11791 Fall 2013 Assignment 1 Report

Yueran Yuan

Notes on formatting

bold is used for data types (when they first appear in a passage)
italics is used for fields of data types (when they first appear in a passage)
Implementation Decision:denotes a passage about a specific decision I made in the implementation that I think warrants some explanation.

Requirements Analysis and Generating Data Model

Just using the nouns that appear in the project description, we find that there are **questions**, **answers, test elements, tokens, ngrams,** scores, and precision among other things.

We note that questions and answers contained in a test element. Questions and answers all contain **sentences.** Tokens and ngrams are contained in the sentence.

Notably, scores and precision are numerical values and are properties of answers and test elements respectively.

Though this is not included in the project description, speaking with a 'client' (the TA) indicated that there is need for a **POS** tag perhaps later down the road.

We also note that the program requires an **Input** and an **Output** and that **Descriptions** are necessary for indicating the parameters for the various taggers we use.

After this initial analysis, we end up with a list of data types:

- **questions**
- **answers**
- **test elements**
- **tokens**
- **ngrams**
- **sentences**
- **posTag**
- **input**
- **output**
- **description**

Filling in gaps (i.e. when there is a one-to-more mapping) we get some additional data types:

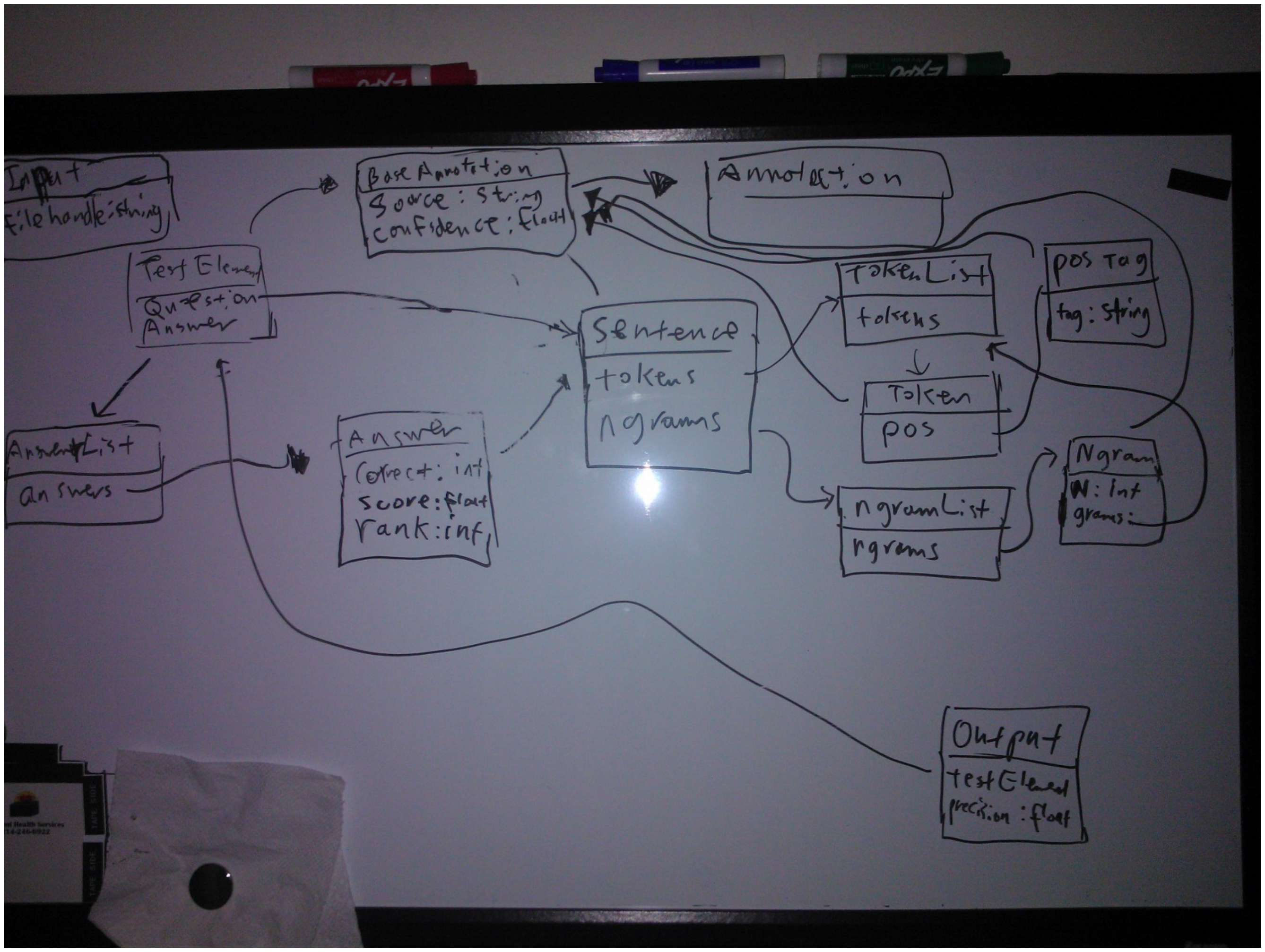
- **answerList**
- **tokenList**
- **ngramList**
- **programDescription (a structured collection of descriptions for various taggers)**

Implementation Decision: why use list structures instead of FSList? I chose to use these list structures because after seeing a similar structure during recitation, I spoke to the TA about it. He told me that the keep a separate list structure makes it easier to iterate. Though there is no reason to keep separate list structures currently, I still have them for future use.

In addition, to track the annotator which created the annotation (and to comply with the project spec) I added a **baseAnnotator** data type.

Implementation Decision: I did not include the string value of the content of the annotation in the baseAnnotator because though it would be quicker to keep the string value of content cached in memory, the solution does not scale with large data sets. By going back to the source text each time we need to read an annotation, we incur an overhead. We are effectively trading time for memory but memory is finite and time is infinite. If we use lots of memory, the program could halt when it runs out. If we use lots of time, we just have to wait a bit longer.

These structures and their relationships can be seen in the following UML sketch below (sketch does not contain description and programDescription)



The data is then placed into various packages. 4 main packages were used: view, model, nlp, and base.

- View consisted of input, output, programDescription, and description. This package holds the data necessary for other programs to communicate with this program, serving as an interface hence the name view (interface is a reserved word in java).
- Model consists of sentence, answer, testElement, and answerList. This package serves to model the structure of the test element.

- NLP consists of the token, ngram, and pos subpackages. This package holds all the nlp annotations which are computed from the testElement components and are given as features to the scorer
- Base consists of the baseAnnotator. Because the baseAnnotator is independent of other packages (and in fact the system) I kept it separate.

Data Flow/Pipeline

My system is a 7 step pipeline where intermediate information between the pipeline is stored exclusively in the **TestElement** data structure. Each pipeline phase stores information for use in the next pipeline phase by annotating components of TestElement.

1. Data Reader: **input** is given to the program along with an optional **programDescription**, the file is loaded and read and a **TestElement** is created. Initially, the *precision* field is empty. Likewise the *tokens*, *ngrams,* and *scores* fields are empty (to be filled by later steps in the pipeline). Various tokenizers and scorers are initialized with the parameters given in the programDescription (these are done as they are needed and not in the data reader phase; I am describing it here to avoid mentioning in every subsequent step)
2. Tokenize: TestElement is passed to a tokenizer which fills the *tokens* annotation of the *question* and *answers*. Starting from this step on to step 5, the processing on each **answer** could be done independently (and in parallel).
3. POS tag (optional): TestElement is passed to a POS tagger which uses various NLP techniques to tag the various tokens with a POS tag.
4. Ngram: TestElement is passed to an ngram tagger which creates ngrams from the token information contained in each *sentence* (i.e. the *question* and *answers*). These ngrams fill the *ngrams* field
5. Scoring: TestElement is passed to a scorer which computes a score for each answer given some match metric between the token and ngram features of the answer and the question. The score could be derived exclusively from the token overlap or the ngram overlap or a weighted average of the two. Each individual token and ngram could further be weighed differentially depending on the POS of the tokens contained within (if POS was computed).
6. Ranking: TestElement is passed to a ranker which sorts the answers by their scores. The index of the sorted list is then recorded under the *rank* field for each answer. *Implementation Decision: I decided to record the index to the sorted list instead of saving the sorted list itself. This is logically the same as using a system of pointers. I do this because I am unfamiliar with the implementation of FSList and do not want to incur extra memory load by duplicating the list of sentences with its associated overhead.*
7. Evaluator: TestElement is passed to an evaluator which counts the correct answers (which is N) then counts the number of answers of lower than N rank which are correct. It computes the precision with that information and returns an **output**.