11-791

Design and Engineering of Intelligent Information System Fall 2012

Homework 1

Report

Name: Lavanya Viswanathan

ID: lviswana

Contents

1. Introduction

In this task the objective is to implement a named entity recognizer using the UIMA framework. Here, the named entity recognition (NER) is performed on genes, and is termed *Gene Mention Tagging*. Gene Mention Tagging task is concerned with the named entity extraction of gene and gene product mentions in text. For example, given a sentence from a biological literature: Comparison with alkaline phosphatases and 5-nucleotidase, we should be able to identify alkaline phosphatases and 5-nucleotidase correspond to gene names.

1.1 Terminology

Here, we look at the various terms that we will be using throughout this task. Firstly, UIMA, Unstructured Information Management applications, are software systems that analyze large volumes of unstructured information in order to discover knowledge that is relevant to an end user. An example UIM application might ingest plain text and identify entities, such as persons, places, organizations; or relations, such as works-for or located-at. UIMA is also adept at handling multimodal inputs like audio, video etc. But the focus of this task will be on text. Given below is a figure that illustrates how UIMA serves as a bridge between structured and unstructured data (source: http://uima.apache.org/d/uimaj-2.3.1/overview_and_setup.html).

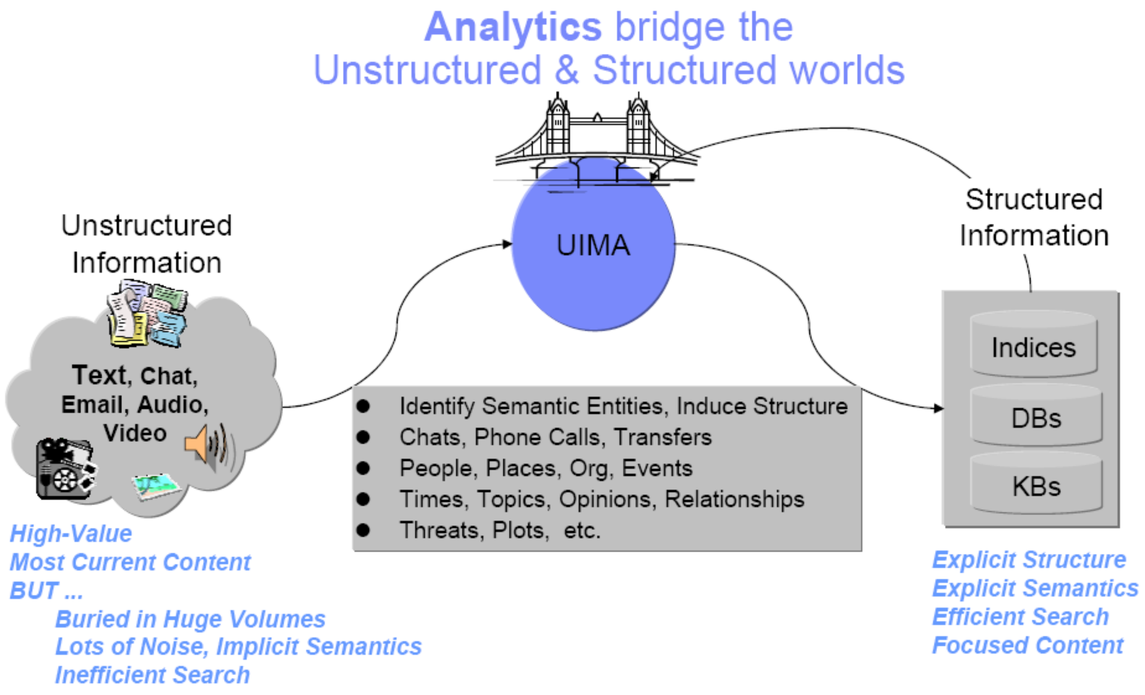


Figure 1

Now let us look at the specific components we will be working with to develop our code with UIMA. The major ones are the Collection Processing Engine, Collection Reader, CAS Consumer and Annotator. Let us briefly examine these.

All feature structures, including annotations, are represented in the UIMA Common Analysis Structure (CAS). The CAS is the central data structure through which all UIMA components communicate.

A Collection Processing Engine includes an Analysis Engine and adds a Collection Reader, a CAS Initializer (deprecated as of version 2), and CAS Consumers. The part of the UIMA Framework that supports the execution of CPEs is called the Collection Processing Manager, or CPM.

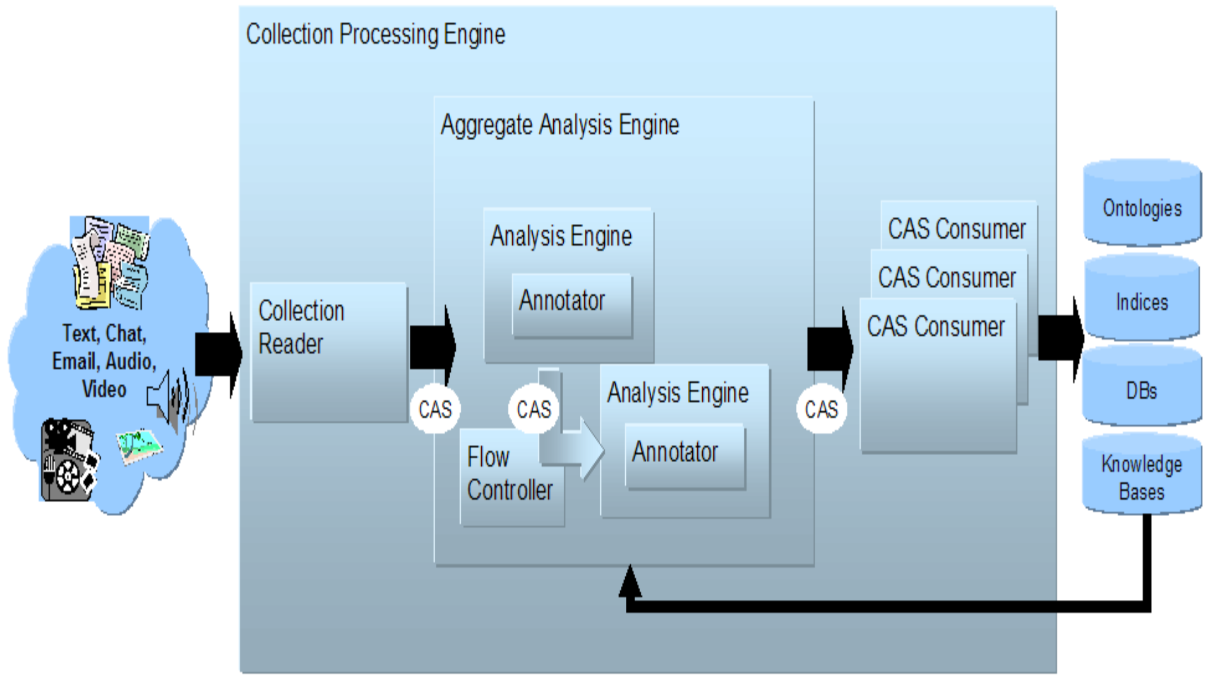
A Collection Reader provides the interface to the raw input data and knows how to iterate over the data collection.

A CAS Consumer extracts analysis results from the CAS and may also perform collection level processing, or analysis over a collection of CASes.

An Analysis Engine (AE) is a program that analyzes artifacts (e.g. documents) and infers information from them.

1.2 Description

Give below (in Figure 2) is how the above-described components are connected and how they form a smooth pipeline (http://uima.apache.org/d/uimaj-2.3.1/overview_and_setup.html).



The pipeline described here is as follows. We have some unstructured data (text, audio, video etc.). This is first read by the Collection Reader and converted to a CAS. Next this is passed to an Analysis Engine (AE). This could be simple (with only one Annotator) or could be aggregate (with multiple Annotators). So the AE produces the annotations, which are then fed into the CAS Consumer. We could implement CAS Consumers in different ways, like persisting in a database, writing into a file etc. This is the pipeline flow that has been followed in this task as well.

2 Approach:

2.1 NLP techniques/components used:

For this task, I made use of LingPipe (http://alias-i.com/lingpipe/), to handle the annotation segment. LingPipe is a tool kit for processing text using computational linguistics. LingPipe's architecture is designed to be efficient, scalable, reusable, and robust. Significantly, it has many modules for NER like Rule-Based Named Entity Detection, Exact Dictionary Based Chunking, Statistical Named Entity Recognizer etc. I made use of the Statistical NER for this task, specifically "First-Best Named Entity Chunking". The facility provided by LingPipe loads a named-entity recognizer as an instance of the Chunker interface and then applies it to the remaining command-line arguments, printing out the results. For our task, instead of command line arguments, we pass the sentences in our input document.

For example, if we have a sentence of the form:

p53 regulates human insulin-like growth factor II gene expression through active P4 promoter in rhabdomyosarcoma cells.

Then we get the following output corresponding to it.

Chunking=p53 regulates human insulin-like growth factor II gene expression through active P4 promoter in rhabdomyosarcoma cells. : [0-3:GENE@-Infinity, 20-54:GENE@-Infinity, 81-92:GENE@-Infinity]

This output repeats the input sentence and then, following the colon, the list of chunks found. Here we found a chunk running from character position 0 (inclusive) to character position 3 (exclusive), with type GENE; this is right, because p53 is a gene. It also find a gene from psoition 20 to 54, for the expression insulin-like growth factor II gene, and a final one from 81 to 92 for P4 promoter.

2.2 Model used:

The model used by this chunker is *ne-en-bio-genetag.HmmChunker*. This one is labeled by task (ne for named-entity recognition), language (en for English), genre (bio for biology) and corpus (genetag for the GENETAG corpus), and suffixed with the name of the class of the serialized object (HmmChunker for com.aliasi.chunk.HmmChunker).

2.3 Data Flow in my components:

The major components of the system are as shown in. 3. As shown, the Collection Reader is used to read the input document and store it as a CAS. Analysis Engine then accesses this CAS, and uses LingPipe to perform the annotations. The required details regarding offset etc. are obtained here and stored in the indexes along with the annotations. The CAS consumer then reads the indexes and prints out the contents to an output file.

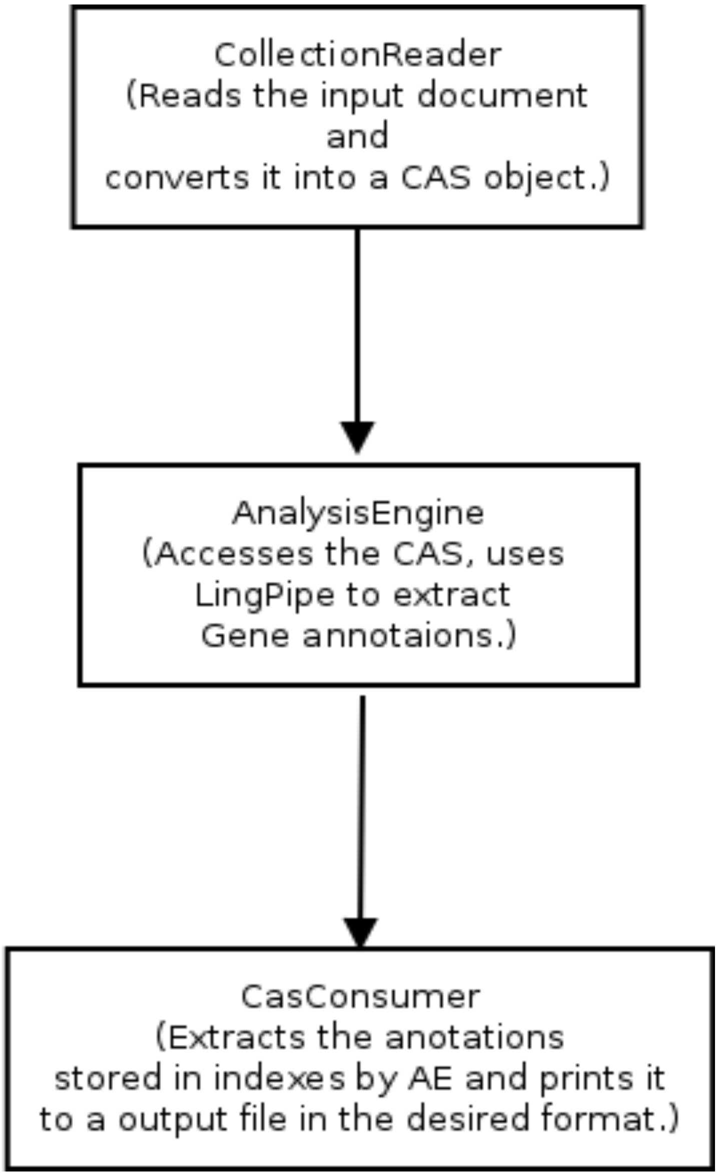

CollectionReader
(Reads the input document
and
converts it into a CAS object.)
AnalysisEngine
(Accesses the CAS, uses
LingPipe to extract
Gene annotaions.)
CasConsumer
(Extracts the anotations
stored in indexes by AE and prints it
to a output file in the desired format.)

Figure 3

2.4 Description of the major components:

The domain model of the system, consisting of major components and the important methods are as shown in Figure 4. The sequence of operations is as in Figure 5.

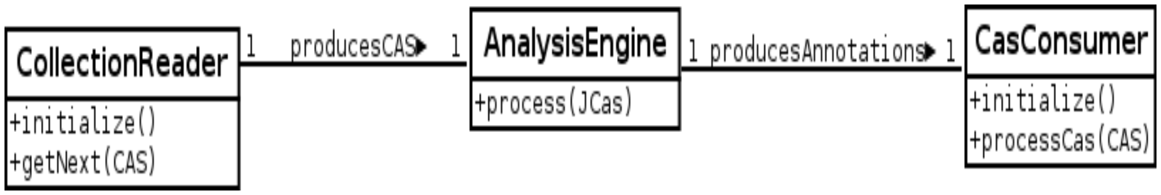


Figure 4

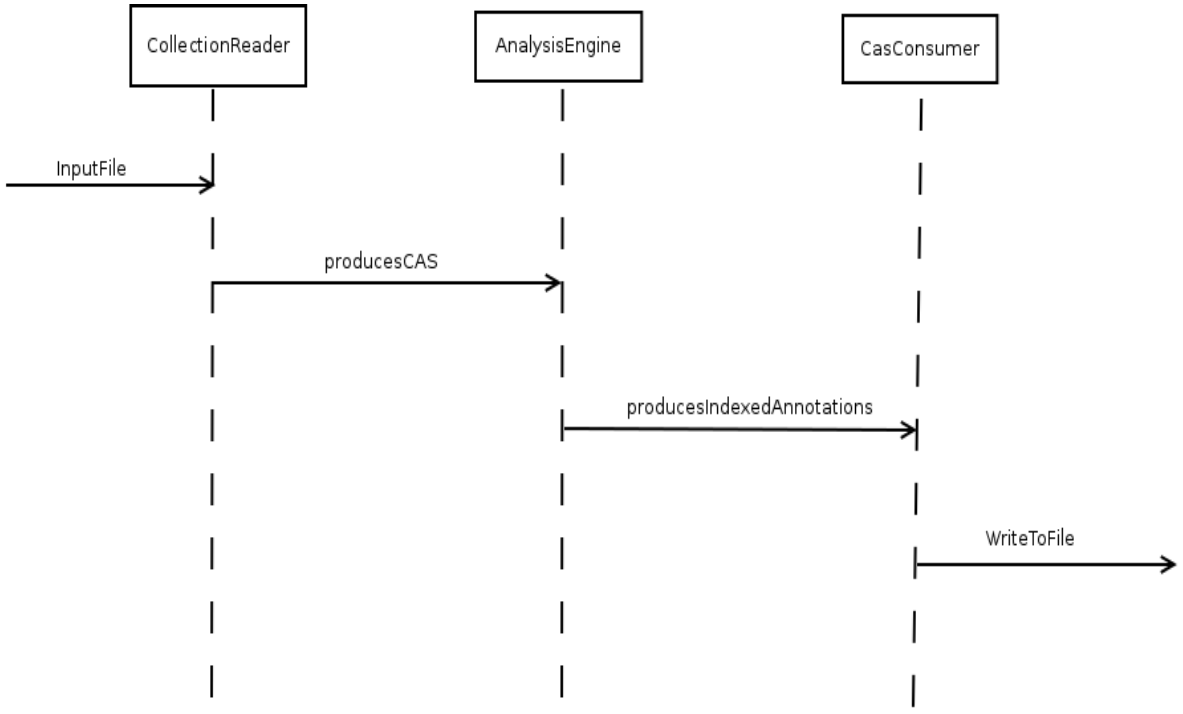


Figure 5

Type System:

I considered a basic type system for this task, containing attributes for sentence ID, the sentence text and the gene tag produced as output.

Collection Reader:

Here, there is just one document to be read, so the input file is specified for the Collection Reader. But when we have a directory of files to be parsed, we can specify the directory and have the reader handle all the files present in it. This then reads the document and stores it as a CAS object.

Analysis Engine (AE):

Here I use a primitive AE that reads the CAS produced by the Collection Reader. The document is then viewed as a collection of lines, and each line is fed into the LingPipe module for annotation. The result obtained is in the form of chunks, as explained earlier. The offsets from the chunks are extracted and some modification is made on them as in the required output format, intermediate spaces are not taken into account.

For example for a sentence, say,
P00001606T0076 Comparison with alkaline phosphatases and 5-nucleotidase
The expected output would be,
P00001606T0076|14 33|alkaline phosphatases
P00001606T0076|37 50|5-nucleotidase

These values are generated for all the lines in the document and stored in indexes.

CAS Consumer:

The CAS consumer is then used to iterate over the annotations stored in the index. They are retrieved and the information is printed out to an output file in the desired format.

CPE Descriptor:

The Collection Processing Architecture defines additional components for reading raw data formats from data collections, preparing the data for processing by Analysis Engines, executing the analysis, extracting analysis results, and deploying the overall flow in a variety of local and distributed configurations. The functionality defined in the Collection Processing Architecture is implemented by a Collection Processing Engine (CPE). A CPE includes an Analysis Engine and adds a Collection Reader, a CAS Initializer (deprecated as of version 2), and CAS Consumers. The part of the UIMA Framework that supports the execution of CPEs is called the Collection Processing Manager, or CPM.

So we can generate a CPE descriptor by including a Collection Reader, Analysis Engine(s) and CAS Consumer(s). Here we have only one AE and one consumer.

3 <u>Method and Results:</u>

Annotations obtained at the end of the pipeline were written onto an output file, which could be used for further analysis. Evaluations can be carried out to assess how well the annotator works. Typically, to gain high precision, we can have different types of annotations made and combine them all to produce a more accurate annotation than would have been possible with just any one of them.

One thing to note is that, in the output generated by my code, is sorted based on the lower bound of the offset. For instance, as given in the homework document, the annotations generated for a line say:

P00001606T0076 Comparison with alkaline phosphatases and 5-nucleotidase are

P00001606T0076|14 33|alkaline phosphatases
P00001606T0076|37 50|5-nucleotidase

In my code, I have used LingPipe. I wrote a wrapper for it that could extract the offsets from its output. I passed each sentence to it, after stripping off the sentence ID. The offsets returned by LingPipe are actually the indexes of the character in the sentence string passed. But since the requirements here are that, we must not take into account the intermediate spaces, I correspondingly modified the offsets to our requirement. And then included the sentence ID, offsets and the annotation text in the annotation. Since I did not carry out the whole pipeline for each sentence, and the file writing happens only once, at the end. This results in the output being ordered in ascending order of the lower bounds (14 in the first, 37 in the second). I think that now that I have an idea that passing each sentence through the pipeline would have given me a file very similar to the gold standard file, I will try implementing the work around in future systems I work on or if there is an opportunity, in the next homework!

To evaluate my system, I wrote a stand-alone java code that compared the output generated by the system with the gold standard output provided. The results are as shown below:

No. Of Annotations in the gold standard file: 18,265
No. Of Annotations in the generated output file: 20,174
Matching Lines: 15504

Precision:
Precision= Number of correct annotations/Total annotations
= 15504/20174
= 0.7685
Therefore, precision= 76.85 %

Recall:
Recall=No. of correct annotations found/Total no. of correct annotations in gold standard
= 15504/18265

=0.8488
Recall = 84.88 %

The precision is a little low because the system only has one annotator for the whole task. If we can include more annotators, then we might be able to prune out the incorrect annotations. Also, we see that recall is quite high, implying that most of the correct annotations were rightly found by the annotator.

4 <u>Conclusion:</u>

The UIMA framework was used to perform the gene mention tagging. The pipeline structure of UIMA gives us a smooth flow of operations with efficient modularity. With the introduction of more sophisticated algorithms or techniques in the annotator, we can expect high throughput results than obtained here. This framework gives us a broad scope of dabbling around with various components on a one-by-one basis, without having to configure the whole system for that each time we wanted a change. Especially in a research environment, I strongly feel that resources like UIMA and CSE are highly significant. UIMA allows teams to match the right skills with the right parts of a solution and helps enable rapid integration across technologies and platforms using a variety of different deployment options. So, if there was a task, that had "n" phases and in case we only wanted to work on a particular problem area, we could just plug in the state-of-art systems for the other "n-1" phases and only concentrate on improving that one phase. This gives us a lot of flexibility and allows us to concentrate on a focused area without having to worry too much about the surrounding processes. This helps in saving time on integrating and thus allowing more time to the particular task itself. This in turn would help us develop standard code, that is interoperable and hence has high reusability.