An NLP Curator (or: How I Learned to Stop Worrying and Love NLP Pipelines)

James Clarke, Vivek Srikumar, Mark Sammons, Dan Roth

Department of Computer Science,
University of Illinois, Urbana-Champaign.
james@jamesclarke.net, {vsrikum2,mssammon,danr}@illinois.edu

Abstract

Natural Language Processing continues to grow in popularity in a range of research and commercial applications, yet managing the wide array of potential NLP components remains a difficult problem. This paper describes CURATOR, a NLP management framework designed to address some common problems and inefficiencies associated with building NLP process pipelines; and EDISON, an NLP data structure library in Java that provides streamlined interactions with CURATOR and offers a range of useful supporting functionality.

Keywords: Unstructured Information, NLP, Architecture

1. Motivation

Natural Language Processing technologies are widely used within the research community, and increasingly being adopted by commercial and governmental organizations as well. Tools that provide analysis of text, marking up syntactic and semantic elements (e.g. named entity recognizers, syntactic parsers, semantic role labelers) are widely used – typically many such components in combination – as inputs to a more specialized (and complex) application. However, support for integrating these tools, and managing their outputs, is still patchy: there are a number of frameworks available that provide services related to managing and integrating NLP components, but they come with significant limitations. These limitations may be even more problematic in a research and development environment, where rapid prototyping is critical.

As an example, consider the task of writing a Semantic Role Labeler that requires as its inputs not only the raw text but also its part-of-speech, named entity, and syntactic parse annotation. Multiple NLP components exist for each of these tasks, and it is usually unclear which version of each will work best for a given end task. The SRL designers may want to experiment with different input components – for example, to try different syntactic parsers. But a syntactic parser may itself have input requirements. Worse still, different parsers may be written in different programming languages. Each component may have its own data structures and interface, and so the research team must spend time writing interface code to handle the different components; tools written in different programming languages are run independently and their results written to file to be read by downstream components. And if another team from the same organization wishes to use the outputs of these components, they must deal with the file format of the generated text files – assuming the first team was thoughtful enough to store them somewhere.

A number of researchers and developers have tried to address the integration problem (see section 4.), but these approaches require users to commit to a single NLP preprocessing framework (which generally limits them to tools from a single source) and/or a single programming language, or end up being an all-encompassing system with a steep learning curve. Members of our research group, which actively prototypes many experimental NLP components, wanted a light-weight approach with minimal overhead. Specifically, they were all interested in a programmatic interface that offered access to the NLP components' outputs. Beyond that – for example, regarding learning frameworks – their needs differed: some wished to use an existing Machine Learning framework, while others wanted to implement their own. They also wanted access to sophisticated components with taxing memory requirements.

We identified a set of desiderata for users designing NLP processing pipelines in term of capabilities. Our users wished to:

1. Incorporate 3rd party NLP components written in a wide range of programming languages with only modest effort, rather than restricting the user to a single programming language or even a single source of components.
2. Distribute NLP components across multiple host machines, but have a single point of access that can be shared by multiple users.
3. Minimize the time required to learn a management framework/build and configure an NLP pipeline.
4. Interact with NLP components via a programmatic interface, using appropriate (intuitive) data structures.
5. Handle errors gracefully.
6. Cache the output of NLP components, avoid-

ing redundant processing of input text within and across research teams, and allowing clients of the NLP components benefit from that cache without having to directly use it.

While there are numerous NLP toolkits and frameworks already available, we found none that satisfied all these requirements. We have therefore developed CURATOR, a light-weight NLP component manager; and EDISON, a Java library that simplifies interaction with CURATOR and provides useful additional functionality. The remaining sections of this paper describe these tools, and describe some key differences with other toolkits and frameworks.

2. Curator

CURATOR[1] comprises a central server and a suite of *annotators* (such as named entity taggers, parsers, etc.), each of which conforms to one of a set of specified annotator interfaces. These annotators are registered with CURATOR via its configuration file, which specifies a host, port, and dependency list for each annotator. The workflow of a call by a client to the curator for an annotation type is illustrated in Figure 1. The user sends text to the CURATOR to be annotated with semantic roles (arrow 1). CURATOR checks to see if the requested annotation is in the cache (2) and if so, returns it (7). If not, it will first check that all the dependencies for annotating text with semantic roles are satisfied. Since the semantic role labeler depends on the parser and the part-of-speech tagger, the CURATOR again checks the cache, and if the text has not been annotated with these resources, requests them from the client components and caches them (3, 4, and 6). With all dependencies satisfied, CURATOR now calls the semantic role labeler for its annotation (5). This new annotation is stored in the cache (6) and the requested annotation is returned to the user (7).

The user stands to gain a lot from this arrangement: 1. CURATOR comes with a straightforward interface in several programming languages based on fundamental NLP data structures – trees, lists of spans, etc. 2. There is a single point of contact: the user does not directly interact with the annotation services, and so doesn't have to parse specialized formats or directly deal with multiple different data structures. 3. If someone else has already processed the same text with the required annotation resource, the user gets the cached version with an associated speedup. This also applies if the user, over the course of a project, needs to run a new version of her system over the same corpus. 4. It is straightforward to write a wrapper for most NLP components to make them CURATOR annotation components. 5. The user with access to several machines with modest memory resources rather than a single machine with large memory can distribute the load of the different annotation components across them. 6. CURATOR supports multiple simultaneous requests, so multiple clients can call the same CURATOR instance.

```
<annotator>
  <type>labeler</type>
  <field>ner</field>
  <host>myhost.at.my.place:8823</host>
  <requirement>sentences</requirement>
  <requirement>tokens</requirement>
  <requirement>pos</requirement>
</annotator>
```

Figure 2: A sample entry from a CURATOR annotator configuration file. The host name and port number desired are both given in the 'host' element.

2.1. Server/Client Infrastructure

CURATOR is built on Thrift[2], a library developed to facilitate uniform serialization and efficient client-server communications across a wide variety of programming languages. The desired data structures and server interfaces are specified in a Thrift definition file. Thrift can then be used to automatically generate CURATOR client libraries for these data structures and interfaces in the desired language; these libraries can be used by the developer to write their application[3].

CURATOR's registration method – via configuration file – makes it easy to add new services that use libraries generated from CURATOR's Thrift definition. Furthermore, CURATOR supports multiple instances of individual annotation services, and will distribute incoming requests to a second instance if the first is busy with a previous request. Figure 2 shows an excerpt from a CURATOR annotator configuration file specifying dependencies.

CURATOR is distributed with a suite of NLP tools that the Cognitive Computation Group has found useful (Illinois POS, Chunker, Basic and Extended NER (Ratinov and Roth, 2009), Coreference (Bengtson and Roth, 2008), and SRL (Punyakanok et al., 2008); a version of the Charniak parser (Charniak and Johnson, 2005); a version of the Stanford constituency and dependency parsers (); and the Illinois Wikifier (Ratinov et al., 2011)). Within the group's own CURATOR instance, we have added the easy-first dependency parser (Goldberg and Elhadad, 2010).

2.2. Representation of Text and Component Annotations

CURATOR's data structures encode annotations (e.g. Named Entity tags) in terms of pairs of character off-

[1] CURATOR documentation and code can be found at http://cogcomp.cs.illinois.edu/trac/wiki/Curator

[2] http://thrift.apache.org

[3] Thrift, and hence CURATOR, currently support C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml.

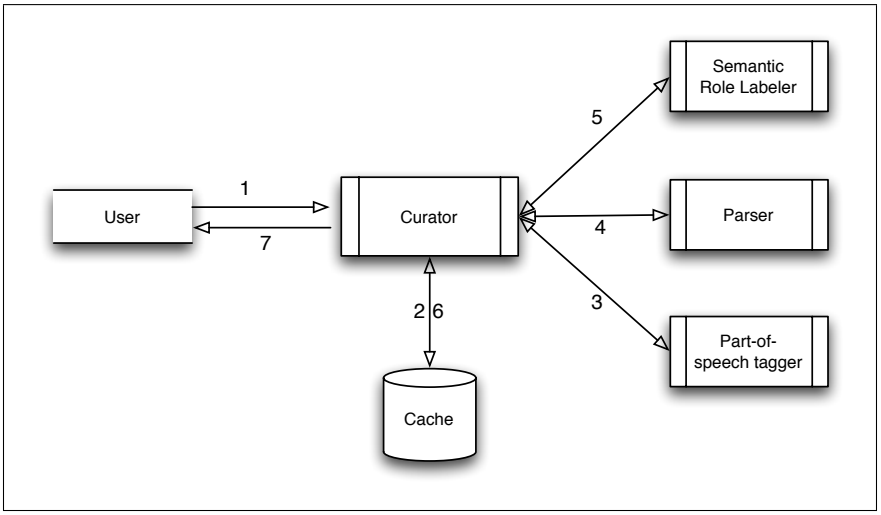

Figure 1: Retrieving semantic roles from the CURATOR. See the text for a detailed description.

sets over the original text input, associated with labels. The design philosophy is that the original text is sacrosanct, and all annotations must map to that text. However, no pair of annotations is required to align with each other – in theory, it is possible for two annotation sources to disagree on token boundaries, for example. This problem is addressed at least in part by EDISON (see Section 3.), but CURATOR's approach is designed to prevent loss of information.

Figure 3 is a schematic showing CURATOR's representation for Tree structures. *Spans* are the most fundamental layer, indicating character offsets in the underlying text and associating with them a label; optionally, a set of attribute-value pairs can be specified for each span. *Nodes* are associated with a Span, a label, and a list of integers corresponding to children (also Nodes). *Trees* are a list of Nodes, and an integer index into this list that indicates the root Node. All these data objects are held in a *Record*, which comprises multiple *Views*, each of which holds the annotations associated with a specific source, and is labeled with that source's identifier to allow retrieval by the client. The Record also holds a copy of the input text. This approach allows for modular extension of the CURATOR system by adding new Views corresponding to new NLP components that are added.

2.3. Using CURATOR services programmatically

CURATOR's services can be accessed using its client classes in the desired language. The invocation sets up a connection with the main server and requests annotation services using the names specified in the CURATOR's annotation configuration file. Figure 4 shows an illustrative example in java; Figure 5 shows an excerpt in php.

The call requires specifying the communication protocol and making/closing a connection; these statements are easily rolled into a method or class for concinnity (see EDISON, section 3. for an example). The examples omit error handling to conserve space, but the Thrift and CURATOR libraries may throw exceptions if there is a problem with the annotation service requested or with the curator itself. The Java example illustrates the use of CURATOR's Record data structure to access elements of a Named Entity annotation.

2.4. Adding a new CURATOR service

CURATOR's Thrift underpinnings allow for automatic generation of libraries for the various data structures and service interfaces specified in the CURATOR project, which simplifies the task of adding a new service. To add a new service to the CURATOR instance, the following steps are required:

1. Create a wrapper ("handler") for the new NLP component. Based on the data structure that the component will return, it will implement one of a small set of pre-defined interfaces: for example, a syntactic parser that returns a single tree for each input sentence will implement the Parser interface. The wrapper will implement a method that takes a CURATOR Record data structure as input, and maps from the relevant View type to the input data structure used by the component. It will also map from the component output structure into the appropriate View type data structure.
2. Create a server to run the handler. This is largely the same for all components, differing mainly in the type and name of handler being run.
3. Add a corresponding annotator entry to the annotator configuration file, naming any dependencies, together with the host and port that the component server will use (see Figure 2).
4. Start the component service, and restart the CURATOR service. This forces the CURATOR service to reread the configuration file and add the service information to its pool.

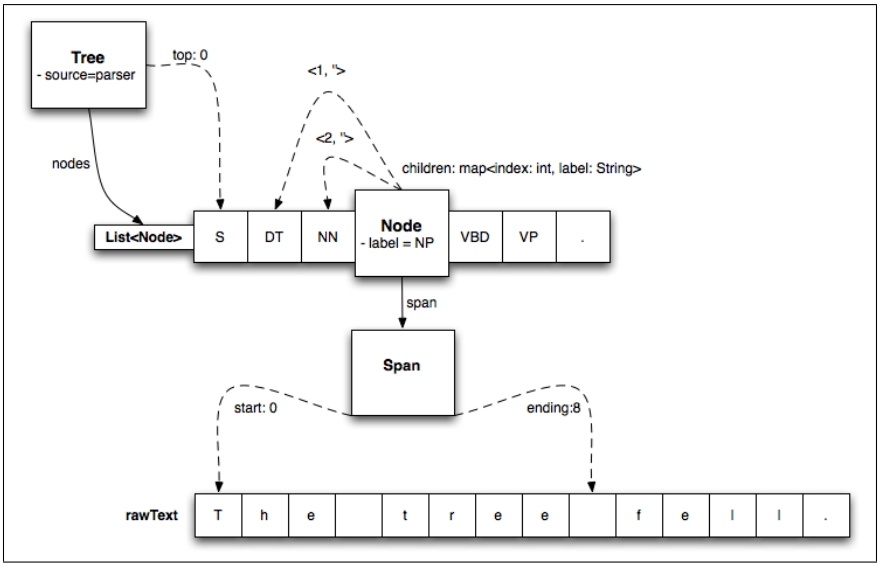

Figure 3: An example of CURATOR data structures: Trees

3. Edison

EDISON is a Java library for representing and manipulating various NLP annotations such as syntactic parses, named entity tags, etc. While the CURATOR's primary goal is to enable multiple NLP resources to be used via a common interface and single point of contact, EDISON's goal is to enable the quick development of NLP applications by integrating multiple views over the text. EDISON provides a uniform representation for diverse views, which lends itself to easy feature extraction from these views.

Structures encountered in NLP are typically graphs, where nodes are labeled spans of tokens. EDISON uses this abstraction, and represents annotations as graphs over *Constituent*s, which are spans of tokens. Labeled directed edges between the constituents represent *Relation*s. Each graph is called a *View*. For any text, EDISON defines a *TextAnnotation* object, which specifies the tokenization of the text and stores a collection of views over it.[4]

Figure 6 shows an example of EDISON's representation. Here, the sentence *"John Smith went to school."* is annotated with three views – named entities, full parse and semantic roles. The nodes (i.e. constituents) assign labels to spans of text. The edges between constituents (i.e. the relations) can have labels, a feature that is needed for representing structures like dependency trees. Note that some views do not have any edges – here, the named entity view is a degenerate graph with a single node, representing the **PER** annotation for the span of tokens *[0,2]*, which corresponds to *"John Smith"*. EDISON provides several specialized views for several types of frequently encountered design patterns: token labels (part of speech, lemma, etc), span labels (shallow parse, named entities, etc), trees (full parse and dependency parse), predicate-argument structures (semantic roles) and coreference. These specializations extend the generic view with specialized accessors that suit the structure.

The main advantage of the uniform representation (i.e., views being graphs over constituents) is that it enables EDISON to provide general-purpose operators to access information from the views while being agnostic to their actual content. This can help us to define a feature representation language over EDISON (such as the Fex language described in (Cumby and Roth, 2000) and related work.)

For example, it is straightforward to extract features such as the part-of-speech tag of a word; all part-of-speech tags of words within a specified span; dependency tree paths from a verb to the nearest preceding noun; dependency tree paths from a verb to the last token of the nearest named entity that precedes it. This last example highlights a key utility of EDISON: the ability to extract features across different levels of analytic markup of a given text span.

To give a sense of how EDISON can help with feature extraction, we provide two examples. These examples highlight how the representation can help in defining complex features and can easily take advantage of the diverse structural annotations that are available. In figure 7, the first snippet extracts the path from a token to the root of the Stanford parse tree, while the second one considers the more complex query of finding SRL predicates whose arguments are named entities.

Creating EDISON objects : One way of creating EDISON objects is to use the CURATOR. It pro-

[4] As noted earlier, CURATOR stores character level offset information for each view to preserve the output of each annotator. While this strategy is useful for retaining the provenance, it can be cumbersome to work with directly. Since EDISON allows the user to provide an arbitrary tokenization, enforcing a token-based representation does not present a serious limitation.

```
public void useCurator()
{
  //First we need a transport
  TTransport transport = new TSocket(host, port);
  //we are going to use a non-blocking server so need framed transport
  transport = new TFramedTransport(transport);
  //Now define a protocol which will use the transport
  TProtocol protocol = new TBinaryProtocol(transport);
  //make the client
  Curator.Client client = new Curator.Client(protocol);

  transport.open();
  Map<String, String> avail = client.describeAnnotations();
  transport.close();

  for (String key : avail.keySet())
    System.out.println(``\t'' + key + `` provided by '' + avail.get(key));

  boolean forceUpdate = true; // force curator to ignore cache

  // get an annotation source named as 'ner' in curator annotator
  //     configuration file
  transport.open();
  record = client.provide( ``ner'', text, forceUpdate);
  transport.close();

  for (Span span : record.getLabelViews().get(``ner'').getLabels()) {
    System.out.println(span.getLabel() + `` : ''
    + record.getRawText().substring(span.getStart(), span.getEnding()));
  }
  ...
}
```

Figure 4: A snippet of Java code using CURATOR services

vides an easy to use Java interface for the CURATOR, which aligns the character-level views to EDISON's own token-oriented data structures. The listing in Figure 8 shows a snippet of Java code which highlight this usage. In addition to the CURATOR interface, EDISON also provides readers for several standard text-based dataset formats like the Penn Treebank and the CoNLL column format.

4. Related Work

A number of NLP frameworks that can combine NLP component exist; we identify some key limitations of the most well-known frameworks that led to the development of CURATOR and EDISON.

4.1. UIMA

UIMA (Götz and Suhre, 2004) is an annotator management system designed to support coordination of annotation tools, satisfying their dependencies and generating a unified stand-off markup. It supports distributed annotation components. A UIMA system could be implemented to wrap NLP components and to cache their outputs. However, it would be a more complex undertaking to use UIMA to make these components and the cache available inline to a client application at runtime. UIMA's main limitation, from our perspective, is its limited support of languages other than Java. A significant C++ component is in development, but support for other languages (Perl, Python, and Tcl) is indirect (via SWIG). UIMA has a very abstract, fairly large API, and therefore has a significant learning curve. Another possible limitation is the efficiency of UIMA's serialization. Thrift, which underpins CURATOR, supports a very wide set of programming languages, and is known to be very efficient in terms of serialization.

4.2. GATE

GATE (Cunningham et al., 2002) is an extensive framework supporting annotation of text by humans and by NLP components, linking the annotations, and applying machine learning algorithms to features extracted from these representations. GATE has some capacity for wrapping UIMA components, so should be able to manage distributed NLP components, though with the caveats above. GATE is written in Java, and directly supports other languages only through the JNI. GATE is a very large and complex system, with a correspondingly steep learning curve.

```
function useCurator() {

// set variables naming curator host and port, timeout, and text
    ...
    $socket = new TSocket($hostname, $c_port);
    $socket->setRecvTimeout($timeout*1000);
    $transport = new TBufferedTransport($socket, 1024, 1024);
    $transport = new TFramedTransport($transport);
    $protocol = new TBinaryProtocol($transport);
    $client = new CuratorClient($protocol);

    $transport->open();
    $record = $client->getRecord($text);
    $transport->close();

    foreach ($annotations as $annotation) {
      $transport->open();
      $record = $client->provide($annotation, $text, $update);
      $transport->close();
    }

    foreach ($record->labelViews as $view_name => $labeling) {
      $source = $labeling->source;
      $labels = $labeling->labels;

      $result = '''';
      foreach ($labels as $i => $span) {
        $result.= ''$span->label;'';
        ...
      }
      ...
    }
    ...
}
```

Figure 5: A snippet of PHP code using CURATOR services

4.3. Other NLP Frameworks

LingPipe[5] is a Java-only NLP development framework, which incorporates support for applying Machine Learning algorithms. NLTK (Loper and Bird, 2002) is a comparable Python-only NLP framework. The Stanford NLP pipeline[6] is monolithic (must run on a single machine) and Java-only.

5. Conclusions

CURATOR and EDISON arose from a constellation of needs that were not satisfied by existing NLP frameworks. The frameworks to which we have compared CURATOR are good for their intended purpose, and many are well-supported by a significant programming community. However, all have limitations we considered problematic for our purposes. CURATOR is not intended to be a replacement for the full frameworks we have named; it is simply a framework that more directly supports management of diverse NLP annotation components and the caching of their outputs, and which foregrounds a modular, multi-view representation. We believe that others may find it useful too.

The combination of the CURATOR and EDISON make it straightforward to harness a diverse range of NLP analytics in a clean, intuitive way. They have been used by several projects within the Cognitive Computation Group – the group's state-of-the-art Semantic Role Labeler (Srikumar and Roth, 2011) and Wikifier (Ratinov et al., 2011), both large, complex systems, strongly depend on CURATOR/EDISON underpinnings and are themselves being released with CURATOR interfaces, so that they can be used in other projects in a similar way. CURATOR and EDISON have also been used to build many prototype systems, greatly reducing the development effort typically involved in building an NLP system.

6. Acknowledgements

We thank our reviewers for their helpful comments. This research is supported by the Defense Advanced Research Projects Agency (DARPA) Machine Read-

[5] http://alias-i.com/lingpipe/

[6] http://nlp.stanford.edu/software/corenlp.shtml

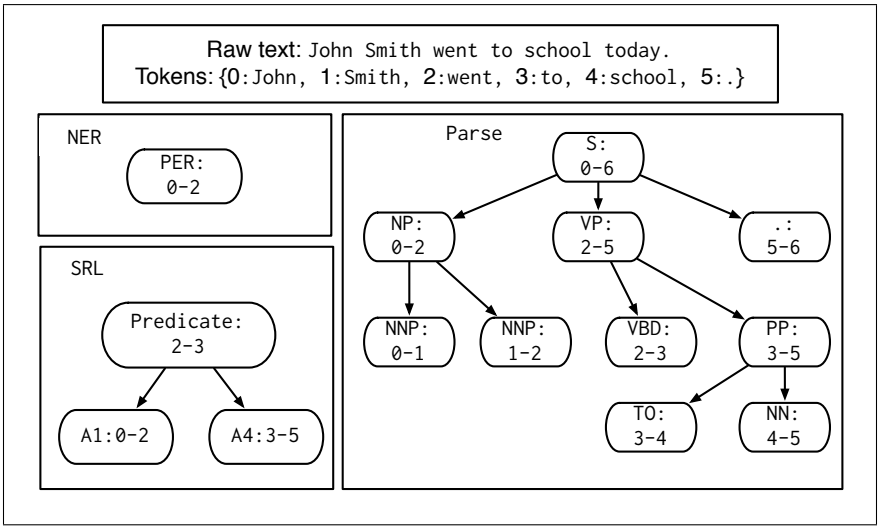

Figure 6: An example of EDISON's representation. The sentence *John Smith went to school.* is annotated with the named entity, parse and semantic role views here.

```
...
CuratorClient client;
String h = // curator host
int p = // curator port
client = new CuratorClient(h, p);

// Should the curator call the
// annotatars if an entry is found
// in the cache?
boolean force = false;

TextAnnotation ta;
ta = client.getTextAnnotation(corpus,
  textId,text, forceUpdate);

client.addNamedEntityView(ta, force)
client.addSRLView(ta, force);

client.addStanfordParse(ta, force);
```

Figure 8: Creating EDISON objects using the CURATOR

ing Program under Air Force Research Laboratory (AFRL) prime contract no. FA8750-09-C-0181. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of the DARPA, AFRL, or the US government.

7. References

E. Bengtson and D. Roth. 2008. Understanding the value of features for coreference resolution. In *EMNLP*, 10.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *In Proceedings of the Annual Meeting of the Association of Computational Linguistics (ACL)*, pages 173–180, Ann Arbor, Michigan. ACL.

C. Cumby and D. Roth. 2000. Relational representations that facilitate learning. In *Proc. of the International Conference on the Principles of Knowledge Representation and Reasoning*, pages 425–434.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *ACL*.

Y. Goldberg and M. Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *NAACL*.

T. Götz and O. Suhre. 2004. Design and Implementation of the UIMA Common Analysis System. *IBM Systems Journal*.

E. Loper and S. Bird. 2002. NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*.

V. Punyakanok, D. Roth, and W. Yih. 2008. The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2).

L. Ratinov and D. Roth. 2009. Design challenges and misconceptions in named entity recognition. In *Proc. of the Annual Conference on Computational Natural Language Learning (CoNLL)*, 6.

L. Ratinov, D. Downey, M. Anderson, and D. Roth. 2011. Local and global algorithms for disambiguation to wikipedia. In *Proc. of the Annual Meeting of the Association of Computational Linguistics (ACL)*.

V. Srikumar and D. Roth. 2011. A joint model for extended semantic role labeling. In *EMNLP*, Edinburgh, Scotland.

```
// The input is a TextAnnotation 'ta' and an integer 'tokenId'

// First, we get the parse tree generated by the Stanford
// parser. TreeView is a specialized View for storing trees. Here,
// we do not use any of the tree-specific functionality and
// instead traverse the graph.
TreeView parse = (TreeView) ta.getView(ViewNames.PARSE_STANFORD);

// Get all constituents whose span includes this token.
List<Constituent> tokenConstituents = parse
    .getConstituentsCoveringToken(tokenId);

// Find the leaf in this list
Constituent node = null;
for (Constituent c : tokenConstituents)
    if (c.getOutgoingRelations().size() == 0)
        node = c;

// Now, build the path by going up the edges
List<String> path = new ArrayList<String>();
do {
    // A Relation is a directed edge that has source and a target
    List<Relation> incomingRelations = node.getIncomingRelations();

    // There can be at most one incoming edge.
    node = incomingRelations.get(0).getSource();

    path.add(node.getLabel());

} while (node.getIncomingRelations().size() > 0);
```

```
// First, SRL and named entity views. PredicateArgumentView and
// SpanLabelView are specializations of View suited for these.
PredicateArgumentView srl = (PredicateArgumentView) ta
    .getView(ViewNames.SRL);

SpanLabelView ne = (SpanLabelView) ta.getView(ViewNames.NER);

List<Constituent> list = new ArrayList<Constituent>();

// PredicateArgumentView allows us to iterate over predicates and get
// its arguments
for (Constituent predicate : srl.getPredicates()) {
    for (Relation r : srl.getArguments(predicate)) {

        // Get the constituent corresponding to the argument edge
        Constituent argumentConstituent = r.getTarget();

        // For any view, we can ask for its nodes which contain a span.
        if (ne.getConstituentsCovering(argumentConstituent).size() > 0) {
            // If there is any such node in the named entity view, we
            // have found a predicate that satisfies the query.
            list.add(predicate);
            break;
        }
    }
}
```

Figure 7: Two snippets of EDISON code. The top one gets the path from a token to the root of the parse tree. If executed on the `TextAnnotation` shown in Figure 6, the variable `path` will be the list `[VBD, VP, S]` at the end of the final loop. The second snippet identifies SRL predicates which have an argument that contains a named entity.