Design and Engineering of Intelligent Information Systems

Homework1

Wei Zhang

Andrew id: weizhan1

Overview:

The type system UML design is shown in diagram 1. The principle behind this Object Oriented design is the abstraction of common concepts, high cohesion and low coupling, and generalization. This means reduction of duplicate definition among classes, decouple the weak conceptual containment, and make it as easy to future modification as possible.

The overall structure is trying to model the specific QA problem (question, answers) as well as making the greatest effort to generalize the type system according to the commonly known concepts in languages (sentence, token, NGram).

The whole package is named as hw1. The Base type is in the top level, and two sub folders nlp and qa represents the basic types for NLP input and output types, and basic types for Question and Answering Test elements.

The idea of this structure is obvious: I want to define a stand-alone set of NLP types that are decoupled from the QA problem. So, if we re-use the type system, we could expand the nlp types without worrying about the merging qa problem types.

The qa package defines several types, Problem class includes all the types as members. This class holds question and answers inside, along with some basic features for measuring and storing the evaluation results. The answer score is a type that contains the answer itself, along with the scores and ranks that are given by the evaluator.

Design details:

Almost in every class there will be some common features:

Integer : begin, which is used to denote the offset of the beginning position in the document

Integer : end, which is used to denote the offset of the end position in the document

String : source, which is the processor ID that created the instance of the class that it belongs.

String : confidence, could be generated in different ways according to the type it belongs to.

So I use uima.tcas.Annotation as the top level class, which contains the begin and end feature, and then inherit hw1.Base type which includes source and confidence. Next, make every type in this homework the subclass of hw1.Base. Then, every type in this type system includes those four features without trouble.

Note that, begin index and end index for tagSequence is different from those for the sentence. Because tagSequence is a sequence of tags generated by Parsers, and the span for each tag could be tagged from th

The types are explained in Fig.1 from the bottom to the top.

1. **Hw1.qa.problem** is aggregation of question and answers that is loaded from the input file, namely a Question and Answering Problem, which includes only one question and arbitrary number of answers in each QA Problem.
   One problem should have one question and more than one answers. And the question and Answers should not exist without a problem. Also, the AnswerScore relies on the existence of Answer. So, the Answer and AnswerScore is a one-to-one mapping, but AnswerScore has to know where the answer and question comes from, so that it could perform the measurement. Then, I choose to include answer in the problem class. Then, this AnswerScore is a big pool of instances, with docID labeling each of them. Then, if doing evaluation, we could iterate all the AnswerScores relate to problem, and get the PatN, and store it back to the Problem instance.

   The important features for Problem are:
   totalAnswers: used to record the total number of answers in this problem by far. This may be useful for doing statistics or as a normalization number.
   numOfCorrectAnswers: the total Number of the Correct Answers, the N in P@N.
   question: The question itself, which is a sentence type. I will explain this later.
   answers: The array of answers.
   PatN: The P@N. The problem is, We may not get only one P@N value, cause we are changing the evaluation method all the time, such as changing N, or the Annotator that we used to generate the score. I was thinking that, if we want to note down different configurations for generating the P@N, we can note them down in a place in the program where we created this QAProblem. So this is not a problem at all.
   docID: the unique file name(path).

2. **Answer and Qusetion**: Answer and Question are the instances of sentences. Sentence is the instance containing token array and NGram array. The difference between answer and question is that, Answer has an isCorrect field, to tell us the gold answer label. Question should not exist alone without a Problem, so does an answer.

   As a matter of fact, Question class seems not to be necessary here. Because, it does not add nay more information when it's created on top of a sentence. However, it is important because it contains a docID, which may help us shatter the Problem into Question and Answer sentences, without maintaining the Problem class itself. This argument may conflict the idea that Question should not exist outside the scope of a Problem. However, if we want to make a parallel program to handle large scale of question and answer pairs, this may be useful. Anyway, the question is created after the Problem is created. We are just copying the question out of the Problem type to attain this "separation".

3. **AnswerScore** is a subclass of the type Type.Answer, which is a sentence in essence. However, it contains additional information for the scoring, the features are:
   score: The score of the answer, which is automatically generated from the annotators (such us NGram overlap annotator, or token overlap annotator).
   rank: the rank that is generated from ranking the scores in the QAProblem. The reason for this rank field is for evaluation consideration.
   We all know that the ranking algorithm takes O(n*log(n)) time at most. For the first time ranking the answers, the time complexity could not be lower than this value. However, when we want to change the N value in P@N, we don't need to take as much time if we use rank. Just imagine if we use a smaller N, we are able to use O(n) to get the new evaluation.

   The problem should only contain answers, but not answer scores. For the answer scores, if we separate them with the problem, we have to keep track of where the answer comes from. So I added a docID field in the answer, then if we create an AnswerScore with an Answer in it, we could keep track of the Problem that it comes from.

   Another way to do this is to add AnswerScore as a member of Problem, instead of Answer type itself. In this sense, the problem contain not only the text and gold standard labels, but also the evaluation.

4. **nlp.Token** is the basic type for the character snippet type.
   The token won't exist if the sentence is not created. This means that, if no sentence, then no tokens, and no ngrams created. Although Token is a standalone class, in the UML I restrict Token to exist only as an array form in the sentence instance. The same for NGram.

5. **nlp.NGram** is a span type, but it has several features:
   numberOfElements: it's useful to tell the difference between 1, 2, and 3 grams.
   Count: the NGram could have duplicates in a NGramSet, so the Count is used to keep track of the duplicates in a NGramSet.

6. **Nlp.NGramSet** is an aggregation of the NGrams, this is used to group the NGrams of a sentence together. This should be a member of sentence. Note that each NGram in a NGramSet has the numberOfElements feature, so we can tell what each NGram is. Then we can put all the 1,2,3 grams into the same NGramSet, and if we want to find all the unigrams, we just check the numberOfElements field to get the unigram subset.

   There's another way to do this: We add a feature for NGramSet, gramtype =1,2,3, then we construct separately the Unigram set, Bigram set and Trigram set for the sentence. And we add the NGramSets into the Sentence. However, we have three NGramSets instead of 1.

   However, I choose to use the One NGramSet method instead of creating separate NGramSets. This will keep our type system look nicer and cleaner.

7. **Sentence:** This is the aggregation of the token and NGramSet. If the sentence does not exist, token and NGramSet should not, too. **Tokens** is the feature array that holds the tokens.

Discussions and Future Expansions

NGram could not only be made of Tokens, but also the NGram of the Tags, which I didn't define in this type system. Tags could be POS, Semantic Role, Entity Mention, …. In order to keep the type system as flexible as possible, I kept the elementType field for the NGram. However, we have to define a super type for token and tag, so that we don't need to create POSNGram, TokenNGram, EntityNGram,.. different versions of NGrams like this.

And also, I added a Language feature in the base type. This is very important to internalization. Just imagine if we are given a English question and a bunch of answers in another language, we have to make sure that the answer has been translated, and the translated text is properly stored in a class. Then we have to create a new sentence for any new answer.

Problems that I encountered

When designing the NGram and Sentence, I figured that they are all sequences, it may be more appropriate to have a Sequence type, which is the super type of both of them, so that I can inherit the element sequence that is defined in the Sequence type. However, Sequence should not be a concrete class, but interface. What I can do is to create a class Sequence, with a feature sequence : FSArray[Base]. However, there is no way to define a template or an interface in this problem.

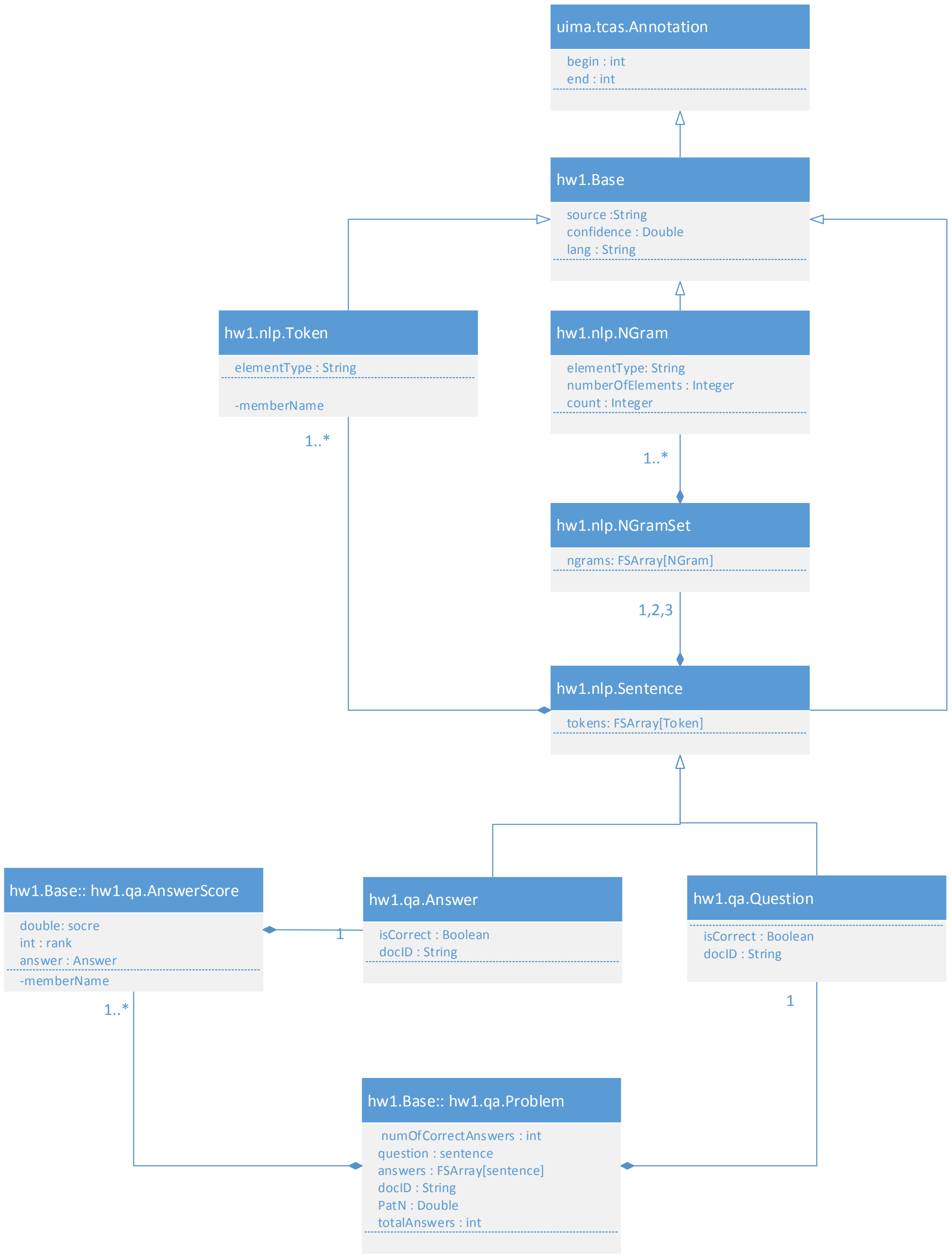

Fig. 1 UML graph for QA Problem