Afshin Jamali
CSC 575
Professor: Bamshad Mobasher
Intelligent Information Retrieval
Version 1.0

Summary:

The program I created contains the following elements:

- Query interface
- Search word by domain: [animal, medical, car, assignment]
- Search word without any domain.
- Read in documents and parse them using inverted file indexing format
- Relative term frequency information
- Utilize existing stemming and stop list tools
- Weighting scheme TF-IDF
- Vector-space model with comparing Cosine similarity measures.
- Save the index to an offline storage (index.csv)
- Open files based on search results

Please refer to README.txt for instructions on how to run the program.

Please refer to sample_run.pdf for illustration of how the program works.

Third party tools:

For stop words, I used a list from the following source:
https://javaextreme.wordpress.com/category/java-j2se/java-string/remove-stop-words-from-a-string/

For stemming, I used the snowball source codes from the following source:
http://snowball.tartarus.org/download.php

Please refer to “SearchEngine/src/org” folder for classes related to stemming.

Detailed description:
Please see JavaDoc folder for program structure.

1. **Main.java**
   The main method calls two functions:
   Database.run(); for creating an index or populating an existing index.
   Program.main(args); this method produces the user interface

2. **Database.java**

This class handles all functions related to creating an inverted index or loading an existing one. The run method first reads all the lines in the "**index.csv**" file. Then it checks to see if it's empty. If so, it will call the "**initializeIndex()**" method. If the index is not empty, it will call the "**loadIndex()**" method.
initializeIndex();The method searches the directory and for each term, does the stemming and calls the addToDictionary method. It creates the index using the stopwords list and the snowball stemmer classes.
For the stopwords it checks to see if a word exists in the stopword list. If so, it will not count it in the dictionary.
For stemming, the englishStemmer class is instantiated. The stemming process goes through each character until it reaches a space, dot, coma, and so on. Then it will call stemmer.stem() to stem the word.
Once the word is stemmed it, will add it to the dictionary. Upon completion, stores the index to the index.csv file for subsequent sessions by calling the "**saveIndex()**" method.

addToDictionary();This method adds the current word to the dictionary, including term frequencies and doc frequencies. In addition it adds each domain (directory) to the docsInDomain variable. This is used for TF-IDF calculation where you would need total number of docs. Two dictionaries are used, dictionary and postings.
"postings": contains the inverted index.
"dictionary": contains additional information.

```
//additional info for each term, contains doc frequency and total occurrence
across all docs
static Map<String, HashMap<String, Term>> dictionary = new HashMap<String,
HashMap<String, Term>>();

//contains postings. For each term, contains key: doc id followed by value:
posting
static Map<String, HashMap<Integer, Document>> postings = new TreeMap<String,
HashMap<Integer, Document>>();

//total number of files in each directory
static Map<String, List<String>> docsInDomain = new HashMap<String,
List<String>>();
```

3. **Program.java**
   This class is used to compute tf x idf weights and cosine similarity depending on the domain. See code below for determining document frequency in a collection:

```
if (domain.contains("not set...")) {
   for (String dom : Database.dictionary.get(term).keySet()) {
      docFreq += Database.dictionary.get(term).get(dom).getDocFrequency();
   }
} else {
   if (Database.dictionary.get(term).containsKey(domain))
      docFreq = Database.dictionary.get(term).get(domain).getDocFrequency();
}
```

TF-IDF are calculated each time user searches since the domain, since domain can change each time. This class is also used for GUI components, including mouse and action listeners. The

results are displayed to the user. includes a no arg constructor. Two listeners are added, Action listener and Mouse listener. The core functionality is in the Action listener, when the user clicks the Search button. The query is produced, the collection of relevant documents is obtained, then calculated using tf x idf weights, and measured using cosine similarity and finally displayed back to user see interaction below interaction when user clicks the search button:

```
searchButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
//produce the query, including term frequencies
        produceQueryDictionary(searchTextField.getText());
//retrieve relevant documents based on domain
        Set<Integer> docCollection = getDocCollection();        // Calculate
TF x IDF
        Map<String, HashMap<Integer, Document>> tf_x_idf =
calculateTFxIDF(docCollection);
//calculate cosine similarity and display results.
        displayResult(calculateCosineSim(tf_x_idf, docCollection));    }
});
```

ProduceQueryDictionary(searchText); This method stems and produces the query dictionary that contains term frequencies. If a user types “engine engine engine car” the term frequency will be 3 for engine. This process is similar to creating an inverted index except that it does not contain a document id.

getDocCollection(); This method collects all relevant document id's based on the domain chosen by the user. If no domain is chosen, it will treat the collection as all the files in all directories. If a term does not appear in the index, the program will check if the query contains any term in the index. See code below:

```
//if term is not in the index, check if a term similar to the index term
exists!!!!!!
if(!Database.postings.containsKey(term)) {
    for (String t : Database.dictionary.keySet()){
        if (term.contains(t)) {
            for (int id : Database.postings.get(t).keySet()) {
                //is the term relevant to the domain?

if(Database.postings.get(t).get(id).getDocName().contains(domain)) {
docCollection.add(id); //add relevant id's by domain
…
```

calculateTFxIDF(docCollection); This method calculates TF-IDF once for the terms in the domain and once for the query . See code below:

```
//*********** calculate tf x idf ***********
weightedTerms.get(term).get(id).setTermFrequency(d.getTermFrequency() *
(Math.log(totalDocs / docFreq) / Math.log(2)));

//*********** calculate tf x idf for query ***********
if(query.containsKey(term)){
    query.get(term).setTermFrequency(query.get(term).getTermFrequency() *
(Math.log(totalDocs / docFreq) / Math.log(2)));
}
```

calculateCosineSim(weighted terms, docCollection); This method calculate the cosine similarity. First it calculates the norm for query and domain. Next it calculates the dot product, and computes the cosine similarity by dividing the dot product by the norm of query multiplied by the norm of document. Finally it sorts the sim in descending order. see code below:

```
//calculate the sum of all the queries squared
for (String term : query.keySet()) {
    sumPower += Math.pow(query.get(term).getTermFrequency(), 2);
}
double sqrt = Math.sqrt(sumPower); // get square root of the sum of queries
squared
norm.put(0, Double.valueOf(df.format(sqrt))); //add to variable, assign index
0 for queries

//calculate the sum of all the terms squared for each document
for (int id : docCollection) {
    sumPower = 0;
    for (String term : Database.postings.keySet()) {
        if (Database.postings.get(term).containsKey(id)) {
            sumPower +=
Math.pow(weightedTerms.get(term).get(id).getTermFrequency(), 2);
        }
    }
    sqrt = Math.sqrt(sumPower); // get square root of the sum of terms
squared
    norm.put(id, Double.valueOf(df.format(sqrt))); //add to variable, retain
id of doc measured
}
//get dot product of query and terms > 0 frequency
for (int id : docCollection) {
    double sumDot = 0;
    for (String term : query.keySet()) {
        if (Database.postings.get(term).containsKey(id)) {
            sumDot += (query.get(term).getTermFrequency() *
weightedTerms.get(term).get(id).getTermFrequency());
        }
    }
    dot.put(id, Double.valueOf(df.format(sumDot)));
}

//***********calculate similarity and add to dictionary**************
for (int id : docCollection){
    sim.put(id, dot.get(id) / (norm.get(0) * norm.get(id)));
}
return sortByValues((HashMap) sim);
```

displayResult(Hashmap); the result is display back to the user. If the user wishes to see inside the document, they can **double click** on the selection.

4. **Document.java**
   Contains document id, document name, and term frequency (number of times term appears inside document)

5. **Term.java**

Contains number of documents containing the word and the total number of appearances across all documents in a collection.

6. **Query.java**
   Contains number of times term appears inside the query and the original non-stemmed word. The original word is not used for anything, however it might be useful for future enhancements.