11-791: Design and Engineering of Intelligent Information Systems

Homework #2 – Lars Mahler – lmahler@andrew.cmu.edu

Implementing a Simple Information Processing Task with UIMA SDK

In implementing this QA system, I used the example given in class as a point of reference, and made several modifications, taking several considerations into account.

Using the deiis_types.xml file as a starting point, I first considered the requirements of the information pipeline, and then broke it into several stages (Analysis Engine components) which are described below.

Stage / Description	Input Types	Output Types
1. TestElementAnnotator	N/a (text file as input)	**Question, Answer**
• The TestElementAnnotator annotates each input document. For each line of the input file, it determines if that line is a Question, or an Answer, and annotates the span accordingly. o For each Answer annotation, the TestElementAnnotator will sets the isCorrect feature to yes / no (1/0), based upon the flag given in the input file.		
2. TokenAnnotator	**Question, Answer**	**Token**

• The TokenAnnotator converts each Question and Answer annotation into a set of Token annotations. The Token annotations are then used in downstream processing.
• Please note: TokenAnnotator.java utilizes the Stanford libraries utilized in HW0. Since they already exist on the mu.lti repository, I assume there will be no problems with loading them on the repository side, but if you run into any issues, please feel free to contact me. Specifically, the script imports the following:
  o edu.stanford.nlp.ling.Word;
  o edu.stanford.nlp.objectbank.TokenizerFactory;
  o edu.stanford.nlp.process.Tokenizer;
  o edu.stanford.nlp.process.PTBTokenizer.PTBTokenizerFactory;
• Note: I had hoped to add the following features to the Token type – these would help build a better machine learning (or rules-based) model, however getting up to speed on Java and UIMA syntax and structures took longer than anticipated. If we do a future iteration of the QA system, I hope to add the following, using Stanford CoreNLP:
  o A **stem** (or **lemma**) feature containing the stem (or lemma) of the token. For n-gram models, using stemmed/lemmatized words can contribute to a smoother n-gram that can better generalize to text that contains new forms of previously seen words (e.g. "Does John love Mary?" → "John loves Mary.")
  o A **pos** feature containing the Part of Speech tag for the token. Introducing this as a feature in an n-gram model would better distinguish between sentences like "John loves Mary" vs. "Mary loves John".		

3. NgramAnnotator	**Token**	**NGram**

• The NgramAnnotator converts sequences of Token annotations into Ngrams.
• A parameter allows the user to specify the which NGram orders will be created.
  o Ex: 1: setting the parameter to "1,2,3" will create unigrams, bigrams, and trigrams.
  o Ex: 2: setting the parameter to "2,4" would create bigrams and 4-grams.
• Each time an **NGram** annotation is created:
  o The **start** feature is set equal to the **start** of the first Token in the NGram.
  o The **end** feature is set equal to the **end** of the last Token in the NGram.		

- The **elements** feature is an array that contains pointers to each Token in the NGram.
- The **elementsType** feature is set to "Token". (Note: this could become parameterized in future releases, if types other than Tokens are used in NGram models)
- The **ngramOrder** feature tells whether the NGram is a unigram ("1"), bigram ("2"), etc – this may be used in future releases if I develop a machine learning model that weights the NGrams differently.
- The **source** feature indicates whether the NGram came from a Question or an Answer. This is used in downstream processing to compare Answer NGrams against their corresponding Question NGrams.

4. **Answer Scoring**	**Question, Answer, NGram**	**AnswerScore**

- The AnswerScorer loops through each **Question** (there should only be one in each document). For each Question, the AnswerScorer loops through the candidate **Answer** annotations and scores them.
- For this implementation, the scoring methodology is a simple NGram overlap rule: for each NGram contained in the candidate Answer, the AnswerScorer looks to see if that NGram is also contained in the Question. If so, it increments a "matching_ngrams" counter by 1. The final score feature is simply the count of "matching_ngrams" / the total number of n-grams contained in the candidate Answer.
- Note: the approach above corresponds to a interpolated n-gram model, with even weights between each n-gram order. For this implementation, I used 1-grams, 2-grams, and 3-grams, so the weights are 1/3 unigram, 1/3 bigram, 1/3 trigram. This is a very simple model. Per the note above (re: adding features to the Token type), in a future release, I would like to have a richer feature set, and perhaps use machine learning techniques to build a more robust scoring mechanism.
- When the score is returned, a new **AnswerScore** annotation is created. This annotation spans the text of the answer sentence. In addition, it has a **score** feature (that stores the score), as well as an **answer** feature (that has a pointer to the Answer object that was used during scoring).

5. **Evaluator**	**AnswerScore**	N/a (results print to screen)

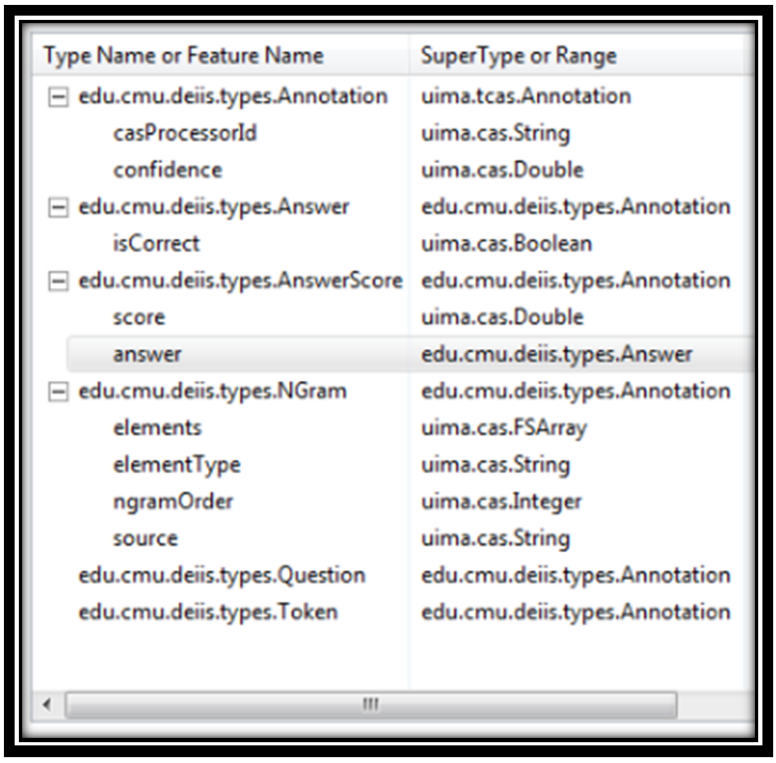
- The Evaluator evaluates the performance of the AAE by comparing the system outputs (**AnswerScore.score** values) with the gold standard outputs. It measures precision by doing the following:
  - 1) Ranks the Answers according to AnswerScore.score (descending).
  - 2) Selects the top N Answers (where N = the total number of correct answers that were possible for that Question).
  - 3) Measures precision by computing how many of the top N ranked Answers were actually correct (based upon the gold standard).
- After performing these calculations, it prints the results to screen.

I enhanced the type system as follows:

- For the **NGram** type:
  - Added the **ngramOrder** feature, which tells whether the NGram is a unigram ("1"), bigram ("2"), etc – this may be used in future releases if I develop a machine learning model that weights the NGrams differently.

- Added the **source** feature, which indicates whether the NGram came from a Question or an Answer. This is used in downstream processing to compare Answer NGrams against their corresponding Question NGrams.

Figure 1: Types and Features



Type Name or Feature Name	SuperType or Range
edu.cmu.deiis.types.Annotation	uima.tcas.Annotation
casProcessorId	uima.cas.String
confidence	uima.cas.Double
edu.cmu.deiis.types.Answer	edu.cmu.deiis.types.Annotation
isCorrect	uima.cas.Boolean
edu.cmu.deiis.types.AnswerScore	edu.cmu.deiis.types.Annotation
score	uima.cas.Double
answer	edu.cmu.deiis.types.Answer
edu.cmu.deiis.types.NGram	edu.cmu.deiis.types.Annotation
elements	uima.cas.FSArray
elementType	uima.cas.String
ngramOrder	uima.cas.Integer
source	uima.cas.String
edu.cmu.deiis.types.Question	edu.cmu.deiis.types.Annotation
edu.cmu.deiis.types.Token	edu.cmu.deiis.types.Annotation