Execution Architecture with CPE and Deployment Architecture with UIMA-AS

Name: Xiang Li Andrew ID: xiangl2 E-mail: lixiangconan@gmail.com

1. Task 1

1.1 Task Requirement

In this task, we need to use a CPE instead of a UIMA Document Analyzer to run our UIMA pipeline. Using a CPE could help us build a more flexible framework and process the analysis result better.

1.2 Overall Structure

When using a CPE, we need to add additional component to the project and combine them with the original Analysis engines. For this task, we do not need to add many things. The overall structure of the system after using the CPE is shown in the figure below:

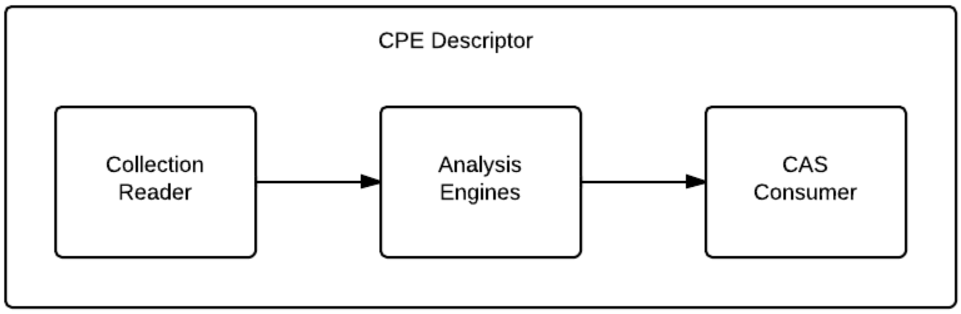

Figure 1 Overall Structure of the System

In the system, the Collection Reader reads input documents from the file system and convert them into JCas instances. Then, the Analysis Engines process the input files and generate several annotations. Finally, the CAS Consumer collect the information offered by the annotations and summary the performance of the system. The overall structure of the pipeline is defined in the CPE Descriptor.

1.3 Component Details

To implement the system, we need to build several java classes and xml files.
The Analysis Engine part is just the same as the Analysis Engine built for UIMA Document Analyzer, so we do not need to modify it.
The CAS Descriptor is a simple xml file (hw3-xiangl2-CPE.xml), which records

the components used by the system and other information such as input path and output path. We can use UIMA Collection Processing Engine Configurator to generate and run this file.

Since this task only requires reading input files from the file system, we can use the FileSystemCollectionReader offered by UIMA framework for the Collection Reader part. As other UIMA components, we need to add both the real class and the descriptor file of the class to our system.

The CAS consumer replace the original Evaluator part in the aggregate analysis engines. This component mainly does two things. First, each time when the system finishes analyzing a document, the CAS consumer collect the information in the Annotations and output answers and precisions. This is just the same with the Evaluator. Second, after all input files being analyzed, the CAS consumer also output the average precision. This helps us understand the performance of the system better.

2. Task 2

2.1 Task Requirement

In this task, we need to call a remote UIMA-AS service and integrate the annotation result into our answer scoring component

2.2 Basic Steps

a) For this task, we should firstly create a client descriptor (scnlp-xiangl2-client) for the remote UIMA-AS service (Stanford CoreNLP). The descriptor mainly specifies the endpoint, broker URL and timeout configurations. Then we can simply use this descriptor to connect the remote service.

b) After creating the client descriptor, we need to combine it with our original pipeline implemented in HW2. To do this, we can modify the CPE descriptor in Task 1 and call the remote Stanford CoreNLP service before call our local pipeline. Then our pipeline can use the annotations made by the remote service to improve its performance.

c) Finally, we should modify the answer scoring component in our pipeline to use the Name Entity annotations made by the remote service. To do this, we should first add the type system of cleartk toolkit to our analysis engine. Then, we can use the Name Entity annotations just like using the N-Gram annotations. That means we now use both the N-Gram annotations and Name Entity annotations to calculate the overlap rate between question and answer, while in the past we only used the N-Gram annotations. In addition, we give Name Entity annotations heavier weight, since there are relatively less Name Entity annotations in the sentences and these annotations should be more important.

2.3 Experiment Result

After integrating the Name Entity annotations into our original pipeline, we did an experiment on the two input data. However, we found that the average precision decreased after integrating the Name Entity annotations. In particular, one input data got the same result as before, and another input data got worse result. This result is quite reasonable, since most sentences in the input data have similar Name Entities, so just using Name Entity annotations may not improve the matching result. In addition, the Name Entity annotations made by Stanford CoreNLP could be seriously interfered by the tags, including ‘Q’, ‘A’ and golden scores in the input file.

2.4Bonus

Finally, we can compare the speed of Stanford CoreNLP annotator running on local machine and remote machine. To do this, we should create an analysis engine descriptor (scnlp-xiangl2-local.xml) for StanfordCoreNLPAnnotator. Then, we can modify the CPE descriptor to use this local annotator rather than the remote one and compare their speed.
When running the annotator on local machine, we found that this annotator is extremely space consuming. In fact, we need to change the max memory size for Java virtual machine to more than 4GB to avoid the out of memory error. In addition, the analysis engine also need a lot of time to extract essential files.
Here is the experiment result. When using the remote annotator, the whole pipeline, including the Stanford CoreNLP annotator and our own aggregate analysis engine, used totally 1297ms to deal with the given two input file, while the remote annotator used 1124ms (and the “End of batch” phase after calling the remote service used 110ms). When using the local annotator, at the first time we ran it, the whole pipeline used totally 3810ms, while the Stanford CoreNLP annotator used 3708ms. At the second time we ran it, the whole pipeline used 898ms, and the annotator used 842ms. This result shows that the local annotator could ran a little bit faster than the remote one. What’s more, it also saves the communication time. However, the local annotator would takes much longer time when called at the first time, perhaps because it needs additional time to prepare essential data.

3. Task 3

3.1Task Requirement

In this task, we need to deploy the aggregate engine on local machine, and then call the aggregate engine as a service.

3.2Detailed Steps

a) For this task, the first step is to create a deployment descriptor (hw2-xiangl2-aae-deploy.xml) for the aggregate engine. The deployment descriptor specifies aggregate engine descriptor, the endpoint and the broker URL.
b) After that, we should start a UIMA-AS broker locally and deploy the service to

the local broker. Before doing this, we should firstly add additional dependencies to local UIMA-AS. Therefore, we can use dependency: copy-dependencies plugin in maven to copy all dependencies used by our analysis engines to a local dictionary. We also need to export java classes for our own annotators to a single jar file and copy that jar file to the same dictionary. Then, we can add that dictionary to environment variable UIMA_CLASSPATH, start the UIMA_AS broker and deploy the service using the deployment descriptor.

c) Next, we need to create a client descriptor (hw2-xiangl2-aae-client.xml) to access our local service. This descriptor defines the endpoint and broker URL as the same value defined in the deployment descriptor. It also defines some values to deal with the timeout situations.

d) Finally, we can create a CPE descriptor to call the local service and combine it with the Collection Reader and CAS consumer. This descriptor is similar to the descriptor we create for task 1, except that it uses the client descriptor rather than aggregate engine descriptor to call the analysis engines.