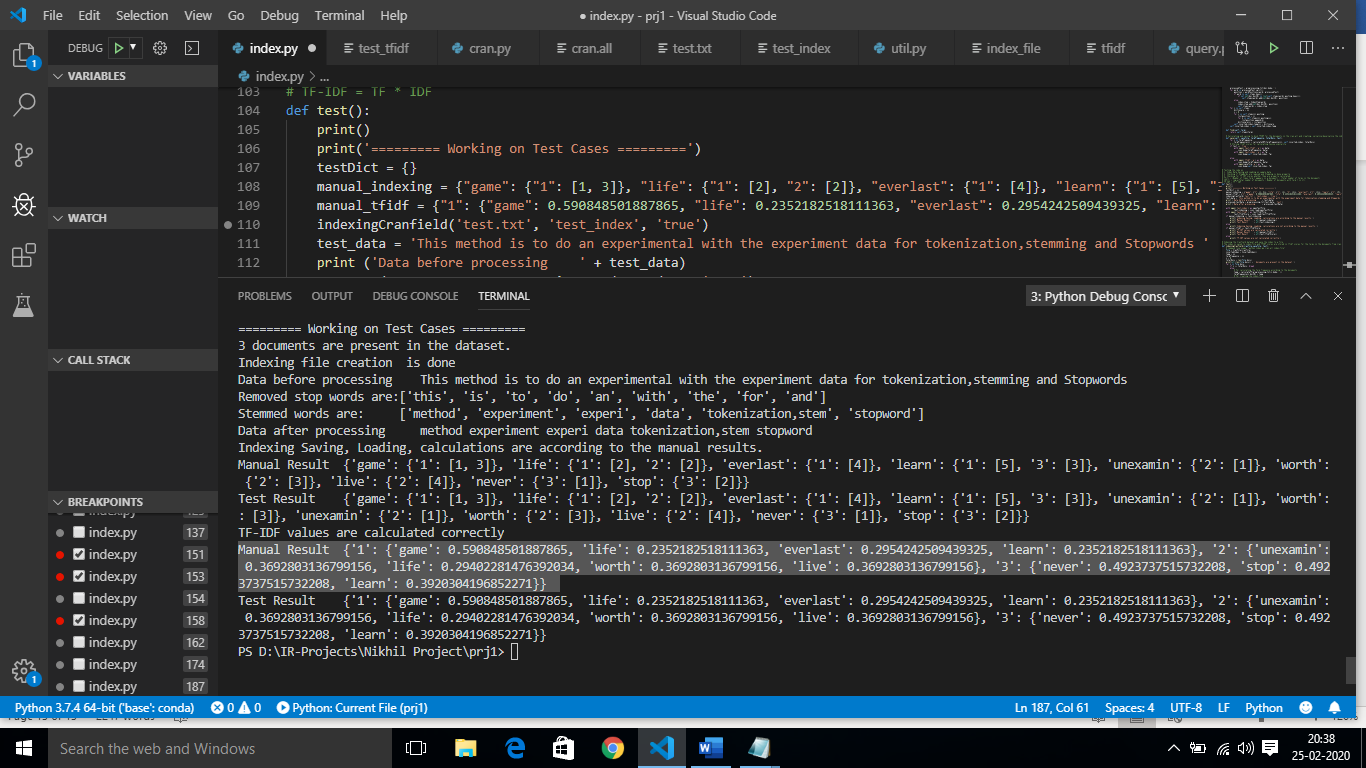
||d1|| = square root(d1[0]2 + d1[1]2 + ... + d1[n]2) = square root(**0.2352182518111363\*\*2 \* 0.2352182518111363\*\*2** ) = square root(**0.11065525196) = 0.33264884181**

Cosine Similarity (query, d1) = **(0.2766381299)/(0.27663812991) = 1**

Cosine Similarity (query, d1) **=** 0.7071067811865476

Cosine Similarity (query, d1) = 0.7071067811865475

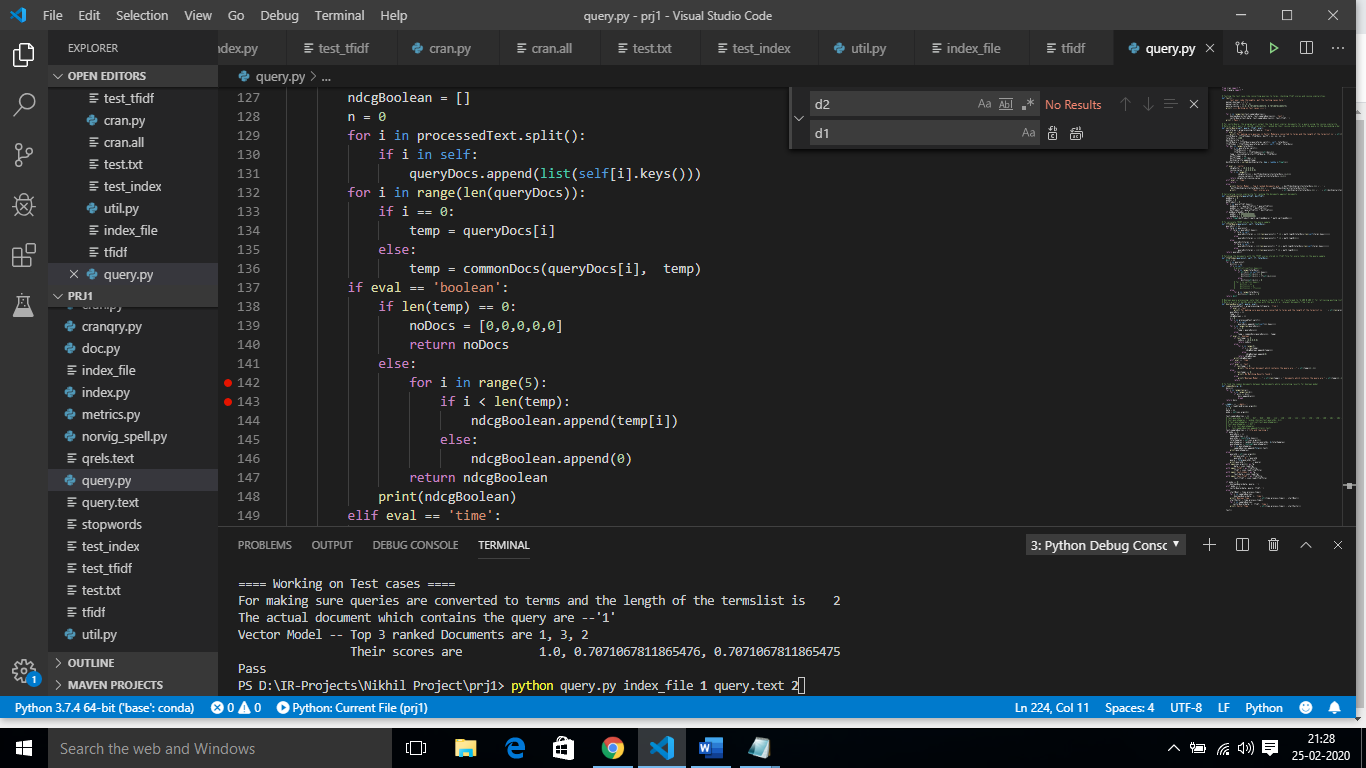




Boolean Result: 1

Vector Model Result: 1, 3, 2

Cosine Similarity: 1.0, 0.7071067811865476, 0.7071067811865475



Actual Data to be fetched for query life and learning is : 1,3

Boolean Model = NDGC\_5(y\_true, y\_score) = NDGC([1,0,0,0,0] + [1,0,0,0,0]) = 1.0

|  |  |  |
| --- | --- | --- |
| docId | y-true | Y\_score |
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

Vector Model = NDGC\_5(y\_true, y\_score) = NDGC([1,0,0,0,0] + [1.0,0.7071067811865476, 0.7071067811865475,0,0]) = 0.6934264036172708

|  |  |  |
| --- | --- | --- |
| docId | y-true | Y\_score |
| 1 | 1 | 1.0 |
| 3 | 1 | 0.7071067811865476 |
| 2 | 0 | 0.7071067811865475 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

