Below I describe the outline of my type system design based on the UIMA framework for Java.

NAMESPACES

model.annotators - This namespace contains all the types of annotators that would be used in the information processing pipeline. All of the annotators extend the BaseAnnotator type which defines *source* and *confidence* as its features. They keep track of where the annotation was intially made, and how confident the annotation was, respectively. The BaseAnnotator type itself extends the UIMA built-in Annotation type which contains features *begin* and *end* which store the character offsets of the span of text to which the annotation refers. The TokenAnnotator type and the NGramAnnotator type have *tokenizer* and *ngramTokenizer* features, respectively, which tokenize a sentence into tokens or n-grams (1,2 or 3). The features use the NLP Toolkit (Open NLP, Stanford NLP etc.) employed by the Tokenizer type to execute their function. The Tokenizer type contains *tokenLength* and *tokens* as features which specify the length of the token (1, 2 or 3) and store the collected tokens in an array, respectively. The SentenceAnnotator type has features *isQuestion* and *isCorrect* which store whether a sentence is a question or not, and if not then whether the answer sentence is correct or not, respectively.

model.scoring - As the name suggests, this namespace contains the necessary types for scoring an answer sentence. The AbstractMetric type is an abstract type which other types in this package extend. The most basic scoring metric I use is the Bag-Of-Words given by the BagOfWords type. Some advanced statistical techniques such as *KL-Divergence* or *Mutual Information* can be used for scoring the answers. These are modeled by the ProbabilisticScoring type. Machine learning classification algorithms such as *Naive Bayes* and *Logistic Regression*, if trained wisely (proper feature selection) on good data (less noise), can assign a score between 0 and 1 to each answer sentence. These are modeled by the MLClassification type.

Note: We can also use a variety of other scoring metrics like cosine similarity, Jaccard's measure etc.

model.evaluation - This namespace only contains the Evaluator type. The job of the Evaluator type is to *sort* the scored answer sentences and produce *precision* at N, where N is the number of correct answers for the question in the given data. It also contains the *ids* feature to keep track of answer sentences during sorting by assigning them unique IDs (like 1, 2, 3 ....).

model.tools - This namespace contains the necessary tools required to execute various functions in the information processing pipeline. The ML type contains the Machine Learning toolkit (e.g., Weka) which is used by the model.scoring.MLClassification type. Similary, the NLP type contains the NLP toolkit (e.g., OpenNLP or Stanford NLP). The Tokenizer type uses the NLP toolkit to tokenize the sentences depending on specified token length (*tokenLength* feature) and returns the *tokens* in an array. The NLP toolkit can be used to perform several other feature extraction techinques such as Named Entity Recognition, POS-tagging etc.

model - The model namespace contains the Input and Output types. The Input type contains the *path* feature to locate the input file. The Output type contains the *display* feature to display the final output.

DESIGN PATTERNS

Singleton - For large scale processing where memory could be an issue, it is always wise to use the Singleton design pattern which allows the creation of only one object in the memory. The Input, Output and Evaluator types can be constructed based on Singleton. The sample Java code to construct such a design for a type is as follows:

```
public class Input  {

        private static Input singleObject;

        private Input (String n){

        }
        public static synchronized Input getInstance(String n) {

                if (singleObject == null){
                singleObject = new Input(n);
                }
        return singleObject;
        }

}
```

The synchronized constructor takes care of multi-threading issues which is an essential tool in large scale software projects.

Observer - If one needs to be notified of the change in precision everytime the input data changes he/she can register as an Observer to the Input type (Subject). This is called Observer design pattern which uses the Subject and Observer interfaces. By registering as an observer, whenever there is a change in the input data (and hence a potential change in precision), the object implementing the observer interface is notified of the change.

Decorator - The Decorator design pattern is all about extending the functionality of a given class. After you've written a class, you can add decorators (additional classes) to extend that class; doing so means that you won't have to keep modifying the original class's code over and over again. This is what I have done while creating the model.annotators namespace. The UIMA built-in Annotation type is extended by BaseAnnotator type, which in turn is extended by the other annotator types.

"IS-A" vs "HAS-A"

As is evident, my design is more "is-a" rather than "has-a". Both designs have their pros and cons, and one should design the architecture based on the application. The "is-a" design gives more functionality with less code writing. It is also more suitable for memory-intensive applications as the number of objects created are less overall. On the other hand, "has-a" designs may run faster because the main class or program doesn't have to create every object by itself. An example of a more "has-a" design could have been putting all the annotator types as features inside the scoring types (viz. BagOfWords). But, since the annotations would be stored in the CAS object, they can be recovered in the main class directly.

All the types and their features in this design, can be directly deployed in a main class to run the whole application.

NAME : DISHAN GUPTA
ANDREW-ID : dishang