Homework 2 Report
Nikolas Wolfe
Andrew ID: nwolfe

Problem Overview

Using an Analysis Engine is an important first step to leveraging the power of the UIMA framework to do data analysis on an unstructured text corpus. In the previous homework, we learned how to set up a single Analysis Engine to do gene-based named entity recognition on medical text data. In this homework, we used an Aggregate Analysis Engine to do different types of analysis simultaneously and then combined the results to improve performance.

Architecture and Design

The basic UIMA pipeline we constructed for the previous homework more or less applies here. The difference is that we want to make a way for multiple Analysis Engines to independently analyze the data, add their Annotations to a CAS object, and then intelligently combine the output using a CAS Consumer.

This pipeline consists of three major steps. The first, predictably, is the use of a Collection Reader to read through an input file and load the data into a CAS object which can be passed to an Aggregate Analysis Engine. The second step is twofold. The first Analysis Engine I use leverages the Lingpipe HMM Part-of-Speech (PoS) tagger, and a second one uses the ABNER Named Entity Recognizer trained on the BioCreative corpus. These two steps are largely straightforward and did not involve anything beyond creating the basic descriptor files and extracting the appropriate annotations using the PoS / NER Analysis Engine components. The final step is the combination step, which takes places entirely inside a CAS Consumer, where I divide the data in the CAS into data structures specific to each preceding Analysis Engine and then combine them once the CAS processing has been completed.

The noteworthy aspects of my design are, first, my use of a Java Enumerator to create a layer of indirection between the parameters specified in the UIMA component descriptor files and the classes that invoke them to initialize objects and/or open files. Typically, people have been invoking UIMA parameters directly. Supposing I wanted an input file parameter inside of an Annotator, I might do the following:

```
String filename = aContext.getConfigParameterValue("inputFile").toString();
```

This seems perfectly innocent however the problem remains that if someone changes the name of this parameter in a descriptor XML file, things break. To get around this I created an Enum class called UIMATypeEnum.java. Using this Enumerator, the code snippet above which extracts the filename becomes:

```
String filename =
aContext.getConfigParameterValue(UIMATypeEnum.HMM_MODEL.getParam())
              .toString();
```

In this case, "modelHMM" and "inputFile" refer to parameters declared various descriptor files. If ever I should add or change another parameter in another UIMA component, I only need to alter this one Enum class, and none of my code will break.

The reason for this is simple: Hard coding a mutable String inside of a fixed class is bad design. It is important to localize mutable state in a well-designed software architecture, and creating a situation in which someone may one day have to dig inside the code of a particular Analysis Engine just to change the value of a String parameter is a recipe for disaster.

Another design consideration has to do with the reusability of my CAS Consumer. There are admittedly a few problems with the design. Since I only used two Analysis Engines I hard-coded the functionality to divide the CAS data based on the Annotator that produced it. To make a truly reusable CAS Consumer I would have to make this behavior more generalized. I did however allow for my Annotation types to be extended without changing the Consumer that uses them. My CAS Consumer NERCasConsumer.java is generic insofar as the parameterized type extends the NamedEntityAnnotation class. This will allow future implementations to subclass my Annotation types if necessary.

My data combination algorithm is fairly straightforward. I know that ABNER performs worse than Lingpipe across the board so I only wanted to use ABNER to corroborate the Lingpipe annotations. Essentially my strategy is a simple intersection between Lingpipe and ABNER, except that if Lingpipe found more Annotations in a given sentence than ABNER I use all of the Lingpipe annotations without question. Finally, upon analyzing the data I realized that Lingpipe often Annotates single characters as gene mentions. I prune these completely in order to improve precision.

Type System

All of my descriptors extend the deiis_types.xml file provided in this homework. I also created my own Annotation to add to this which is called NamedEntityAnnotation, and this is what I use to annotate in my Analysis Engines.

Results

I was able to improve the results of the Lingpipe annotation baseline on the provided sample data. Combining with the results of the ABNER annotator, I get the following:

```
Precision: 0.796
Recall: 0.827
F1: 0.811
```