Spark-based Information Retrieval System

Project Report

Zongming Yang
Haocheng Li

April 2017

Table of Contents

Introduce

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections. In this project, we need to build an spark based information retrieval system for given Reuter news XML file to search specific topic article.

The project has three stages, each stage is based on the previous stage's result.

In first stage, we build a Boolean retrieval system based on three indices: uniword inverted index, byword inverted index and uniword positional index. All documents and indexes are stored in HDFS in palmetto.

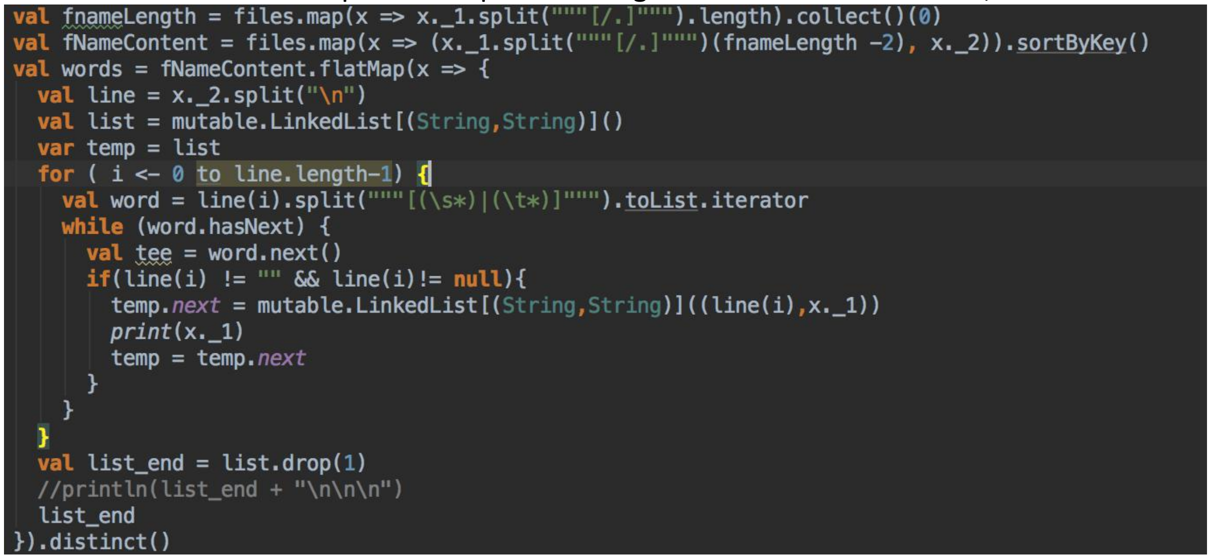
In second stage and third stage, we build a ranked retrieval system based on TF-IDF algorithm and another ranked retrieval system based on Okapi BM25 algorithm. We also create an interface for BM25 retrieval system to make it easier to access. Finally, we calculate the precision of retrieval results manually and evaluate the results of each stage to find some best retrieval method.

Stage 1

In this stage, we need to deployed Hadoop and spark in our palmetto account and create three basic inverted index for the Boolean retrieval system.

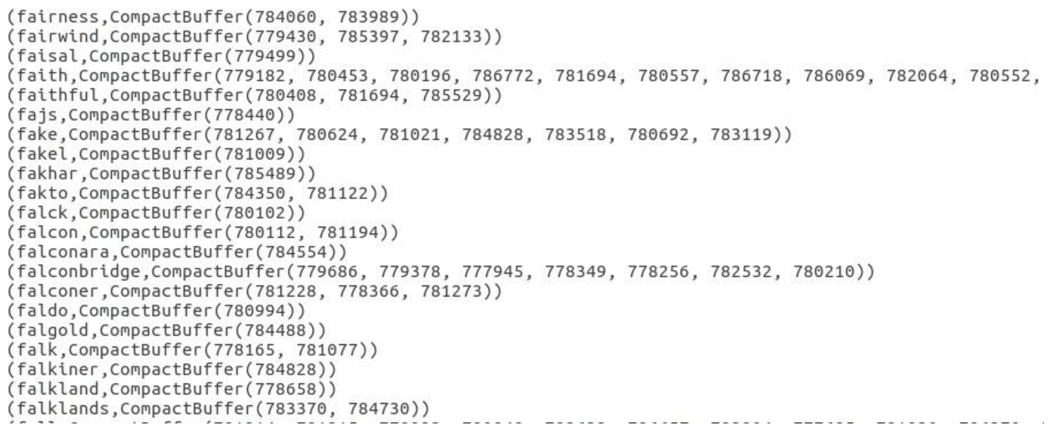
Uniword Index

We use scala to do map reduce operation to get uniword inverted index, the code is here:



```
val fnameLength = files.map(x => x._1.split("""[/.]""").length).collect()(0)
val fNameContent = files.map(x => (x._1.split("""[/.]""")(fnameLength -2), x._2)).sortByKey()
val words = fNameContent.flatMap(x => {
  val line = x._2.split("\n")
  val list = mutable.LinkedList[(String,String)]()
  var temp = list
  for ( i <- 0 to line.length-1) {
    val word = line(i).split("""[(\s*)|(\t*)]""").toList.iterator
    while (word.hasNext) {
      val tee = word.next()
      if(line(i) != "" && line(i)!= null){
        temp.next = mutable.LinkedList[(String,String)]((line(i),x._1))
        print(x._1)
        temp = temp.next
      }
    }
  }
  val list_end = list.drop(1)
  //println(list_end + "\n\n\n")
  list_end
}).distinct()
```

After converting all Reuter XML files to RDD in spark, first we add every (word, docID) record into a mutable list, then doing map reduce operation to the list to get the result like (word, docID1, docID2……). Finally, we invert the index to get inverted index. The format of result of inverted index is here:

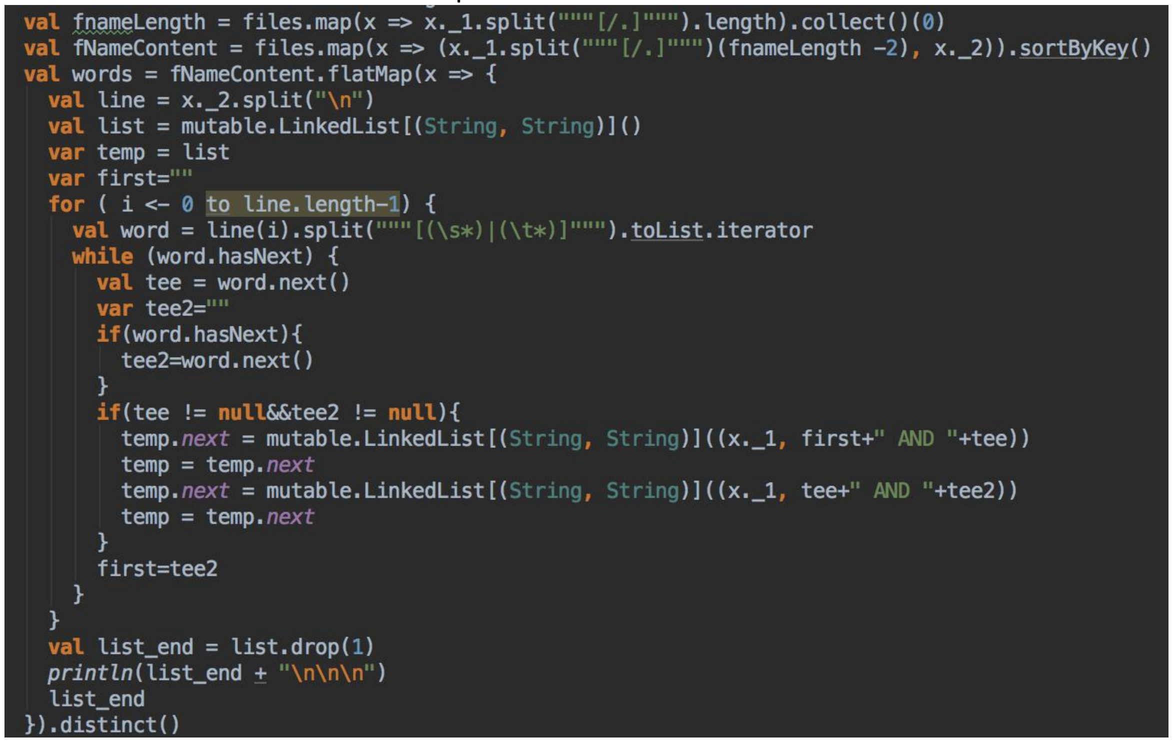


```
(fairness,CompactBuffer(784060, 783989))
(fairwind,CompactBuffer(779430, 785397, 782133))
(faisal,CompactBuffer(779499))
(faith,CompactBuffer(779182, 780453, 780196, 786772, 781694, 780557, 786718, 786069, 782064, 780552,
(faithful,CompactBuffer(780408, 781694, 785529))
(fajs,CompactBuffer(778440))
(fake,CompactBuffer(781267, 780624, 781021, 784828, 783518, 780692, 783119))
(fakel,CompactBuffer(781009))
(fakhar,CompactBuffer(785489))
(fakto,CompactBuffer(784350, 781122))
(falck,CompactBuffer(780102))
(falcon,CompactBuffer(780112, 781194))
(falconara,CompactBuffer(784554))
(falconbridge,CompactBuffer(779686, 779378, 777945, 778349, 778256, 782532, 780210))
(falconer,CompactBuffer(781228, 778366, 781273))
(faldo,CompactBuffer(780994))
(falgold,CompactBuffer(784488))
(falk,CompactBuffer(778165, 781077))
(falkiner,CompactBuffer(784828))
(falkland,CompactBuffer(778658))
(falklands,CompactBuffer(783370, 784730))
```

each line in this uniword inverted index represent a term and the ID of document where the term occurs in.

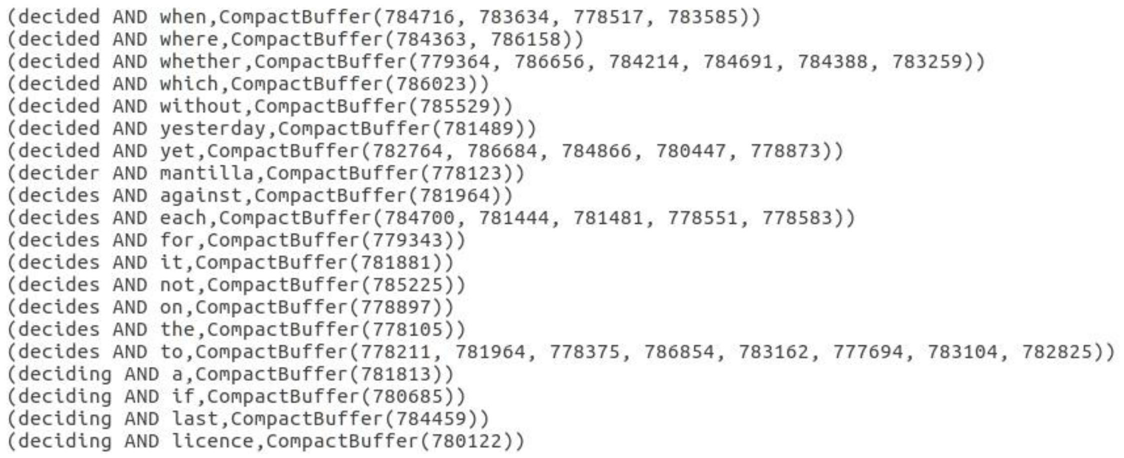
Biword Index

To get byword inverted index, first we need to get every two successive terms that occur in same document. The scala code of implementation is here:



```
val fnameLength = files.map(x => x._1.split("""[/.]""").length).collect()(0)
val fNameContent = files.map(x => (x._1.split("""[/.]""")(fnameLength -2), x._2)).sortByKey()
val words = fNameContent.flatMap(x => {
  val line = x._2.split("\n")
  val list = mutable.LinkedList[(String, String)]()
  var temp = list
  var first=""
  for ( i <- 0 to line.length-1) {
    val word = line(i).split("""[(\s*)|(\t*)]""").toList.iterator
    while (word.hasNext) {
      val tee = word.next()
      var tee2=""
      if(word.hasNext){
        tee2=word.next()
      }
      if(tee != null&&tee2 != null){
        temp.next = mutable.LinkedList[(String, String)]((x._1, first+" AND "+tee))
        temp = temp.next
        temp.next = mutable.LinkedList[(String, String)]((x._1, tee+" AND "+tee2))
        temp = temp.next
      }
      first=tee2
    }
  }
  val list_end = list.drop(1)
  println(list_end + "\n\n\n")
  list_end
}).distinct()
```

after getting two successive terms in documents, we add (docID, Term1 AND Term2) into mutable list when traverse all data files. After get all records, we use sortBykey to reduce all records according to their docID. Similarly, next step is inverting index and sort index by Term1 AND Term2. The result of byword inverted index looks like:



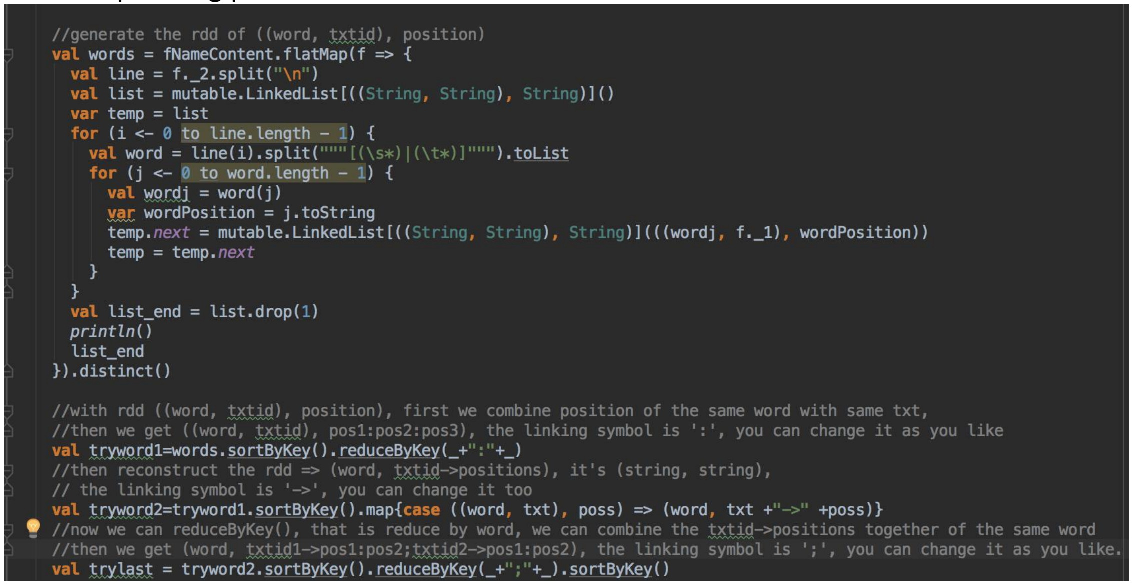
```
(decided AND when,CompactBuffer(784716, 783634, 778517, 783585))
(decided AND where,CompactBuffer(784363, 786158))
(decided AND whether,CompactBuffer(779364, 786656, 784214, 784691, 784388, 783259))
(decided AND which,CompactBuffer(786023))
(decided AND without,CompactBuffer(785529))
(decided AND yesterday,CompactBuffer(781489))
(decided AND yet,CompactBuffer(782764, 786684, 784866, 780447, 778873))
(decider AND mantilla,CompactBuffer(778123))
(decides AND against,CompactBuffer(781964))
(decides AND each,CompactBuffer(784700, 781444, 781481, 778551, 778583))
(decides AND for,CompactBuffer(779343))
(decides AND it,CompactBuffer(781881))
(decides AND not,CompactBuffer(785225))
(decides AND on,CompactBuffer(778897))
(decides AND the,CompactBuffer(778105))
(decides AND to,CompactBuffer(778211, 781964, 778375, 786854, 783162, 777694, 783104, 782825))
(deciding AND a,CompactBuffer(781813))
(deciding AND if,CompactBuffer(780685))
(deciding AND last,CompactBuffer(784459))
(deciding AND licence,CompactBuffer(780122))
```

each line in this biword inverted index represent two successive terms and the ID of document where these terms occurs in.

Uniword positional Index

The aim of uniword index is to get term's every position in every document so that we can find the if positions of all term in query are successive. If all positions of terms in query are successive and all terms occur in the same document, this document do contain this query so we can return this document as retrieval result.

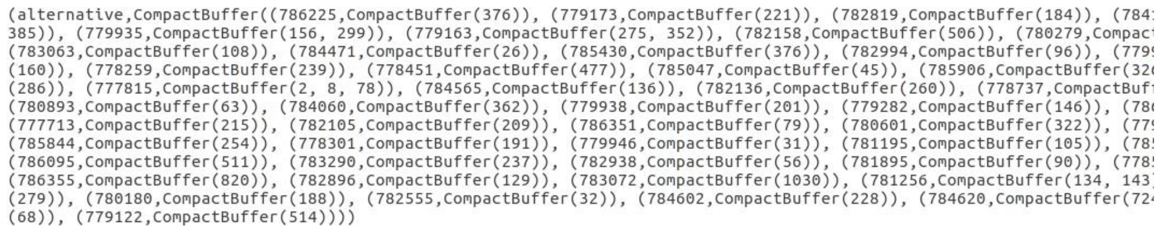
To implement the positional retrieval, we need to generate index that contains all terms and their corresponding positions. The code is here:



```scala
//generate the rdd of ((word, txtid), position)
val words = fNameContent.flatMap(f => {
  val line = f._2.split("\n")
  val list = mutable.LinkedList[((String, String), String)]()
  var temp = list
  for (i <- 0 to line.length - 1) {
    val word = line(i).split("""[(\s*)|(\t*)]""").toList
    for (j <- 0 to word.length - 1) {
      val wordj = word(j)
      var wordPosition = j.toString
      temp.next = mutable.LinkedList[((String, String), String)](((wordj, f._1), wordPosition))
      temp = temp.next
    }
  }
  val list_end = list.drop(1)
  println()
  list_end
}).distinct()

//with rdd ((word, txtid), position), first we combine position of the same word with same txt,
//then we get ((word, txtid), pos1:pos2:pos3), the linking symbol is ':', you can change it as you like
val tryword1=words.sortByKey().reduceByKey(_+":"+_)
//then reconstruct the rdd => (word, txtid->positions), it's (string, string),
// the linking symbol is '->', you can change it too
val tryword2=tryword1.sortByKey().map{case ((word, txt), poss) => (word, txt +"->" +poss)}
//now we can reduceByKey(), that is reduce by word, we can combine the txtid->positions together of the same word
//then we get (word, txtid1->pos1:pos2;txtid2->pos1:pos2), the linking symbol is ';', you can change it as you like.
val trylast = tryword2.sortByKey().reduceByKey(_+";"+_).sortByKey()
```

for each ((word, docID), position), we combine positions of same word in same document, then we change ((word, docID), position) to (word, docID, position). Finally, we reduce the list by word to get result. The format of final uniword positional index is
<word <txt1 pos1: pos2: pos3>, <txt2 pos1: pos2: pos3>...>:



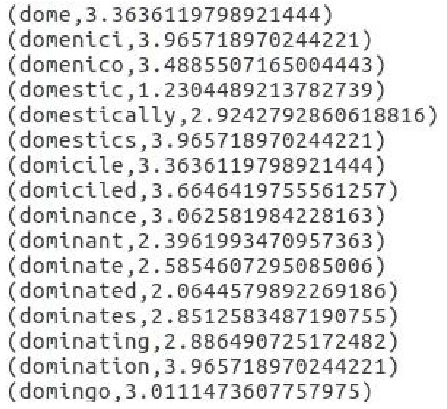
```
(alternative,CompactBuffer((786225,CompactBuffer(376)), (779173,CompactBuffer(221)), (782819,CompactBuffer(184)), (784
385)), (779935,CompactBuffer(156, 299)), (779163,CompactBuffer(275, 352)), (782158,CompactBuffer(506)), (780279,Compac
(783063,CompactBuffer(108)), (784471,CompactBuffer(26)), (785430,CompactBuffer(376)), (782994,CompactBuffer(96)), (779
(160)), (778259,CompactBuffer(239)), (778451,CompactBuffer(477)), (785047,CompactBuffer(45)), (785906,CompactBuffer(32
(286)), (777815,CompactBuffer(2, 8, 78)), (784565,CompactBuffer(136)), (782136,CompactBuffer(260)), (778737,CompactBuf
(780893,CompactBuffer(63)), (784060,CompactBuffer(362)), (779938,CompactBuffer(201)), (779282,CompactBuffer(146)), (78
(777713,CompactBuffer(215)), (782105,CompactBuffer(209)), (786351,CompactBuffer(79)), (780601,CompactBuffer(322)), (77
(785844,CompactBuffer(254)), (778301,CompactBuffer(191)), (779946,CompactBuffer(31)), (781195,CompactBuffer(105)), (78
(786095,CompactBuffer(511)), (783290,CompactBuffer(237)), (782938,CompactBuffer(56)), (781895,CompactBuffer(90)), (778
(786355,CompactBuffer(820)), (782896,CompactBuffer(129)), (783072,CompactBuffer(1030)), (781256,CompactBuffer(134, 143
(279)), (780180,CompactBuffer(188)), (782555,CompactBuffer(32)), (784602,CompactBuffer(228)), (784620,CompactBuffer(72
(68)), (779122,CompactBuffer(514))))
```

each line contains a term and an array of docID, each docID has an array of positions.

Stage 2

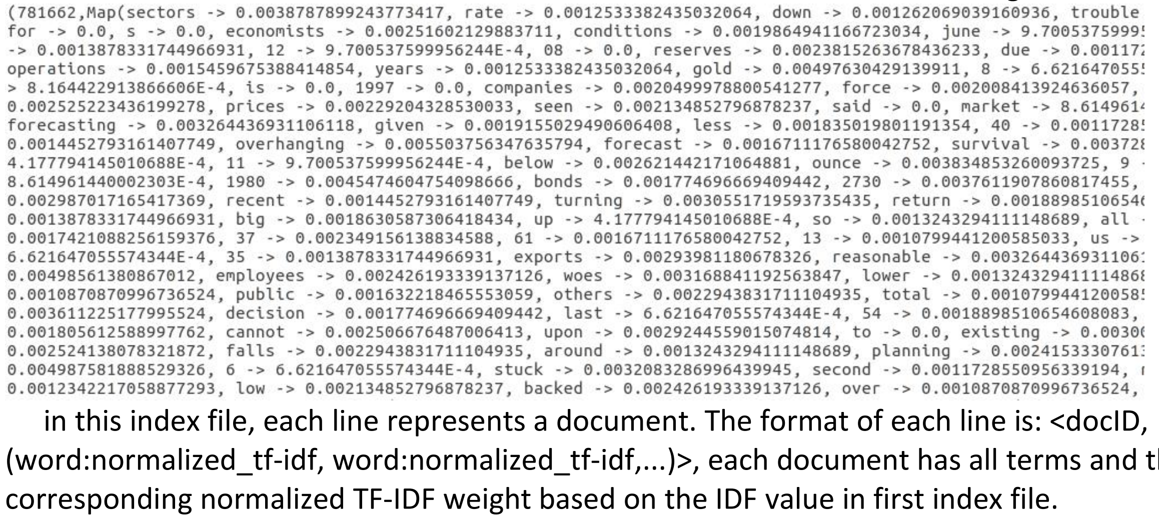
In this stage, we need to implement ranked retrieval system. We use ltc (logarithm-idf-cosine).ltc for query and documents to compute their weight. Basically, the weight of query is much easier to compute compared to the weight of terms in documents. For each document, first we need to calculate the TF-IDF weight of every term, then taking cosine normalization on each weight based on all terms weight in document.

Thus, we need two indexes file. The first index file contains all terms and IDF value in all documents. The format of this index is as below:



```
(dome,3.3636119798921444)
(domenici,3.965718970244221)
(domenico,3.4885507165004443)
(domestic,1.2304489213782739)
(domestically,2.9242792860618816)
(domestics,3.965718970244221)
(domicile,3.3636119798921444)
(domiciled,3.6646419755561257)
(dominance,3.062581984228163)
(dominant,2.3961993470957363)
(dominate,2.5854607295085006)
(dominated,2.0644579892269186)
(dominates,2.8512583487190755)
(dominating,2.886490725172482)
(domination,3.965718970244221)
(domingo,3.0111473607757975)
```

as you can see, each line contains a term and its corresponding IDF value. Based on this index file, we create second index file , which contains normalized TF-IDF weight of each term:



```
(781662,Map(sectors -> 0.0038787899243773417, rate -> 0.0012533382435032064, down -> 0.001262069039160936, trouble
for -> 0.0, s -> 0.0, economists -> 0.00251602129883711, conditions -> 0.0019864941166723034, june -> 9.700537599
-> 0.0013878331744966931, 12 -> 9.700537599956244E-4, 08 -> 0.0, reserves -> 0.0023815263678436233, due -> 0.00117
operations -> 0.0015459675388414854, years -> 0.0012533382435032064, gold -> 0.00497630429139911, 8 -> 6.621647055
> 8.164422913866606E-4, is -> 0.0, 1997 -> 0.0, companies -> 0.0020499978800541277, force -> 0.002008413924636057,
0.002525223436199278, prices -> 0.00229204328530033, seen -> 0.002134852796878237, said -> 0.0, market -> 8.614961
forecasting -> 0.003264436931106118, given -> 0.0019155029490606408, less -> 0.001835019801191354, 40 -> 0.0011728
0.0014452793161407749, overhanging -> 0.005503756347635794, forecast -> 0.0016711176580042752, survival -> 0.00372
4.177794145010688E-4, 11 -> 9.700537599956244E-4, below -> 0.002621442171064881, ounce -> 0.003834853260093725, 9
8.614961440002303E-4, 1980 -> 0.0045474604754098666, bonds -> 0.001774696669409442, 2730 -> 0.0037611907860817455,
0.002987017165417369, recent -> 0.0014452793161407749, turning -> 0.0030551719593735435, return -> 0.0018898510654
0.0013878331744966931, big -> 0.0018630587306418434, up -> 4.177794145010688E-4, so -> 0.0013243294111148689, all
0.0017421088256159376, 37 -> 0.002349156138834588, 61 -> 0.0016711176580042752, 13 -> 0.0010799441200585033, us ->
6.621647055574344E-4, 35 -> 0.0013878331744966931, exports -> 0.00293981180678326, reasonable -> 0.00326443693110
0.00498561380867012, employees -> 0.002426193339137126, woes -> 0.003168841192563847, lower -> 0.00132432941111486
0.0010870870996736524, public -> 0.001632218465553059, others -> 0.0022943831711104935, total -> 0.00107994412005
0.003611225177995524, decision -> 0.001774696669409442, last -> 6.621647055574344E-4, 54 -> 0.0018898510654608083,
0.001805612588997762, cannot -> 0.002506676487006413, upon -> 0.0029244559015074814, to -> 0.0, existing -> 0.0030
0.002524138078321872, falls -> 0.0022943831711104935, around -> 0.0013243294111148689, planning -> 0.00241533307613
0.004987581888529326, 6 -> 6.621647055574344E-4, stuck -> 0.0032083286996439945, second -> 0.0011728550956339194,
0.0012342217058877293, low -> 0.002134852796878237, backed -> 0.002426193339137126, over -> 0.0010870870996736524,
```

in this index file, each line represents a document. The format of each line is: <docID, (word:normalized_tf-idf, word:normalized_tf-idf,...)>, each document has all terms and their corresponding normalized TF-IDF weight based on the IDF value in first index file.

When doing search, first we compute the TD-IDF weight of terms in query and normalize these value, second we search these terms in second index file for every document. Finally, we multiply normalized query weight and document weight for each term to get the similarity result. Usually we pick top 20 results to the users.

Stage 3

This stage is document ranking based on the probability model. Here we use the BM25 model to implement the retrieval system.

Here we use the below equation:

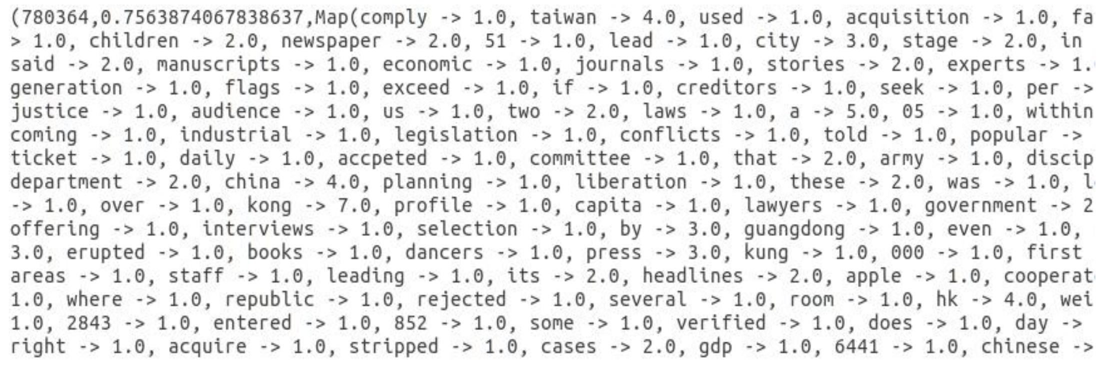
$$RSV_d = \sum_{t \in q} \left[\log \frac{N}{\mathrm{df}_t} \right] \cdot \frac{(k_1 + 1)\mathrm{tf}_{td}}{k_1((1-b) + b \times (L_d / L_{\mathrm{ave}})) + \mathrm{tf}_{td}} \cdot \frac{(k_3 + 1)\mathrm{tf}_{tq}}{k_3 + \mathrm{tf}_{tq}}$$

Tf means term frequency, and df means document frequency. And L_d means document length, L_{ave} means average document length. We can see in this equation there are 2 parameters, k1, b, k3.

In our implementation, we set k1=1.5, k3=1.5,b=0.75.

The BM25 model also take document length into consideration which can be seen from the length ration of Ld/Lave.

We firstly build two index. One is the (word, idf) index, which is same as first index in stage 2 so we can easily compute the logN/df in the equation. The other is <txt, length_ration,(tf)>. Here we just compute the length ration in advance to reduce the real query time. Here is the second index file:



```
(780364,0.7563874067838637,Map(comply -> 1.0, taiwan -> 4.0, used -> 1.0, acquisition -> 1.0, fa
> 1.0, children -> 2.0, newspaper -> 2.0, 51 -> 1.0, lead -> 1.0, city -> 3.0, stage -> 2.0, in
said -> 2.0, manuscripts -> 1.0, economic -> 1.0, journals -> 1.0, stories -> 2.0, experts -> 1.
generation -> 1.0, flags -> 1.0, exceed -> 1.0, if -> 1.0, creditors -> 1.0, seek -> 1.0, per ->
justice -> 1.0, audience -> 1.0, us -> 1.0, two -> 2.0, laws -> 1.0, a -> 5.0, 05 -> 1.0, within
coming -> 1.0, industrial -> 1.0, legislation -> 1.0, conflicts -> 1.0, told -> 1.0, popular ->
ticket -> 1.0, daily -> 1.0, accpeted -> 1.0, committee -> 1.0, that -> 2.0, army -> 1.0, discip
department -> 2.0, china -> 4.0, planning -> 1.0, liberation -> 1.0, these -> 2.0, was -> 1.0, l
-> 1.0, over -> 1.0, kong -> 7.0, profile -> 1.0, capita -> 1.0, lawyers -> 1.0, government -> 2
offering -> 1.0, interviews -> 1.0, selection -> 1.0, by -> 3.0, guangdong -> 1.0, even -> 1.0,
3.0, erupted -> 1.0, books -> 1.0, dancers -> 1.0, press -> 3.0, kung -> 1.0, 000 -> 1.0, first
areas -> 1.0, staff -> 1.0, leading -> 1.0, its -> 2.0, headlines -> 2.0, apple -> 1.0, cooperat
1.0, where -> 1.0, republic -> 1.0, rejected -> 1.0, several -> 1.0, room -> 1.0, hk -> 4.0, wei
1.0, 2843 -> 1.0, entered -> 1.0, 852 -> 1.0, some -> 1.0, verified -> 1.0, does -> 1.0, day ->
right -> 1.0, acquire -> 1.0, stripped -> 1.0, cases -> 2.0, gdp -> 1.0, 6441 -> 1.0, chinese ->
```

With these two indices built, we can easily compute the RSV for BM25 retrieval system.

Evaluation and Comparison

After constructing three different retrieval system, we use four queries to test these systems. We also evaluate the results based on manual work and compare these results. Here are the results of two queries of three stages.

For query "volkswagen"

Stage 1	Stage 2	Stage 3
584944	427215	299919
100853	157438	247092
332035	299973	311853
230389	247312	492742
265355	353262	300319
493491	304104	196089
306644	372072	459593
706114	454324	48284
248081	459593	50190
687278	25269	206461
388242	376426	21409
197235	376891	24364
109387	19758	23461
226441	508994	690536
299958	288007	114144
506756	503624	317465
355292	539549	197353
299542	481857	276840
615659	747904	161814
89056	416857	230757

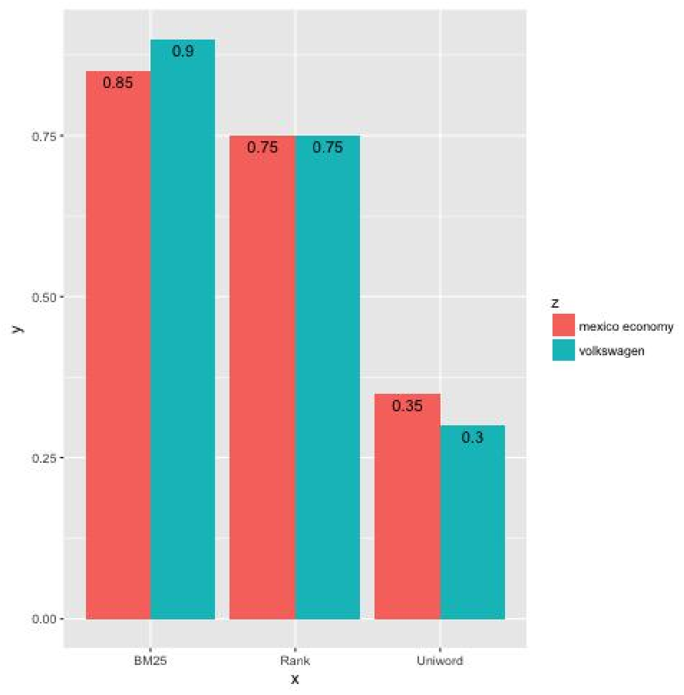
For query "mexico economy"

Stage 1	Stage 2	Stage 2
258484	378277	642999
96376	666495	633651
708637	327897	114131
31797	685801	224418
686442	105093	224147
81272	644047	749065
420949	738932	223994
574167	183191	224414
222861	793669	484260
165319	509147	387732
448780	315013	387764
749065	546517	601691
126886	285292	3325
188827	191638	39696
459522	586113	77783
117253	619975	31826
600947	531798	808301
50913	24175	116778
452880	134670	524498
509173	70254	426384

After manual test, we get the precision of these two queries based on three different stages:

precision	Stage 1	Stage 2	Stage 3
Volkswagen	0.3	0.75	0.9
mexico economy	0.35	0.75	0.85

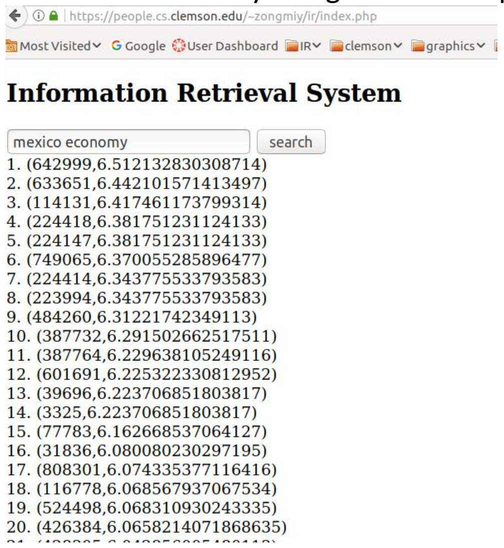
As we can see from table, the 30% precision of result of uniword retrieval system is low compared to the other two method. Also, the relevance between query and results of uniword is much less than the other two method because it is boolean retrieval. It seems that the results of stage 1 are randomly picked from dataset. In order to comparing these results directly, we make this graph:



As you can see from the figure above, the results of BM25 have highest precision and best quality. An interesting result is that, compared with BM25, the length of top results of ranked retrieval system is short so the results are kind meaningless. Because the key words' weight is larger after normalization when the length of document is shorter. Thus, the length of top 5 results of ranked retrieval system are short.

Thus, we draw conclusion that BM25 is the best in these three method.

After finishing the functional parts of stage 3, we build a webpage interface for users to access. We deploy our website on palmetto node based on PHP library. The webpage link to palmetto node directly using PHP techniques, here is our webpage:



Information Retrieval System

mexico economy

```
1. (642999,6.512132830308714)
2. (633651,6.442101571413497)
3. (114131,6.417461173799314)
4. (224418,6.381751231124133)
5. (224147,6.381751231124133)
6. (749065,6.370055285896477)
7. (224414,6.343775533793583)
8. (223994,6.343775533793583)
9. (484260,6.31221742349113)
10. (387732,6.291502662517511)
11. (387764,6.229638105249116)
12. (601691,6.225322330812952)
13. (39696,6.223706851803817)
14. (3325,6.223706851803817)
15. (77783,6.162668537064127)
16. (31836,6.080080230297195)
17. (808301,6.074335377116416)
18. (116778,6.068567937067534)
19. (524498,6.068310930243335)
20. (426384,6.0658214071868635)
```

Discussion

Based on the evaluation and comparison above, we can find that the results of Boolean retrieval system are too bad to implement for information retrieval. As for TF-IDF retrieval system in stage 2, it overweighs the short texts than long texts so the length of top results are usually short, sometime too short to provide useful information. The BM25 retrieval system is the best of these three methods, the results of BM25 are more meaningful than others. But it prefers long texts than short texts because of its limitation.

Also, after implementing retrieval systems of three stages, we find some ways to improve performance of retrieval system:

1. Save all indexed results as object file instead of TXT file. So program can read index directly without changing format
2. Split index files into several pieces, which can reduce the time of generating index files.
3. Uploading java library com. cotdp. hadoop. ZipFileInputFormat to read zip files directly instead of uncompressing all zip files into XML files, which make system more effective.

Using these methods, we can significantly improve the performance. After optimizing, the time search top 100 results of total 80K documents is just 10+ seconds while it nearly takes 1 minute to search without optimizing.