Ryan Carlson (rcarlson)
11791 – HW 2 Report
September 23, 2013

In this brief report I describe my Aggregated Analysis Engine for the given information processing task. First, I touch on each primitive analysis engine and how they are combined, and then I go into a bit more depth on each primitive engine. As needed, I also describe the additions to the type system I made.

Recall that the information processing tasks has five parts: Test Element Annotation, Token Annotation, NGram Annotation, Answer Scoring, and Evaluation. For my assignment, it seemed to make the most sense to collapse Token Annotation and NGram Annotation into the same phase, since when N=1 (unigrams) we just have tokens. While this increases the coupling of the two phases, it also means we have much less bookkeeping to worry about, since we can easily associate the ngrams with a question or an answer without adding redundancy. That is, we don't need to store the tokens *and* the unigrams, which seems wasteful. I recognize that we might have slightly different notions of "tokens" and "unigrams" (e.g., unigrams might not include punctuation, while tokens would) but I chose not to distinguish in this assignment, since that information could be relevant.

So, my aggregate analysis engine has the following primitive components (in fixed flow):

1. `TestElementAnnotator`
2. `NGramAnnotator`
3. `GoldStandardScoreAnnotator`
4. `TokenOverlapScoreAnnotator`
5. `CosineSimilarityScoreAnnotator`
6. `EvaluationAnnotator`

Note that annotators 3, 4, and 5 are all part of the Answer Scoring phase – I discuss them a bit more later on.

The `TestElementAnnotator` takes a raw document as input and outputs `Questions` and `Answers`. Each `Question` knows where its text starts and ends. Each `Answer` also knows its start and end position in the original text, and is annotated as Correct or Incorrect (a boolean). Both are assigned "`TestElementAnnotator`" as their `CasProcessorId`.

The `NGramAnnotator` takes `Question` and `Answer` objects as input and modifies them, adding lists of `NGrams` to each `Question` and `Answer` it sees. There is no direct output from this component – instead it has side effects. Instead of changing the `CasProcessorId` and following that around, it seemed that these objects should know about their own features, so I added `FSArrays` of `NGram` objects to each `Question` and `Answer` type. During NGram Annotation unigrams, bigrams, and trigrams are extracted and added to an `FSArray` of `NGrams` and then added to each `Question` or `Answer`, respectively. Again, I could have just added

NGram objects to the index and then in future steps either used the CasProcessorId or the begin/end markers to determine which Question/Answer the NGrams were associated with, but I decided to assign responsibility to the Question/Answer objects.

After annotating the ngrams, we need to assign scores to each answer. So we grab the Question, and for each answer we can assign a score. I created three different Score Annotators:

- GoldStandardScoreAnnotator: for each answer, if isCorrect, then assign score=1, otherwise score=0. This should return perfect precision, and is just used during testing.
- TokenOverlapScoreAnnotator: for each answer, count the number of words the (question, answer) pair have in common. Set the score to that count.
  CosineSimilarityScoreAnnotator: here we calculate the cosine similarity between the question and each answer. To do that, we need to know all the features we'll be using (the full vocabulary), since we put them into vectors that we take the dot product / magnitude of. So we assign each feature to a position in the vector, and then count up those features and fill out the vectors for the question and for each answer. Then we find the cosine similarity, for each answer A_i:
  $$\frac{Q \cdot A_i}{\|Q\| \times \|A_i\|}$$
  Note that the features for the question and answers are the counts of each unigram, bigram, and trigram in each segment.

Finally, after scoring each answer, we just need to sort the answers by score and then evaluate Precision@N. To do this, I needed to create a new type Evaluation that contained an FSArray of Answer objects, and a double precisionAtN that records the precision. To calculate precision, we count the number of correct answers (N), and of the first N answers in our list sorted by score, we calculate the fraction of those that are in fact correct.

Also, the Evaluation Annotator filters the AnswerScore objects so that only certain similarity score annotators are evaluated – in this case, I decided to use only the CosineSimilarityScoreAnnotator. In the future we might consider more sophisticated algorithms that combine multiple types of score annotators. For example, we might have a ScoreMergeAnnotator that takes some linear interpolation of the scores. On the training set, this method had a P@N = 0.5 for the question about Lincoln, and P@N = 0.67 for the question about John and Mary. By a sorting quirk, the TokenOverlapScoreAnnotator actually achieved perfect precision on the John/Mary question, but I believe the Cosine Similarity metric to be much more valid, so I'm using that in my final submission.

You can find all my java classes are in the `src/main/java/edu.cmu11791.rcarlson` package.

All my primitive analysis engines are in `src/main/resources/analysis_engines`.