Retrieval Functions and Evaluations

HOMEWORK WRITEUP

Tianye Song | CS 6501 | 4/13/17

Question 1:

Ranking Algorithm Implementations:

<table>
<tr><td>Boolean Dot Product</td><td><pre>
    protected float score(BasicStats stats, float termFreq,
float docLength) {
        return 1;
    }
</pre></td></tr>
<tr><td>TFIDF Dot Product</td><td><pre>
    protected float score(BasicStats stats, float termFreq,
float docLength) {
        double firstComp = 1 + Math.log(termFreq);
        double secondComp =
Math.log((stats.getNumberOfDocuments() +
1)/(stats.getDocFreq()));
        double s = firstComp * secondComp;
        return (float)s;
    }
</pre></td></tr>
<tr><td>Pivoted Length Normalization</td><td><pre>
    protected float score(BasicStats stats, float termFreq,
float docLength) {
        double s = 0.75;
        double comp1, comp2, comp3;
        comp1 = (1 + Math.log(1 + Math.log(termFreq))) / (1 - s
+ s * docLength / stats.getAvgFieldLength());
        comp2 = 1;
        comp3 = Math.log((stats.getNumberOfDocuments() + 1) /
(stats.getDocFreq()));
        double result = comp1 * comp2 * comp3;
        return (float)result;
    }
</pre></td></tr>
<tr><td>Okapi BM25</td><td><pre>
    protected float score(BasicStats stats, float termFreq,
float docLength) {
        double k1=1.5, k2=750, b=1.0;
        double comp1, comp2, comp3;
        comp1 = Math.log((stats.getNumberOfDocuments() -
stats.getDocFreq() + 0.5) / (stats.getDocFreq() + 0.5));
        comp2 = ((k1+1) * termFreq) / (k1 * (1 - b + b *
docLength/stats.getAvgFieldLength()) + termFreq);
        comp3 = ((k2 + 1) * 1) / (k2 + 1); // here we assume
thatt c(w;q)=1
        double s = comp1 * comp2 * comp3;
        return (float)s;
    }
</pre></td></tr>
<tr><td>Jelinek Mercer</td><td><pre>
    protected float score(BasicStats stats, float termFreq,
float docLength) {
        double lambda = 0.1;
        double a,b;
        double pwc = model.computeProbability(stats), pml =
termFreq/docLength;
        a = (1-lambda) * pml + lambda * pwc;
        b = lambda * pwc;
        double result = Math.log(a/b);
        return (float)result;
    }
</pre></td></tr>
</table>

Dirichlet Prior	<pre> protected float score(BasicStats stats, float termFreq, float docLength) { docLen = docLength; double mu = 2500; // default double alphad = mu / (mu+docLength); double a,b; double pwc = model.computeProbability(stats); a = (termFreq + mu * pwc)/(mu + docLength); b = alphad * pwc; double result = Math.log(a/b); return (float)result; } public float getDocLen() { return docLen; }</pre>

To add the term $|q|\log(a_d)$ back to the Language Models, updated runSearch method in Searcher.java:

```
private SearchResult runSearch(Query luceneQuery, SearchQuery searchQuery)
    {
        try
        {
            System.out.println("\nScoring documents with " +
indexSearcher.getSimilarity().toString());
            Similarity sim = indexSearcher.getSimilarity();
            double len = 0; // have to do this to figure out query length in
the LM scorers
            if(sim instanceof JelinekMercer)
            {
                Set<Term> terms = new HashSet<Term>();
                luceneQuery.extractTerms(terms);
                ((JelinekMercer) sim).setQueryLength(terms.size());
                len = terms.size();
            }
            else if(sim instanceof DirichletPrior)
            {
                Set<Term> terms = new HashSet<Term>();
                luceneQuery.extractTerms(terms);
                ((DirichletPrior) sim).setQueryLength(terms.size());
                len = terms.size();
            }

            TopDocs docs = indexSearcher.search(luceneQuery,
searchQuery.fromDoc() + searchQuery.numResults());

            ScoreDoc[] hits = docs.scoreDocs;
```

```java
            if(sim instanceof JelinekMercer)
            {
             for(ScoreDoc hit : hits)
                    hit.score += len * Math.log(0.1); // add back

            }
            else if(sim instanceof DirichletPrior)
            {
                float docL = ((DirichletPrior) sim).getDocLen();
                for(ScoreDoc hit : hits){
                    double alphad = 2500 / (2500+docL);
                    hit.score += len * Math.log(alphad); // add back
                }
            }
            // sort
            boolean swapped = true;
            int j = 0;
            ScoreDoc tmp;
            while (swapped) {
                  swapped = false;
                  j++;
                  for (int i = 0; i < hits.length - j; i++) {
                        if (hits[i].score < hits[i + 1].score) {
                              tmp = hits[i];
                              hits[i] = hits[i+1];
                              hits[i+1] = tmp;
                              swapped = true;
                        }
                  }
            }
            //
            String field = searchQuery.fields().get(0);

            SearchResult searchResult = new SearchResult(searchQuery,
docs.totalHits);
            for(ScoreDoc hit : hits)
               {

               … // the rest code remains unchanged.

               }
```

Evaluation Function Implementations

Modified defaultNumResults to be 100000, so that all documents in the corpus are returned.

Average Precision

```
private static double AvgPrec(String query, String docString) {
		ArrayList<ResultDoc> results =
_searcher.search(query).getDocs();
		if (results.size() == 0)
			return 0; // no result returned
		
		HashSet<String> relDocs = new
HashSet<String>(Arrays.asList(docString.split(" ")));
		
		int i = 1;
		double avgp = 0.0;
		double numRel = 0;
		System.out.println("\nQuery: " + query);
		for (ResultDoc rdoc : results) {
			if (relDocs.contains(rdoc.title())) {
				numRel ++;
				avgp += (numRel / i);
				System.out.print("  ");
			} else {
				System.out.print("X ");
			}
			System.out.println(i + ". " + rdoc.title());
			++i;
		}
		// compute average precision here
		if (numRel == 0)
			return 0;
		
		avgp /= numRel;
		return avgp;
	}
```

P@10

```
private static double Prec(String query, String docString, int k)
{
		ArrayList<ResultDoc> results =
_searcher.search(query).getDocs();
		if (results.size() == 0)
			return 0;
		if (results.size() < k)
			k = results.size();
		
		HashSet<String> relDocs = new
HashSet<String>(Arrays.asList(docString.split(" ")));
		double p_k = 0;
		double numRel = 0;
		
		for(int i = 0; i < k; i++){
			if (relDocs.contains(results.get(i).title())){
				numRel ++;
```

<table>
<tr><td></td><td><pre>
                }
            }
            p_k = numRel / k;
            return p_k;
        }
</pre></td></tr>
<tr><td>Reciprocal Rank</td><td><pre>
private static double RR(String query, String docString) {
            ArrayList<ResultDoc> results =
_searcher.search(query).getDocs();
            if (results.size() == 0)
                return 0;
            HashSet<String> relDocs = new
HashSet<String>(Arrays.asList(docString.split(" ")));
            double relPosition = 0;
            double rr = 0;
            for (ResultDoc rdoc : results) {
                if (relDocs.contains(rdoc.title())) {
                    relPosition = results.indexOf(rdoc)+1;
                    break;
                }
            }
            if (relPosition == 0)
                return 0;

            rr = 1/relPosition;
            return rr;
        }
</pre></td></tr>
<tr><td>NDCG@10</td><td><pre>
private static double NDCG(String query, String docString, int k)
{
            ArrayList<ResultDoc> results =
_searcher.search(query).getDocs();
            if (results.size() == 0)
                return 0;
            if (results.size() < k)
                k = results.size();
            HashSet<String> relDocs = new
HashSet<String>(Arrays.asList(docString.split(" ")));
            double numRel = 0;
            double DCG = 0, IDCG=0;
            for(int i = 0; i < k; i++){

                if (relDocs.contains(results.get(i).title())){
                    DCG += 1/(Math.log(2+i) / Math.log(2));
                    numRel ++;
                }
                else {
                    DCG += 0/(Math.log(2+i) / Math.log(2));
                }
            }
            // calculate ideal DCG
            for(int i = 0; i<numRel; i++){
                IDCG += 1/(Math.log(2+i) / Math.log(2));
            }
            if (IDCG == 0)
</pre></td></tr>
</table>

	`` return 0; double ndcg = DCG / IDCG; return ndcg; } ```

Performances with Default Parameter Settings

	MAP	P@10	MRR	NDCG@10
Boolean Dot Product	0.23	0.29	0.59	0.62
TFIDF Dot Product	0.26	0.31	0.68	0.68
Pivoted Length Normalization	0.17	0.24	0.44	0.52
Okapi BM25	0.24	0.31	0.59	0.63
Jelinek-Mercer	0.28	0.34	0.68	0.69
Dirichlet Prior	0.20	0.24	0.54	0.55

Questions 2:

BM25 PARAMETER TUNING:

There are three parameters to tune for BM25 model: k1, k2 and b, where:

$k1 \in [1.2,2], k2 \in (0,1000), b \in [0.75, 1.2]$

In this case, I manipulate the three parameters sequentially, starting with k1.

First, try extreme values for k1, with k2 = 750 and b = 1:

K1	1.2	1.5	2
MAP	0.252	0.238	0.219

In this case, k1 = 1.2 yields max MAP.

Now try extreme values for k2 and see how MAP changes, with k1 = 1.2 and b = 1:

K2	1	750	1000
MAP	0.252	0.252	0.252

Surprisingly, the changes in k2 does not affect MAP in this case.

The last parameter to tune is b. Let's try corner cases for b, with k1 = 1.2 and k2 = 750

b	0.75	1	1.2
MAP	0.283	0.252	0.216

Seems like b = 0.75 gives the best MAP.

In summary, the parameter set with highest MAP yield is k1 = 1.2, b = 0.75, where k2 can be any real number R, such that R $\in$(0,1000).

DP (Dirichlet Prior) PARAMETER TUNING:

There is one parameter, mu, for DP. The homework instruction spells out specifically that empirically the value for mu lies between 2000 to 3000.

I start out with testing MAP values for extreme mu values, namely mu = 2000, as well as mu = 3000

Mu	2000	2500	3000
MAP	0.200	0.196	0.190

Thus, empirically speaking, with mu = 2000 the DP model gives the best MAP of 0.200.

Question 3:

With the optimal parameter setting for BM25 (k1 = 1.2, b = 0.75, k2 = 750), I obtain the following results:

	MAP	P@10	MRR	NDCG@10
All filters active	0.28	0.35	0.68	0.70
No lowercase Filter	0.28	0.35	0.68	0.70
No LengthFilter	0.28	0.35	0.68	0.70
No stopFilter	0.19	0.24	0.56	0.58
No PorterStemFilter	0.22	0.27	0.62	0.63
Only lowercase and length filters	0.28	0.35	0.68	0.70
No filters at all	0.14	0.19	0.48	0.51

For this given corpus, the filters stopFilter PorterStemFilter are very influential. Taking either one out leads to a noticeably decrease in the effectiveness of the BM25 retrieval model. However, when both stopFilter and PorterStemFilter are taken out, the model performs equally well as when all filters are active. In addition, when no filter is active, the model performs the worst comparing with all other scenarios.

In conclusion, the filters definitely improve model performance. Moreover, stopFilter and PorterStemFilter should either both be included, or both be excluded from the model for optimal performance.

Question 4:

Models compared	queries	Average precisions (tfidf vs bdp)
Tfidf vs bdp	measurement of plasma temperatures in arc discharge using shock wave techniques	1.0 vs 0.25
Tfidf vs bdp	variable ultra high frequency attenuators	0.59 vs 0
Tfidf vs BM25	measurement of plasma temperatures in arc discharge using shock wave techniques	1.0 vs 0.5
Tfidf vs BM25	methods of calculating instantaneous power dissipation in reactive circuits	0.83 vs 0.27
BM25 vs DP	characteristics of the single electrode discharge in the rare gases at low pressures	1.0 vs 0
BM25 vs DP	information on high current transistor switches	0.5 vs 0

For tfidf vs bdp: some words in the query, like plasma, arc and shock are rare in the documents. Tfidf model retrieves such rare terms better than bdp. This is because the term log((N+1)/df) magnifies the effect of rare terms on the score of a term.

Tfidf vs BM25: again, the tfidf model performs better with queries that have rare terms in them. BM25 accounts for rare term by the ln((N-df+0.5)/(df+0.5)) term, but penalize the effect. As a result, the tfidf model performs better.

BM25 vs DP:

Question 5:

???