Researching and Building IR systems on a Java-based IR Platform

WSM Project 2

何子安

資訊科學所

111753229

National Cheng Chi University

1. ABSTRACT

In this project, students are asked to implement several different retrieval methods and perform some experiments on the Web/Blog research collections (corpus) by using the open source toolkits that support research and development of Information Retrieval (IR). In this report, there will be showcases on the different results and analysis.

Keywords: Information Retrieval, Retrieval Model, Evaluation, Tuning, Relevance, Similarity

2. INTRODUCTION

2.1 Information Retrieval

Information Retrieval (IR) is the process of searching and retrieving information from a collection of documents or data. IR systems aim to index, analyze, and store documents so that users can quickly and easily find the relevant information they are looking for. IR systems have become increasingly important in recent years as they are used to power search engines and other web-based applications. Developing effective information retrieval models has been a long standing challenge in Information Retrieval, and significant progresses have been made over the years.

2.2 Goals

In this project, students are asked to implement/modify several different retrieval methods and perform some experiments by using well-known open source toolkit/platform, i.e. algorithms that given a user's request (query) and a corpus of documents assign a score to each document according to its relevance to the query. Some of these retrieval methods will be the implementation of the basic retrieval models studied in the class (e.g. TF-IDF, BM25, Language Models with different Smoothing).

2.3 Data

The WT2g data collection for this project is distributed by the University of Glasgow to

support research on information retrieval and related technologies. The collection contains Web documents, with a 2GB corpus. We will use the corpus to test the algorithms, and run experiments.

And the queries for our tasks is a set of 50 TREC queries for the corpus, with the standard TREC format having topic title, description and narrative. Documents from the corpus have been judged with respect to their relevance to these queries by NIST assessors.

3. RELATED WORK

3.1 Pre-tasks

In this case, we will need the toolkits which include search engines, browser toolbars, text analysis tools, and data resources that support research and development of information retrieval and text mining. These toolkits aim to help researchers compare different ways to query information easier and faster by providing ways of efficient retrieval functions. In order to achieve the goal, students are also required to understand how to use those toolkits by reading heavily on their documentations.

3.2 Tasks

3.2.1 For Corpus

For the WT2g corpus, students are asked to construct two indexes, (a) with stemming, and (b) without stemming. Both indexes contain stopwords. We can use the *porter* (Porter) or the *kstem* (Korvatz) stemmer for stemming. Below is an example of index information:

IndexID	Index Description	Statistics		
0	with stemming	terms=261,742,791	unique_terms=1,391,908	docs=247,491
1	without stemming	terms=261,742,791	unique_terms=1,526,004	docs=247,491

3.2.2 For Ranking Functions

Our task is to run the set of queries against the WT2g collection, return a ranked list of documents (the top 1000) in a particular format, and then evaluate the ranked lists. We are asked to implement the following variations of a retrieval system as below.

1. Vector space model, terms weighted by Okapi TF times an IDF value, and inner product similarity between vectors. We will have to use for the weights OKAPI TF * IDF where

   $$OKAPI\ TF\ =\ tf\ /\ (tf\ +\ 0.5\ +\ 1.5\ \times \frac{doclen}{avgdoclen})$$

   For queries, Okapi TF can also be computed in the same way, just use the length of the query to replace doclen. The definition of OKAPI TF is

   $$tf\ /\ tf\ +\ k1((1\ -\ b)\ +\ b\ \times \frac{doclen}{avgdoclen})$$

   In the above formula, set $k1\ =\ 2,\ b\ =\ 0.75$, can end up with the same idea.
   (code trace please refer to 'Appendix 1.1')

2. Language modeling, maximum likelihood estimates with Laplace smoothing only, query likelihood. Note that on multinomial models, for every document, only the probabilities associated with terms in the query must be estimated because the others are missing from the query-likelihood formula. For model estimation use maximum-likelihood and Laplace smoothing. Use formula (for term i)

   $$\rho_i = \frac{m_i+1}{n+k}$$

   where m=term frequency, n=number of terms in document (doc length) , k=number of unique terms in corpus.
   (code trace please refer to 'Appendix 1.2')

3. Language modeling, Jelinek-Mercer smoothing using the corpus, 0.8 of the weight attached to the background probability, query likelihood. The formula for Jelinek-Mercer smoothing is,

   $$\rho_i = \lambda P\ +\ (1\ -\ \lambda)Q$$

   where P is the estimated probability from document (max likelihood=$\frac{m_i}{n}$) and Q is the estimated probability from corpus (background probability=$\frac{cf}{terms\ in\ the\ corpus}$).

(code trace please refer to 'Appendix 1.3')

4. Implement any ideas to improve one of the above three IR models so that they can really improve the rank quality of the chosen IR models, and explain and showcase why these modifications can work.

   To improve the previous Vector Space Model, we may implement a feedback algorithm. Let R and $\overline{R}$ be the relevant and non-relevant document set. This algorithm simply "moves" the query vector closer to the centroid vector of R and away from the centroid vector of $\overline{R}$.

   $$q_{new} = q_{old} + \alpha d_R - \beta d_{\overline{R}}$$

   where d_R is the centroid vector of all weighted document vectors in R, and $d_{\overline{R}}$ that in $\overline{R}$.

   Motive: If we know that some documents are relevant to a query and some are not, we could be able to learn from the example documents to improve a query (Pseudo Feedback). @<*Notes on the Lemur TFIDF model. Chenxiang Zhai, 2001*>

   (code trace please refer to 'Appendix 1.4')

3.3 Extra

There are many conventional ranking models that need hyper-parameters, in order to get a reasonably good ranking performance. Nevertheless, the parameter tuning is far from trivial, especially considering that IR evaluation measures are non-continuous and non-differentiable with respect to the parameters. And machine learning has been demonstrating its effectiveness in automatically tuning parameters, combining multiple evidences, and avoiding overfitting. Therefore, it seems quite promising to adopt ML technologies to solve the aforementioned problems. @<*Learning to Rank for Information Retrieval, Tie-Yan Liu, 2009*>

That's why I try to implement some Learning-to-Rank retrieval models with standard and easy-to-use library. Conceptually, learning to rank consists of three phases:

1. identifying a candidate set of documents for each query
2. computing extra features on these documents
3. using a learning model to re-rank the candidate documents

3.4 Evaluation

Run all 50 queries and return at top 1,000 documents for each query. Do not return documents with a score equal to zero. If there are only N<1000 documents with non-zero scores then only return these N documents. Save the 50 ranked lists of documents in a single file. Each file should contain at most 50*1000 = 50,000 lines in it. Each line in the file must have the following format:

<query-number Q0 document-id rank score Exp>

where query-number is the number of the query (i.e., 401 to 450), document-id is the external ID for the retrieved document, rank is the rank of the corresponding document in the returned ranked list (1 is the best and 1000 is the worst; break the ties either arbitrarily or lexicographically), and score is the score that your ranking function outputs for the document. Scores should descend while rank increases. "Q0" (Q zero) and "Exp" are constants that are used by some evaluation software. The overall file should be sorted by ascending rank (so descending score) within ascending query-number. Run all four retrieval models against the two WT2g indexes which will generate 4 (models) * 2 (indexes) = 8 files, with at most 50,000 lines in total.

To evaluate a single run (i.e. a single file containing 50,000 lines or less), first download the *qrel file* for the WT2g corpus. Then use the evaluation tool (*ireval.jar*) in Lemur Toolkit or you can download the script of *trec_eval.pl* and run:

```
perl  trec_eval.pl  [-q]  qrel_file  results_file
```

trec_eval provides a number of statistics about how well the retrieval function corresponding to the *results_file* did on the corresponding queries and includes average precision, precision at various recall cut-offs, and so on.

4. RESULTS

4.1 Without Stemming and Remaining Stopwords

A) 1. Original TF-IDF, 2. Changed TF-IDF, 3. Lemur TF-IDF(modified model)

model	Recall@10	Precision@10	Precision@200	MAP	MAP@10	MAP@100	NDCG@100	Precision at R
1	0.148628	0.424	0.1026	0.240036	0.109276	0.213615	0.410719	0.291548
2	0.146765	0.406	0.0962	0.218836	0.103962	0.193756	0.388225	0.275651
3	0.137643	0.398	0.0987	0.225822	0.097015	0.198821	0.380935	0.270290

B) 1. BM25, 2. Hiemstra, 3. Dirichlet

model	Recall@10	Precision@10	Precision@200	MAP	MAP@10	MAP@100	NDCG@100	Precision at R
1	0.149539	0.428	0.1028	0.240074	0.111066	0.212931	0.406756	0.289685
2	0.108699	0.300	0.0895	0.186521	0.075063	0.160870	0.325105	0.232424
3	0.157051	0.454	0.1200	0.287661	0.120503	0.254720	0.446827	0.324640

C) 1. Re-rank(Dirichlet -> BM25), **2. LearningToRank(RandomForest)**, 3. LearningToRank(XGBoost)

model	Recall@10	Precision@10	Precision@200	MAP	MAP@10	MAP@100	NDCG@100	Precision at R
1	0.159474	0.458	0.0816	0.233738	0.123547	0.233738	0.422101	0.303166
2	**0.338517**	**0.940**	**0.1495**	**0.670546**	**0.337199**	**0.670053**	**0.769093**	**0.672199**
3	0.165583	0.460	0.1063	0.260117	0.125760	0.233210	0.440568	0.305627

4.2 With Stemming and Remove Stopwords

A) 1. Original TF-IDF, 2. Changed TF-IDF, 3. Lemur TF-IDF(modified model)

model	Recall@10	Precision@10	Precision@200	MAP	MAP@10	MAP@100	NDCG@100	Precision at R
1	0.151732	0.432	0.1205	0.259946	0.109068	0.222950	0.426088	0.314013
2	0.135986	0.410	0.1135	0.241770	0.098658	0.207276	0.408773	0.293293
3	0.128323	0.388	0.1159	0.240003	0.089670	0.200419	0.394980	0.282482

B) 1. BM25, 2. Hiemstra, 3. Dirichlet

model	Recall@10	Precision @10	Precision @200	MAP	MAP@10	MAP@100	NDCG@100	Precision at R
1	0.151732	0.432	0.1209	0.258559	0.108608	0.221041	0.422593	0.313925
2	0.103248	0.302	0.1018	0.194431	0.073939	0.159929	0.333360	0.221769
3	0.151239	0.450	0.1341	0.303469	0.119195	0.259979	0.469072	0.310013

C) 1. Re-rank(Dirichlet -> BM25), **2. LearningToRank(RandomForest)**, 3. LearningToRank(XGBoost)

model	Recall@10	Precision @10	Precision @200	MAP	MAP@10	MAP@100	NDCG@100	Precision at R
1	0.153752	0.460	0.0908	0.246076	0.120448	0.246076	0.444932	0.308065
2	**0.354433**	**0.960**	**0.1746**	**0.751290**	**0.352482**	**0.746346**	**0.831033**	**0.751959**
3	0.163500	0.466	0.1198	0.272107	0.118581	0.235235	0.448227	0.320876

For all models, Learning-to-Rank(RandomForest) model has the best results in two index scenario.

List of the future works:

- Learning to rank's model should have higher result, probably because of the lack of tuning.
- Learning to rank's methods need validation data, in this report, have not consider in this report.

5. ACKNOWLEDGMENT

This material is based upon work supported by the Prof. Tsai Ming Feng, TAs, and also shout out to many other Open-Source-Contributors.

6. REFERENCES

1. Terrier IR Platform, Information Retrieval Group, School of Computing Science, University of Glasgow, 2021.
2. Indri-5.20, Lemur Project, the Center for Intelligent Information Retrieval (CIIR) at the University of Massachusetts, Amherst, and the Language Technologies Institute (LTI) at Carnegie Mellon University, 2020.
3. Introduction to Information Retrieval, Cambridge University Press, Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, 2008.

7. APPENDICES

Appendix 1.1:

```
/**
* Uses TF_IDF to compute a weight for a term in a document.
* @param tf The term frequency of the term in the document
* @param docLength the document's length
* @return the score assigned to a document with the given
*                      tf and docLength, and other preset parameters
*/
public final double score(double tf, double docLength) {
        double Robertson_tf = k_1*tf/(tf+k_1*(1-b+b*docLength/averageDocumentLength));
        double idf = WeightingModelLibrary.log(numberOfDocuments/documentFrequency+1);
        return keyFrequency * Robertson_tf * idf;
}
```

Appendix 1.2:

```
/** Bayesian smoothing with Dirichlet Prior. This has one parameter, mu &gt
 */
public class DirichletLM extends WeightingModel {

	private static final long serialVersionUID = 1L;
	double mu = 2500d;

	/**
	 * Constructs an instance of DirichletLM
	 */
	public DirichletLM() {
		super();
		mu = 2500;
	}

	@Override
	public void prepare() {
		if (rq != null) {
			if (rq.hasControl("dirichletlm.mu")) {
				mu = Double.parseDouble(rq.getControl("dirichletlm.mu"));
			}
		}
		super.prepare();
	}

	@Override
	public double score(double tf, double docLength) {
		return WeightingModelLibrary.log(1 + (tf/(mu * (super.termFrequency / numberOfTokens))) )
+ WeightingModelLibrary.log(mu/(docLength+mu));
	}

	@Override
	public String getInfo() {
		return "DirichletLMmu"+mu;
	}

}
```

Appendix 1.3:

```
/**
 * This class implements the tf normalisation based on Jelinek-Mercer smoothing
 * for language modelling
  */
public class NormalisationJN extends Normalisation{

	private static final long serialVersionUID = 1L;
	/** The name of the normalisation method .*/
	protected final String methodName = "JN";
	/**
	 * Get the name of the normalisation method.
	 * @return Return the name of the normalisation method.
	 */
	public String getInfo(){
		String info = this.methodName+"lambda"+parameter;
		return info;
	}
	/**
	 * This method gets the normalised term frequency.
	 * @param tf The frequency of the query term in the document.
	 * @param docLength The number of tokens in the document.
	 * @param termFrequency The frequency of the query term in the collection.
	 * @return The normalised term frequency.
	 */
	public double normalise(double tf, double docLength, double termFrequency){
		if (docLength == 0)
			return tf;
		double tfn =
((1-parameter)*tf/docLength+parameter*(double)Nt/numberOfDocuments)*docLength;
		return tfn;
	}
}
```

Appendix 1.4:

```
/**
 * Uses LemurTF_IDF to compute a weight for a term in a document.
 * @param tf The term frequency in the document
 * @param docLength the document's length
 * @return the score assigned to a document with the given
 *	tf and docLength, and other preset parameters
 */
public final double score(double tf, double docLength) {
	double Robertson_tf = k_1*tf/(tf+k_1*(1-b+b*docLength/averageDocumentLength));
	return keyFrequency*Robertson_tf *
		Math.pow(WeightingModelLibrary.log(numberOfDocuments/documentFrequency), 2);
}
```