14. Projects with NLTK-Lite

14.1 Introduction

This document describes a variety of possible natural language processing projects that can be undertaken using NLTK-Lite.

The NLTK team welcomes contributions of good student projects, and some past projects (e.g. the Brill and HMM taggers) have been incorporated into the toolkit.

14.2 Project Topics

14.2.1 Computationally Oriented

1. NLTK will soon include a temporal expression identifier (i.e., a system capable of identifying expressions such as "last Christmas", "a forthnight ago"), build a temporal expression grounder that will assign specific timestamps to these expressions, e.g. [Day: 25; Month:12; Year: 2005]. Test the accuracy of your system on the TIMEX dataset.

2. Taking the VerbOcean data which captures semantic relationships between verbs (**`http://semantics.isi.edu`** generate a semantic network of verb relationships and implement a tree traversal algorithm that can calculate the similarity between two verbs, e.g. "fly" and "crash". You can find a demo of this system at: **`http://falcon.isi.edu/cgi-bin/graph-analysis/view-graph.pl`**

3. News stories from different sources often contain contradictory information regarding a particular event such as the number of people killed in an earthquake. Build a numerical expression recogniser and resolver that can identify equality and contradiction between numerical expression such as: "5 adults" != "3 children and 2 adults", but "5 people" = "3 children and 2 adults".

4. Implement the TnT statistical tagger in NLTK-Lite. **`http://www.aclweb.org/anthology/A00-1031`**

5. Develop a maximum-entropy POS tagger for NLTK-Lite (e.g. see MXPOST)

6. Develop a sentence tokenizer for NLTK-Lite (e.g. see MXTerminator).

7. Develop a semantic similarity module based on WordNet and two of Pedersen's Wordnet_Similarity algorithms. Develop a lexical-chain based WSD system, and evaluate it using the SEMCOR corpus (corpus reader provided in NLTK).

8. Re-implement any NLTK-Lite functionality for a language other than English (tokenizer, tagger, chunker, parser, etc). You will probably need to collect suitable corpora, and develop corpus readers.

9. Create a database of named entities, categorised as: person, location, organisation, cardinal, duration, measure, date. Train a named-entity tagger using the NIST IEER data (included with NLTK) and use it to tag more text and collect an expanded set of named entities.
10. Port the NLTK text classification system to NLTK-Lite.
11. Implement a chat-bot that incorporates a more sophisticated dialogue model than **`nltk_lite.chat.eliza`**.
12. Implement a feature-based grammar and parser in NLTK-Lite (incorporate nltk.contrib.mit.rspeer)
13. Implement a categorial grammar parser, including semantic representations.
14. Implement Pereira and Warren's Chat-80 system for geographical question answering. **`http://www.cis.upenn.edu/~pereira/oldies.html`**.
15. Develop a prepositional phrase attachment classifier, using the **`ppattach`** corpus for training and testing.
16. Develop a program for unsupervised learning of phonological rules, using the method described by Goldwater and Johnson: http://acl.ldc.upenn.edu/acl2004/sigphon/pdf/goldwater.pdf
17. Use WordNet to infer lexical semantic relationships on the entries of a Shoebox lexicon for some arbitrary language.
18. Add semantic interpretation to a CFG, generating propositions that can be evaluated against a database. Start with a lambda calculus interpreter in Python (e.g. http://www.alcyone.com/software/churc and Rob Speer's feature-based CFG (nltk_contrib/mit/rspeer/parser), and augment a feature-based grammar with attributes to hold the semantic representations.

14.2.2 Linguistically Oriented

1. Develop a morphological analyser for a language of your choice.
2. Write a soundex function that is appropriate for a language you are interested in. If the language has clusters (consonants or vowels), consider how reliably people can discriminate the second and subsequent member of a cluster. If these are highly confusible, ignore them in the signature. If the *order* of segments in a cluster leads to confusion, normalise this in the signature (e.g. sort each cluster alphabetically, so that a word like **`treatments`** would be normalised to **`rtaemtenst`**, before the code is computed). (NB. See field.html for more details.)
3. Develop a text classification system which efficiently classifies documents in two or three closely related languages. Consider the discriminating features between languages despite their apparent similarity. Implementation should be evaluated using unseen data.
4. Explore the phonotactic system of a language you are interested in. Compare your findings to a published phonological or grammatical description of the same language.
5. Implement a structured text rendering module which takes linguistic data from a source such as Shoebox and generates XML based lexicon or interlinear text based on user preferences for field exports.

6. Develop a grammatical paradigm generation function which takes some form of tagged text as input and generates paradigm representations of related linguistic features.

14.2.3 Previous Projects

The following projects were undertaken in 2005 and implementations will be made available in a planned NLTK-Lite-Contrib distribution. More work can be done on these, building on the existing work.

1. Develop a concordance system for the Brown Corpus, supporting searches that include part-of-speech tags, replicating some of the functionality of commercial software (e.g. MonoConc). Investigate indexing for more efficient searches.

2. Build a language-guesser, to classify text documents by language, replicating some of the functionality of TextCat. Create an evaluation dataset, possibly using the Opus_Corpus.

3. Implement Penton's paradigm visualisation tool. **`http://www.cs.mu.oz.au/~djpenton/`**

4. Implement a system for cascaded_chunking.

5. Build a text compression system combining a dictionary-based compression algorithm (such as the ones described in Managing_Gigabytes) along with information provided by a part-of-speech tagger which should lower the conditional entropy of the following word.

14.3 Assessment

This section describes the project assessment requirements for *433-460 Human Language Technology* at the University of Melbourne. Project assessment has three components: an oral presentation (5%), a written report (10%), and an implementation (20%).

14.3.1 Oral Presentation

Students will give a 10-minute oral presentation to the rest of the class in the second-last week of semester. This will be evaluated for the quality of content and presentation:

- presentation (clarity, presentation materials, organization)
- content (defining the task, motivation, data, results, outstanding issues)

14.3.2 Written Report

Students should submit a ~5-page written report, with approximately one page covering each of the following points:

- introduction (define the task, motivation)
- method (any algorithms, data)
- implementation (description, how to run it)

- results (e.g. show some output and discuss)
- evaluation (your critical discussion of the work)

This should be prepared using the Python **`docutils`** and **`doctest`** packages. These are easily learnt, and ideally suited for creating reports with embedded program code, and they have been used for all NLTK-Lite documentation. For a detailed example, see the text source for the NLTK-Lite tagging tutorial (text, html).

- Docutils: an open-source text processing system for processing plaintext documentation into useful formats, such as HTML or LaTeX. It includes reStructuredText, the easy to read, easy to use, what-you-see-is-what-you-get plaintext markup language.
- Doctest*: a standard Python module that searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown.

14.3.3 Implementation

Marks will be be awarded for the basic implementation and for various kinds of complexity, as described below:

- Basic implementation (10%)
  - we are able to run the system
  - we can easily test the system (interface is usable, output is appropriately detailed and clearly formatted)
  - we can easily work out how the system is implemented (understandable code, inline documentation; you can assume we read the report first)
  - the system implements NLP algorithms (i.e. relevant to the subject, re-using existing NLP algorithms wherever possible instead of reinventing the wheel)
  - the NLP algorithms are correctly implemented
- Complexity (10%)
  - the system implements a non-trivial problem
  - the system combines multiple HLT components as appropriate
  - appropriate training data is used (effort in obtaining and preparing the data will be considered)
  - the system permits exploration of the problem domain and the algorithms (e.g. through appropriate parameterization)
  - a range of system configurations/modifications are explored (e.g. classifiers trained and tested using different parameters)