Your Name: Junjia He

Your Andrew ID: junjiah

Homework 3 Report

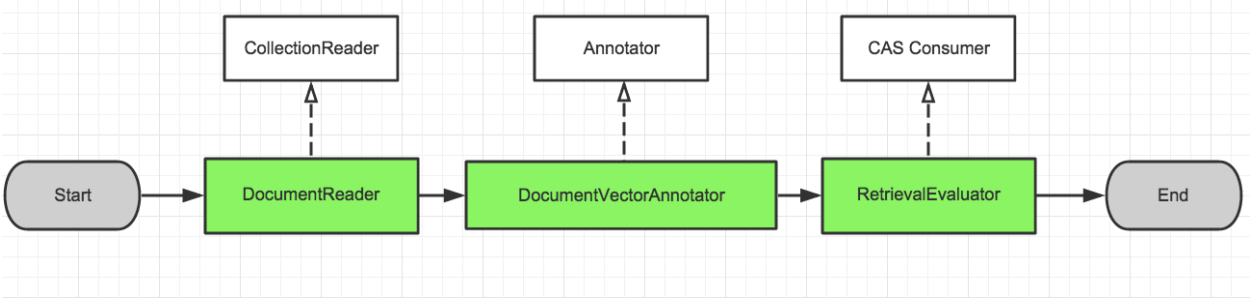
1. Statement of Assurance

You must certify that all of the material that you submit is original work that was done only by you. If your report does not have this statement, it will not be graded.

I hereby certify that all of the material that I submit is original work that was done only by me.

2. System Architecture

2.1 Pipeline Architecture



In this pipeline *DocumentReader* reads one document a time (both the query and the document), parses the document and extracts useful information such as relevance and query ID and stores them into CAS.

Then the *DocumentVectorAnnotator* tokenizes the passed documents and records each token's frequency. The token information is also stored into CAS.

Finally the *RetrievalEvaluator* computes the similarity between the query and the document, ranks the document according to their scores and outputs the results conforming to the specified format.

2.2 Implementation

The implementation in *DocumentReader* and *DocumentVectorAnnotator* is trivial. However in *RetrievalEvaluator*, I tried to **decouple** the CAS consumer (processing CAS, control output format, etc.) and the real evaluation process (computing cosine similarity, ranking the documents, etc.). Therefore I created a utility class called ***QueryProcessor***, which is responsible solely for the evaluation parts while keeping *RetrievalEvaluator* simple.

QueryProcessor only has three public APIs:

1. The constructor.
2. *processDocument(Document):* After *RetrievalEvaluator* processes the CAS and extracts the document it would send the document to **QueryProcessor** via this API. **QueryProcessor** would sequentially process the documents, calculate the similarity, and save the score. Once it receives a query, **QueryProcessor** would rank the stored documents and save the rank of the relevant document.
3. *generateReport():* In this method ***QueryProcessor*** simply outputs the saved ranking of relevant documents along with other required information.

In this way the code structure became much clearer, and if I need to change the similarity measure the only thing I need to do is to modify ***QueryProcessor*** without touching the CAS consumer.

3. Error Analysis and Iterative Improvement

3.1 First Round - stop words

After checking the *report.txt* generated by Taks 1, I found all the ranking errors (*i.e.* in which the relevant document didn't rank first) are caused by the same reason, that is for each query, the document which has most the same stop-words (such as *the, and, where* and so forth) as in the query is always ranked higher. So my first strategy to improve the results is to filter the stop words, which is implemented in tokenization using the provided *stopwords.txt*.

Result Mean Reciprocal Rank: 0.4342 → 0.4912

3.2 Second Round - stemming

Similarly I analyzed the new results, and indeed some relevant documents are now ranked first, but there are still many errors. I classified the errors into following types along with their counts.

- Word mismatch
  - Punctuation in word: 4 (like *china"*, *china]* in the token list)
  - Multiple representation of the same word: 3 (like *moves* and *move*)
- Semantics: 7 (where the relevant answer has few words in common with the query)

Then I used Stanford lemmatizer to stem the words to their canonical form, which gives me following results:

Result Mean Reciprocal Rank: 0.4912 → 0.5439

3.3 Third Round - better tokenization

Obviously in the next step I tried to eliminate the punctuations in words, where the default Java `StringTokenizer` class comes to help. I filtered the punctuations from the word and ran again, producing following performance improvement:

Result Mean Reciprocal Rank: 0.5439 → 0.6009

3.4 Fourth Round - tf-idf

Then I tried to improve the retrieval model. Since the default is a vector space model, I tried to incorporate the inverse document frequency to better distinguish the relevant documents.

The cosine similarity I chose is lnc.lnc model:

$$\frac{\sum d_i \cdot q_i}{\sqrt{\sum d_i^2} \cdot \sqrt{\sum q_i^2}} = \frac{\sum (\log(tf)+1) \cdot \left((\log(qtf)+1) \log \frac{N}{df}\right)}{\sqrt{\sum (\log(tf)+1)^2} \cdot \sqrt{\sum \left((\log(qtf)+1) \log \frac{N}{df}\right)}}$$

However the results didn't improve a lot.

Result Mean Reciprocal Rank: 0.6009 → 0.6140

Then I checked again the error types, I realized this is not a typical search problem but a *machine understanding* problem. Most of the *Semantic* errors happened because the answer (relevant document) expressed the desired information in a different way, which cannot be captured by term frequency-based retrieval model. Therefore I doubt whether advanced models would improve the results, especially in our circumstance where we lack the whole documents and inverted lists. Maybe incorporating external information such as synonyms or expanding the query terms using pseudo relevance feedback could improve the results in this task, but I didn't try them.