UNIVERSITA DEGLI STUDI DI TRENTO

Faculty of Science

NATURAL LANGUAGE PROCESSING AND INFORMATION RETRIEVAL

UIMA FOR PYTHON

(Academic year 2015-2016)

Report by: Duc Manh Hoang (MAT. 180387)

Submitted to: Prof. Alessandro Moschitti

Massimo Nicosia

July, 2016

Table of Contents

1. Introduction to the designed system

This project requires to build a client and server application on HTTP protocol. The client will be a program in Python. The server will be in Java. This project require as follows:

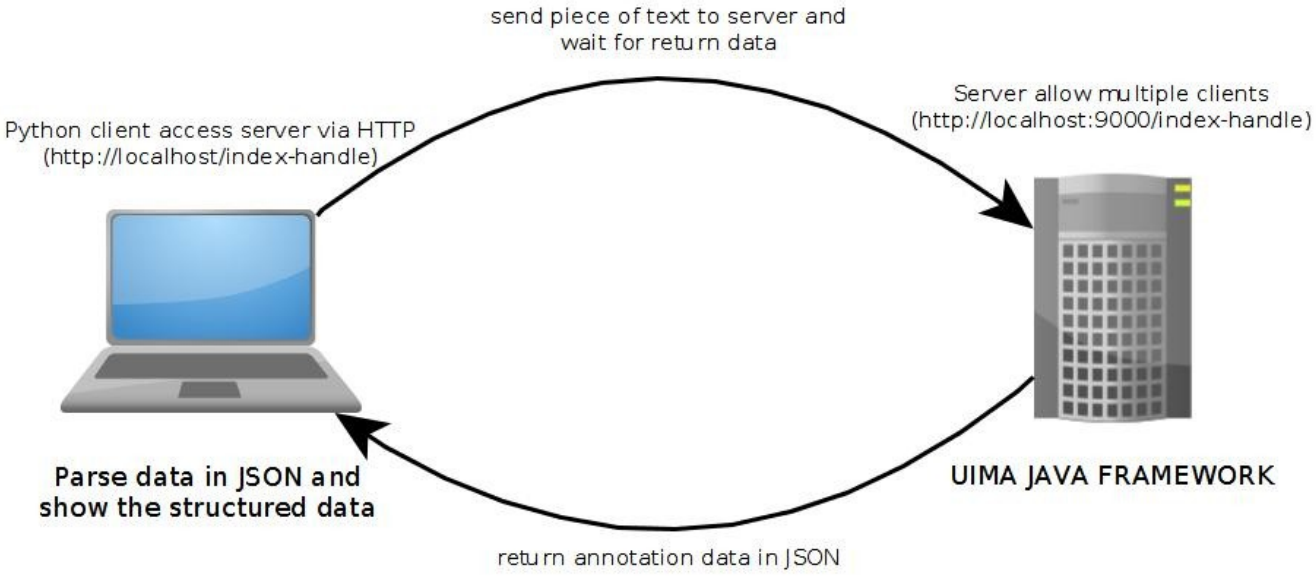


Figure 1: Requirements of projects.

The requirements can be detailed as the following:

- The server accepts HTTP requests from client
- The client sends a piece of text in the HTTP (preferably POST) request
- The server reads the piece of text from the HTTP request and runs on it a pipeline of annotators.
- The CAS should be serialized in JSON format
- The server sends the JSON to the client
- The client parses the JSON in some object that would be convenient to query with methods such as Select or Select Covered.

This project apply to specific case that I wish my server that will receive the piece of text from sending of client. The piece of text is related to a film. It is a description of a film. Inside the text, it can contain some entities such Time, Location, Organization, Person, Money, Percent, Date entities. This project is going to create a server that identify those entities. Then, a collecting functionality will collect them to the CAS object. Finally, the server will use the

library “JsonCasSerializer” to convert the content of the CAS into JSON format and send back to client.

In fact, the movie text can contain a name of a/an Actor, Director, Screenwriter, Cinematographer. Actually, those names of entities is the names to be included in the names of the Person class. Thus, the server also have a functionality that can classify a name of a Person belong to a/an Actor, Director, Screenwriter, Cinematographer.

The server will send a CAS data in JSON format back to the client. Then client will receive the return data that will be parsed. The pared data will be showed in the command line in clean presentation.

2. The technologies

2.1. UIMA

The server will use UIMA Java Framework to create a annotator. In order to proceed with annotator development, we need to configure an IDE to support Apache UIMA. Apache provides supports for Eclipse, to add UIMA nature to projects that are created through Eclipse. After the configuration process is complete. The project are ready to develop a custom annotator. The project will be using Apache UIMA tools to analyze plain text using CAS Visual Debugger.

For the purpose of this project, the server will use the UIMA pipelines to process the unstructured data and churns out structured data. The UIMA perform Name-Entity Recognition (NER). NER is a process of extracting identified instances of classes from content. For example, the task of searching the name of Person in the unstructured text.

The UIMA is normally used to annotate text. It mean that, it creates meta data files that describe where the name of the entity in the text we are looking for was found. To make annotations, we need to create annotators, and defined a UIMA pipeline where our documents will be the input, and out of which we will get the annotations we seek, if they are found.

2.2. HttpServer

This class implements a simple HTTP server. A HttpServer is bound to an IP address and port number and listens for incoming TCP connections from clients on this address. The sub-class HttpsServer implements a server which handles HTTPS requests.

One or more HttpHandler objects must be associated with a server in order to process requests. Each such HttpHandler is registered with a root URI path which represents the location of the application or service on this server. The mapping of a handler to a HttpServer is encapsulated by a HttpContext object. HttpContexts are created by calling createContext(String,HttpHandler). Any request for which no handler can be found is rejected with a 404 response. Management of threads can be done external to this object by providing a Executor object. If none is provided a default implementation is used.

For mapping request URIs to HttpContext paths, when a HTTP request is received, the appropriate HttpContext (and handler) is located by finding the context whose path is the longest matching prefix of the request URI's path. Paths are matched literally, which means that the strings are compared case sensitively, and with no conversion to or from any encoded forms. For example, this project will use a HttpServer with the following HttpContexts configured and request URIs.

Context	Context path	Request URI
Ctx1	/index	http://localhost:9000/index
Ctx2	/index-handle	http://localhost:9000/index-handle

Table 1: The configuration of HttpContexts and the request URIs

2.3. JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is built on two structures:

- A collection of name/value pairs: in various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array. In this project, JSON will use this structure to construct the Time, Location, Organization, Person, Money, Percent, Date, Actor, Screenwriter, Director and Cinematographer entity.

- An ordered list of values: in most languages, this is realized as an *array*, vector, list, or sequence. In this project, JSON will be used list to construct a array of those above entities.

The functionality of the server for creating JSON format will be automatically created by using the utility library “JsonCasSerializer”. The server analyze the movie text to find out the names

of entities, POS, sentence, token and etcetera. They are stored into the CAS object. The "JsonCasSerializer" is a convenient class for CAS object for converting the CAS object in to JSON data. The following figure is a example of the CAS object is converted to JSON format.

Figure 2: The CAS object is converted to JSON format.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures. This convenience will be used as a bridge connect a HTTP server with a Python client.

2.4. DKPro Core - Stanford CoreNLP

Stanford CoreNLP provides a set of nature language analysis tools. It can give the base form of word, their parts of speech, whether they are names of companies, people, etc., normalize dates, times, and numeric quantities, and mark up the structure of sentences in terms of phrases and word dependencies, indicate which noun phrases refer to the same entities, indicate sentiment, extract open-class relations between mentions, etc.

Stanford CoreNLP is an integrated framework. Its goal is to make it very easy to apply a bunch of linguistic analysis tools to a piece of text. A CoreNLP tool pipeline can be run on a piece of plain text with just two lines of code. It is designed to be highly flexible and extensible. With an single option you can change which tools should be enabled and which should be disabled. Stanford CoreNLP integrates many of Stanford's NLP tools, including the part-of-speech (POS) tagger, the named entity recognizer (NER), the parser, the coreference resolution system,

sentiment analysis, bootstrapped pattern learning, and the open information extraction tools.

This project will apply 3 linguistic analysis tools to a piece of movie text. They are StanfordSegmenter, StanfordNamedEntityRecognizer, StanfordPosTagger. The purpose of using this 3 Stanford CoreNLP components is to extract the names of 11 entity types that contain in the text. The entity types are Time, Location, Organization, Person, Money, Percent, Date, Actor, Screenwriter, Director and Cinematographer. Then store all of them into a CAS object.

StanfordSegmenter

The Stanford Segmenter performs tokenizations and makes sentence and token annotations and can serve as a high quality alternative for other segmenter components and other components as StanfordNameEntityRecognizer and StanfordPosTagger.

The input of the server is a piece of movie text. Thus, StanfordSegmenter read that piece of text and output a list of sentences that contain in it. For example a sentence like “The Rookie is a 1990 buddy cop film directed by Clint Eastwood and produced by Howard G. Kazanjian, Steven Siebert and David Valdes.”

StanfordNamedEntityRecognizer

The Stanford NER is a CRFClassifier implementation of a Named Entity Recognizer to label sequences of words in a text which are the names or value of things, such as Time, Location, Organization, Person, Money, Percent and Date. For example, in the following sentence “The Rookie is a 1990 buddy cop film directed by Clint Eastwood and produced by Howard G. Kazanjian, Steven Siebert and David Valdes.” , the StanfordNamedEntityRecognizer can find out the name of Person like “Clint Eastwood”, “Howard G. Kazanjian”, “Steven Siebert”, “David Valdes”

StanfordPosTagger

StanfordPosTagger read text in some language and assigns parts of speech to each word and other token, such as noun, verb, adjective, etc. For example, a sentence after POS tagging look like “The/DT Rookie/NNP is/VBZ a/DT 1990/CD buddy/NN cop/NN film/NN **directed/VBN by/IN Clint/NNP Eastwood/NNP** and/CC produced/VBN by/IN Howard/NNP G./NNP Kazanjian/NNP ,/, Steven/NNP Siebert/NNP and/CC David/NNP Valdes/NNP ./.”

The POS tagging is a very important part in this project, because each sentence that contain the name of Person will be considered to search the name of specific type of movie terms such as name of Actor, Director, Cinematographer and Screenwriter. In order to increase accuracy of the searching functionality, This project apply combining two techniques that are POS tagging and Regular expression matching to search the names of the movie entities.

2.5. Regular Expression

In theoretical computer science and formal language theory, a regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. find and replace “like” operations.

Regular expression are so useful in computing that the various system to specify regular expression have evolved to provide both a basic and extended standard for the grammar and syntax; modern regular expression heavily augment the standard. Regular expression processors are found in several search engines, search and replace dialogs of several word processors and text editors, in the command lines of text processing utilities, and in nature language processing. Many programming language provide regular expression capabilities, some built-in and others via a standard library such as Java, Python and etcetera.

This project is going to work on regular expression in the NER task. Regular expression help to identify the name of the specific type of Person. At the beginning point, the server use the Stanford CoreNLP to identify the names of entities in the piece of movie text. The names of entities are belong to Time, Location, Organization, Person, Money, Percent, Date type. As we known, the movie text can contain many type of Person such as Director, Actor, Screenwriter, Cinematographer. In order to output above movie entities, the names of Person type are use as input of this task to specify which sentences contain these name. Then, each sentence that contain name of the Person should be considered. This task also combine between making use of POS tagging and running the patterns in regular expression in order to increase accuracy of the NER process. Depending on the visualization of these sentences and their POS tagging, the Regular expression will be built.

For example, this is a POS tagging sentence “The/DT Rookie/NNP is/VBZ a/DT 1990/CD buddy/NN cop/NN film/NN **directed/VBN by/IN Clint/NNP Eastwood/NNP** and/CC

produced/VBN by/IN Howard/NNP G./NNP Kazanjian/NNP ,/, Steven/NNP Siebert/NNP and/CC David/NNP Valdes/NNP ./." The Stanford CoreNLP identify for us all person names in this sentence such as "Clint Eastwood", "Howard G. Kazanjian", "Steven Siebert" and "David Valdes". But the name of the director containing in this sentence should be identify and output "Clint Eastwood". By the visualization, the regular expression can be specified as "(directed/VBN by/IN)((([A-z]*)/NNP)+)" that used to match with the sentence above to find out the name of director.

3. System implementation description

3.1. Create a web server

3.1.1. Create a HTTP server

The HttpServer class provides a simple high-level HTTP server API, which can be used to build embedded HTTP servers.

```
public void Start(int port) {
    try {
        this.port = port;
        server = HttpServer.create(new InetSocketAddress(port), 0);
        System.out.println("server started at " + port);
        server.createContext("/index", new Handlers.EchoIndexHandler());
        server.createContext("/index-handle", new Handlers.IndexHandler());
        server.setExecutor(null);
        server.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 3: The code for creating a HTTP server.

The server will listen at port 9000. The contexts of the HTTP server that are described on the "Table 1" will invoke to the handling functionalities showing on the Figure 3.

3.1.2. Create a IndexHandler

IndexHandler are associated with HTTP server in order to process requests. The IndexHandler is registered with a context path "/index-handle" which represents the location of service on this server. Requests for a defined URI path "http://localhost:9000/index-handle" is mapped to the IndexHandler.

```
public static class IndexHandler implements HttpHandler {
    public void handle(HttpExchange he) throws IOException {
        Map<String, Object> parameters = new HashMap<String, Object>();
        InputStreamReader isr = new InputStreamReader(he.getRequestBody(), "utf-8");
        BufferedReader br = new BufferedReader(isr);
        String query = br.readLine();
        new Handlers().parseQuery(query, parameters);
        String content = "";
        for (String key : parameters.keySet()) {
            if (key.equals("content"))
                content = parameters.get(key).toString();
        }
        he.getResponseHeaders().add("Content-type", "text/plain");
        he.sendResponseHeaders(200, 0);
        OutputStream os = he.getResponseBody();
        new Handlers().process(content, os);
    }
}
```

Figure 4: The codes of the IndexHandler class.

The purpose of IndexHandler class is to provide for HTTP server a functionality processing and get the piece of text that is sent from client. Inside of IndexHandler, “handle()” is override itself. The function that get the content of the body page that is filled out by the piece of movie text and submitted to server. The “handle()” function get the content and invoke a parsing function “parseQuery()” to get the piece of text sent by client. Invoking of it pass to parameters. The first one is the content of page in binary string. The second one is a HashMap for storing the key and the text after parsing.

```
@SuppressWarnings("unchecked")
public void parseQuery(String query, Map<String, Object> parameters) throws UnsupportedEncodingException {
    if (query != null) {
        String pairs[] = query.split("[&]");
        for (String pair : pairs) {
            String param[] = pair.split("[=]");
            String key = null;
            String value = null;
            if (param.length > 0) {
                key = URLDecoder.decode(param[0], System.getProperty("file.encoding"));
            }
            if (param.length > 1) {
                value = URLDecoder.decode(param[1], System.getProperty("file.encoding"));
            }
            if (parameters.containsKey(key)) {
                Object obj = parameters.get(key);
                if (obj instanceof List<?>) {
                    List<String> values = (List<String>) obj;
                    values.add(value);
                } else if (obj instanceof String) {
                    List<String> values = new ArrayList<String>();
                    values.add((String) obj);
                    values.add(value);
                    parameters.put(key, values);
                }
            } else {
                parameters.put(key, value);
            }
        }
    }
}
```

Figure 5. The codes of the “parseQuery()” function.

The “parseQuery()” parse the content in binary string and put the key and the content of text into the HashMap parameter.

The “handle()” get the piece of text from HashMap variable and prepare for responding mechanism. The most important responding variable should create to be a OutputStream. The OutputStream write out the content of responding page. Finally, the “process()” function is called to run the Pipeline. The invoking of “process()” pass two parameters. The first parameter is the content of the piece of movie text. The other one is the OutputStream for responding page.

3.1.3. Create a Process function

Process function mainly have a responsible for run the Pipeline of Apache UIMA and get back the structured data containing in CAS object and put that data into the JSON type. The unstructured data is a piece of text that is sent form client side and passed under a variable “content” of String type. The structured data is the CAS object that contain the annotations of Time, Location, Organization, Person, Money, Percent, Date, Actor, Screenwriter, Director and

Cinematographer type.

```
public void process(String content, OutputStream os) {
    try {
        cas.reset();
        cas.setDocumentText(content);
        cas.setDocumentLanguage("en");
        SimplePipeline.runPipeline(cas, seg, pos, ner, movie);
        try {
            jsonCasSerializer.jsonSerialize(cas, os);
            os.close();
        } catch (Exception e) {
            os.close();
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 6: The codes of process() function.

The “process()” function use 5 main annotator variables “cas”, “seg”, “pos”, “ner”, “movie” that are created and initialized in the construction function of Handlers class.

```
private AnalysisEngineDescription seg = null;
private AnalysisEngineDescription pos = null;
private AnalysisEngineDescription movie = null;
private AnalysisEngineDescription ner = null;
private CAS cas = null;
private JsonCasSerializer jsonCasSerializer = null;
public Handlers() {
    try {
        seg = createEngineDescription(StanfordSegmenter.class);
        pos = createEngineDescription(StanfordPosTagger.class);
        movie = createEngineDescription(MovieAnnotator.class);
        ner = createEngineDescription(StanfordNamedEntityRecognizer.class,
                                      StanfordNamedEntityRecognizer.PARAM_VARIANT,
                                      "muc.7class.distsim.crf");
        cas = CasCreationUtils.createCas(TypeSystemDescriptionFactory.createTypeSystemDescription(), null, null);
        jsonCasSerializer = new JsonCasSerializer();
    } catch (Exception e) {
    }
}
```

Figure 7. The codes for constructor of Handlers class.

The constructor of Handlers class is creating the annotator variables “cas”, “seg”, “pos”, “ner”, “movie” in AnalysisEngineDescription type. They are associated with the library classes TypeSystemDescriptionFactory, StanfordSegmenter, StanfordPosTagger, StanfordNameEntityRecognizer, and MovieAnnotator.

The “process()” function use those the annotator variables. “cas” is always reset to clear

previous data. After that, it is added the piece of unstructured text in English.

Now, the important steps are coming. All annotators are put them together. The “runPipeline()” is called and the pipeline process actually run. The pipeline will run the segment annotator, POS tagging annotator, name entity recognition annotator and the defined movie annotator. At the finishing pipeline running, the “cas” contain the data in CAS type. A JsonCasSerializer variable initialized in the constructor of Handlers class will respond the client the structured data in JSON format by converting the CAS object “cas” to JSON and print it on an OutputStream “os”.

3.2. Creating Types System file

This project aim to find out all names of entities containing in the movie descriptions. In addition, the project also classify the names in Person type into 4 specific types as Actor, Cinematographer, Director and Screenwriter. The Person type do not need to be defined because in UIMA library has already provided a Person type. The project's types are defined in something called a type system in UIMA.

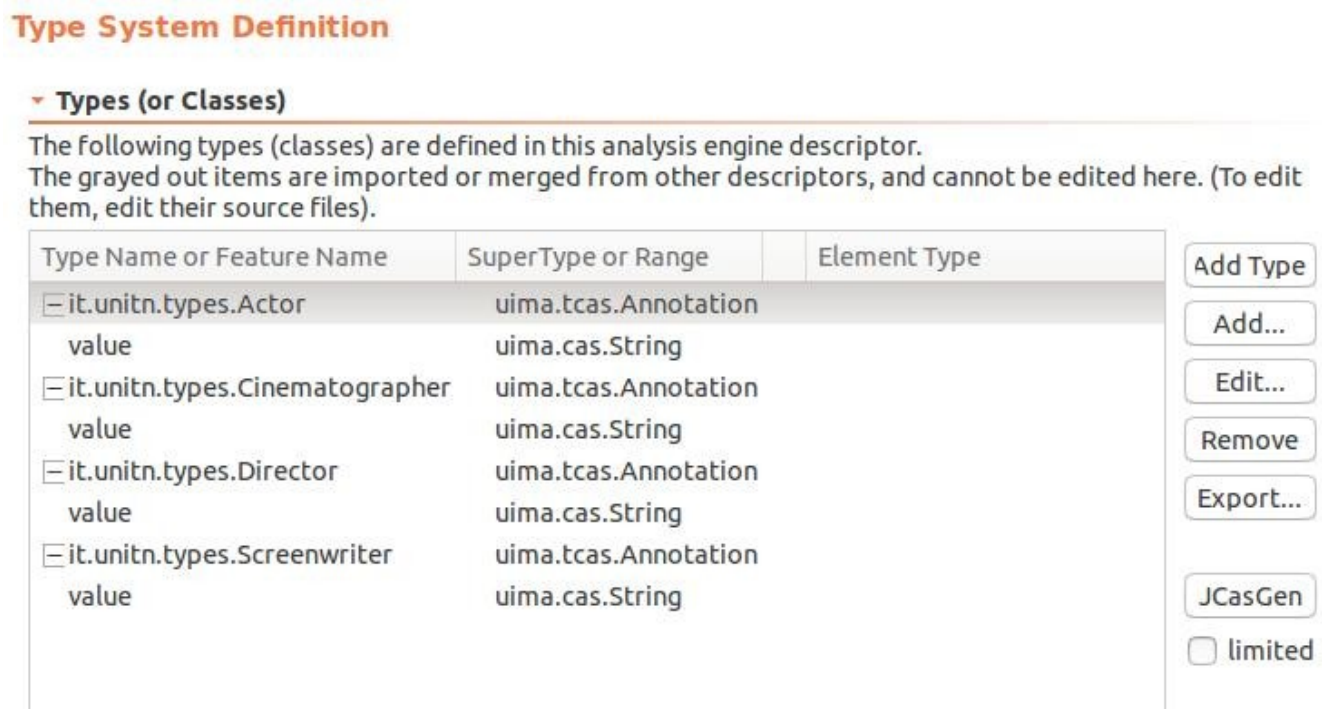


Figure 8: The visualization of the Type system file.

For each type of the entity, the Type system file need to be declared the features of the types. The features are the information that the project want to extract. Here, This project want to

extract the name of the entities in the piece of movie description text. Hence, the Type system file is added the “name” feature with the data type to be “String”. In order to have a convenience in querying data at the client side. The “name” feature is replaced by “value”. Then, Type system file is triggered to generate the object classes correspond to the Type name of the annotations.

3.3. Create MovieAnnotator

There are two part of annotation task in this project. The first part identify the names and values of entities in the piece of text such Time, Date, Location, Organization, Percent, Money and Person. The part two classify the names of Person entities in the text as they are in Actor, Director, Cinematographer, and Screenwriter.

For the part one, due to this project use the UIMA annotation types that are already existed in the UIMA platform. Thus, these annotation types do not need to be created again. This project also use the annotator already exist in UIMA to store the values and names in meta-data format. Our tasks are just get the structured data from the CAS after pipeline finish runing.

For the part two, to perform other document annotations with some thing existed in UIMA, the project need to be created this annotators. An annotator, in UIMA, is a class that performs the logic of annotating document. In this case, the annotator that is defined, named MovieAnnotator, is going to search on movie text to find the names of Actor, Director, Cinematographer, and Screenwriter entities.

The annotator is the component that are usually necessary to extract language units from textual data for sake of search and other applications. The MovieAnnotator is created by creating a class that inherits from org.apache.uima.analysiscomponent.JcasAnnotatorImplBase. This class contain two methods that can override to be “initialize()”, and “process()”.

```
@Override
public void initialize(UimaContext aContext) throws ResourceInitializationException {
    // TODO Auto-generated method stub
    super.initialize(aContext);
    String[] directorPatternStrings = {"(directed/VBN by/IN )((([A-z]*)/NNP )+)",
            "(((directed/JJ by/IN )((([A-z]*)/NNP )+))|(((directed/JJ) ,/, )((((([A-z]*)/JJ ),/, )|(((
    this.directorPatterns = new ArrayList<Pattern>();
    for (String directorPatternString : directorPatternStrings) {
        this.directorPatterns.add(Pattern.compile(directorPatternString));
    }
    String[] actorPatternStrings = {"(stars/VBZ )((([A-z]*)/NNP )+)",
            "(stars/VBZ )(((([A-z]*)/NNP )+,/, )+)(and/CC )((([A-z]*)/NNP )+)",
            "(([A-z]*/NNP )+)(plays/VBZ )((a|an)/DT )(([A-z]*/JJ )|([A-z]*/NN ))",
            "(played/VBN by/IN )(([A-z]*)/NNP )",
            "(stars/VBZ )((([A-z]*)/NNP )+)(and/CC )((([A-z]*)/NNP )+)"};
    this.actorPatterns = new ArrayList<Pattern>();
    for (String actorPatternString : actorPatternStrings) {
        this.actorPatterns.add(Pattern.compile(actorPatternString));
    }
    String[] screenwriterPatternStrings = {"screenwriter[s]?/NN[S]? ([A-z]*)/(?:NN[S]?|JJ) ([A-z]*)/(?
            "(((written/JJ by/IN )((([A-z]*)/NNP )+))|(((written/JJ) ,/, )((((([A-z]*)/JJ ),/, )|((((A
    this.screenwriterPatterns = new ArrayList<Pattern>();
    for (String screenwriterPatternString : screenwriterPatternStrings) {
        this.screenwriterPatterns.add(Pattern.compile(screenwriterPatternString));
    }
    String[] cinematographerPatternStrings = {"cinematographer/NN(?: ,/,)? ([A-z]*)/NN ([A-z]*)/NN",
            "cinematographer/NN(?: ,/,)? ([A-z]*)/NN ([A-z]*)/IN ([A-z]*)/NN"};
    this.cinematographerPatterns = new ArrayList<Pattern>();
    for (String cinematographerPatternString : cinematographerPatternStrings) {
        this.cinematographerPatterns.add(Pattern.compile(cinematographerPatternString));
    }

}
```

Figure 9. The codes of “initialize()” function.

This function initialize the Regular expression patterns that are used in the “process()” function. They match the POS tagging sentence to get the names of the movie entities. There are 4 array of pattern strings “directorPatternString” to detect name of Director, “actorPatternString” to detect name of Actor, “screenwriterPatternString” to detect name of Screenwriter, and “cinematographerPatternString” to detect name of Director of a movie in the piece of movie text. Each time we work on different input data, if we identify other patterns string, we can add them to respective array pattern that they belong to. The main task of this annotator is in “process()” function.

```
@Override
public void process(JCas arg0) throws AnalysisEngineProcessException {
    ArrayList<Integer> eIndex = new ArrayList<Integer>();
    for (Sentence sentence : JCasUtil.select(arg0, Sentence.class)) {
        for (NamedEntity ne : JCasUtil.selectCovered(arg0, NamedEntity.class, sentence)) {
            boolean save = true;
            if (eIndex.size() != 0) {
                for (int index : eIndex) {
                    if (ne.getBegin() == index) {
                        save = false;
                    }
                }
            }
            if (ne.getType().toString().equals("de.tudarmstadt.ukp.dkpro.core.api.ner.type.Person") && save) {
                String namePos = "";
                for (Token token : JCasUtil.selectCovered(arg0, Token.class, ne)) {
                    namePos = token.getCoveredText() + "/" + token.getPos().getPosValue() + " ";
                }
                String posedSentence = "";
                for (POS pos : JCasUtil.selectCovered(arg0, POS.class, sentence)) {
                    posedSentence = posedSentence + pos.getCoveredText() + "/" + pos.getPosValue() + " ";
                }
                //System.out.println(posedSentence);

                if (save(arg0, ne, posedSentence, namePos, this.directorPatterns, "Director")) {
                    eIndex.add(ne.getBegin());
                }
                if (save(arg0, ne, posedSentence, namePos, this.actorPatterns, "Actor")) {
                    eIndex.add(ne.getBegin());
                }
                if (save(arg0, ne, posedSentence, namePos, this.screenwriterPatterns, "Screenwriter")) {
                    eIndex.add(ne.getBegin());
                }
                if (save(arg0, ne, posedSentence, namePos, this.cinematographerPatterns, "Cinematographer")) {
                    eIndex.add(ne.getBegin());
                }
            }

        }

    }
```

Figure 9. The codes of “process()” function in the MovieAnnotator.

The “process()” function with a JCas parameter will classify the names of Person type in UIMA to 4 specific movie type. The codes check each name of value that are detected by Stanford model. If a name is in Person type, it will be consider as a candidate of the classifying processes. The POS tagging of that name and sentence containing it is taken out. Then, the code invoke to classifying processes is called “save()”along with passing 6 parameter as “arg0” in JCas type for storing the classifying data into meta-data, “ne” in NameEntity type for getting the meta-data information from Person annotation, “posedSentence” in String type for searching and matching the patterns on it, “namePos” in String type for checking whether the matching the patterns on the “posedSentence” writing out the same “namePos” or not, “patterns” in List of Pattern for converting to the patterns and providing the patterns to proceed

matching, and “type” in String for classifying currently considering name belong to which type. The “save()” function take these 6 parameters and implement the classifying process.

```
public boolean save(JCas arg0, NamedEntity ne, String posedSentence, String namePos, List<Pattern> patterns, String type) {
    boolean save = false;
    for (Pattern pattern : patterns) {
        Matcher matcher = pattern.matcher(posedSentence);
        String foundS = "";
        while (matcher.find()) {
            StringBuilder foundSB = new StringBuilder();

            for (int i = 1; i <= matcher.groupCount(); i++) {
                foundSB.append(matcher.group(i));
                if (i != matcher.groupCount()) {
                    foundSB.append(" ");
                }
            }

            foundS = foundSB.toString();
        }
        if(foundS.contains(namePos)) {
            save = true;
        }
    }

    if(save) {
        if(type == "Director") {
            //System.out.println(ne.getCoveredText() + "Director" + ne.getBegin());
            Director director = new Director(arg0);
            director.setBegin(ne.getBegin());
            director.setEnd(ne.getEnd());
            director.setValue(ne.getCoveredText());
            director.addToIndexes(arg0);
        } else if(type == "Actor") {
            //System.out.println(ne.getCoveredText() + "Actor" + ne.getBegin());
            Actor actor = new Actor(arg0);
            actor.setBegin(ne.getBegin());
            actor.setEnd(ne.getEnd());
            actor.setValue(ne.getCoveredText());
            actor.addToIndexes(arg0);
        } else if(type == "Screenwriter") {
            //System.out.println(ne.getCoveredText() + "Screenwriter" + ne.getBegin());
            Screenwriter screenwriter = new Screenwriter(arg0);
            screenwriter.setBegin(ne.getBegin());
            screenwriter.setEnd(ne.getEnd());
            screenwriter.setValue(ne.getCoveredText());
            screenwriter.addToIndexes(arg0);
        } else if(type == "Cinematographer") {
            //System.out.println(ne.getCoveredText() + "Cinematographer" + ne.getBegin());
            Cinematographer cinematographer = new Cinematographer(arg0);
            cinematographer.setBegin(ne.getBegin());
            cinematographer.setEnd(ne.getEnd());
            cinematographer.setValue(ne.getCoveredText());
            cinematographer.addToIndexes(arg0);
        }
    }
    return save;
}
```

Figure 10. The codes of “save()” function.

For each pattern, the process match them with the “posedSentence”. If a string is taken out, the codes check the found string whether it contain the “namePos” or not. If it contain that “namePos”, it belong to the “type”. So, the process continue with putting it to the JCas object “arg0” and the “save()” function has accomplished. So far, we was considering server side.

Now, we continue with client side. The client is written in Python.

3.4. Create a client in Python

The client shall send a piece of text to the server by HTTP protocol. The server shall receive the piece of text and analyze it. The result of analyzing of server shall send back to client in server's responding. The data that server send back to client is the CAS object to be converted to JSON format. The client receive the JSON data and store them in running time of program. The client program always wait for user querying. Each querying time, client's user will provide the type of annotation that corresponding to one in the gotten data and execute that query. The respond of program will be a annotation data and its represent. The responding data contain in the JSON data that client received from server, but it is in specific annotation type.

The client is designed to always live and keep to get the control program from client's user. It also provide the instructions of the tasks that user need for manipulating. It should show and present to user the instruction that user should do and the errors that are met. The entering mechanism also provided in this program. The entering mechanism is very important in this program because it take input from user such as the piece of movie text, the type of annotation for querying, and the controlling input for controlling the client program.

```
# This function is in charge of manipulating all process.
# The input is retrived from user input.
# The user's input is checked as the declared option of the input (-i input text, -a anotation type)
# The user's input is the file names to be stored in the array.
# The array of the file names is checked the extension.
# The file names is uploaded.
def main():
   pcas = None
   while True:
      print ('Here is a list of possible choice:')
      print ("\t1: Enter your text and run Pipeline server.")
      print ("\t2: Enter your annotation type to show the data.")
      print ("\t3: Exit.")
      var = input('Please enter your choice: ')

      if (var == "1"):
         content = input('Please enter your text: ')
         print ('Runing pipeline...')
         rdata = query(content)
         pcas = PCas(content, rdata)
         print ('The data is already for querying in step 2!')
      elif (var == "2"):
         if(pcas is None):
            print ('Currently, the program does not have the retreived data. You should complete step 1.')
         else:
            atypes = input('Please enter your annotation type: ')
            atypes = re.sub("[^\w]", " ",  atypes).split()
            data = PCasUtil.select(pcas, atypes)
            while (len(data) == 0):
               print ("Sorry, the annotation does not exist. Please try again!")
               atypes = input('Please enter your annotation type: ')
               atypes = re.sub("[^\w]", " ",  atypes).split()
               data = PCasUtil.select(pcas, atypes)
            print ('{:^30}{:^15}{:^15}{:^15}'.format("VALUE", "BEGIN", "END", "TYPE"))
            for i in range(len(data)):
               values = data[i];
               if (len(values) != 0):
                  for value in values:
                     print ('{:^30}{:^15}{:^15}{:^15}'.format(content[value["begin"]:value["end"]],
                        value["begin"], value["end"], atypes[i].upper()))
      elif (var == "3"):
         print ("you entered", var)
         sys.exit(2)
      else:
         print ('Please enter the right choice!')


main()
```

Figure 11. The codes of the “main()” function.

The first doing, the “main()” function show a list of instructions that can be chosen by user. If user choose for entering the piece of movie text and running the pipeline annotator, an other instruction appear ask user for entering the piece of text. Of course, client's user should choose this step before choosing the others. After inputing, the program invoke to the “query()” function to make the HTTP request to the server. The request contain the piece of user's input text.

```
# This function send the piece of text to server and then get the return data
# The input is the piece of text and the type of annotation that user wish to get
# The output is the return data of server
def query(content):
    c = pycurl.Curl()
    c.setopt(c.URL, 'http://127.0.0.1:9000/index-handle')
    post_data = {'content': content}
    postfields = urlencode(post_data)
    c.setopt(c.POSTFIELDS, postfields)
    buffer = BytesIO()
    c.setopt(c.WRITEDATA, buffer)
    c.perform()
    c.close()
    body = buffer.getvalue().decode('iso-8859-1')
    return body
```

Figure 12. The codes of “query()” function.

This function require the “pycurl” library in Python. It initialize the HTTP method and set some attribute for the server request such as the URL of the server's functionality to accept this client's request, the field that contain the piece of user input text, and the method of the request is POST. Then, this function send to server the piece of text to server and wait for responding of server and return it. The server will respond a piece of data in JSON. The client program store this data into a class named “Pcas” along with the piece of movie text.

```
# This class store the JSON data and the text content of user's input
class PCas(object):
    def __init__(self, content, rdata):
        self.content = content
        self.jdata = json.loads(rdata)

    def get_content(self):
        return self.content

    def get_jdata(self):
        return self.jdata
```

Figure 13. The codes of the “PCas” class.

The “PCas” class includes a constructor function and to getting function, one for getting content of the piece of movie text and one for getting the JSON data. This user task is over and the client program wait for an other user choice.

If user chose the second one, this task require user to enter the types of annotation that user wish to see. The types of annotation should be separated by a space between them. The types of annotation are collected in a string array. Then, These types and a “pcas” variable are passed along with the invoking statement to the “select()” function in the “PCasUtil” class.

```
# This class access the annotation in the CAS
class PCasUtil(object):

   @staticmethod
   def select(pcas, atypes):
      jdata = PCas.get_jdata(pcas)
      content = PCas.get_content(pcas)
      ja_types = jdata['_views']['_InitialView']
      jc_types = []

      for atype in atypes:
         try:
            jc_types.append(ja_types[atype])
         except KeyError:
            jc_types.append([])
            print ('The data does not contain >>', atype)

      return jc_types
```

Figure 14. The codes of “PCasUtil” class.

The idea is to build a interface for the Python classes would have been similar to the UIMA/uimaFit interface. The “PCasUtil” class contain a static method “select()” that takes in input the “pcas” in PCas object and a array of annotations to retrieve . The “select()” method should return a list of Annotation objects containing the information about the annotation they represent . Now, the return data is represented in a visualization. This task should catch some exceptions and some user input error to instruct user's control in right way.

Finally, the client program provide a mechanism to exit the keeping. This client program is a cleaner API. It is also better to take inspiration from what is already there in UIMA/uimaFit.

4. Result presentation

The first task of client's user input the piece of movie text and submit it to the server. The client will present the waiting notice. When client received the responding data, user of client can work on this data by selecting second option. The second option require to enter all types of annotation that user wish. Then, the program will show the result in the table. The following figures illustrate above description.

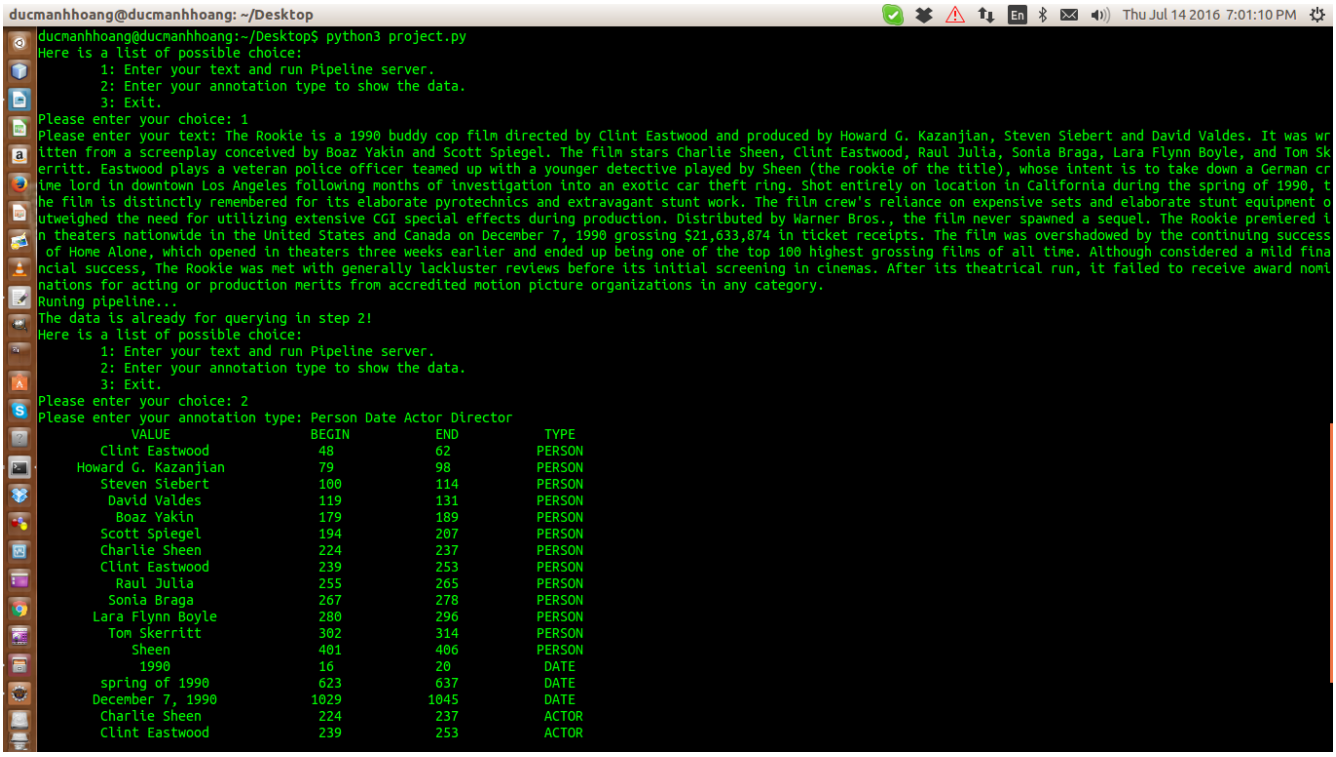


Figure 15. The control panel of the client program.

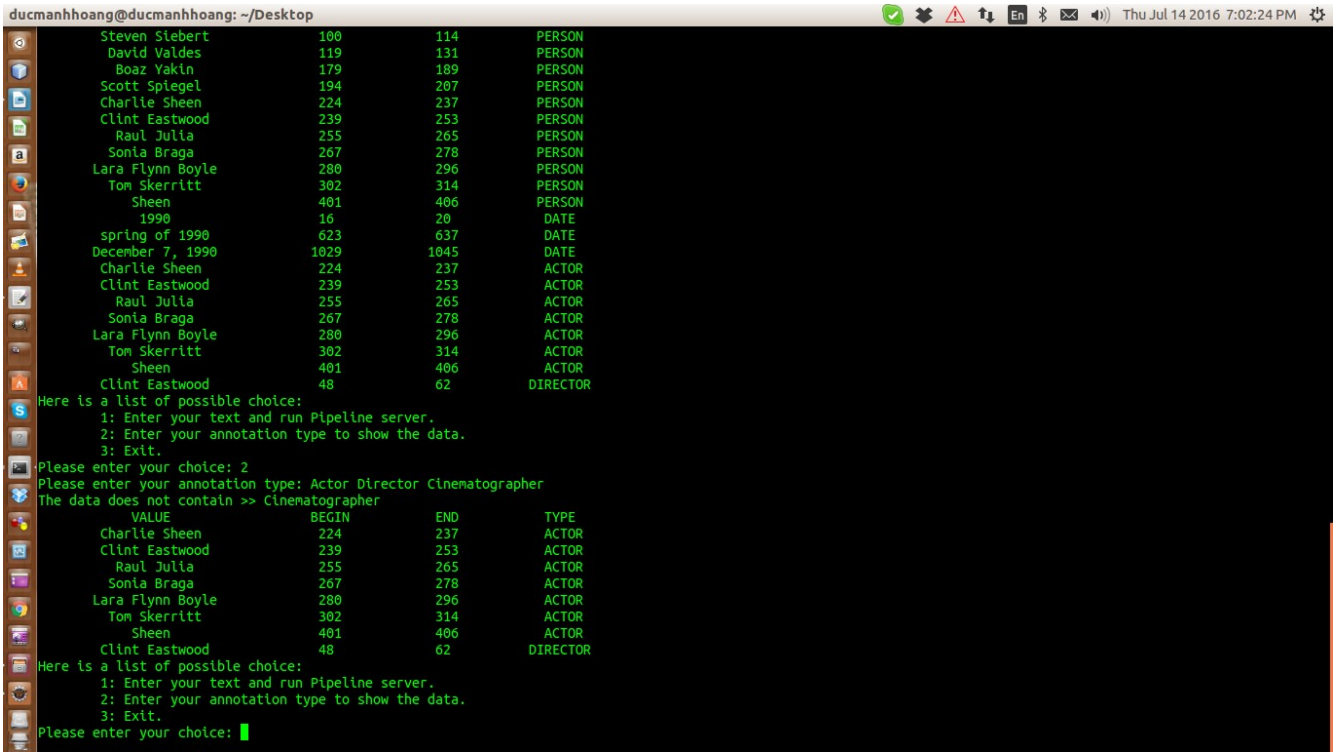
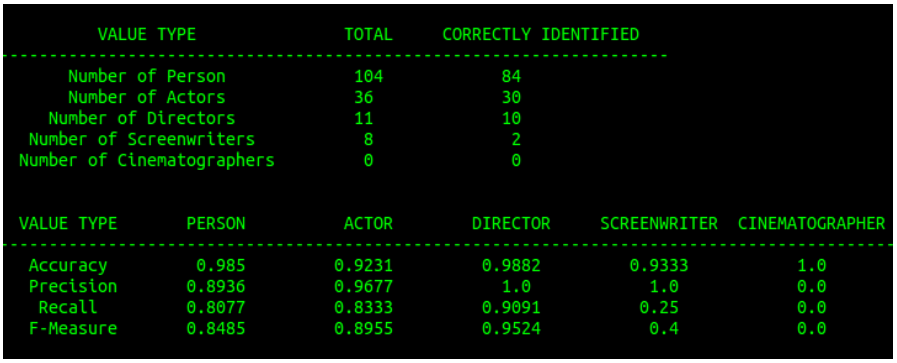


Figure 16. The control panel of the client program.



VALUE TYPE	TOTAL	CORRECTLY IDENTIFIED
Number of Person	104	84
Number of Actors	36	30
Number of Directors	11	10
Number of Screenwriters	8	2
Number of Cinematographers	0	0

VALUE TYPE	PERSON	ACTOR	DIRECTOR	SCREENWRITER	CINEMATOGRAPHER
Accuracy	0.985	0.9231	0.9882	0.9333	1.0
Precision	0.8936	0.9677	1.0	1.0	0.0
Recall	0.8077	0.8333	0.9091	0.25	0.0
F-Measure	0.8485	0.8955	0.9524	0.4	0.0

Figure 17. The measurement that compute on the classes.

5. Conclusion

In this project, I tried to build a HTTP server and perform one of the tasks in NLP: Named-entity recognition, or NER. This project used UIMA, a document processing engine, and Stanford CoreNLP library to identify the basic types of entities such as Time, Date, Location, Organization, Person, Percent, Money. I also to create a new annotator in order to take out the name of film persons in a corpus of movie description. The new annotator come from inspiration of using the Regular expression patterns to match the text. To increase the accuracy of this approach, I use the text that is combined with its POS tagging text.

To be favorable for interchange data between server and client, I was using the JSON format on the annotation data. In the UIMA platform also provide for us a utility library “JsonCasSerializer”. It is very convenient and powerful.

The benefits of using UIMA is that we build complex data processing pipelines that can evolve with time, and as an output we get structured meta-data that is ready to be further processed. We can use the approach I used here to perform other kinds of NLP related tasks on documents, and combine it with other tasks to extract, and analyze rich data sets.

The HTTP server was created to be fast enough as the topic requirement. It also allow multiple client accessing at the same time.

For the client, I tried to build a HTTP client in Python perform sending a piece of text to the server and retrieving the return data. The client also parsed the data from JSON format to the clean data.

The HTTP server and the client connect together by using HTTP protocol. They use the JSON format to interchange the data that achieved from the unstructured text.