11791 Homework 3 report - Execution Architecture with CPE and Deployment Architecture with UIMA-AS

Zhengzhong Liu
Language Technology Institute
Carnegie Mellon University
Pittsburgh, PA 15213

1 CPE execution

The first task in this homework is to execute the annotation pipeline with a Collection Processing Engine (CPE). A basic CPE will contain 3 main components, including the collection reader, the analysis engine(s), and the cas consumer(s). The following sections outline the design and implementations of these components.

1.1 Collection Reader

A collection reader is supposed to be able to read from a general source of input, which abstract the downstream pipeline with the I/O details. In this homework, the requirement is simplified to handle only file system reader. There is an implementation from the UIMA example project named FileSystem Collection Reader, which convert each file in a file system as a CAS. Despite from the basic DocumentText attribute, this implementation also add a ”SourceDocumentInformation” type that store the file uri and and size information for future usage. My work here involves to put a descriptor for this reader, the descriptor is also adopted from UIMA example implementation, which add the SourceDocumentInformation type into the descriptor’s type system.

1.2 Cas consumers

Task 1.2 in this homework does not involves developing new annotators. However, there is a need to convert the evaluator component from the last homework into a Cas consumer. Basically, it means to change the extended parent class to CasConsumer_ImplBase. Methods in the consumer is quite similar to what we have in a analysis engine, with some changes in method signatures.

1. The ”process” method is changed to ”processCas” and the parameter has become a ”CAS”, the exception is also different. One simple modification in order to adapt this method is to simply get the JCas from the Cas provided, and the rest of process is exactly the same with the analysis engine implementation.
2. Another item that is used in the evaluator analysis engine is the collectionProcessComplete method, here there is also a change the parameter and the exceptions thrown by the method. After changing to the correct implementation, this method works normally.

It seems that the Consumer is designed to process a more general case, where the input could be from various sources. Thus it take a CAS data structure that could even cross different languages. Although the functionality is quite similar to an analysis engine, conceptually, it take a different role in the whole pipeline and act as a downstream information consumer instead of creator.

I’ve done a little additional work here to use another cas consumer, the XMI writer, which is also adopted from the UIMA example, for the purpose of writing out the results for debugging.

1.3 Running with CPE

FInally, a CPE is used to connect up all the components. This step is rather straightforward. After getting the CPE GUI, we can picked for each step what descriptor to use. On the consumer part, I put two consumers, the evaluator consumer and the writer consumer. The CPE GUI also provide a saving function to save the configuration as an XML.

The whole pipeline then can be run independently. CPE provide a nice way to easily plug in new functionalities for future scale up.

2 UIMA-AS

A UIMA-AS is designed for deploying service for clients to call remotely. The UIMA-AS project essentially contains the UIMA project, but add some dependencies to use the ActiveMQ broker to provide remote services like JMS. The setup for UIMA-AS is simply changing the UIMA_HOME environment variable.

2.1 Using a remote service

The first task is to simply run a client to call the Stanford CoreNLP annotator. Here we need to write a UIMA-AS-client descriptor. I copied a descriptor from the UIMA-AS examples and modified it to call the "scnlp" service. This step is quite straight forward, and using the descriptor is the same as using a normal aggregate analysis engine descriptor. The design of UIMA make it transparent to users which make remote service could be handled like a local one. But there is one thing here is that using a remote service we at least need to be award of the type system they used. One possible solution to make this even simpler is to establish a type system server for others to get them.

2.2 Using Stanford CoreNLP

The CoreNLP provides a lot of annotations, such as Entity Mention, POS, Lemma and Coreference. It is first required in the homework to use entity mentions. However, this type is already used in the previous homework. The performance has changed from 0.56 to 0.78 in terms of precision at N. I also use the remote services to get the annotations, the annotations are almost the same, except that the ClearTK implementation includes the Stanford Coreference engine, which annotate more named entities. The named entities provided by the coreference engine are quite noisy so I have to filter them. Filtering is simply done by removing the named entities that does not contains a named entity type. The final outcome is then the same as using a local Stanford annotator.

The time of running the remote service only takes about 1095 ms and the time for running local annotator takes 1258 ms. The running time is similar but the service call require less time. We could see that the power of this remote call is that we could use a larger computational resource and the initialization time for a local annotator can be avoided.

Additionally, I also tried to use other Stanford types as the Bonus questions requires, such as Lemma and POS annotator. With my cosine similarity framework, these bonuses can be easily adopted by considering them as bag of words. The lemma and pos combination (such as "Lincoln-N") is put into another bag of word. However, this time the performance is not improved. This is not surprising, because in order to select the rest of the correct answers, one need to consider dependency or semantic parsing, as well as word sense similarities.

2.3 Deploying my services

Following the UIMA-AS documentation, I also deployed my own aggregate engine as a service. To make the deployment successful, one need to output the dependencies of maven. I did this by using the maven copy dependencies plugin. Then the UIMA_CLASSPATH is set to the output dependencies folder and the project target folder.

2.4 DBpedia

I also deploy a DBpedia Spotlight service as a stand-alone service beside the homework. The code can be found at ”https://github.com/hunterhector/DBpedia-uima-as-service”. A version also is deployed to the maven repo under the homework section. This service wrap a simple DBpedia annotator that calls the DBpedia spotlight remote server by sending the document text. The DBpedia spotlight server will annotate the text into disambiguated DBpedia resources. My annotator then get the resource uri and query the DBpedia sparql endpoint to get more information about each resource. However, because there are no specified information need, I only ask for an ”abstract” for each DBpedia resource. These resources are annotated on the JCas and returned to the user. In order to prevent excessive call to the DBpedia server, I waited 1 second after each call. Before deployed onto Maven, I tested it locally, and there is one other classmate call my service successfully.