The Natural Language Toolkit

Montréal-Python 18 (Boat-shaped Benefactress)

Pablo Ariel Duboue

Les Laboratoires Foulab
999 Rue du College
Montreal, H4C 2S3, Quebec

January 31st 2011

Outline

What is Natural Language Processing

- Processing language with computers.
  - Plenty of practical applications (blogs, twitter, phones, etc).
  - The ultimate AI frontier?
- However...
  - People are good at language, it comes naturally to them (their mother tongue, that is).
  - NLP practitioners have a mixture of interest in both language and mathematics, an unusual combination.
  - Lots of engineering and tweaking (there must be a better way!).

NLTK

- A **toolkit**, a collection of python packages and objects very tailored to NLP subtasks.
- NLTK is both a tool to introduce newcomers to the state of the art of NLP practice, plus allow seasoned practitioners to feel at home within the environment.
- Compared to other **frameworks**, the NLTK has very strong defaults (e.g., a text is a sequence of words), which can be changed.
- NLTK not only targets NLP practitioners with a computer science background but it also provides valuable tools for corpus-based linguists fieldwork.
- NLTK blends with Python rather than “being implemented” in Python.

Important facts

- NLTK started at University of Pennsylvania, as part of **computational linguistics** course.
- Available at `http://www.nltk.org/`.
- Source code distributed under the Apache License version 2.0.
- There is a 500-page book authored by Bird, Klein and Loper available via O'Reilly "Natural Language Processing with Python", highly recommendable.
- The toolkit includes also **data** in the form of text collections (many with **annotations**) and **statistical models**.

NLTK and Python

- The toolkit tries to get itself out of the day and allows you to do most things with Python idioms, such as slices and list comprehensions.
- Therefore, a text is list of words and any list of words can be transformed into a text.
- The toolkit design goals (Simplicity, Consistency, Extensibility and Modularity) go hand in hand with Python own design.
  - Faithful to these goals, NLTK refrains from creating its own classes when Python defaults dictionaries, tuples and lists suffice.

NLTK main packages

Accessing corpora Interfaces to collections of documents and dictionaries.

String processing Tokenization, sentence detection, stemmers.

Collocation discovery A collocation is a pair of tokens that occur most often than chance.

Part-of-speech tagging Telling nouns apart from verbs, etc.

Classification General classifiers, based on training material provided as a Python dictionary.

Chunking Splitting a sentence into coarsed-units

Parsing Full-fledged (syntactic, and others) parsing.

Semantic interpretation Lambda calculus, first-order logic, etc.

Evaluation metrics Precision, recall, etc.

Probability and estimation Frequency distributions, estimators, etc.

Applications WordNet browser, chatbots

Some examples

- For trying, importing everything in `nltk.book` puts your python interpreter ready to go.
  - You might need to download first some data blobs first, by importing nltk and issuing nltk.download().
- You can then ask, for example for words similar based on their contexts, given a text.

NLTK similar words by context

```
>>> from nltk.book import *
# long comment, skipped
>>> moby_dick = text1
>>> moby_dick.similar('poor')
Building word-context index...
old sweet as eager that this all own help peculiar german crazy three
at goodness world wonderful floating ring simple
>>> inaugural_addresses = text4
>>> inaugural_addresses.similar('poor')
Building word-context index...
free south duties world people all partial welfare battle settlement
integrity children issues idealism tariff concerned young recurrence
charge those
```

Some background

- This topic was suggested by Yannick, as he pointed out the importance of telling apart English vs. French documents for Montrealers.
- In language identification, there are multiple approaches, although the two preferred involve using word dictionaries or distribution of characters tuples (or *n*-grams).
- I prefer character-based methods as they do not require tokenization (splitting the text into words), which is language dependent.
  - Moreover, you can detect the language with **very few characters**, and when none of them are a common dictionary word (Twitter anyone?).
- The canonical citation here would be:
  - Dunning, T. (1994) “Statistical Identification of Language”. Technical Report MCCS 94-273, New Mexico State University, 1994.

The data

- For data, we will be using the “European Parliament Proceedings Parallel Corpus 1996-2009”
  - Parallel corpus of 11 European languages, aligned sentence by sentence.
  - Heavily used in Statistical Machine Translation.
  - `http://www.statmt.org/europarl/`
  - License: “We are not aware of any copyright restrictions of the material.”
- We will be using the first 1,000 sentences from the parallel corpus French-English.
  - The full corpus is 176 MB, and goes from 04/1996-10/2009

In a nutshell

- First, we want to build a statistical model and see if there is some "signal" in the data until we get the right *n* for the *n*-grams (sequences of *n*-characters).
- Then, we want to select a few *n*-grams that are highly discriminatory between English and French.
  - To do that, we train a classifier based on one feature (named 'ngram') and using all available *n*-grams.
  - We then extract the most **informative** features.

    informativeness of a feature $(name, val)$ *is equal to the highest value of* $P(name = val|label)$*, for any label, divided by the lowest value of* $P(name = val|label)$*, for any label:*

    $$max_{l_1}(P(name = val|l_1))/min_{l_2}(P(name = val|l_2))$$

In a nutshell (cont.)

- Now that we have the features, we train second classifier.
- Each feature is now a most informative *n*-gram, as computed in the previous step.
- In this example, the value for a feature will be '1' (just a binary feature indicating whether the *n*-gram is present or not).
  - An alternative approach is to use how many times the *n*-gram actually occurs, but using binary features is more robust for short texts.

Console

```
Python 2.5.5 (r255:77872, Nov 28 2010, 16:43:48)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import nltk
>>> en = open("en.txt").read()
>>> fr = open("fr.txt").read()
>>> for n in range(1,5):
...     fm_fr=nltk.model.NgramModel(n,[x for x in fr])
...     fm_en=nltk.model.NgramModel(n,[x for x in en])
...     "".join(fm_fr.generate(50))
...     "".join(fm_en.generate(50))
...     print "\n"
...
',.....................................................',
' .....................................................',
'r.....................................................'


"Rais amerl tt lagrvent ert l'\xc3\xa9sess \xc3\xa9n itrercutal"
'REurecte irgo co ored owon oeniere ss, iast thof m'


'Restres unenneur pui souseurammient \xc3\xaachansien ces'
'Reget be a reaso, taturproder, region to loyme gre'


"Repriode 34 ann\xc3\xa9e d'int\xc3\xa9.\nUne pour les esports 1"
'Resulture, for Objects pointy-five alongrammently '
```

What we just used

- `nltk.model.NgramModel`
- It takes the value of *n* for the *n*-gram model and the list of events over which take the *n*-grams.
- The class is usually employed over words, but NLTK is very flexible and it operates over any list of items.
- A model is a probabilistic description of a set of instances.
  - A model can tell you how **likely** a new instance is to belong to that set.
    - The method `prob` from `ModelI`, superclass of `NgramModel`.
  - And it can also produce random sequences representative of the model.
    - The method `generate` just employed.

Console (cont.)

```
>>> def make_dict(x):
...     return dict(ngram=x)
...
>>> train = [(make_dict(x),'en') for x in nltk.util.ingrams(en,4)] + \
...   [(make_dict(x),'fr') for x in nltk.util.ingrams(fr,4)]
>>>
>>> classifier = nltk.NaiveBayesClassifier.train(train)
>>> classifier.show_most_informative_features()
Most Informative Features
                ngram = ('i', 'n', 'g', '␣')     en : fr     =    455.3 : 1.0
                ngram = ('␣', 'd', 'e', '␣')     fr : en     =    200.6 : 1.0
                ngram = ('t', 'r', 'e', '␣')     fr : en     =    192.5 : 1.0
                ngram = ('n', 'g', '␣', 't')     en : fr     =    161.9 : 1.0
                ngram = ('s', '␣', 't', 'h')     en : fr     =    155.2 : 1.0
                ngram = ('d', 'e', '␣', 'c')     fr : en     =    137.0 : 1.0
                ngram = ('t', '␣', 'o', 'f')     en : fr     =    135.1 : 1.0
                ngram = ('u', 'r', '␣', 'l')     fr : en     =    132.7 : 1.0
                ngram = ('o', 'n', 't', '␣')     fr : en     =    132.2 : 1.0
                ngram = ('i', 'n', '␣', 't')     en : fr     =    123.5 : 1.0
>>>
>>> features = [x[1] for x in classifier.most_informative_features(100)]
>>> # [('i', 'n', 'g', ' '), (' ', 'd', 'e', ' '), ('t', 'r', 'e', ' '), ('n', 'g', ' ', 't')
```

What we just used

- `nltk.util.ingrams`,
  - takes as input a sequence and the *n* for the *n*-grams
    - Our sequence is the training string, a sequence of characters.
  - returns a list of tuples of the specified size.
    - `nltk.util.ingrams([1,2,3,4,5],3)` → [(1, 2, 3), (2, 3, 4), (3, 4, 5)]
- The training data for the NLTK classifiers is a list of pairs.
  - The first entry is a dictionary of feature names mapped into feature values.
    - We only have one feature 'ngram', hard-coded in the make_dict helper function.
  - The second entry is the class label (French and English in our case).
- To train we use `nltk.NaiveBayesClassifier`.
  - There are other classifiers and bindings to external packages like Weka.

Console (cont.)

```
>>> def gen_feats(s):
...     return dict([(x,1) for x in nltk.util.ingrams(s,4) if x in features])
...
>>> final_train = [ (gen_feats(line),'en') for line in en.split('\n')] + \
...     [(gen_feats(line),'fr') for line in fr.split('\n')]
>>> final_classifier=nltk.NaiveBayesClassifier.train(final_train)
>>> final_classifier.show_most_informative_features()
Most Informative Features
    ('i', 'n', 'g', '␣') = 1                en : fr     =    286.3 : 1.0
    ('t', 'r', 'e', '␣') = 1                fr : en     =    174.3 : 1.0
    ('d', 'e', '␣', 'c') = 1                fr : en     =    133.7 : 1.0
    ('s', '␣', 't', 'h') = 1                en : fr     =    128.3 : 1.0
    ('n', 'g', '␣', 't') = 1                en : fr     =    126.3 : 1.0
    ('o', 'n', 't', '␣') = 1                fr : en     =    125.0 : 1.0
    ('t', '␣', 'q', 'u') = 1                fr : en     =    109.7 : 1.0
    ('t', '␣', 'o', 'f') = 1                en : fr     =    105.7 : 1.0
    ('u', 'r', '␣', 'l') = 1                fr : en     =    104.6 : 1.0
    ('␣', 'd', 'e', '␣') = 1                fr : en     =    101.8 : 1.0
>>> en_tweet = "It's␣not␣the␣least␣bit␣selfish␣to␣be␣committed␣to␣yourself␣because␣being␣your
>>> fr_tweet = "vos␣retours␣pour␣la␣qualite␣de␣la␣TV␣chez␣Bouygues␣sur␣iPhone,␣c'est␣bien␣?"
>>> final_classifier.batch_classify([ (gen_feats(en_tweet)) ] + [ (gen_feats(fr_tweet)) ] )
['en', 'fr']
>>> final_classifier.prob_classify((gen_feats(en_tweet))).prob('en')
0.99991573927860278
>>> final_classifier.prob_classify((gen_feats(fr_tweet))).prob('fr')
0.999999926520446
```

What we just used

- Interestingly, there's very little new NLTK-specific "magic" in the last bit, it is just Python!
  - That is why we can say NLTK blends itself into Python rather than use it as a black-box.
- We want to train a classifier over the top *M* most informative 4-grams (we use 100, the default for `most_informative_features`) as features.
  - Before, `train` had inside a dictionary with only one key 'ngram', now it will have at most 100 keys, each one for a top informative 4-gram.
  - That process is accomplished by the `gen_feats` helper function, it goes through all the 4-grams in the string and filters only the ones in most informative features.
  - Using the default constructor for `dict` means duplicates are not accounted for.
    - Some Python magic can change that, although I prefer binary features.

What we just used (cont.)

- Once trained, we are ready to predict!
  - We took two tweets randomly from `http://twitter.com/public_timeline`.
  - One in French, one in English.
  - Using the `batch_classify` method, both tweets are correctly identified.
  - Using the `prob_classify` method, we can also see they do so with very high probability.

About The Speaker

- PhD in CS, Columbia University (NY)
  - Natural Language Generation
- Joined IBM Research in 2005
  - Worked in
    - Question Answering
    - Expert Search
    - DeepQA (Jeopardy!)
- Nowadays
  - Left IBM Research in August, currently on sabbatical
  - Helping organize the Debian Conference
  - Member at Foulab
  - Cooking some start-up ideas
  - Will be teaching in Argentina March-June
  - `mailto:pablo.duboue@gmail.com`

Some comments about the state of the art in NLP

- Open discussion.

Is NLTK production ready?

- Sure, if your production environment tolerates Python, then it will fit fine with NLTK.
- However, the maturity of the different packages varies wildly and you might end up with memory hog components that are just unusable.
  - E.g., the source language identification piece I just showed (wink)

NLTK vs The World

- GATE
- UIMA

Where to go from here

- NLTK is **awesome**
- Give it a try...
- Read the book on-line or buy it to support the project

- We can do some NLTK related sprints.
- I will most likely do a NLTK hands on tutorial covering more material at Foulab.
- Ping me if you like to chat about NLP, particularly the generation bit.