NLTK MODULE FOR NATURAL LANGUAGE PROCESSING

ABSTRACT

NLTK, the Natural Language Toolkit,is a suite of open source program modules, tutorials and problem sets,providing ready-to-use computational linguistics courseware. NLTK covers symbolic and statistical natural language processing, and is interfaced to annotated corpora. Students augmentand replace existing components, learn structured programming by example,and manipulate sophisticated models from the outset.

INTRODUCTION

Teachers of introductory courses on computational linguistics are often faced with the challenge of setting up a practical programming component for student assignments and projects. This is a difficult task because different computational linguistics domains require a variety of different data structures and functions, and because a diverse range of topics may need to be included in the syllabus.A widespread practice is to employ multiple programming languages, where each language provides native data structures and functions that are a good fit for the task at hand. For example, a course might use Prolog for parsing, Perl for corpus processing, and a finite statetoolkit for morphological analysis. By relying on the built-in features of various languages, theteacher avoids having to develop a lot of software infrastructure.An unfortunate consequence is that a significant part of such courses must be devoted to teaching programming languages. Further,many interesting projects span a variety of domains, and would require that multiple languages be bridged. For example, a student project that involved syntactic parsing of corpus data from a morphologically rich language might involve all three of the languages mentioned above: Perl for string processing; a finite state toolkit for morphological analysis; and Prolog for parsing. It is clear that these considerable overheads and short comings warrant a fresh approach.

Apart from the practical component, computational linguistics courses may also depend on software for in class demonstrations. This context calls for highly interactive graphical user interfaces, making it possible to view programstate (e.g. the chart of a chart parser), observe program execution step-by-step (e.g. execution of a finite-state machine), and even make minor modifications to programs in response to“what if” questions from the class. Becauseof these difficulties it is common to avoid livedemonstrations, and keep classes for theoreti-cal presentations only. Apart from being dull,this approach leaves students to solve important practical problems on their own, or to deal withthem less efficiently in office hours.

In this paper we introduce a **new approach** to the above challenges, a streamlined and flexible way of organizing the practical component of an introductory computational linguistics course. We describe NLTK, **the Natural Language Toolkit**, which we have developed in conjunction with a course we have taught atthe University of Pennsylvania.The Natural Language Toolkit is available under an open source license fromhttp://nltk.sf.net/. NLTK runs on allplatforms supported by Python, includingWindows, OS X, Linux, and Unix.

CHOICE OF PROGRAMMING LANGUAGE

The most basic step in setting up a practical component is choosing a suitable programming language.A number of considerations influenced our choice. First, the language must have a shallow learning curve, so that no vice programmers get immediate rewards for their efforts. Second, the language must support rapid prototyping and a short develop/testcycle; an obligatory compilation step is a serious detraction. Third, the code should beself documenting, with a transparent syntax and semantics. Fourth, it should be easy to writestructured programs, ideally object-oriented butwithout the burden associated with languageslike C++. Finally, the language must havean easy-to-use graphics library to support the development of graphical user interfaces.In surveying the available languages, webelieve that Python offers an especially goodfit to the above requirements. Python is an object-oriented scripting language developedby Guido van Rossum and available on allplatforms (www.python.org). Python offers a shallow learning curve; it was designed tobe easily learnt by children (van Rossum,1999). As an interpreted language, Python issuitable for rapid prototyping. Python code is exceptionally readable, and it has been praised as “executable pseudocode.” Python is an object-oriented language, but not punitively so, and it is easy to encapsulate data andmethods inside Python classes. Finally, Python has an interface to the Tk graphics toolkit(Lundh, 1999), and writing graphical interfacesis straightforward.

DESIGN CRITERIA

Several criteria were considered in the designand implementation of the toolkit. These designcriteria are listed in the order of their impor-tance. It was also important to decide whatgoals the toolkit wouldnotattempt to accom-plish; we therefore include an explicit set of non-requirements, which the toolkit is not expected to satisfy.

Requirements

Ease of Use.

The primary purpose of the toolkit is to allow students to concentrate on building natural language processing (NLP) systems.

Consistency.

The toolkit should use consistent data structures and interfaces.

Extensibility.

The toolkit should easily accommodate new components, whether those components replicate or extend the toolkit’sexisting functionality. The toolkit should be structured in such a way that it is obvious where new extensions would fit into the toolkit’s infrastructure.

Documentation.

The toolkit, its data structures, and its implementation all need to be carefully and thoroughly documented. All nomenclature must be carefully chosen and consistently used.

Simplicity.

The toolkit should structure thecomplexities of building NLP systems, not hidethem. Therefore, each class defined by thetoolkit should be simple enough that a studentcould implement it by the time they finish anintroductory course in computational linguistics.

Modularity.

The interaction between differ-ent components of the toolkit should be keptto a minimum, using simple, well defined interfaces. In particular, it should be possible tocomplete individual projects using small partsof the toolkit, without worrying about how theyinteract with the rest of the toolkit. This allows students to learn how to use the toolkit incrementally throughout a course. Modularity also makes it easier to change and extend the toolkit.

MODULES

The toolkit is implemented as a collection of independent modules, each of which defines aspecific data structure or task.A set of core modules defines basic datatypes and processing systems that are used throughout the toolkit. The token module provides basic classes for processing individualelements of text, such as words or sentences.The tree module defines data structures for representing tree structures over text, suchas syntax trees and morphological trees. The probability module implements classes thatencode frequency distributions and probabilitydistributions, including a variety of statisticalsmoothing techniques.The remaining modules define data structuresand interfaces for performing specific NLP tasks.This list of modules will grow over time, as weadd new tasks and algorithms to the toolkit.

Parsing Modules

The parser module defines a high-level inter-face for producing trees that represent the struc-tures of texts. The chunk parser module defines a sub-interface for parsers that identify non-overlapping linguistic groups (such as base nounphrases) in unrestricted text.Four modules provide implementationsfor these abstract interfaces. These parser module implements a simple shift reduce parser. The chart parser module defines a flexible parser that uses a chartt o record hypotheses about syntactic constituents. The pcfg parsermodule provides a variety ofdifferent parsers for probabilistic grammars.And there chunk parser module defines at ransformational regular expression based implementation of the chunk parser interface.

Tagging Modules

The tagger module defines a standard interfacefor augmenting each token of a text with supple-mentary information, such as its part of speech or its WordNet synset tag; and provides several different implementations for this interface.

Finite State Automata

The fsa module defines a data type for encod-ing finite state automata; and an interface forcreating automata from regular expressions.

Type Checking

Debugging time is an important factor in the toolkit's ease of use. To reduce the amount oftime students must spend debugging their code,we provide a type checking module, which can be used to ensure that functions are given valid arguments. The type checking module is used by all of the basic data types and processing classes.Since type checking is done explicitly, it can slow the toolkit down. However, when efficiency is an issue, type checking can be easily turned off; and with type checking is disabled, there is no performance penalty.

Visualization

Visualization modules define graphical interfaces for viewing and manipulating data structures, and graphical tools for experimenting with NLP tasks. The **draw.tree** module provides a simple graphical interface for displaying tree structures.The **draw.treeedit** module provides an interface for building and modifying tree structures.The **draw.plotgraph** module can be used tograph mathematical functions. The draw.fsamodule provides a graphical tool for displayingand simulating finite state automata. Thedraw.chartmodule provides an interactive graphical tool for experimenting with chart parsers.The visualization modules provide interfacesfor interaction and experimentation; they do not directly implement NLP data structures ortasks. Simplicity of implementation is thereforeless of an issue for the visualization modulesthan it is for the rest of the toolkit.

Text Classification

The classifier module defines a standard interface for classifying texts into categories.This interface is currently implemented by two modules. The classifier.naivebayes module defines a text classifier based on the Naive Bayes assumption. The classifier.maxent module defines the maximum entropy model for text classification, and implements two algorithmsfor training the model: Generalized Iterative Scaling and Improved Iterative Scaling.The classifier.feature module provides a standard encoding for the information thatis used to make decisions for a particular classification task. This standard encodingallows students to experiment with thedifferences between different text classificationalgorithms, using identical feature sets.The classifier.feature selection module defines a standard interface for choosing whichfeatures are relevant for a particular classification task. Good feature selection can significantly improve classification performance.

USES OF NLTK

Assignments

NLTK can be used to create student assign-ments of varying difficulty and scope. In thesimplest assignments, students experiment withan existing module. The wide variety of existingmodules provide many opportunities for creat-ing these simple assignments. Once studentsbecome more familiar with the toolkit, they canbe asked to make minor changes or extensions toan existing module. A more challenging task isto develop a new module. Here, NLTK provides some useful starting points: predefined interfaces and data structures, and existing modulesthat implement the same interface.

Example:

Chunk Parsing

As an example of a moderately difficult assignment, we asked students to constructa chunk parser that correctly identifies base noun phrase chunks in a given text, bydefining a cascade of transformational chunkingrules. The NLTK rechunkparser module provides a variety of regular expression based rule types, which the students caninstantiate to construct complete rules.

Class demonstrations

NLTK provides graphical tools that can be usedin class demonstrations to help explain basic NLP concepts and algorithms. 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 controlflow can be easily modified during the demon-stration, in response to questions from the class.Since these graphical tools are included withthe toolkit, they can also be used by students.This allows students to experiment at home withthe algorithms that they have seen presented in class.

Example:

The Chart Parsing Tool

The chart parsing tool is an example of a graphical tool provided by NLTK. This tool canbe used to explain the basic concepts behindchart parsing, and to show how the algorithmworks. Chart parsing is a flexible parsing algorithm that uses a data structure called a chart to record hypotheses about syntactic constituents.Each hypothesis is represented by a single edge on the chart. A set of rules determine when new edges can be added to the chart. This set of rules controls the overall behavior of the parser (e.g.,whether it parses top-down or bottom-up).The chart parsing tool demonstrates the pro-cess of parsing a single sentence, with a givengrammar and lexicon. Its display is divided intothree sections: the bottom section displays thechart; the middle section displays the sentence;and the top section displays the partial syntaxtree corresponding to the selected edge. But-tons along the bottom of the window are usedto control the execution of the algorithm. Themain display window for the chart parsing tool is shown in Figure 1.

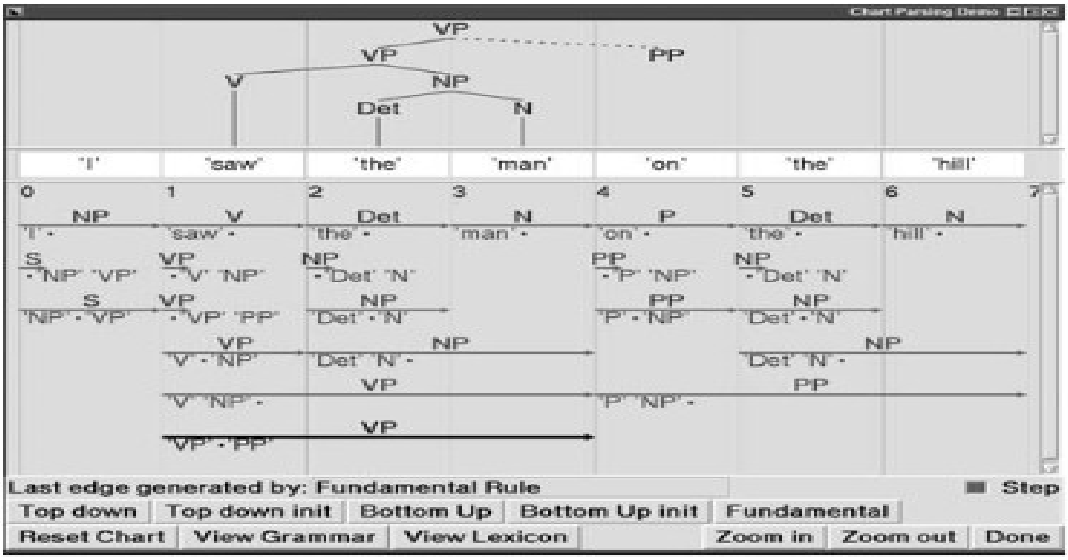

Figure 1: Chart Parsing Tool

This tool can be used to explain several different aspects of chart parsing. First, it can beused to explain the basic chart data structure,and to show how edges can represent hypotheses about syntactic constituents. It can thenbe used to demonstrate and explain the individual rules that the chart parser uses to create new edges. Finally, it can be used to show how these individual rules combine to find a completeparse for a given sentence.To reduce the overhead of setting up demon-strations during lecture, the user can define alist of preset charts. The tool can then be resetto any one of these charts at any time.The chart parsing tool allows for flexible con-trol of the parsing algorithm. At each step ofthe algorithm, the user can select which rule orstrategy they wish to apply. This allows the userto experiment with mixing different strategies(e.g., top-down and bottom-up). The user canexercise fine-grained control over the algorithmby selecting which edge they wish to apply a rule to. This flexibility allows lecturers to use thetool to respond to a wide variety of questions;and allows students to experiment with differentvariations on the chart parsing algorithm.

CONCLUSIONS AND FUTURE WORK

NLTK provides a simple, extensible, uniformframework for assignments, projects, and classdemonstrations. It is well documented, easy tolearn, and simple to use. We hope that NLTK will allow computational linguistics classes to include more hands-on experience with usingand building NLP components and systems.NLTK is unique in its combination of three factors. First, it was deliberately designed ascourseware and gives pedagogical goals primary status. Second, its target audience consists of both linguists and computer scientists, and itis accessible and challenging at many levels of prior computational skill. Finally, it is based on an object-oriented scripting language supporting rapid prototyping and literate programming.We plan to continue extending the breadth of materials covered by the toolkit. We arecurrently working on NLTK modules for Hidden Markov Models, language modeling, and tree adjoining grammars. We also plan to increase the number of algorithms implemented by some existing modules, such as the text classification module.Finding suitable corpora is a prerequisite for many student assignments and projects. We aretherefore putting together a collection of corporacontaining data appropriate for every module defined by the toolkit.NLTK is an open source project, and we wel-come any contributions. Readers who are interested in contributing to NLTK, or who have suggestions for improvements, are encouraged to contact the authors.

REFERENCES

Jason Baldridge, John Dowding, and Susana Early.2002a. Leo: an architecture for sharing resourcesfor unification-based grammars. In Proceedings of the Third Language Resources and EvaluationConference. Paris: European Language ResourcesAssociation.http://www.iccs.informatics.ed.ac.uk/~jmb/leo-lrec.ps.gz

Fernando C. N. Pereira and David H. D. Warren.1980. Definite clause grammars for language anal-ysis – a survey of the formalism and a comparisonwith augmented transition grammars.ArtificialIntelligence, 13:231–78.

Carl Pollard and Ivan A. Sag. 1994.Head-DrivenPhrase Structure Grammar.
Chicago UniversityPress.

Guido van Rossum. 1999. Computer program-ming for everybody. Technical report, Corpo-ration for National Research Initiatives.http://www.python.org/doc/essays/cp4e.html.