Name: Yan Zhao

Andrew ID: yanzhao2

11791 HW3 Report

Engineering and Error Analysis with UIMA

Running Result

Raw:

(MRR) Mean Reciprocal Rank ::MRR=0.4375

Total time taken: 0.935

Final:

(MRR) Mean Reciprocal Rank ::MRR=0.6458

Total time taken: 0.973

Error Analysis

Error Analyze

I check the error of ranks in different queries and try to find the reason for the errors. Here are a few examples showing the way I identify the error types.

qid=12 rank=3 What is the height of the tallest redwood?

Cos1: 0.31622776601683794

Cos2: 0.5720775535473553 Mendocino Tree is the tallest redwood in the world.

Cos3(rel=1): 0.3834824944236852 Cos4: 0.29814239699997197

The high cosine similarity of document 2 and 3 is affected significantly by syntactic expletive word "the".

qid=13 rank=3 How deep is Crater Lake?

Cos1: 0.11952286093343936 Cos2: 0.26967994498529685

Cos3(rel=1): 0.1348399724926 Oregon's Crater Lake tops it at 1,932 feet at its greatest depth.

Unable to match “depth” to “deep”.

qid=20 rank=2 What is the Keystone State?

Cos1: 0.3077935056255462

Cos2(rel=1): 0.42163702136 They call it the Keystone State, ….

Cos3: 0.14142135623730948

Unable to match “State.” to “State?”

Error Type

According to table above, we can conclude a few types of errors.

1. Format Errors: errors affected by tense of vocabulary, or letter case in vocabulary like “Sorrow” and “sorrow”.

2. Punctuation Errors: errors affected by punctuations like “?” in query and “.”. Specifically, one word may be identified as two because of punctuations.

3. Stop words Errors: words that have no meaning, but since their commonly existing in documents, they will influence our ranking significantly.

4. Text size Errors: lower cosine similarity because of bigger denominator affected by the length of sentence in doc file.

The statistical data of these four types of errors can be seen in the following table.

Error Types	QueryId	Total Number
Format errors	1, 3, 5, 6, 8, 9, 11, 15, 16, 17, 19	11
Punctuation errors	4,6, 7, 8, 9, 13, 14, 15, 17, 19, 20	11
Stop words errors	1, 3, 12, 13, 14,	5
Text size errors	2, 15, 18	3

Improvement

Implemented Improvements

For different types of errors, I use different ways to escape for them.

1. For format errors, I use Stanford Lemmatizer to improve the performance. This is a library that contain a function called *stemText()*, which can change different tenses of one word into its original format. Meanwhile, this function can also change all the upper letters in one word to lower letters. I used this function in Document Vector Annotator just before I add the word in different documents into token types. After using this method, I improve the permanence of the system, changing MRR coefficient from 0.4375 up to 0.5500.

The code I used to remove format errors can be seen as follows.

```
/*
 * use Stanford lemmatizer to transform different types of tokens
 */
word = StanfordLemmatizer.stemText(word);
```

2. For punctuation errors, I use Stanford PTBtokenizer to identify words with punctuations and remove the punctuations. This library can split words away from punctuations, thus can remove punctuation in tokens. I use this library in the given function tokenize0() when split the documents into words. After using this method, I continuously improve the permanence of the system, changing MRR coefficient from 0.5500 up to 0.6125.

The code I used to remove punctuation errors can be seen as follows.

```
List<String> tokenize(String doc) {
  List<String> res = new ArrayList<String>();
  TokenizerFactory factory = PTBTokenizerFactory.newTokenizerFactory();
  // split based on white-space(one or more)
  for (String s: doc.split("\\s+")) {
      Tokenizer tokenizer = factory.getTokenizer(new StringReader(s));
      Object temp = tokenizer.peek();
      s = temp.toString();
      if (!s.matches("['a-zA-Z0-9_]*$") )
          continue;
//      System.out.print(s + "  ");
      res.add(s);
  }

  return res;
}
```

3. For stop words errors, I store the file *stopwords.txt* which contain nearly all kinds of stop words into hash map. I used this hash map in Document Vector Annotator just before I add words in different documents into token types. Actually, if one word is seemed as stop words, I would consider it as meaningless and choose not to add it to the vector space(tokens) for further calculation. After using this method, I continuously improve the permanence of the system, changing MRR coefficient from 0.6125 up to 0.6458.

The code I used to remove stop words errors can be seen as follows.

```
HashMap<String,Integer> stopWords = new HashMap<String,Integer>();
file = new File("src/main/resources/stopwords.txt");

BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(file));
    } catch (FileNotFoundException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
    String temp = null;
    try {
        while ((temp = reader.readLine()) != null) {
            System.out.print(temp);
            stopWords.put(temp + " ", 1);
        }
    } catch (IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
    try {
        reader.close();
    } catch (IOException e) {
        // TODO Auto-generated catch block
if (stopWords.containsKey(word)) {
   System.out.println("df");
    continue;
}
```

After all the steps above, I increase the MRR coefficient significantly. Also, I choose different similarity measures including the Dice coefficient and the Jaccard coefficient. Jaccard coefficient equals to $|A \cap B|/|A \cup B|$, where A and B are two vector space contains the frequency of words in a sentence. Similarly, Dice coefficient equals to $2|A \cap B|/(|A|+|B|)$, where A and B are two vector space contains the frequency of words in a sentence. Using Jaccard coefficient, the MRR decrease a little bit from 0.6458 (using Cosine similarity) to 0.6292. And for Dice coefficient, it has

almost the same performance as Jaccard coefficient. Thus, it seems that Cosine similarity might has a better performance on calculating the similarity between two vector space.

The code I used to implement Dice and Jaccard coefficient can be seen as follows.

```
private double computeJaccardSimilarity2(Map<String, Integer> queryVector,
        Map<String, Integer> docVector) {
      double jaccard_similarity=0.0;
      int aAndB = 0;
      int aOrB= 0;

      Iterator<Entry<String, Integer>> it = queryVector.entrySet().iterator();
      while (it.hasNext()) {
        Map.Entry<String, Integer> queMap = it.next();
        if (docVector.containsKey(queMap.getKey())) {
            aAndB ++;
        }
      }
      aOrB = queryVector.size() + docVector.size() - aAndB;
      jaccard_similarity = (double) aAndB / (double) aOrB;
      return jaccard_similarity;
    }
private double computeDiceSimilarity2(Map<String, Integer> queryVector,
      Map<String, Integer> docVector) {
    double dice_similarity=0.0;
    int aAndB = 0;
    int aOrB= 0;

    Iterator<Entry<String, Integer>> it = queryVector.entrySet().iterator();
    while (it.hasNext()) {
      Map.Entry<String, Integer> queMap = it.next();
      if (docVector.containsKey(queMap.getKey())) {
          aAndB ++;
      }
    }
    aOrB = queryVector.size() + docVector.size();
    dice_similarity = (double) 2 * aAndB / (double) aOrB;
    return dice_similarity;
  }
```

Unimplemented Improvements

Except for the method I have implemented, there are other effective ways to improve the performance. One is called tf-idf weight. In this method, the frequency of one word t in a sentence can be represented by the multiple of two coefficients. One is called tf, $w_{tf} = 1+\log_{10}tf$, where tf is the frequency of word t in a document. The other is called df, which equals to $idf_t = \log10(N/df_t)$, where N is the number of documents and df_t is the number of documents that contain the word t. This method is based on the theory that rare terms are usually more informative than frequency terms, so in the calculation of similarity, it makes the rare terms have higher

weights. After altering the frequency of words into tf-idf weight, we might improve the performance using cosine similarity.

Architecture

The design of this system shows as follow.

The Document Reader identifies the sentence with its query id, relevant scores and contents. DocumentVectorAnnotator identifies the word and word frequency in each sentence, and store the token list back into document type. RetrievalEvaluator computes the similarity and performance metric of the collection.

For Document Reader I doesn't change its code. For DocumentVectorAnnotator, I improve white-space based tokenize0 to split every sentence to construct a word without punctuations, and then utilize the format of words as well as remove stop words to update the tokenList in the Document Type.

For the RetrievalEvaluator, I use two array list to store query id and relevant scores. And I use one array list to store the rank of different right answers for different query id, thus can calculate the MRR more easily. Then for the term frequency, I use an array list which has a type of hash map, to store different documents. I use ProcessCas() to calculate the MRR and store them in output files, in this function, I used a number of different functions to help calculating, which can be seen in the following UML Diagram. I design the system in this way so that I can change any algorithm to calculate similarity without changing the ProcessCas() function, just modifying the sub function can work.

UML Diagram