Name: Yan Zhao

Andrew ID: yanzhao2

September 24, 2014

11791 HW1 Report

UML Design and Named Entity Recognition

Implementation with UIMA SDK

UIMA Architecture

In this task, we use CPE to read, annotate and consume gene words into the output file. CPE is called Collection Processing Engine. In my project, I construct a CPE including an Analysis Engine, a Collection Reader, and CAS Consumers. The whole architecture can be seen in the following picture.

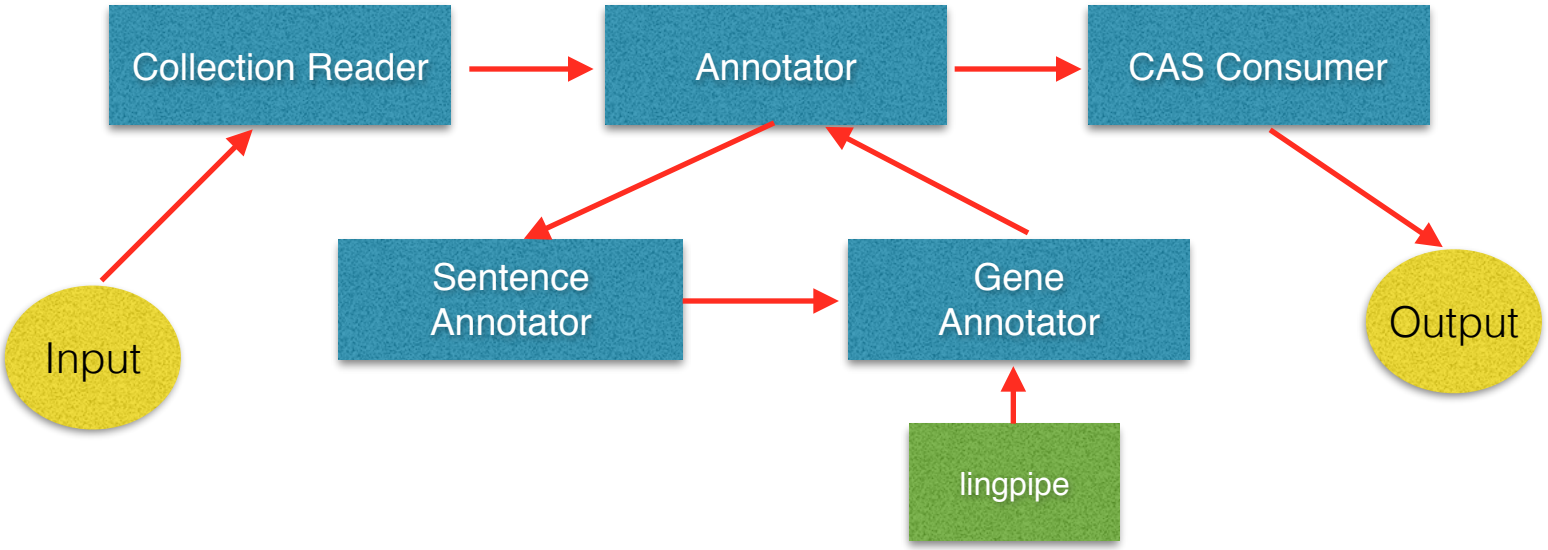

figure 1

The UIMA architecture requires that descriptive information represented in an XML file and provided along with the class files to the UIMA framework at run time. In the Type System Descriptor, I add two type *SentenceTag* and *GeneTag*, and I add *id* and *content* for each type. Then I add a Collection Reader Descriptor, two Analysis Engine Descriptors, an Aggregate Analysis Engine Descriptor, and a Cas Consumer Descriptor. And I generated an CPE named Result.xml.

A Collection Reader provides the interface to the raw input data and knows how to iterate over the data collection. A CAS Consumer extracts analysis results from the CAS and may also perform collection level processing, or analysis over a collection of CASes.

Here is some detail components of a CPE:

• Collection Reader – interfaces to a collection of data items (e.g., documents) to be analyzed. Collection Readers return CASes that contain the documents to analyze, possibly along with additional metadata.

• Analysis Engine – takes a CAS, analyzes its contents, and produces an enriched CAS. Analysis Engines can be recursively composed of other Analysis Engines (called an Aggregate Analysis Engine). Aggregates may also contain CAS Consumers.

• CAS Consumer – consume the enriched CAS that was produced by the sequence of Analysis Engines before it, and produce an application-specific data structure, such as a search engine index or database.

Collection Reader

The collection Reader is used to read the input file into the CAS. Here in my project, it mainly has two functions.

hasNext()

The hasNext() method returns whether or not there are any documents remaining to be read from the collection.

getNext(CAS)

The getNext() method reads the next document from the collection and populates a CAS. In my project, I read the input file wholly, not line by line.

Consumer

A CAS Consumer in my project receives each CAS after it has been analyzed by the Analysis Engine. In my project, it mainly contain one function.

processCas()

The processCas() method is where the CAS Consumer does most of its work. In my project, the XMI Writer CAS Consumer obtains an iterator over the document meta-

data in the CAS and extracts the URI for the current document. From this the output filename is constructed in the output directory and a subroutine (writeXmi) is called to generate the output file(hw1-yanzhao2.out). Also, in this function, I add some codes to calculate the precision and recall of my result, and F1-Score is calculated based on these two parameters. As can be seen, by comparing with the standard result, I get a precision of 0.815 and a recall of 0.810, thus get the F1-Score 0.813, which is very good.

Annotator

The first step in developing an annotator is to define the CAS Feature Structure types that it creates. This is done in an XML file called a *Type System Descriptor*. UIMA also defines the built-in types TOP, which is the root of the type system, analogous to Object in Java; FSArray, which is an array of Feature Structures (i.e. an array of instances of TOP).

The built-in Annotation type declares three fields (called Features in CAS terminology). The features begin and end store the character offsets of the span of text to which the annotation refers. In my project, I return the begin and end offsets without spaces, so I calculate the offset by removing the number of spaces in a sentence.

After saving a descriptor, the Component Descriptor Editor will automatically generate Java classes corresponding to the types that are defined in that descriptor (unless this has been disabled), using a utility called JCasGen. These Java classes will have the same name (including package) as the CAS types, and will have get and set methods for each of the features that define.

Then I add two annotator class into the project, one is anno_sentence and the other is anno_gene. These two classes extend the JCasAnnotator_ImplBase. These two classes mainly contain one function.

process()

This function in class anno_sentence is used to read content from CAS, split it by line, and annotate them by their ID and content, then transport them into CAS. This function in class anno_gene is used to read sentences from CAS iteratively, use *lingpipe gene library* to check if a string is genetic, and tore the ID and content of gene words into CAS by calling annotation.addToIndexes(). The relationship between these two annotator class and the aggregate annotator is in the following figure.

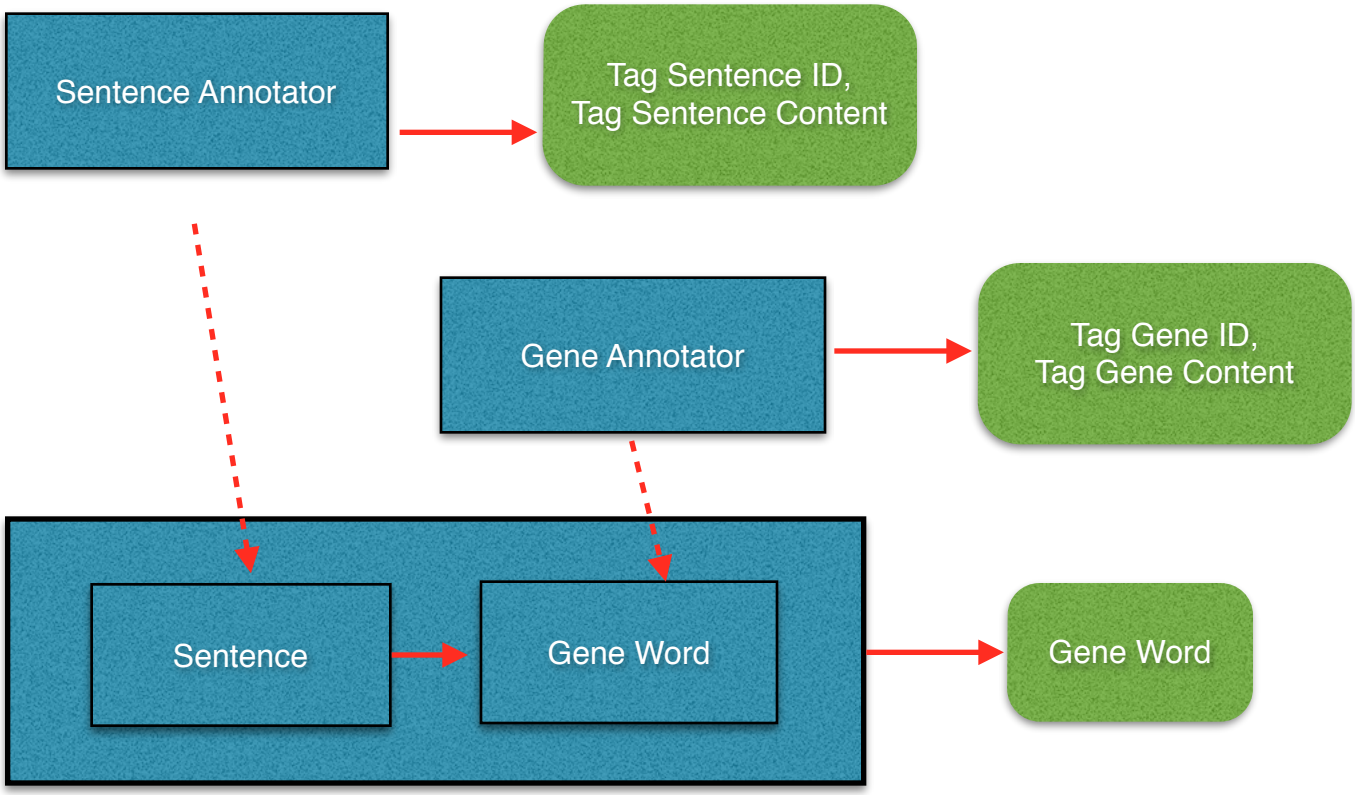

figure 2

Technology on Recognizing Gene

In my project, I use *lingpipe library* to help recognize Gene word. The program begins by reconstituting a ConfidenceChunker. Then I walk over the arguments, extracting a character array, and then providing it to the chunker, calling the nBestChunks method. With a confidence chunker, the result iterator is over chunks, so the cast is to (Chunk) in the body of the iteration. I then compute the confidence and after experiments, I set the confidence value to 0.6. As can be seen, by comparing with the standard result, I get a precision of 0.815 and a recall of 0.810, thus get the F1-Score 0.813, which is very good.

Future Improvements

1. I could add some Machine Learning techniques myself, to fulfill my project.

2. I could use some external training data to help to improve my performance.
3. I could read the file by line in the Reader so that to improve the runtime.