COGCOMPNLP: Your Swiss Army Knife for NLP

Daniel Khashabi\(^1\), Mark Sammons\(^1\), Ben Zhou\(^2\), Tom Redman\(^2\)
Christos Christodoulopoulos\(^3\), Vivek Srikumar\(^4\), Nicholas Rizzolo\(^5\), Lev Ratinov
Guanheng Luo\(^6\), Quang Do\(^6\), Chen-Tse Tsai\(^7\), Subhro Roy\(^8\), Stephen Mayhew\(^1\)
Zhili Feng\(^8\), John Wieting\(^10\), Xiaodong Yu\(^2\), Yangqiu Song\(^11\), Shashank Gupta\(^2\)
Shyam Upadhyay\(^1\), Naveen Arivazhagan\(^5\), Qiang Ning\(^2\), Shaoshi Ling\(^2\), Dan Roth\(^1,2\)

\(^1\)Univ. of Pennsylvania \(^2\)Univ. of Illinois at Urbana-Champaign \(^3\)Amazon Research Cambridge \(^4\)Univ. of Utah \(^5\)Google
\(^6\)Anduin Transactions \(^7\)Bloomberg L.P. \(^8\)MIT CSAIL \(^9\)Univ. of Wisconsin-Madison \(^10\)Carnegie Mellon University \(^11\)HKUST

Abstract

Implementing a Natural Language Processing (NLP) system requires considerable engineering effort: creating data-structures to represent language constructs; reading corpora annotations into these data-structures; applying off-the-shelf NLP tools to augment the text representation; extracting features and training machine learning components; conducting experiments and computing performance statistics; and creating the end-user application that integrates the implemented components. While there are several widely used NLP libraries, each provides only partial coverage of these various tasks. We present our library COGCOMPNLP, which simplifies the process of design and development of NLP applications by providing modules to address different challenges: we provide a corpus-reader module that supports popular corpora in the NLP community, a module for various low-level data-structures and operations (such as search over text), a module for feature extraction, and an extensive suite of annotation modules for a wide range of semantic and syntactic tasks. These annotation modules are all integrated in a single system, PIPELINE, which allows users to easily use the annotators with simple direct calls using any JVM-based language, or over a network. The sister project COGCOMPNLFP enables users to access the annotators with a Python interface. We give a detailed account of our system’s structure and usage, and where possible, compare it with other established NLP frameworks. We report on the performance, including time and memory statistics, of each component on a selection of well-established datasets. Our system is publicly available for research use and external contributions, at: http://github.com/CogComp/cogcomp-nlp.

1. Motivation

Natural Language Processing (NLP) is one of the fastest-growing fields both in research and industrial work. Tools that provide analysis of text by identifying 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, the process of managing and aggregating these tools and their inputs and outputs is typically labor-intensive and error-prone, requiring significant engineering effort before the target application can be built and evaluated. With the introduction of new, complex tasks such as event detection or cross-document coreference, and the increasing commercial demand for text analysis, it is critical to build software frameworks that give easy access to a wide range of existing NLP annotators and support straightforward extension to others.

We introduce COGCOMPNLP, an NLP ecosystem published under an academic-use license, which is designed for management, aggregation, and application of NLP analytic components. It comes with a suite of NLP components for syntactic and shallow semantic analysis, but also word and phrase similarity metrics. It provides essential support for text processing applications, including classes for text cleaning and for reading a number of popular NLP corpora. In addition to describing some key functionalities, we illustrate the use of COGCOMPNLP components in developing a Semantic Role Labeling application: reading data from a corpus; augmenting the resulting data-structures with NLP components using the NLP pipeline; extracting features for input to machine learning algorithms; training classifiers using LBJAVA – another CogComp project (Rizzolo and Roth, 2010); serializing the system outputs; and adding the new SRL application to the pipeline for other applications to use.

2. Terminology

The COGCOMPNLP framework builds on the conceptual design and data-structures described in (Clarke et al., 2012; Sammons et al., 2016). Here we give a brief summary of the main structures and keywords used. A View is a data-structure which contains an annotation structure of a text; examples are tokens, lemmas or dependency parse trees. An Annotator is a class which produces a View given a text, and potentially some other Views. The main data-structure used is TextAnnotation, which contains a document (e.g. a phrase, a sentence, a paragraph) and its various Views.

3. Framework Design

A high-level view of the system is depicted in Figure 1. The boxes show modules and edges show the dependencies between them (with the targets being the dependencies). We describe each of these modules in the following sections.

**Core Utilities.** Contains the fundamental data-structures and operators; hence many of the other modules depend on it. A selection of important basic functionalities supported by this module are:

- SQL-like operations on TextAnnotation for extracting patterns
- Experiment utilities, such as P/R/F1 reporting, statistical significance testing and cross-validation helpers
- String pattern-matching algorithms
- Utilities for reading and writing files, resources and annotations.
The key data-structures for interaction between components – TextAnnotation, View, and Constituent – are illustrated in Figure 2. The TextAnnotation contains the raw source text together with its tokenization and other annotation layers added either by reading corpora annotations or applying NLP components. Token indexes are used to align constituents from different views, but each constituent tracks its character offsets in the raw text. The figure shows Part-of-Speech, Named Entity, Quantity, and Semantic Role Label annotations, each in a separate View.

**Corpus Utilities.** While often overlooked and purely an engineering effort, implementing code that reads data correctly and efficiently can be a time-consuming task. COGCOMPNLP’s corpus reader module includes NLP corpus readers that populate TextAnnotation objects. A few of the important datasets supported by this module are:

- Propbank Semantic Role Labeling
- Treebank Shallow Parse
- PennTreebank Constituency Parse
- Nombank Semantic Role Labeling
- Ontonotes 5.0 POS, Named Entity, Syntactic Parse, and Semantic Role Labeling
- TAC/ERE Event, Relation, and Named Entity

**Similarity Utilities.** For calculating semantic similarity between words, phrases, and entities using both structured and distributional representations. Each similarity function compares objects (words, phrases, named entities, sentences) and returns a score indicating how similar they are. Depending on the inputs, different algorithms are available:

- **Word Similarity:** For computing the similarity between two words. The following representations are currently supported: \( \text{word2vec} \) (Mikolov et al., 2013), \( \text{paragram} \) (Wieting et al., 2015), \( \text{esa} \) (Gabrilovich and Markovitch, 2007), \( \text{glove} \) (Pennington et al., 2014), \( \text{wordnet} \) (Do et al., 2009), \( \text{phrase2vec} \) (Yin and Schütze, 2014). Here is a sample usage:

```java
String s1 = "Jack bought Alex's car";
String s2 = "Alex sold his car to Jack.";
llm.compare(s1, s2); // 0.75
```

- **Named-Entity Similarity:** Comparing named entities requires a different class of algorithm. COGCOMPNLP’s current algorithm is based on (Do et al., 2009):

```java
NESim nesim = new NESim();
nesim.compare("Donald Trump", "Trump"); // 0.9
```

- **Phrasal Similarity:** Algorithms to combine lexical-level systems to make sentence-level decisions (Do et al., 2009):

```java
Metric llm = new LLMStringSim(config);
String s1 = "Jack bought Alex's car";
String s2 = "Alex sold his car to Jack.";
llm.compare(s1, s2); // 0.75
```

**Edison.** Feature extraction is a crucial element in design of NLP systems. **EDISON** (Sammons et al., 2016) is a feature extraction framework that uses the data-structures of COGCOMPNLP core-utilities to extract features to be used by machine learning algorithms. EDISON enables users to define feature extraction functions that take as input the Views and Constituents created by COGCOMPNLP’s Annotators. This makes it possible to not only develop feature sets like words, \( n \)-grams, and paths in parse trees, which work with a single View, but also more complex features that combine information from several Views. This library has been successfully used to facilitate the feature extraction for several higher level NLP applications like Semantic Role Labeling (Punyakanok et al., 2005), Co-reference resolution (Rizzolo and Roth, 2016) and, Textual Entailment (Sammons et al., 2010), which use information across several Views over text to make a decision.
Annotators. Given a set of corpus readers that load input data into unified data-structures, we build NLP annotators which share many features from EDISON and train with LBJAVA (Rizzolo and Roth, 2010) or Illinois-SL (Chang et al., 2015). This means that they are very easy to retrain or adapt to new domains and new languages. COGCOMP-NLP includes a wide range of annotators and supporting functionality. Many of these annotators are state-of-the-art and are widely usable across different tasks.

The complete list of syntactic and semantic annotations we currently support is provided in Table 1, which also shows NLP components of other NLP frameworks. To the best of our knowledge this is the only NLP framework with such a large variety of syntactic and semantic components (in comparison, Stanford’s CORENLP offers 14 components, OPENNLP 8, SPACY 9).

All of the the modules in our systems are actively updated to improve their quality, while maintaining their interoperability. These systems typically began as individual projects in previous work and have evolved over the years. In some cases the performance of components has been reduced for engineering purposes (e.g. faster speed, smaller memory footprint, etc). In Table 2 we present a qualitative assessment of the major components in COGCOMP-NLP. The components have state-of-the-art quality or very close to the best existing results.

Pipeline. With all the Annotators generating the same data-structures, the PIPELINE project provides a simple interface to access Annotator components either individually or as a group, with a single function call. Use of PIPELINE is illustrated in Figure 3. A demo of PIPELINE is accessible online at http://nlp.cogcomp.org.

Accessibility from other programming languages. The majority of the implementation is in Java and hence COGCOMP-NLP is easily accessible to JVM-based languages with direct calls.

To go beyond memory limitations of making direct calls to PIPELINE and make it available to other programming languages, the PIPELINE can be made accessible over a network. Using an internal web-server, PIPELINE can be made available over a network, making it accessible for a variety of programming languages. Essentially a user can instantiate an instance of PIPELINE server on a single machine (with sufficient memory), which can be shared by many users and queried from different languages.

We have created a Python interface, COGCOMP-NLPY\footnote{https://github.com/CogComp/cogcomp-nlpy} which works with direct calls (using PyJnius and Cython (Behnel et al., 2011)) as well as over-network calls to the Java back-end. Here is an example snippet showing how to annotate a sentence with COGCOMP-NLPY:

```python
from ccg_nlpy import remote_pipeline
pipeline = remote_pipeline.RemotePipeline()
text = "Hello, how are you. I am doing fine"
ta = pipeline.doc(text)
print(ta.get_pos)
# (UH Hello) (, ,) (WRB how) (VBP are) (PRP you)...
```

4. Related Work

One important aspect of our work is the collection of the major NLP annotators. Table 1 contains a summary of components that exist in other well-established NLP libraries. CORENLP (Manning et al., 2014) is a popular
<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Measure</th>
<th>Setting</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokenization</td>
<td>MASC (Ide et al., 2010)</td>
<td>Accuracy</td>
<td>–</td>
<td>97</td>
</tr>
<tr>
<td>POS (Roth and Zelenko, 1998)</td>
<td>Penn Treebank (Bies et al., 2015)</td>
<td>F1</td>
<td>–</td>
<td>96.13</td>
</tr>
<tr>
<td>NER (Ratinov and Roth, 2009; Redman et al., 2016; Tsai and Roth, 2016)</td>
<td>CoNLL (T. Kim Sang and De Meuler, 2003)</td>
<td>F1</td>
<td>–</td>
<td>91.12</td>
</tr>
<tr>
<td></td>
<td>Ontonotes (Hovy et al., 2006)</td>
<td>F1</td>
<td>–</td>
<td>84.61</td>
</tr>
<tr>
<td></td>
<td>MUC-7 (Chinchor, 1998)</td>
<td>F1</td>
<td>–</td>
<td>88.37</td>
</tr>
<tr>
<td></td>
<td>Enron (Shetty and Adibi, 2004)</td>
<td>F1</td>
<td>English</td>
<td>88.3</td>
</tr>
<tr>
<td></td>
<td>TAC-KBP 2016 EDL shared task</td>
<td>F1</td>
<td>Spanish</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Chinese</td>
<td>79.3</td>
</tr>
<tr>
<td>Temporal Normalization (Zhao et al., 2012)</td>
<td>TempEval3 (UzZaman et al., 2013)</td>
<td>Exact match F1 / Relaxed match F1</td>
<td>Temporal Span Extraction</td>
<td>79.35/83.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Temporal normalization, given a predicted temporal span</td>
<td>70.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Head detection</td>
<td>89.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Boundary detection given the head</td>
<td>89.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Head detection</td>
<td>81.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Boundary detection given the head</td>
<td>88.74</td>
</tr>
<tr>
<td>Mention Detection</td>
<td>ACE-05 (Walker et al., 2006)</td>
<td>F1</td>
<td>Gold mention - Coarse Type</td>
<td>62.54</td>
</tr>
<tr>
<td></td>
<td>ERE</td>
<td>F1</td>
<td>Gold mention - Fine Type</td>
<td>58.35</td>
</tr>
<tr>
<td>Relation Extraction</td>
<td>ACE 2005 (Walker et al., 2006)</td>
<td>F1</td>
<td>Gold mention - Coarse Type</td>
<td>77.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Gold mention - Fine Type</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Gold mention - Coarse Type</td>
<td>68.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Gold mention - Fine Type</td>
<td>66.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Average of F1 score of MUC, B^3</td>
<td>77.05</td>
</tr>
<tr>
<td></td>
<td>CoNLL-12 (Pradhan et al., 2012)</td>
<td>F1</td>
<td>Gold mentions</td>
<td>79.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Predicted Mentions</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>ACE-04 (Doddington et al., 2004)</td>
<td>F1</td>
<td>English: NER+ link to freebase</td>
<td>76.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Spanish: NER+ link to freebase</td>
<td>75.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
<td>Chinese: NER+ link to freebase</td>
<td>70.6</td>
</tr>
</tbody>
</table>

Table 2: Qualitative evaluation of major components included in COGCOMPNLP.

library which contains various NLP components. While missing a few modules (OpenIE and Constituency parsing), COGCOMPNLP contains high-level annotators for Semantic Role Labeling of verbs, nouns, commas, and prepositions, while also offering a much wider range of supporting functionality. Compared to OPENNLPL (Balridge, 2005), SPACY\(^2\), NLTK (Bird, 2006) and TEXTBLOB (Loria et al., 2014), COGCOMPNLP has many extra annotators. While COGCOMPNLP is implemented in Java, its sister project COGCOMPNLPY makes it accessible in Python.

**Memory and speed comparison.** To measure speed and memory, we compile a collection of English raw text files extracted from a news corpus (NYT files from English Gigaword v5). The collected plain text corpus has 999 files, 665233 words (using Linux `wc -w` command).

To measure time and memory, we use the GNU time command, which is programming-language agnostic. It reports the time that it takes to run the program from start to end, and it also captures the maximum resident set size of the process during its lifetime. In reporting time we use three definitions: (a) wall-clock time is the time that a clock on the wall (or a stopwatch in hand) would measure as having elapsed between the start and end of the process. (b) user time is the amount of time spent in user code, and (c) system time the amount of time spent in the kernel. For each of the systems we use their latest available version (summarized in Table 3).

Often NLP systems are implemented as a sequence of interdependent components, and one would ideally measure their specifications using only the subsequence up to and including the component being evaluated. However, not all systems have the same ordering of the components internally. In what follows, we show the speed/memory it takes

\(^2\)https://spacy.io
to run the component being evaluated, with everything not needed being turned off.

The results of the speed evaluation for the three tasks of sentence-splitting, tokenizing, and POS tagging are summarized in Figure 4. Overall, JDK-based systems tend to need more memory (subfigure (a)). Differences in speed are often related to the task at hand; for example for POS tagging or tokenizing we do not see any significant difference.

In Figure 4 we do not show any results on SPACY since unlike other tools, in SPACY we did not find a clean-cut way to study individual modules in isolation. In order to compare other tools with SPACY we perform another experiment where all the systems annotate Tokenizing, Sentence Splitting, POS tagging and Named Entity Recognition. TEXTBLOB is not presented since it does not have an NER.

Figure 4: Speed and memory comparison between major NLP pipelines. SPACY is not shown in any of the figures, since its components are not easy to separate.

Figure 5: Speed: Wall clock time (thousand tokens / second): **higher is better**. The systems are set to produce Tokenizing, Sentence Splitting, POS tagging and Named Entity Recognition. TEXTBLOB is not presented since it does not have an NER.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CogCompNLP</td>
<td>4.0.2</td>
</tr>
<tr>
<td>CORENLP</td>
<td>3.8.0</td>
</tr>
<tr>
<td>SPACY</td>
<td>2.0.5</td>
</tr>
<tr>
<td>NLTK</td>
<td>3.2.5</td>
</tr>
<tr>
<td>TEXTBLOB</td>
<td>0.15.1</td>
</tr>
<tr>
<td>Maven</td>
<td>3.5.2</td>
</tr>
<tr>
<td>Java</td>
<td>1.8.0-151</td>
</tr>
<tr>
<td>Python</td>
<td>3.5.2</td>
</tr>
</tbody>
</table>

Table 3: Specs of the software and hardware used in our evaluation. For a fair comparison, we used the same machine to run all components.

5. **SRL: A Sophisticated Application**

In this section we illustrate the ways CogCompNLP supports development and evaluation of complex NLP applications using the task of Semantic Role Labeling (SRL) as an example. SRL identifies predicate-argument structures in
String corpusDir = "/local/path/to/ontonotes5.0/corpus/";
OntonotesReader srlReader = new OntonotesSrlReader(corpusDir);
while (srlReader.hasNext()) {
    TextAnnotation documentSrl = srlReader.next();
    PredicateArgumentView srlView = (PredicateArgumentView) documentSrl.getView(ViewNames.SRL_VERB);
    for (Constituent verbPredicate : srlView.getPredicates()) {
        for (Relation argumentRelation : srlView.getArguments(verbPredicate)) {
            String argumentType = argumentRelation.getRelationName();
            Constituent argument = argumentRelation.getTarget();
            ...
        }
    }
}

Figure 6: Code snippet for reading OntoNotes and iterating over SRL predicates

Annotator mySrlApp = new MySrlApp(); // initialize the SRL annotator
AnnotatorService pipeline = PipelineFactory.buildPipeline(ViewNames.Lemma, ViewNames.POS);
pipeline.addAnnotator(mySrlApp); // add the annotator to an pipeline
String text = "John Smith said Jane Smith bought four cakes and two apples."
TextAnnotation outputTa = pipeline.createAnnotatedTextAnnotation(text); // annotate with annotators

Figure 7: Code snippet for adding a new component to the pipeline. The expected output is shown in Figure 2.

5.4 Training and Evaluating a Classifier

Now that we have features extracted, we can generate examples for a classifier by comparing candidate boundaries with gold standard SRL argument boundaries to determine the appropriate label. These examples can be collected and passed to a learning component defined using LB-JAVA (Rizzolo and Roth, 2010), as shown in Figure 9. (see (Rizzolo and Roth, 2010) for a detailed overview and examples of LBJAVA).

The learner’s performance can be easily computed using classes from COGCOMPNLPS’s core-utilities module. The library provides a range of supporting functionality for cross-validation and computation of experimental statistics that goes well beyond what we can illustrate here.

5.5 Using the New NLP Component

Once a new component/application has been developed, its outputs can easily be made available to other applications either via serialization, or by making the component an Annotator and adding it to the pipeline. Figure 7 shows how these tasks can be done.

6. Conclusion

COGCOMPNLP is a mature, well-architected, and largely well-documented NLP framework available under an academic license. It has been developed with the twin goals of making it easy to obtain NLP annotations from off-the-shelf components, and developing and evaluating new applications. In this paper we explain the NLP tools it contains; the pipeline application for applying these tools to input text; its word and phrase similarity metrics. In an earlier work (Sammons et al., 2016) we have described its support for feature extraction. We have illustrated its use to develop sophisticated NLP applications by showing how to implement Semantic Role Labeling, a key NLP component.
public static class IsVerb extends Predicate<Constituent> {
    @Override
    public Boolean transform(Constituent c) {
        return c.getLabel().startsWith("V");
    }
}

/** extract positive and negative examples for training. */
public static List<Pair<String, Pair<Constituent, Constituent>>> generateExamples(TextAnnotation ta) {
    List<Pair<String, Pair<Constituent, Constituent>>> examples = new ArrayList<>();
    PredicateArgumentView verbSrlView = (PredicateArgumentView) ta.getView(ViewNames.SRL_VERB);
    IQueryable<Constituent> verbs = new QueryableList<>(ta.getView(ViewNames.POS).getConstituents()).where(new IsVerb());
    TreeView parseView = (TreeView) ta.getView(ViewNames.PARSE_STANFORD);
    for (Constituent verb : verbs) {
        List<Constituent> allArgCandidates = getArgCandidates(parseView, verb.getStartSpan());
        for (Constituent candidate : allArgCandidates) {
            String label = "negative";
            if (findGoldArgMatch(candidate, verb, verbSrlView)) label = "positive";
            examples.add(new Pair(label, new Pair(candidate, verb)));
        }
    }
    return examples;
}

/** check in SRL View whether candidate argument, predicate match a gold argument, predicate pair */
public static boolean findGoldArgMatch(Constituent candidate, Constituent verb, PredicateArgumentView srlView) {
    List<Constituent> srlArgMatches = srlView.getConstituentsMatchingSpan(candidate.getStartSpan(), candidate.getEndSpan());
    Set<Constituent> srlPredMatches = new HashSet<>(srlView.getConstituentsMatchingSpan(verb.getStartSpan(), verb.getEndSpan()));
    srlPredMatches.retainAll(srlView.getPredicates());
    for (Constituent srlArgMatch : srlArgMatches)
        for (Relation rel : srlArgMatch.getIncomingRelations())
            if (srlPredMatches.contains(rel.getSource()))
                return true;
    return false;
}

/** use syntactic parse to identify parse constituents near the predicate */
public static List<Constituent> getArgCandidates(TreeView parseView, int predicateIndex) {
    List<Constituent> argumentCandidates = new ArrayList<>();
    Tree<Constituent> t = parseView.getConstituentTree(0);
    Tree<Constituent> predicateTree = t.getYield().get(predicateIndex).getParent();
    Tree<Constituent> predicateParent = predicateTree.getParent();
    for (Tree<Constituent> sibling : predicateParent.getChildren())
        argumentCandidates.add(sibling.getLabel());
    for (Tree<Constituent> child : predicateTree.getChildren())
        if (child != predicateTree) argumentCandidates.add(child.getLabel());
    return argumentCandidates;
}

Figure 8: Code snippet for generating verb SRL argument examples.

Acknowledgement

The authors would like to thank all the other contributors to the project. This material is partly based on research sponsored by DARPA under agreement number FA8750-13-2-0008. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was also supported by Contract HR0011-15-2-0025 with the US Defense Advanced Research Projects Agency (DARPA). Approved for Public Release, Distribution Unlimited. This work was partly funded by a grant from the Allen Institute for Artificial Intelligence (allenai.org); by Google; by NSF grant BCS-1348522; and by NIH grant R01-HD054448. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any research sponsors listed above.

References

/** a trivial classifier, to return a feature for a predicate-argument structure */
public class SrIWordAndPos extends Classifier {
    private WordAndPos wordAndPos = new WordAndPos();
    @Override
    public FeatureVector classify(Object arg) {
        Pair<Constituent, Constituent> argAndPred = ((Pair<String, Pair<Constituent, Constituent>>) arg).getSecond();
        return FeatureUtilities.getLBJFeatures(FeatureUtilities.conjoin(
            wordAndPos.getFeatures(argAndPred.getFirst()), wordAndPos.getFeatures(argAndPred.getSecond())));
    }
}

// LBJava code, to define a classifier, its input features and the expected output
import SrIWordAndPos;

discrete ArgumentIdentifier(Pair<String, Pair<Constituent, Constituent>> argAndPred) <-
learn IsArgument
using SrIWordAndPos
from new SrIArgumentReader(Constants.corpusHome) 50 rounds
with SparseNetworkLearner {
    SparseAveragedPerceptron.Parameters p = new SparseAveragedPerceptron.Parameters();
    p.learningRate = .1;
    baseLU = new SparseAveragedPerceptron(p);
}

Figure 9: LBJava code for training and evaluating an SRL argument identifier classifier. The ArgParser argument is a class that extends LBJava’s Parser interface, and returns an argument example (label ‘positive’ or ‘negative’, argument constituent, and predicate constituent) for each call of its next() method.
