11-791 Design and Engineering of Intelligent Information System

HW #3: Execution Architecture with CPE and Deployment Architecture with UIMA-AS

Keerthiram Murugesan

Andrew-id: **kmuruges**

Requirement Analysis:

In the previous homework, we get to create an aggregate pipeline for our QA system. In this homework, we need to three simple tasks over the aggregate pipeline we designed for the previous homework (See Figure 1).

Task 1 involves us to encapsulate this pipeline with Collection Procession Engine (CPE) with Collection Reader and CAS Consumer (See Figure 2). This task allows us to use the default file reader that comes with the UIMA library (FileSystemCollectionReader). All we have to do is to create the CAS Consumer and the CPE descriptor file.

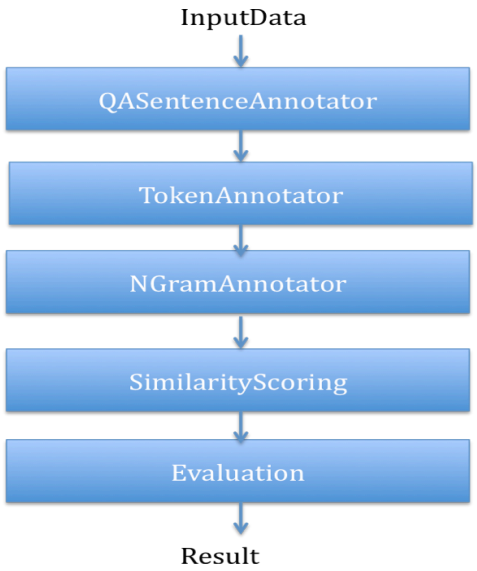

Figure 1: Aggregate pipeline from HW2

Task 2.1 extends our previous work to use a remote NLP service in our pipeline by using UIMA Asynchronous Scaleout (UIMA-AS) library (See Figure 3) and Task 2.2 requires us to deploy our pipeline created in hw2 as a remote service (deploy locally, see Figure 4).

The main requirement analysis in this task involves gathering knowledge about how these tasks fit in the UIMA architecture (in terms of descriptor file). The following figure shows the main components of the Task 1, 2.1 and 2.2.

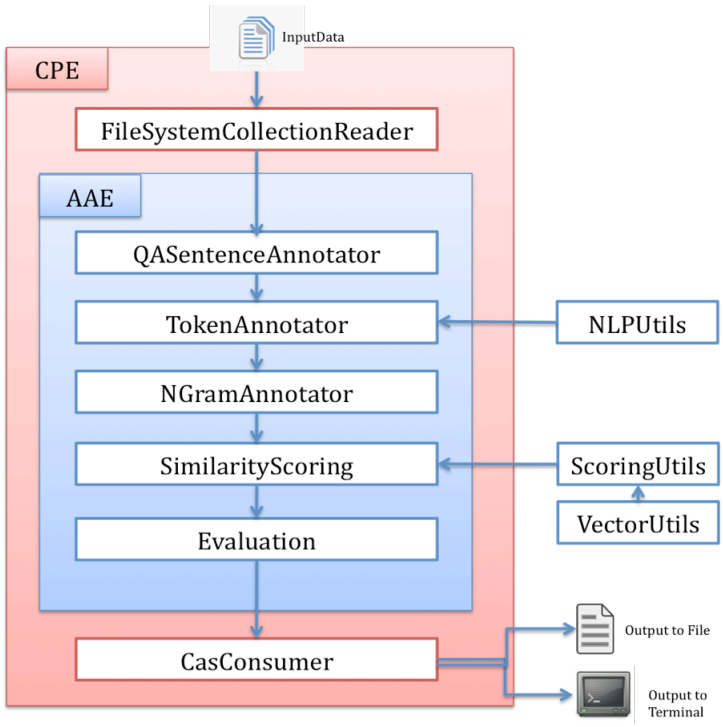

Figure 2: Component Diagram for Task 1

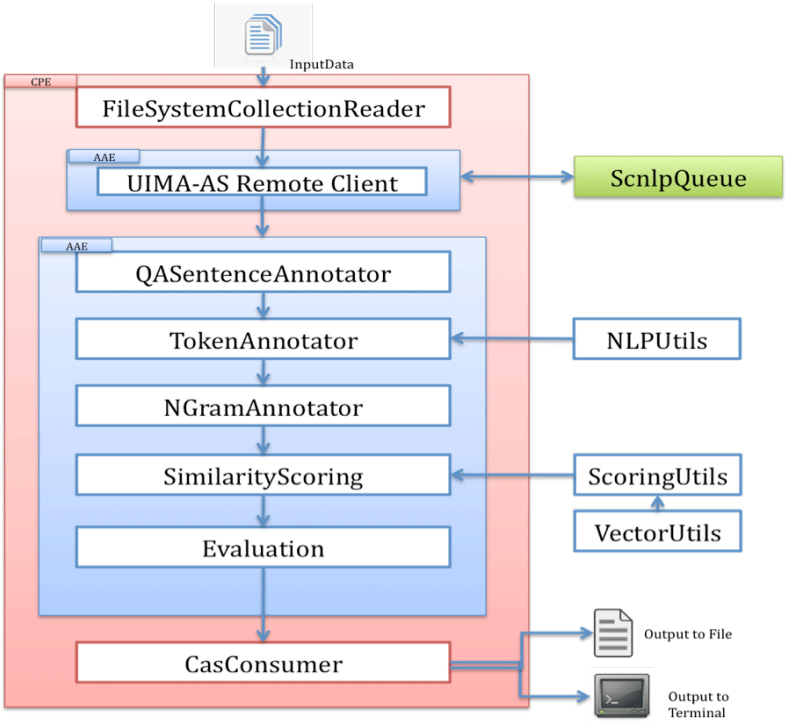

Figure 3: Task 2.1 with UIMA-AS Client to remote service

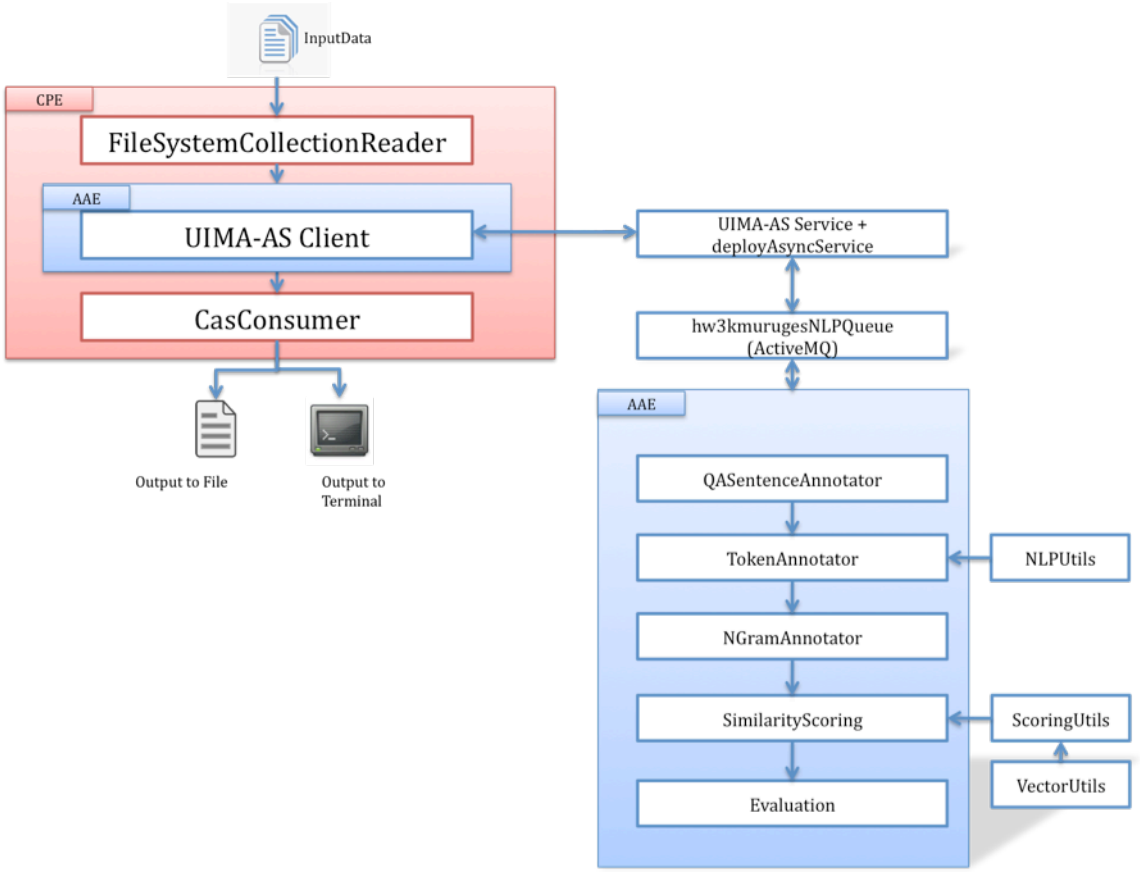

Figure 4: Task 2.2 with locally deployed hw2 pipeline with UIMA-AS service

Perspective: CPE engine and UIMA-AS Library

The entire processing pipeline in Aggregate Analysis Engine (AAE) needs are dedicated to creating annotations for the given input document (QA Pair) and use NLP/ML tools to compute how similar an answer is to the given question. The annotations created throughout the pipeline are identified using casProcessorId field in the custom Annotation class.

Each Annotation also contains confidence parameter, which is used in the Scoring phase for weighted scoring. More confident the annotation is good for computing similarity, the confidence score will be higher. Figure 5 shows the Input/Output types used in the system. We use *UIMAFit* to access the annotation types in each annotator of the pipeline.

Collection Reader: Like we mentioned in the requirement analysis, we will the default FileSystemCollectionReader provided by UIMA library to read the QA pairs in the text documents.

Sentence Annotator: It extracts the Question and Answer annotation from the raw input text. The annotation contains the entire sentence including the sentence indicator (Q or A). This is especially useful in the next phase when we need to compute the start and end

index for the token annotation. It uses the regular expression to find whether the sentence is of type Question or Answer. The *regex* pattern is passed to the annotator using the UIMA configuration parameter to enable easy configuration in the future.

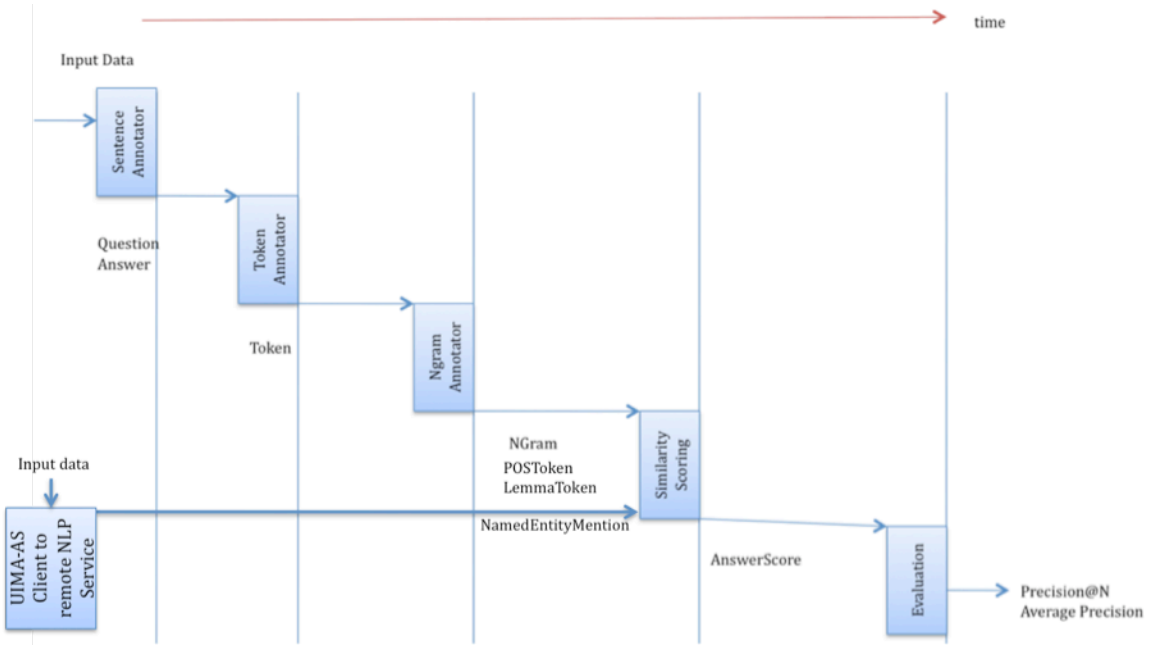

Figure 5: Activity Diagram with I/O and Intermediate types

Token Annotator: The Question and Answer inputs from previous annotator are used to generate tokens. We use Stanford CoreNLP library to tokenize/POS tag the sentence. We remove the Q or A tag that is used to identify the types before the annotation (using indices). We especially consider the token that splits words like *doesn't* into *does* + *n't.* These tokens are then added to the annotation index.

NGram Annotator: This annotator uses the tokens generated from the previous component to generate the unigram, bigram, trigram tokens. It removes any tokens that contains (dot) or (comma) to avoid any unnecessary count to the final score. Each NGram contains a set of tokens based on whether it is unigram, bigram or trigram.

* Average of the scores given to the correct answers (This score is different from the Precision@K score)

Similarity Scoring: This component uses the NGrams generated in the previous step to compute the similarity between the question ngram tokens and the answer ngram tokens. The same is repeated for POSTokens and LemmaTokens. In task 2.1, we get the NamedEntityMention types from the remote NLP service. We compute scores for these NamedEntityMention types just like other types. It uses the NGram Overlap Scoring methods mentioned in the slides using ScoringUtils as a ***delegate*** class.

It follows a simple *bag of word* model. Each operation (such as unigram, trigram, NamedEntityMention etc.) type is scored separately and the set of these scores are merged using the weighted score function that gives different weight to different

operation. The operation weights sum to one (the code automatically forces this constraint). We also use simple heuristics to check the score metric is computed correctly (such as the number of negation, SVO check, etc..).

Evaluation: This component simply computes the Precision at N from the AnswerScore types generated in the previous step. It sorts the answers based on the computed score and the precision at N metric is computed. Finally, it creates the QASystemOutput I/O types to be used by the CASConsumer specified in CPE.

CAS Consumer: The CPE CasConsumer uses the QASystemOutput types created by the Evaluation phase (one for each QA pair) and sends the results to both output terminal and output file.

UIMA-AS Client Descriptor: UIMA-AS client (scnlp-kmuruges-client.xml or hw2-kmuruges-aae-client.xml) simply call the remote service mentioned in the brokerURL and endpoint parameters.

Deploy Descriptor: This xml descriptor file will be used by deployAsyncService to create a remote service for other clients.

Bonus Questions:

This project also uses POS and Lemma features from the Stanford CoreNLP for scoring an answer. POSToken and LemmaToken are subtypes for Token type.

System/Measure	Speed (in sec)	Average Score (QA Pair - 1)*	Average Score (QA Pair - 2)*	Accuracy (Average Precision)
HW2 AAE Pipeline	1.109	0.6675	0.6867	0.75
HW3 CPE with Stanford CoreNLP Remote service	1.923	0.6675	0.6867	0.75

Figure 6: Comparision of HW2 System and HW3 System