

# The Stanford CoreNLP Natural Language Processing Toolkit

**Christopher D. Manning**  
Linguistics & Computer Science  
Stanford University  
manning@stanford.edu

**Mihai Surdeanu**  
SISTA  
University of Arizona  
msurdeanu@email.arizona.edu

**John Bauer**  
Dept of Computer Science  
Stanford University  
horatio@stanford.edu

**Jenny Finkel**  
Prismatic Inc.  
jrfinkel@gmail.com

**Steven J. Bethard**  
Computer and Information Sciences  
U. of Alabama at Birmingham  
bethard@cis.uab.edu

**David McClosky**  
IBM Research  
dmcclosky@us.ibm.com

## Abstract

We describe the design and use of the Stanford CoreNLP toolkit, an extensible pipeline that provides core natural language analysis. This toolkit is quite widely used, both in the research NLP community and also among commercial and government users of open source NLP technology. We suggest that this follows from a simple, approachable design, straightforward interfaces, the inclusion of robust and good quality analysis components, and not requiring use of a large amount of associated baggage.

## 1 Introduction

This paper describes the design and development of Stanford CoreNLP, a Java (or at least JVM-based) annotation pipeline framework, which provides most of the common core natural language processing (NLP) steps, from tokenization through to coreference resolution. We describe the original design of the system and its strengths (section 2), simple usage patterns (section 3), the set of provided annotators and how properties control them (section 4), and how to add additional annotators (section 5), before concluding with some higher-level remarks and additional appendices. While there are several good natural language analysis toolkits, Stanford CoreNLP is one of the most used, and a central theme is trying to identify the attributes that contributed to its success.

## 2 Original Design and Development

Our pipeline system was initially designed for internal use. Previously, when combining multiple natural language analysis components, each with their own ad hoc APIs, we had tied them together with custom glue code. The initial version of the

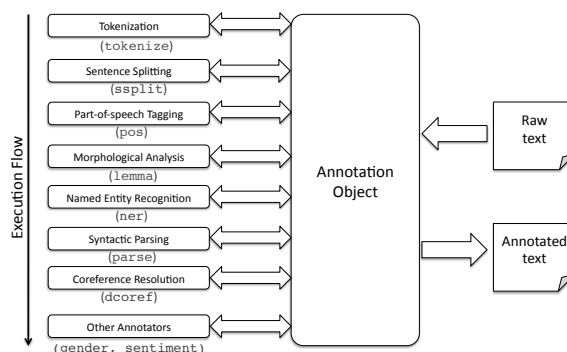


Figure 1: Overall system architecture: Raw text is put into an Annotation object and then a sequence of Annotators add information in an analysis pipeline. The resulting Annotation, containing all the analysis information added by the Annotators, can be output in XML or plain text forms.

annotation pipeline was developed in 2006 in order to replace this jumble with something better. A uniform interface was provided for an Annotator that adds some kind of analysis information to some text. An Annotator does this by taking in an Annotation object to which it can add extra information. An Annotation is stored as a typesafe heterogeneous map, following the ideas for this data type presented by Bloch (2008). This basic architecture has proven quite successful, and is still the basis of the system described here. It is illustrated in figure 1. The motivations were:

- To be able to quickly and painlessly get linguistic annotations for a text.
- To hide variations across components behind a common API.
- To have a minimal conceptual footprint, so the system is easy to learn.
- To provide a lightweight framework, using plain Java objects (rather than something of heavier weight, such as XML or UIMA's Common Analysis System (CAS) objects).

In 2009, initially as part of a multi-site grant project, the system was extended to be more easily usable by a broader range of users. We provided a command-line interface and the ability to write out an Annotation in various formats, including XML. Further work led to the system being released as free open source software in 2010.

On the one hand, from an architectural perspective, Stanford CoreNLP does not attempt to do everything. It is nothing more than a straightforward pipeline architecture. It provides only a Java API.<sup>1</sup> It does not attempt to provide multiple machine scale-out (though it does provide multi-threaded processing on a single machine). It provides a simple concrete API. But these requirements satisfy a large percentage of potential users, and the resulting simplicity makes it easier for users to get started with the framework. That is, the primary advantage of Stanford CoreNLP over larger frameworks like UIMA (Ferrucci and Lally, 2004) or GATE (Cunningham et al., 2002) is that users do not have to learn UIMA or GATE before they can get started; they only need to know a little Java. In practice, this is a large and important differentiator. If more complex scenarios are required, such as multiple machine scale-out, they can normally be achieved by running the analysis pipeline within a system that focuses on distributed workflows (such as Hadoop or Spark). Other systems attempt to provide more, such as the UIUC Curator (Clarke et al., 2012), which includes inter-machine client-server communication for processing and the caching of natural language analyses. But this functionality comes at a cost. The system is complex to install and complex to understand. Moreover, in practice, an organization may well be committed to a scale-out solution which is different from that provided by the natural language analysis toolkit. For example, they may be using Kryo or Google's protobuf for binary serialization rather than Apache Thrift which underlies Curator. In this case, the user is better served by a fairly small and self-contained natural language analysis system, rather than something which comes with a lot of baggage for all sorts of purposes, most of which they are not using.

On the other hand, most users benefit greatly from the provision of a set of stable, robust, high

---

<sup>1</sup>Nevertheless, it can call an analysis component written in other languages via an appropriate wrapper Annotator, and in turn, it has been wrapped by many people to provide Stanford CoreNLP bindings for other languages.

quality linguistic analysis components, which can be easily invoked for common scenarios. While the builder of a larger system may have made overall design choices, such as how to handle scale-out, they are unlikely to be an NLP expert, and are hence looking for NLP components that just work. This is a huge advantage that Stanford CoreNLP and GATE have over the empty toolbox of an Apache UIMA download, something addressed in part by the development of well-integrated component packages for UIMA, such as ClearTK (Bethard et al., 2014), DKPro Core (Gurevych et al., 2007), and JCoRe (Hahn et al., 2008). However, the solution provided by these packages remains harder to learn, more complex and heavier weight for users than the pipeline described here.

These attributes echo what Patricio (2009) argued made Hibernate successful, including: (i) do one thing well, (ii) avoid over-design, and (iii) up and running in ten minutes or less! Indeed, the design and success of Stanford CoreNLP also reflects several other of the factors that Patricio highlights, including (iv) avoid standardism, (v) documentation, and (vi) developer responsiveness. While there are many factors that contribute to the uptake of a project, and it is hard to show causality, we believe that some of these attributes account for the fact that Stanford CoreNLP is one of the more used NLP toolkits. While we certainly have not done a perfect job, compared to much academic software, Stanford CoreNLP has gained from attributes such as clear open source licensing, a modicum of attention to documentation, and attempting to answer user questions.

### 3 Elementary Usage

A key design goal was to make it very simple to set up and run processing pipelines, from either the API or the command-line. Using the API, running a pipeline can be as easy as figure 2. Or, at the command-line, doing linguistic processing for a file can be as easy as figure 3. Real life is rarely this simple, but the ability to get started using the product with minimal configuration code gives new users a very good initial experience.

Figure 4 gives a more realistic (and complete) example of use, showing several key properties of the system. An annotation pipeline can be applied to any text, such as a paragraph or whole story rather than just a single sentence. The behavior of

```

Annotator pipeline = new StanfordCoreNLP();
Annotation annotation = new Annotation(
    "Can you parse my sentence?");
pipeline.annotate(annotation);

```

Figure 2: Minimal code for an analysis pipeline.

```

export StanfordCoreNLP_HOME /where/installed
java -Xmx2g -cp $StanfordCoreNLP_HOME/*
    edu.stanford.nlp.StanfordCoreNLP
    -file input.txt

```

Figure 3: Minimal command-line invocation.

```

import java.io.*;
import java.util.*;
import edu.stanford.nlp.io.*;
import edu.stanford.nlp.ling.*;
import edu.stanford.nlp.pipeline.*;
import edu.stanford.nlp.trees.*;
import edu.stanford.nlp.trees.TreeCoreAnnotations.*;
import edu.stanford.nlp.util.*;

public class StanfordCoreNlpExample {

    public static void main(String[] args) throws IOException {
        PrintWriter xmlOut = new PrintWriter("xmlOutput.xml");
        Properties props = new Properties();
        props.setProperty("annotators",
            "tokenize, ssplit, pos, lemma, ner, parse");
        StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
        Annotation annotation = new Annotation(
            "This is a short sentence. And this is another.");

        pipeline.annotate(annotation);
        pipeline.xmlPrint(annotation, xmlOut);

        // An Annotation is a Map and you can get and use the
        // various analyses individually. For instance, this
        // gets the parse tree of the 1st sentence in the text.

        List<CoreMap> sentences = annotation.get(
            CoreAnnotations.SentencesAnnotation.class);
        if (sentences != null && sentences.size() > 0) {
            CoreMap sentence = sentences.get(0);
            Tree tree = sentence.get(TreeAnnotation.class);
            PrintWriter out = new PrintWriter(System.out);
            out.println("The first sentence parsed is:");
            tree.pennPrint(out);
        }
    }
}

```

Figure 4: A simple, complete example program.

annotators in a pipeline is controlled by standard Java properties in a Properties object. The most basic property to specify is what annotators to run, in what order, as shown here. But as discussed below, most annotators have their own properties to allow further customization of their usage. If none are specified, reasonable defaults are used. Running the pipeline is as simple as in the first example, but then we show two possibilities for accessing the results. First, we convert the Annotation object to XML and write it to a file. Second, we show code that gets a particular type of information out of an Annotation and then prints it.

Our presentation shows only usage in Java, but the Stanford CoreNLP pipeline has been wrapped by others so that it can be accessed easily from many languages, including Python, Ruby, Perl, Scala, Clojure, Javascript (node.js), and .NET lan-

guages, including C# and F#.

## 4 Provided annotators

The annotators provided with StanfordCoreNLP can work with any character encoding, making use of Java's good Unicode support, but the system defaults to UTF-8 encoding. The annotators also support processing in various human languages, providing that suitable underlying models or resources are available for the different languages. The system comes packaged with models for English. Separate model packages provide support for Chinese and for case-insensitive processing of English. Support for other languages is less complete, but many of the Annotators also support models for French, German, and Arabic (see appendix B), and building models for further languages is possible using the underlying tools. In this section, we outline the provided annotators, focusing on the English versions. It should be noted that some of the models underlying annotators are trained from annotated corpora using supervised machine learning, while others are rule-based components, which nevertheless often require some language resources of their own.

**tokenize** Tokenizes the text into a sequence of tokens. The English component provides a PTB-style tokenizer, extended to reasonably handle noisy and web text. The corresponding components for Chinese and Arabic provide word and clitic segmentation. The tokenizer saves the character offsets of each token in the input text.

**cleanxml** Removes most or all XML tags from the document.

**ssplit** Splits a sequence of tokens into sentences.

**truecase** Determines the likely true case of tokens in text (that is, their likely case in well-edited text), where this information was lost, e.g., for all upper case text. This is implemented with a discriminative model using a CRF sequence tagger (Finkel et al., 2005).

**pos** Labels tokens with their part-of-speech (POS) tag, using a maximum entropy POS tagger (Toutanova et al., 2003).

**lemma** Generates the lemmas (base forms) for all tokens in the annotation.

**gender** Adds likely gender information to names.

**ner** Recognizes named (PERSON, LOCATION, ORGANIZATION, MISC) and numerical (MONEY, NUMBER, DATE, TIME, DURATION, SET) entities. With the default

annotators, named entities are recognized using a combination of CRF sequence taggers trained on various corpora (Finkel et al., 2005), while numerical entities are recognized using two rule-based systems, one for money and numbers, and a separate state-of-the-art system for processing temporal expressions (Chang and Manning, 2012).

**regexner** Implements a simple, rule-based NER over token sequences building on Java regular expressions. The goal of this Annotator is to provide a simple framework to allow a user to incorporate NE labels that are not annotated in traditional NL corpora. For example, a default list of regular expressions that we distribute in the models file recognizes ideologies (IDEOLOGY), nationalities (NATIONALITY), religions (RELIGION), and titles (TITLE).

**parse** Provides full syntactic analysis, including both constituent and dependency representation, based on a probabilistic parser (Klein and Manning, 2003; de Marneffe et al., 2006).

**sentiment** Sentiment analysis with a compositional model over trees using deep learning (Socher et al., 2013). Nodes of a binarized tree of each sentence, including, in particular, the root node of each sentence, are given a sentiment score.

**dcoref** Implements mention detection and both pronominal and nominal coreference resolution (Lee et al., 2013). The entire coreference graph of a text (with head words of mentions as nodes) is provided in the Annotation.

Most of these annotators have various options which can be controlled by properties. These can either be added to the Properties object when creating an annotation pipeline via the API, or specified either by command-line flags or through a properties file when running the system from the command-line. As a simple example, input to the system may already be tokenized and presented one-sentence-per-line. In this case, we wish the tokenization and sentence splitting to just work by using the whitespace, rather than trying to do anything more creative (be it right or wrong). This can be accomplished by adding two properties, either to a properties file:

```
tokenize.whitespace: true
ssplit.eolonly:     true
```

in code:

```
/** Simple annotator for locations stored in a gazetteer. */
package org.foo;
public class GazetteerLocationAnnotator implements Annotator {
    // this is the only method an Annotator must implement
    public void annotate(Annotation annotation) {
        // traverse all sentences in this document
        for (CoreMap sentence:annotation.get(SentencesAnnotation.class)) {
            // loop over all tokens in sentence (the text already tokenized)
            List<CoreLabel> toks = sentence.get(TokensAnnotation.class);
            for (int start = 0; start < toks.size(); start++) {
                // assumes that the gazetteer returns the token index
                // after the match or -1 otherwise
                int end = Gazetteer.isLocation(toks, start);
                if (end > start) {
                    for (int i = start; i < end; i++) {
                        toks.get(i).set(NamedEntityTagAnnotation.class,"LOCATION");
                    }
                }
            }
        }
    }
}
```

Figure 5: An example of a simple custom annotator. The annotator marks the words of possibly multi-word locations that are in a gazetteer.

```
props.setProperty("tokenize.whitespace", "true");
props.setProperty("ssplit.eolonly", "true");
```

or via command-line flags:

```
-tokenize.whitespace -ssplit.eolonly
```

We do not attempt to describe all the properties understood by each annotator here; they are available in the documentation for Stanford CoreNLP. However, we note that they follow the pattern of being  $x.y$ , where  $x$  is the name of the annotator that they apply to.

## 5 Adding annotators

While most users work with the provided annotators, it is quite easy to add additional custom annotators to the system. We illustrate here both how to write an Annotator in code and how to load it into the Stanford CoreNLP system. An Annotator is a class that implements three methods: a single method for analysis, and two that describe the dependencies between analysis steps:

```
public void annotate(Annotation annotation);
public Set<Requirement> requirementsSatisfied();
public Set<Requirement> requires();
```

The information in an Annotation is updated in place (usually in a non-destructive manner, by adding new keys and values to the Annotation). The code for a simple Annotator that marks locations contained in a gazetteer is shown in figure 5.<sup>2</sup> Similar code can be used to write a wrapper Annotator, which calls some pre-existing analysis component, and adds its results to the Annotation.

<sup>2</sup>The functionality of this annotator is already provided by the regexner annotator, but it serves as a simple example.

Stanford CoreNLP can apply additional annotators, which are loaded using reflection. To provide a new Annotator, the user extends the class `edu.stanford.nlp.pipeline.Annotator` and provides a constructor with the signature `(String, Properties)`. Then, the user adds the property

```
customAnnotatorClass.FOO: BAR
```

This says that if FOO appears in the list of annotators, then the class BAR will be loaded to instantiate it. The Properties object is also passed to the constructor, so that annotator-specific behavior can be initialized from the Properties object. For instance, for the example above, the properties file lines might be:

```
customAnnotatorClass.loggaz: org.foo.GazetteerLocationAnnotator
annotators: tokenize,ssplit,loggaz
loggaz.maxLength: 5
```

## 6 Conclusion

In this paper, we have presented the design and usage of the Stanford CoreNLP system, an annotation-based NLP processing pipeline. We have in particular tried to emphasize the properties that we feel have made it successful. Rather than trying to provide the largest and most engineered kitchen sink, the goal has been to make it as easy as possible for users to get started using the framework, and to keep the framework small, so it is easily comprehensible, and can easily be used as a component within the much larger system that a user may be developing. The broad usage of this system, and of other systems such as NLTK (Bird et al., 2009), which emphasize accessibility to beginning users, suggests the merits of this approach.

## A Pointers

Website: <http://nlp.stanford.edu/software/corenlp.shtml>

Github: <https://github.com/stanfordnlp/CoreNLP>

Maven: <http://mvnrepository.com/artifact/edu.stanford.nlp/stanford-corenlp>

User Questions Mailing list: <https://mailman.stanford.edu/mailman/listinfo/java-nlp-user>

License: GPL v2+

Stanford CoreNLP keeps the models for machine learning components and miscellaneous other data files in a separate models jar file. If you are using Maven, you need to make sure that you

list the dependency on this models file as well as the code jar file. You can do that with code like the following in your pom.xml. Note the extra dependency with a classifier element at the bottom.

```
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-corenlp</artifactId>
  <version>3.3.1</version>
</dependency>
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-corenlp</artifactId>
  <version>3.3.1</version>
  <classifier>models</classifier>
</dependency>
```

## B Human language support

We summarize the analysis components supported for different human languages in early 2014.

Annotator	Ara- bic	Chi- nese	Eng- lish	Fre- nch	Ger- man
Tokenize	✓	✓	✓	✓	✓
Sent. split	✓	✓	✓	✓	✓
Truecase			✓		
POS	✓	✓	✓	✓	✓
Lemma			✓		
Gender			✓		
NER		✓	✓		✓
RegexNER	✓	✓	✓	✓	✓
Parse	✓	✓	✓	✓	✓
Dep. Parse		✓	✓		
Sentiment			✓		
Coref.			✓		

## C Getting the sentiment of sentences

We show a command-line for sentiment analysis.

```
$ cat sentiment.txt
I liked it.
It was a fantastic experience.
The plot move rather slowly.
$ java -cp "*" -Xmx2g edu.stanford.nlp.pipeline.StanfordCoreNLP -annotators
tokenize,ssplit,pos,lemma,parse,sentiment -file sentiment.txt
Adding annotator tokenize
Adding annotator ssplit
Adding annotator pos
Reading POS tagger model from edu/stanford/nlp/models/pos-tagger/
english-left3words/english-left3words-distsim.tagger ... done [1.0 sec].
Adding annotator lemma
Adding annotator parse
Loading parser from serialized file edu/stanford/nlp/models/lexparser/
englishPCFG.ser.gz ... done [1.4 sec].
Adding annotator sentiment

Ready to process: 1 files, skipped 0, total 1
Processing file /Users/manning/Software/stanford-corenlp-full-2014-01-04/
sentiment.txt ... writing to /Users/manning/Software/
stanford-corenlp-full-2014-01-04/sentiment.txt.xml {
  Annotating file /Users/manning/Software/stanford-corenlp-full-2014-01-04/
sentiment.txt [0.583 seconds]
} [1.219 seconds]
Processed 1 documents
Skipped 0 documents, error annotating 0 documents
Annotation pipeline timing information:
PTBTokenizerAnnotator: 0.0 sec.
WordsToSentencesAnnotator: 0.0 sec.
POSTaggerAnnotator: 0.0 sec.
```

```

MorphaAnnotator: 0.0 sec.
ParserAnnotator: 0.4 sec.
SentimentAnnotator: 0.1 sec.
TOTAL: 0.6 sec. for 16 tokens at 27.4 tokens/sec.
Pipeline setup: 3.0 sec.
Total time for StanfordCoreNLP pipeline: 4.2 sec.
$ grep sentiment sentiment.txt.xml
<sentence id="1" sentimentValue="3" sentiment="Positive">
<sentence id="2" sentimentValue="4" sentiment="Verypositive">
<sentence id="3" sentimentValue="1" sentiment="Negative">

```

## D Use within UIMA

The main part of using Stanford CoreNLP within the UIMA framework (Ferrucci and Lally, 2004) is mapping between CoreNLP annotations, which are regular Java classes, and UIMA annotations, which are declared via XML type descriptors (from which UIMA-specific Java classes are generated). A wrapper for CoreNLP will typically define a subclass of `JCasAnnotator_ImplBase` whose process method: (i) extracts UIMA annotations from the CAS, (ii) converts UIMA annotations to CoreNLP annotations, (iii) runs CoreNLP on the input annotations, (iv) converts the CoreNLP output annotations into UIMA annotations, and (v) saves the UIMA annotations to the CAS.

To illustrate part of this process, the ClearTK (Bethard et al., 2014) wrapper converts CoreNLP token annotations to UIMA annotations and saves them to the CAS with the following code:

```

int begin = tokenAnn.get(CharacterOffsetBeginAnnotation.class);
int end = tokenAnn.get(CharacterOffsetEndAnnotation.class);
String pos = tokenAnn.get(PartOfSpeechAnnotation.class);
String lemma = tokenAnn.get(LemmaAnnotation.class);
Token token = new Token(jCas, begin, end);
token.setPos(pos);
token.setLemma(lemma);
token.addToIndexes();

```

where `Token` is a UIMA type, declared as:

```

<typeSystemDescription>
<name>Token</name>
<types>
<typeDescription>
<name>org.cleartk.token.type.Token</name>
<supertypeName>uima.tcas.Annotation</supertypeName>
<features>
<featureDescription>
<name>pos</name>
<rangeTypeName>uima.cas.String</rangeTypeName>
</featureDescription>
<featureDescription>
<name>lemma</name>
<rangeTypeName>uima.cas.String</rangeTypeName>
</featureDescription>
</features>
</typeDescription>
</types>
</typeSystemDescription>

```

## References

Steven Bethard, Philip Ogren, and Lee Becker. 2014. ClearTK 2.0: Design patterns for machine learning in UIMA. In *LREC 2014*.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O’Reilly Media.

Joshua Bloch. 2008. *Effective Java*. Addison Wesley, Upper Saddle River, NJ, 2nd edition.

Angel X. Chang and Christopher D. Manning. 2012. SUTIME: A library for recognizing and normalizing time expressions. In *LREC 2012*.

James Clarke, Vivek Srikumar, Mark Sammons, and Dan Roth. 2012. An NLP Curator (or: How I learned to stop worrying and love NLP pipelines). In *LREC 2012*.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: an architecture for development of robust HLT applications. In *ACL 2002*.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *LREC 2006*, pages 449–454.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10:327–348.

Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating non-local information into information extraction systems by Gibbs sampling. In *ACL 43*, pages 363–370.

I. Gurevych, M. Mühlhäuser, C. Müller, J. Steimle, M. Weimer, and T. Zesch. 2007. Darmstadt knowledge processing repository based on UIMA. In *First Workshop on Unstructured Information Management Architecture at GLDV 2007*, Tübingen.

U. Hahn, E. Buyko, R. Landefeld, M. Mühlhausen, Poprat M, K. Tomanek, and J. Wermter. 2008. An overview of JCoRe, the Julie lab UIMA component registry. In *LREC 2008*.

Dan Klein and Christopher D. Manning. 2003. Fast exact inference with a factored model for natural language parsing. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15, pages 3–10. MIT Press.

Heeyoung Lee, Angel Chang, Yves Peirsman, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. 2013. Deterministic coreference resolution based on entity-centric, precision-ranked rules. *Computational Linguistics*, 39(4).

Anthony Patricio. 2009. Why this project is successful? <https://community.jboss.org/wiki/WhyThisProjectIsSuccessful>.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP 2013*, pages 1631–1642.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL 3*, pages 252–259.