CA4009: Search Technologies
Laboratory Session 4 & 5
24th November 2016

Jordan Healy - 13379226
Tríona Barrow - 11319851

Laboratory 4

- *Query Expansion using Relevance Feedback*

URL -
http://136.206.115.117:8080/IRModelGenerator/SearchServlet?query="bone%20disease"&simf=BM25&k=1.2&b=0.75&numwanted=10

To begin our program, we started out using a URL container to store the URL String, and a URLConnection to open a HTTP request. We handled the parameters by storing them into separate variables and passing them into the URL variable at initialisation. We decided to store numwanted, query, k and b as strings. To test our connection, we started with a BufferedReader to read in the webpage and print it to console.

```
String query1 = "bone";
String query2 = "disease";
String query = query1 + "%20" + query2;
String simf = "BM25";
String k = "1.2";
String b = "0.75";
String numwanted = "10";

URL url = new
URL("http://136.206.115.117:8080/IRModelGenerator/SearchServlet?q
uery=\"" + query +"\"&simf=" + simf +"&k="+ k +"&b="+ b
+"&numwanted=" + numwanted);
URLConnection uc = url.openConnection();
BufferedReader in = new BufferedReader(new InputStreamReader(
                                uc.getInputStream()));

String inputLine;
inputLine = in.readLine(); // whole page
System.out.println(inputLine);
in.close();
```

This prints out the entire web page, including HTML tags - so we needed to update this to parse out these. From looking at the web elements on the page, we can see that each result is embedded in an `<li>` element, and each section of the result is in a `<div>` within that. The second `<div>` included the term frequency, which is what we needed to use for Robertson's Offer Weight calculation.

We began storing the input into a String array, after splitting on the `<li>` element. This results in the initial `<ul>` element being stored in the first index, however each result is stored in each index after this. We then split this again on the closing `<div>` tag to get to the middle `<div>` and stored these in an array. From there we can focus on getting the terms and the frequencies associated with them.

```
String[] eachDocLi = inputLine.split("<li>");
String[] eachDoc = Arrays.copyOfRange(eachDocLi, 1,
                                        eachDocLi.length);
String[] eachDocFreqVector = new String[eachDoc.length];
String[] docNames = new String[eachDoc.length];
for(int i=0; i<eachDoc.length; i++) {
      docNames[i] =
eachDoc[i].substring(eachDoc[i].indexOf("target=\"_blank\">")+"ta
rget=\"_blank\">".length(), eachDoc[i].indexOf("</a></div>"));
      String[] temp = eachDoc[i].split("</div>");
      eachDocFreqVector[i] = temp[1];
}
```

`eachDocLi` is `eachDocument` but also has the “<ul>” in the first position. So we removed it. At the end of this snippet `eachDocFreqVector[]` has all the frequency vectors for each document.

We initialised a HashMap to link the documents to the terms, frequencies and IDF values (String to String Array). We then initialised a Tokenizer and passed in the array that stored the middle div (including the terms, idf and frequencies) for each result, and parsed out each of these. We then calculated DF based on the IDF from the results page by using it as follows - *(1/IDF) * 500,000*

```
HashMap<String, List<String[]>> allDocs = new HashMap<>();
for(int i=0; i<docNames.length; i++) {
      List<String[]> wordsInDoc = new LinkedList<>();
      String docFreqVector =
eachDocFreqVector[i].substring(eachDocFreqVector[i].indexOf("Freq
Vector: <br>")+"Freq Vector: <br>".length());
      StringTokenizer tokenizer = new
StringTokenizer(docFreqVector);
      while (tokenizer.hasMoreTokens()) {
           String[] toAdd = new String[4];
           String[] temp3 = tokenizer.nextToken().split(":");
           toAdd[0] = temp3[0]; // word "bone"
           toAdd[1] = temp3[1].substring(0,
temp3[1].length()-1); // freq 28
           toAdd[2] = tokenizer.nextToken(); // idf 168.25581
           toAdd[3] =
String.valueOf((1/Double.parseDouble(toAdd[2]))*500000);
                 wordsInDoc.add(toAdd);
      }
      allDocs.put(docNames[i], wordsInDoc);
}
```

`allDocs` has the following nested structure. (These are example values)

```
{ "LA020490-0136": [
  [bone, 28, 168.25581, 47.555],
  [million, 4, 18.25381, 7.23],
  ...],
  "LA030689-0082": [
  [],
  ...],
  ...
}
```

We then used the formula from the lab sheet to calculate rw(i) using these values in a method. To store these for each document, we created another HashMap to store the rw(i) value as a Double against the document name as a String. We created a method to count the relevant documents, relying on each word previously stored and just having a count to calculate r(i). Then we used a method for calculating r(i) and stored it in another HashMap, using the terms as the keys to the r(i) values. Our value for N is 500000 whereas the actual total number of documents in the collection is more than that. But this approximate value is okay since N is used in a log function, meaning that as N goes to infinity n(i) will be closer to 1 (in n(i) = N/idf(i)).

```
HashMap<String, Double> allRwi = new HashMap<>();
HashMap<String, Integer> allRi = new HashMap<>();
for(Map.Entry<String, List<String[]>> entry : allDocs.entrySet())
{
   String docName = entry.getKey();
   List<String[]> wordsInDoc = entry.getValue();
   ListIterator iterator = wordsInDoc.listIterator();
   while(iterator.hasNext()) {
      String[] wordContents = (String[])iterator.next();
      String word = wordContents[0];
      int ri = 0;
      double ni = N/Double.parseDouble(wordContents[2]);
      for(Map.Entry<String, List<String[]>> entry2 :
allDocs.entrySet()) {
            String docName2 = entry2.getKey();
            List<String[]> wordsInDoc2 = entry2.getValue();
            ListIterator iterator2 = wordsInDoc2.listIterator();
            while(iterator2.hasNext()) {
                  String[] wordContents2 =
(String[])iterator2.next();
                  if(word.equals(wordContents2[0])) {
                        ri++;
                        break;
                  }
            }
            allRi.put(word, ri);
      }
      double rwi = rwi(ri, N, ni, R);
      allRwi.put(word, rwi);
   }
```

```
}
```

We then iterated over HashMaps and retrieved the r(i) and rw(i) values for each term. We then used those with the formula from the lab sheet again - *ow(i) = r(i) × rw(i)*
These were stored with the terms in another HashMap, then we copied the values from the HashMap pairs to a Set, then to a List and used `Collections.sort()` on the List to organise them from smallest to biggest. We then retrieved the term from the HashMap using the ow(i) value, and grabbed the last 5 terms as our top ranked terms.

```
HashMap<Double, String> allOwi = new HashMap<>();
for(Map.Entry<String, Double> entry : allRwi.entrySet()) {
	String word = entry.getKey();
	double rwi = entry.getValue();
	double owi = allRi.get(word) * rwi;
	allOwi.put(owi, word);
}
```

These turned out to be:

Term	**ow(i)**
osteoporosi	73.08601786324982
bone	67.80955500667315
diseas	59.03415757000548
fractur	57.3465143731149
calcium	52.86989297745741
menopaus	49.200240940392895
medic	46.28548805421567
risk	42.73051443167924
elderli	41.860922182715406
health	41.76691746828662

Laboratory 5

- *Experimenting with Query Expansion in IR*

We modified our original program to retrieve the document names as well as the queries, so that we can use the previous results file for evaluating them with trec_eval. We did this for the initial expanded query - “bone disease”, and then used the top two distinct ranked results (“osteoporosi” and “fractur”) with this again. We set R as 10, returning only 10 documents, and then repeated this with the expanded query terms.

After carrying out this query - we found that the top ranked terms had changed, and the ow(i) value had increased. This suggests that the relevancy of the documents increased with the expanded queries. These turned out as follows:

Term	**ow(i)**
osteoporosi	105.18191164860225
menopaus	85.66582859963782
calcium	81.82910310656314
fractur	78.4399570761149
bone	67.80955500667315
diseas	52.58696136870442
hormon	51.84287728455569
women	48.43596857252667
mass	46.67220758041752
calcitonin	45.699229410463964

Looking at the value for ow(i) we can see that “osteoporosi” is now more relevant than before. Also, bone has the same ow(i) value but is less relevant in comparison to the other terms. This could be due to the relevancy of the query terms themselves, as the expanded terms generate their own set of results, and will appear with different term frequencies.

We also decided to carry this out with the terms that had the lowest ow(i) value, to see how we can compare the changes in results from highly relevant queries to lower relevant queries.

The documents we retrieved were as follows:

“bone” “disease”	**“bone” “disease” “osteoporosi” “fractur”**	**“bone” “disease” “countri” “london”**
LA020490-0136	LA071290-0133	FT943-12269
LA030689-0082	LA020490-0136	FR940317-0-00022
FR940317-0-00022	LA030689-0082	FT931-12903
FT943-12269	FR940525-1-00062	LA020490-0136
FR940603-2-00065	FR940525-1-00078	LA030689-0082
LA022290-0150	LA011389-0029	FT941-2976
FR940525-1-00062	FR940603-2-00065	FR940603-2-00065
FR940525-1-00078	LA032290-0151	LA022290-0150
LA111789-0114	FT932-1044	FR940525-1-00062
LA071290-0133	LA051490-0120	FT931-3937

We then used our previous files from lab 2, and used this layout to find a topic that related to our queries. In this case, topic 403 seemed the most related to both search queries, as it dealt with osteoporosis and bone decay. We created a res file ourselves using the results.test file from lab 2, and updated this with the topic and document numbers. We then ran trec_eval with the qrels file from that lab and our new res file.

Our MAP results were:

Query	**MAP**
“bone” “disease”	0.1474
“bone” “disease” “osteoporosi” “fractur”	0.3088
“bone” “disease” “countri” “london”	0.0421

We have noticed that using expanded queries with relevant terms has a positive effect on the relevance and precision of the documents returned. As the individual terms have strong related meaning, this increases the amount of related documents returned to the user.

We also tried to use the same document for our less relevant query - as we can see this has a far worse precision value. We picked the two least relevant terms from the expanded query. The terms in the query seem less related to one another than our other queries, which seems to be causing less relevant documents to be returned.