COMP4133 Information Retrieval

Technical Report

Kindly note that the source code of this project will not be available since it may break the rules.

Contents

1. Introduction

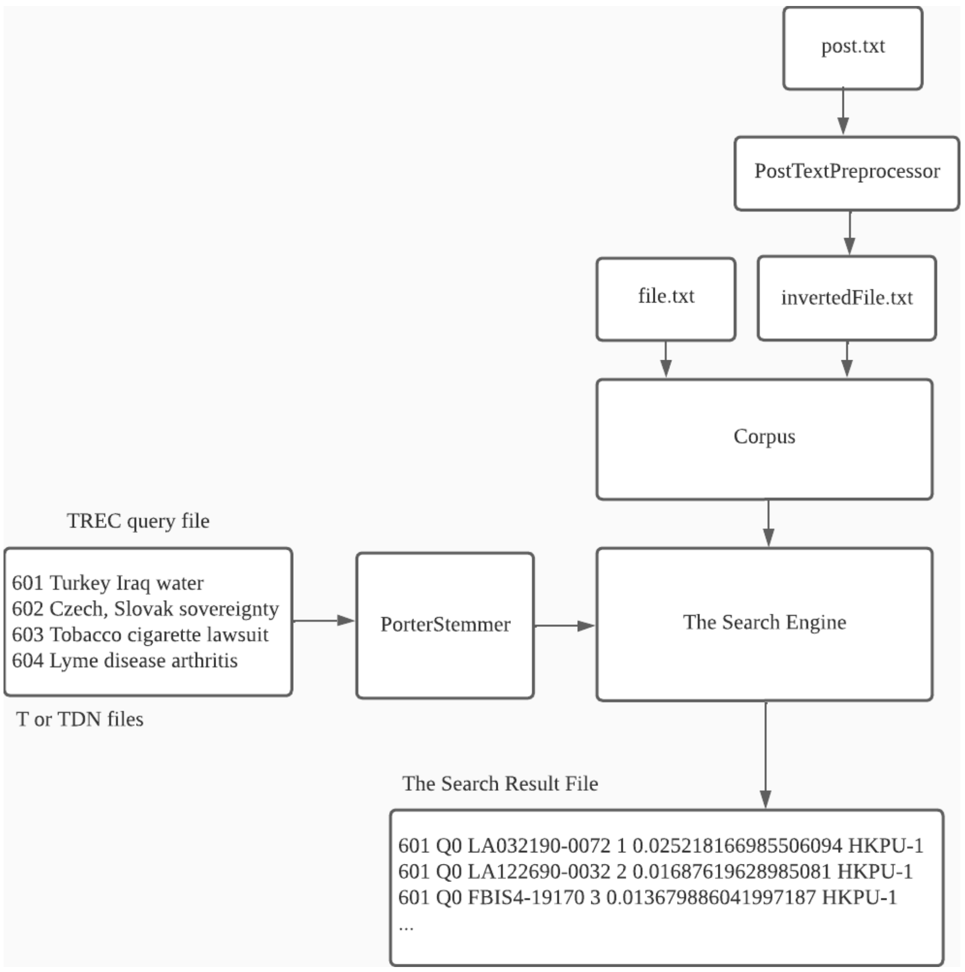
This report is to introduce and explain the search engine we implemented in details, which will include its architecture, retrieval models and the experiment results. And this report will also compare and analyze the advantages of one retrieval model over the other by numerical measurements and different evaluations.

The whole system was entirely developed in Java, and to further adapt different implementation, the system has three optional retrieval models: Boolean, Fuzzy Boolean and Vector Space retrieval models, as well as an advanced model that was innovated by us and with the hope of improving the effectiveness and usage of our system.

At the end of the section, the report will be about the management of this project.

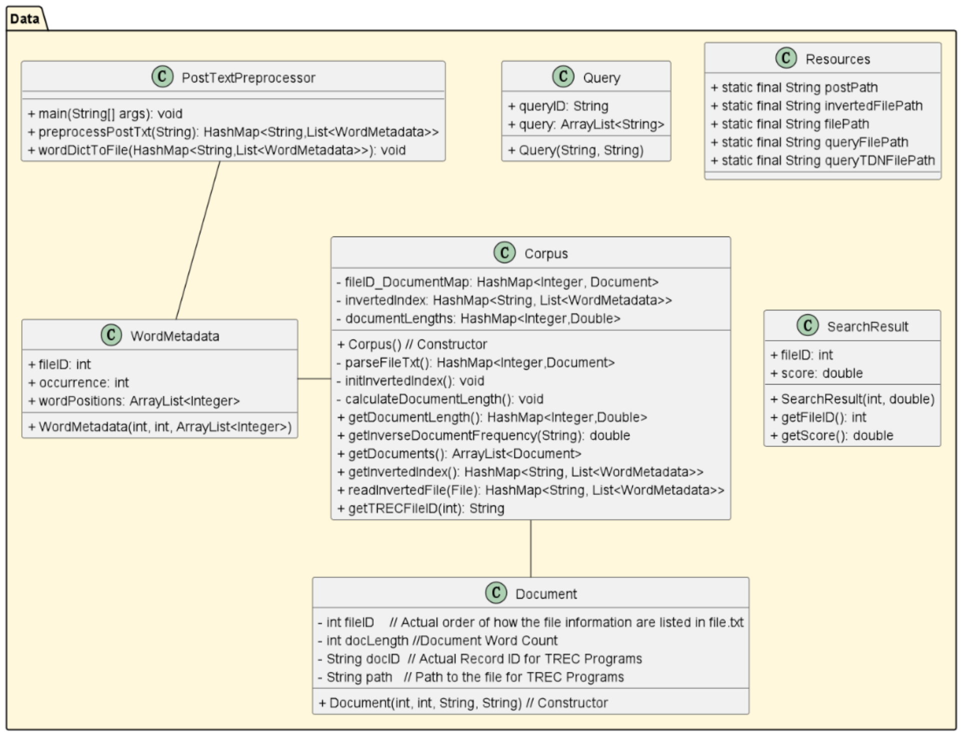
2. Search Engine Architecture

The architecture organized the process of retrieving results.


post.txt
PostTextPreprocessor
file.txt
invertedFile.txt
Corpus
TREC query file
601 Turkey Iraq water
602 Czech, Slovak sovereignty
603 Tobacco cigarette lawsuit
604 Lyme disease arthritis
T or TDN files
PorterStemmer
The Search Engine
The Search Result File
601 Q0 LA032190-0072 1 0.025218166985506094 HKPU-1
601 Q0 LA122690-0032 2 0.01687619628985081 HKPU-1
601 Q0 FBIS4-19170 3 0.013679886041997187 HKPU-1
...

We would preprocess post1.txt, and create a new inverted file, which would be the content of the corpus together with file.txt. We use porter stemmer for processing queries. After running the program, it will produce the search result file, which can be used for evaluation.

Data classes, which are essential for models to process data with different format.


Data
PostTextPreprocessor
+ main(String[] args): void
+ preprocessPostTxt(String): HashMap<String,List<WordMetadata>>
+ wordDictToFile(HashMap<String,List<WordMetadata>>): void
Query
+ queryID: String
+ query: ArrayList<String>
+ Query(String, String)
Resources
+ static final String postPath
+ static final String invertedFilePath
+ static final String filePath
+ static final String queryFilePath
+ static final String queryTDNFilePath
Corpus
- fileID_DocumentMap: HashMap<Integer, Document>
- invertedIndex: HashMap<String, List<WordMetadata>>
- documentLengths: HashMap<Integer,Double>
+ Corpus() // Constructor
- parseFileTxt(): HashMap<Integer,Document>
- initInvertedIndex(): void
- calculateDocumentLength(): void
+ getDocumentLength(): HashMap<Integer,Double>
+ getInverseDocumentFrequency(String): double
+ getDocuments(): ArrayList<Document>
+ getInvertedIndex(): HashMap<String, List<WordMetadata>>
+ readInvertedFile(File): HashMap<String, List<WordMetadata>>
+ getTRECFileID(int): String
WordMetadata
+ fileID: int
+ occurrence: int
+ wordPositions: ArrayList<Integer>
+ WordMetadata(int, int, ArrayList<Integer>)
SearchResult
+ fileID: int
+ score: double
+ SearchResult(int, double)
+ getFileID(): int
+ getScore(): double
Document
- int fileID // Actual order of how the file information are listed in file.txt
- int docLength //Document Word Count
- String docID // Actual Record ID for TREC Programs
- String path // Path to the file for TREC Programs
+ Document(int, int, String, String) // Constructor

Data Preprocessing

We choose to preprocess post1.txt and create a new inverted file based on the format below

4. invertedFile.txt

Field name:	**Term:**	**File-id/**	**Occurrence/**	**Word position, Word position,**	**:**	**File-id/**	**Occurrence/**	**Word position, Word position,**	**:**
Example:	Kosta:	16358/	2/	422,496	:	32398/	1/	34	:

In the data directory, we have 7 classes, which would be used by our numerous models in data processing.

The Corpus class denotes a corpus of documents, as its name suggests, and we have three fields in this class:

1. fileID_DocumentMap (HashMap), which maps fileID to TREC document.

2. invertedIndex (HashMap), which maps a term to a list of WordMetadata.

3. documentLengths (HashMap), which maps fileID Integer to Document Length.

The WordMetaData class is comprised of individual postings, each of which consists of a file

ID and payloads. In our implementation, the payloads include

1. TF (term frequency);

2. The position of every occurrence of the term in that document.

The Document class, which is a class to abstract documents, and we have 4 fields in this class:
1. fileID (Document ID/File ID - Actual order of how the file information are listed in file.txt)
2. docLength (Document Word Count)
3. docID (Actual Record ID for TREC Programs)
4. path (Path to the file for TREC Programs)

The Query class is used for abstracting every query.
The SearchResult class is used for abstracting the search results.
The Resource class is used to store the paths of files.
The PostTestProcessor class is used to preprocess the post1.txt and generate inverted file.

3. Retrieval Models

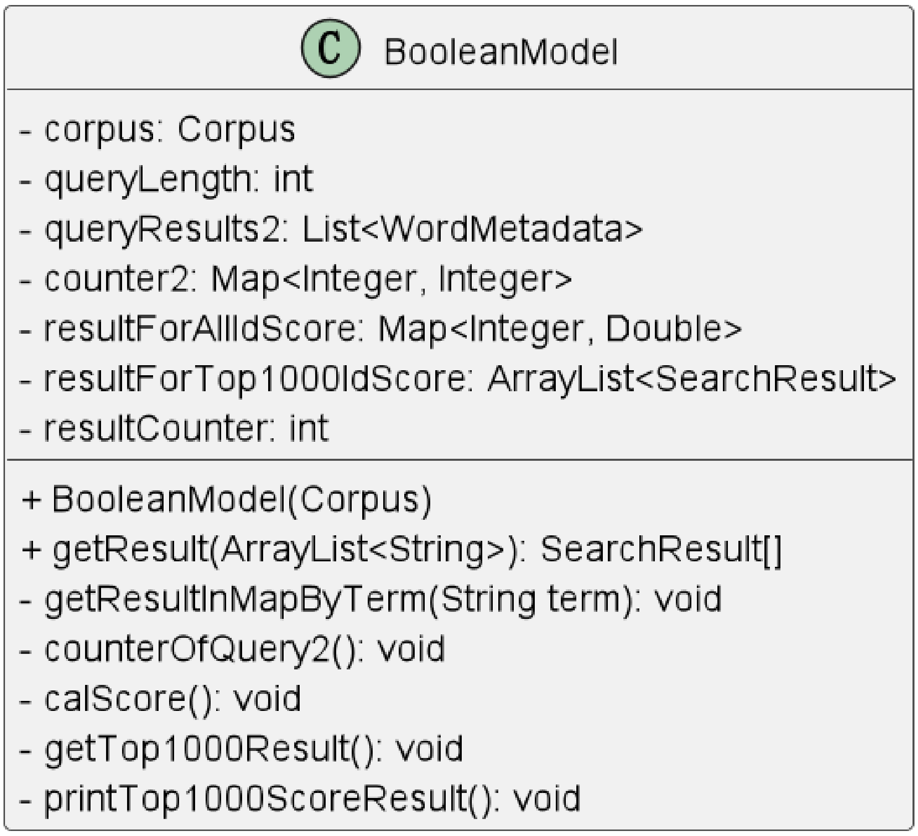
3.1 Boolean Model

The Boolean Model uses “OR”, “AND”, “NOT” to do searching. We find in the query files, however, that the short queries and long queries do not explicitly use “OR”, “AND”, “NOT” to do searching. In our Boolean Model, we only choose “OR” to do searching, for example input “Turkey Iraq water”, the Model will identify it as ““Turkey” OR “Iraq” OR “water””, that means if any one of these three words matches any documents, the Boolean Model see that document as a result and output that result.

Typically, a normal Boolean Model does not have similarity score function, but we managed to create a formula for Our Boolean Model to calculate the similarity score, the formula is:

Score = word of document matched by the query / The input word count of query

Here, we will demonstrate how to get top search results from Boolean Model.


C
BooleanModel
- corpus: Corpus
- queryLength: int
- queryResults2: List<WordMetadata>
- counter2: Map<Integer, Integer>
- resultForAllIdScore: Map<Integer, Double>
- resultForTop1000IdScore: ArrayList<SearchResult>
- resultCounter: int
+ BooleanModel(Corpus)
+ getResult(ArrayList<String>): SearchResult[]
- getResultInMapByTerm(String term): void
- counterOfQuery2(): void
- calScore(): void
- getTop1000Result(): void
- printTop1000ScoreResult(): void

Attributes of BooleanModel:

Attribute	**Description**
private Corpus corpus	This is corpus
private int queryLength	Used for storing the word count of query.
private List<WordMetadata> queryResults2	Used for storing the corresponding inverted index of a target term after running getResultInMapByTerm(String term).
private Map<Integer, Integer> counter2	Used to store the occurrences of the document ID appear in **queryResults2**.
private Map<Integer, Double> resultForAllIdScore	Used to store all similarity score between the query and the retrieved documents.
private ArrayList<SearchResult>	Used to store top 1000 results of **resultForAllIdScore**.

resultForTop1000IdScore	
private int resultCounter	Used to store the number of results.

Methods of BooleanModel:

getResult method details

public SearchResult[] getResult(ArrayList<String> querys)

This method can be considered as the main method of this class, it has the responsibility to run other methods to search documents, calculate similarity score and get top 1000 searched results.

It will return SearchResult[] **topResults**, which is top 1000 searched results.

getResultInMapByTerm method details

private void getResultInMapByTerm(String term)

In this method, it will get the corresponding inverted index of a single term and append the result into the List<WordMetadata> **queryResults2**.

counterOfQuery2 method details

private void counterOfQuery2()

In this method, it will count the occurrences of same **fileID** in Array **queryResults2** and then append the result into Map<Integer fileID, Integer occurrences> **counter2**.

calScore method details

private void calScore()

In this method, it will calculate the similarity score between the query and the retrieved documents.

<u>**The similarity score formula**</u>:

Let X = the occurrences of corresponding **fileID** in Array **queryResults2** (it is stored in **counter2**)

Let Y = The word count of **query** (if query is “Turkey Iraq water”, that mean there are 3 words)

Similarity score = X / Y

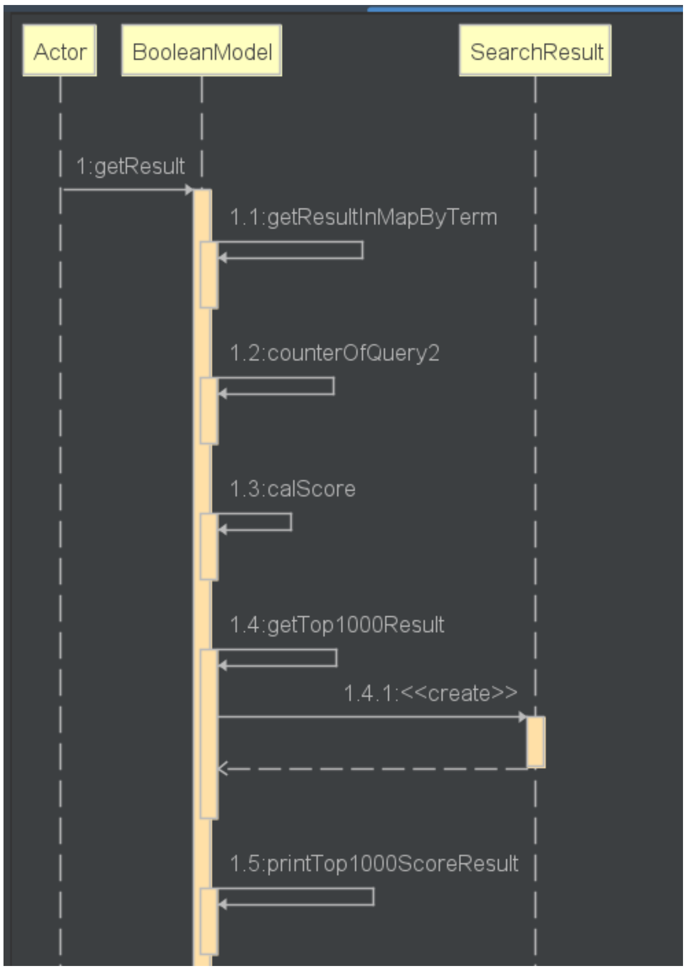
Finally, the similarity score of between the query and the retrieved documents will be append into Map<Integer documentID, Double score> **resultForAllIdScore**.

getTop1000Result method details

private void getTop1000Result()

This method is to load top 1000 results in **resultForAllIdScore** and store these top 1000 results into ArrayList<SearchResult> **resultForTop1000IdScore.**

This is the sequence diagram of the Boolean Model class.



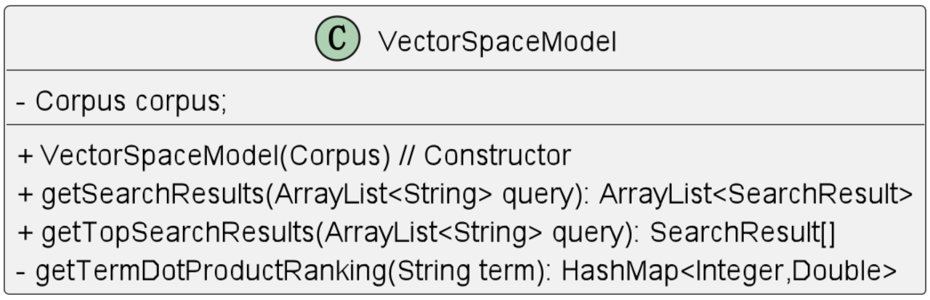
3.1.1 Ranking Techniques

In the normal Boolean Model, there is no ranking and similarity score function to do the ranking of each query. To achieve ranking, we use the first in first out, which does not consider the similarity score and only considers the order of the No. document.

For example, there is two matched document – “FT923-5358” and “LA122690-0032”. The Boolean Model will see the “FT923-5358” as rank 1 and “LA122690-0032” as rank 2.

3.2 Vector Space Model

Here, we will demonstrate how to get top search results from vector space model.



VectorSpaceModel
- Corpus corpus;
+ VectorSpaceModel(Corpus) // Constructor
+ getSearchResults(ArrayList<String> query): ArrayList<SearchResult>
+ getTopSearchResults(ArrayList<String> query): SearchResult[]
- getTermDotProductRanking(String term): HashMap<Integer,Double>

As for the logic of this class, simply put, we iterate all the query terms that have been processed with Porter Stemmer, and we incrementally compute cosine similarity score of each document as query words are processed one by one. We store the information in hash map, the file ID as the key and score as the value. Finally, we will get the top search result based on that.

When we initialize this class, the constructor will take the Corpus as parameter and assign it to the local field.

getSearchResults method details

public ArrayList<SearchResult> getSearchResults(ArrayList<String> query)

In this method, we firstly iterate all the terms in the query, and then it will invoke and get the HashMap from a method called getTermDotProductRanking in each iteration loop. According to the results of each iteration loop, the key/value will be merged as well in the HashMap<Integer,Double>. The dot product will be normalized and sorted, and the result will be returned as an array list.

getTopSearchResults method details

public SearchResult[] getTopSearchResults(ArrayList<String> query)

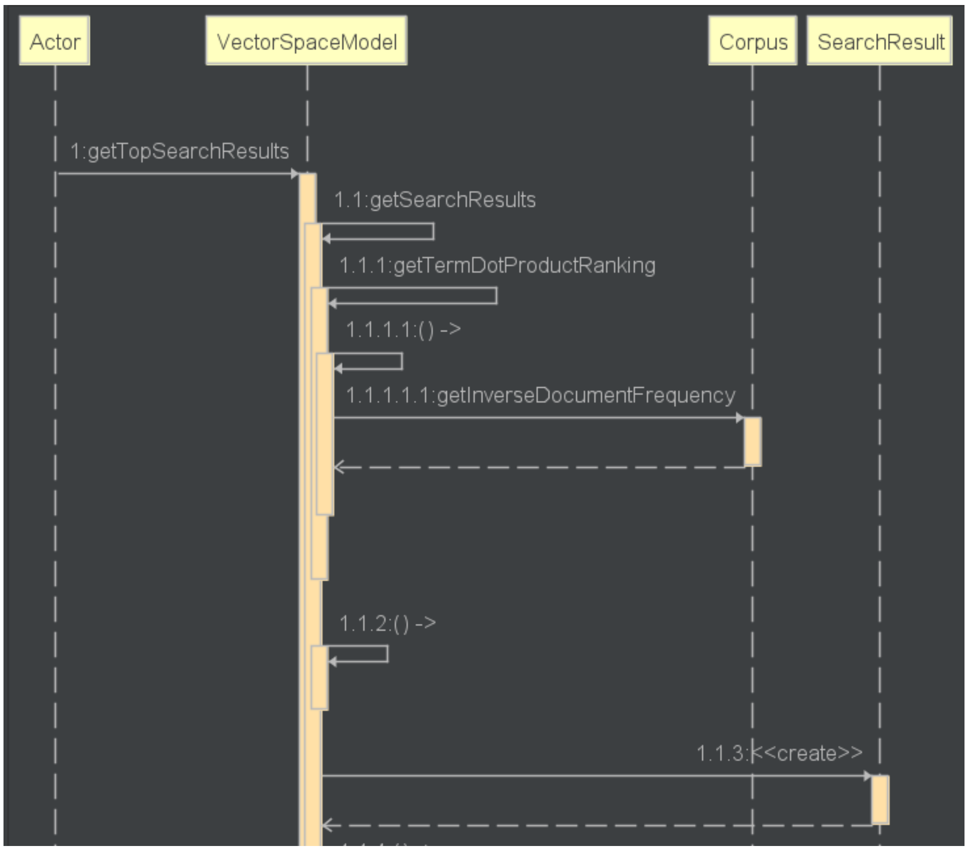
In this method, we will get the array list of SearchResult from the getSearchResults method, and we will compute the size of this array list. If it's larger than or equal to 1000, we will only get the first 1000 results and omit the rest one. We will accept all if the size is smaller than 1000. And we will return the top search results for creating TREC search result.

getTermDotProductRanking method details

private HashMap<Integer, Double> getTermDotProductRanking(String term)

In this method, it will check if the word can be found in the corpus. If there is no such term(key) in the inverted index, it will return an empty HashMap. If there is such a key, it will return a HashMap<Integer, Double>, the key of which is file ID, and the value is TF*IDF.

The sequence diagram of this class is as follows.



3.2.1 Ranking Techniques

The ranking of the documents is based on sorting the document's cosine similarity scores produced by the vector space model in descending order. So, the document with the highest score is 1st in rank and the lowest is the last in rank.

3.3 Fuzzy Boolean Model

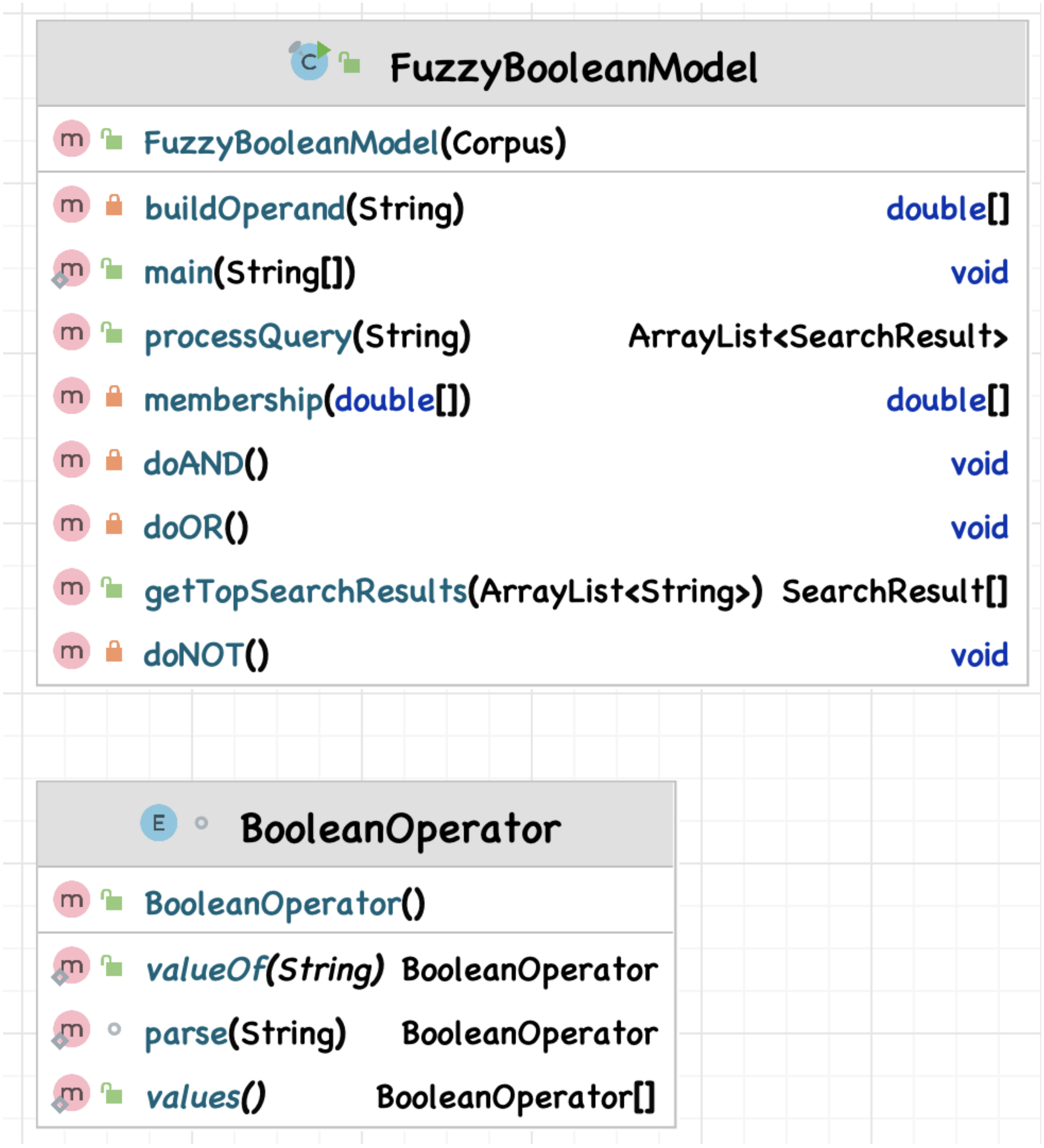
3.3.1 Introduction

The fuzzy Boolean model parses the query and calculates scores for each document based on the membership function. In our implementation, the membership function is defined as the term weight divided by the term weights in all documents. For example, for a query word "turkey" and 3 relevant documents. Our model will calculate the term weights of "turkey" in

all 3 documents as t1, t2 and t3. Then, the term weights after being processed by the membership function should be t1/(t1+t2+t3), t2/(t1+t2+t3), t3/(t1+t2+t3).
For the short and long query, this membership function has proved itself to work well.
Note that the ranking criteria is same as Boolean model, based on the score of each document.

3.3.2 Class Model

The graph below shows the UML diagram of the java class model. The BooleanOperator is an enumerator class including five major members, AND, OR, and NOT, left and right parenthesis (‘(‘and ‘)’).
The FuzzyBooleanModel is the processor class which parses the query string and outputs the top search results based on the algorithm.


FuzzyBooleanModel
FuzzyBooleanModel(Corpus)
buildOperand(String) double[]
main(String[]) void
processQuery(String) ArrayList<SearchResult>
membership(double[]) double[]
doAND() void
doOR() void
getTopSearchResults(ArrayList<String>) SearchResult[]
doNOT() void
BooleanOperator
BooleanOperator()
valueOf(String) BooleanOperator
parse(String) BooleanOperator
values() BooleanOperator[]

3.3.3 Algorithm

The algorithm is divided into two major components. The first component is parsing the input query string. The second component is calculating the scored based on the term and boolean operators.

For the query string parsing, the algorithm has two stacks initially. One for boolean operators, and the other for the operands. The operand is a double array storing the scores of each document for a query word. When the algorithm encounters a boolean operator, the operator is pushed into the operator stack for later calculation. When the algorithm encounters a word, it will build an operand, which is a double array representing the membershiped scores for each document.

The algorithm will perform a calculating when it encounters one of the following situations.

1. The top of the operator stack is “NOT”, and the operator stack in not empty.
2. The top of the operator stack is “)”（right parenthesis）. If this happens, then the algorithm will keep doing calculating until “(“（left parenthesis） is met.
3. The query string is ended. The algorithm will keep doing calculating until the operator stack is empty. If the query string is valid, then the operand stack will have one operand array only, which is supposed to be the overall scores.

The algorithm will throw out an exception if one of the following situation is met.

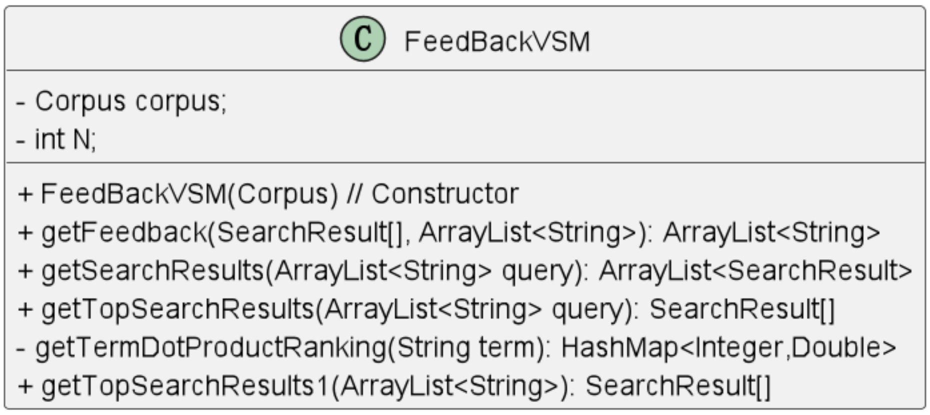
1. The right paranthesis is met but there is no left parenthesis in the operator stack.
2. There are consecutive AND, OR. (for example,w1AND OR w2 is invalid)
3. NOT is followed by AND, OR.

In the end, the algorithm will return a sorted array containing 1000 documents along with their scores to the caller.

3.4 Advance Model

3.4.1 Introduction

In the advance model part, we are inspired from pseudo relevance feedback. We will make use of automatic feedback to expand the query, which will make the query more specific. We first get the search result of the vector space model, then expand the query according to the search result.


FeedBackVSM
- Corpus corpus;
- int N;
+ FeedBackVSM(Corpus) // Constructor
+ getFeedback(SearchResult[], ArrayList<String>): ArrayList<String>
+ getSearchResults(ArrayList<String> query): ArrayList<SearchResult>
+ getTopSearchResults(ArrayList<String> query): SearchResult[]
- getTermDotProductRanking(String term): HashMap<Integer,Double>
+ getTopSearchResults1(ArrayList<String>): SearchResult[]

3.4.2 Techniques and Analysis

STEP 1. Feed the original query into the vector space model and get the top1000 relevant files. (We will use TF^2*IDF as the score.)

STEP 2. After obtaining the ranked searching results of the vector space model, we will calculate the document frequency of each term in the query, if the document frequency is larger than 0.1, then we will delete the term, since it implies the term is a very general one. This operation will help us to find more relevant documents.

STEP 3. We will choose the most relevant document in our first search, then we will calculate the TF-IDF score for each term, **then we will add the top3 highest score term** into the original query (we assume these terms are also related to the user's query).

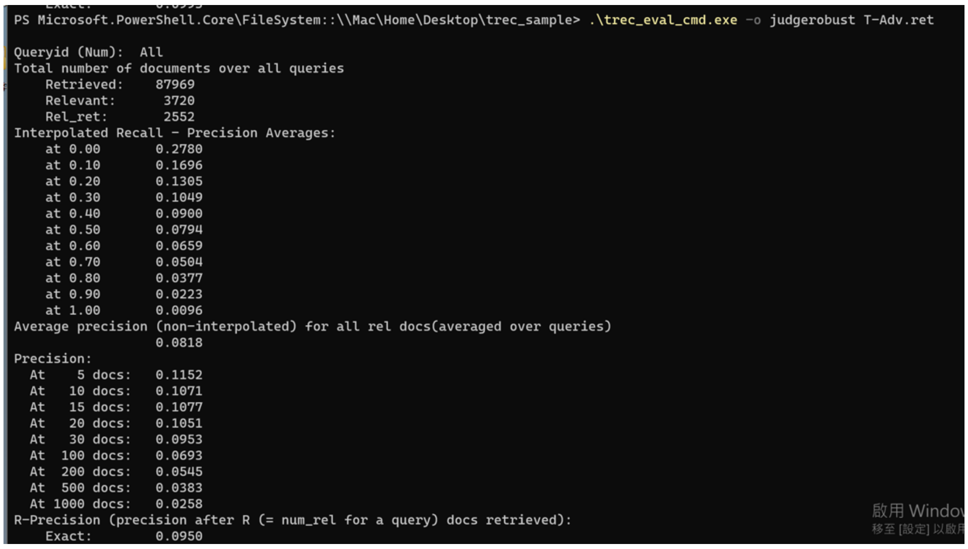
STEP 4. In the new modified query, we will delete the term with large document frequency as we did in step2. After that, we will use the modified query to search for the result.

The reason we choose to expand the original query is that:
There are only a small number of terms in a query, which means the given information is limited. When we want to find a relevant file, we may want to use more information to help us get the most relevant one. For example, when we want to use the query "Turkey Iraq water" to find files containing news of water issues between Turkey and Iraq, we can provide

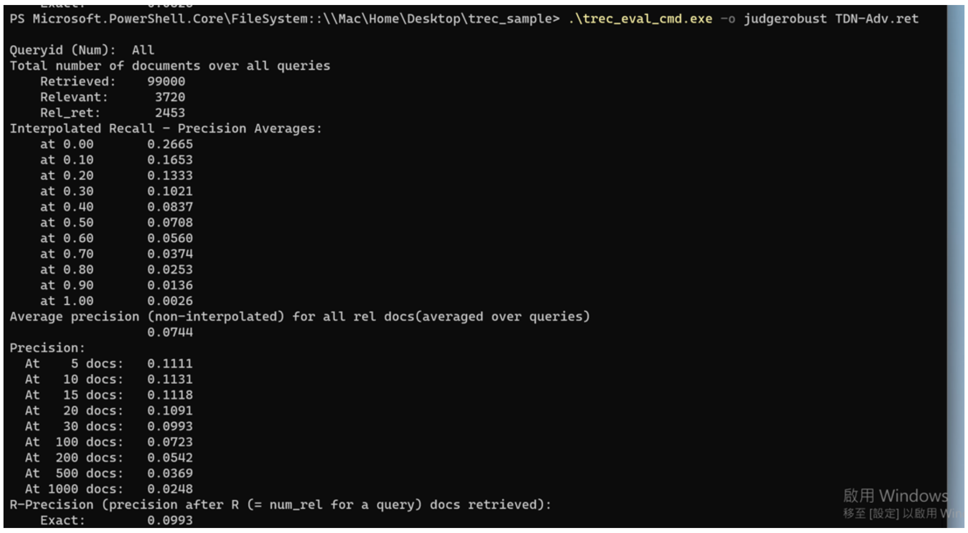
more information such as “Euphrates”, “hydropower station” to enhance the searching process. Even if we have preprocessed the query using porter stemmer, we cannot manually add new terms to the query. Then we use the top1 relevant files in the search result to provide more information for the search engine.

3.4.3 Effective Evidence

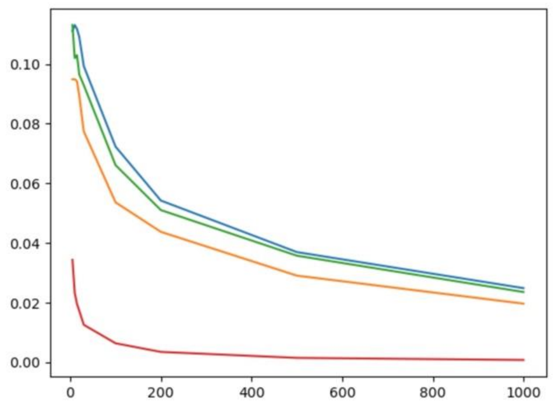
The result of our advanced model is better than vector space model, boolean model and fuzzy boolean model.



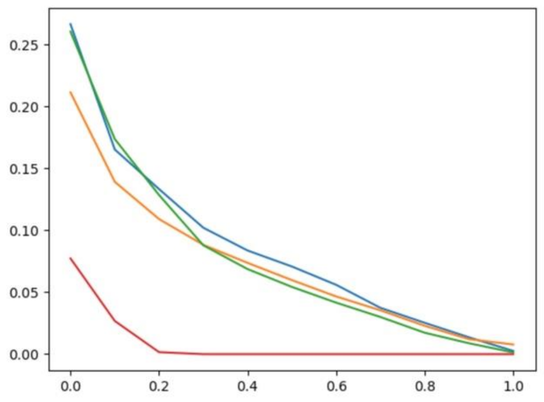
```
PS Microsoft.PowerShell.Core\FileSystem::\\Mac\Home\Desktop\trec_sample> .\trec_eval_cmd.exe -o judgerobust T-Adv.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    87969
    Relevant:      3720
    Rel_ret:       2552
Interpolated Recall - Precision Averages:
    at 0.00       0.2780
    at 0.10       0.1696
    at 0.20       0.1305
    at 0.30       0.1049
    at 0.40       0.0900
    at 0.50       0.0794
    at 0.60       0.0659
    at 0.70       0.0504
    at 0.80       0.0377
    at 0.90       0.0223
    at 1.00       0.0096
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0818
Precision:
  At    5 docs:   0.1152
  At   10 docs:   0.1071
  At   15 docs:   0.1077
  At   20 docs:   0.1051
  At   30 docs:   0.0953
  At  100 docs:   0.0693
  At  200 docs:   0.0545
  At  500 docs:   0.0383
  At 1000 docs:   0.0258
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0950
```



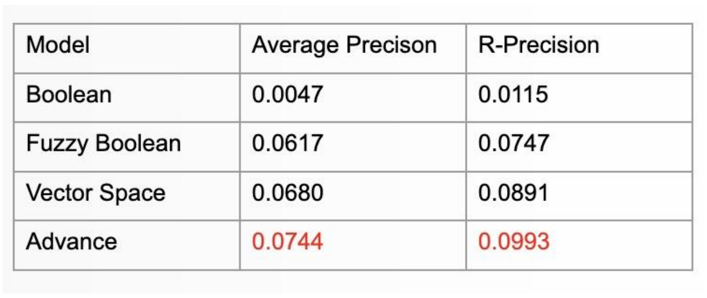
```
PS Microsoft.PowerShell.Core\FileSystem::\\Mac\Home\Desktop\trec_sample> .\trec_eval_cmd.exe -o judgerobust TDN-Adv.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    99000
    Relevant:      3720
    Rel_ret:       2453
Interpolated Recall - Precision Averages:
    at 0.00       0.2665
    at 0.10       0.1653
    at 0.20       0.1333
    at 0.30       0.1021
    at 0.40       0.0837
    at 0.50       0.0708
    at 0.60       0.0560
    at 0.70       0.0374
    at 0.80       0.0253
    at 0.90       0.0136
    at 1.00       0.0026
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0744
Precision:
  At    5 docs:   0.1111
  At   10 docs:   0.1131
  At   15 docs:   0.1118
  At   20 docs:   0.1091
  At   30 docs:   0.0993
  At  100 docs:   0.0723
  At  200 docs:   0.0542
  At  500 docs:   0.0369
  At 1000 docs:   0.0248
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0993
```



The above figure shows the interpolated recall-precision plot. (advance model: blue, vector space model: green, fuzzy boolean model: orange, boolean model: red)



The above figure shows the document-precision plot. (advance model: blue, vector space model: green, fuzzy boolean model: orange, boolean model: red)

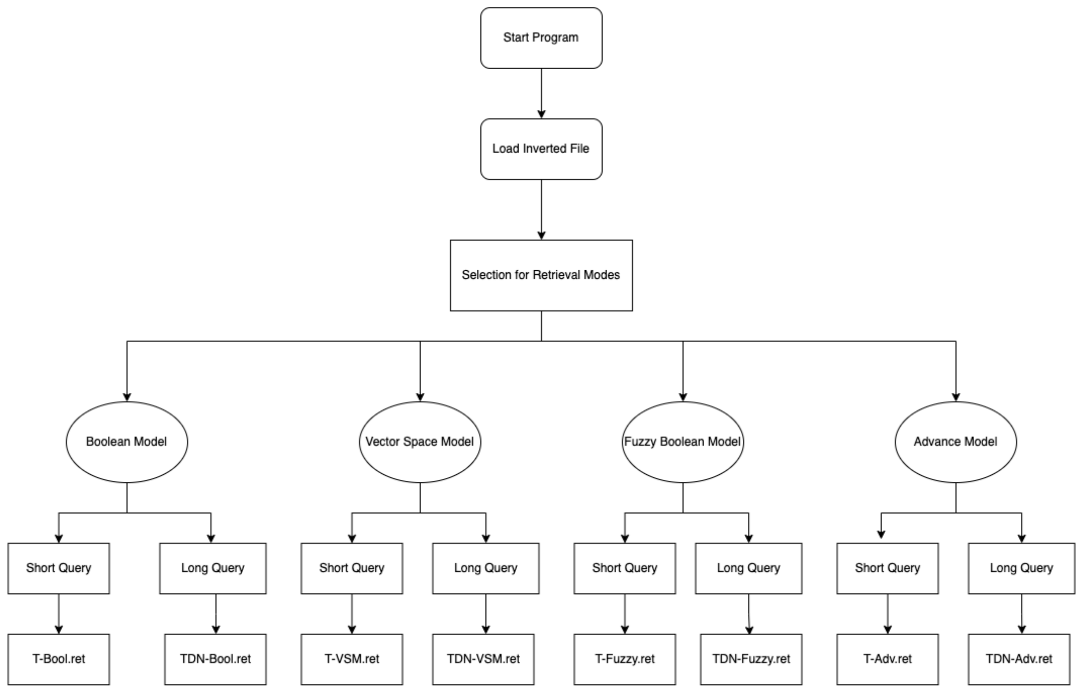


Model	Average Precison	R-Precision
Boolean	0.0047	0.0115
Fuzzy Boolean	0.0617	0.0747
Vector Space	0.0680	0.0891
Advance	0.0744	0.0993

4. Experiments

4.1 Experimental Design

The workflow of generating .ret file:



The experiment contains the following steps:

1. Run the program, the program will find the inverted file and read the file data. After that the program write the inverted file data into the Memory.

2. After the step 1, the CLI interface will display, and user can choose which model they want to use and use the short queries or long queries. After the choose, the program will be generating result .ret file and save in /SearchEngine folder. After generating, the CLI interface will back, user can choose another model and queries, no need do step 1 again.

3. The generating result is following this step:
    1. short queries and Boolean model
    2. long queries and Boolean model
    3. short queries and Vector Space model

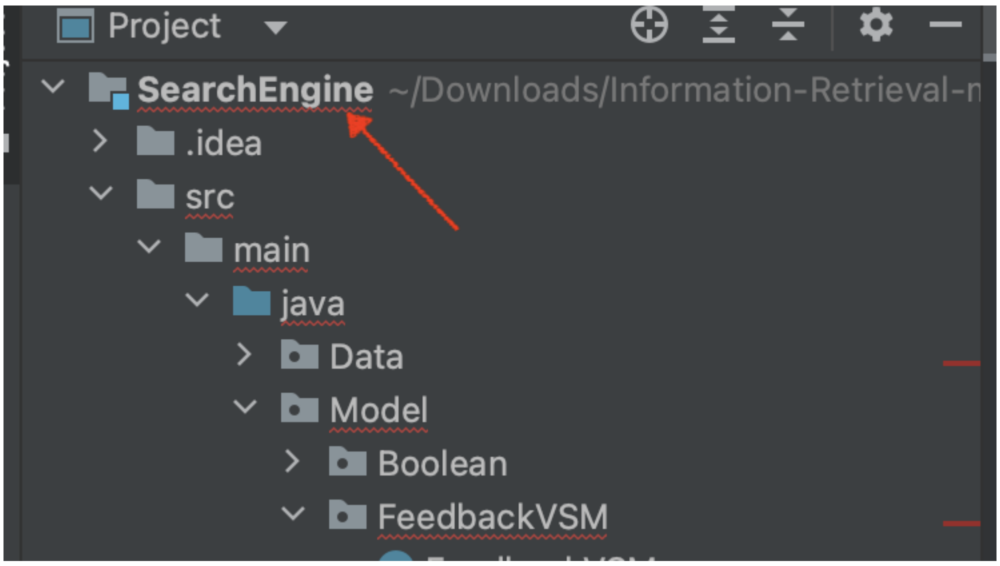
4. long queries and Vector Space model
5. short queries and Fuzzy Boolean model
6. Long queries and Fuzzy Boolean model
7. short queries and Advance model
8. Long queries and Advance model

4. The Experimental is end.

4.1.1 Retrieval Program Execution

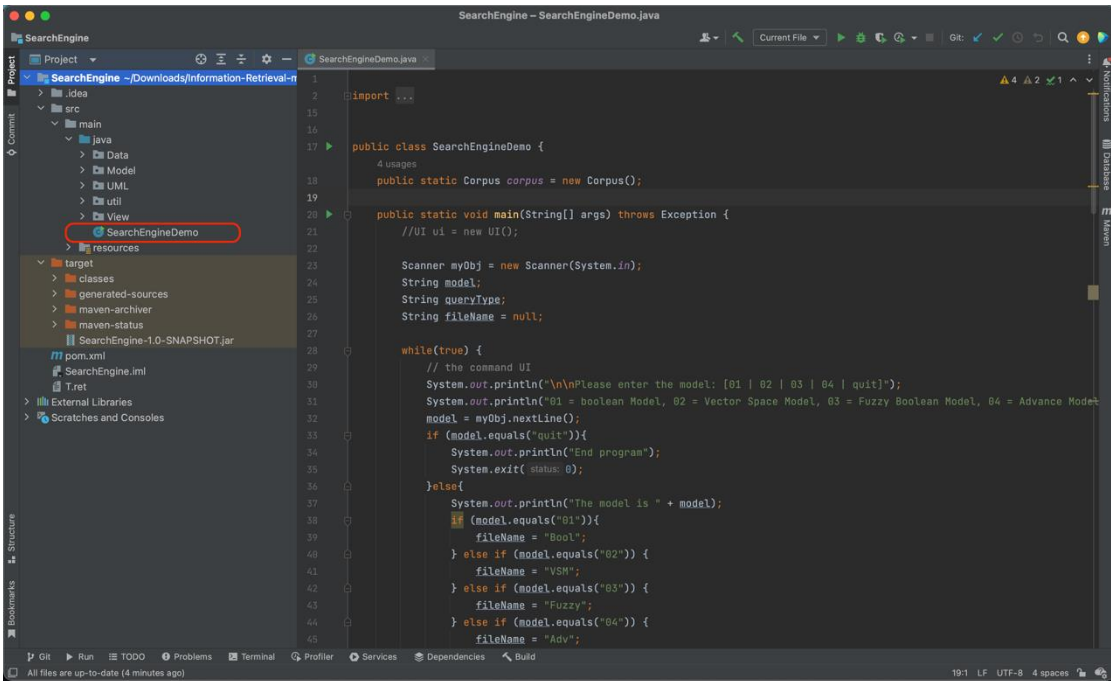
The experiment is conducted in specific Windows PC and used an IDE to run the program. Here is the workflow to execute the program and get the TREC search result.

1. Open **IntelliJ IDEA Ultimate** and open the directory **GroupCzip/Information-Retrieval-main/SearchEngine** (**Do not directly open the Information-Retrieval-main folder**)



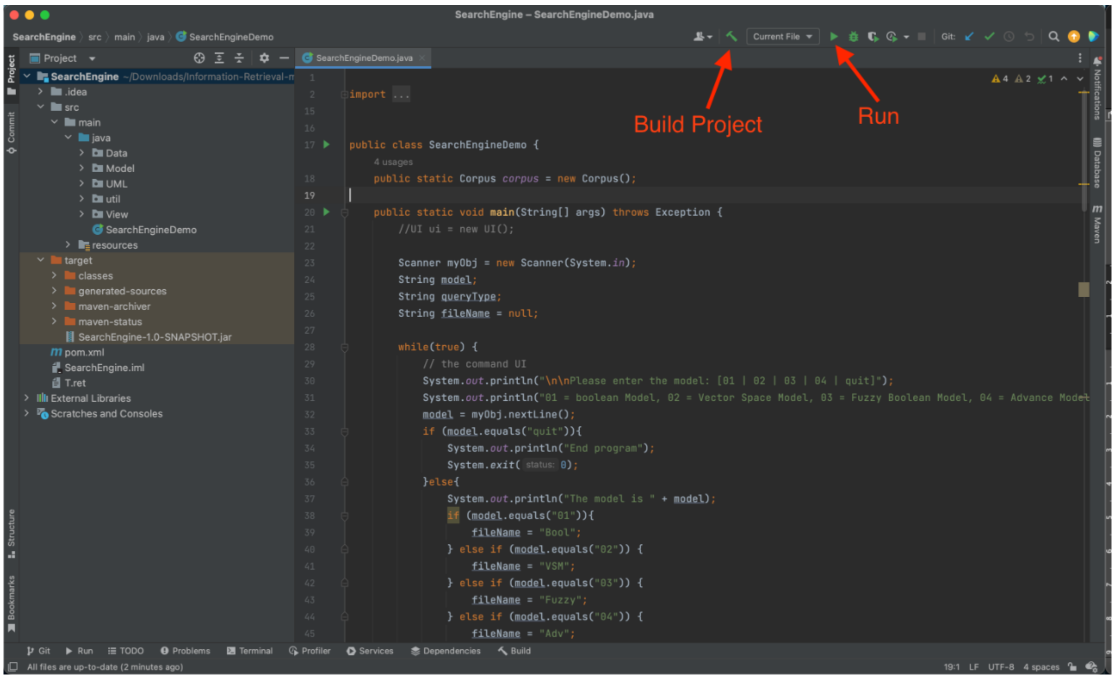
2. Open the **SearchEngineDemo.java** file, it locates on **SearchEngine/src/main/java/Search/**

SearchEngineDemo.java



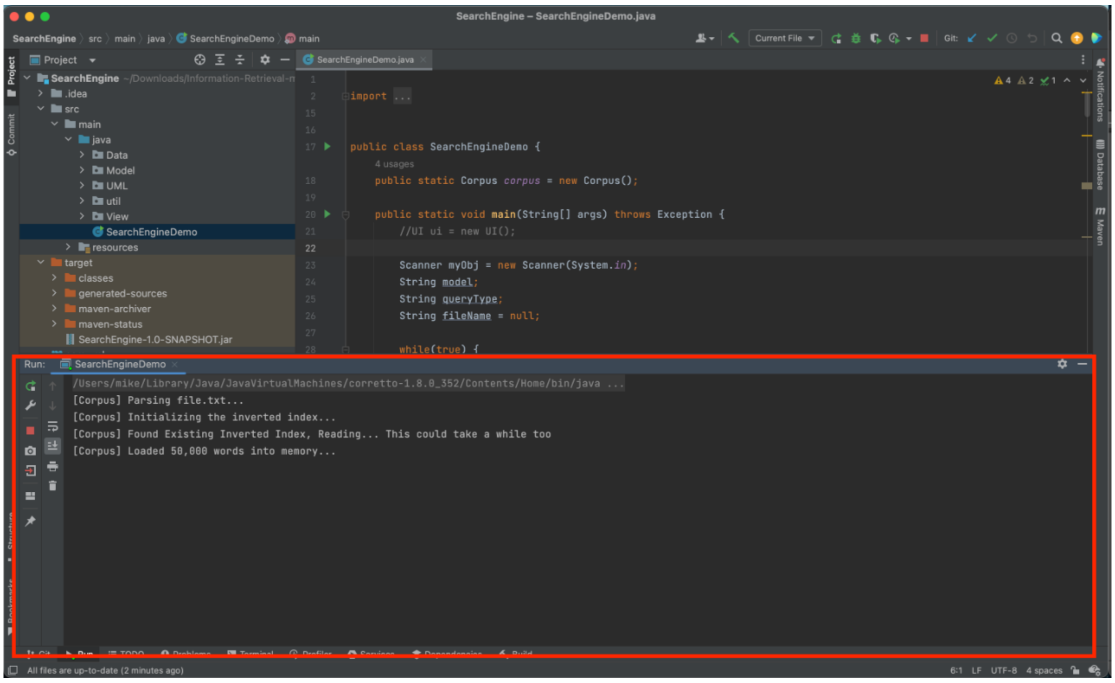
```java
import ...


public class SearchEngineDemo {
    4 usages
    public static Corpus corpus = new Corpus();

    public static void main(String[] args) throws Exception {
        //UI ui = new UI();

        Scanner myObj = new Scanner(System.in);
        String model;
        String queryType;
        String fileName = null;

        while(true) {
            // the command UI
            System.out.println("\n\nPlease enter the model: [01 | 02 | 03 | 04 | quit]");
            System.out.println("01 = boolean Model, 02 = Vector Space Model, 03 = Fuzzy Boolean Model, 04 = Advance Mode
            model = myObj.nextLine();
            if (model.equals("quit")){
                System.out.println("End program");
                System.exit( status: 0);
            }else{
                System.out.println("The model is " + model);
                if (model.equals("01")){
                    fileName = "Bool";
                } else if (model.equals("02")) {
                    fileName = "VSM";
                } else if (model.equals("03")) {
                    fileName = "Fuzzy";
                } else if (model.equals("04")) {
                    fileName = "Adv";
```

3. Click the **Build Project** on the function bar, after it finished building the project, click **Run** on the function bar.


SearchEngine – SearchEngineDemo.java
SearchEngine › src › main › java › SearchEngineDemo
Build Project
Run
Current File
Project
SearchEngineDemo.java
All files are up-to-date (2 minutes ago)
19:1 LF UTF-8 4 spaces

```java
import ...


public class SearchEngineDemo {
    4 usages
    public static Corpus corpus = new Corpus();

    public static void main(String[] args) throws Exception {
        //UI ui = new UI();

        Scanner myObj = new Scanner(System.in);
        String model;
        String queryType;
        String fileName = null;

        while(true) {
            // the command UI
            System.out.println("\n\nPlease enter the model: [01 | 02 | 03 | 04 | quit]");
            System.out.println("01 = boolean Model, 02 = Vector Space Model, 03 = Fuzzy Boolean Model, 04 = Advance Mode
            model = myObj.nextLine();
            if (model.equals("quit")){
                System.out.println("End program");
                System.exit( status: 0);
            }else{
                System.out.println("The model is " + model);
                if (model.equals("01")){
                    fileName = "Bool";
                } else if (model.equals("02")) {
                    fileName = "VSM";
                } else if (model.equals("03")) {
                    fileName = "Fuzzy";
                } else if (model.equals("04")) {
                    fileName = "Adv";
```

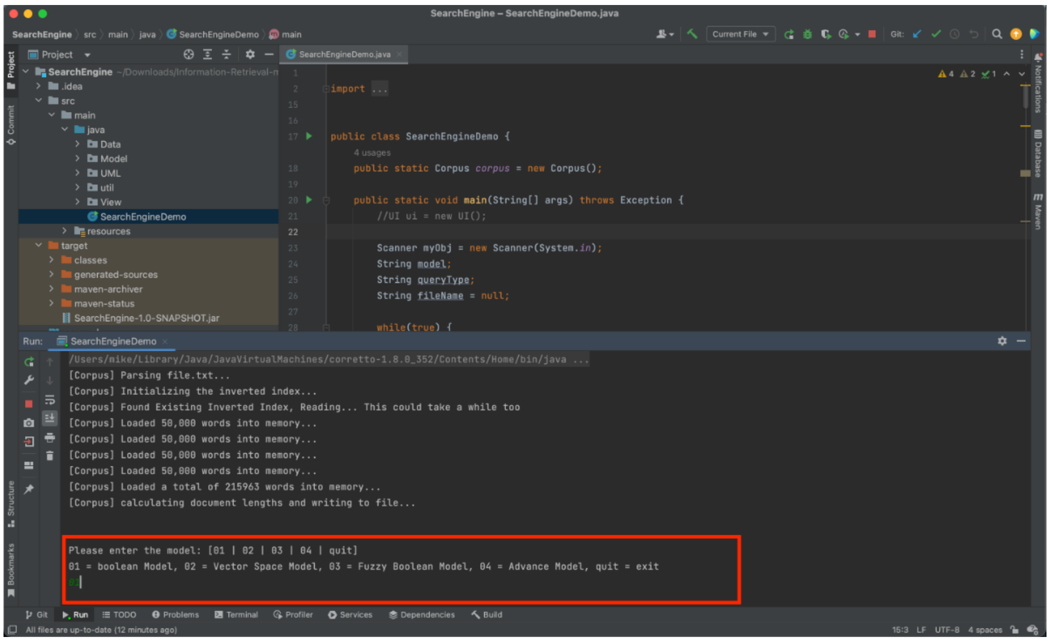
4. The program will read the inverted file to the memory; it takes a while to process.



```java
import ...


public class SearchEngineDemo {
    4 usages
    public static Corpus corpus = new Corpus();

    public static void main(String[] args) throws Exception {
        //UI ui = new UI();

        Scanner myObj = new Scanner(System.in);
        String model;
        String queryType;
        String fileName = null;

        while(true) {
```

```
/Users/mike/Library/Java/JavaVirtualMachines/corretto-1.8.0_352/Contents/Home/bin/java ...
[Corpus] Parsing file.txt...
[Corpus] Initializing the inverted index...
[Corpus] Found Existing Inverted Index, Reading... This could take a while too
[Corpus] Loaded 50,000 words into memory...
```

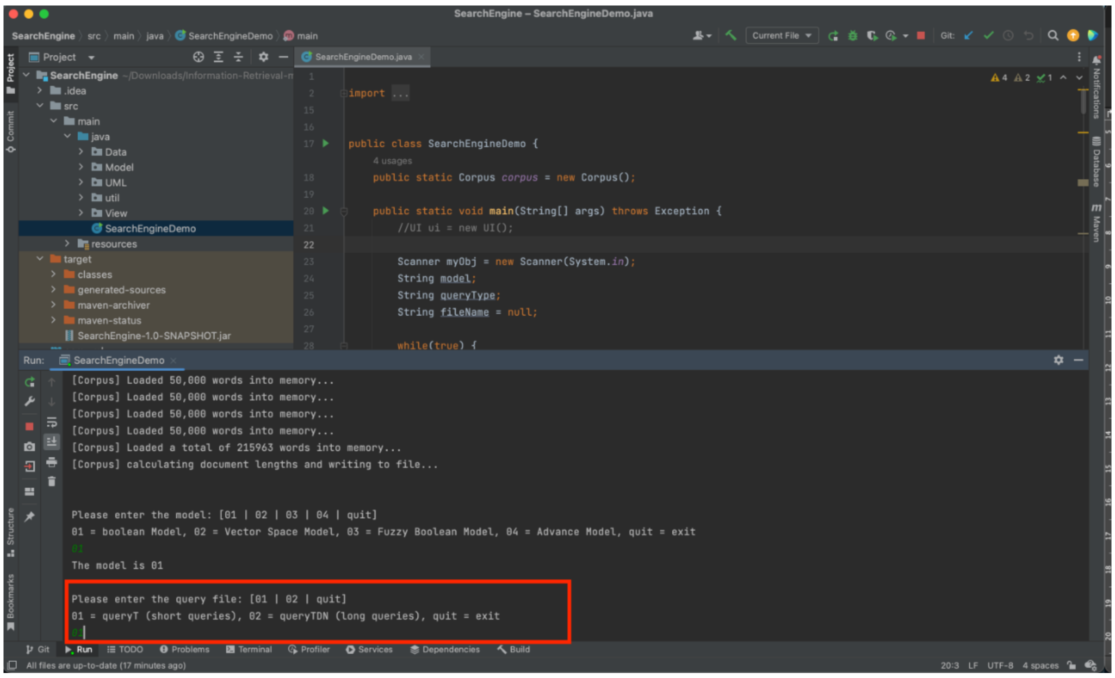
5. There are some options for user to choose the retrieval model that need to execute follow the instruction, for example if the result needs to be generated **from Boolean Model, type 01.**



```java
import ...


public class SearchEngineDemo {
    4 usages
    public static Corpus corpus = new Corpus();

    public static void main(String[] args) throws Exception {
        //UI ui = new UI();

        Scanner myObj = new Scanner(System.in);
        String model;
        String queryType;
        String fileName = null;

        while(true) {
```

```
/Users/mike/Library/Java/JavaVirtualMachines/corretto-1.8.0_352/Contents/Home/bin/java ...
[Corpus] Parsing file.txt...
[Corpus] Initializing the inverted index...
[Corpus] Found Existing Inverted Index, Reading... This could take a while too
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded a total of 215963 words into memory...
[Corpus] calculating document lengths and writing to file...

Please enter the model: [01 | 02 | 03 | 04 | quit]
01 = boolean Model, 02 = Vector Space Model, 03 = Fuzzy Boolean Model, 04 = Advance Model, quit = exit
01
```

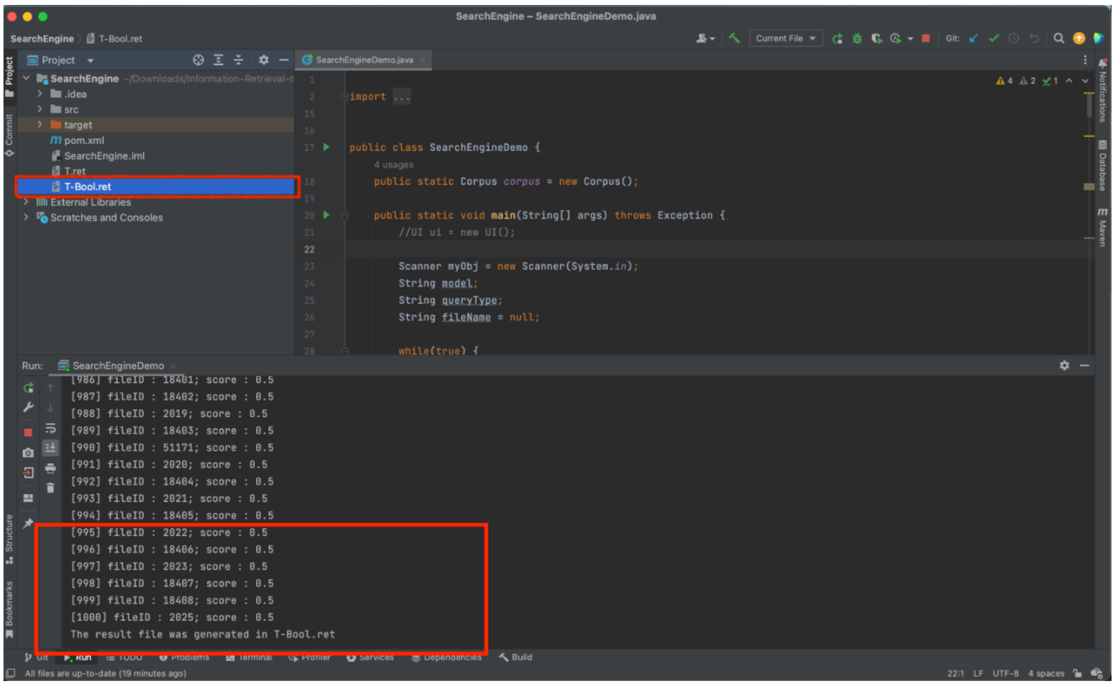
6. Choose the short query or long query file that needs to generate, for example if the result needs to be generated by short query which is **quertT, type 01 and enter.**


SearchEngine – SearchEngineDemo.java
SearchEngine 〉 src 〉 main 〉 java 〉 SearchEngineDemo 〉 main
Project
SearchEngine ~/Downloads/Information-Retrieval-
.idea
src
main
java
Data
Model
UML
util
View
SearchEngineDemo
resources
target
classes
generated-sources
maven-archiver
maven-status
SearchEngine-1.0-SNAPSHOT.jar
SearchEngineDemo.java
Run: SearchEngineDemo
Git Run TODO Problems Terminal Profiler Services Dependencies Build
All files are up-to-date (17 minutes ago)

```java
import ...


public class SearchEngineDemo {
    4 usages
    public static Corpus corpus = new Corpus();

    public static void main(String[] args) throws Exception {
        //UI ui = new UI();

        Scanner myObj = new Scanner(System.in);
        String model;
        String queryType;
        String fileName = null;

        while(true) {
```

```
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded 50,000 words into memory...
[Corpus] Loaded a total of 215963 words into memory...
[Corpus] calculating document lengths and writing to file...


Please enter the model: [01 | 02 | 03 | 04 | quit]
01 = boolean Model, 02 = Vector Space Model, 03 = Fuzzy Boolean Model, 04 = Advance Model, quit = exit
01
The model is 01

Please enter the query file: [01 | 02 | quit]
01 = queryT (short queries), 02 = queryTDN (long queries), quit = exit
01
```

7. The result has been generated and store in **SearchEngine folder**


SearchEngine – SearchEngineDemo.java
SearchEngine 〉 T-Bool.ret
Project
SearchEngine ~/Downloads/Information-Retrieval-
.idea
src
target
pom.xml
SearchEngine.iml
T.ret
T-Bool.ret
External Libraries
Scratches and Consoles
SearchEngineDemo.java
Run: SearchEngineDemo
Git Run TODO Problems Terminal Profiler Services Dependencies Build
All files are up-to-date (19 minutes ago)

```java
import ...


public class SearchEngineDemo {
    4 usages
    public static Corpus corpus = new Corpus();

    public static void main(String[] args) throws Exception {
        //UI ui = new UI();

        Scanner myObj = new Scanner(System.in);
        String model;
        String queryType;
        String fileName = null;

        while(true) {
```

```
[986] fileID : 18401; score : 0.5
[987] fileID : 18402; score : 0.5
[988] fileID : 2019; score : 0.5
[989] fileID : 18403; score : 0.5
[990] fileID : 51171; score : 0.5
[991] fileID : 2020; score : 0.5
[992] fileID : 18404; score : 0.5
[993] fileID : 2021; score : 0.5
[994] fileID : 18405; score : 0.5
[995] fileID : 2022; score : 0.5
[996] fileID : 18406; score : 0.5
[997] fileID : 2023; score : 0.5
[998] fileID : 18407; score : 0.5
[999] fileID : 18408; score : 0.5
[1000] fileID : 2025; score : 0.5
The result file was generated in T-Bool.ret
```

8. Repeat the above step to generate short query or long query result from different models.

4.2 Experiment Environment

We used the COMP Lab PC for the experiment, specification is below:

Item	Description
CPU	Inter(R) Core (TM) i7 – 10700 CPU @ 2.90GHz
Operation System	Windows 10
Memory	32GB
IDE	IntelliJ IDEA Ultimate Version: 2022.2.4 Build: 222.4459.24 23 November 2022
Java JDK	Java 1.8 or Java 17

4.3 Experimental Result

4.3.1 Time-efficiency

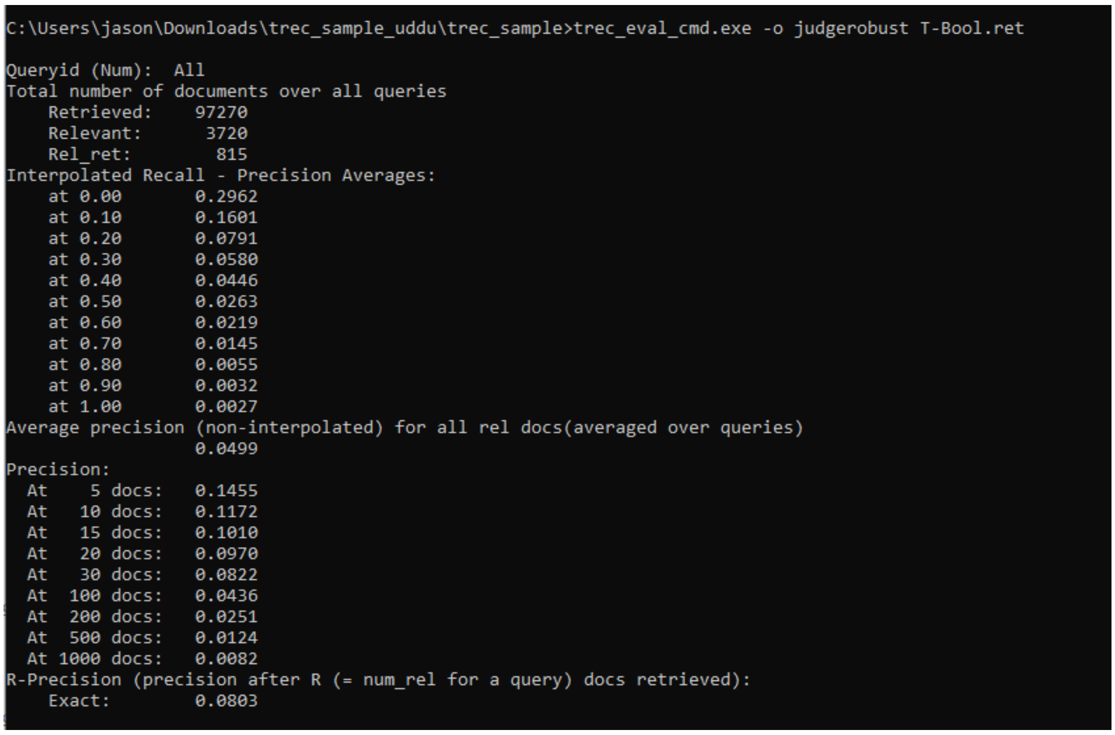
Task No.	Task	Time
1	Read the inverted file data write into the Memory	6 min 52 Second
2	Use short queries and Boolean model generating the result	3 Second
3	Use long queries and Boolean model generating the result	4 Second
4	Use short queries and Vector Space Model generating the result	3 Second
5	Use long queries and Vector Space Model generating the result	5 Second
6	Use short queries and Fuzzy Boolean model generating	4 Second

	the result	
7	Use long queries and Fuzzy Boolean model generating the result	5 Second
8	Use short queries and Advance model generating the result	1 min 6 Second
9	Use long queries and Advance model generating the result	1 min 8 Second

4.3.2 TREC Evaluation Result

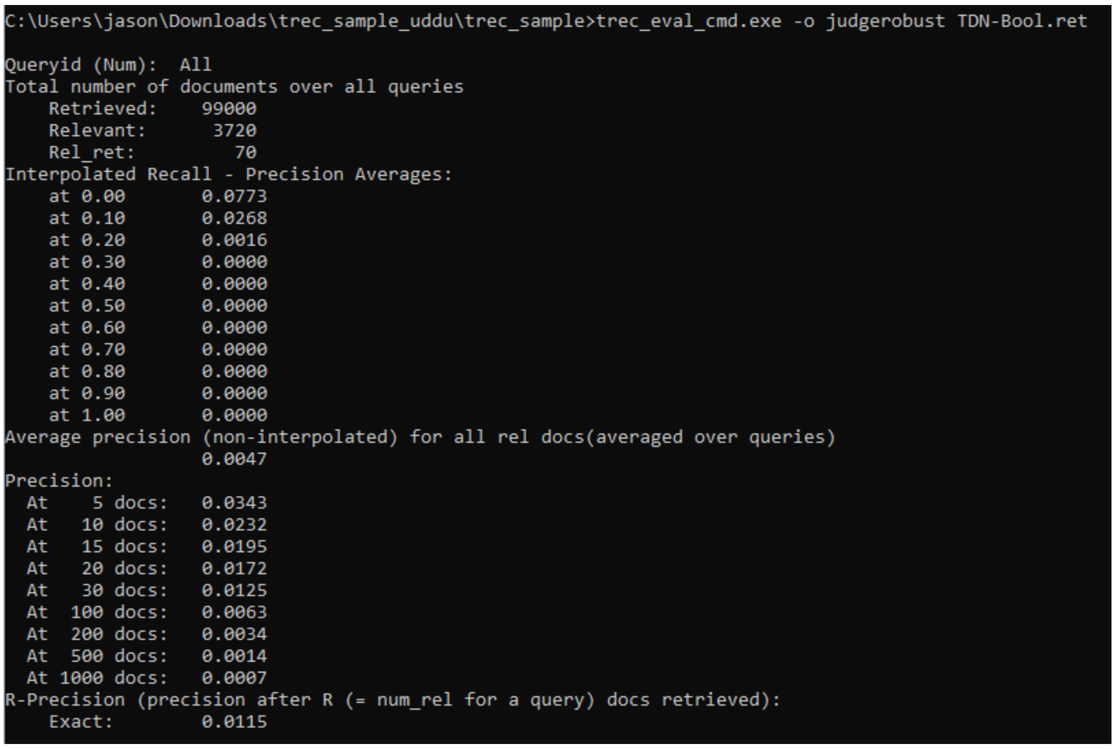
Put all the .ret file into the TREC evaluation program folder, on Windows CMD run command: **trec_eval_cmd.exe -o judgerbust XXXX.ret ,** XXXX which is the .ret file name, will produce the result.

4.3.2.1 Boolean Model with Short Query



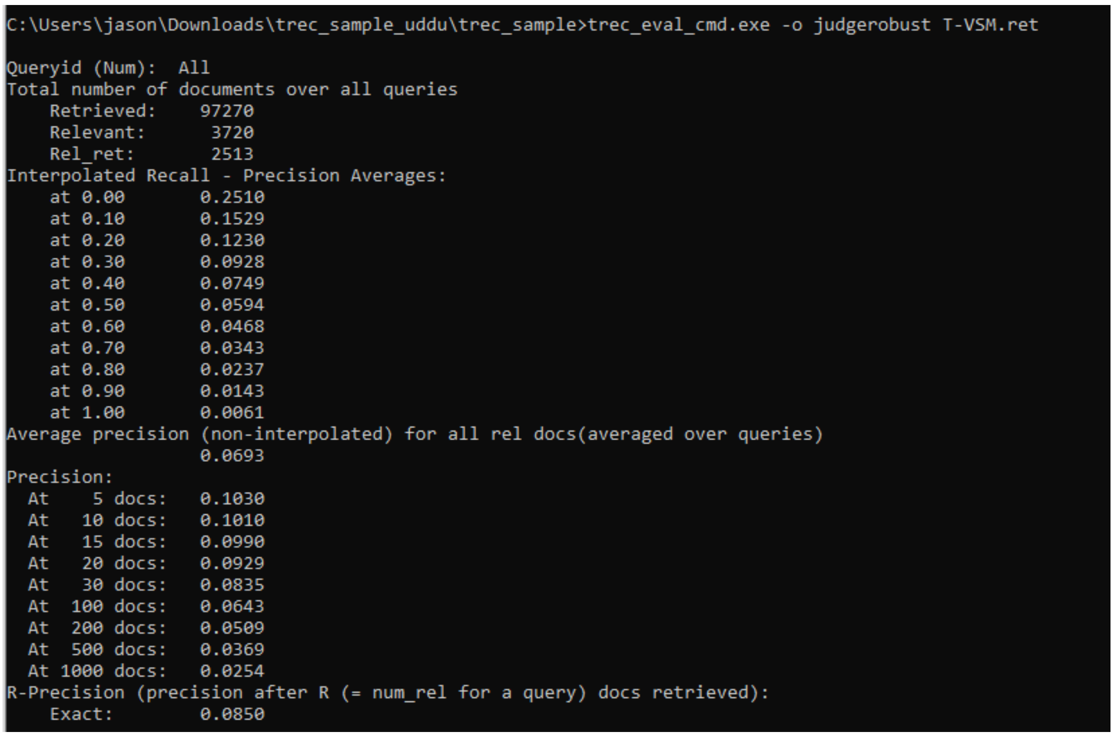
```
C:\Users\jason\Downloads\trec_sample_uddu\trec_sample>trec_eval_cmd.exe -o judgerobust T-Bool.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:     97270
    Relevant:       3720
    Rel_ret:         815
Interpolated Recall - Precision Averages:
    at 0.00       0.2962
    at 0.10       0.1601
    at 0.20       0.0791
    at 0.30       0.0580
    at 0.40       0.0446
    at 0.50       0.0263
    at 0.60       0.0219
    at 0.70       0.0145
    at 0.80       0.0055
    at 0.90       0.0032
    at 1.00       0.0027
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0499
Precision:
  At    5 docs:   0.1455
  At   10 docs:   0.1172
  At   15 docs:   0.1010
  At   20 docs:   0.0970
  At   30 docs:   0.0822
  At  100 docs:   0.0436
  At  200 docs:   0.0251
  At  500 docs:   0.0124
  At 1000 docs:   0.0082
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0803
```

4.3.2.2 Boolean Model with Long Query



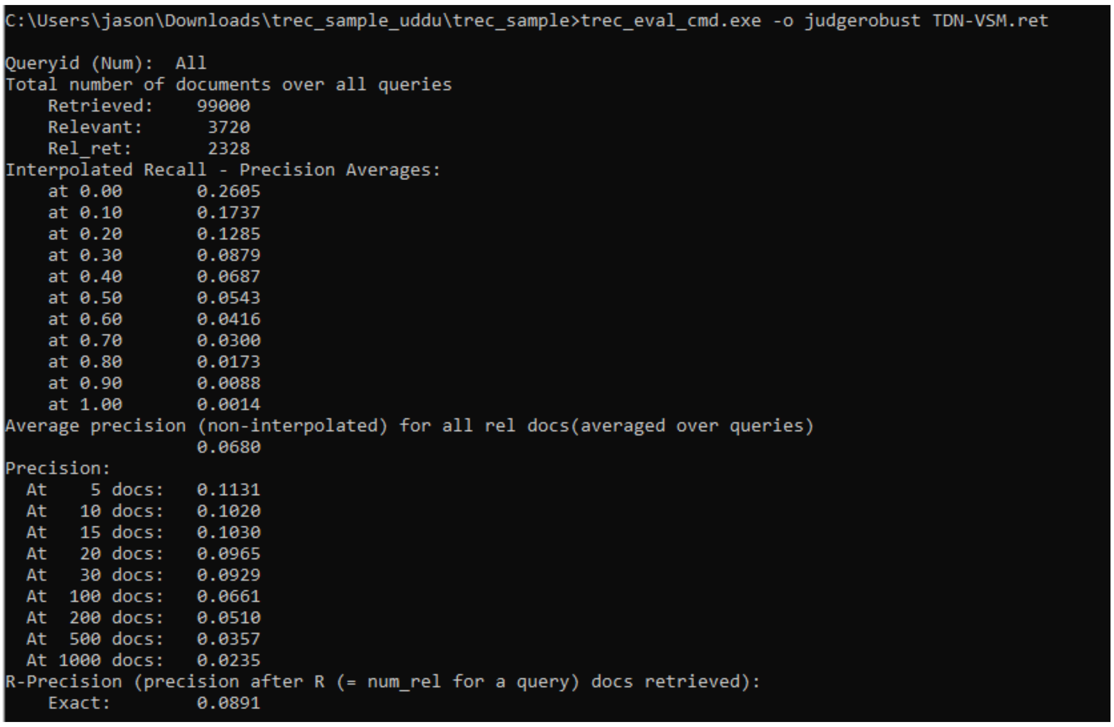
```
C:\Users\jason\Downloads\trec_sample_uddu\trec_sample>trec_eval_cmd.exe -o judgerobust TDN-Bool.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    99000
    Relevant:      3720
    Rel_ret:         70
Interpolated Recall - Precision Averages:
    at 0.00       0.0773
    at 0.10       0.0268
    at 0.20       0.0016
    at 0.30       0.0000
    at 0.40       0.0000
    at 0.50       0.0000
    at 0.60       0.0000
    at 0.70       0.0000
    at 0.80       0.0000
    at 0.90       0.0000
    at 1.00       0.0000
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0047
Precision:
  At    5 docs:   0.0343
  At   10 docs:   0.0232
  At   15 docs:   0.0195
  At   20 docs:   0.0172
  At   30 docs:   0.0125
  At  100 docs:   0.0063
  At  200 docs:   0.0034
  At  500 docs:   0.0014
  At 1000 docs:   0.0007
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0115
```

4.3.2.3 Vector Space Model with Short Query



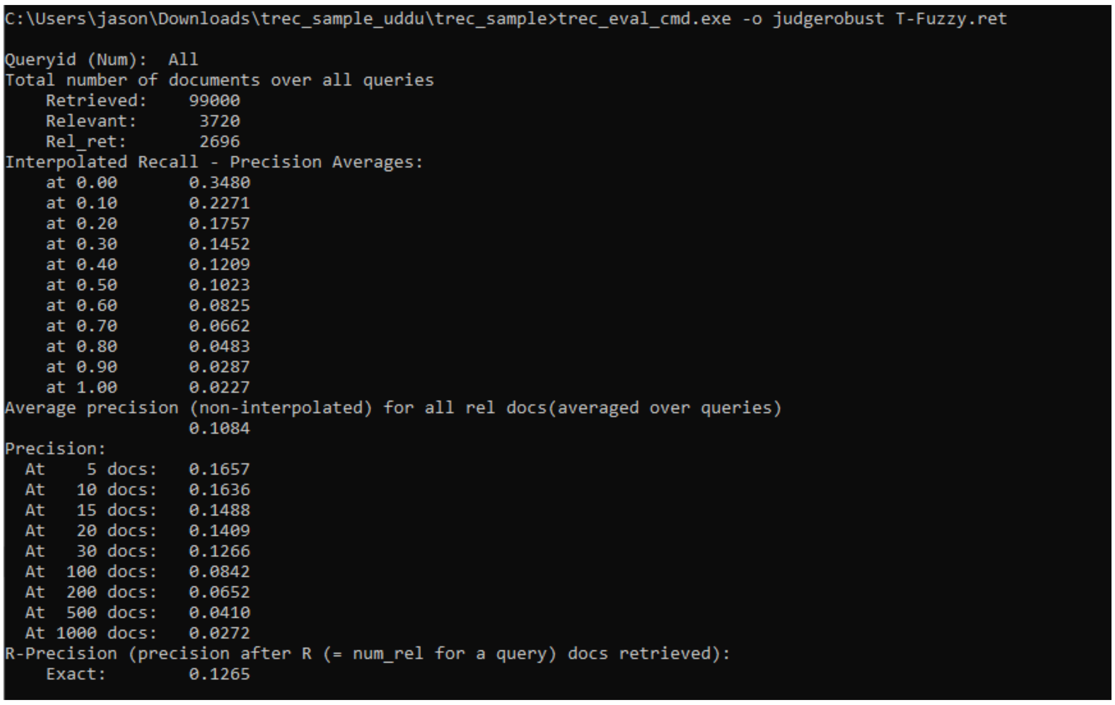
```
C:\Users\jason\Downloads\trec_sample_uddu\trec_sample>trec_eval_cmd.exe -o judgerobust T-VSM.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    97270
    Relevant:      3720
    Rel_ret:       2513
Interpolated Recall - Precision Averages:
    at 0.00       0.2510
    at 0.10       0.1529
    at 0.20       0.1230
    at 0.30       0.0928
    at 0.40       0.0749
    at 0.50       0.0594
    at 0.60       0.0468
    at 0.70       0.0343
    at 0.80       0.0237
    at 0.90       0.0143
    at 1.00       0.0061
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0693
Precision:
  At    5 docs:   0.1030
  At   10 docs:   0.1010
  At   15 docs:   0.0990
  At   20 docs:   0.0929
  At   30 docs:   0.0835
  At  100 docs:   0.0643
  At  200 docs:   0.0509
  At  500 docs:   0.0369
  At 1000 docs:   0.0254
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0850
```

4.3.2.4 Vector Space Model with Long Query



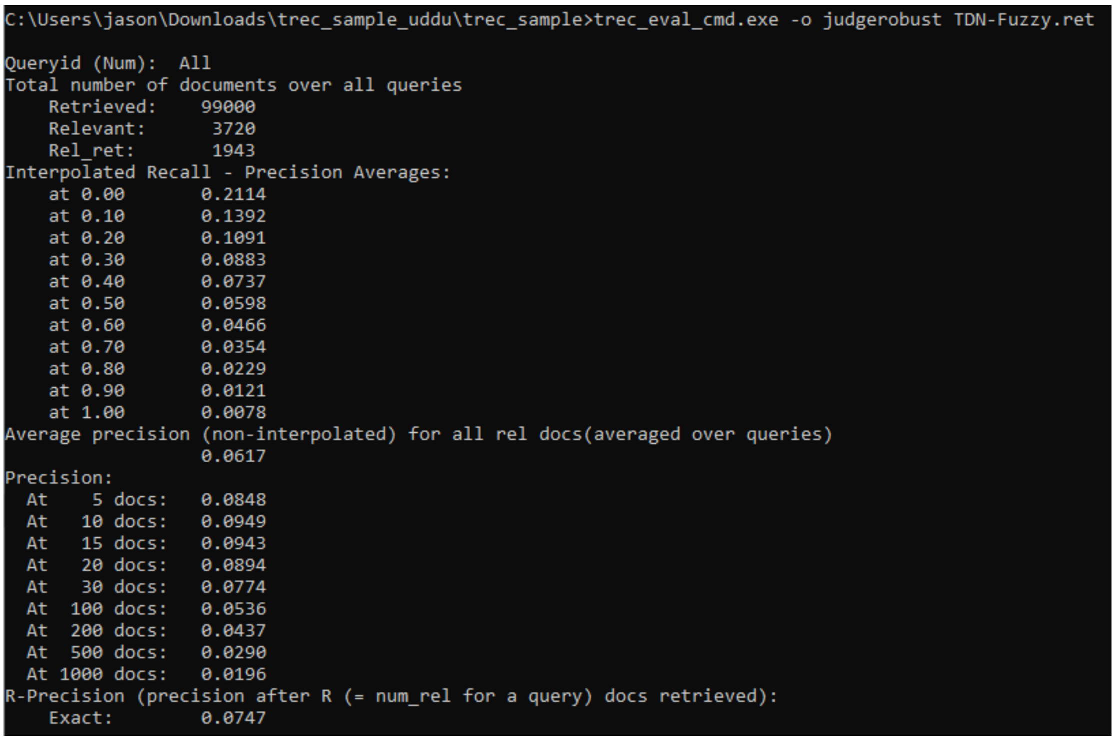
```
C:\Users\jason\Downloads\trec_sample_uddu\trec_sample>trec_eval_cmd.exe -o judgerobust TDN-VSM.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    99000
    Relevant:      3720
    Rel_ret:       2328
Interpolated Recall - Precision Averages:
    at 0.00       0.2605
    at 0.10       0.1737
    at 0.20       0.1285
    at 0.30       0.0879
    at 0.40       0.0687
    at 0.50       0.0543
    at 0.60       0.0416
    at 0.70       0.0300
    at 0.80       0.0173
    at 0.90       0.0088
    at 1.00       0.0014
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0680
Precision:
  At    5 docs:   0.1131
  At   10 docs:   0.1020
  At   15 docs:   0.1030
  At   20 docs:   0.0965
  At   30 docs:   0.0929
  At  100 docs:   0.0661
  At  200 docs:   0.0510
  At  500 docs:   0.0357
  At 1000 docs:   0.0235
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0891
```

4.3.2.5 Fuzzy Boolean Model with Short Query



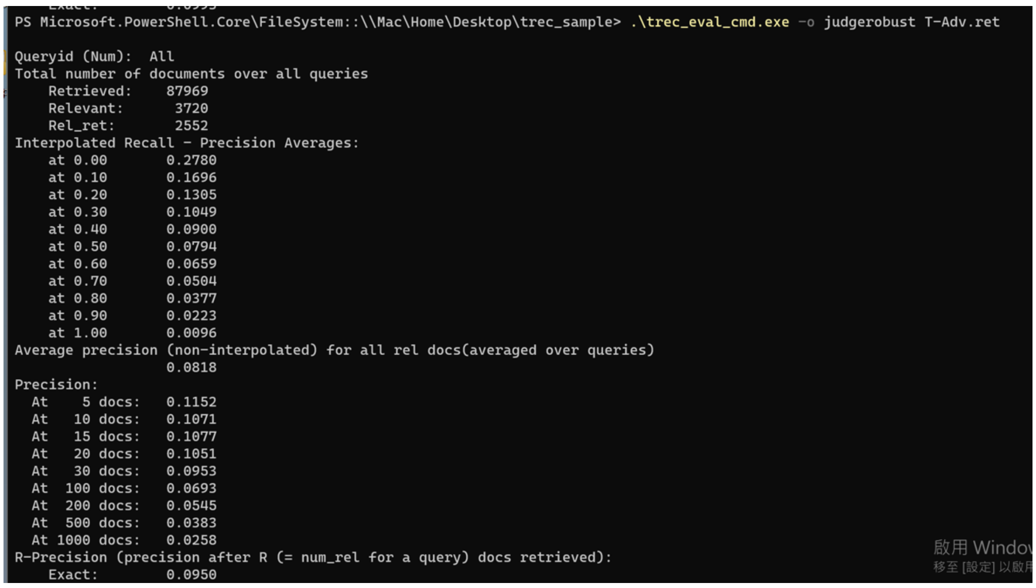
```
C:\Users\jason\Downloads\trec_sample_uddu\trec_sample>trec_eval_cmd.exe -o judgerobust T-Fuzzy.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    99000
    Relevant:      3720
    Rel_ret:       2696
Interpolated Recall - Precision Averages:
    at 0.00       0.3480
    at 0.10       0.2271
    at 0.20       0.1757
    at 0.30       0.1452
    at 0.40       0.1209
    at 0.50       0.1023
    at 0.60       0.0825
    at 0.70       0.0662
    at 0.80       0.0483
    at 0.90       0.0287
    at 1.00       0.0227
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.1084
Precision:
  At    5 docs:   0.1657
  At   10 docs:   0.1636
  At   15 docs:   0.1488
  At   20 docs:   0.1409
  At   30 docs:   0.1266
  At  100 docs:   0.0842
  At  200 docs:   0.0652
  At  500 docs:   0.0410
  At 1000 docs:   0.0272
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.1265
```

4.3.2.6 Fuzzy Boolean Model with Long Query



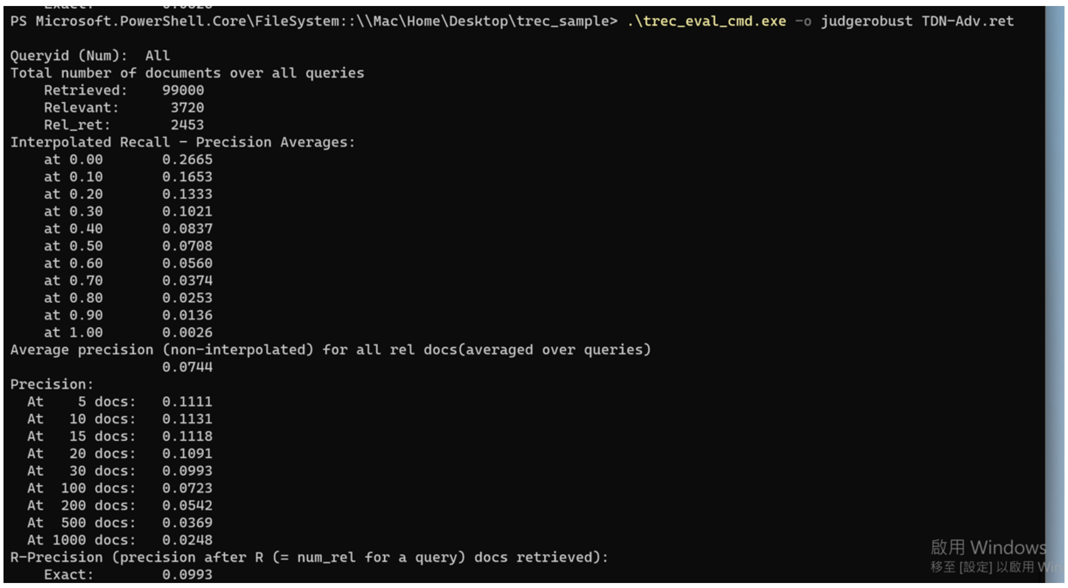
```
C:\Users\jason\Downloads\trec_sample_uddu\trec_sample>trec_eval_cmd.exe -o judgerobust TDN-Fuzzy.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:     99000
    Relevant:       3720
    Rel_ret:        1943
Interpolated Recall - Precision Averages:
    at 0.00       0.2114
    at 0.10       0.1392
    at 0.20       0.1091
    at 0.30       0.0883
    at 0.40       0.0737
    at 0.50       0.0598
    at 0.60       0.0466
    at 0.70       0.0354
    at 0.80       0.0229
    at 0.90       0.0121
    at 1.00       0.0078
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0617
Precision:
  At    5 docs:   0.0848
  At   10 docs:   0.0949
  At   15 docs:   0.0943
  At   20 docs:   0.0894
  At   30 docs:   0.0774
  At  100 docs:   0.0536
  At  200 docs:   0.0437
  At  500 docs:   0.0290
  At 1000 docs:   0.0196
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0747
```

4.3.2.7 Advance model with Short Query



```
PS Microsoft.PowerShell.Core\FileSystem::\\Mac\Home\Desktop\trec_sample> .\trec_eval_cmd.exe -o judgerobust T-Adv.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:     87969
    Relevant:       3720
    Rel_ret:        2552
Interpolated Recall - Precision Averages:
    at 0.00       0.2780
    at 0.10       0.1696
    at 0.20       0.1305
    at 0.30       0.1049
    at 0.40       0.0900
    at 0.50       0.0794
    at 0.60       0.0659
    at 0.70       0.0504
    at 0.80       0.0377
    at 0.90       0.0223
    at 1.00       0.0096
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0818
Precision:
  At    5 docs:   0.1152
  At   10 docs:   0.1071
  At   15 docs:   0.1077
  At   20 docs:   0.1051
  At   30 docs:   0.0953
  At  100 docs:   0.0693
  At  200 docs:   0.0545
  At  500 docs:   0.0383
  At 1000 docs:   0.0258
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0950
```

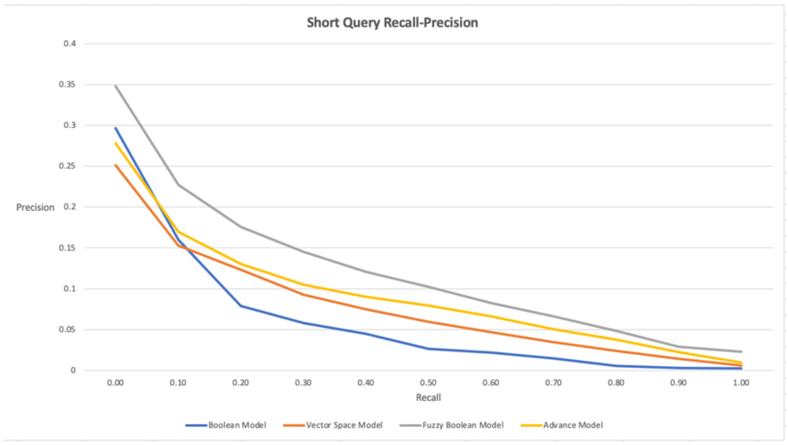
4.3.2.8 Advance model with Long Query



```
PS Microsoft.PowerShell.Core\FileSystem::\\Mac\Home\Desktop\trec_sample> .\trec_eval_cmd.exe -o judgerobust TDN-Adv.ret

Queryid (Num):  All
Total number of documents over all queries
    Retrieved:    99000
    Relevant:      3720
    Rel_ret:       2453
Interpolated Recall - Precision Averages:
    at 0.00       0.2665
    at 0.10       0.1653
    at 0.20       0.1333
    at 0.30       0.1021
    at 0.40       0.0837
    at 0.50       0.0708
    at 0.60       0.0560
    at 0.70       0.0374
    at 0.80       0.0253
    at 0.90       0.0136
    at 1.00       0.0026
Average precision (non-interpolated) for all rel docs(averaged over queries)
                  0.0744
Precision:
  At    5 docs:   0.1111
  At   10 docs:   0.1131
  At   15 docs:   0.1118
  At   20 docs:   0.1091
  At   30 docs:   0.0993
  At  100 docs:   0.0723
  At  200 docs:   0.0542
  At  500 docs:   0.0369
  At 1000 docs:   0.0248
R-Precision (precision after R (= num_rel for a query) docs retrieved):
    Exact:        0.0993
```

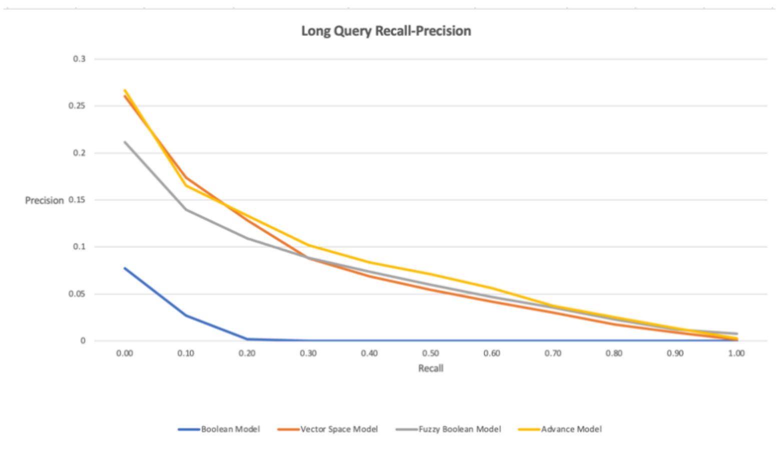
4.4 Analysis

- **Short Query Recall-Precision Analysis**



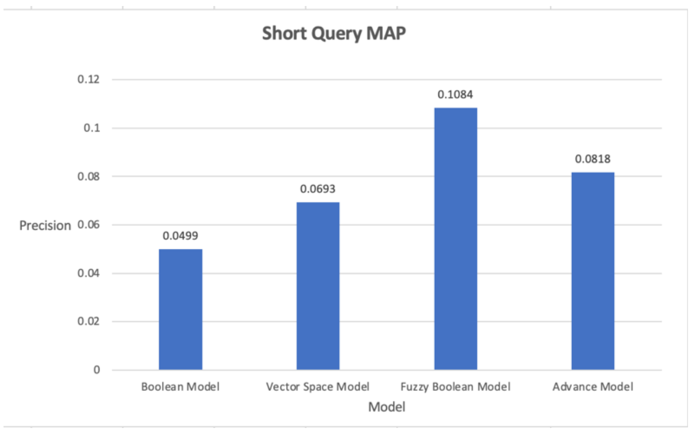
In short query recall-precision, fuzzy Boolean Model has the highest performance compared to the other models.

- **Long Query Recall-Precision Analysis**



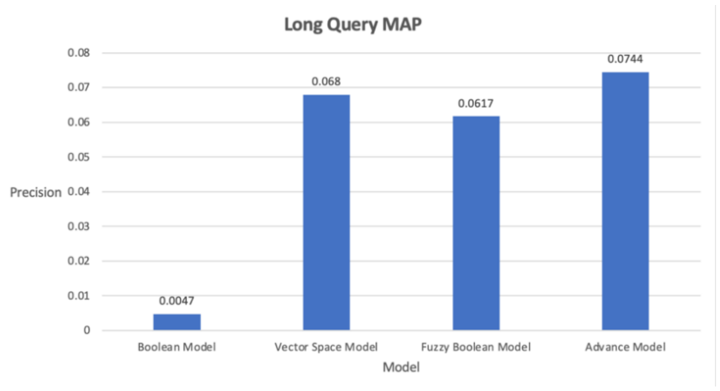
In long query recall-precision, Advance Model and Vector Space get the similar high performance compared to the other models.

- **Short Query MAP Analysis**



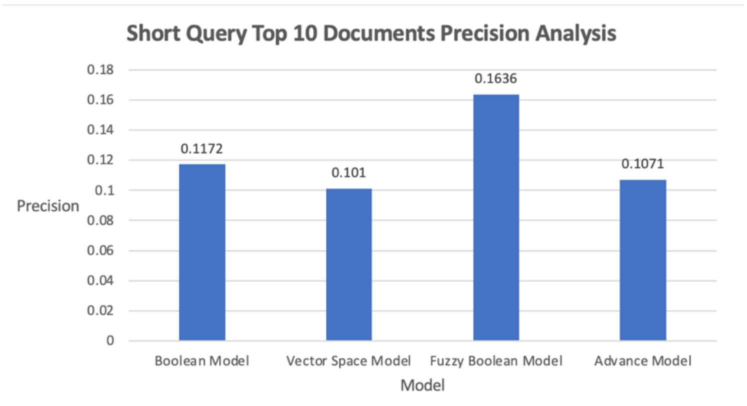
In short query MAP, Fuzzy Boolean Model has the highest performance compared to the other models.

- **Long Query MAP Analysis**



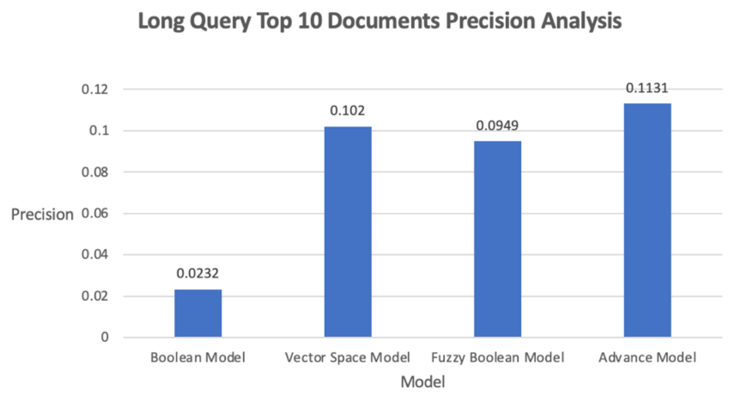
In long query MAP, Advance Boolean Model has the highest performance compared to the other models.

- **Short Query Top 10 Documents Precision Analysis**



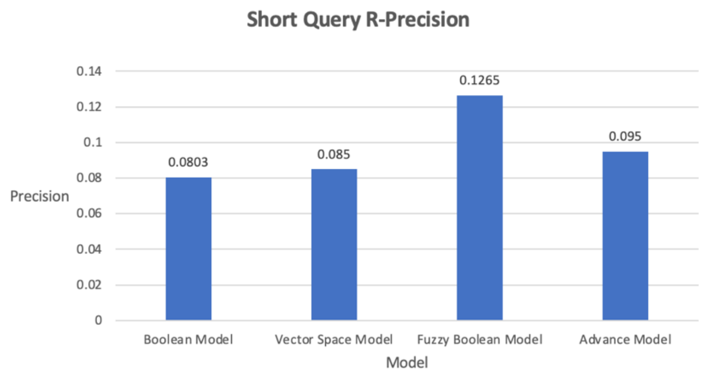
In short query top 10 documents precision, Fuzzy Boolean Model has the highest performance compared to the other models.

- **Long Query Top 10 Documents Precision Analysis**



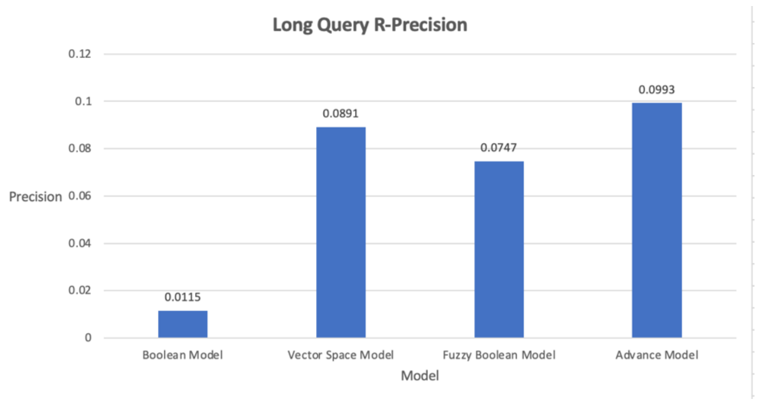
In long query top 10 documents precision, Advance Model has the highest performance compared to the other models.

- **Short Query R-Precision Analysis**



In short query r-precision, Fuzzy Boolean Model has the highest performance compared to the other models.

- **Long Query R-Precision Analysis**



In long query r-precision, Advance Model has the highest performance compared to the other models.

5. Conclusion

The report has explained the logic and implementation of the models which are Boolean, Vector Space, Fuzzy Boolean and Advance Model. As for the experiment part, we explained the method we used for experimenting as well as the result.