Jigg: A Framework for an Easy Natural Language Processing Pipeline

Hiroshi Noji
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, Japan
noji@is.naist.jp

Yusuke Miyao
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku,
Tokyo, Japan
yusuke@nii.ac.jp

Abstract

We present Jigg, a Scala (or JVM-based) NLP annotation pipeline framework, which is easy to use and is extensible. Jigg supports a very simple interface similar to Stanford CoreNLP, the most successful NLP pipeline toolkit, but has more flexibility to adapt to new types of annotation. On this framework, system developers can easily integrate their downstream system into a NLP pipeline from a raw text by just preparing a wrapper of it.

1 Introduction

A common natural language processing system works as a component in a pipeline. For example, a syntactic parser typically requires that an input sentence is correctly tokenized or assigned part-of-speech (POS) tags. The syntactic trees given by the parser may be required in further downstream tasks such as coreference resolution and semantic role labelling. While this pipeline-based approach has been quite successful due to its modularity, it suffers from several drawbacks from a viewpoint of software use and development:

- For a *user*, building a pipeline connecting existing tools and aggregating the outputs are painful, since often each system outputs the results in a different format;
- For researchers or tool developers of downstream tasks, supporting the full pipeline from an input text in their software is boring and time consuming.

For example, two famous dependency parsing systems, MaltParser (Nivre et al., 2006) and MSTParser (McDonald et al., 2005), both assume that an input sentence is already tokenized and assigned POS tags, and encoded in a specific format, such as the CoNLL format.

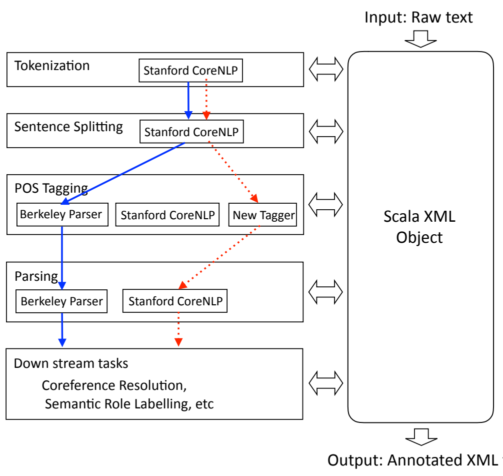


Figure 1: In a pipeline, annotations are performed on a Scala XML object. A pipeline is built by choosing annotator tools at each step, e.g., the bold or dotted lines in the figure. Each component is implemented as a wrapper, which manipulates the XML object. If we prepare a new wrapper of some component, one can integrate it in a pipeline (e.g., the POS tagger in the dotted lines).

In this paper, we present Jigg, which aims to make it easy to incorporate an existing or new tool (component) in an NLP pipeline. Figure 1 describes the overview. Using Jigg, a user can easily construct a pipeline by choosing a tool at each step on a command-line interface. Jigg is written in Scala, and can easily be extended with JVM languages including Java. A new tool can be incorporated into this framework by writing a wrapper of that to follow the common API of Jigg (Scala XML object), which requires typically several dozes of lines of code.

The software design of Jigg is highly inspired by the success of Stanford CoreNLP (Manning et al., 2014), which is now the most widely used NLP toolkit supporting pipeline processing from raw texts. One characteristic of Stanford CoreNLP is its simplicity of API, which allows wider users to easily get linguistic annotations for a text. Following this strategy, Jigg is also quite simple to use; all the basic components are included into one jar file, so a user need not install the external dependencies. The basic usage of Jigg is command-line interface, and the behavior can be customized with a Java properties file. On the other hand, it focuses just on processing of a single document on a single machine, and does not provide the solution to more complex scenarios such as distributed processing or visualization, which UIMA and related projects (Ferrucci and Lally, 2004; Kano et al., 2011) may provide.

The largest difference between Jigg and Stanford CoreNLP is the focused NLP components. Stanford CoreNLP is basically a collection of NLP tools developed by the Stanford NLP group, e.g., Stanford POS tagger (Toutanova et al., 2003) and Stanford parser (Socher et al., 2013). Jigg, on the other hand, is an integration framework of various NLP tools developed by various groups. This means that adding a new component in Jigg is easier than Stanford CoreNLP. Also as indicated in Figure 1, Jigg provides a wrapper to Stanford CoreNLP itself, so a user can enjoy combination of Stanford CoreNLP and other tools, e.g., Berkeley parser (Petrov and Klein, 2007) (see Section 2). This difference essentially comes from the underlying object annotated on each step, which is CoreMap object in Stanford CoreNLP, and Scala XML object in Jigg, which gives more flexibility as we describe later (Section 5). Before that, in the following, we first describes the concrete usage (Section 2), the core software design (Section 3), and a way to add a new component (Section 4).

The code is open-source under the Apache License Version 2.0. Followings are the pointers to the related websites:

- Github: `https://github.com/mynlp/jigg`
- Maven: `http://mvnrepository.com/artifact/com.github.mynlp/jigg`

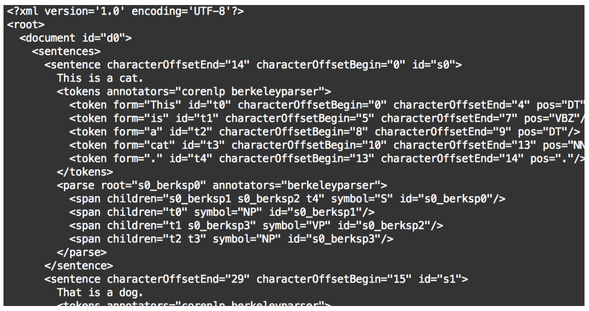
Jigg is also available from Maven, so it can easily be incorporated into another JVM project. See REAME on the project Github for this usage.

2 Basic Usages

As an example, let us consider the scenario to run the Berkeley parser on a raw text. This parser is state-of-the-art but it requires that the input is correctly tokenized and splitted on sentences. Figure 2 shows a concrete command-line to build a pipeline, on which tokenization and sentence splitting are performed using the components in Stanford CoreNLP. This pipeline corresponds to the bold lines in Figure 1. jigg.pipeline.Pipeline is the path to the main class. −annotators argument is essential, and specifies which components (tools) one wishes to apply. In the command-line, corenlp[tokenize, ssplit] is an abbreviation of two components, corenlp[tokenize] (tokenization) and corenlp[ssplit] (sentence splitting by CoreNLP).[1] The last argument −berkeleyparser.grFileName is necessary and specifies the path to the parser model (learned grammar).

```
$ cat sample.txt
This is a cat. That is a dog.
$ echo sample.txt | java -cp "*" \
  jigg.pipeline.Pipeline\
  -annotators "corenlp[tokenize,ssplit],berkeleyparser"\
  -berkeleyparser.grFileName ./eng_sm6.gr > sample.xml
```

Figure 2: A command-line usage to run the Berkeley parser on sentences tokenized and splitted by Stanford CoreNLP.



```
<?xml version='1.0' encoding='UTF-8'?>
<root>
  <document id="d0">
    <sentences>
      <sentence characterOffsetEnd="14" characterOffsetBegin="0" id="s0">
        This is a cat.
        <tokens annotators="corenlp berkeleyparser">
          <token form="This" id="t0" characterOffsetBegin="0" characterOffsetEnd="4" pos="DT"
          <token form="is" id="t1" characterOffsetBegin="5" characterOffsetEnd="7" pos="VBZ"/
          <token form="a" id="t2" characterOffsetBegin="8" characterOffsetEnd="9" pos="DT"/>
          <token form="cat" id="t3" characterOffsetBegin="10" characterOffsetEnd="13" pos="NN
          <token form="." id="t4" characterOffsetBegin="13" characterOffsetEnd="14" pos="."/>
        </tokens>
        <parse root="s0_berksp0" annotators="berkeleyparser">
          <span children="s0_berksp1 s0_berksp2 t4" symbol="S" id="s0_berksp0"/>
          <span children="t0" symbol="NP" id="s0_berksp1"/>
          <span children="t1 s0_berksp3" symbol="VP" id="s0_berksp2"/>
          <span children="t2 t3" symbol="NP" id="s0_berksp3"/>
        </parse>
      </sentence>
      <sentence characterOffsetEnd="29" characterOffsetBegin="15" id="s1">
        That is a dog.
        <tokens annotators="corenlp berkeleyparser">
```

Figure 3: The output of the command in Figure 2 (sample.xml).

XML output In the current implementation, the output format of annotations is always XML. Figure 3 shows the output for this example. In this output, parse element specifies a (constituent) parse tree with a collection of spans, each of which consists of a root symbol (e.g., S) and child nodes (ids). This format is intended to be easily processed with a computer, and differs in several points from the outputs of Stanford CoreNLP, which we describe more in Section 5.

[1]Precisely, the two commands have different meanings and the former abbreviated form is recommended. In the latter separated form, transformation between CoreMap object and Scala XML is performed at each step (twice), while it occurs once in the former one after ssplit.

```
import jigg.pipeline.Pipeline
import scala.xml.Node
import java.util.Properties

object ScalaExample {
  def main(args: Array[String]): Unit = {
    val props = new Properties()
    props.setProperty("annotators",
      "corenlp[tokenize,ssplit],berkeleyparser")
    props.setProperty("berkeleyparser.grFileName",
      "eng_sm6.gr")
    val pipeline = new Pipeline(props)
    val annotation: Node = pipeline.annotate(
      "This is a cat. That is a dog")

    // Find all sentence elements recursively,
    // and get the first one.
    val firstSentence = (annotation \\ "sentence")(0)

    // All tokens on the sentence
    val tokens = firstSentence \\ "token"

    println("POS tags on the first sentence: " +
      (tokens map (_ \@ "pos") mkString " "))
    // Output "DT VBZ DT NN ."
  }
}
```

Figure 4: A programmatic usage from Scala.

Properties As in Stanford CoreNLP, these arguments can be customized through a Java properties file. For example, the following properties file customizes the behavior of corenlp besides the parser:

```
$ cat sample.properties
annotators: corenlp[tokenize,ssplit],berkeleyparser
berkeleyparser.grFileName: ./eng_sm6.gr
corenlp.tokenize.whitespace: true
corenlp.ssplit.eolonly: true
```

This file can be used as follows:

```
jigg.pipeline.Pipeline -props sample.properties
```

Each annotator-specific argument has the form annotator_name.key. In the case of corenlp, all keys of the arguments prefixed with that are directly transferred to the CoreNLP object, so the all arguments defined in Stanford CoreNLP can be used to customize the behavior. The setting above yields tokenization on white spaces, and sentence splitting on new lines only (i.e., the input text is assumed to be properly preprocessed beforehand).

Programmatic usage Jigg can also be used as a Scala library, which can be called on JVM languages. Figure 4 shows an example on a Scala code. The annotate method of Pipeline object performs annotations on the given input, and returns the annotated XML object (Node class). The example also shows how we can manipulate the Scala XML object, which can be searched with methods similar to XPath, e.g., \\. \@ key returns the attribute value for the key if exists. Figure 5 shows that Jigg can also be used via a Java code.

Another example Jigg is a growing project, and the supported tools are now increasing. Historically, Jigg has been started as a pipeline framework focusing on Japanese language processing. Jigg thus supports many Japanese processing tools such as MeCab (Kudo et al., 2004), a famous morphological analyzer, as well as a Japanese CCG parser based on the Japanese CCGBank (Uematsu et al., 2013). For English, currently the core tool is Stanford CoreNLP. Here we present an interesting application to integrate Berkeley parser into the full pipeline of Stanford CoreNLP:

```
Properties props = new Properties();
props.setProperty("annotators",
  "corenlp[tokenize,ssplit],berkeleyparser");
props.setProperty("berkeleyparser.grFileName",
  "eng_sm6.gr");
Pipeline pipeline = new Pipeline(props);
Node annotation = pipeline.annotate(
  "This is a cat. That is a dog");

// Though the search methods such as \\ cannot be
// used on Java, we provide utilities to support
// Java programming.
List<Node> sentences = jigg.util.XMLUtil.findAllSub(
    annotation, "sentence");
Node firstSentence = sentences.get(0);
List<Node> tokens = jigg.util.XMLUtil.findAllSub(
    firstSentence, "token");
System.out.print("POS tags on the first sentence: ");
for (Node token: tokens) {
  String pos = XMLUtil.find(token, "@pos").toString();
  System.out.print(pos + " ");
}
```

Figure 5: Jigg also supports Java programming.

```
-annotators "corenlp[tokenize,ssplit],berkeleyparser,
  corenlp[lemma,ner,dcoref]"
```

where dcoref is a coreference resolution system relying on constituent parse trees (Recasens et al., 2013). This performs annotation of coreference resolution based on the parse trees given by the Berkeley parser instead of the Stanford parser. Using Jigg, a user can enjoy these combinations of existing tools quite intuitively. Also if a user has her own (higher-performance) system on the pipeline, one can replace the existing component with that in a minimal effort, by writing a wrapper of that tool in JVM languages (see Section 4).

3 Design

We now describe the internal mechanisms of Jigg, which comprise of two steps: the first is a check for correctness of the given pipeline, and the second is annotations on a raw text with the constructed pipeline. We describe the second annotation step first (Section 3.1), and then discuss the first pipeline check phase (Section 3.2).

3.1 Annotation on Scala XML

As shown in Figure 1, each annotator (e.g., the tokenizer in Stanford CoreNLP) communicates with the Scala XML object. Basically, each annotator only adds new elements or attributes into the received XML.[2] For example, the Berkeley parser receives an XML, on which each sentence element is annotated with tokens elements lacking pos attribute on each token. Then, the parser (i.e., the wrapper of the parser) adds the predicted syntactic tree and POS tags on each sentence XML (see Figure 3). Scala XML (Node object) is an immutable data structure, but it is implemented as an immutable tree, so a modification can be performed efficiently (in terms of memory and speed).

3.2 Requirement-based Pipeline Check

On this process, the essential point for the pipeline to correctly work is to guarantee that all the required annotations for an annotator are provided at each step. For example, the berkeleyparser annotator assumes each sentence element in the XML has the following structure:

```
<sentence id="...">
  sentence text
  <tokens>
    <token form="..." id="..."/>
    <token form="..." id="..."/>
    ...
  </tokens>
</sentence>
```

where form means the surface form of a token. How do we guarantee that the XML given to berkeleyparser satisfies this form?

Currently, Jigg manages these dependencies between annotators using the concept of Requirement, which we also borrowed from Stanford CoreNLP. Each annotator has a field called requires, which specifies the type of necessary annotations that must be given before running it. In berkeleyparser it is defined as follows:

```
override def requires:Set[Requirement] =
  Set(Tokenize, Ssplit)
```

where Ssplit is an object (of Requirement type), which guarantees that sentences element (a collection of sentence elements) exists on the current annotation, while Tokenize guarantees that each sentence element has tokens element (a collection of token elements), and each token has four attributes: id, form, characterOffsetBegin, and characterOffsetEnd.

Each annotator also has requirementsSatisfied field, which declares which Requirements will be satisfied (annotated). In the above requirements, Ssplit is given by corenlp[ssplit] while Tokenize is given by corenlp[tokenize]. In berkeleyparser, it is POS and Parse; POS guarantees that each token element has pos attribute. Before running annotation, Jigg checks whether the constructed pipeline correctly works by checking that all elements in requires for each annotator are satisfied by (included in) the requirementsSatisfied elements of the previous annotators. For example, if we run the pipeline with −annotators berkeleyparser argument, the program fails with an error message suggesting missing Requirements.

Note that currently Requirement is something just like a contract on the structure of annotated XML, and it is the author's responsibility to implement each annotator to output the correct XML structure. Currently the correspondence between each Requirement and the satisfied XML structure is managed with a documentation on the wiki of the project Github. We are seeking a more sophisticated (safe) mechanism to guarantee these correspondences in a code; one possible solution might be to define the skeletal XML structure for each Requirement, and test in each annotator whether the annotated object follows the defined structure.

4 Adding New Annotator

Here we describe how to implement a new annotator and integrate it into the Jigg pipeline. We also discuss a way to distribute a new system in Jigg.

Implementing new annotator We focus on implementation of Berkeley parser as an example to get intuition into what we should do. Annotator is the base trait[3] of all annotator classes, which defines the following basic methods:

- def annotate(annotation : Node) : Node
- def requires : Set[Requirement]
- def requirementsSatisfied : Set[Requirement]

We have already seen the roles of requires and requirementsSatisfied in Section 3.2. Note that in many cases including the Berkeley parser, annotation is performed on each *sentence* independently. For this type of annotation, we provide a useful trait SentenceAnnotator, which replaces the method to be implemented from annotate to newSentenceAnnotation, which has the same signature as annotate.[4]

[2]One exception in the current implementation is ssplit in corenlp, which breaks the result of tokenize (one very long tokenized sentence) into several sentences.

[3]Trait is similar to interface in Java.

[4]This trait implements annotate to traverse all sentences and replace them using newSentenceAnnotation method.

```
package jigg.pipeline
import ...

// By supporting a constructor with signature
// (String, Properties), the annotator can be
// instantiated dynamically using reflection.
class BerkeleyParserAnnotator(
  override val name: String,
  override val props: Properties) extends SentenceAnnotator {

  // Instantiate a parser by reading the gramar file.
  val parser: CoarseToFineMaxRuleParser = ...

  override def newSentenceAnnotation(sentence: Node): Node = {

    val tokens: Node = (sentence \ "tokens").head
    val tokenSeq: Seq[Node] = tokens \ "token"

    // (1) Get a list of surface forms.
    val formSeq: Seq[String] = tokenSeq.map(_ \@ "form")

    // (2) Parse the sentence by calling the API.
    val binaryTree: Tree[String] = parser.
      getBestConstrainedParse(formSeq.asJava, null, null)
    val tree =
      TreeAnnotations.unAnnotateTree(binaryTree, true)

    // (3) Convert the output tree into annotation.
    val taggedTokens = addPOSToTokens(tree, tokens)
    val parse = treeToNode(tree, tokenSeq)

    // (4) Return a new sentence node with updated
    // child elements.
    XMLUtil.addOrOverrideChild(
     sentence, Seq(newTokens, parseNode))
  }
  // Return the new tokens element on which each element has
  // pos attributes.
  def addPOSToTokens(tree: Tree[String], tokens: Node): Node
    = { ... }

  // Convert the Tree object in Berkeley parser into XML.
  def treeToNode(
    tree: Tree[String], tokenSeq: Seq[Node]): Node = { ... }

  override def requires = Set(Tokenize)
  override def requirementsSatisfied = Set(POS, Parse)
}
```

Figure 6: Core parts in BekeleyParserAnnotator.

Figure 6 shows an excerpt of essential parts in BerkeleyParserAnnotator. It creates a parser object in the constructor, and then in each newSentenceAnnotation, it first extracts a sequence of (yet annotated) tokens (1), gets a tree object from the parser (2), converts the tree into Scala XML object (3), and returns the updated sentence XML object (4). This workflow to encode to and decode from the API-specific objects is typical when implementing new annotators.

Calling with reflection The class in Figure 6 has a constructor with the signature (String, Properties), which allows us to instantiate the class dynamically using reflection. To do this, a user has to add a new property prefixed with customAnnotatorClass (the same as Stanford CoreNLP). In the case above, the property

customAnnotatorClass.berkeleyparser : jigg.pipeline.BerkeleyParser

makes it possible to load the implemented annotator with the name berkeleyparser.

Distributing new annotators An ultimate goal of Jigg is that the developers of a new tool in a pipeline distribute their system along with the wrapper (Jigg annotator) when releasing the software. If the system is JVM-based, the most stable way to integrate it is releasing the annotator (along with the software) into Maven repositories. Then, a user can build an extended Jigg by adding the dependency to it. For example, now the annotator for the MST parser is implemented, but is not included in Jigg, as it is a relatively old system. One way to extend Jigg with this tool is to prepare another project, on which its build.sbt may contain the following lines:[5]

```
libraryDependencies ++= Seq(
  "com.github.mynlp" % "jigg" % "VVV",
  "com.github.mynlp" % "jigg-mstparser" % "0.1-SNAPSHOT")
```

Jigg itself focuses more on the central NLP tools for wider users, but one can obtain the customized Jigg in this way.

Tools beyond JVM So far we have only dealt with JVM softwares such as Stanford CoreNLP, but Jigg can also wraps the softwares written in other languages such as C++ and python. In fact, many existing tools for Japanese are implemented in C or C++, and Jigg provides wrappers for those softwares. One problem of these languages is that installation is sometimes hard due to complex dependencies to other libraries. We thus put a priority on supporting the tool written in JVM languages in particular on Maven first, which can be safely incorporated in general.

5 Comparison to Stanford CoreNLP

As we have seen so far, Jigg follows the software design of Stanford CoreNLP in many respects. Finally, in this section, we highlight the important differences between two approaches.

Annotated objects Conceptually this is the most crucial difference as we mentioned in Section 1. In Stanford CoreNLP, each annotator manipulates an object called CoreMap. A clear advantage of this data structure is that one can take out a *typed* data structure, such as a well implemented Sentence or Graph object, which is easy

Another advantage of this trait is that annotations are automatically performed in parallel if the code is thread-safe. One can also prohibit this behavior by overriding nThreads variable by 1 in the annotator class.

[5]To call a new annotator, a user have to give a class path to the annotator with the property. Note that the mappings for the built-in annotators such as berkeleyparser are preserved in the Jigg package, so they can be used without any settings.

to use. In Jigg's XML, on the other hand, one accesses the fields through literals (e.g., \@ "pos" to get the POS attribute of a token). This may suggests Jigg needs more careful implementation for each annotator. However, we note that the problem can be alleviated by adding a simple unit test, which we argue is important as well in other platforms.

The main advantage of using Scala XML as a primary object is its flexibility for adapting to new types of annotations. It is just an XML object, so there is no restriction on the allowed structure. This is not the case in Stanford CoreNLP, where each element in CoreMap must be a proper data structure defined in the library, which means that the annotation that goes beyond the assumption of Stanford CoreNLP is difficult to support. Even if we define a new data structure in CoreMap, another problem occurs when outputting the annotation into other formats such as XML. In Stanford CoreNLP, this output component is hard-coded in the outputter class, which is difficult to extend. This is the problem that we encountered when we explored an extension to Stanford CoreNLP for Japanese processing pipeline as our initial attempt. Historically in Japanese NLP, the basic analyzing unit is called *bunsetsu*, which is a kind of chunk; a syntactic tree is often represented as a dependency tree on bunsetsu. Jigg is preferable to handle these new data structures, which go beyond the assumption on typical NLP focusing primarily on English, and we believe this flexibility make Jigg suitable for an integration framework, which has no restrictions on the applicable softwares and languages.

Output format Another small improvement is that our XML output format (Figure 3) is (we believe) more machine-friendly. For example, in Stanford CoreNLP, the parse element is just a Lisp-style tree like (S (NP (DT This)) ((VBZ is) (NP (DT a) (NN cat))) (. .)), which is parsable elements in Jigg. For some attribute names we employ different names, e.g., surface form is called form in Jigg instead of word in Stanford CoreNLP. We decide these names basically following the naming convention found in Universal Dependencies[6], which we expect becomes the standard in future NLP. Finally, now we implement each wrapper so that each id attribute is unique across the XML, which is not the case in Stanford CoreNLP. This makes search of elements more easier.

[6]http://universaldependencies.org/docs/

6 Conclusion

We presented Jigg, an open source framework for an easy natural language processing pipeline both for system developers and users. We hope that this platform facilitates distribution of a new high quality system on the pipeline to wider users.

Acknowledgments

This work was supported by CREST, JST.

References

David A. Ferrucci and Adam Lally. 2004. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Y. Kano, M. Miwa, K. B. Cohen, L. E. Hunter, S. Ananiadou, and J. Tsujii. 2011. U-compare: A modular nlp workflow construction and evaluation system. *IBM Journal of Research and Development*, 55(3):11:1–11:10, May.

Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. 2004. Applying conditional random fields to japanese morphological analysis. In Dekang Lin and Dekai Wu, editors, *EMNLP*, pages 230–237, Barcelona, Spain, July.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *ACL: System Demonstrations*, pages 55–60, Baltimore, Maryland, June.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT-EMNLP*, October.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: a data-driven parser-generator for dependency parsing. In *LREC*.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *HLT-NAACL*, pages 404–411, Rochester, New York, April.

Marta Recasens, Marie-Catherine de Marneffe, and Christopher Potts. 2013. The life and death of discourse entities: Identifying singleton mentions. In *NAACL: HLT*, pages 627–633, Atlanta, Georgia, June.

Richard Socher, John Bauer, Christopher D. Manning, and Ng Andrew Y. 2013. Parsing with compositional vector grammars. In *ACL*, pages 455–465, Sofia, Bulgaria, August.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL: HLT*, pages 173–180, Morristown, NJ, USA.

Sumire Uematsu, Takuya Matsuzaki, Hiroki Hanaoka, Yusuke Miyao, and Hideki Mima. 2013. Integrating multiple dependency corpora for inducing wide-coverage japanese ccg resources. In *ACL*, pages 1042–1051, Sofia, Bulgaria, August.