NLTK: THE NATURAL LANGUAGE TOOLKIT

Sruthi Paramkusham
Department of Computer Science and Software Engineering,
Jawaharlal Nehru Technological University, Hyderabad India

Abstract: The Natural Language Toolkit is a suite of program modules, data sets and tutorials supporting research and teaching in computational linguistics and natural language processing. NLTK is written in Python and distributed under the GPL open source license. Over the past year the toolkit has been rewritten, simplifying many linguistic data structures and taking advantage of recent enhancements in the Python language. This paper reports on the simplified toolkit and explains how it is used in teaching NLP.

I. INTRODUCTION

NLTK, the Natural Language Toolkit, is a suite of Python modules providing many NLP data types, processing tasks, corpus samples and readers, together with animated algorithms, tutorials, and problem sets (Loper and Bird, 2002). Data types include tokens, tags, chunks, trees, and feature structures. Interface definitions and reference implementations are provided for tokenizers, stemmers, taggers (regexp, ngram, Brill), chunkers, parsers (recursive-descent, shift-reduce, chart, probabilistic), clusterers, and classifiers. Corpus samples and readers include: Brown Corpus, CoNLL-2000 Chunking Corpus, CMU Pronunciation Dictionary, NIST IEER Corpus, PP Attachment Corpus, Penn Treebank, and the SIL Shoebox corpus format. NLTK is ideally suited to students who are learning NLP or conducting research in NLP or closely related areas. NLTK has been used successfully as a teaching tool, as an individual study tool, and as a platform for prototyping and building research systems (Liddy and McCracken, 2005; Sætre et al., 2005). We chose Python for its shallow learning curve, transparent syntax, and good string-handling. Python permits exploration via its interactive interpreter. As an object-oriented language, Python permits data and code to be encapsulated and re-used easily. Python comes with an extensive library, including tools for graphical programming and numerical processing (Beasley, 2006). Over the past four years the toolkit grew rapidly and the data structures became significantly more complex. Each new processing task added new requirements on input and output representations. It was not clear how to generalize tasks so they could be applied independently of each other. As a simple example, consider the independent tasks of tagging and stemming, which both operate on sequences of tokens. If stemming is done first, we lose information required for tagging. If tagging is done first, the stemming must be able to skip over the tags. If both are done independently, we need to be able to align the results. As task combinations multiply, managing the data becomes extremely difficult. To address this problem, NLTK 1.4 introduced a blackboard architecture for tokens, unifying many data types, and permitting distinct tasks to be run independently. Unfortunately this architecture also came with a significant overhead for programmers, who were often forced to use “rather awkward code structures” (Hearst, 2005). It was clear that the re-engineering done in NLTK 1.4 unduly complicated the programmer’s task. This paper presents a brief overview and tutorial on a new, simplified toolkit, and describes how it is used in teaching.

II. SIMPLE PROCESSING TASKS

A. Tokenization and Stemming

The following three-line program imports the tokenize package, defines a text string, and tokenizes the string on whitespace to create a list of tokens. (NB. ‘>>>’ is Python’s interactive prompt; ‘...’ is the continuation prompt.)

```python
text = 'This is a test.'
list(tokenize.whitespace(text))
```

Several other tokenizers are provided. We can stem the output of tokenization using the Porter Stemmer as follows:

```python
my_tokenizer = tokenize.WhitespaceTokenizer()
my_stemmer = stemmer.PorterStemmer()
for token in tokens:
    my_tokenizer.stem(token)
    my_stemmer.stem(token)
```

B. Tagging

The simplest possible tagger assigns the same tag to each token:

```python
my_tagger = tag.Default('nn')
list(my_tagger.tag(tokens))
```

The regular expression tagger assigns a tag to a token according to a series of string patterns. For instance, the following tagger assigns cd to cardinal numbers, nns to...
words ending in the letter s, and nn to everything else:
>>> patterns = [
... (r'\d+(\d+)?S', 'cd'),
... (r'\w+S', 'nnm'),
... (r'.*s', 'nm')
]
simple_tagger = tag.Regexp(patterns)
list(simple_tagger.tag(tokens)) [('John', 'mn'), ('saw', 'mn'), ('3', 'cd'), ('polar', 'mn'), ('bears', 'nm'), ('', 'nm')]
The tag.Unigram class implements a sim-ple statistical tagging algorithm: for each token, it assigns the tag that is most likely for that token. For example, it will assign the tag jj to any occur-rence of the word frequent, since frequent is used as an adjective (e.g. a frequent word) more often than it is used as a verb (e.g. I frequent this cafe). Before a unigram tagger can be used, it must be trained on a corpus, as shown below for the first section of the Brown Corpus.

Once a unigram tagger has been trained, it can be used to tag new text. Note that it assigns the default tag None to any token that was not encountered during training.

We can instruct the unigram tagger to back off to our default simple_tagger when it cannot assign a tag itself. Now all the words are guaran-teed to be tagged:
>>> grammar =
... (r'\d+(\d+)?S', 'cd'),
... (r'\w+S', 'nnm'),
... (r'.*s', 'nm')
... simple_tagger = tag.Regexp(patterns)
list(simple_tagger.tag(tokens)) [('John', 'mn'), ('saw', 'mn'), ('3', 'cd'), ('polar', 'mn'), ('bears', 'nm'), ('', 'nm')]

We can go on to define and train a bigram tagger, as shown below:
>>> grammar =
... (r'\d+(\d+)?S', 'cd'),
... (r'\w+S', 'nnm'),
... (r'.*s', 'nm')
... simple_tagger = tag.Regexp(patterns)

We can easily evaluate this tagger against some gold-standard tagged text, using the tag.accuracy() function. NLTK also includes a Brill tagger (contributed by Christopher Maloof) and an HMM tagger (con-tibuted by Trevor Cohn).

III. CHUNKING AND PARSGING

Chunking is a technique for shallow syntactic analysis of (tagged) text. Chunk data can be loaded from files that use the common bracket or IOB notations. NLTK also supports simple cascading of chunk parsers. Corpus readers for chunked data in Penn Treebank and CoNLL-2000 are provided, along with comprehensive support for evaluation and error analysis. NLTK provides several parsers for context-free phrase-structure grammars. Grammars can be defined using a series of productions as follows:

Now we can tokenize and parse a sentence with a recursive descent parser. Note that we avoided left-recursive productions in the above grammar, so that this parser does not get into an infinite loop.

>>> text = "John saw a cat with my ball"
>>> sent = list(tokenize.whitespace(text))
>>> rd = parse.RecursiveDescent(grammar)

The same sentence can be parsed using a grammar with left-recursive productions, so long as we use a chart parser. We can invoke NLTK's chart parser with a bottom-up rule-invocation strategy with chart.ChartParse(grammar, chart.BU STRATEGY). Tracing can be turned on in order to display each step of the process. NLTK also supports probabilistic context free grammars, and provides a Viterbi-style PCFG parser, together with a suite of bottom-up probabilistic chart parsers.

IV. Teaching with NLTK

Natural language processing is often taught within the confines of a single-semester course, either at advanced undergraduate level or at postgradu-ate level. Unfortunately, it turns out to be rather difficult to cover both the theoretical and practi-cal sides of the subject in such a short span of time. Some courses focus on theory to the exclu-sion of practical exercises, and deprive students of the challenge and excitement of writing programs to automatically process natural language. Other courses are simply designed to teach programming for linguists, and do not manage to cover any sig-nificant NLP content. NLTK was developed to address this problem, making it feasible to cover a substantial amount of theory and practice within a single-semester course. A significant fraction of any NLP course is made up of fundamental data structures and algorithms. These are usually taught with the help of formal notations and complex
diagrams. Large trees and charts are copied onto the board and edited in tedious slow motion, or laboriously prepared for presentation slides. A more effective method is to use live demonstrations in which those diagrams are generated and updated automatically. NLTK provides interactive graphical user interfaces, making it possible to view program state and to study program execution step-by-step (e.g. see Figure 1). Most NLTK components have a demonstration mode, and will perform an interesting task without requiring any special input from the user. It is even possible to make minor modifications to programs in response to “what if” questions. In this way, students learn the mechanics of NLP quickly, gain deeper insights into the data structures and algorithms, and acquire new problem-solving skills. Since these demonstrations are distributed with the toolkit, students can experiment on their own with the algorithms that they have seen presented in class.

![Figure 1: Two Parser Demonstrations: Shift-Reduce and Recursive Descent Parsers](image)

NLTK can be used to create student assignments of varying difficulty and scope. In the simplest assignments, students experiment with one of the existing modules. Once students become more familiar with the toolkit, they can be asked to make minor changes or extensions to an existing module (e.g. build a left-corner parser by modifying the recursive descent parser). A bigger challenge is to develop one or more new modules and integrate them with existing modules to perform a sophisticated NLP task. Here, NLTK provides a useful starting point with its existing components and its extensive tutorials and API documentation. NLTK is a unique framework for teaching natural language processing. NLTK provides comprehensive support for a first course in NLP which tightly couples theory and practice. Its extensive documentation maximizes the potential for independent learning. For more information, including documentation, download pointers, and links to dozens of courses that have adopted NLTK, please see: http://nltk.sourceforge.net/.

Acknowledgements
I am grateful to Mr. Siddarth Ghosh, HOD of Computer Science Engineering Dept who guided me in writing paper.

REFERENCES


