31 Case Studies: Java Natural Language Tools Available on the Web

Chapter Objectives

This chapter provides a number of sources for open source and free atural language understanding software on the web.

Chapter Contents

31.1 Java NLP Software
31.2 LingPipe from the University of Pennsylvania
31.3 The Stanford Natural Language Processing Group Software
31.4 Sun's Speech API

31.1 Java Natural Language Processing Software

There are several popular java-based open-source natural language understanding software packages available on the internet. We describe three of these in Chapter 31. It must be remembered both that these web sites may change over time and that new sites will appear focusing on problems in NLP.

31.2 LingPipe from the University of Pennsylvania

Background

LingPipe is an available Java resource from Alias-I, http://www.alias-i.com/lingpipe/. Alias-i began in 1995 as a collaboration of students at University of Pennsylvania. After competing in different events (including DARPA MUC-6), the group was awarded a research contract under the TIDES (Trans-Lingual Information Detection Extraction and Summarization) program. Starting as Baldwin Language Technologies, the company's name later changed to Alias-i. LingPipe was used in two of Alias-i's products, FactTracker and ThreatTracker. In 2003, LingPipe was released as open source software with commercial licenses available as well. LingPipe contains many tools for linguistic analysis of human language, including tools for sentence-boundary detection; and a part-of-speech tagger and phrase chunker.

LingPipe is easy to download from the website. The download contains demos, documentation and there are models available to download as well. On the website there are tutorials, documentation, and a FAQ. Also there are links to the community of LingPipe consumers. This includes a listing of some commercial customers, as well as research patrons. There is a newsgroup for discussion as well as a blog for being kept up to date on the suite.

We next take a look at some of the tools provided by LingPipe.

Sentence-boundary Detection

To start with, there are tutorials contained in the download for the sentence-boundary detection classes. These tutorials contain example programs that use and extend the *com.aliasi.sentences* classes. If you follow

the tutorials, you will get suggestions for how some of the classes can be extended to do sentence detection for other corpora.

The *AbstractSentenceModel* class contains the basic functionality needed to detect sentences. When extending this class, definitions of possible stops, impossible penultimates, and impossible starts are needed. Possible stops are any token that can be placed at the end of a sentence. This includes '.' and '?'. Impossible penultimates are tokents that cannot precede an end of the sentence. An example would be 'Mr' or 'Dr'. Impossible starts are normally punctuation that should not start a sentence and should be associated with the end of the last sentence. These can be things like end quotes.

The *AbstractSentenceModel* is already extended to the *HeuristicSentenceModel*, which is extended to the *IndoEuropeanSentenceModel*, and the *MedlineSentenceModel*. These last two provide good examples of definitions for the possible stops, impossible penultimates and the impossible starts. From these examples, *HeuristicSentenceModel* can be extended to suit a data set. After creating an example set with known sentence boundaries, running the evaluator contained in the tutorials for the sentences class gives an idea of fallacies of the current model. From the evaluator's output files, corrections can be made to the possible stops, impossible penultimates and impossible starts. Be careful though; when attempting to remove all false positives and all false negatives, the definitions can be come too rigid and cause more errors when run with more then the example set. So try to find a good balance.

Within the download, the *AbstractSentenceModel* is only extended to the *HeuristicSentenceModel*. This does not mean that you must use the *HeuristicSentenceModel*. The *HeuristicSentenceModel* can be used as an example to create a new class that extends only *AbstractSentenceModel*. Therefore if you have a different type of model that you would like to use, extend *AbstractSentenceModel* and try it out.

Part-of-speech Tagger

The part-of-speech (POS) tagger is a little more complicated then the *sentences* classes. To use it the POS tagger must be trained. After it is trained, the tagger can be used to produce a couple of different statistics about its confidence of the tags it applies to input. In the download there are examples of code for the Brown, Genia and MedPost corpora. The classes used in making a POS tagger come from the *com.aliasi.hmm* package. The tagger is a *HmmDecoder* defined by a *HiddenMarkovModel*.

To train a tagger, you need first a corpus or test set that has been tagged. Using this tagged set, the *HmmCharLmEstimator* (in the *com.aliasi.hmm* package) can read the training set and create a *HiddenMarkovModel*. This model can be used immediately, or it can be written out to file and used at a later time. The file can be useful when evaluating different taggers. For each test on different corpora, the exact tagger can be used without having to recreate it each time.

Now that we have a tagger, we can use it to tag input. Within one *HmmDecoder* there are a couple of different ways to tag; all are methods of the decoder you create from the *HiddenMarkovModel*. Based on what kind of information you need, the options are first-best, n-best and confidence-

based. First-best returns only the "best" tagging of the input. N-best returns the first n "best" taggings. Confidence-based results are the entire lattice of forward/backward scores.

Provided in the tutorials is an evaluator of taggers. This uses pre-tagged corpora and trains a little, then evaluates a little. It parses reference taggings, uses the model to tag, and evaluates how well the model did. The reference tags are then added to the training data, and the parser moves on. The arguments to the evaluator will determine how well the model learns and how long it will take. Experiment with this package to see what is appropriate for your own data set.

A tagger produced by this package could be used in other algorithms. Whether as tags needed for the algorithm or as a source to produce a grammar, this POS tagger is useful. A future project might be to create a parse tree from the POS tagger, but that functionality is not within LingPipe. An exercise might be to extend LingPipe to create parse trees.

31.3 The Stanford Natural Language Processing Group

The Stanford NLP group is a team of faculty, postdoctoral researchers, graduate, and undergraduate students, members from both the Computer Science and Linguistics departments. The site http://nlp.stanford.edu describes the team members, their publications, and the libraries that can be downloaded.

Exploring this Stanford website, the reader finds, as of January 2008, six Java libraries available for work in natural language processing. These include a parser as well as a part-of-speech tagger. Although the information contained in the introduction for each package is extensive and contains a set of "frequently asked questions", the code documentation is often sparse without sufficient design documentation. The libraries are licensed under the full GNU Public License, which means that they are available for research or free software projects.

The Stanford Parser

The parser makes up a major component of the Stanford NLP website. There is background information for the parser, an on-line demo, and answers for frequently asked questions. The Stanford group refers to their parser as "a program that works out the grammatical structure of sentences". The software package includes an optimized probabilistic context-free grammar (Luger 2009, Section 15.4).

Within the download of the Stanford parser is a package called edu.stanford.nlp.parser. This parser interface contains two functions: One function determines whether the input sentence can be parsed or not. The other function determines whether the parse meets particular parsing goals, for example, that it is a noun phrase followed by a verb phrase sentence. There are also a number of sub-interfaces provided, including *ViterbiParser*, see Chapter 30, and *KBestViterbiParser*, the first supporting the most likely probabilistic parse of the sentence and the second giving the *K* best parses, where all parses are returned with their "scores".

Within the interface *edu.stanford.nlp.parser* there is a further interface, edu.stanford.nlp.parser.lexparser, which supports parsers for English,

German, Chinese, and Arabic expressions. There are also classes, that once implemented, can be used to train the parser for use on other languages. To train the parser, a training set needs to include systematically annotated data, specifically in the form of a Treebank (a corpus of annotated data that explicitly specifies parse trees). Once trained the parser contains three components: grammars, a lexicon, and a set of option values. The grammar itself consists of two parts, unary (NP = N) and binary (S = NP VP) rewrite rules. The lexicon is a list of lexical (word) entries) preceeded by a keyword followed by a raw count score. The options are persistent variable-value pairs that remain constant across the training and parsing stages.

The Stanford tools also include a GUI for easy visualization of the trees produced through parsing the input streams. The training stages require much more time and memory than using the already trained parser. Thus, for experimental purposes, it is convenient to use the already trained parsers, although there is much that can be learned by stepping through the creation of a parser.

Named-Entity Recognition

A program that performs *named–entity recognition* (NER) locates and classifies elements of input strings (text) into predefined categories. For the Stanford NER the primary categories for classification are *name*, *organization*, *location*, and *miscellaneous*. There are two recognizers, the first classifying the first three of these categories and trained on a corpus created from data from conference proceedings. The second is trained on more specific data, the proceedings from one conference.

Using the text classifiers is straightforward. They can be run either as embedded in a larger program or by command line. When run as part of a program the classifier is read in using a function associated with *CRFClassifier*. This function returns an *AbstractSequenceClassifier* that uses methods to classify the contents of a *String*. An example of one (of the three possible) output formats, called /Cat is: My/O name/O is/O Bill/PERSON Smith/PERSON ./O. /O indicates that the text string is not recognized as a named-entity. There are a number of issues involved in this type classification, for example that at this point Bill Smith is not recognized as the name of a single person but rather as two consecutive PERSON tokens. When working with this type pattern matching it is important to monitor issues in *over-learning* and *under-learning*: when one pattern matching component is created to fit a complex problem situation, another set of patterns may not then be classified properly.

Unfortunately the documentation for the Named-Entity package is minimal. Although it contains a set of JavaDocs they can be both wrong (referring to classes that are not included) or simply unhelpful.

31.4 Sun's Speech API

To this point we have focused on Java-based natural language processing tools analyzing written language. Speech recognition and synthesis are also important components of NLP. To assist developers in these areas Sun Microsystems has created an API for speech. This Java API can be found at http://research.sun.com/speech. From this page there is also a link to a

free speech recognizer developed using this API at Carnegie Mellon University, as well as a speech written by Sun that is based on *Flite*, a speech synthesis engine also developed at Carnegie Mellon University. To run programs written in the Java Speech API needs a compliant speech recognizer and synthesizer, audio hardware for output and a microphone for input.

The API contains three packages: speech, speech recognition, and speech synthesis. The speech package contains several packages and interfaces used by both the recognition and synthesis systems. The main interface is an *Engine* that is the parent interface for all speech systems. The engine contains the procedures for communicating with other classes as well as allocation/deallocation methods for moving between states. These states determine whether the engine has acquired resources sufficient for executing a function. The engine also provides methods to pause and resume play and to access all properties including, listeners, audio, and vocabulary managers. The speech class also contains procedures for listeners as well as a Word class that contains the written and spoken pronunciation forms for words. The collection of words, the vocabulary, is controlled by the *Vocabulary* manager.

The main class of the speech recognizer is *Recognizer*. An instance of recognizer creates listener events and passes them to all registered event listeners much the same way as action listeners work for GUI applications. The events are either accepted or rejected based on sets of grammar rules. There are two forms of grammar rules: rule-based and dictation. Dictation rules offer fewer constraints on what can be said with a resulting higher cost in computational resources an often lesser quality results. Rule-based grammars are constrained to the Java speech grammar format (JSGF) and as a result impose a greater constraint on the recognizer. They also require fewer resources with a reasonable freedom for expressions. A tutorial for the JSGF is located at http://java.sun.com/products/java-media/speech/forDevelopers/JSGF and can be used to create grammars.

Once a grammar is created it is passed to a recognizer and activated. Then the recognizer begins processing and sending out events to all registered listeners. Sample applications are linked to the previously mentioned web site.

The Java speech API also contains a package for synthesis. Analogous to the recognizer, the synthesizer package contains a *Synthesis* class. The synthesizer is able to speak instances of the *Speakable* class in a voice constructed by an instance of the *Voice* class. This class contains both make and female, as well as young, middle-aged, and older voices. It's only task is to output a "speakable" text, as defined by a Java speech markup language (JSML) specification. These specifications can be found at http://java.sun.com/products/java-media/speech/forDevelopers/JSML/.

Demonstrations of the Sun speech synthesizer are available at http://fretts.sourceforge.net/docs/index.php. where the source code can be downloaded. The Sun speech API comes with a wealth of documentation and example source code. Which is fairly transparent and easy to follow.

There are a number of other web-based sources that support tasks in natural language text and speech understanding. These range from phoneme capture, the development of word and language models using probabilistic finite-state acceptors and various forms of Markov models. There are also parsers and recognizers of sentence structures as well as more examples of speech recognizers and synthesizers. There are also a number of tools available for speech to text conversion. Besides tools in Java, many also exist in other languages including C++ and Python.