Euromasters summer school 2005

Introduction to NLTK

Trevor Cohn

July 12, 2005

Course Overview

- Morning session
  - tokenization
  - tagging
  - language modelling
  - followed by laboratory exercises
- Afternoon session
  - shallow parsing
  - CFG parsing
  - followed by laboratory exercises

Why NLTK?

NLTK: a software package for manipulating linguistic data and performing NLP tasks

- advanced tasks are possible from an early stage
- permits projects at various levels:
  - individual components vs complete systems
- consistent interfaces:
  - sets useful boundary conditions
  - models structured programming
  - facilitates reusability

Introduction to NLTK

- NLTK provides:
  - Basic classes for representing data relevant to NLP
  - Standard *interfaces* for performing NLP tasks
    - Tokenization, tagging, parsing
  - Standard *implementations* of each tasks
    - Combine these to solve complex problems
- Organization:
  - Collection of task-specific modules and packages
  - Each contains:
    - Data-oriented classes to represent NLP information
    - Task-oriented classes to encapsulate the resources and methods needed to perform a particular task

NLTK Modules

- token: classes for representing and processing individual elements of text, such as words and sentences
- probability: probabilistic information
- tree: hierarchical structures over text
- cfg: context free grammars
- fsa: finite state automata
- tagger: tagging each word with part-of-speech, sense, etc.
- parser: building trees over text
  - chart, chunk, probabilistic
- classifier: classify text into categories
  - Feature, maxent, naivebayes
- draw: visualize NLP structures and processes

Using NLTK

- Download distribution from nltk.sf.net
  - 1.4 released recently
- Check out CVS tree
  - cvs -d:pserver:anonymous@cvs.nltk.sourceforge.net:/cvsroot/nltk
- Use version installed on DICE:
  - /usr/bin/python2.3
- Documentation:
  - http://nltk.sf.net/docs.html
  - tutorials and API documentation

The Token Module (nltk.token)

- Motivation: *divide a text into manageable units, recognize them individually, model their arrangement*
- Tokens and types:
  - "word": abstract vocabulary item, or an instance in a text?
  - e.g. "my dog likes your dog": 5 tokens, 4 types
  - NLTK tokens are a kind of Python dictionary
- Text locations (cf Python *slices*)
  - @[s:e] specifies a region of text (s=start, e=end (exclusive))

```
>>> Token(TEXT='dog', LOC=CharSpanLocation(0,4))
<dog>@[0:4c]
```


Tokenizers (nltk.tokenizer)

- Tokenizers convert a string into a list of tokens
  - Each token has a type and a location
- Example: white-space tokenizer

```
>>> from nltk.tokenizer import *
>>> text_token = Token(TEXT='My dog likes your dog')
>>> ws = WhitespaceTokenizer(SUBTOKENS='WORD');
>>> ws.tokenize(text_token, add_locs=True)
>>> print text_token
<[<My>@[0:2c], <dog>@[3:6c], <likes>@[7:12c],
<your>@[1317c],<dog>@[18:21c]]>
```


Tokenizers cont.

- Other tokenizers in NLTK:
  - LineTokenizer – split the text into lines
  - RegexpTokenizer – split the text into units matching the RE

```
>>> from nltk.tokenizer import *
>>> text_token = Token(
    TEXT='My dog, Suzie, doesn\'t like your dog!')
>>> tokenizer = RegexpTokenizer(r'\w+|[^\s\w]+',
    SUBTOKENS='WORDS')
>>> tokenizer.tokenize(text_token)
>>> text_token
<[<My>, <dog>, <,>, <Suzie>, <,>, <doesn>, <'>, <t>,
  <like>, <your>, <dog>, <!>]>
```


Part-of-speech Tagging

- Tags
  - introduction
  - tagged corpora, tagsets
  - representing tags in NLTK
- Tagging
  - motivation
  - default tagger; unigram tagger; n-gram tagger
  - Brill tagger & transformation-based learning
- Evaluation

Tags 1: ambiguity

- fruit flies like a banana
- ambiguous headlines
  - http://www.snopes.com/humor/nonsense/head97.htm
  - "British Left Waffles on Falkland Islands"
  - "Juvenile Court to Try Shooting Defendant"

Tags 2: Representations to resolve ambiguity

Table 1.

Fruit	flies	like	a	banana
noun	verb	prep	det	noun

Table 2.

Fruit	flies	like	a	banana
noun	noun	verb	det	noun

Tags 3: Tagged Corpora

The/at Pantheon's/np$ interior/nn ,/, still/rb in/in its/pp$ original/jj form/nn ,/, is/bez truly/ql majestic/jj and/cc an/at architectural/jj triumph/nn ./. Its/pp$ rotunda/nn forms/vbz a/at perfect/jj circle/nn whose/wp$ diameter/nn is/bez equal/jj to/in the/at height/nn from/in the/at floor/nn to/in the/at ceiling/nn ./. The/at only/ap means/nn of/in interior/jj light/nn is/bez the/at twenty-nine-foot-wide/jj aperture/nn in/in the/at stupendous/jj dome/nn ./.

Source: Brown Corpus (nltk/data/brown/cf41)

Another kind of tagging: Sense Tagging

The Pantheon's interior/a , still in its original/a form/a ,

interior: (a) inside a space; (b) inside a country and at a distance from the coast or border; (c) domestic; (d) private.

original: (a) relating to the beginning of something; (b) novel; (c) that from which a copy is made; (d) mentally ill or eccentric.

form: (a) definite shape or appearance; (b) body; (c) mould; (d) particular structural character exhibited by something; (e) a style as in music, art or literature; (f) homogenous polynomial in two or more variables; ...

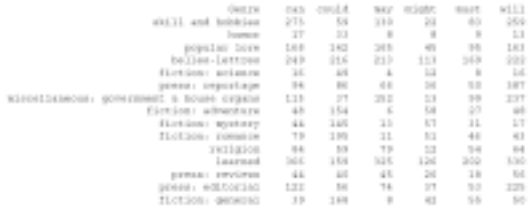
Significance of Parts of Speech

- a word's POS tells us a lot about the word and its neighbors
  - limits the range of meanings (*deal*), pronunciations (*object* vs *object*), or both (*wind*)
  - helps in stemming
  - limits the range of following words for ASR
  - helps select nouns from a document for IR
- More advanced uses (these won't make sense yet):
  - basis for chunk parsing
  - basis for searching for linguistic constructions (e.g. contexts in concordance searches)
  - parsers can build trees directly on the POS tags instead of maintaining a lexicon

Tagged Corpora

- Brown Corpus:
  - The first digital corpus (1961), Francis and Kucera, Brown U
  - Contents: 500 texts, each 2000 words long
    - from American books, newspapers, magazines, representing 15 genres
  - See /usr/share/nltk-data/brown/
  - See tokenization tutorial section 6 for discussion of Brown tags
- Penn Treebank:
  - First syntactically annotated corpus
  - Contents: 1 million words from WSJ; POS tags, syntax trees
  - See /usr/share/nltk-data/treebank/ (5% sample)

Application of tagged corpora: genre classification



Important Treebank Tags

NN noun	JJ adjective
NNP proper noun	CC coord conj
DT determiner	CD cardinal number
IN preposition	PRP personal pronoun
VB verb	RB adverb

-R comparative

-S superlative or plural

-$ possessive

Verb Tags

VBP base present	*take*
VB infinitive	*take*
VBD past	*took*
VBG present participle	*taking*
VBN past participle	*taken*
VBZ present 3sg	*takes*
MD modal	*can, would*

Representing Tags in NLTK

- Tokens

```
>>> tok = Token(TEXT='dog', TAG='nn')
<dog/nn>
>>> tok['TEXT']
'dog'
>>> tok['TAG']
'nn'
```

Simple Tagging in NLTK

- Reading Tagged Corpora:

```
>>> from nltk.corpus import brown
>>> brown.items()
['ca01', 'ca02', ...]
>>> tok = brown.read('ca01')
[<The/at>, <Fulton/np-tl>, <County/nn-tl>, ...
```

- Tagging a string

```
>>> from nltk.token import *
>>> from nltk.tokenreader.tagged import TaggedTokenReader
>>> text_str = """
... John/nn saw/vb/ the/at book/nn on/in the/at table/nn ./end
... """
>>> reader = TaggedTokenReader(SUBTOKENS='WORDS')
>>> text_token = reader.read_token(text_str)
>>> print text_token['WORDS']
[<John/nn>, <saw/vb>, <the/at>, <book/nn>, <on/in>, <the/at>...]
```

Tagging Algorithms

- default tagger
  - guess the most common tag
  - inspect the word and guess a likely tag
- unigram tagger
  - assign the tag which is the most probable for the word in question, based on frequency in a training corpus
- bigram tagger, n-gram tagger
  - Inspect one or more tags in the context (usually, immediate left context)
- backoff tagger
- rule-based tagger (Brill tagger), HMM tagger

Default Tagger

```
>>> text_token = Token(TEXT="John saw 3 polar bears .")
>>> WhitespaceTokenizer().tokenize(text_token)
>>> print text_token
<[<John>, <saw>, <3>, <polar>, <bears>, <.>]>
>>> my_tagger = DefaultTagger('nn')
>>> my_tagger.tag(text_token)
>>> print text_token
<[<John/nn>, <saw/nn>, <3/nn>, <polar/nn>, <bears/nn>,
  <./nn>]>
```

Regular Expression Tagger

```
>>> NN_CD_tagger = RegexpTagger(
    [(r'^[0-9]+(.[0-9]+)?$', 'cd'),
     (r'.*', 'nn')])
>>> NN_CD_tagger.tag(text_token)
>>> print text_token
<[<John/nn>, <saw/nn>, <3/cd>, <polar/nn>,
  <bears/nn>, <./nn>]>
```

Unigram Tagger

- Unigram = table of tag frequencies for each word
  - e.g. in tagged WSJ sample (from Penn Treebank):
    - deal: NN (11); VB (1); VBP (1)
- Training
  - load a corpus
  - access its tokens
  - train the tagger on the tokens: tagger.train()
- Tagging
  - use the tag method: tagger.tag()

Unigram Tagger (cont)

```
>>> from nltk.tagger import *
>>> from nltk.corpus import brown
>>> mytagger = UnigramTagger()
>>> for item in brown.items()[:10]:
...     tok = brown.tokenize(item)
...     mytagger.train(tok)
>>> text_token = Token(
        TEXT="John saw the book on the table")
>>> WhitespaceTokenizer().tokenize(text_token)
>>> mytagger.tag(text_token)
>>> print text_token
<[<John/np>, <saw/vbd>, <the/at>, <book/None>, <on/in>,
  <the/at>, <table/nn>]>
```


What just happened?

- 90% accuracy
- How does unigram tagger work? (See the code!)
  - TRAINING:

```
for subtok in tok[SUBTOKENS]:
    word = subtok[TEXT]
    tag = subtok[TAG]
    self._freqdist[word].inc(tag)
```

  - TAGGING:

```
context = subtok[i][TEXT]
return self._freqdist[context].max()
```

- freqdist: a convenient method for counting events

Aside: Frequency Distributions

- freq dist records the number of times each outcome of an experiment has occurred

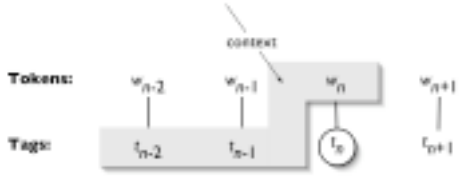
```
>>> from nltk.probability import FreqDist
>>> fd = FreqDist()
>>> for tok in text['WORDS']:
...     fd.inc(tok['TEXT'])
>>> print fd.max()
'the'
```

- Other methods:
  - fd.count('the') -> 25
  - fd.freq('the') -> 0.025
  - fd.N() -> 1000
  - fd.samples() -> ['the', 'cat', ...]
- Conditional frequency distribution: a hash of freq dists

Fixing the problem using a bigram tagger

- construct sentences involving a word which can have two different parts of speech
  - e.g. wind: noun, verb
  - The wind blew forcefully
  - I wind up the clock
- gather statistics for current tag, based on:
  - (i) current word; (ii) previous tag
  - result: a 2-D array of frequency distributions
  - what does this look like?

Generalizing the context



Bigram & n-gram taggers

- n-gram tagger: consider *n-1* previous tags
  - tagger = NthOrderTagger(n-1)
  - *how big does the model get?*
  - *how much data do we need to train it?*
- Sparse-data problem:
  - As *n gets large, the chances of having seen all possible patterns of tags during training diminishes (large: >3)*
- *Approaches:*
  - Combine taggers (backoff, weighted average)
  - statistical estimation (for the probability of unseen events)
  - throw away order (naive Bayes)

Combining Taggers: Backoff

- Try to tag w_n with trigram tagger: Cond = (w_n, t_{n-1}, t_{n-2})
  - If cond wasn't seen during training, backoff to bigram tagger
- Try to tag w_n with bigram tagger: Cond = (w_n, t_{n-1})
  - If cond wasn't seen during training, backoff to unigram tagger
- Try to tag w_n with unigram tagger: Cond = w_n
  - If cond wasn't seen during training, backoff to default tagger
- Tag w_n using default tagger: Cond = 0
- NLTK:
  - tagger = BackoffTagger([tagger1, tagger2, tagger3, tagger4])
- Are there any problems with this approach?

Evaluating Tagger Performance

Need an objective measure of performance. Steps:

- tagged tokens - the original `gold standard' data
  [<John/nn>, <saw/vb>, <the/dt>, ...]
- untag the data
  [<John>, <saw>, <the>, ...]
- tag the data with your own tagger
  [<John/nn>, <saw/nn>, <the/nn>, ...]
- compare the original and new tags
  - accuracy(orig, new) = fraction correct
  - **nltk.eval.{accuracy,precision,...}** functions

Language Modelling

- Goal: Find the probability of a "text"
  - "text" can be a word, an utterance, a document, etc.
- Texts are generated by an unknown probability distribution
  - A "language model" captures a priori information about the likelihood of a text
  - We are more likely to predict a text with a higher a priori probability
- Why do language modelling?
  - Speech recognition: Predict likely word sequences
  - Spelling correction: Suggest likely words
  - Machine translation: Suggest likely translations
  - Generation: Generate likely sentences

Language Modelling (2)

- Each text generates an output form, which we can directly observe (we want to discover the input form)
  - **Speech recognition: a sequence of sounds**
  - **Spelling correction: a sequence of characters**
  - **Machine translation: a source language text**
- No way to determine *P(output)*
- Task: Find the most likely text for an output form:

$$argmax_{text} P(text \mid output)$$

Language Modelling (3)

- Bayes Rule:

Language Model

$$P(text \mid output) = \frac{P(text)P(output \mid text)}{P(output)}$$

- Recovering the "underlying" form:

$$argmax_{text} P(text \mid output) =$$

$$argmax_{text} \frac{P(text)P(output \mid text)}{P(output)} =$$

fixed

$$argmax_{text} P(text)P(output \mid text)$$

Language Modelling (4) Equivalence Classes

- Estimating P(text)
  - $P(w_1 \ldots w_n) = P(w_n|w_1 \ldots w_{n-1})\ P(w_{n-1}|w_1 \ldots w_{n-2}) \ \ldots\ P(w_2|w_1)$
  - $P(w_n|w_1, \ldots, w_{n-1})$ has a large sample space
- Divide $P(w_n|w_1, \ldots, w_{n-1})$ into equivalence classes
  - Example: $P(w_n|w_1, \ldots, w_{n-1}) \cong P(w_n|w_{n-1})$
- Estimate the probability of each equivalence class
  - Training data
  - Count the number of training instances in each equivalence class
  - Use these counts to estimate the probability for each equivalence class

Language Modelling (5) Maximum Likelihood Estimation

- Predict the probability of an equivalence class using its relative frequency in the training data:

$$P(x) = \frac{C(x)}{N}$$

  - C(x) = count of x in training, N = number of training instances
- Problems with MLE:
  - Underestimates the probability for unseen data: **C(x)=0**
    - **Maybe we just didn't have enough training data**
  - Overestimates the probability for rare data: **C(x)=1**
    - **Estimates based on one training sample are unreliable**
  - **Solution: smoothing**

NLTK Example

```
>>> from nltk.corpus import gutenberg
>>> from nltk.probability import ConditionalFreqDist
>>> text_token = gutenberg.read('chesterton-thursday.txt')
>>> cfdist = ConditionalFreqDist()

>>> prev = '<s>'
>>> for word in text_token['WORDS']:
...     cfdist[prev].inc(word['TEXT'])
...     prev = word['TEXT']
>>> print cfdist['red'].count('hair')
9
>>> print cfdist['red'].N()
40
>>> print cfdist['red'].freq('hair')
0.225
>>> print cfdist['red']
<FreqDist: 'and': 5, 'head': 1, 'flames,': 1, 'rosette,': 1, 'hair': 9,
'houses': 1, 'mouth': 1, 'hair,': 2, 'wine,': 1, 'subterranean': 1,
'face': 1, 'eye.': 1, 'flower': 2, 'sky,': 1, 'thread': 1, 'sun': 1,
'rosette': 1, 'light.': 1, 'up': 1, 'patch': 1, 'mane': 1, 'clay': 1,
'cloud.': 1, 'river': 1, 'or': 1, 'sunset.': 1>
>>> pdist = MLEProbDist(cfdist['red'])
>>> print pdist.prob('hair')
0.225
```


Laplace Smoothing

- Mix the MLE estimate with uniform prior
  - $P_0(w_1, \ldots, w_n) = 1 / B$
    (B is the number of distinct n-grams)
  - $P_{MLE}(w_1, \ldots, w_n) = C(w_1, \ldots, w_n) / N$
    (N is the total number of n-grams in training)
- Relative weighting: $P = \alpha P_0 + (1-\alpha) P_{MLE}$
  - $\alpha = B/(N+B)$
  - $P_{Lap}(w_1, \ldots, w_n) = (C(w_1, \ldots, w_n)+1)/(N+B)$
  - "add-one smoothing"

NLTK Example

```
>>> from nltk.corpus import gutenberg
>>> from nltk.probability import *
>>> text_token = gutenberg.tokenize('chesterton-thursday.txt')
>>> cfdist = ConditionalFreqDist()

>>> prev = '<s>'
>>> for word in text_token['WORDS']:
...     cfdist[prev].inc(token['TEXT'])
...     prev = token['TEXT']

>>> mle = MLEProbDist(cfdist['red'])
>>> laplace = LaplaceProbDist(cfdist['red'], 11200)
>>> for s in mle.samples():
...     print s, mle.prob(s), laplace.prob(s)
and 0.125 0.000533807829181
head 0.025 0.00017793594306
flames, 0.025 0.00017793594306
rosette, 0.025 0.00017793594306
hair 0.225 0.000889679715302
houses 0.025 0.00017793594306
mouth 0.025 0.00017793594306
hair, 0.05 0.000266903914591
subterranean 0.025 0.00017793594306
face 0.025 0.00017793594306
flower 0.05 0.000266903914591
```


Other smoothing methods

- ELE and Lidstone smoothing
  - Instead of "add 1", "add 1/2" (ELE), or "add λ"
  - $P_{ELE}(w_1, \ldots, w_n) = (C(w_1, \ldots, w_n)+0.5)/(N+0.5*B)$
  - $P_{Lid}(w_1, \ldots, w_n) = (C(w_1, \ldots, w_n)+\lambda)/(N+\lambda B)$
  - In NLTK
    - nltk.probability.ELEProbDist
    - nltk.probability.LidstoneProbDist
- Also to be found
  - heldout estimation, Good-Turing, Witten-Bell ...