**How to use stanford core nlp**

1. Command line usage
   1. java -cp "\*" -Xmx2g edu.stanford.nlp.pipeline.StanfordCoreNLP -annotators tokenize,ssplit,pos,lemma,ner,parse,dcoref -file input.txt (add all jar files under core nlp directory which is also currently work directory, annotators will be specified in the command line as well as the relevant jar files).
   2. java -cp "/Users/me/corenlp/\*" -Xmx2g edu.stanford.nlp.pipeline.StanfordCoreNLP -file inputFile (Specify the path of nlp core and import all jar files, if annotators were not specified here, it will call the built in properties files).
   3. java -cp stanford-corenlp-VV.jar;stanford-corenlp-VV-models.jar:xom.jar;joda-time.jar;jollyday.jar:ejml-VV.jar -Xmx2g edu.stanford.nlp.pipeline.StanfordCoreNLP -file inputFile (import needed jar files individually and the extensions characters should be adjusted accordingly).
   4. java -cp "\*" -Xmx2g edu.stanford.nlp.pipeline.StanfordCoreNLP -props sampleProps.properties (we can place all the annotators and input, output files in properties file and specify -props in the command line. The format of prop file is annotators = tokenize, ssplit, pos \n outputExtension = .output \n file = input.txt).
   5. add other languages than English, download jar files of specified language and extract it to correct directory, then change pom.xml file like this:

<dependency>

<groupId>edu.stanford.nlp</groupId>

<artifactId>stanford-corenlp</artifactId>

<version>3.6.0</version>

<classifier>models-chinese</classifier>

</dependency>

* 1. the command line is (java -mx3g -cp "\*" edu.stanford.nlp.pipeline.StanfordCoreNLP -props StanfordCoreNLP-chinese.properties -file chinese.txt -outputFormat text) or (java -mx1g -cp "\*" edu.stanford.nlp.pipeline.StanfordCoreNLP -props StanfordCoreNLP-french.properties -annotators tokenize,ssplit,pos,depparse -file french.txt -outputFormat conllu).
  2. To process one file, use the -file option. To process a list of files, use the -filelist parameter: (java -cp "\*" -Xmx2g edu.stanford.nlp.pipeline.StanfordCoreNLP [ -props myprops.props ] -filelist filelist.txt). where the -filelist parameter points to a file whose content lists all files to be processed (one per line).
  3. The file ‘CoreNLP-to-HTML.xsl’ can display the output xml file in browser in reasonable format. And following are optional output properties.
  4. -outputDirectory : By default, output files are written to the current directory. You may specify an alternate output directory with the flag -outputDirectory.
  5. -outputExtension : Output filenames are the same as input filenames but with -outputExtension added to them (.xml by default).
  6. -noClobber : By default, files are overwritten (clobbered). Pass -noClobber to avoid this behavior.
  7. -replaceExtension : If you’d rather replace the extension with the -outputExtension, pass the -replaceExtension flag. This will result in filenames like input.xml instead of input.txt.xml (when given input.txt as an input file).
  8. -outputFormat : Different methods for outputting results. Can be:
     1. “text”: An ad hoc human-readable text format. Tokens, s-expression parse trees, relation(head, dep) dependencies. Output file extension is .out. This is the default output format only if the XMLOutputter is unavailable.
     2. “xml”: An XML format with accompanying XSLT stylesheet, which allows web browser rendering. Output file extension is .xml. This is the default output format, unless the XMLOutputter is unavailable.
     3. “json”: JSON. Output file extension is .json. ‘Nuf said.
     4. “conll”: A tab-separated values (TSV) format. Output extension is .conll. This representation may give only a partial view of an Annotation and doesn’t correspond to any particular CoNLL format. Columns are: wordIndex, token, lemma, POS, NER, head, depRel.
     5. “conllu”: CoNLL-U output format, another tab-separated values (TSV) format. Output extension is .conllu. This representation may give only a partial view of an Annotation.
     6. “serialized”: Produces some serialized version of each Annotation. May or may not be lossy. What you actually get depends on the outputSerializer property, which you should also set. The default is the GenericAnnotationSerializer, which uses the built-in Java object serialization and writes a file with extension .ser.gz.

1. Using the Stanford CoreNLP API
   1. Annotations are the data structure which hold the results of annotators. Annotations are basically maps, from keys to bits of the annotation, such as the parse, the part-of-speech tags, or named entity tags.
   2. Annotators are a lot like functions, except that they operate over Annotations instead of Objects. They do things like tokenize, parse, or NER tag sentences.
   3. Annotators and Annotations are integrated by AnnotationPipelines, which create sequences of generic Annotators. Stanford CoreNLP inherits from the AnnotationPipeline class, and is customized with NLP Annotators.
   4. To construct a Stanford CoreNLP object from a given set of properties, use StanfordCoreNLP(Properties props). This method creates the pipeline using the annotators given in the “annotators” property:

import edu.stanford.nlp.pipeline.\*;

import java.util.\*;

public class BasicPipelineExample {

public static void main(String[] args) {

// creates a StanfordCoreNLP object, with POS tagging, lemmatization, NER, parsing, and coreference resolution

Properties props = new Properties();

props.setProperty("annotators", "tokenize, ssplit, pos, lemma, ner, parse, dcoref");

StanfordCoreNLP pipeline = new StanfordCoreNLP(props);

// read some text in the text variable

String text = "...";

// create an empty Annotation just with the given text

Annotation document = new Annotation(text);

// run all Annotators on this text

pipeline.annotate(document);

}

}

-------------------------------------------------------------------------------------------------------------------

StanfordCoreNLP pipeline = new StanfordCoreNLP(

PropertiesUtils.asProperties(

"annotators", "tokenize,ssplit,pos,lemma,parse,natlog",

"ssplit.isOneSentence", "true",

"parse.model", "edu/stanford/nlp/models/srparser/englishSR.ser.gz",

"tokenize.language", "en"));

// read some text in the text variable

String text = ... // Add your text here!

Annotation document = new Annotation(text);

// run all Annotators on this text

pipeline.annotate(document);

* 1. The output of the Annotators is accessed using the data structures CoreMap and CoreLabel.

// these are all the sentences in this document

// a CoreMap is essentially a Map that uses class objects as keys and has values with custom types

List<CoreMap> sentences = document.get(SentencesAnnotation.class);

for(CoreMap sentence: sentences) {

// traversing the words in the current sentence

// a CoreLabel is a CoreMap with additional token-specific methods

for (CoreLabel token: sentence.get(TokensAnnotation.class)) {

// this is the text of the token

String word = token.get(TextAnnotation.class);

// this is the POS tag of the token

String pos = token.get(PartOfSpeechAnnotation.class);

// this is the NER label of the token

String ne = token.get(NamedEntityTagAnnotation.class);

}

// this is the parse tree of the current sentence

Tree tree = sentence.get(TreeAnnotation.class);

// this is the Stanford dependency graph of the current sentence

SemanticGraph dependencies = sentence.get(CollapsedCCProcessedDependenciesAnnotation.class);

}

// This is the coreference link graph

// Each chain stores a set of mentions that link to each other,

// along with a method for getting the most representative mention

// Both sentence and token offsets start at 1!

Map<Integer, CorefChain> graph =

document.get(CorefChainAnnotation.class);

1. Using the simple CoreNLP API
   1. A constructor is provided for both the Document and Sentence class. For the former, the text is treated as an entire document containing potentially multiple sentences. For the latter, the text is forced to be interpreted as a single sentence.

import edu.stanford.nlp.simple.\*;

public class SimpleCoreNLPDemo {

public static void main(String[] args) {

// Create a document. No computation is done yet.

Document doc = new Document("add your text here! It can contain multiple sentences.");

for (Sentence sent : doc.sentences()) { // Will iterate over two sentences

// We're only asking for words -- no need to load any models yet

System.out.println("The second word of the sentence '" + sent + "' is " + sent.word(1));

// When we ask for the lemma, it will load and run the part of speech tagger

System.out.println("The third lemma of the sentence '" + sent + "' is " + sent.lemma(2));

// When we ask for the parse, it will load and run the parser

System.out.println("The parse of the sentence '" + sent + "' is " + sent.parse());

// ...

}

}

}

* 1. The interface is not guaranteed to support all of the annotators in the CoreNLP pipeline. However, most common annotators are supported.

FUNCTIONALITY | ANNOTATOR IN CORENLP | IMPLEMENTING CLASS | FUNCTION

Tokenization tokenize Sentence .words() / .word(int)

Sentence Splitting ssplit Document .sentences() / .sentence(int)

Part of Speech Tagging pos Sentence .posTags() / .posTag(int)

Lemmatization lemma Sentence .lemmas() / .lemma(int)

Named Entity Recognition ner Sentence .nerTags() / .nerTag(int)

Constituency Parsing parse Sentence .parse()

Dependency Parsing depparse Sentence .governor(int) / .incomingDependencyLabel(int)

Coreference Resolution dcoref Document .coref()

Natural Logic Polarity natlog Sentence .natlogPolarities() / natlogPolarity(int)

Open Information Extraction openie Sentence .openie() / .openieTriples()

1. REPL
   1. Setting the classpath
   2. Run the XXX-gui.bat under CoreNLP-master\doc