Project Two

News Searcher – Version Draft

CSE 435 / 535 – Information Retrieval (Fall 2014)

Table of Contents

Version History

S.no	Version #	Date	Author	Comments
1.	1.0	28 Sep 14	Nikhil L	Initial version

1. Project Description

This project aims to build a fully functional News search engine with the following goals:

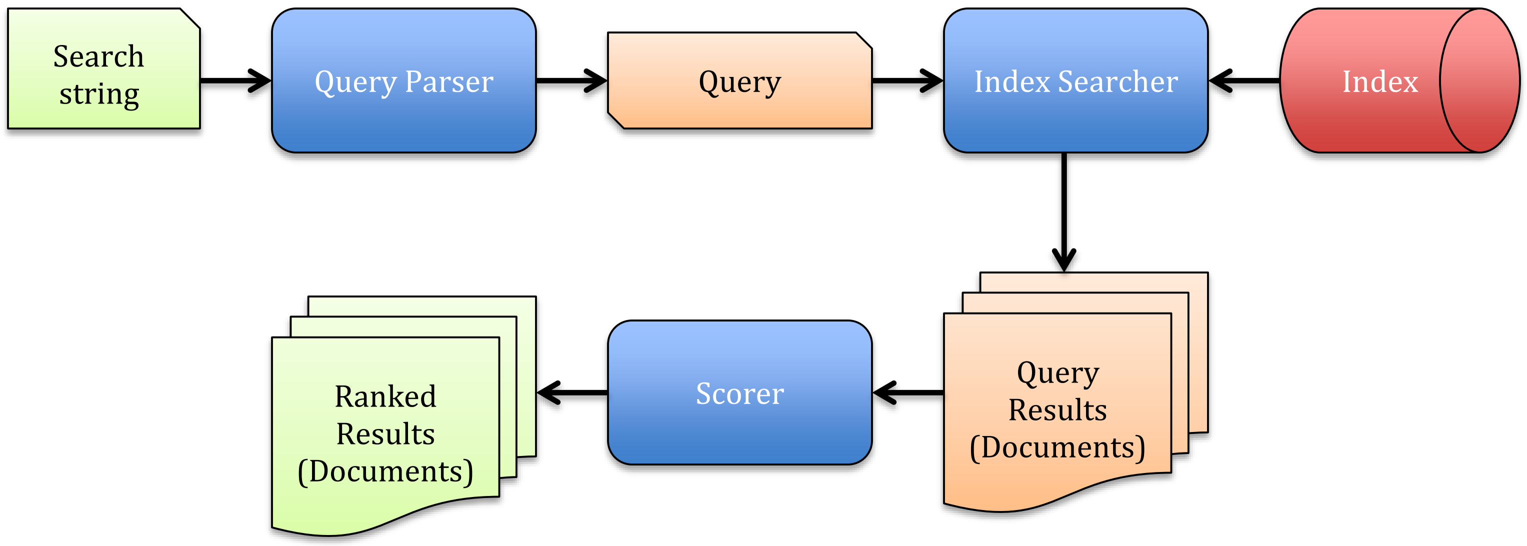
- Implement a query parser that can parse free text into structured query terms
- Implement a querying mechanism that can ingest query terms and find matching documents
- Implement two popular scoring mechanisms to rank retrieved documents
- Implement one or more methods to improve search engine relevancy scores.

As before, you will be implementing this project purely in Java and some starter code would be provided with detailed comments, documentation etc.

All deliverables are due by **19th October 2014, 23:59 EST/EDT**.

2. System components

The system has three main components: a query parser, an index searcher and a scorer. An overall system diagram is shown below:



The following subsections describe the function of each subcomponent in greater detail.

2.1. Query Parser

This component is responsible for converting a given raw string query into a Query representation. A Query is nothing but an ordered collection of Clauses that are connected by Boolean operators. Refer to the grammar below:

```
Query ::= ( Clause )*
Clause ::= [<Index> ":"] ( <Term> | "(" Query ")" )
Index ::= Term | Category | Author | Place
Term :: = Query term | " Query phrase "
Query phrase ::= ( Query term ) *
Operators ::= AND | OR | NOT
```

Thus, all that the Query Parser does is parse the given user string and pass on the Query object to downstream classes for them to process.

2.2. Index Searcher

As the name suggests, this component is responsible for consuming the given Query object and use it to search through the given Index to retrieve documents. This component would be responsible thus, to perform the following operations:

- Determine the order of execution of different Clauses for a given Query
- Evaluate each Clause by retrieving postings through the corresponding indices and intersecting them as necessary
- Generate a final list of retrieved documents for the corresponding Query by evaluating all Clauses.

Observe that an accurate Query Parser implementation is crucial for this component to perform correctly. The above list is meant to illustrate the intended functionality of this component and is not binding to the underlying implementation. Further note that the "documents" as mentioned above need not be the Document class encountered during the Indexing phase. At the very least, it is elementary to see that a document while retrieval must have some statistics populated pertaining to the query (i.e. term frequency, document length, etc.)

2.3. Scorer

The set of documents retrieved by the Index Searcher need not be in any order. The Scorer is responsible for ranking the results based on their relevance to the query. At the very least, you will implement two different relevance models: tf-idf based vector similarity model and BM-25 or the Okapi probabilistic model. Thus as the diagram illustrates, the Scorer works on a set of unranked documents and generates a set of ranked documents.

3. Technical Details

This project ships with minimal amount of starter code and it is left up to the students to design their architecture and classes beyond the code entry points as detailed below.

3.1. SearchRunner

This is the main class used to interact with the searcher. Before we describe the methods, let us enumerate the different modes under which this class runs:

- Query mode: The program works as query engine that runs in an infinite loop until terminated. It accepts user queries, validates them and displays search results (see details below). This mode is interactive and more user friendly. There is no strict standardization on how the output must be formatted in this case.
- Evaluation mode: The program works as a barebones query engine that reads queries from a given file and logs the ranked results to a given file (see details below). This is a non-interactive mode and expects results to be formatted as specified for correct evaluations.

Irrespective of the mode, the searcher is expected to return at most 10 results.

Methods to be implemented:

	SearchRunner (constructor)	The default constructor for this class.
1.	String indexDir	Root directory where the indexes reside. Same as that passed to IndexWriter.
2.	String corpusDir	Root directory where the corpus resides. **Note that the files would need to be flattened into a single directory as against a hierarchical structure used during indexing. This will allow easy reads using just file ids.**
3.	char mode	Mode as defined above, one of Q or E
4.	PrintStream stream	The print stream to which the search results must be written

void	query	This method parses, validates and executes the given user query in the “Q” mode and prints the results to the stream passed in the constructor.
1.	String userQuery	The query to be executed
2.	ScoringModel model	An optional parameter indicating what scoring model must be used (TFIDF or OKAPI)

void	query	This method parses, validates and executes

		queries from the given file in the "E" mode and prints the results to the stream passed in the constructor
1.	File queryFile	The file from which queries to be read

void	close	Utility method for cleaning up resources, releasing locks etc.

boolean	wildcardSupported	Return true if and only if you have implemented wildcard query support

Map<String, List<String>>	getQueryTerms	Method to get expanded terms for a wildcard query. The key of the map is the original term with the wildcard; the value is the list of expanded terms. The list is not expected to be in any order.

boolean	spellCorrectSupported	Return true if and only if you have implemented spell correct support

List<String>	getCorrections	Method to get ranked queries with corrected spellings. Note that the full query must be returned and not just the corrected word. If no suggestions were found, return null.

We now describe the formatting requirements:

3.1.1. Query mode

This mode has the least formatting requirements, but the following information is required to be printed:

a. Query: The user query as input
b. Query time: Time taken to execute the query, prepare results and print them in ms.
c. Result rank: Rank of a returned result (document) starting from 1.
d. Result title: The title of the news article
e. Result snippet: A short 2-3 line snippet from the news article indicating its relevance to the query.
f. Result relevancy: The relevancy score for the result.
g. Term highlighting (optional): If needed you can highlight query terms as found in the results using the HTML markup <b>…</b>

3.1.2. Evaluation mode

We first describe the formatting for the input file and then how the output should be formatted.

3.1.2.1. Input file

A sample input file is shown below, followed by explanations

```
numQueries=3
Q_1A63C:{hello world}
Q_6V87S:{Category:oil AND place:Dubai AND ( price OR
cost )}
Q_4K66L:{long query with several words}
```

As can be observed, the file follows the following conventions:

- The first line will always be numQueries=XX where XX denotes the total number of queries in the file
- Each line contains one query, a query is identified by an id and enclosed within { and }. So in the first line – the query id is Q_1A63C and the query is ***hello world***
- No assumptions can and should be made on query ids in terms of their length or characteristics etc. These are ids for TA usage and reference only.

3.1.2.2. Output file

A sample output file is shown below, followed by explanations

```
numResults=2
Q_1A63C:{00217#0.97632, 00062#0.85213}
Q_6V87S:{00088#0.55173, 00712#0.44333,
00262#0.21763}
```

As can be observed, the file follows the following conventions:

- The first line will always be numResults=XX where XX denotes the total number of results in the file. This must always be less than or equal to numQueries from the input file.
- Each line contains one result. As in the query file, the query id appears as the prefix and the results within { and }
- It is not required for the results to be in the same order as the input. As long as the correct query ids are referenced, you should be fine.
- A result set contains at most 10 ordered tuples separated by commas.
- A tuple consists of the file id and its relevancy score denoted as <id>#<score>.
- The file ids must match the ones in the index / corpus.
- The relevancy scores must be between 0 and 1 (normalize if needed). They must have at most five places of decimal.

- The results must be ordered in decreasing order of relevancy.

3.2. QueryParser

This class as noted before parses a given string into a Query object (refer to Query class in the following section). You are expected to implement just one method as follows:

Query	(static) parse	This method parses the given user string into a Query object if a valid query was provided. A QueryParserException should be thrown if the provided query is invalid
1.	String userQuery	The query to be parsed
2.	String defaultOperator	Default operator (AND or OR) that should used to join clauses in the query. By default this should be “OR” and this parameter can be optionally provided to override this behavior.

3.3. Query

This class denotes a parsed Query. You are free to add any functionality and code as you may like here. We however expect the following method to be implemented.

String	toString	Returns a string representation of the query as detailed in the following section.

3.3.1. Query syntax and string representation

This user query should follow the grammar specified. It would perhaps be most instructive to show some examples.

S.No	User query	String representation
1.	hello	{ Term:hello }
2.	hello world	{ Term:hello OR Term:world }
3.	“hello world”	{ Term:”hello world” }
4.	orange AND yellow	{ Term:orange AND Term:yellow }
5.	(black OR blue) AND bruises	{ [Term:black OR Term:blue] AND Term:bruises }
6.	Author:rushdie NOT jihad	{ Author:rushdie AND <Term:jihad> }
7.	Category:War AND Author:Dutt AND Place:Baghdad AND	{ Category:War AND Author:Dutt AND Place:Baghdad AND [

	prisoners detainees rebels	Term:prisoners OR Term:detainees OR Term:rebels] }
8.	(Love NOT War) AND Category:(movies NOT crime)	{ [Term:Love AND <Term:War>] AND [Category:movies AND <Category:crime>] }

4. Testing and evaluation

We would use a hybrid methodology to evaluate this project. Largely, the project evaluation can be broken down into three components:

- Automated tests: We would evaluate the QueryParser and Query classes using automated tests like in Project 1.
- Manual testing: We would manually evaluate your relevance models in "Query" mode.
- Performance testing: We would evaluate the overall performance of your searcher using standard metrics in the "Evaluate" mode.

We first present the grading guidelines followed by details of individual testing modes.

4.1. Grading guidelines

The following table summarizes the overall grading guidelines:

S.no.	Functionality / Evaluation Method	Points
1.	**Automated: QueryParser and Query**	**20**
2.	**Manual: Relevancy models**	**40**
i)	Basic result list: Ranking, Title, Time	5
ii)	Relevancy scores: Tf-Idf	10
iii)	Relevancy scores: Okapi	15
iv)	Snippet generation	10
3.	**Performance evaluation**	**40**
i)	Simple term queries (1 – 3 words)	10
ii)	Simple faceted queries (1 – 3 words, 1-2 facets)	10
iii)	Lengthy term queries (5 – 10 words)	10
iv)	Complex faceted queries (1-3 words, multiple clauses)	5
v)	Heterogeneous queries (combination of all above)	5
4.	**Bonus (Optional) – One amongst the following**	**20**
i)	Wildcard queries	
ii)	Spelling correction and suggestion	

4.2. Automated tests

Like in project 1, we would release some test cases to evaluate the QueryParser and Query classes as a pair. Unless otherwise stated, each test would have equal credit.

4.3. Manual testing

In this mode, we would largely evaluate your code on three levels:

- Basic results and information: We would run some queries and validate that for the results returned, your results show a rank, title and the total query time. We would not validate the accuracy of the results here.
- Relevance scoring: We would evaluate a given query set for both the relevancy models. We would validate if the presented relevancy scores seem reasonable.
- Snippet generation: Lastly we would evaluate the snippets generated for the given query – the quality and relevance of the snippets would be the main criteria.

4.4. Performance evaluation

This forms the crux of the project. After you have implemented the relevancy models, you would try and improve the system performance for a variety of queries for a given set of metrics. We would score your system on two metrics:

- $F_{0.5} = \frac{1.25*precision*recall}{0.25*precision+recall}$ (precision is twice as important as recall)
- Normalized Discounted Cumulative Gain

We would compute these values for every query in a given set and average them to be representative for a set. We would release a set of training queries with their relevance judgments for you to train your systems. By train here, we refer to tuning parameters for the relevance models – some examples as given below:

- You could use different tf-idf schemes and/or length normalization schemes
- Tuning parameters for pruning and tf scaling for okapi
- Switching models based on query lengths, etc.

We would evaluate your systems on a test set with similar queries as provided during training. Note that the difference between the Q and E modes is that the former should use some variant of the provided model whereas in the latter you can pick & choose or combine them as needed. You are of course free to switch between variants of a given model in the Query mode too (i.e. change tf scaling based on query length for example, etc.).

The way your system would be graded would depend upon the band in which your metric scores lie for each set. For example, if your F-measure scores are more than 0.8 for a set you will receive full credit etc. We would release the scoring information shortly for each set.

4.5. Indexing and code reuse

So far we have addressed only the part about querying but not about the index on which we are working. For this project, as hinted before, we would continue to work on the index created during project 1. You are free to modify your code from project 1 (with some exceptions as below) to build the index. You would need to re-submit your code from project 1 as well as code for this project. More details can be found in the following section.

- Do not modify any code that is referenced from the Runner class
- Thus, no changes to Document or IndexWriter's addToIndex method.
- **You can change other classes including their functionality as you like**

However, not all teams may be comfortable to use their code from project 1. In such a case you can request us to provide you with a copy of our implementation. However, there are some caveats:

- The code would be provided with absolutely no guarantees. Any bug fixes, changes etc. would be your responsibility.
- **A penalty of 10% would be applied to your final score**. This means if you score say 85/100, we would rescale your score to 76.5. This is irrespective of whether you use the code or not in your final submission.

4.6. Bonus

In this project, if you so choose to do – you can pick one amongst two bonus options as described below.

4.6.1. Wildcard queries

As part of this requirement, you would be expected to support two wildcard operators:

- ? : Used to signify one character
- *: Used to signify multiple characters

The objective of implementing wildcard queries is two-fold:

1) Correctly substitute wildcard characters in the given query term to convert them to dictionary terms
2) Execute the expanded query and rank the retrieved documents.

We would use automated tests to validate your implementation. We will invoke your code in the evaluation mode and also call the getWCQueryTerms method to test both components. Your MAP scores would determine 50% of your points and the expanded terms would determine the rest.

4.6.2. Spelling correction

As part of this requirement, you are expected to implement spell correct for seemingly incorrect queries. We would use automated tests to compute the MAP of your returned results. Based on the band in which they lie, would determine your total score.

5. Testing and evaluation

We expect ONLY the source code to be submitted. Please zip your entire source code including the code from project 1 (the src directory, edu should be the top directory) and name it as cse535_<team name>.zip. Please convert team name to all lowercase on your submission. Please replace all spaces in the team name with underscores.

To submit, from any cse unix machine (timberlake / metallica / etc.) invoke:

```
submit_cse535 <zip file name>
```

and press enter. You would receive a confirmation message. You can make multiple submissions; any new submissions would overwrite the old ones. In case of name confusions, the latest file will be used.

Late days: The following late day penalties exist –

- 1 day: Lose 10 points
- 2 days: Lose 20 points
- 3 days: Lose 30 points

These will be applied on top of your total score. If you make any submissions after the deadline, it would be considered as a late day is being used.