Logical Architecture and UIMA Analysis Engines Design & Implementation

Chenying Hou
Andrew ID:chenyinh

September 23, 2013

1 Introduction

The information processing task is as follows:
The task Given a question and a set of sentences, determine which sentences answer the question:

a. Assign a score to each sentence [0..1];

b. Rank the sentences according to score;

c. Select the top N sentences where N = the number of correct answers;

d. Measure performance by Precision@N (how many of the top N are correct).

In order to process the task, we need to design a pipe to deal with it. We should first add the document type into CAS and than get the Question, Answer, Token and Ngram annotations. Then based on the annotations we get, using different methods computes scores for each answer and finally evaluate the result. The detailed steps in the pipeline are presented below:

a. **Test Element Annotation**: The system will read in the input file as a UIMA CAS and annotate the question and answer spans. Each answer annotation will also record whether or not the answer is correct.

b. **Token Annotation**: The system will annotate each token span in each question and answer (break on whitespace and punctuation).

c. **NGram Annotation**: The system will annotate 1-, 2- and 3-grams of consecutive tokens.

d. **Answer Scoring**:The system will incorporate a component that will assign an answer score annotation to each answer. The answer score annotation will record the score assigned to the answer.

e. **Evaluation**:The system will sort the answers according to their scores, and calculate precision at N (where N is the total number of correct answers).

2 Analysis Engines Design

According the the pipeline mentioned in the previous section, Figure 1 shows the structure of my aggregate Analysis Engines. *TestElementAnnotator* gets the Question and Answer information. *TokenAnnotator* gets the token information. *TokenUniGramAnnotatorDescriptor*, *TokenBiGramAnnotatorDescriptor* and *TokenTriGramAnnotatorDescriptor* generate the *1-Gram*, *2-Gram* and *3-Gram* objects we need in the following scoring part.

We choose fixed flow and set the order or the annotators (Some steps are independent such as *TokenUniGramAnnotatorDescriptor*, *TokenBiGramAnnotatorDescriptor* and *TokenTriGramAnnotatorDescriptor* , while some steps are dependent to others results, so the order is important). All the annotators in my project are extended from the class *JCasAnnotator_ImplBase*. The main part is the the *process()* function that need to be implemented. I will now describe each annotator in detail.

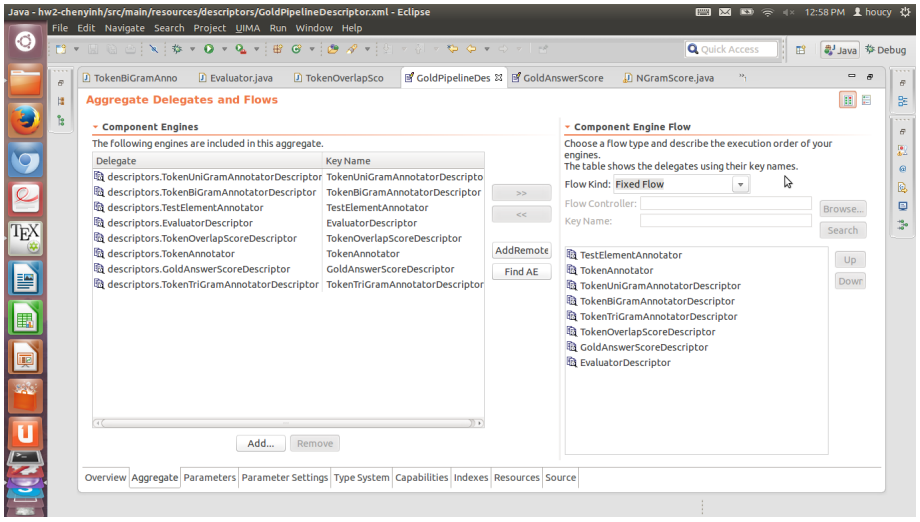

Figure 1: The Pipeline for Information Processing Task

2.1 TestElementAnnotator

The *TestElementAnnotator* generates questions and answers from the document as shown in Figure 2 and Figure 3. Using regular expression can efficiently retrieve questions and answers, which means that when the first letter in a line is *Q*, it leads a question sentence and when the first letter is *A*, it leads an answer sentence. I set the span information in the feature *begin* and *end*,and fill in the *casProcessId* and *confiedence* information. For an answer, I also record whether it is correct in the feature *IsCorrect* with boolean value.

Besides the java code for *TestElementAnnotator*, we should also build the descriptor XML file and import the type systems into it and set the output type of the Annotator be Question and Answer.(All the following annotators are the same with XML file creating, I will not discuss any more in the following part)

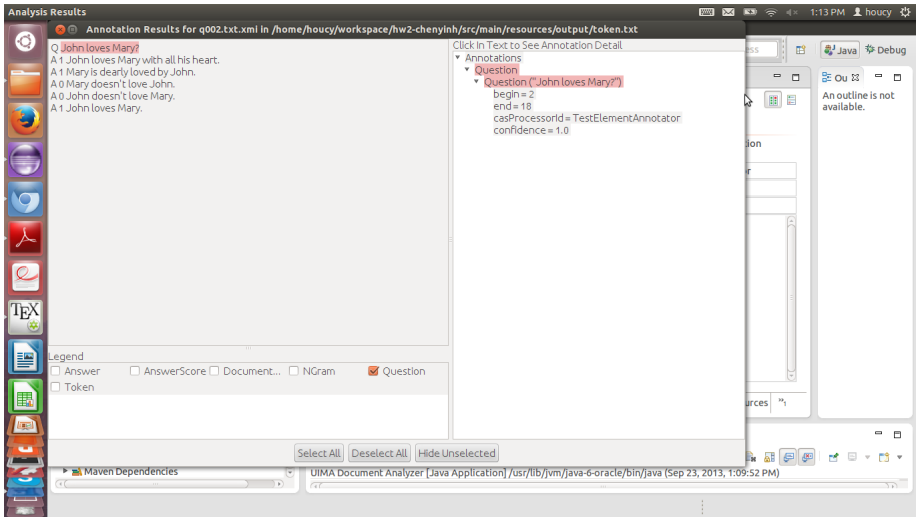

Figure 2: Question Annotation

2.2 TokenAnnotator

Regular expression can also serve for the token annotation generation. One thing need to mention is that, in a sentence, there can be a word with ' like *doesn't* so we should also consider that condition when using regular expression to match tokens.

In my realization, I adopt the *aJCas.getAnnotationIndex(Answer.type)* method to get the specific annotation index, which can filter other useless annotations when trying to retrieve tokens. Figure 4 shows token annotation in my implementation. I also record the *casPrcessorId* and *confidence* information.

The *TokenAnnotator* must be executed after the *TestElementAnnotator*, because it replies on the result of question and answer.

2.3 Ngram

This system also generate 1-gram, 2-gram and 3-gram annotations. *TokenUniGramAnnotator*, *TokenBiGramAnnotator* and *TokenTriGramAnnotator* are three annotators generating all Ngram annotations. *TokenUniGramAnnotator* can be

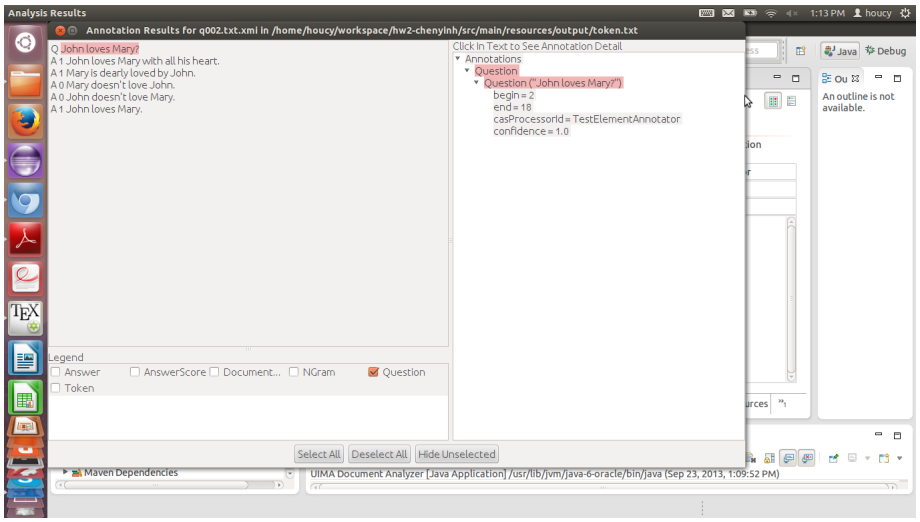

Figure 3: Answer Annotation

directly gotten from the *Token* annotation. *TokenBiGramAnnotator* and *TokenTriGramAnnotator* can than be generated. Figure 5 shows the NGram result of this system and in the figure, it is 1-gram 2-gram and 3-gram about token 'Mary'.
I generated 1-gram, 2-gram and 3-gram annotations in order to use NGram scoring method in the next step.

2.4 Scoring

During the scoring section, this annotator engine realizes two different scoring method, which are *TokenOverlapScore* and *NGramScore*. The scoring method generates the AnswerScore annotations and Figure 6 shows the result of this part.

2.4.1 TokenOverlapScoring

The rule of *TokenOverlapScore* is

$$score = questiontokensfound/totalanswertokens \quad (1)$$

So what we need to do is to separately collect the tokens in each question and answer. Then compare them and find how much proportion is overlapped and get the final score based on the equation. We need to generate the AnswerScore annotation in this step and record the score and answer information in the AnswerScore features. Furthermore, in order to compare between different scoring algorithms, it is necessary to annotate that which processor produces this score,

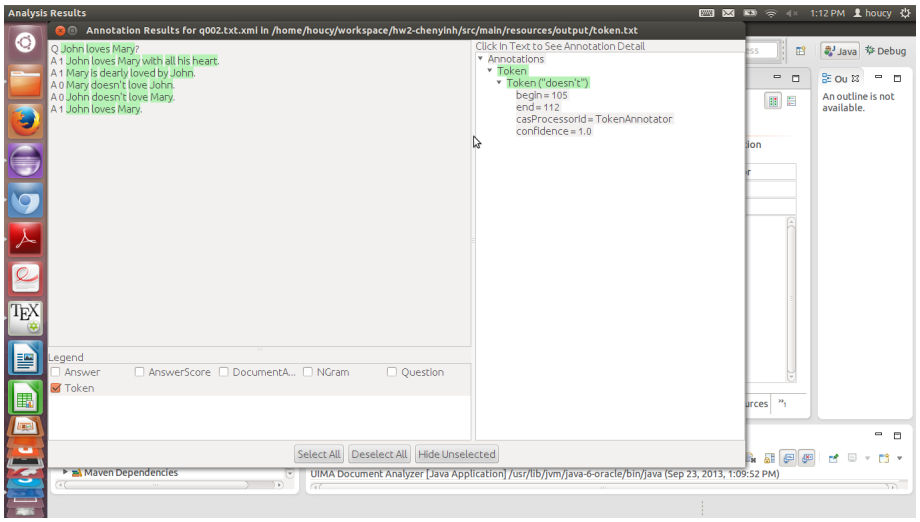

Figure 4: Token Annotation

which is related with *casProcessorId* feature.

In Figure 6, the example is the answer "John loves Mary with all his heart". The first AnswerScore annotation is generated by *TokenOverlapScore*, which shows by *casProcessorId* feature.The score is about 0.43 for this answer.

2.4.2 NGramScoring

The rule of *NGramScoring* is

$$score = questionNGramsfound/totalanswerNGrams \quad (2)$$

Very similar with *TokenOverlapScore* method, the main difference is that we try to match Ngrams in *NGramScore* method. For each answer, I generate a new AnswerScore annotation and marked *casProcessorId* with *NGramScore* and record its corresponding score. As shown in Figure 6, the second AnswerScore annotation in the right column is what *NGramScore* produced.

2.5 Evaluate

The evaluate part in the pipeline is to compare the scores generated by different scoring methods with the right answer. Figure 7 and Figure 8 shows the precision results and the precisions are also given. I ranked the answer by its score and the second column gives the true value of the answer. The precision for first input sample is 0.5 under both scoring methods. For the second input sample, the precision of *TokenOverlapScore* is about 0.33 and *NGramScore* can

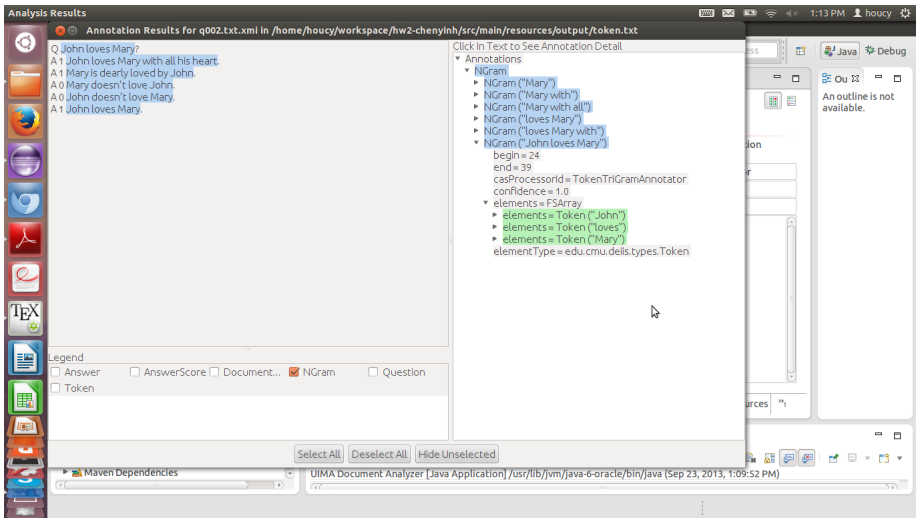

Figure 5: NGram Annotation

achieve about 0.67. (I adopt precisioin@N to calculate the precision, see section 1)

3 Conclusion and Discussion

From the results of different scoring methods, we can get some conclusions.
First, in the input samples, the *NGramScore* method outperforms *TokenOverlapScore* method in predicting whether the answer is true or false. Since *NGramScore* makes more kinds of comparisons between questions and answers, it is easy to arrive to the conclusion that it should be better than the *TokenOverlapScore* method which only relies on token.
Second, *TokenOverlapScore* methods although cannot achieve good precision compared with *NGramScore*, it is easy to implement and does not need the extra NGram annotations, so the cost of this method is cheaper.
Third, both two scoring methods do not retrieve the lexical features and semantic roles of the tokens. I think this is the where the scoring algorithm can improve a lot. Such as matching the tokens with similar meaning, and identify the negative and positive attribute of a sentence or words/tokens.

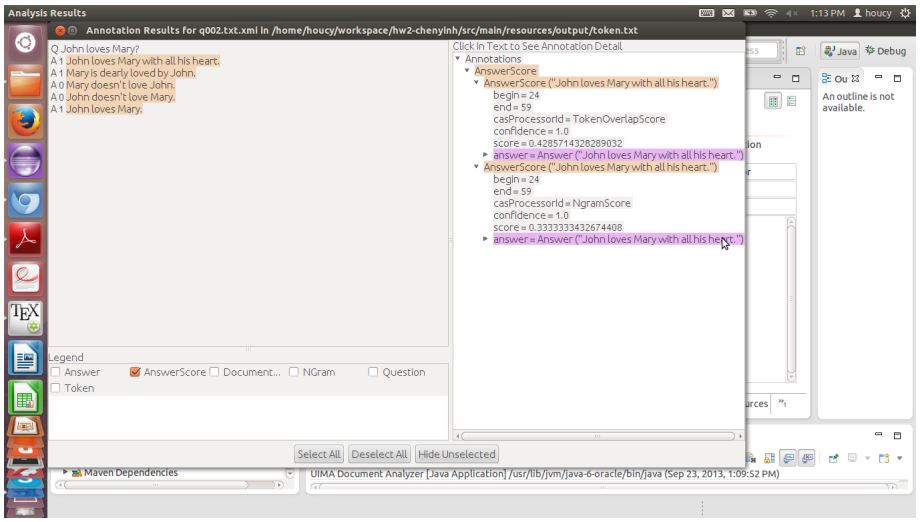

Figure 6: AnswerScore Annotation

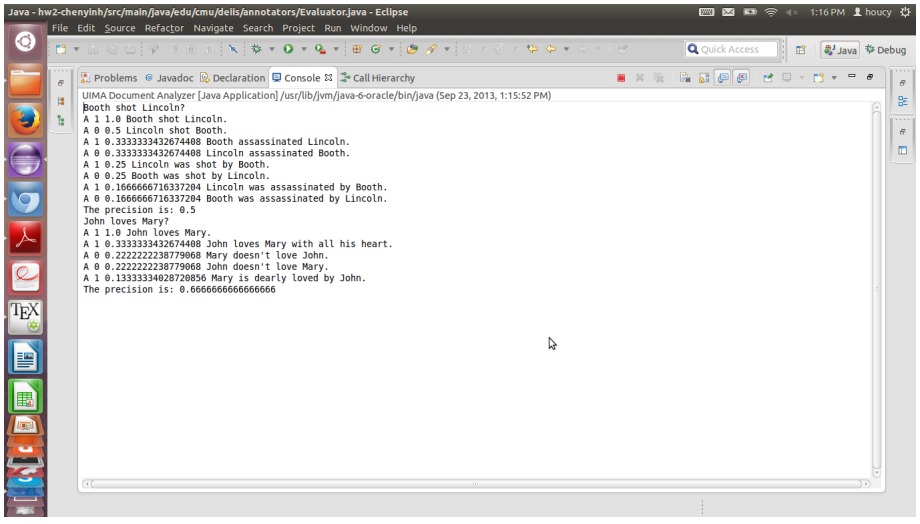

Figure 7: NGramScore Result

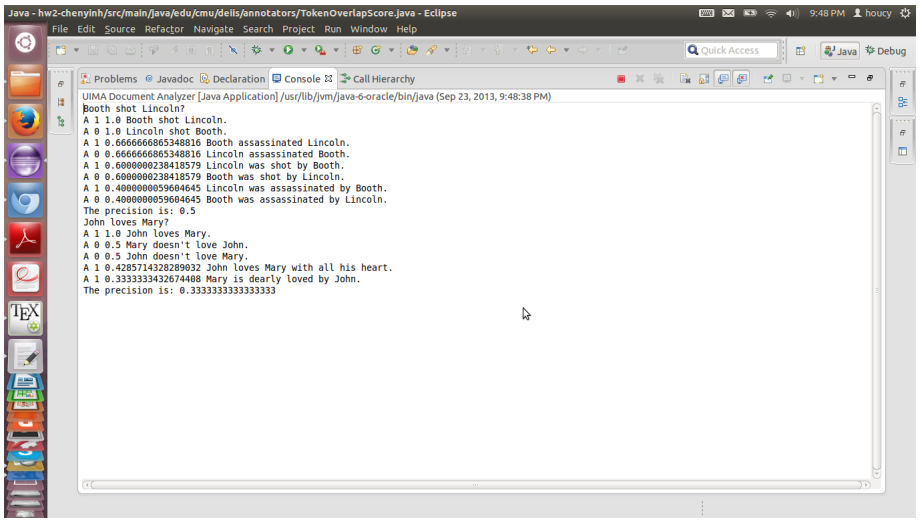

Figure 8: TokenOverlapSocre Result