Yueran Yuan (yuerany)
11791 HW 4 Writeup

OVERALL PROJECT PIPELINE

The pipeline that I modified consists of 2 major parts. An annotation section which tokenizes the document and refines the tokens and an evaluation section which computes similarity scores between query and documents and computes MRR. Detail of the system design decisions is covered in a later section. The following is a brief description of each phase.

Annotation Section

The DocumentVectorAnnotator is the entry point into this section of the pipeline. In this section of the pipeline, I use the stanford tokenizer to create tokens. These tokens are then processed with a number of independent refinement processes. Each process is simple and could be switched on and off with the UIMA parameters for the annotator. In my initial implementation, I had no refinement processes and the processes were gradually added as I did error analysis to try to improve my results. More detail on the processes are described in later sections

Evaluation Section

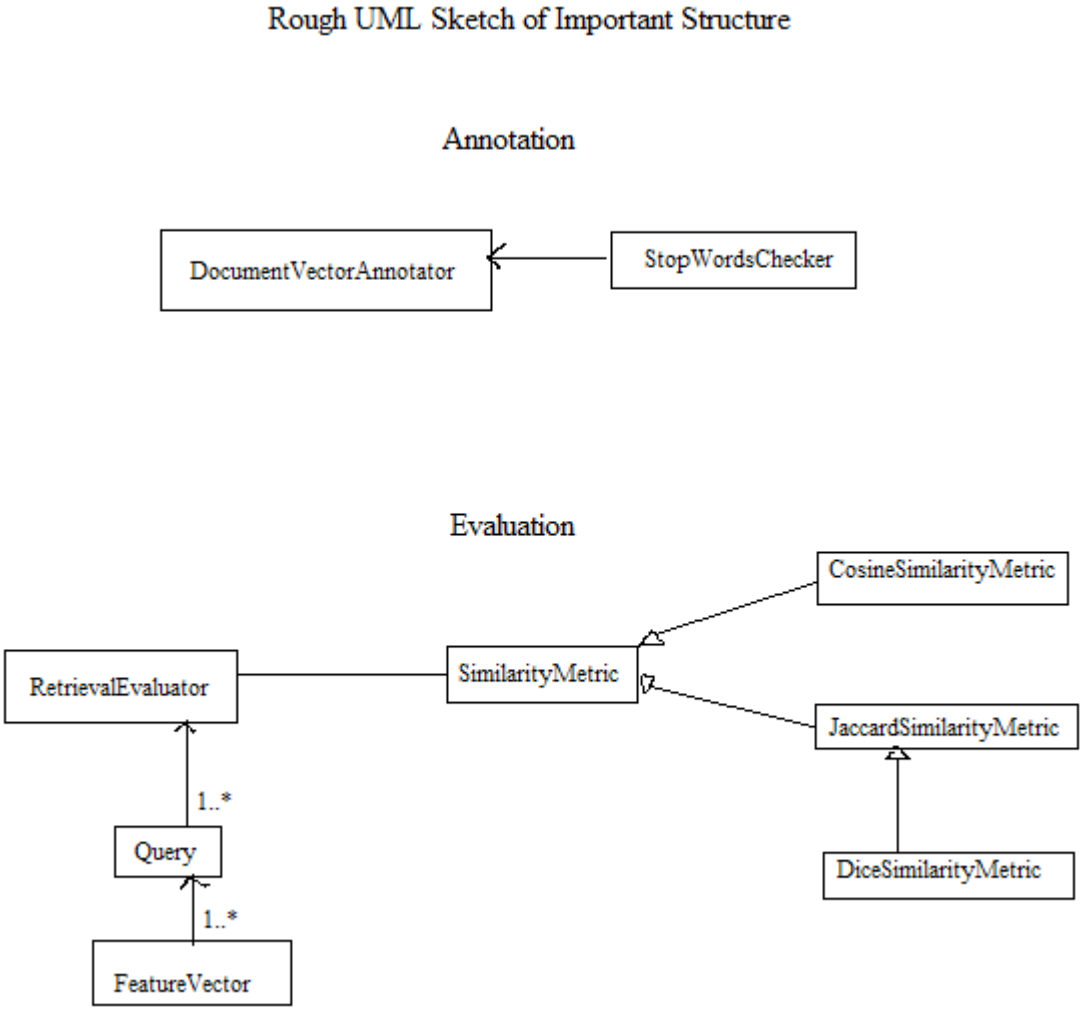
The Evaluation section consists of 2 steps: similarity scoring and MRR. During the similarity scoring phase, I compute the rank. There are 3 possible similarity scoring metrics that I implemented (Dice, Jaccard, and Cosine). These metrics could be set with the SimilarityMetric parameter for the UIMA annotator.

I want to describe briefly how I was able to compute the rank for MRR without sorting the answers. First I computed the similarity score for the correct answer. I then computed the similarity scores for the incorrect answers, keeping track of the fact that each incorrect answer with a higher score than the correct answer pushes the rank of the correct answer down.

As I wrote on a Piazza post asking about this method, "this is superior to sort-based ranking in 2 manners. (1) complexity of rank determination is O(n) and not O(n^2). And (2) it is a deterministic ranking algorithm. Score collisions are resolved deterministically instead of depending on the arbitrary order by which the answers appear in the document (assuming the sorting algorithm is in place. If the sorting algorithm is not in place, collision resolution is random)."

SYSTEM DESIGN DECISIONS

A rough UML diagram of the structure is shown below



I want to describe a couple of notable decisions.

1) In the evaluation phase, I converted the data contained in document annotations into a structured form (Query and FeatureVector) as they were sent to the RetrievalEvaluator. This is because the structure of the UIMA annotations is not particularly good for doing the manipulation I want to do (there's a lot of boilerplate involved in removing information etc. and look up is not efficient since everything has to be looped through). I stored each query and its answers in a structured form (i.e. with fields for the correct answer, query, and incorrect answers) inside a map indexed by the query_id. In addition, FeatureVector converts the tokenList into a Map. This again allows look-up and comparison between different tokenList to be easy and decouples my similarity code from the UIMA FSList structure.
2) I factored SimilarityMetrics into their own class. I mainly wanted to separate this logic from the evaluation so that evaluation could have less responsibility. Critically, because Jaccard and Dice are very similar, I had Dice inherit from Jaccard. As you can see in the code, the code for Dice is very short.

3) In the annotation stage, I made each token refinement processing set (e.g. stemming, lowercasing) its own function. I did this so that they can be turned on and off independently to tune the pipeline. I also made the StopWordsChecker a separate class (to again separate logic) and the stopwords list can be sent as a parameter so that it could be altered easy to try different combinations of settings.

IMPROVING THE SYSTEM

note: the speed report are representative samples (picked with my judgement as criteria) because speed changes with every run.

Tokenize

When I initially ran the system without any refinement for the tokens (i.e. just tokenization). This resulted in an accuracy of 0.77. From an error analysis, I could see that "If you see a friend without a smile, give him one of yours " was being penalized because it had too many words in it (increasing the divisor) so that even though its content was more relevant, it scored worse than the other answers. To deal with this, I added stopwords.

Tokenize + Google Stopwords

I used the stopwords already included in the resources folder (I think those are the 'google stopwords'). I added punctuation to this stopwords list for completeness. This gave a 0.8 error. I see that at "If you see a friend without a smile, give him one of yours " moved up by one. I realized at this point that "If" was not being recognized as a stop word because it was capitalized and was contributing to the length of the vector, lowering its score. I then lowercased all my words.

Tokenize + Google Stopwords + Lowercase

I saw that this did not improve performance, though it did solve the problem I noticed. I now noticed another issue "friend" from "If you see a friend without a smile, give him one of yours " was not being matched to the plural "friends" which was used in the query. If the system recognized that those were the same features, the score for that sentence would be higher. I therefore decided to add stemming

Tokenize + Google Stopwords + Lowercase + Stemming

I stemmed the words using the Stanford stemmer. I expected the pipeline to become slower, but it only slowed by 0.03 seconds (from 0.97 to 1.0) so it wasn't really a major problem. I saw that performance had now increased to 0.9 and that the "If you see a friend without a smile, give him one of yours " sentence that I had been focusing on had reached rank 1. I now focus on the remaining error: "The best mirror is an old friend".

I notice that the reason that the incorrect sentence which ranks better is "My best friend is the one who brings out the best in me." I notice that the reason that this sentence has a higher rank is due to "one" being recognized as a feature. In all these cases, "one" is effectively used as a pronoun but it's not in the stopwords list. Looking through the stopwords list, I realize that a couple other critical things were missing (e.g. "whose"). I decided to go online to look for a more complete stopwords list and found this: http://dev.mysql.com/doc/refman/5.6/en/fulltext-stopwords.html

Tokenize + MySql Stopwords + Lowercase + Stemming

After I used this set of stopwords, I had 1.0 accuracy. Luckily, the larger stopword list did not slow down the system by much (1.0 to 1.05 seconds). Though there were more sophisticated things I could

add (e.g. ngrams, coreference resolution) that I'd already implemented in earlier homeworks, I decided not to add them at this point because there's no need to slow down the system further since it was already accurate enough.

BONUS

I implemented Jaccard and Dice similarities. Notably, the way I implemented them tests for the *presence* of the feature (i.e. whether the feature is positive), it doesn't account for when a word occurs multiple times in a sentence. That said, this does not damage performance and we see that there is an accuracy of 1.0 for both of these two similarity measures. Speed was also comparable to cosine (though these two metrics appear *a little* faster at 1.026 seconds).

ABLATION

Because we are at 1.0 accuracy, I wanted to see if I could speed up or simplify the system by removing components and maintain good accuracy.

Tokenize + MySql Stopwords + Lowercase
performance (cosine, dice, and jaccard): 0.9

Tokenize + MySql Stopwords + Stem
performance (cosine, dice, and jaccard): 1.0

Tokenize + MySql Stopwords
performance (cosine, dice, and jaccard): 0.9

Tokenize + Lowercase + Stemming
performance (cosine, dice, and jaccard): 0.799 (baseline level)

From this I surmise that it is the MySql stopwords that do the heavy lifting for our performance. We also note interestingly that although the 3 similarity metrics differ, they give the same results.

I see that lowercasing could be turned off and we could still get 1.0 performance. However, it does not save much speed so I leave it on.