11-791 Design and Engineering of Intelligent Information System

HW #1: Logical Data Model and UIMA Type System

Keerthiram Murugesan

Andrew-id: **kmuruges**

Requirement Analysis:

Specific to HW#1, we focus our attention to the types that are needed for the system. Based on this requirement, we can categorize the types into **I/O types** or **Intermediate types**. I/O types are used to represent the structure of the input and output data passed between the components in the pipeline, where as, Intermediate types are specific to the processing requirement of each component.

HW#1 relaxed the system input requirement by given each Question-Answer (QA) pair in a separate text file. But the system should also handle multiple QAs in a single text file, which is practical in real world applications. With this in mind, the input to the system is a raw text file with specific format given in the HW#1 requirement.

The intermediate processing step uses the NGrams of an answer sentence to measure how relevant it is to the question under consideration. Although, we don't need to focus on the processing requirement in this homework, we can define UIMA types based on the NGrams generated for each QA.

The output should contain an ordered list of answers for each question based on the score assigned by the system. In addition, the output should include a score metric (precision@K, where K is the total number of correct answers) to measure the performance of the system. In the next section, we will see how we can organize these requirements into a set of layers, each with certain functional goals.

Perspective: Layered Architecture

The sample information processing system given in HW#1 can be organized based on its algorithmic position in the architecture. It is easier to discuss the types used in the system in the perspective of layered architecture. As discussed in the requirement analysis, the system can handle multiple QA in a single input file. Based on the functional requirement for the system, we can see a layered architecture in the system. In this section, we define each layer along with its I/O and Intermediate types.

Notation:

Though out this document, we will use Type::Typename to specifically identify a type. All our I/O types are represented using Type::<Function>QAEntityList. When we see Type::<something>Question, or Type::<something>Answer, then this type is a subtype of Type::Question, or Type::Answer respectively (See domain-level types for more information about its features).

- *BaseAnnotation:*

Each annotation-based types should contain two features: source, confidence (as specified in the document). We created a BaseAnnotation (BaseTypeSystem.xml) with these 2 features. Almost all intermediate types (annotation) inherits BaseAnnotation to avail these features. This also has two important feature specific to Annotation: beginInd (to store beginning of string) and endInd (to store end of the string).

- *System-level/Domain-level Types:*

Before discussing about each layer, we need to explain the domain-specific/domain-level types: Base types. These are the types that are used throughout the annotation pipeline. In our system design, we have two such types. Question and Answer. These are the basic domain specific types that need to be accessed at each component in the architecture. Thoughout this document, we consider that each Question contains a set of Answers. It is easier to represent the types in this way so that we can just need a handle to Question instead of bosth Question and a set of Answers.

Type::Answer includes id, answerString features. The id feature uniquely identifies an answer, answerString contains the actual answer sentence.

Type::Question includes id, questionString and candidateAnswerList features. The id feature uniquely identifies a question, questionString contains the actual question sentence and the candidateAnswerList contains a list of Answer.

Tokenizer:

The raw input data given to the system is stripped into question (Type::Question) and answers (Type::Answer). To account for handling multiple QAEntity, we use a list to store them, in this way, we have a structured output, which can be easily passed down and processed in the pipeline. We created a list of Question-Answer Entity (QAEntity), one for each QA pair.

The QAEntityList should be tokenized so that the tokens can be handled individually in the further processing step. The tokenizer component takes care of this process. The output of this component is a tokenized question and answers. Each token is stored in Type::SimpleToken, which forms the basis for Type:TokenizedQuestion and Type::TokenizedAnswer. Similar to above discussion, the output is stored in Type:TokenizedQAEntityList.

NLP Task:

The output from tokenizer is passed down to NLP layer where the NLP related task are performed. We need to extract the semantic structure of the given QA. For simpilicity, we consider just the NGrams. We consider unigram, bigram and trigram of both question and answers constructed from their corresponding generated in the previous layer. Unigram is simply the tokens mentioned in the previous and has the similar set of types: Type::Unigram, Type::UnigramAnswer, Type::UnigramQuestion, Type::UnigramQAEntityList. Similary we defined types for Bigram and Trigram.

Model:

The model contains the core processing unit of the system. It takes the NGrams (Type::UnigramQAEntityList, Type::BigramQAEntityList, Type::TrigramQAEntityList) and compute a score for each answer and in turn for each question. The model could be a simple string matching operation between question and answer ngram or it could be a complex probabilistic model that maps the NGrams to [0,1]. Since we consider just the type system design, we simply included Type::ScoredAnswer, Type:ScoredQuestion, Type::ScoredQAEntityList. Each of these types contains a score feature (except I/O type) and the score feature in the Type::ScoredQuestion is the result of a function of score of its respective candidate answers (Type::ScoredAnswer); for e.g., max(ans1_score, ans2_score, ans3_score, …).

Evaluation:

Once the score for each QA pair is computed, we need to compute how good the system is. We use the precision @ K to measure the performance of the system. Type::ScoreMetrics contains the actual result of the measure and we can add more metric later in the development and the types defined can be easily extended. Type::EvaluatedQuestion contains the sorted answers based on the scores computed in the model layer, with the actual question (from the parent Type::Question), and the score metrics computed for that question. The output of the system is given as Type:QAResult which contains a list of evaluated questions.

Dependency of the XML files: (Inheritance)

The following shows the dependency structure of the descriptor files (both type system and AE files).

. *hw1-kmuruges-analysisengine.xml*
    -hw1-kmuruges-typesystem.xml
. *BaseTypeSystem.xml*
    -BaseAnnotation.xml
. *hw1-kmuruges-typesystem.xml*
    -TestElementTypeSystem.xml
        --BaseTypeSystem.xml
    -TokenTypeSystem.xml
        --BaseTypeSystem.xml
    -NGramTypeSystem.xml
        --BaseTypeSystem.xml
    -ScoringTypeSystem.xml
        --BaseTypeSystem.xml
    -EvaluationTypeSystem.xml
        --BaseTypeSystem.xml