

11-791 Project Individual 4:

Component and Type Design Patterns

Name: Ruochen Xu

Andrew id: ruochenx

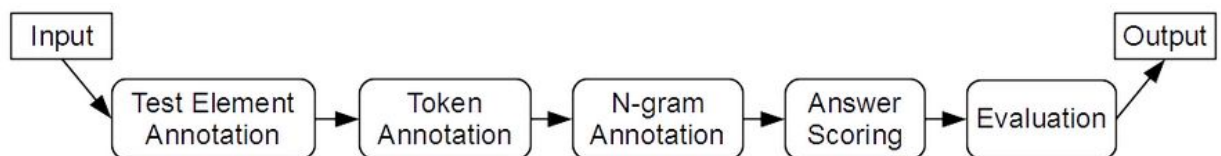
Logical Data Model Design:

I mainly used the given type system. The updating and extension are described below.

- I update all FSList features to FSArray, for that FSArray is easier to initialized and processed. For ranking method I follow PI3 to use n-gram overlap as similarity measure between question and passage.
- **Token**: Stores a token annotation. Inherits **ComponentAnnotation**.
- **Ngram**: Stores the information about an n-gram. Inherits **ComponentAnnotation**.
 - *n*: The number of tokens in this n-gram.
 - *tokens*: Tokens contained in this n-gram.
- **Score**: Store the score for all passages. Inherits **ComponentAnnotation**. In the final output file we want the passages are sorted by their score and grouped by their questions. To do so, I use a new type that contain a FSArray of **Question** instances where each question's passages(FSArray of **Passage**) have been scored with regard to this question. In this way it is very easy (in the Cas Consumer) to firstly ranks over all questions (by Id) and ranks passages within each question (by score).
 - *questions*: The questions whose passages have been scored.
 - *answer*: This feature stores the Answer type for the corresponding score.

Overall Features:

The system follows the following design diagram (without Evaluation):



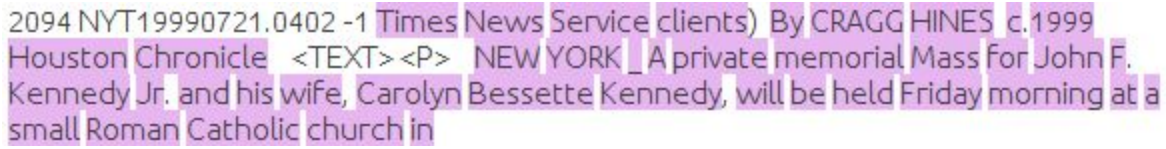
The input raw data is firstly processed by a very simple Collection Reader, just like PI3. Note that the collection reader just reads the whole document without any preprocessing. The JCas(after Collection Reader) is the input of an aggregated Analysis Engine, which includes test element annotation, token annotation, N-gram annotation and passage scoring. Each component is implemented by a primitive Analysis Engine and there is a aggregate Analysis

Engine that links all those primitive Analysis Engines. The output of the aggregated Analysis Engine is given to a CAS Consumer whose job is to output the evaluation result to file system in the format required in this PI. The evaluation part is not required in this PI so it is removed.

Each primitive Analysis Engine is well-defined and records the name of a component that produces the annotation and the component's confidence score assigned to the annotation. The intermediate results are defined mainly by very intuitive type system, therefore they are very easy to debug or visualize them using UIMA tools like Document Analyzer.

For the main annotators I mainly reused the ones in PI3. However the scalability becomes a major concern here since we have much bigger input data. In PI3 when I tried to find tokens(n-grams) in an annotation(question or passage), I simply iterated over all tokens(n-grams) to see which is inside the annotation. In this case I have $O(mn)$ complexity where m is number of questions and passages and n is number of tokens(n-grams). In PI4 this becomes the main bottleneck of computing time. To fix this I firstly sort all tokens(n-grams) and annotations by position, then determine which tokens(n-grams) an annotation should contain by iterating over two ranked list for only once. The complexity reduces to $O(m+n)$. The running time with all questions and passages is only around 10 seconds.

Analysis Engines(in flow order):

- **Question Passage Annotator:** Reads input document and output a **InputDocument** type annotation. This annotator is able to identify and record the question and passages with labels. The implementation mainly relies on Java regular expression (java.util.regex).
- **Token Annotator:** Reads **InputDocument** type annotation. Tokenizes all sentences in question and answers. The implementation does not rely on other tokenizer, instead I use a simple Java regular expression to extract tokens. The preprocessing is implemented in this annotator to remove html tags. When tokenize the text, this annotator ignores tokens in html tags(e.g. <P>, <TEXT>). Therefore those tokens will not affect the output of following annotators. The view from Document Analyzer indicates <P> and <TEXT> are ignored from analysis.

- **NGram Annotator:** Reads **InputDocument** and **Token** type annotation. With parameter *NumberOfNGram* indicating the length of n-grams, this annotator extracts tokens with specific length within sentence. This annotator firstly find all tokens inside a given

sentence, then it generates multiple n-grams (FSAArray of length *NumberOfNGram*) within each sentence.

- **Passage Score Annotator:** Reads **InputDocument** and **Ngram** type annotation. This annotator assigns a score with each passage by checking the overlap of n-gram between the given passage and question. The annotator iterates over all question-answer pairs and find corresponding list of **Ngram** type annotations inside question and answer, then with two lists of **Ngram**, it computes score for an answer by counting the overlapping **Ngram** between answer and question.
- **Aggregate Annotator:** This annotator links all previous annotator in fixed order. Note that it overrides the parameter *NumberOfNGram* in **Ngram Annotator** to enable passage of parameter from CPE.

Collection Reader and CAS Consumer:

For collection reader, I simply use the `FileSystemCollectionReader` in UIMA tutorial. For CAS Consumer, I modified `XmiWriterCasConsumer` in UIMA tutorial. The CAS consumer firstly ranks over all questions (by Id) and ranks passages within each question (by score).

Conclusion:

The major challenge for this PI is design data model and annotator for given data format. The given type system stores passages in group of question, which enables intuitive implementation of following pipeline. The preprocessing is crucial. In my implementation the preprocessing is postponed until token annotator, because the original text can be kept in **Question** and **Passage** in this way. Also the token is still based on the offset for original input document, therefore is easier for visualization and debugging. The ranking part requires ranking over question and ranking over passages for a given question, with **Score** described above this is trivially done by using JAVA build-in sorting algorithm.