A Domain-Specific Language for Agile Pattern-Based Natural Language Processing

Will Thompson, PhD[1], Luke Rasmussen[1], Jennifer Pacheco[1], Leigh Baumgart, PhD[2]
[1]Northwestern University, Feinberg School of Medicine, Chicago, IL
[2]NorthShore HealthSystem, Center for Biomedical Informatics Research, Evanston, IL

Abstract

Wide-spread adoption of electronic health records (EHR) has resulted in rapid growth in the number of projects using EHR for research and quality improvement. This has spurred the development of supporting methods and tools for extracting information from EHR, including natural language processing (NLP) techniques for extracting structured information from clinical notes. An important recent contribution has been the creation of the clinical Text and Knowledge Extraction System (cTAKES), a comprehensive clinical NLP system based on modern statistical machine learning methods. However, there are often cases where it is desirable to apply simpler pattern-based methods to an NLP task, such as the annotation of section boundaries in semi-structured documents using cascading sets of regular expressions. We describe a domain-specific language (DSL) that is fully interoperable with cTAKES and which eases the development of pattern-based components by providing expressive functionality for the selection, creation, filtering, and matching of cTAKES annotations.

Introduction

The wide-spread adoption of EHR has resulted in an ever expanding number of secondary use projects for research, quality measurement, and clinical decision support. Many researchers have realized that in order to reach full potential, these projects require NLP of clinical notes, since much of the information in EHR is stored as unstructured text.[1,2] Over the years there have been several efforts at creating comprehensive NLP tools for clinical text, notably including the Medical Language Extraction and Encoding System (MedLEE),[3] MetaMap,[4] and the Health Information Text Extraction (HITEx).[5]

An important recent contribution in this area is the clinical Text and Knowledge Extraction System (cTAKES), originally developed at the Mayo Clinic,[6] and subsequently contributed to the Apache Software Foundation (`http://ctakes.apache.org/`). cTAKES is currently under active development by a distributed community of developers and is rapidly maturing into a comprehensive clinical NLP framework. cTAKES has been used in mulitple large-scale research projects, including SHARPn,[7] eMERGE,[8] and PGRN.[9] Three features jointly distinguish cTAKES from alternatives in this space. First, the software is distributed open source under the business friendly Apache 2 licence. Second, cTAKES uses the Apache Unstructured Information Management Architecture (UIMA), a robust framework for constructing and executing pipelines of components that build successive layers of annotations over a piece of unstructured data.[10] Third, cTAKES has taken a modern approach to NLP, by using statistical machine learning methods to train NLP components where appropriate. Machine learning methods have proven essential to progress in many areas of NLP and are regularly used in systems showing best performance in competitions such as the i2b2 clinical NLP challenges.[11]

Although machine learning methods produce state of the art NLP components, there are also situations that call for simpler pattern-based techniques, either alone or in combination with more sophisticated components. One example is the creation of section boundaries for clinical notes that have a fairly predictable report structure with reliably indicated section headings. In this case, regular expressions plus some supporting code can be sufficient for developing a high performance sectionizer. In other cases, pattern-based methods may be appropriate for annotating clinical elements of interest (such as disease mentions), because the concepts are denoted using fairly constrained natural language that straightforwardly indicates the underlying concepts. Another example where regular expressions are useful is the widely used NegEx algorithm for negation detection,[12] and its extension to detecting the experiencer and temporal status of concepts in the ConText algorithm.[13]

In order to support agile creation of such components within the cTAKES framework, we have developed a domain-specific language (DSL) for defining pattern-based UIMA annotators. A DSL is a programming mini-language that is specialized to a particular application domain, resulting in more intuitive and compact code than is typically possible with general-purpose computer languages. DSLs exist across many application domains and have been in use for

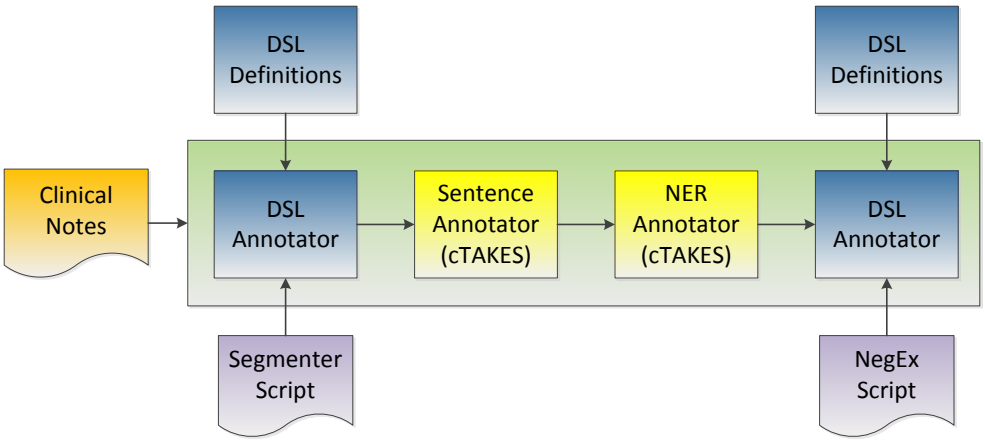


Figure 1: UIMA pipeline combining cTAKES and DSL components

decades.[14] Our DSL for the cTAKES framework was designed to ease the development of pattern-based components by providing flexible and intuitive functions for selecting, creating, filtering, and matching cTAKES types.[15] Pattern-based components are created by writing DSL script files that call a library of functions for supporting these tasks. In the sections that follow, we describe these functions and the general architecture of their implementation. We also demonstrate their use in two specific applications: (1) sectionizing colonoscopy reports, and (2) a compact and intuitive implementation of the NegEx algorithm, with full code provided in the appendices.

Methods

One of our design goals for the DSL was to make annotator creation as simple as possible, but also allow developers the flexibility to easily extend the DSL when desired, or to go beyond it if the situation demands. Therefore, we chose the route of implementing an *internal DSL*,[14] a DSL that is implemented as an API within a general-purpose host language. For the host language, we chose Groovy (`http://groovy.codehaus.org/`). Groovy is a popular dynamic language for the Java platform, which compiles directly to Java Virtual Machine (JVM) bytecode. In the current context, this is useful because cTAKES is implemented in Java, and it makes extensive use of the Java implementation of the UIMA standard (`http://uima.apache.org`). Groovy is fully interoperable with Java – Java methods and classes can be called from within Groovy, and Groovy scripts compile to bytecode class files that can be called from Java. For implementing a DSL, Groovy offers a variety of advantages over pure Java, including much more compact syntax, dynamic typing, closures, named arguments to functions, literals for maps and lists, and first class support for regular expressions.[16] All of these features are used in our DSL implementation and demonstrated in the examples.

In order to load DSL scripts that implement UIMA annotators, we have constructed a `DSL_Annotator` class that extends the UIMA `JCasAnnotator_ImplBase` class. `DSL_Annotator` dynamically loads Groovy-based DSL scripts authored by component developers in order to implement the functionality of the annotator's `process` method. Each instance of `DSL_Annotator` loads the core DSL definitions, which consist of a set of Groovy functions and classes. Figure 1 schematically shows an NLP pipeline that mixes together DSL and cTAKES components. The annotators for sectionizing (also referred to as "segmenting") and for negation detection are based on the DSL, while the annotators for sentence detection and named entity recognition (NER) are taken from cTAKES.

The code for constructing the pipeline in Figure 1 is shown in Listing 4 in the appendix. This Groovy script calls the uimaFIT[17] Java library (`http://uima.apache.org/uimafit.html`) to instantiate and assemble the annotators into a pipeline. The uimaFIT library (increasingly used by cTAKES developers) simplifies common UIMA tasks such as pipeline instantiation, and the ability to use uimaFIT seamlessly in the script demonstrates one of the key advantages of using a JVM-based language. A `DSL_Annotator` instance that loads in a sectionizing script is shown in Listing 5 lines 3–5, while another instance that loads in a NegEx script is shown on lines 6–8.

Results

Here we describe in more detail the core functions of the DSL. There are three categories of functionality: (1) selecting annotations, (2) creating annotations, and (3) pattern-based matching of annotations and text. Example code listings that show the use of these functions are given in this section, while code for two example applications, *sectionizing*, and *negation detection*, is given in the appendices. These applications demonstrate practical use of the DSL and how the DSL functions work together to produce compact and expressive pattern-based annotators. Listing 5 in the appendix shows a simple but working sectionizer, with a sample colonoscopy report in Figure 2 as the target. Listing 6 and Listing 7 show an implementation of the NegEx algorithm, divided into the two sub-tasks of scope determination and negation marking of entity mentions.

Selecting annotations

Selecting annotations is basic yet core functionality that is used frequently in the implementation of UIMA components. The uimaFIT library provides a set of selection methods, with each method implementing different selection criteria. For example, the `selectCovered` method retrieves a list of annotations of a specified type, each of which is "covered" by a specified annotation instance. We make use of these uimaFIT methods in our implementation of the DSL, but make them simpler to use by wrapping them in a single configurable `select` function. This allows us to select collections of annotations using complex combinations of criteria using a minimum amount of syntactic notation.

Listing 1: Selecting annotations

```
1 // Simple example: select all Sentence annotations with an EntitytMention
2 Collection sents = select(type:Sentence, filter:contains(EntityMention))
3
4 // Complex example: nested selects with multiple filtering criteria
5 select(type:Segment).grep {seg -> seg.id == "FINDINGS"}.each { seg ->
6     select(type:Sentence, filter:(
7         and (coveredBy(seg),
8              {sent -> sent.coveredText ==~ /(.+)tubular adenoma(.+)/},
9              not(contains(EntityMention))))).each{sent -> println(sent)}}
```

Listing 1 demonstrates the basic functionality of the `select` function. The function takes two (optional) named arguments: an annotation type, and a filter predicate function (closure), which accepts an annotation of the specified type and returns a boolean value of `true` or `false`. These filter functions are combined using the boolean functions `and`, `or` and `not`, as demonstrated in the `select` statement on lines 6–9. This call to `select` is passed a complex set of filtering criteria, combined together using the `and` function. Besides these boolean functions, other pre-defined DSL predicates usable here are: `contains`, `coveredBy`, `between`, `before`, and `after`. The filter predicate on line 8 of the listing demonstrates the use of a function that is created on the fly (using the Groovy curly brace notation for defining closures). This example shows how easy it is to seamlessly extend the DSL as needed.

Practical uses of the `select` function are demonstrated in the Segmenter script (Listing 5 line 24), and in the NegEx implementation (Listing 6 line 3; Listing 7 lines 5, 21, 29, 34)

Creating annotations

The DSL provides several methods for creating annotations, two of which are shown in Listing 2. The `applyPatterns` function takes three arguments: (1) a collection of annotations, (2) a set of regular expression patterns, and (3) a function that takes an instance of `java.util.regex.Matcher` as its argument. The `applyPatterns` function applies each pattern to the text span covered by each member of the annotation set, and executes the supplied function argument against each resulting `Matcher` instance. Listing 2 line 3 shows a call to `applyPatterns`, with the function argument creating an instance of the cTAKES class `EntityMention` whenever a match occurs. This latter step is implemented using the DSL `create` function, which takes a type argument, and a variable number of optional named arguments. The optional arguments are used to set property values on the created annotation; these properties

can be primitives (such as booleans or strings), or other embedded annotations. On line 6 of Listing 2 a nested call to `create` adds a cTAKES `UMLSConcept` to the `EntityMention` instance.

Listing 2: Creating annotations

```
1 // Annotations are automatically added to CAS object and its indices
2 pattern = ~/(?i)(tubular|serrated)\s+adenoma/
3 applyPatterns(select(type:Sentence), [pattern], {Matcher m ->
4     create(type:EntityMention, begin:m.start(0), end:m.end(0),
5         polarity:1, uncertainty:0, ontologyConcepts:[
6             create(type:UmlsConcept, codingScheme:"CS1", cui:"C01")])})
```

Practical uses of the `create` function are shown in the Segmenter script (Listing 5 line 12) and in the NegEx script (Listing 6 lines 10, 17, 18, 21; Listing 7 lines 9, 13, 17). The `applyPatterns` function is used in the NegEx script (Listing 6 line 9; Listing 7 lines 8, 12, 16).

Matching annotations

The `applyPatterns` function described above combines pattern matching using standard regular expressions together with annotation creation when a pattern matches occurs. Groovy provides first class support for regular expressions, as shown in line 2 of Listing 2. The Groovy ~/<pattern>/ expression creates a compiled Java `java.util.regex.Pattern` object from the given pattern string. This pattern string can use standard regular expression special character classes (such as "\s") without the need to escape the '\' symbol, a problem that plagues regular expression strings in Java. Groovy also supplies two regular expression operators: ~=, which returns a `Matcher` instance, and ~==, which returns a boolean value indicating if an entire string matches a given pattern.

Listing 3: Matching annotations

```
1  // Map of pattern keys associated with string values
2  patternMap = [
3      (~/(FINDINGS.*:)/):"FINDINGS",
4      (~/(IMPRESSION:)/):"IMPRESSION",
5      ...
6      ]
7  // Try each pattern on each SegmentHeading, and assign value on match
8  select(type:SegmentHeading).each{heading ->
9      patternMap.each {pat, val ->
10         if (heading.coveredText ==~ pat) {heading.id = val}
11     }}
12 // Create segments between pairs of segment headings, or between
13 // segment heading and end of text
14 AnnotationMatcher matcher =
15     (~/(?s)(?<S1>@SegmentHeading)(?=(?<S2>@SegmentHeading)|\Z)/).matcher(
16         coveringAnn:jcas.documentAnnotationFs, types:[SegmentHeading],
17         includeText:false)
18 matcher.each{Map b ->
19     create(type:Segment, begin:b.get("S1").begin,
20     end:(b.get("S2") ? b.get("S2").begin : jcas.documentText.length()))}
```

Listing 3 demonstrates two other DSL idioms for regular expression based pattern matching. The first idiom, starting on line 2, uses a Groovy `Map` literal (enclosed in square brackets) to declare a set of key-value pairs, with the key consisting of a regular expression pattern, and the value a string denoting the semantics of the pattern being matched. The `select` function starting on line 8 selects all instances of `SegmentHeading`, and iterates through the resulting collection. The function that is applied to these instances then iterates through the key-value pairs of the map, and checks for a match with the text covered by the heading using the Groovy ~== operator. In this example, when a match occurs the `SectionHeading.id` property is assigned the value from the key-value pair.

The second matching idiom uses the `AnnotationMatcher` class, which is designed to handle pattern matching over annotation graphs rather than simple text strings. Instances of this class are created by calling an extended version of the `Pattern.matcher` method. The arguments to this method are `coveringAnn`, `types`, and `includeText`. The `coveringAnn` argument is used to select the matching span of the annotation graph. The `types` argument is used to select which annotations will be available for matching by the regular expression pattern string. For each annotation type specified in the `types` list, instances of these types within the match span can be referred to within the pattern string using `@<TypeName>` notation, where `TypeName` is the name of the annotation class. The `includeText` argument specifies if text not covered by one of the annotations in the `types` list should also be visible for matching. Applying an `AnnotationMatcher` to an annotation graph generates a set of *bindings*: maps from regular expression group names to the annotations they are associated with. When the matcher engine encounters a type reference associated with a group name (such as the two references to `SegmentHeading` on line 16), the group name and the matched annotation are added to the binding as a key-value pair. Retrieving the matched annotation using the group name is shown in lines 19–20 of the listing.

A more complete version of the pattern map idiom is shown in the Segmenter script (Listing 5 lines 17–28). Practical uses of the `AnnotationMatcher` class are shown in the Segmenter script (Listing 5 lines 9–14) and the NegEx script (Listing 7 lines 15–22).

Discussion

The combination of selection, creation, and matching functions provided by the DSL provide a powerful and intuitive means to create pattern-based annotators that can easily interoperate with cTAKES components. Implementing the DSL as a set of functions and classes inside of the Groovy language provides several advantages, including the ability to easily extend the DSL with new functions, and to seamlessly call out to arbitrary Java code when desired. It is impossible to predict in advance the needs of every user of the DSL, and we did not want any existing gaps in its implementation to be a show-stopper that would prevent its use in a particular project. Another advantage of using Groovy to implement the DSL is support for the Groovy language among all major Java IDEs, including Eclipse (`http://www.eclipse.org/`). This is the development platform of choice for cTAKES, since the Java-based implementation of UIMA comes with tooling and editing support for Eclipse.

Our DSL is similar to several other projects in terms of its goals. The closest is the Apache UIMA Rule-based Text Annotation (RUTA) language (`http://uima.apache.org/ruta.html`). In contrast to our approach, RUTA is implemented as an *external* DSL:[14] an independent mini-language with its own syntax, not hosted within a general-purpose language. A RUTA-based annotator is defined by authoring a RUTA language script, which is loaded and executed by a RUTA analysis engine. The RUTA project also provides Eclipse-based tooling and rule authoring support. Using RUTA it is possible to create highly compact pattern-based annotation engines, and in situations where RUTA provides the necessary functionality it is a powerful tool. However, it is much more difficult to extend RUTA than it is to extend our DSL, and it is not possible to make arbitrary calls to Java APIs from within RUTA. If the needs of an annotator are not completely met by the RUTA language, it becomes necessary to use an alternative approach to implement the desired functionality.

Another project with overlapping goals is the Java Annotations Pattern Engine (JAPE), which is part of the General Architecture for Text Engineering (GATE) framework.[18] JAPE provides a language for expressing pattern matching rules over GATE annotation graphs. It is a mixture of an internal and an external DSL, with the ability to attach Java statements to JAPE rules. JAPE is more mature and feature-rich than the current version of our DSL, and thus has advantages in these respects. However, it is not a seamless experience to combine JAPE with cTAKES, and support for authoring and debugging JAPE rules requires use of the custom-built GATE IDE. Our decision to make our DSL completely internal to Groovy enables us to leverage a richer and more diverse ecosystem of third party libraries and tools than is possible using JAPE.

Moving forward, we plan on releasing our DSL as open source, preferably as a sandbox project associated with the cTAKES platform. We also plan on extending functionality to cover some of the additional useful features currently supported by RUTA and JAPE. Eclipse support for the DSL can also be improved by using Groovy DSL descriptors (`http://groovy.codehaus.org/`), which provide enhanced editing support for developers of DSL scripts.

Acknowledgements

The eMERGE Network was initiated and funded by the National Human Genome Research Institute, with additional funding from the National Institute of General Medical Sciences through the following grant for Northwestern University: U01-HG-004609. Funding for the Northwestern Medicine EDW is provided by Northwestern University Clinical and Translational Sciences Institute (NUCATS) grant UL1RR025741.

References

1 Friedman C, Rindflesch TC, Corn M. Natural language processing: State of the art and prospects for significant progress, a workshop sponsored by the National Library of Medicine. Journal of Biomedical Informatics. 2013 Jun;Available from: http://linkinghub.elsevier.com/retrieve/pii/S1532046413000798.

2 Pai VM, Rodgers M, Conroy R, Luo J, Zhou R, Seto B. Workshop on using natural language processing applications for enhancing clinical decision making: an executive summary. Journal of the American Medical Informatics Association. 2013;Available from: http://jamia.bmj.com/content/early/2013/08/06/amiajnl-2013-001896.abstract.

3 Friedman C. A broad-coverage natural language processing system. Proceedings of the AMIA Symposium. 2000;p. 270–4.

4 Aronson AR, Lang FM. An overview of MetaMap: historical perspective and recent advances. Journal of the American Medical Informatics Association: JAMIA. 2010 Jun;17(3):229–236. PMID: 20442139.

5 Zeng QT, Goryachev S, Weiss S, Sordo M, Murphy SN, Lazarus R. Extracting principal diagnosis, co-morbidity and smoking status for asthma research: evaluation of a natural language processing system. BMC medical informatics and decision making. 2006;6:30. PMID: 16872495.

6 Savova GK, Masanz JJ, Ogren PV, Zheng J, Sohn S, Kipper-Schuler KC, et al. Mayo clinical Text Analysis and Knowledge Extraction System (cTAKES): architecture, component evaluation and applications. Journal of the American Medical Informatics Association: JAMIA. 2010 Oct;17(5):507–513. PMID: 20819853.

7 Rea S, Pathak J, Savova G, Oniki TA, Westberg L, Beebe CE, et al. Building a robust, scalable and standards-driven infrastructure for secondary use of EHR data: the SHARPn project. Journal of biomedical informatics. 2012 Aug;45(4):763–771. PMID: 22326800.

8 Gottesman O, Kuivaniemi H, Tromp G, Faucett WA, Li R, Manolio TA, et al. The Electronic Medical Records and Genomics (eMERGE) Network: past, present, and future. Genetics in medicine: official journal of the American College of Medical Genetics. 2013 Jun;PMID: 23743551.

9 Lin C, Karlson EW, Canhao H, Miller TA, Dligach D, Chen PJ, et al. Automatic prediction of rheumatoid arthritis disease activity from the electronic medical records. PloS one. 2013;8(8):e69932. PMID: 23976944.

10 Ferrucci D, Lally A. UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering. 2004;10(3-4):327348. Available from: http://journals.cambridge.org/production/action/cjoGetFulltext?fulltextid=252254.

11 Uzuner O, South BR, Shen S, DuVall SL. 2010 i2b2/VA challenge on concepts, assertions, and relations in clinical text. Journal of the American Medical Informatics Association. 2011 Jun;18(5):552–556. Available from: http://jamia.bmj.com/cgi/doi/10.1136/amiajnl-2011-000203.

12 Chapman WW, Bridewell W, Hanbury P, Cooper GF, Buchanan BG. A simple algorithm for identifying negated findings and diseases in discharge summaries. Journal of biomedical informatics. 2001 Oct;34(5):301–310. PMID: 12123149.

13 Harkema H, Dowling JN, Thornblade T, Chapman WW. ConText: an algorithm for determining negation, experiencer, and temporal status from clinical reports. Journal of biomedical informatics. 2009 Oct;42(5):839–851. PMID: 19435614.

14 Fowler M. Domain-specific languages. Upper Saddle River, NJ: Addison-Wesley; 2011.

15 Wu ST, Kaggal VC, Dligach D, Masanz JJ, Chen P, Becker L, et al. A common type system for clinical natural language processing. Journal of biomedical semantics. 2013;4(1):1. PMID: 23286462.

16 Dearle F. Groovy for domain-specific languages extend and enhance your Java applications with domain-specific languages in Groovy. Birmingham, UK: Packt Pub.; 2010. Available from: http://site.ebrary.com/id/10441081.

17 Ogren PV, Bethard SJ. Building test suites for UIMA components. In: Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing; 2009. p. 14. Available from: http://dl.acm.org/citation.cfm?id=1621948.

18 Cunningham H, Bonceva K, Maynard D. Text Processing with GATE. Sheffield: University of Sheffield Dept. of Computer Science; 2011.

Appendix: Building an NLP Pipeline with DSL and cTAKES components

Listing 4: Constructing an NLP pipeline

```
// DSL annotator construction (cTAKES annotators not shown)
def tsd = TypeSystemDescriptionFactory.createTypeSystemDescription()
def segmenter = AnalysisEngineFactory.createPrimitiveDescription(
    DSL_Annotator, tsd,
    DSL_Annotator.PARAM_SCRIPT_FILE, "groovy/Segmenter.groovy")
def negex = AnalysisEngineFactory.createPrimitiveDescription(
    DSL_Annotator, tsd,
    DSL_Annotator.PARAM_SCRIPT_FILE, "groovy/NegEx.groovy")

// Pipeline construction
def builder = new AggregateBuilder()
builder.add(segmenter)                // DSL annotator
builder.add(sentenceDetector)         // cTAKES annotator
builder.add(namedEntityRecognizer)    // cTAKES annotator
builder.add(negex)                    // DSL annotator

def engine = builder.createAggregate()
```

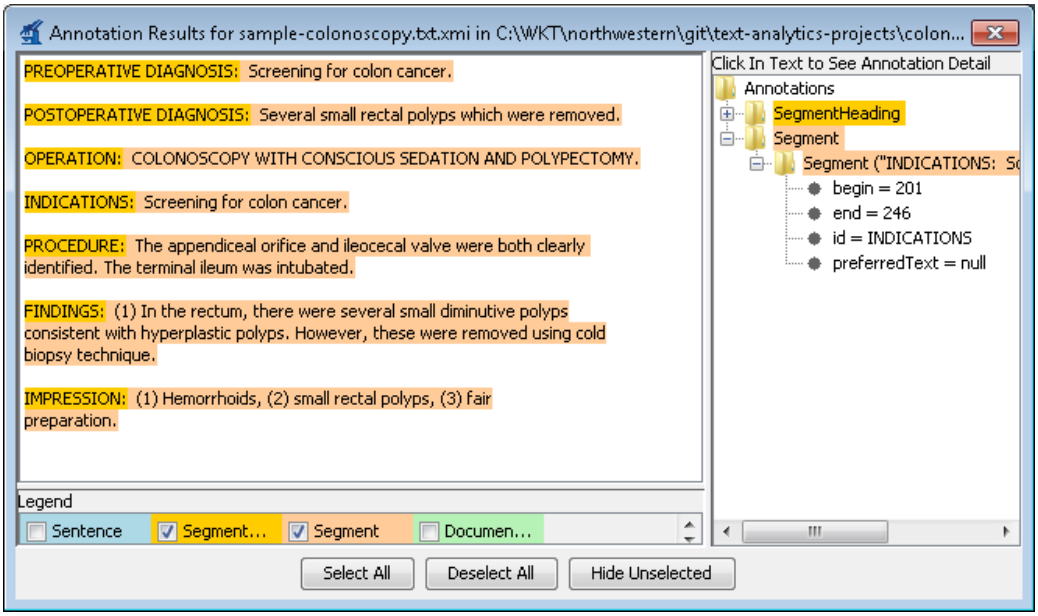
Appendix: Colonoscopy Report Sectionizing

Listing 5: Colonoscopy Report Segment Creation

```
//--------------------------------------------------------------------------
// match segment headings
Matcher matcher = jcas.documentText =~ /(?m)^[A-Z]+(\s+[A-Z]+)*:/
matcher.each {create(type:SegmentHeading, begin:matcher.start(0),
        end:matcher.end(0))}

//--------------------------------------------------------------------------
// create segments between pairs of segment headings
(~/(?s)(?<S1>@SegmentHeading)(?=(?<S2>@SegmentHeading)|\Z)/).matcher(
        coveringAnn:jcas.documentAnnotationFs, types:[SegmentHeading],
        includeText:false).each {Map b ->
                create(type:Segment, begin:b.get("S1").begin,
        end:(b.get("S2") ? b.get("S2").begin : jcas.documentText.length()
        ))}
//--------------------------------------------------------------------------
// categorize segments
patternMap = [
    (~/(PREOPERATIVE\s+DIAGNOSIS):|(INDICATIONS?):/):"INDICATIONS",
    (~/((POSTOPERATIVE|FINAL)\s+DIAGNOS(I|E)S:)/):"DIAGNOSIS",
    (~/PROCEDURE(\s*NOTE)?:/):"PROCEDURE"
    (~/(FINDINGS.*:)/):"FINDINGS",
    (~/(IMPRESSION:)/):"IMPRESSION",
]
select (type:Segment).each {Segment seg ->
    headings = select(type:SegmentHeading, filter:coveredBy(seg))
    patternMap.each {pat, val ->
        if (headings.count{it.coveredText ==~ pat} > 0 ) {seg.id = val}
    }}
```

Figure 2: Colonoscopy report segment annotations after applying Segmenter from Listing 5



Appendix: NegEx Implementation

Listing 6: Negation scope annotation

```
//--------------------------------------------------------------------------------
// scope of algorithm consists of sentences containing an EntityMention
sents = select(type:Sentence, filter:contains(EntityMention))

//--------------------------------------------------------------------------------
// create negation scope terminators
// -- scope terminator patterns abbreviated to fit page
scopeTerm = ~/(?i)(presented)|(presents)|(presenting)/
applyPatterns(sents, [scopeTerm], {Matcher m ->
    create(type:NegationScopeTerminator, begin:m.start(0), end:m.end(0))})

//--------------------------------------------------------------------------------
// create a negation scope between each pair of negation scope terminators
// -- add scope terminators at beginning and end of every sentence
p = ~/(?<N1>@NegationScopeTerminator)(?=(?<N2>@NegationScopeTerminator))/
sents.each {sent ->
    create(type:NegationScopeTerminator, begin:sent.begin, end:sent.begin)
    create(type:NegationScopeTerminator, begin:sent.end, end:sent.end)
    p.matcher(coveringAnn:sent, types:[NegationScopeTerminator],
        includeText:false).each {binding ->
            create(type:NegationScope, begin:binding.get("N1").end,
                end:binding.get("N2").begin)}}
```

Listing 7: Negation annotation

```
1
2  //-------------------------------------------------------------------------------
3  // mark pre-negation, post-negation, and pseudo-negation triggers
4  // -- regex patterns abbreviated to fit page
5  scopes = select(type:NegationScope, filter:contains(EntityMention)
6
7  triggerPreNeg = ~/(?i)(no\s+evidence(\s+of|for)?)/
8  applyPatterns(scopes, [triggerPreNeg],
9      {create(type:PreNegationTrigger, begin:it.start(0), end:it.end(0))})
10
11 triggerPostNeg = ~/(?i)(unlikely)|(was\s+ruled\s+out)/
12 applyPatterns(scopes, [triggerPostNeg],
13     {create(type:PostNegationTrigger, begin:it.start(0), end:it.end(0))})
14
15 triggerPseudoNeg = ~/(?i)(no\s+increase)|(no\s+suspicious\s+change)/
16 applyPatterns(scopes, [triggerPseudoNeg],
17     {create(type:PseudoNegationTrigger, begin:it.start(0), end:it.end(0))
18     })
19 //-------------------------------------------------------------------------------
20 // remove negation triggers contained inside pseudo-negation triggers
21 select(type:PseudoNegationTrigger).each {pseudo ->
22     select(type:NegationTrigger, filter:coveredBy(pseudo)).each {neg ->
23         neg.removeFromIndexes()}}
24
25 //-------------------------------------------------------------------------------
26 // mark selected entities as negated
27 scopes.each{scope ->
28     // handle pre-negation terms
29     select(type:PreNegationTrigger, filter:coveredBy(scope)).each{trig ->
30         select(type:EntityMention,
31         filter:and(coveredBy(scope), after(trig.end))).each{mention ->
32             mention.polarity = -1}}
33     // handle post-negation terms
34     select(type:PostNegationTrigger, filter:coveredBy(scope)).each{trig ->
35         select(type:EntityMention,
36         filter:and(coveredBy(scope), before(trig.begin))).each{mention ->
37             mention.polarity = -1}}
38 }
```