Information Retrieval Spring 2012 Safa Khan Project report

**Final Project**

Indexing:

**Project:**

1. Building an Inverted Index using the CACM documents
2. Using this Inverted Index, run the retrieval algorithms and evaluate them with the provided cacm.rel file

Creating the inverted Index:

In this Project we were asked to replicate the functionality of Lemur index.

I downloaded the CACM document collection of Glasgow and processed it. Firstly I compared the document against the stoplist provided from Project 1. The words that were present in both the document collection ad stoplist were to generate a new document list.

To this new document list I applied Stemming using the Porter Stemmer.

Document Processing:

1) The Glasgow cacm.all document has many identification symbols. For eg: “**.I”** represents the document id, “.T” represents the title, “.W” represents the content , “.K” represents keywords and “.A” represents Authors.

2) I read the contents of the document into a list. This list was compared with the stoplist provided (‘stoplist.txt’) . The words that were intersecting between the two were discarded and a new list without the stopwords was generated.

3) To this filtered text I applied Porter\_Stemmer.py to generate a new document “stemmed.txt”

I read the contents of stemmed.txt into a list called stemtext.

4) Now I start the process of collecting document information as follows:

**Please note:** After stemming the words become lower case

1. I removed the first **“.i”** and read the contents between the first **.t** an **.n**
2. From this list I removed punctuation via a procedure (rem\_p) and then removed unnecessary alphabets (procedure rem\_a). I also removed numbers in the range 1 to 65.
3. I then stored this processed list into another list along with the document id.
4. From here I continue from step1 to step 3 until I have a list of all document words in a list along with the document id.
5. This List is called as spacelist.

5) I created a list called as termlist which processed the spacelist such that it separated the term, its frequency in a document and the length of the document.

6) Using the termlist I created files with their names as the term names in the termlist containing the docid, term frequency(tf) and docid. All these files were put in a folder called “irfiles”

7) Next I concatenated all the information in all these files into a single inverted index file and recorded the offset and length of each file in a dictionary in python.

Query processing:

Using the querytext file provided by Glasgow I applied the same technique as for document processing.

I read the content into a list compared with the stoplist and generated a new list. To This I applied Stemming and separated the queries into different lists.

I compared each query word with the names of the term files and removed the query word that does not exist in my collection.

**How do I retrieve information for terms in the query?**

For each query term I find their location in the inverted index using the offset dictionary I created earlier. For each term I have docid, tf and doclen.

I have used these updated queries for the different retrieval models.

**What is the format of individual termfiles and Inverted Index?**

Each of my term files has the format:

Docid tf Doclen

Docid tf Doclen

**:**

**:**

So all the document ids are placed one below the other for every occurrence of that term in different documents

The inverted index simply concatenates all the term files into a single file called “inv\_index” in my case.

For the term files I had calculated their offsets and their lengths and stored them separately in a hash Table. In my case the dictionary is called termdict, and it has the offset and lengths for each termfile.

For ctf and df , I have two simple procedures that calculate them at run time.

df procedure simply counts the number of docid ‘s for a term and ctf procedure adds up the corresponding tf values of the term .

Following are values that I get for total doclength, average document and query length.

N =3204 //Number of documents

Total document length (total\_dlen) = 129682

Avg doc length = (129682/3204) = 40.475031211

Avg query length = 13.125

Running the Retrieval Algorithms:

**Retrieval Models:**

The retrieval models I implemented are as follows:

* Vector space model, Okapi-tf
* Vector space model, Okapi tf\*idf
* Language model, Laplace Smoothing
* Jelinek Mercer Smoothing
* BM-25

The following table gives the Average Precision(non-interpolated), R-precision, Precision at 10 and 30 docs for the 5 models:

Overall Evaluation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Okapi-tf** | **Okapi tf\*idf** | **Laplace Smoothing** | **Jelinek Mercer** | **BM-25** |
| **Average Precision (non-interpolated)** | 0.2539 | 0.3767 | 0.2529 | 0.2655 | 0.3703 |
| **R-Precision** | 0.2412 | 0.3733 | 0.2395 | 0.2647 | 0.3502 |
| **Precision at 10 docs** | 0.2692 | 0.3769 | 0.2788 | 0.2577 | 0.3558 |
| **Precision at 30 docs** | 0.1429 | 0.2231 | 0.1481 | 0.1596 | 0.2186 |

The following bar chart shows the overall evaluation of the 5 models:

**Interpolated Recall-Precision Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Interpolated recall-precision** | **Okapi-tf** | **Okapi tf\*idf** | **Laplace Smoothing** | **Jelinek Mercer** | **BM-25** |
| 0.00 | 0.6740 | 0.7914 | 0.6595 | 0.6432 | 0.8150 |
| 0.10 | 0.5276 | 0.7132 | 0.5428 | 0.5197 | 0.7212 |
| 0.20 | 0.4074 | 0.5878 | 0.4100 | 0.4176 | 0.5700 |
| 0.30 | 0.3300 | 0.5013 | 0.3288 | 0.3583 | 0.4949 |
| 0.40 | 0.2703 | 0.4276 | 0.2625 | 0.2900 | 0.4183 |
| 0.50 | 0.2283 | 0.3603 | 0.2291 | 0.2476 | 0.3569 |
| 0.60 | 0.1722 | 0.2944 | 0.1786 | 0.2106 | 0.2868 |
| 0.70 | 0.1245 | 0.2323 | 0.1245 | 0.1542 | 0.2259 |
| 0.80 | 0.1103 | 0.1985 | 0.1097 | 0.1271 | 0.1918 |
| 0.90 | 0.0839 | 0.1323 | 0.0796 | 0.0955 | 0.1276 |
| 1.00 | 0.0703 | 0.0959 | 0.0649 | 0.0707 | 0.0935 |

**Figure:** Graph shows the Interpolated Average Precision for the different retrieval models

From the above figure we observe that okapi tf\*idf and BM-25 models have the best performance among all the five retrieval systems. In this graph okapi tf\*idf and BM-25 almost overlap.

**Why do Okapi tf\*idf perform better than other models?**

In the case of Okapi tf\*idf , the idf values make the difference. Idf is the inverse document frequency.

The idf for a rare term is high, whereas the idf for a frequent term is usually low. The tf\*idf weights filter out the common terms.

The idf is represemted as the log of the quotient of the Total number of documents and the document frequency. If a term occurs in more documents, this reduces the idf value, making the the tf\*idf approaching to a 0. If 1 is added to the denominator a term that occurs in most of the documents will have a negative idf, as opposed to a term that occurs in a single document which will have idf as 0.

This can predict the similarity between the documents and Queries.

In the case of BM-25, it ranks the documents bases on the query terms that appear in each document.

The Following Table lists the Average Precision values(non-interpolated) for the queries using the different retrieval models:

Average precision for queries:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Qid | Okapi-tf | Okapi tf\*idf | Laplace Smoothing | Jelinek Mercer | BM-25 |
| 1 | 0.0405 | 0.1576 | 0.0747 | 0.1903 | 0.3086 |
| 2 | 0.1452 | 1.0000 | 0.3373 | 1.0000 | 1.0000 |
| 3 | 0.0740 | 0.1929 | 0.0520 | 0.1129 | 0.1984 |
| 4 | 0.1687 | 0.1381 | 0.1464 | 0.1080 | 0.1230 |
| 5 | 0.0745 | 0.1086 | 0.0889 | 0.1245 | 0.0956 |
| 6 | 0.3076 | 0.3611 | 0.3366 | 0.2059 | 0.2687 |
| 7 | 0.1620 | 0.4040 | 0.2360 | 0.2890 | 0.3939 |
| 8 | 0.2203 | 0.1931 | 0.4513 | 0.3409 | 0.2758 |
| 9 | 0.1521 | 0.1599 | 0.0999 | 0.0725 | 0.1159 |
| 10 | 0.2324 | 0.6093 | 0.2258 | 0.3602 | 0.5137 |
| 11 | 0.4397 | 0.4006 | 0.3548 | 0.3476 | 0.3546 |
| 12 | 0.2900 | 0.4824 | 0.2697 | 0.4763 | 0.4577 |
| 13 | 0.3270 | 0.2944 | 0.2933 | 0.0640 | 0.1756 |
| 14 | 0.2011 | 0.3179 | 0.1168 | 0.0739 | 0.1609 |
| 15 | 0.0700 | 0.1224 | 0.0729 | 0.1271 | 0.1760 |
| 16 | 0.1121 | 0.1531 | 0.1255 | 0.0807 | 0.1261 |
| 17 | 0.1127 | 0.1940 | 0.1241 | 0.1835 | 0.2497 |
| 18 | 0.1477 | 0.1654 | 0.0865 | 0.0562 | 0.0867 |
| 19 | 0.5901 | 0.7257 | 0.4801 | 0.7119 | 0.6974 |
| 20 | 0.1204 | 0.5639 | 0.3333 | 0.5599 | 0.5636 |
| 21 | 0.0634 | 0.2933 | 0.0642 | 0.1845 | 0.2102 |
| 22 | 0.0387 | 0.3517 | 0.0869 | 0.1335 | 0.2009 |
| 23 | 0.0409 | 0.1013 | 0.0362 | 0.0298 | 0.1049 |
| 24 | 0.0127 | 0.1674 | 0.0104 | 0.0340 | 0.1655 |
| 25 | 0.2966 | 0.4027 | 0.3001 | 0.2963 | 0.3695 |
| 26 | 0.2023 | 0.4585 | 0.2043 | 0.3700 | 0.5012 |
| 27 | 0.2451 | 0.3212 | 0.2224 | 0.1548 | 0.1996 |
| 28 | 0.7056 | 0.8192 | 0.7667 | 0.6481 | 0.7782 |
| 29 | 0.4383 | 0.7760 | 0.2649 | 0.4407 | 0.5807 |
| 30 | 0.0404 | 0.6218 | 0.1381 | 0.2543 | 0.5167 |
| 31 | 0.8333 | 1.0000 | 0.7000 | 0.1043 | 0.4500 |
| 32 | 0.3739 | 0.4385 | 0.4086 | 0.2801 | 0.5639 |
| 33 | 1.0000 | 0.2000 | 0.2500 | 0.0139 | 0.0625 |
| 36 | 0.2644 | 0.3739 | 0.3255 | 0.4360 | 0.4156 |
| 37 | 0.2349 | 0.1981 | 0.2035 | 0.1252 | 0.1636 |
| 38 | 0.6134 | 0.4883 | 0.5076 | 0.1560 | 0.2631 |
| 39 | 0.3195 | 0.3463 | 0.2969 | 0.1976 | 0.2572 |
| 40 | 0.1145 | 0.4637 | 0.1987 | 0.3117 | 0.5149 |
| 42 | 0.1943 | 0.0813 | 0.1290 | 0.0666 | 0.0574 |
| 43 | 0.0388 | 0.1781 | 0.1152 | 0.1988 | 0.1964 |
| 44 | 0.1306 | 0.1376 | 0.1326 | 0.1478 | 0.1542 |
| 45 | 0.1890 | 0.4183 | 0.2365 | 0.2738 | 0.4296 |
| 48 | 0.1574 | 0.1804 | 0.0379 | 0.0288 | 0.0339 |
| 49 | 0.0300 | 0.1823 | 0.0725 | 0.3626 | 0.3533 |
| 57 | 1.0000 | 0.5000 | 1.0000 | 0.5000 | 0.5000 |
| 58 | 0.1660 | 0.3254 | 0.1383 | 0.1263 | 0.2902 |
| 59 | 0.2024 | 0.4335 | 0.2429 | 0.2596 | 0.4005 |
| 60 | 0.1026 | 0.2186 | 0.1123 | 0.0943 | 0.1719 |
| 61 | 0.3584 | 0.6146 | 0.4317 | 0.4465 | 0.5942 |
| 62 | 0.1216 | 0.0921 | 0.0786 | 0.0360 | 0.0495 |
| 63 | 0.6120 | 0.6608 | 0.6311 | 0.6074 | 0.4421 |
| 64 | 0.0714 | 1.0000 | 0.5000 | 1.0000 | 1.0000 |

The Following Table shows the R-precision for the queries using the different retrieval models

R-precision for Queries:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Qid | Okapi-tf | Okapi tf\*idf | Laplace Smoothing | Jelinek Mercer | BM-25 |
| 1 | 0.0000 | 0.2000 | 0.0000 | 0.2000 | 0.2000 |
| 2 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 1.0000 |
| 3 | 0.1667 | 0.1667 | 0.1667 | 0.1667 | 0.1667 |
| 4 | 0.1667 | 0.1667 | 0.1667 | 0.0833 | 0.1667 |
| 5 | 0.1250 | 0.2500 | 0.1250 | 0.2500 | 0.0000 |
| 6 | 0.0000 | 0.3333 | 0.3333 | 0.3333 | 0.3333 |
| 7 | 0.2143 | 0.4286 | 0.2857 | 0.3571 | 0.4286 |
| 8 | 0.3333 | 0.0000 | 0.3333 | 0.3333 | 0.0000 |
| 9 | 0.2222 | 0.2222 | 0.2222 | 0.0000 | 0.2222 |
| 10 | 0.2286 | 0.6000 | 0.2571 | 0.4000 | 0.5429 |
| 11 | 0.4211 | 0.4211 | 0.4737 | 0.3684 | 0.3648 |
| 12 | 0.2000 | 0.4000 | 0.2000 | 0.4000 | 0.4000 |
| 13 | 0.3636 | 0.3636 | 0.2727 | 0.0000 | 0.2727 |
| 14 | 0.2500 | 0.3864 | 0.1364 | 0.1136 | 0.1591 |
| 15 | 0.0000 | 0.2000 | 0.1000 | 0.2000 | 0.2000 |
| 16 | 0.1765 | 0.2353 | 0.1176 | 0.1765 | 0.1176 |
| 17 | 0.1875 | 0.3125 | 0.1875 | 0.1875 | 0.3125 |
| 18 | 0.0909 | 0.1818 | 0.1818 | 0.0909 | 0.0909 |
| 19 | 0.5455 | 0.6364 | 0.5455 | 0.6364 | 0.6364 |
| 20 | 0.0000 | 0.6667 | 0.3333 | 0.6667 | 0.6667 |
| 21 | 0.0909 | 0.2727 | 0.0000 | 0.1818 | 0.1818 |
| 22 | 0.0000 | 0.3529 | 0.1765 | 0.0588 | 0.4118 |
| 23 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 24 | 0.0000 | 0.1538 | 0.0000 | 0.0769 | 0.2308 |
| 25 | 0.3529 | 0.4510 | 0.3922 | 0.4118 | 0.4510 |
| 26 | 0.2667 | 0.5000 | 0.2667 | 0.3667 | 0.5667 |
| 27 | 0.2759 | 0.3448 | 0.2759 | 0.1724 | 0.3448 |
| 28 | 0.6000 | 0.8000 | 0.8000 | 0.6000 | 0.8000 |
| 29 | 0.4211 | 0.7368 | 0.3158 | 0.2632 | 0.7368 |
| 30 | 0.0000 | 0.5000 | 0.2500 | 0.2500 | 0.5000 |
| 31 | 0.5000 | 1.0000 | 0.5000 | 0.0000 | 0.5000 |
| 32 | 0.3333 | 0.3333 | 0.3333 | 0.3333 | 0.6667 |
| 33 | 1.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 36 | 0.3000 | 0.4000 | 0.3000 | 0.4500 | 0.4500 |
| 37 | 0.2500 | 0.3333 | 0.1667 | 0.0833 | 0.1667 |
| 38 | 0.6250 | 0.5625 | 0.6250 | 0.1875 | 0.1875 |
| 39 | 0.3333 | 0.4167 | 0.3333 | 0.2500 | 0.3333 |
| 40 | 0.1000 | 0.4000 | 0.1000 | 0.3000 | 0.5000 |
| 42 | 0.1905 | 0.1905 | 0.1429 | 0.0952 | 0.0952 |
| 43 | 0.0488 | 0.3171 | 0.1707 | 0.3171 | 0.3659 |
| 44 | 0.1176 | 0.1176 | 0.1176 | 0.1176 | 0.1176 |
| 45 | 0.2308 | 0.4231 | 0.2308 | 0.3462 | 0.4615 |
| 48 | 0.1667 | 0.0833 | 0.0833 | 0.0000 | 0.0833 |
| 49 | 0.0000 | 0.2500 | 0.0000 | 0.2500 | 0.2500 |
| 57 | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 0.0000 |
| 58 | 0.2000 | 0.4000 | 0.2000 | 0.2667 | 0.3667 |
| 59 | 0.2791 | 0.4419 | 0.2558 | 0.3023 | 0.3953 |
| 60 | 0.0741 | 0.2963 | 0.0741 | 0.1481 | 0.1852 |
| 61 | 0.3871 | 0.5806 | 0.3226 | 0.3871 | 0.5806 |
| 62 | 0.1250 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 63 | 0.5833 | 0.5833 | 0.5833 | 0.5833 | 0.4167 |
| 64 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 1.0000 |

Experimenting with different Lambda values for Jelinek Mercer:

Laplace Smoothing Comparison of Different values:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Jelinek Mercer** | **=0.1** | **=0.2** | **=0.4** | **=0.6** | **=0.8** |
| **Average Precision** | 0.2653 | 0.2655 | 0.2653 | 0.2658 | 0.2660 |
| **R-Precision** | 0.2647 | 0.2647 | 0.2608 | 0.2608 | 0.2603 |
| **Precision at 10 docs** | 0.2558 | 0.2577 | 0.2615 | 0.2596 | 0.2596 |
| **Precision at 30 docs** | 0.1596 | 0.1596 | 0.1590 | 0.1596 | 0.1609 |

Following is the graph showing Average Precision,R-precision and precision at 10 and 30 docs for different values of Lambda:

From the above graph we observe that choosing Lambda as 0.8 gives better average precision.

Following is the Interpolated precision recall table for different Lambda values:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Interpolated Recall-Precision** | **=0.1** | **=0.2** | **=0.4** | **=0.6** | **=0.8** |
| 0.00 | 0.6432 | 0.6432 | 0.6420 | 0.6419 | 0.6418 |
| 0.10 | 0.5207 | 0.5197 | 0.5188 | 0.5232 | 0.5249 |
| 0.20 | 0.4175 | 0.4176 | 0.4147 | 0.4146 | 0.4134 |
| 0.30 | 0.3575 | 0.3583 | 0.3582 | 0.3582 | 0.3571 |
| 0.40 | 0.2883 | 0.2900 | 0.2899 | 0.2911 | 0.2911 |
| 0.50 | 0.2468 | 0.2476 | 0.2470 | 0.2472 | 0.2476 |
| 0.60 | 0.2102 | 0.2106 | 0.2107 | 0.2106 | 0.2108 |
| 0.70 | 0.1541 | 0.1542 | 0.1551 | 0.1555 | 0.1579 |
| 0.80 | 0.1271 | 0.1271 | 0.1271 | 0.1271 | 0.1299 |
| 0.90 | 0.0955 | 0.0955 | 0.0956 | 0.0956 | 0.0956 |
| 1.00 | 0.0707 | 0.0707 | 0.0707 | 0.0697 | 0.0692 |

**Experimenting with different values of b in BM-25:**

The following table shows the Average Precision , R-precision and Precision at 10, 30 docs for different values of b in BM-25, the bar chart for these values is below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **BM25 b = 0** | **BM25 b = 0.5** | **BM25 b=0.75** | **BM25 b =0.75** |
| **Average Precision** | 0.3265 | 0.3712 | 0.3703 | 0.3597 |
| **R-Precision** | 0.3202 | 0.3704 | 0.3502 | 0.3392 |
| **Precision at 10 docs** | 0.3058 | 0.3500 | 0.3558 | 0.3615 |
| **Precision at 30 docs** | 0.1917 | 0.2154 | 0.2186 | 0.2179 |

As we can observe from the above figure, the b value of 0.5 is the best, and b = 0.75 is second best. However, the Average precision is low for b=1 and b=0.

Following is the table for the interpolated recall-precision of different b values:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Interpolated recall-precision** | **BM25 b =0** | **BM25 b=0.5** | **BM25 b=0.75** | **BM25 b=1** |
| 0.00 | 0.7713 | 0.7908 | 0.8150 | 0.7858 |
| 0.10 | 0.6407 | 0.7111 | 0.7212 | 0.6883 |
| 0.20 | 0.4954 | 0.5724 | 0.5700 | 0.5476 |
| 0.30 | 0.4227 | 0.4818 | 0.4949 | 0.4839 |
| 0.40 | 0.3630 | 0.4275 | 0.4183 | 0.4143 |
| 0.50 | 0.3020 | 0.3676 | 0.3569 | 0.3499 |
| 0.60 | 0.2617 | 0.3014 | 0.2868 | 0.2909 |
| 0.70 | 0.1968 | 0.2263 | 0.2259 | 0.2292 |
| 0.80 | 0.1653 | 0.1899 | 0.1918 | 0.1837 |
| 0.90 | 0.1207 | 0.1315 | 0.1276 | 0.1205 |
| 1.00 | 0.0962 | 0.1002 | 0.0935 | 0.0872 |

Following is the graph showing the Interpolated average precision for different values of b:

As we can observe from the Figure,b =0.75 and b=0.5 give better Average Precision

Compare and Contrast your results with those of Project 1:

Following table shows the Trec evaluation on database 3 using CS6200 qrel file of Project 1

**Trec evaluation using CS6200qrel file(qrel.IRclass.10X1):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Okapi tf | Okapi tf\*df | Laplace Smoothing | Jelinek-Mercer | BM25 |
| AP | 0.2482 | 0.3113 | 0.2395 | 0.2483 | 0.2691 |
| RP | 0.2553 | 0.3125 | 0.2528 | 0.2623 | 0.2657 |
| P at 10 docs | 0.2640 | 0.3120 | 0.3120 | 0.2320 | 0.2800 |
| P at 30 docs | 0.2000 | 0.2560 | 0.2427 | 0.1973 | 0.2067 |

Comparing the Project 1 values with Final Project values:

**Project1:** Using the Average precision in Project 1, tf\*idf performed the best , next BM-25, next Jelinek Mercer, followed by Okapi-tf and Laplace smoothing.

**Final project:** Using the Average Precision values, Okapitf\*idf and BM-25 perform the best. Followed by Jelinek Mercer, Okapi-tf and then Laplace Smoothing.

However the AP values n Project 3 are higher than Project 1.

**Do the same retrieval formulae work best?**

In project 1 Okapi tf\*idf was the best followed by BM-25. In Project 3(Final Project), Okapi tf\*idf is still the best and BM-25 is the second best. Therefore in my case bot the retrieval formulae work best in both the projects.

**Do the optimal retrieval formulae parameters change?**

Yes, they do change. For example in BM-25, the different ‘b’ values give different results. In my case b value of 0.5 gave the best result.

**Why are the Average Precision values in Project 3 greater than Project 1?**

In the Final Project we have a collection of only 3204 documents, with relatively small document length. So it was very flexible for creating the inverted index, by processing the information in the documents.

Therefore running the retrieval Algorithms on the relatively small inverted index, gave better results.

In project 1 we had a huge collection and the size of the corpus also affected the evaluations.

Extra Credit Section:

I have done the following for Extra credit:

**1)Extra Retrieval Models:**

* Raw-tf
* Cosine tf\*idf model
* Witten Bell Smoothing
* Dirichlet Smoothing

**2)Compression using Lempel Ziv Algorithm**

**Extra Retrieval Models:**

The following table shows the Average Precision(non-interpolated), R-precision, Precision at 10 and 30 docs for the Retrieval Models:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Raw tf** | **Cosine tf\*idf** | **Witten-Bell** | **Dirichlet Smoothing** |
| **Average Precision** | 0.1729 | 0.3767 | 0.2656 | 0.2647 |
| **R-Precision** | 0.1606 | 0.3733 | 0.2647 | 0.2652 |
| **Precision at 10 docs** | 0.1654 | 0.3769 | 0.2558 | 0.2558 |
| **Precision at 30 docs** | 0.1128 | 0.2231 | 0.1596 | 0.1583 |

Fig: Bar chart showing the Average Precision(non-interpolated), R-precision and Precision at 10 and 30 docs for the Extra retrieval models.

The following table shows the Recall–precision values fro the different Models:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Interpolated recall-precision** | **Rawtf** | **Cosine-tf\*idf** | **Witten-Bell** | **Dirichlet-Smoothing** |
| 0.00 | 0.4792 | 0.7914 | 0.6432 | 0.6478 |
| 0.10 | 0.3638 | 0.7132 | 0.5186 | 0.5133 |
| 0.20 | 0.2615 | 0.5878 | 0.4158 | 0.4115 |
| 0.30 | 0.2320 | 0.5013 | 0.3589 | 0.3529 |
| 0.40 | 0.1726 | 0.4276 | 0.2911 | 0.2910 |
| 0.50 | 0.1509 | 0.3603 | 0.2476 | 0.2493 |
| 0.60 | 0.1290 | 0.2944 | 0.2120 | 0.2156 |
| 0.70 | 0.0965 | 0.2323 | 0.1552 | 0.1535 |
| 0.80 | 0.0849 | 0.1985 | 0.1272 | 0.1260 |
| 0.90 | 0.0645 | 0.1323 | 0.0955 | 0.0924 |
| 1.00 | 0.0545 | 0.0959 | 0.0707 | 0.0707 |

Following is the graph for Interpolated Average Precision of extra retrieval models

Fig: Graph of Average Precision (interpolated) for different retrieval models.

As we can observe from the above graph Cosine tf\*idf performs the best and raw-tf has extremely low Average precision. Witten Bell and Dirichlet Smoothing almost overlap.

**Raw-tf:**

In raw-tf the score of the documents is simply the sum of term frequencies of the query terms in that document. So this is the simplest of all the vector space retrieval models. As it only calculates the sum of the total count of the terms in a document , its results are the lowest.

**Cosine tf\*idf:**

The formula is as follows:

cosÕ = ( ¬doc \* ¬query)/(|| ¬doc||.||¬query||)

Here ¬doc is the document term vector and ¬query is the query term vector.

||¬doc|| is the document vector magnitude and ||¬query|| is the query vector magnitude.

For this Project I have used the tf\*idf values of the queries . For calculating the document magnitude I wrote a small script to collect the count of words in a document, add the squares of the counts and give its square root. I made a separate hash table for each docid as the key I have the Document magnitude as its value.

Since the Query magnitude is a constant it is ignored.

This retrieval function performs the best among the others as shown in the above graph of Average Precision. I have named the dictionary as docvectors.

**Witten-Bell Smoothing:**

The formula for Witten Bell smoothing is ((N/N+V)\* ML estimate)/ ((V/N+V)\*Background Probability)

V represents the number of unique words in a document and N represents the document length.

For calculating the number of unique words in a document I wrote a Python script to process the documents and count the number of unique words. I stored this a s a hash table with the docid as key and Number of unique words as the values. In my case I have named the dictionary as docunique.

**Dirichlet Smoothing:**

The formula for Dirichlet Smoothing is ((N/N+Mu)\* ML estimate)/ ((Mu/N+Mu)\*Background Probability)

N represents the document length, and Mu is a constant. I my case it is 2500.

Compression Experiment:

Using the algorithm provided on Rosettacode.org for Lempelziv compression, I compressed my whole inverted index file into a file called compressed.

However this increases the disk space of the compressed file. The original inverted index file occupies 938 KB on disk, but the compressed file occupies 946 KB. So this did not help.

The size may have increased due to the presence of delimiters(tab, newline) in the inverted index file.

This also requires decompressing the entire compressed file to read just a portion from the inverted index.

References:

<http://digitalhistoryhacks.blogspot.com/2006/08/easy-pieces-in-python-removing-stop.html>

<http://www.penzilla.net/tutorials/python/>

<http://docs.python.org/library/>

<http://rosettacode.org/wiki/LZW_compression#Python>

Discussed with:

Varun Ramachandran, Swetha Sasikumar, Rosina Joy and Trijito Santoso