NLTK-Lite: Efficient Scripting for Natural Language Processing

Steven Bird
Department of Computer Science and Software Engineering
University of Melbourne, Victoria 3010, AUSTRALIA
Linguistic Data Consortium, University of Pennsylvania,
Philadelphia PA 19104-2653, USA

Abstract

The Natural Language Toolkit is a suite of program modules, data sets, tutorials and exercises covering symbolic and statistical natural language processing. NLTK is popular in teaching and research, and has been adopted in dozens of NLP courses. NLTK is written in Python and distributed under the GPL open source license. Over the past year the toolkit has been completely rewritten, simplifying many linguistic data structures and taking advantage of recent enhancements in the Python language. This paper reports on the resulting, simplified toolkit, NLTK-Lite, and shows how it is used to support efficient scripting for natural language processing.

1 Introduction

NLTK, the Natural Language Toolkit, is a suite of Python libraries and programs for symbolic and statistical natural language processing (Loper and Bird, 2002; Loper, 2004). NLTK includes graphical demonstrations and sample data. It is accompanied by extensive documentation, including tutorials that explain the underlying concepts behind the language processing tasks supported by the toolkit.

NLTK is ideally suited to students who are learning NLP (natural language processing) or conducting research in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning. 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 because it has a shallow learning curve, its syntax and semantics are transparent, and it has good string-handling functionality. As an interpreted language, Python facilitates interactive exploration. As an object-oriented language, Python permits data and methods to be encapsulated and re-used easily. Python comes with an extensive standard library, including tools for graphical programming and numerical processing (Rossum, 2003a; Rossum, 2003b).

Over the past four years the toolkit grew rapidly and the data structures became significantly more complex. Each new processing task brought with it 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 new architecture for tokens based on Python's native dictionary data type. Tokens could have an arbitrary number of named properties, like `TAG`,

and STEM. Whole sentences, and even whole documents, were represented as single tokens having a SUBTOKENS attribute to hold sequences of smaller tokens. Parse trees were likewise tokens, with a special CHILDREN property. The advantage of this token architecture was that it unified many different data types, and permitted distinct tasks to be run independently. Unfortunately this architecture also came with a significant overhead for programmers, who had to keep track of a growing set of property names, and who were often forced to use "rather awkward code structures" (Hearst, 2005). It was clear that the re-engineering done in NLTK 1.4 mainly got in the way of efficient authoring of NLP scripts.

This paper presents a new, simplified toolkit called NLTK-Lite. This paper presents a brief overview and tutorial on NLTK-Lite, and identifies some areas where more contributions would be welcome.

2 Overview of NLTK-Lite

NLTK-Lite is a suite of Python packages providing a range of standard NLP data types, interface definitions and processing tasks, corpus samples and readers, together with animated algorithms, extensive tutorials, and problem sets. 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). Corpus samples and readers include: Brown Corpus, CoNLL-2000 Chunking Corpus, CMU pronunciation dictionary, NIST Information Extraction and Entity Recognition Corpus, Ratnaparkhi's Prepositional Phrase Attachment Corpus, Penn Treebank, and the SIL Shoebox corpus format.

NLTK-Lite differs from NLTK in the following key respects: fundamental representations are kept as simple as possible (e.g. strings, tuples, trees); all streaming tasks are implemented as iterators instead of lists in order to limit memory usage and to ensure that data-intensive tasks produce output as early as possible; the default pipeline processing paradigm leads to more transparent code; taggers incorporate backoff leading to much smaller models and faster operation; method names are shorter (e.g. tokenizer.RegexpTokenizer becomes tokenize.regexp, and the barrier to entry for contributed software is removed now that there is no requirement to support the special NLTK token architecture.

3 Simple Processing Tasks

In this section we review some simple NLP processing tasks, and show how they are performed in NLTK-Lite.

3.1 Tokenization and Stemming

The following three-line program imports the tokenize package, defines a text string, and then tokenizes the string on whitespace to create a list of tokens. (Note that ">>>" is Python's interactive prompt; "..." is the second-level prompt.)

```
>>> from nltk_lite import tokenize
>>> text = 'Hello world.  This is a test.'
>>> list(tokenize.whitespace(text))
['Hello', 'world.', 'This', 'is', 'a', 'test']
```

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

```
>>> text = 'stemming can be fun and exciting'
>>> tokens = tokenize.whitespace(text)
>>> porter = tokenize.PorterStemmer()
>>> for token in tokens:
...     print porter.stem(token),
stem can be fun and excit
```

The corpora included with NLTK-Lite come supplied with corpus readers that understand the file structure of the corpus, and load the data into Python data structures. For example, the following code reads the first sentence of part a of the Brown Corpus. It prints a list of tuples, where each tuple consists of a word and its tag.

```
>>> from nltk_lite.corpora\
...     import brown, extract
>>> print extract(0, brown.tagged('a'))
[('The', 'at'), ('Fulton', 'np-tl'),
('County', 'nn-tl'), ('Grand', 'jj-tl'),
('Jury', 'nn-tl'), ('said', 'vbd'), ...]
```

NLTK-Lite provides support for conditional frequency distributions, making it easy to count up items of interest in specified contexts. The code sample and output in Figure 1 counts the usage of modal verbs in the Brown Corpus, displaying them

```
>>> cfdist = ConditionalFreqDist()
>>> for genre in brown.items():                  # each genre
...     for sent in brown.tagged(genre):        # each sentence
...         for (word,tag) in sent:             # each tagged token
...             if tag == 'md':                 # found a modal
...                 cfdist[genre].inc(word.lower())
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> print "%-40s" % 'Genre', ' '.join([("%6s" % m) for m in modals])
>>> for genre in cfdist.conditions():    # generate rows
...     print "%-40s" % brown.item_name[genre],
...     for modal in modals:
...         print "%6d" % cfdist[genre].count(modal),
...     print
Genre                                     can  could    may  might   must   will
press: reportage                           94     86     66     36     50    387
press: reviews                             44     40     45     26     18     56
press: editorial                          122     56     74     37     53    225
skill and hobbies                         273     59    130     22     83    259
religion                                   84     59     79     12     54     64
belles-lettres                            249    216    213    113    169    222
popular lore                              168    142    165     45     95    163
miscellaneous: government & house organs  115     37    152     13     99    237
fiction: general                           39    168      8     42     55     50
learned                                   366    159    325    126    202    330
fiction: science                           16     49      4     12      8     16
fiction: mystery                           44    145     13     57     31     17
fiction: adventure                         48    154      6     58     27     48
fiction: romance                           79    195     11     51     46     43
humor                                      17     33      8      8      9     13
```

Figure 1: Program to Generate a Table of Modals and their Frequency of Use in Different Genres

by genre. Such information may be useful for studies in stylistics, and also in text categorization.

3.2 Tagging

The simplest possible tagger assigns the same tag to each token regardless of the token's text. The DefaultTagger class implements this kind of tagger. In the following program, we create a tagger called `my_tagger` which tags everything as a noun.

```
>>> from nltk_lite import tag
>>> my_tagger = tag.Default('nn')
>>> list(my_tagger.tag(tokens))
[('John', 'nn'), ('saw', 'nn'),
 ('3', 'nn'), ('polar', 'nn'),
 ('bears', 'nn'), ('.', 'nn')]
```

This is a simple algorithm, and it performs poorly when used on its own. On a typical corpus, it will tag only 20–30% of the tokens correctly. However, it is a very reasonable tagger to use as a default, if a more advanced tagger fails to determine a token's tag.

The regular expression tagger assigns tags to tokens on the basis of matching patterns in the token's text. For instance, the following tagger assigns `cd` to cardinal numbers, and `nn` to everything else:

```
>>> patterns = [
...     (r'^[0-9]+(.[0-9]+)?$', 'cd'),
...     (r'.*', 'nn')]
>>> nn_cd_tagger = tag.Regexp(patterns)
>>> list(nn_cd_tagger.tag(tokens))
[('John', 'nn'), ('saw', 'nn'),
 ('3', 'cd'), ('polar', 'nn'),
 ('bears', 'nn'), ('.', 'nn')]
```

The `UnigramTagger` class implements a simple statistical tagging algorithm: for each token, it assigns the tag that is most likely for that token's text. For example, it will assign the tag `jj` to any occurrence 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.

```
>>> from nltk_lite.corpora import brown
>>> unigram_tagger = tag.Unigram()
>>> unigram_tagger.train(brown('a'))
```

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.

```
>>> text = "John saw the book on the table"
>>> tokens = list(tokenize.whitespace(text))
>>> list(unigram_tagger.tag(tokens))
[('John', 'np'), ('saw', 'vbd'),
 ('the', 'at'), ('book', None),
 ('on', 'in'), ('the', 'at'),
 ('table', None)]
```

We can instruct the unigram tagger to back off to our default `nn_cd_tagger` when it cannot assign a tag itself. Now all the words are guaranteed to be tagged:

```
>>> unigram_tagger =
...     tag.Unigram(backoff=nn_cd_tagger)
>>> unigram_tagger.train(train_sents)
>>> list(unigram_tagger.tag(tokens))
[('John', 'np'), ('saw', 'vbd'),
 ('the', 'at'), ('book', 'nn'),
 ('on', 'in'), ('the', 'at'),
 ('table', 'nn')]
```

We can go on to define and train a bigram tagger, as shown below:

```
>>> bigram_tagger =\
...     tag.Bigram(backoff=unigram_tagger)
>>> bigram_tagger.train(brown.tagged('a'))
```

We can easily evaluate this tagger against some gold-standard tagged text, using the `tag.accuracy()` function.

NLTK-Lite also includes a Brill tagger (contributed by Christopher Maloof) and an HMM tagger (contributed by Trevor Cohn).

4 Chunking and Parsing

4.1 Chunking

Chunking is a technique for shallow syntactic analysis of (tagged) text. Chunk data can be loaded from files that use the bracket notation (e.g. Penn Treebank) or the IOB (INSIDE/OUTSIDE/BEGIN) notation (e.g. CoNLL-2000).

```
>>> from nltk_lite import parse
>>> text = """
... he PRP B-NP
... accepted VBD B-VP
... the DT B-NP
... position NN I-NP
... of IN B-PP
... vice NN B-NP
... chairman NN I-NP
... of IN B-PP
... Carlyle NNP B-NP
... Group NNP I-NP
... , , O
... a DT B-NP
... merchant NN I-NP
... banking NN I-NP
... concern NN I-NP
... . . O
... """
>>> sent = parse.conll_chunk(text)
```

Internally, the data is stored in a tree structure. We can print this as a nested bracketting, and we can conveniently access its children using indexes:

```
>>> print sent
(S:
  (NP: ('he', 'PRP'))
  ('accepted', 'VBD')
  (NP: ('the', 'DT') ('position', 'NN'))
  ('of', 'IN')
  (NP: ('vice', 'NN') ('chairman', 'NN'))
  ('of', 'IN')
  (NP: ('Carlyle', 'NNP') ('Group', 'NNP'))
  (',', ',')
  (NP:
    ('a', 'DT')
    ('merchant', 'NN')
    ('banking', 'NN')
    ('concern', 'NN'))
  ('.', '.'))
>>> print sent[2]
(NP: ('the', 'DT') ('position', 'NN'))
>>> print sent[2][1]
('position', 'NN')
```

We can define a regular-expression based chunk parser for use in chunking tagged text, as shown in Figure 2. NLTK-Lite also supports several other operations on chunks, such as merging, splitting, and chinking. Corpus readers for chunked data in Penn Treebank and CoNLL-2000 are provided, along with comprehensive support for evaluation and error analysis.

4.2 Simple Parsers

NLTK-Lite provides several parsers for context-free phrase-structure grammars. Grammars can be defined using a series of productions as follows:

```
>>> from nltk_lite.parse import cfg
>>> grammar = cfg.parse_grammar('''
...     S -> NP VP
...     VP -> V NP | V NP PP
...     V -> "saw" | "ate"
...     NP -> "John" | Det N | Det N PP
...     Det -> "a" | "an" | "the" | "my"
...     N -> "dog" | "cat" | "ball"
...     PP -> P NP
...     P -> "on" | "by" | "with"
...     ''')
```

Now we can tokenize and parse a sentence. Here we use a recursive descent parser. Note that we have

```
>>> sent = tag.string2tags("the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN")
>>> rule1 = parse.ChunkRule('<DT><JJ><NN>', 'Chunk det+adj+noun')
>>> rule2 = parse.ChunkRule('<DT|NN>+', 'Chunk sequences of NN and DT')
>>> chunkparser = parse.RegexpChunk([rule1, rule2], chunk_node='NP', top_node='S')
>>> chunk_tree = chunkparser.parse(sent, trace=1)
Input:
                <DT>  <JJ>  <NN>  <VBD>  <IN>  <DT>  <NN>
Chunk det+adj+noun:
               {<DT>  <JJ>  <NN>} <VBD>  <IN>  <DT>  <NN>
Chunk sequences of NN and DT:
               {<DT>  <JJ>  <NN>} <VBD>  <IN> {<DT>  <NN>}
>>> print chunk_tree
(S:
  (NP: ('the', 'DT') ('little', 'JJ') ('cat', 'NN'))
  ('sat', 'VBD')
  ('on', 'IN')
  (NP: ('the', 'DT') ('mat', 'NN')))
```

Figure 2: Regular-Expression based Chunk Parser

had to avoid using left-recursive productions in the above grammar, so that this parser does not end up in an infinite loop.

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

The recursive descent parser `rd` can be used many times over. Here we apply it to our sentence, and iterate over all the parses that it generates. Observe that two parses are possible, thanks to prepositional phrase attachment ambiguity.

```
>>> for p in rd.get_parse_list(sent):
...     print p
(S:
  (NP: 'John')
  (VP:
    (V: 'saw')
    (NP:
      (Det: 'a')
      (N: 'cat')
      (PP: (P: 'with')
           (NP: (Det: 'my') (N: 'ball'))))))
(S:
  (NP: 'John')
  (VP:
    (V: 'saw')
    (NP: (Det: 'a') (N: 'cat'))
    (PP: (P: 'with')
         (NP: (Det: 'my') (N: 'ball')))))
```

The same sentence can be parsed using a grammar with left-recursive productions, so long as we use a chart parser. Here we use the bottom-up rule-invocation strategy `BU_STRATEGY`.

```
>>> from nltk_lite.parse import cfg, chart
>>> grammar = cfg.parse_grammar('''
...     S -> NP VP
...     VP -> V NP | VP PP
...     V -> "saw" | "ate"
...     NP -> "John" | Det N | NP PP
...     Det -> "a" | "an" | "the" | "my"
...     N -> "dog" | "cat" | "ball"
...     PP -> P NP
...     P -> "on" | "by" | "with"
...     ''')
>>> parser = chart.ChartParse(grammar,
...                           chart.BU_STRATEGY)
>>> for tree in parser.get_parse_list(sent):
...     print tree
(S:
  (NP: 'John')
  (VP:
    (VP: (V: 'saw')
         (NP: (Det: 'a') (N: 'cat')))
    (PP: (P: 'with')
         (NP: (Det: 'my') (N: 'ball')))))
(S:
  (NP: 'John')
  (VP:
    (V: 'saw')
    (NP:
      (NP: (Det: 'a') (N: 'cat'))
      (PP: (P: 'with')
           (NP: (Det: 'my') (N: 'ball'))))))
```

Tracing can be turned on, to display each step of the parsing process, as shown in Figure 3. Each row represents an edge of the chart, with a specified span, together with a (possibly incomplete) dotted grammar production.

Other rule-invocation strategies are top-down, alternating top-down/bottom-up, and Earley (contributed by Jean Mark Gawron).

```
>>> parser = ChartParse(grammar, BU_STRATEGY, trace=2)
>>> parser.get_parse(sent)
|. John. saw .  a  . cat . with.  my .ball .|
Bottom Up Init Rule:
|[-----]     .     .     .     .     .     .| [0:1] 'John'
|.     [-----]     .     .     .     .     .| [1:2] 'saw'
|.     .     [-----]     .     .     .     .| [2:3] 'a'
|.     .     .     [-----]     .     .     .| [3:4] 'cat'
|.     .     .     .     [-----]     .     .| [4:5] 'with'
|.     .     .     .     .     [-----]     .| [5:6] 'my'
|.     .     .     .     .     .     [-----]| [6:7] 'ball'
Bottom Up Predict Rule:
|>     .     .     .     .     .     .     .| [0:0] NP -> * 'John'
|.     >     .     .     .     .     .     .| [1:1] V  -> * 'saw'
|.     .     >     .     .     .     .     .| [2:2] Det -> * 'a'
|.     .     .     >     .     .     .     .| [3:3] N  -> * 'cat'
|.     .     .     .     >     .     .     .| [4:4] P  -> * 'with'
|.     .     .     .     .     >     .     .| [5:5] Det -> * 'my'
|.     .     .     .     .     .     >     .| [6:6] N  -> * 'ball'
Fundamental Rule:
|[-----]     .     .     .     .     .     .| [0:1] NP -> 'John' *
|.     [-----]     .     .     .     .     .| [1:2] V  -> 'saw' *
|.     .     [-----]     .     .     .     .| [2:3] Det -> 'a' *
|.     .     .     [-----]     .     .     .| [3:4] N  -> 'cat' *
|.     .     .     .     [-----]     .     .| [4:5] P  -> 'with' *
|.     .     .     .     .     [-----]     .| [5:6] Det -> 'my' *
|.     .     .     .     .     .     [-----]| [6:7] N  -> 'ball' *
Bottom Up Predict Rule:
|>     .     .     .     .     .     .     .| [0:0] S  -> * NP VP
|>     .     .     .     .     .     .     .| [0:0] NP -> * NP PP
|.     >     .     .     .     .     .     .| [1:1] VP -> * V NP
|.     .     >     .     .     .     .     .| [2:2] NP -> * Det N
|.     .     .     .     >     .     .     .| [4:4] PP -> * P NP
|.     .     .     .     .     >     .     .| [5:5] NP -> * Det N
Fundamental Rule:
|[-----]     .     .     .     .     .     .| [0:1] S  -> NP * VP
|[-----]     .     .     .     .     .     .| [0:1] NP -> NP * PP
|.     [-----]     .     .     .     .     .| [1:2] VP -> V * NP
|.     .     [-----]     .     .     .     .| [2:3] NP -> Det * N
|.     .     [-----------]     .     .     .| [2:4] NP -> Det N *
|.     .     .     [-----]     .     .     .| [4:5] PP -> P * NP
|.     .     .     .     [-----]     .     .| [5:6] NP -> Det * N
|.     .     .     .     [-----------]     .| [5:7] NP -> Det N *
|.     [-----------------]     .     .     .| [1:4] VP -> V NP *
|.     .     .     .     [-----------------]| [4:7] PP -> P NP *
|[------------------------]     .     .     .| [0:4] S  -> NP VP *
Bottom Up Predict Rule:
|.     .     >     .     .     .     .     .| [2:2] S  -> * NP VP
|.     .     >     .     .     .     .     .| [2:2] NP -> * NP PP
|.     .     .     .     .     >     .     .| [5:5] S  -> * NP VP
|.     .     .     .     .     >     .     .| [5:5] NP -> * NP PP
|.     >     .     .     .     .     .     .| [1:1] VP -> * VP PP
Fundamental Rule:
|.     .     [-----------]     .     .     .| [2:4] S  -> NP * VP
|.     .     [-----------]     .     .     .| [2:4] NP -> NP * PP
|.     .     .     .     .     [-----------]| [5:7] S  -> NP * VP
|.     .     .     .     .     [-----------]| [5:7] NP -> NP * PP
|.     [-----------------]     .     .     .| [1:4] VP -> VP * PP
|.     .     [-----------------------------]| [2:7] NP -> NP PP *
|.     [-----------------------------------]| [1:7] VP -> VP PP *
|.     .     [-----------------------------]| [2:7] S  -> NP * VP
|.     .     [-----------------------------]| [2:7] NP -> NP * PP
|.     [-----------------------------------]| [1:7] VP -> VP * PP
|.     [-----------------------------------]| [1:7] VP -> V NP *
|[=======================================]| [0:7] S  -> NP VP *
|[=======================================]| [0:7] S  -> NP VP *
|.     [-----------------------------------]| [1:7] VP -> VP * PP
(S: (NP: 'John') (VP: (VP: (V: 'saw') (NP: (Det: 'a') (N: 'cat')))
(PP: (P: 'with') (NP: (Det: 'my') (N: 'ball')))))
```

Figure 3: Trace of Edges Created by the Bottom-Up Chart Parser

4.3 Probabilistic Parsing

A probabilistic context free grammar (or PCFG) is a context free grammar that associates a probability with each production. It generates the same set of parses for a text that the corresponding context free grammar does, and it assigns a probability to each parse. The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it. NLTK-Lite provides a Viterbi-style PCFG parser, together with a suite of bottom-up probabilistic chart parsers.

5 Contributing to NLTK-Lite

NLTK-Lite includes a variety of other modules supporting natural language processing tasks. Many more are in the planning stages, including fieldwork analysis tools, a concordancer, feature-based grammars, a cascaded chunk parser, semantic interpretation via the lambda-calculus, and several others.

NLTK-Lite is an open source project, being developed by a community of NLP researchers and teachers. It is continually being expanded and improved, with the help of interested members of the community.

There are several ways to contribute. Many users have suggested new features, reported bugs, or contributed patches via the Sourceforge site `nltk.sourceforge.net`. Several teachers and students have submitted NLTK-based projects for inclusion in the contrib directory, and in some cases these have made it into the core toolkit. The tutorials are continually being expanded and refined, with the help of input from users. The tutorials are being translated into Portuguese (by Tiago Tresoldi), and we hope to find translators for other languages.

6 Teaching with NLTK-Lite

NLTK-Lite provides ready-to-use courseware and a flexible framework for project work. Students augment and replace existing components, learn structured programming by example, and manipulate sophisticated models from the outset. Tutorials describe each component of the toolkit, and include a wide variety of student exercises and project ideas.

NLTK-Lite can be used to create student assignments of varying difficulty and scope. In the simplest assignments, students experiment with an existing module. The wide variety of existing modules provide many opportunities for creating these simple assignments. Once students become more familiar with the toolkit, they can be asked to make minor changes or extensions to an existing module. A more challenging task is to develop a new module. Here, NLTK-Lite provides some useful starting points: predefined interfaces and data structures, and existing modules that implement the same interface.

NLTK-Lite provides animated algorithms that can be used in class demonstrations. These interactive tools can be used to display relevant data structures and to show the step-by-step execution of algorithms. Both data structures and control flow can be easily modified during the demonstration, in response to questions from the class. Since these graphical tools are included with the toolkit, they can also be used by students. This allows students to experiment at home with the algorithms that they have seen presented in class.

7 Conclusion

Python is a particularly convenient language to use for writing scripts to perform natural language processing tasks. NLTK-Lite provides ready access to standard corpora, along with representations for common linguistic data structures, reference implementations for many NLP tasks, and extensive documentation including tutorials and library reference.

Acknowledgements

The work reported here has been supported by the US National Science Foundation, the Australian Research Council, and NICTA Victoria Laboratory. Much of the original NLTK implementation and documentation work was done in close collaboration with Edward Loper. Ewan Klein has provided substantial input to several of the tutorials. I am grateful to James Curran for persuading me of the need to develop this lightweight version of NLTK. Dozens of others have provided valuable contributions and feedback; they are named on the NLTK contributors page, linked from `nltk.sourceforge.net`.

References

Marti Hearst. 2005. Teaching applied natural language processing: Triumphs and tribulations. In *Proceedings of the Second ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL*, pages 1–8, Ann Arbor, Michigan, June. Association for Computational Linguistics.

Elizabeth Liddy and Nancy McCracken. 2005. Hands-on NLP for an interdisciplinary audience. In *Proceedings of the Second ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL*, pages 62–68, Ann Arbor, Michigan, June. Association for Computational Linguistics.

Edward Loper and Steven Bird. 2002. NLTK: The natural language toolkit. In *Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, pages 62–69. Somerset, NJ: Association for Computational Linguistics. `http://arXiv.org/abs/cs/0205028`.

Edward Loper. 2004. NLTK: Building a pedagogical toolkit in Python. In *PyCon DC 2004*. Python Software Foundation. `http://www.python.org/pycon/dc2004/papers/`.

Guido Van Rossum. 2003a. *An Introduction to Python*. Network Theory Ltd.

Guido Van Rossum. 2003b. *The Python Language Reference*. Network Theory Ltd.

Rune Sætre, Amund Tveit, Tonje S. Steigedal, and Astrid Lægreid. 2005. Semantic annotation of biomedical literature using google. In *Data Mining and Bioinformatics Workshop*, volume 3482 of *Lecture Notes in Computer Science*. Springer.