11-791: Design and Engineering of Intelligent Information Systems

Homework #1 – Lars Mahler – lmahler@andrew.cmu.edu

Designing and Implementing the Logical Data Model

In designing the Logical Data Model, I used the example given in class as a starting point, and then modified it, taking several considerations into account:

1) The **requirements** of the information pipeline.
2) Possible **future enhancements** or requirements.
3) **Best practices** in type design.

Where applicable, I will discuss each of these in the table below.

I first considered the requirements of the information pipeline – the stages, and requirements for each stage – and then determined the set of types needed to support each stage. This set was similar to the example from class – however several modifications were made:

Stage / Description	Input Types	Output Types
1. 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.	N/a (text file as input)	**Sentence, Question, Answer**
• At this stage, the TestElementAnnotator will convert the input document into a set of sentences, and annotate each **Sentence** span accordingly. • For each **Sentence**, the TestElementAnnotator will determine whether it is a **Question** or an **Answer**, and annotate the span accordingly. • For each **Answer**, the TestElementAnnotator will set its **isCorrect** feature to yes / no (1/0) depending on whether it is a correct answer or not. See Fig. 1 for listing of types and features.		
2. Token Annotation: The system will annotate each token span in each question and answer (break on whitespace and punctuation).	**Sentence**	**Token**

• At this stage, the TokenAnnotator will loop through each **Sentence**, and annotate **Token** spans based on the whitespace / punctuation specifications.
• For each token, I introduced several features that would be populated:
  o The **word** feature contains the raw text string of the token. Holding this information in a feature may make future subprocessing steps easier: they can look through each token and grab its Token.word feature.
  o The **stem** feature contains the word stem of the token. For n-gram models, using stemmed words can contribute to a smoother n-gram that can better generalize to text that contains new forms of previously seen words.
  o The **pos** feature contains the Part of Speech tag for the token. Introducing this as a feature in an n-gram model can lead to better performance (since the n-gram can take into account the POS information in the n-gram).
  o The **orthographicShape** feature indicates whether the token's word shape is "allCaps", "upperInitial", "lowercase", etc. This information can be useful in n-grams, and also in regex patterns and named entity recognition.		

<table>
<tr><td>3. NGram Annotation: The system will annotate 1-, 2- and 3-grams of consecutive tokens.</td><td>Token</td><td>NGram</td></tr>
<tr><td colspan="3">

- At this stage, the NGramAnnotator will be called 3 times – each time with a different parameter indicating the degree of the n-gram (1-gram, 2-gram, 3-gram). Each instance of the NGramAnnotator will loop through each **Sentence**, and in each sentence will build **NGram** annotations based upon the **Tokens** in the sentence.
- Each time an **NGram** annotation is created:
  - The **start** feature is set equal to the **start** of the first **Token** in the **NGram**.
  - 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**.

</td></tr>
<tr><td>4. 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.</td><td>Question, Answer</td><td>AnswerScore</td></tr>
<tr><td colspan="3">

- At this stage, the AnswerScorer will loop through each **Question** (there should only be one in each document). For each Question, the AnswerScorer will loop through the candidate **Answer** annotations and send them to the scoring engine.
- 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).
- It is anticipated that the scoring engine will be parameterizable – that is, there may be an abstract scoring engine, and then by passing parameters, we can tell it whether to use machine learning models, rules, or other heuristics to return a score. However this engine is out of the scope for this data model, since it will be an instance of an Analysis Engine (and not something that would live as a CAS object).

</td></tr>
<tr><td>5. 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).</td><td>AnswerScore</td><td>N/a (text file or print results to screen)</td></tr>
<tr><td colspan="3">

- At this stage, the Evaluator will sort **Answers** according to their **AnswerScore.score** values. It will perform precision (and any other desired metrics), and will return the results either to a file or print to screen.
- It is anticipated that the Evaluator might be parameterizable as well – that is, by passing parameters we can tell it which metrics to generate. However this engine is out of the scope for this data model, since it will be an instance of an Analysis Engine (and not something that would live as a CAS object).

</td></tr>
</table>

These types and features (see Fig. 1 below) are almost exactly the same as the types and features shown in class. The main differences were as follows:

- First of all, I created a namespace, “qa” (for “Question Answering”). This will allow me to create other name spaces for other uses, and ensure that the types don’t get mixed up (i.e. qa.Token can have features different from ir.Token).
- I created an **AbstractAnnotation** type. This is the most abstract Annotation type in this namespace (see Fig. 2 below).
  - It contains two features:
    - **source** - the UIMA Analysis Engine (or other source) that generated this annotation.

        - **confidence** - the confidence level of the Annotation. Although there is no upper or lower bound on this value, the confidence level is expected to fall within the range [0.0, 1.0].
    - I created this type because most other Annotations need these features. In addition, this gives me a nice, future-proof way to add “universal” features – features needed by all Annotations – if future requirements dictate this.
- I created an **AbstractSpanAnnotation** type. This type inherits from **AbstractAnnotation**, and the main semantic difference is that this is specifically for types of **AbstractAnnotations** that cover text spans (as opposed to other media such as images or video).
    - It contains two features:
        - **begin** - the offset (from the beginning of the document) that indicates where the text span begins.
        - **end** - the offset (from the beginning of the document) that indicates where the text span ends.
    - I created this abstract type because all of the other Annotations (Sentence, Answer, etc) need these features. In addition, this gives me a nice, future-proof way to “fork” Annotation types based on modality. For example, if future requirements necessitate that we do question answering based on video or audio, then I would create an **AbstractVideoAnnotation** type (or **AbstractAudioAnnotation** type) to cover those modalities. In this case, **AbstractSpanAnnotation** would still give me the ability to have “universal” features for text-based annotations.
    - All of the Annotations listed above inherit from AbstractSpanAnnotation (which in turn inherits from AbstractAnnotation). As a result, all Annotations have four basic features:
        - **source**
        - **confidence**
        - **begin**
        - **end**
- Other than these abstractions, I did not create any additional abstract (or concrete) types. Following the iterative modeling principle of “Don’t guess”, I only abstracted (**AbstractAnnotation, AbstractSpanAnnotation**) where I knew there would be current benefits or likely future benefits – I did not try to anticipate as-yet-unknown changes to requirements.
- I did not create any types related to machine learning engines / algorithms, NLP algorithms, gazetteers, or ontologies. Several of those (gazetteers, ontologies) might be used as resources that could create Annotations in the future – if this were the case, they would fit neatly under the existing AbstractSpanAnnotation structure. And the others (machine learning engines / algorithms, NLP algorithms) would not be types that live as CAS objects – instead they would either be parameters to Analysis Engines, or actual code underneath the hood of Analysis Engines.

Figure 1: Types and Features

Type Name or Feature Name	SuperType or Range		Element Type
qa.AbstractAnnotation	uima.tcas.Annotation		
source	uima.cas.String		
confidence	uima.cas.Double		
qa.AbstractSpanAnnotation	qa.AbstractAnnotation		
begin	uima.cas.Integer		
end	uima.cas.Integer		
qa.Answer	qa.Sentence		
isCorrect	uima.cas.Boolean		
qa.AnswerScore	qa.Sentence		
score	uima.cas.Double		
answer	qa.Answer		
qa.NGram	qa.AbstractSpanAnnotation		
elements	uima.cas.FSArray		qa.Token
qa.Question	qa.Sentence		
qa.Sentence	qa.AbstractSpanAnnotation		
qa.Token	qa.AbstractSpanAnnotation		
word	uima.cas.String		
stem	uima.cas.String		
pos	uima.cas.String		
orthographicShape	uima.cas.String		

Figure 2: UML Class Diagram