11791 Homework 2 Report

Yueran Yuan

Overview and Goal

The purpose of the system is to evaluate text documents of a certain format (specified in Homework writeup) containing a question and several answers and produce:

1. AnswerScore annotations which annotates each answer (other annotations are produced incidentally) for by the TA's evaluation script.
2. Console writeout of the question, the answer scores, and a precision evaluation measure. Example is given in **appendix a**

This is achieved by building an aggregated analysis engine inside the UIMA framework.

System Overview

The analysis pipeline consists of 4 main parts which occur in sequential order:

1. Read test element components (question and answer annotations) from the raw text
2. Compute various annotations for the components (create Token and Ngram annotations)
3. Compute a score for the each answer (creates AnswerScore)
4. Evaluate the answers as a whole (compute precision) and write to the console

The pipeline is separated in that fashion in compliance with the semantics of the data types provided for the assignment and to ensure *separation of concerns* and *low coupling*.

The output of steps 1-3 are Annotations. Because there is a large amount of redundancy in the creation of Annotations (i.e. all annotations must have confidence, and casProcessorId set), Factories are used. Processors do not need to know how to create annotations, this logic is located in Factories. A Factory exists for each data type. Some factories are smarter than others, for example the NgramAnnotationFactory internalizes the processing of computing the begin and end of the annotation from the tokens the Ngram is made of.

An adapter (of sorts) is used between the UIMA annotator system and processors that I have created. This is because I wanted the processors to take advantage of inheritance from abstract classes so that functionality between similar annotators could be inherited. UIMA annotators cannot be abstract so I've created a series of “Processors” that the UIMA annotator wraps around. These processors are used whenever inheritance from abstract classes is necessary.

Step 1 Reading Test Elements

This step is accomplished quite simply by using regular expressions to extract questions and answers from the raw text

Step 2 Annotate Test Elements

This step uses two Annotators. For this step (and the next one) I used the AbstractTestElementProcessor. This is because all of these annotators need to load the question and answer Cas and looping through the answer Cas to perform processing. This processing is factored out into a single abstract class and each process that extends it inserts its own logic into process question and process answer. This also allows malformed structure exception handling (like what to do when the text contains now question) to be done in one place.

There are two substeps in this step. The first is a processor to tokenize the input. Tokenization is done with the stanford.nlp.tokenizer. Tokenization is difficult and there's no reason to implement my own tokenizer. I removed punctuation in the tokenization because they didn't not hold much semantic information (including them does not improve performance).

The second substep is a processor to connect the tokens into Ngrams. I chose to use Ngrams of any length (i.e. from unigram to infinite-gram) as opposed to capping the Ngrams at some threshold. I did this because this is the most general solution.

Step 3 Scoring Answers

Answer scores were computed using Ngrams although the processor extends an abstract scoring processor that could use other features to score. I built two Ngram scoring algorithms, both of which extends an abstract ngram scoring processor. The abstract ngram scoring processor handles all the logic of reading out the Ngrams corresponding to each answer and integrating the score for each individual Ngram into a whole so that the class of each algorithm only needs to know how to compute a score for a pair of Ngrams.

Design Decisions

There is a notable trade-off here. In order to reduce the redundancy in the code and to add a layer of indirection to isolate separate concerns, I am increasing the coupling between the Ngram processor, the scoring algorithm, and the Ngram type system. I chose my solution because it makes the code more readable.

To reduce coupling and to separate concerns, I've separated the code to compare Ngrams and computed similarity between Ngrams into its own class. This way, none of the code inside the Ngram processor (other than this class) actually needs to know how to parse the Ngram Annotation CAS.

Scoring Algorithms

My processor checks every Ngram in the question against every Ngram in a given answer. This results in X * Y scores (where X is the # of ngrams in the question and Y is the # of ngrams in the answer). These scores floats between 0 and 1 and the sum of the scores is divided by X * Y. This results in a total score for the answer which is between 0 and 1

I implemented 2 simple per-ngram scoring algorithms which I compared. I called them Basic and Fuzzy.

- **Basic** uses a flat weight on every Ngram and only counts exact matches. That is to say, the score of each Ngram pair is 1 if there is a match and 0 if there is no match.
- **Fuzzy** uses a similarity measure between the Ngrams where the tokens of the Ngrams are matched sequentially. Then the number of matches is divided by the length of the longest Ngram. This similarity roughly represents the substitution-only edit-distance between the two Ngrams. The fuzzy algorithm approximates the performance of skip Ngrams without needing to create skip ngrams.
    - Example: “hello world” and “hello world world” = 2 / 3
    - “hello good world” and “hello cruel world” = 2 / 3

    - “hello” and “hello cruel world” = 1 / 3

Step 4 Evaluating Answers Scores

This step consists of a processor that (1) ranks the answers by score (2) prints question the answers in order by score and (3) computes the precision@N. This step also extends the abstractTestElementProcessor class but overrides the inner loop (because it has to deal with answer scores rather than answers).

Results

I was able to match the baseline results (see Appendix A for my results). Although both the fuzzy and basic ngram scoring algorithms got the same precision score, we note that the fuzzy scores were able to break ties between two similar evaluations. If we consider that ties. And except for passive sentences, the fuzzy scores resulted better rank accuracy (because it breaks ties which were formerly broken arbitrarily). This is presumably because we assume that though some minor wording may differ, when most of the words in a sentence are in similar relative places, the answer is good.

Appendix A

Basic Ngram Console Writeout:

```
Question: Booth shot Lincoln?
+ 0.17 Booth shot Lincoln.
- 0.08 Lincoln shot Booth.
+ 0.06 Booth assassinated Lincoln.
- 0.06 Lincoln assassinated Booth.
+ 0.03 Lincoln was shot by Booth.
- 0.03 Booth was shot by Lincoln.
+ 0.02 Lincoln was assassinated by Booth.
- 0.02 Booth was assassinated by Lincoln.
Precision at 4: 0.50
Question: John loves Mary?
+ 0.17 John loves Mary.
+ 0.04 John loves Mary with all his heart.
- 0.02 Mary doesn't love John.
- 0.02 John doesn't love Mary.
+ 0.02 Mary is dearly loved by John.
Precision at 3: 0.67
```

Fuzzy Ngram Console Writeout:

```
Question: Booth shot Lincoln?
+ 0.28 Booth shot Lincoln.
- 0.21 Lincoln shot Booth.
+ 0.17 Booth assassinated Lincoln.
- 0.14 Booth was shot by Lincoln.
+ 0.10 Lincoln was shot by Booth.
- 0.10 Lincoln assassinated Booth.
- 0.08 Booth was assassinated by Lincoln.
+ 0.05 Lincoln was assassinated by Booth.
Precision at 4: 0.50
Question: John loves Mary?
+ 0.28 John loves Mary.
+ 0.11 John loves Mary with all his heart.
- 0.08 John doesn't love Mary.
- 0.05 Mary doesn't love John.
+ 0.03 Mary is dearly loved by John.
Precision at 3: 0.67
```