An Interface for Rapid Natural Language Processing Development in UIMA

Balaji R. Soundrarajan, Thomas Ginter, Scott L. DuVall
VA Salt Lake City Health Care System and University of Utah
balaji@cs.utah.edu, {thomas.ginter, scott.duvall}@utah.edu

Abstract

This demonstration presents the Annotation Librarian, an application programming interface that supports rapid development of natural language processing (NLP) projects built in Apache Unstructured Information Management Architecture (UIMA). The flexibility of UIMA to support all types of unstructured data – images, audio, and text – increases the complexity of some of the most common NLP development tasks. The Annotation Librarian interface handles these common functions and allows the creation and management of annotations by mirroring Java methods used to manipulate Strings. The familiar syntax and NLP-centric design allows developers to adopt and rapidly develop NLP algorithms in UIMA. The general functionality of the interface is described in relation to the use cases that necessitated its creation.

1 Introduction

In the days when public libraries were the center of information exchange, the job of the librarian was to serve as an interface between the complex library system and the average user. The librarian made it possible for one to access specific sources of information without memorizing the Dewey Decimal System or flipping through the card catalog. Analogous to the great librarians of yesteryear, the Annotation Librarian serves the average Java developer in the creation and management of annotations within natural language processing (NLP) projects built using the open source Apache Unstructured Information Management Architecture (UIMA)[1].

Many NLP tasks are performed in processing steps that build upon one another. Systems designed in this fashion are called *pipelines* because text is processed and then passed from one step to the next like water flowing through a pipe. Each step in the pipeline adds structured data on top of the text called *annotations*. An annotation can be as simple as a classification of a span of text or complex with attributes and mappings to coded values. As pipeline systems have caught on, the ability to standardize functionality in and even across pipelines has emerged. UIMA provides a powerful infrastructure for the storage, transport, and retrieval of document and annotation knowledge accumulated in NLP pipeline systems (Ferrucci 2004). UIMA provides tools that allow testing and visualizing system results, integration with Eclipse[2], and use of standard XML description files for maintainability and interoperability. Because UIMA provides the underlying data model for storing meta-data and annotations with document text and the interface for interacting between processing steps, it has become a popular platform for the development of reusable NLP systems (D'Avolio 2010, Coden 2009, Savova 2008). The most notable example of UIMA capabilities is Watson, the question-answering system that competed and won two Jeopardy! matches against the all-time-winning human champions (Ferrucci 2010).

In addition to its successful implementations in NLP, UIMA supports all types of unstructured information – video, audio, images, etc – and so all UIMA constructs generalize beyond text. While handling multiple data types increases the utility of the framework, developers new to UIMA may feel they need to understand the entire framework before being able to distinguish and focus solely on text. The Annotation Librarian aids both novice and experienced UIMA developers by providing intuitive and NLP-centric functionality.

[1] Apache UIMA is available from http://uima.apache.org/

[2] Eclipse Development Platform is available from http://www.eclipse.org

Proceedings of the ACL-HLT 2011 System Demonstrations, pages 139–144,
Portland, Oregon, USA, 21 June 2011.

2 System Overview

The Annotation Librarian was developed as an interface that synthesizes many of the most frequent annotation management tasks encountered in NLP system development and presents them in a manner easily accessed for those familiar with general Java development methods. It provides convenience methods that mirror Java String manipulation, allowing developers to seamlessly combine document text and annotations with the same commands familiar to anyone who has parsed a String or written a regular expression. Advanced functionality allows developers to examine spatial relationships among annotations and perform annotation pattern matching. In this demonstration, we present the general functionality of the Annotation Librarian in the context of the health care research projects that necessitated the creation of the interface.

The interface does not replace the need for NLP algorithms – developers have a plethora of patterns and decision rules, symbolic grammars, and machine learning techniques to create annotations. The Annotation Toolkit, though, provides a convenient way for developers to use existing annotations in their algorithms. This feeds the pipeline workflow that allows more complex annotations to be built in later processing steps using the annotations created in earlier steps.

The Annotation Librarian was developed and modified in response to four research projects in the health care domain that relied on NLP extraction of concepts from clinical text. The diversity of the different tasks in each of these use cases allowed the interface to include functionality common to different types of NLP system development. Interface functionality will be described as groups of related methods in the context of the four research projects and cover pattern matching, span overlap, relative position, annotation modification, and retrieval. All projects received Institutional Review Board approval for data use and only synthetic documents, not real patient records, are shown in the examples presented in this paper.

3 Pattern Matching

Name entity recognition and semantic classification tasks often require advanced concept identification techniques. Identifying mentions of prescriptions in a document using regular expressions, for example, would require hundreds of thousands of patterns for names of medicines and have to account for misspelling, abbreviations, and acronyms. Regular expressions are commonly used to solve simple NLP tasks, though, and can be utilized as part of a more complex information extraction strategy, such as understanding the context in which a term is used in the text (Garvin 2011, McCrae 2008, Frenz 2007, Chapman 2001). Negex (Chapman 2001) is an algorithm for identifying words before or after a term that suggest, for example, that a particular symptom is not present in a patient: "the patient has **no** fever." Other methods for understanding the context around terms include the use of an inclusion and exclusion list (Akbar 2009), temporal locality search (Grouin 2009), window search (Li 2009), and combinations of the above techniques (Hamon 2009).

The Annotation Librarian allows patterns to be built using existing annotations along with document text. This functionality combines the power of finding concepts that require complex means with the simplicity of regular expressions. The syntax mirrors that of the Java Pattern[3] and Matcher[4] classes, but allows for an extended regular expression grammar to identify Annotations. Pattern matching is accomplished in three phases: the input pattern is compiled, the document and annotations are analyzed for matches, and matches are returned along with span information.

A project identifying positive microbiology cultures will illustrate the use of pattern matching with the Annotation Librarian. Clinicians order microbiology cultures to determine whether a patient has a bacterial infection and which antibiotics would be most effective at treating the infection. Susceptibility is the measure of whether an antibiotic can effectively treat an organism or whether the organism is resistant to it.

A sample of microbiology report text is shown in Figure 1 and visualized annotations for the same sample are shown in Figure 2.

[3] Documented at http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

[4] Documented at http://download.oracle.com/javase/6/docs/api/java/util/regex/Matcher.html

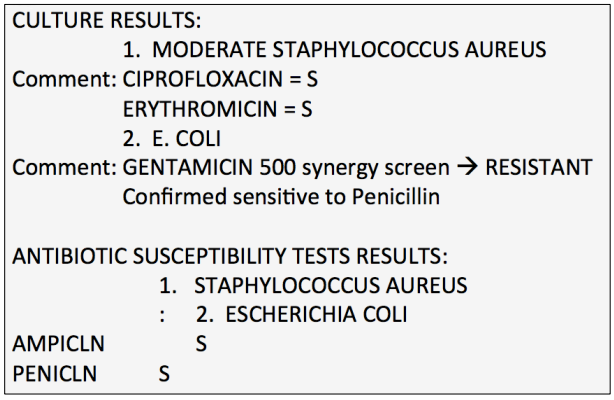

Figure 1: Microbiology Report Text

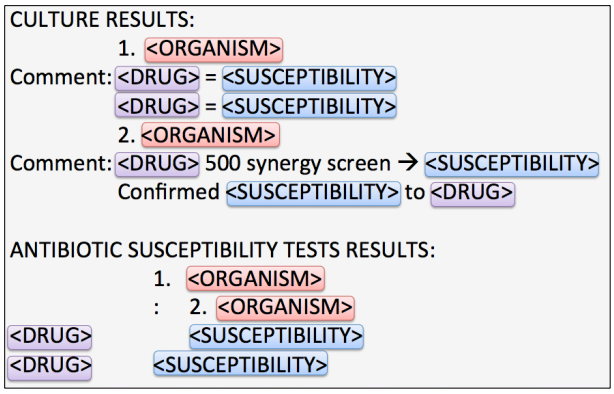

Figure 2: Annotated Report

To demonstrate pattern matching in this sample, the simple pattern of a drug annotation followed by an equals sign and then by a susceptibility annotation will be used.

3.1 Pattern Compilation

The pattern matching process begins when a new instance of an AnnotationPattern is created from the static `compile` method. AnnotationPattern is analogous to the Java Pattern[3] class.

```
AnnotationPattern susceptibilityPattern =
  AnnotationPattern.compile("pattern");
```

The method takes advantage of the UIMA implementation of annotations. Each annotation is an instance of a class that inherits from the UIMA class Annotation[5]. UIMA allows developers to create new types of annotations (in this example Organism, Antibiotic, and Susceptibility) that become Java classes.

[5] Documented at http://uima.apache.org/d/uimaj-2.3.1/api/index.html

The `compile` method input string pattern uses XML tags to represent Annotation classes and tag attributes to denote the name of method calls and return values in the format of:

```
<AnnotationClass methodName="expected value" />
```

When the extra constraint of matching on some method return values is not needed, the tag attribute is left blank. Portions of the pattern that are not contained in XML tags are compiled as Java regular expressions. For our example, the input pattern would be:

```
<Antibiotic /> = <Susceptibility />
```

or further constrained as:

```
<Antibiotic getMedName="ciprofloxacin" /> =
<Susceptibility getValue="S" />
```

which would only match if the particular medication (ciprofloxacin) and susceptibility (S) matched as well.

The pattern is converted into a finite state machine (FSM) in a process described by Fegaras (2005). With our pattern, a four-state FSM would be generated. To arrive in State 1, an Antibiotic annotation must match. To arrive in State 2, a regular expression for "=" must match. The Final State is reached when a matching Susceptibility annotation is found. Any other input would result in a transition back to the Start State.

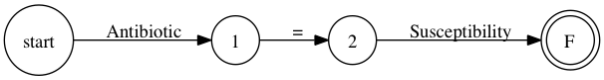

Figure 3: FSM for Antibiotic Susceptibility

3.2 Match Analysis

The second phase of pattern matching processes the document text and annotation set to determine if any matches can be found. This phase is triggered by a call to the static `matcher` method that returns a new instance of an AnnotationMatcher object. AnnotationMatcher is analogous to the Java Matcher[4] class.

```
AnnotationMatcher suscMatcher =
  susceptibilityPattern.matcher(cas);
```

This phase just checks to ensure that each annotation type has at least one instance in the document. Otherwise, a pattern match is not possible. Here, the `cas` parameter refers to the UIMA

Common Analysis Structure, the object containing the document and annotation information.

3.3 Finding Matches

The final phase of pattern matching involves a call to the AnnotationMatcher `find` method. This call results in a FSM traversal at the starting position parameter. Duplicate match candidates starting at the same point are pooled in each state. The candidate pool in each state is traversed with a binary search algorithm, which reduces overall traversal time. Note the following example in which a relationship is created through a new user-defined Annotation class type.

```
int position = 0 ;
while(suscMatcher.find(position))
{
    AntibioticSusceptibility annotation =
      new AntibioticSusceptibility(cas) ;
    annotation.setBegin(suscMatcher.start()) ;
    annotation.setEnd(suscMatcher.end()) ;
    annotation.addToIndexes() ;
    position = matcher.end() ;
}//while
```

Similar to the Java Matcher[4] `find` method, the first match is found from the starting position. The start and end positions are also set within the AnnotationMatcher instance object, which facilitates the creation of new annotations that span the complete pattern. The Annotation Librarian pattern matching functionality allows the inclusion of annotations, which provides an added level of power beyond regular expressions on text data only.

4 Retrieval

The retrieval methods allow developers to interact with annotations and metadata. This set of methods includes the ability to get the file name and path of the document, get all annotations in the document, and get all annotations of just a particular type.

```
getDocumentPath()
getAllAnnotations()
getAllAnnotationsOfType( int type )
```

Ejection fraction is a heart health measurement. An NLP system was developed to identify the ejection fraction from echocardiogram reports. In this project, the Annotation Librarian facilitated the extraction of specific annotation types (the section the concept was found in) in order to discover relevant concept-value pairs.

In Figure 4, ejection fraction annotations are shown in red and quantitative and qualitative values in blue. Because "systolic function" can be used to report ejection fraction, but only when referring to the left side of the heart, it was important to retrieve the section annotations and check the header.

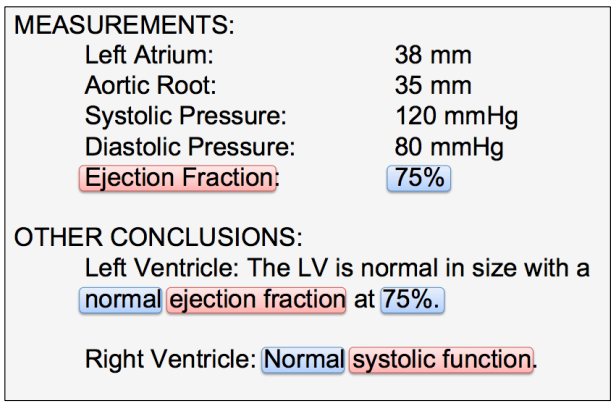

Figure 4: Annotated Echocardiogram Report

5 Annotation Modification

The annotation modification methods allow previous annotations to be altered by trimming whitespace and removing punctuation. While these are trivial tasks performed on Java Strings, an annotation is just a pointer to the text. Updating the annotation with the correct character span requires understanding of UIMA functions and can introduce errors if not done carefully. The Annotation Librarian ensures accuracy by handling these tasks with straightforward programmatic calls.

```
trim( Annotation annotation )
removePunctuation( Annotation annotation )
```

Identifying the organisms from the microbiology reports relied on splitting template text. The project described in Section 3 for pattern matching utilized the Annotation Librarian functionality to clean up spurious characters and whitespace included in annotations.

6 Span Overlap

This set of methods describes how annotations relate to each other spatially by answering questions such as: Does one annotation completely contain the other? Do the annotations overlap in the text? Do they both cover the same span of text?

```
overlaps( Annotation a1, Annotation a2 )
contains( Annotation a1, Annotation a2 )
coversSameSpan( Annotation a1, Annotation a2 )
```

In a system built for identifying medications in discharge summaries, the brand and generic names would often both be listed. Name entity recognition would end up mapping at multiple granularities – brand name only, generic name only, brand and generic name combinations, and even name and dose combinations. The span overlap methods were used to identify and combine overlapping names. Figure 5 shows the annotations that were found and resolved using span overlaps.

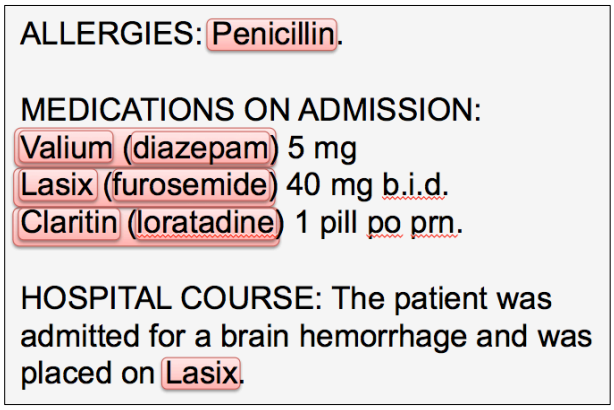

Figure 5: Medication Extraction Use Case

7 Relative Position

The relative position methods allow developers to access annotations based on their position in the text to each other. These methods can determine the next or previous adjacent annotation or the text that exists between two annotations. Often, a task required determining which concepts were found in the same sentence or finding all concepts in a certain section. Methods in this set provide functionality to find annotations that covering the span of another annotations or all annotations contained within the span of another annotation.

```
getContainingAnnotations( Annotation a1 )
getNextClosest( Annotation a1 )
getPreviousClosest( Annotation a1 )
getTextBetween( Annotation a1, Annotation a2 )
```

As part of a project to determine coreference in disease outbreak reports, the ability to determine relative position facilitated coreference resolution. It was also necessary to determine relationships between certain types of annotations from the window of the text. The Annotation Librarian simplified the task of determining co-location by providing the functionality within a single method call. Text between two Annotation objects was similarly identified with a single method call.

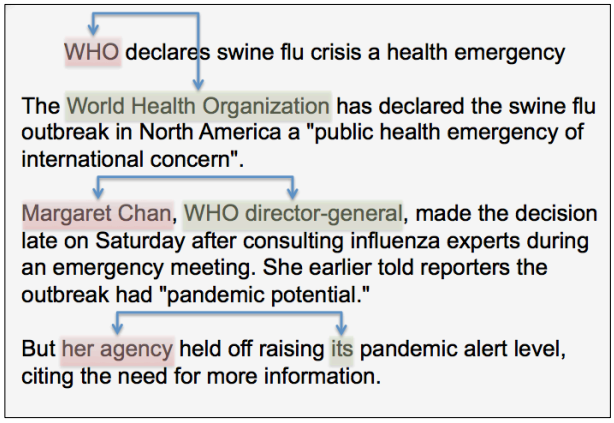

Figure 6: Disease Outbreak Reports Use Case

8 Conclusion

The Annotation Librarian was developed and modified over a number of different NLP use cases. Because of the diversity of tasks in each of these use cases, the toolkit includes functionality common to various types of NLP system development. It includes over two-dozen functions that were used more than one hundred times in each of the four systems listed above. Use of this interface reduced the amount of repeated code; it simplified common tasks, and provided an intuitive interface for NLP-centric annotation management without requiring the presence of an NLP developer who has intimate knowledge of the UIMA data structure. The extended capability provided by the pattern matching methods allows system developers to capitalize on the pipeline approach to NLP development in determining patterns. The ability to use annotations along with text significantly increases the types of patterns that can be identified without complex regular expressions.

9 Future Plans

The Annotation Librarian has been enhanced over the course of a number of biomedical NLP use cases and we plan to continue to enhance the interface as new use cases arise. Some planned enhancements include performance improvements and expanding the AnnotationPattern input pattern syntax to include regular expressions for method return values and annotation class names. We plan to provide additional functionality such as pattern frequency counts.

We see the ability for the Annotation Librarian to help identify patterns through active learning or

unsupervised techniques. In this way, relationships between annotations could be inferred based on those existing in the document set. Such functionality would also provide the ability for more intelligent analysis of future document sets or observation systems by allowing previously identified relationships to be utilized in other use cases.

Acknowledgments

This work was supported using resources and facilities at the VA Salt Lake City Health Care System with funding support from the VA Informatics and Computing Infrastructure (VINCI), VA HSR HIR 08-204 and the Consortium for Healthcare Informatics Research (CHIR), VA HSR HIR 08-374. Views expressed are those of the authors and not necessarily those of the Department of Veterans Affairs.

References

Annin Coden, Guergana K. Savova, Igor L. Sominsky, Michael A. Tanenblatt, James J. Masanz, Karin Schuler, James W. Cooper, Wei Guan, Piet C. de Groen. 2009. Automatically extracting cancer disease characteristics from pathology reports into a Disease Knowledge Representation Model. J Biomed Inform. 2009 Oct;42(5):937-49.

Christopher M. Frenz. 2007. Deafness mutation mining using regular expression based pattern matching. BMC Med Inform Decis Mak. 2007 Oct 25;7:32.

Cyril Grouin, Louise Deléger, and Pierre Zweigenbaum. 2009. COKAINE, A Simple Rule-based Medication Extraction System. i2b2 Workshop in conjunction with the AMIA Annual Symposium, San Francisco, CA; November 13, 2009.

David Ferrucci and Adam Lally. 2004. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. Natural Langage Engineering 10(3–4): 327–348.

David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. 2010. Building Watson: An Overview of the DeepQA Project. AI Magazine. Vol 31. No 3.

Guergana K. Savova, Karin Kipper-Schuler, James D. Buntrock, Christopher G. Chute. 2008. UIMA-based clinical information extraction system. LREC 2008: Towards enhanced interoperability for large HLT systems: UIMA for NLP.

Jennifer H. Garvin, Brett R. South, Dan Bolton, Shuying Shen, Scott L. DuVall, Bruce Bray, Paul Heidenreich, Matthew H. Samore, and Mary K. Goldstein. 2011. Automated Extraction of Ejection Fraction (EF) for Heart Failure (HF) from VA Echocardiogram Reports. Department of Veterans Affairs Health Services Research and Development National Meeting. 2011 Feb 16.

John McCrae, Nigel Collier. 2008. Synonym set extraction from the biomedical literature by lexical pattern discovery. BMC Bioinformatics. 2008 Mar 24;9:159.

Leonard W. D'Avolio, Thien M. Nguyen, Wildon R. Farwell, Yong Chen, Felicia Fitzmeyer, Owen M. Harris, Louis D. Fiore. 2010. Evaluation of a generalizable approach to clinical information retrieval using the automated retrieval console (ARC). J Am Med Inform Assoc. 2010 Jul-Aug;17(4):375-82.

Leonidas Fegaras. 2005. Converting a Regular Expression into a Deterministic Finite Automaton. http://lambda.uta.edu/cse5317/notes/node9.html. Pulled February 2011.

Saiful Akbar, Thomas Brox Røst, Laura Slaughter, and Øystein Nytrø. 2009. Extracting Medication Information from Patient Discharge Summaries. i2b2 Workshop in conjunction with the AMIA Annual Symposium, San Francisco, CA; November 13, 2009.

Thierry Hamon and Natalia Grabar. 2009 . Concurrent linguistic annotations for identifying medication names and the related information in discharge summaries. i2b2 Workshop in conjunction with the AMIA Annual Symposium, San Francisco, CA; November 13, 2009.

Wendy W. Chapman, Will Bridewell, Paul Hanbury, Gregory F. Cooper, and Bruce G. Buchanan. 2001. A Simple Algorithm for Identifying Negated Findings and Diseases in Discharge Summaries. Chapman WW, Bridewell W, Hanbury P, Cooper GF, Buchanan BG. J Biomed Inform. 2001 Oct;34(5):301-10.

Zuofeng Li, Yonggang Cao, Lamont Antieau, Shashank Agarwal, Qing Zhang, and Hong Yu. 2009. Extracting Medication Information from Patient Discharge Summaries. i2b2 Workshop in conjunction with the AMIA Annual Symposium, San Francisco, CA; November 13, 2009.