Assignment 2
11-791: DEIIS

Submitted by: Ankur Gandhe, ankurgan

September 23, 2013

1 Design of Analysis Engines

The task of this assignment is to design analysis engines for the sample information processing task that reads a file containing a question and a set of sentences, determines which sentences answer the question. Each line in the file is either the question or a candidate answer:

```
Questions are of the form: Q <question>.
Answers are of the form: A <isCorrect=0/1> <answer>
```

The type system was provided to us.

1.1 Modified Type System

The type system provided to us was sufficient for most annotations. However, we also wanted to store the final result in an annotation so that the .xmi files contain the score appropriately. Hence, we created a 'Evaluation' type with feature 'precision' in the deiis type system.

1.2 Analysis Engine Design overview

For the given task, we have tried to create a modular, hierarchical analysis engine to produce the final result. We explain our design methodology according to the processing pipeline the data follows in the aggregate analysis engine:

1. **QuestionAnnotator:** The test document contains a single question which needs to be annotated. Hence, we use a java regular expression to mark the begin and end of the Question in the document.

2. **AnswerAnnotator:** The test document contains several answers. Each of the correct answers (pattern: A 1) and wrong answers (pattern: A 0) is identified with a corresponding regular expression and marked in the text. Along with begin and end, the feature 'isCorrect' of type *Answer* is also recorded with its correct value.

3. **TokenAnnotator**: We create a class called *TokenAnnotator* to annotate the tokens appearing in the test document. Special care is taken not to annotate the meta information(Q,A or the score of answers) and annotate only the text. We annotate using a java regular expression, which makes the task extremely fast. Also, using java's ‘’_ to mark word-boundaries, we can detect punctuations such as ‘,’,‘.’,‘?’ correctly. All token annotations are marked with a begin and end.

4. **NGramAnnotator:** We create a class called *NGramAnnotator* to annotate 1-gram, 2-grams and 3-grams. Since n-grams are built on top of tokens, *NGramAnnotator* takes tokenAnnotations as inputs. Hence, it is important that in the aggregate analysis engine, TokenAnnotation happens before NGramAnnotation. We want to create n-grams for tokens falling in the same sentence only. Hence, we create a regular expression to match sentence begin and end. Next, we create the 1-gram, 2-grams and 3-grams out of the tokens falling within that sentence. We mark each element by its type(1gram,2gram or 3gram).

5. **AnswerScoreAnnotator:** This is the most critical part of our system as well as the one that should be the most flexible (so that we can easily test different scoring algorithms). It takes *AnswerAnnotator*'s, *QuestionAnnotator*'s and *NGramAnnotator*'s outputs as inputs, and hence should be run only after them in the aggregate engine. The scoring begins by collecting the NGrams that fall within the QuestionAnnotation and storing them in a List. Next, we iterate over the AnswerAnnotations. For each AnswerAnnotation, we compare the NGrams in the Question with the NGrams in the Answer and scoring for number of n-gram matches. The final score is obtained by dividing the number of n-gram matches with total number of n-grams. We then create the AnswerScore annotation and store the Answer and score values in that Annotation.

6. **EvaluationAnnotator:** The system needs to sort the answers according to score and calculate precisionN. We define a class *EvaluationAnnotator* to accomplish this. It takes the output of AnswerScoreAnnotator as an input, along with the QuestionAnnotation. From the AnswerScoreAnnotations, we create a list of all AnswerScore objects and sort them according to the decreasing value of scores. While doing this, we also store the number of correct answers in the list of answers. Final precision is then calculated by going over the ranked list. The final precision value is stored in a Evaluation object. We also store the total precision and number of documents processed to output the average precision across the documents. The final output is written to stdout, in the format suggested by the instructor.

1.3 Other concerns

While designing our analysis engine, we chose one of several options available for design on several occasions.

- While designing the question/answer annotators, we had an option of creating a FSList and storing the n-grams that occur in that answer/question. However, this would have had 2 disadvantages : a) We would have to decide on the 'ngram' created by a single process. Instead, we can leave this decision untill the end. b) This would require much more space.

- While choosing between simple n-gram matching versus weighted n-gram matching ,we did not see much difference in the output of the two methods.