Mario Piergallini
mpiergal

Homework 2 Report: Creating UIMA annotators

This assignment primarily consisted of creating the annotators for the type system, their **descriptors** and combining them into the **Aggregate Analysis Engine** (AAE), and only a few minor changes to the type system. The task for the descriptors and AAE was fairly simple. For each annotator, simply create the xml file, associate it with the proper java file, and give it the capability to output the type it's annotating. For the AAE, then add those descriptors to the **Component Engines** and **Component Engine Flow** in the proper order (i.e. ensure that no annotator comes before an annotation it needs).

The first step, then, was to create the **TokenAnnotator**. To do this, I simply copied the UIMA tutorial file to give me the skeleton of what classes I needed to import and implement, etc. I created a regular expression which would find tokens composed of alphanumeric characters and apostrophes, and used negative lookbehind to ensure I was not marking the "Q" or "A" markers or gold standard isCorrect annotations as tokens. As long as the token doesn't come at the beginning of the document or following a newline, or following a newline and an "A", it will be annotated. Then simply iterate over the regular expression matches and create new Token annotations and add them to the indices.

The second annotator was the **NGramAnnotator**. This used a similar regular expression to the TokenAnnotator to find bigrams and trigrams. I iterated over the Tokens in the JCas, then applied the bigram and trigram regular expressions from the beginning of the token. If the token was part of a bigram or trigram, then those two or three tokens were associated to the NGram. Finally, unigrams were created essentially by copying the tokens. While this seems redundant, as I could have simply used the tokens as unigrams later in the pipeline instead, this makes it so that if I had some restriction on unigrams (such as wanting to stem them, or exclude stop words) I would only need to change the NGramAnnotator.

The third and fourth annotators were the **QuestionAnnotator** and **AnswerAnnotator**. These two work very similarly. They each apply a regular expression to find the location of question or answer and annotate it. They both associate all the NGrams within the question or answer to that annotation. The primary difference is that the AnswerAnnotator also checks whether the answer is correct and sets the isCorrect feature accordingly. Since the Question and Answer types did not start out with these NGram FSArrays, I had to add an FSArray for each of the unigrams, bigrams and trigrams to both types.

The **AnswerScoreAnnotator** then iterates through the Answers in the JCas and compares the NGrams associated with them to the NGrams in the question. The score is initially equal to the number of NGrams from the question that are found in the answer divided by the total number of NGrams in the question. Then I use a regular expression to detect whether the answer sentence contains any negation (negative contraction, not, no, never, nothing, etc.) and applies a penalty to it. This is because for the questions we're answering, only a sentence formulated like "Booth didn't not shoot Lincoln" could both contain negation and be correct. I suppose debatably "John doesn't merely like Mary" could be considered correct, but I wasn't sure how such a case would be handled in the first place.

Finally, the **EvaluationAnnotator** iterates over the AnswerScores in the JCas and stores their scores and gold standard annotations in a couple ArrayLists. As it does so, it counts the number of correct answers, which defines the N in the precision at N that will be calculated. Then it iterates over those ArrayLists N times and removes the N top scores and while counting the number of them that were correct. After finishing with the Nth best scoring answer, it checks the remaining answers for ties. I wasn't sure how we were supposed to handle those, but since otherwise the score would be affected by the order of the answers in the document, I decided that N should increase if there is a tie for Nth best score. In other words, if we were supposed to calculate precision at 3, but the top four answers were all tied, instead I would calculate precision at 4. The precision at N is then simply the number of correct answers in the top N scoring answers, divided by N. This is then annotated on the JCas on the Q of the question. I also used print statements to output scoring and evaluation information in more detail on the console.

The system would be much improved if I detected verb tenses (or simply lemmatized words), incorporated Wordnet to detect synonyms or used a dependency parser to correct for passive voice. That would likely be adequate for the task, since I assume we wouldn't be given any test data requiring co-reference resolution or recognizing alternate references to the proper nouns (e.g. recognizing that "the 16th president" means the same thing as "Lincoln"), for example.