**Information Retrieval :: Simple Search Engine**

**Sai Sri Malempati ()**

**SiriValli ()**

**Part 1: Building Inverted Index**

**Design:**

1. Convert the documents words to tokens by performing the Preprocessing steps.

Preprocessing steps involves:

* Convert to Lowercase.
* Tokenize and remove words like [ ; : ! \* . , /] in order to avoid the noise in the data.
* Remove StopWords.
* Perform Stemming.

1. Calculate each token position in the documents and sort the positions and calculate the term Frequency.
2. Create an IndexItem for all terms with term and its occurrence position in Document
3. If the term already exists then add its DocumentID, Positions, termFrequency to the existing term values of the items or else add these term : (DocumentID , Positions, termfrequency) to the items.
4. Calculate the Inverse Document Frequency for all the terms in the items.
5. Save the InvertedIndex object to the file name index\_file.

**Implementation:**

We have implemented the building of InvertedIndex using three Classes Posting, IndexItem, InvertedIndex in index.py file.

Read all the documents form Cran.all using CranFile(“cran.all”) from cran.py. Consider the title and body for each document and perform the preprocessing steps in InvertedIndex.indexDoc() . For preprocessing, first lowercase all the words using ConvertLowerCase() function in util.py and then tokenize them using Tokenize in util.py. Next we must remove the stopwords and perform stemming using removeStopWords, stemming respectively. We have removed stopwords, by using stopwords.text, **stopwords from nltk.corpus** . For Stemming we have used **PorterStemmer from NLTK** . Then for each of these stemmed words we must calculate the position of occurrence in the document using Find\_positions() method. Create IndexItem() class object with stemmed tokens and add these position list using Posting class and termFrequency is also calculated here. If the InvertedIndex.items has already these token then add to the existing token or else add as new one. Calculate the Inverted Document Frequency for each term in the InvertedIndex items. After Processing all the Document then call save method to save InvertedIndex() object. In save method we have used jsonpickle.encode() from jsonpickle to serialize the InvertedIndex object. Figure 1.2 and Figure 1.3 has image of index\_file

### TermFrequency(t)= The Frequency (Number of times) of a term ‘t’ Occurs in the Document

### Inverse Document Frequency (IDF) = Log(Total Number of Documents/Number of Documents Contains the Term ‘t’)

Run the Index.py ::: python index.py cran.all index\_file

A close up of a logo

Description automatically generated

Figure:: 1.1

**A close up of a map

Description automatically generated**

Figure:: 1.2

**A close up of a window

Description automatically generated**

Figure:: 1.3

**TestCases:**

1. Considered a sample document and have run the

* Removal of stop words
* Tokenize
* Stemmed

1. Loading index File

Test\_all has code for running the test method in index.py.

Command to run:: python test\_all.py

A screenshot of a social media post

Description automatically generated

Figure 1.4

### Part 2: Query Processing

### Design:

### Load the Index\_file which is generated from step1

### Fetch the Query for the given query\_id from query.text

### Perform the preprocessing step on query and turn them to terms

### Boolean Model: Find the all Query term existence in the document using index\_file. If the every term in Query present in the document then return document ID or else don’t return the DocumentID [We need to perform AND operation in search of query terms in the document]

### Vector Model: Calculate the termFrequency and Inverse Document Frequency for TF-IDF calculations of the Query terms and Document terms. Calculate cosine similarity between the query terms(tf-idf) and document terms (tf-idf). Iterate this process For all the documents for given Query and return the Top 3 Ranked documents i.. e., with highest cosines values.

### TermFrequency(t)= The Frequency (Number of times) of a term ‘t’ Occurs in the Document

### Inverse Document Frequency (IDF) = Log(Total Number of Documents/Number of Documents Contains the Term ‘t’)

### TF-IDF= (1+log(termfrequency)) \* IDF

### Cosine Similarity formula is in image: let a and b the query and document vectors with TF-idf values

### 

### Batch Mode: we have randomly selected 20 queries from query.text file and Executed for each query both Boolean and vector model. we have calculated the time taken by each query for both models using process\_time from time. Sum the time for all 20 queries for each model which is the total time for all 20 queries. Repeat this process for 5 times. At end write these to file Processing\_Time.csv

### Implementation:

### Query.py has QueryProcessor class which has booleanQuery, vectorQuery, BatchEvaluation methods.

### Create an object for QueryProcessor object with initializing of this class variable like raw\_query (query.text data),index(loaded index\_file), docs (Cran.all documents read), querynumber (query number given through runtime arguments).based on model\_type given we run the model, i.e., 0,1,2 for Boolean, Vector, Batch execution respectively.

### Boolean:

**Method Name:**booleanQuery()

### Fetch the Query for given querynumber and run the preprocessing steps for query given using preprocessing method which does lower case , tokenize , stopwords removal ,stemming by calling methods from util.py. Then find Each query terms exists in the all document if so add the document ID to list else don’t add them .Repeat this process for all the Query terms .After that find the intersection of those list formed with Document IDs and return those DocumentIDs. Here we are performing the AND operation if all terms exists in a document then return or else doesnot return that document.

### Run command for Boolean:: python query.py index\_file 0 query.text 029

### A screenshot of a social media post Description automatically generated

### Figure 2.1

### Vector:

Method name: vectorQuery(k) where k is the top number of rankings to be returned

### Fetch the Query for given querynumber and run the preprocessing steps for query given using preprocessing method which does lower case , tokenize , stopwords removal ,stemming by calling methods from util.py and form the query terms .Calculate the TF-IDF for query terms and Document terms for each of the document. Now calculate the cosines similarities for document terms and query terms. After the process return Top 3 results using nlargest method from heapq module.

### Run Command :: python query.py index\_file 1 query.text 04

### A screenshot of a social media post Description automatically generated

### Batch Execution:

**Method name:** BatchEvaluation()

### We have considered randomly 20 from query\_id list and for each query start the timer and called Boolean model once returned from stop timer and the start the timer again before calling vector model and stop timer once returned from vector model .Add these times to booleanTime, vectorTime respectively. Repeat this process for 5 times. At end write each iteration ID,booleanTime,VectorTime to Processing\_time.csv using file operations in python. This is done to evaluate the processing time taken by Boolean and vector models

### Command to Run:: python query.py index\_file 2 query.text 20

### A screenshot of a social media post Description automatically generated

### Figure :: 2.3

### Report ::

|  |  |  |
| --- | --- | --- |
| **Iteration** | **booleanModel(seconds)** | **vectorModel(seconds)** |
| 1 | 2.875 | 741.75 |
| 2 | 3.875 | 634.109375 |
| 3 | 7.390625 | 673.71875 |
| 4 | 2.84375 | 650.421875 |
| 5 | 5.375 | 1185.0625 |

### Part 3: Relevance Evaluation

### Design:

### 

### 