Basic NLTK Tutorial

The Natural Language Tool Kit (NLTK) is the most popular NLP library for python.

Now let's walk through some simple NLTK use cases that concern this assignment. For that, you will need first to open a python interactive shell. Just type `python` on the command line and you should see the interactive shell start.

Import the NLTK package

To use NLTK package, you must include the following line at the beginning of your code (or in this case just type in the interactive shell):

```
import nltk
```

Tokenization

To tokenize means to break a continuous string into tokens (usually words, but a token could also be a symbol, punctuation, or other meaningful unit). In NLTK, text can be tokenized using the `word_tokenize()` method. It returns a list of tokens that will be the input for many methods in NLTK.

```
sentence = "At eight o'clock on Thursday morning on Thursday morning on Thursday morning."
tokens = nltk.word_tokenize(sentence)
```

N-grams Generation

An n-gram (in the context of this assignment) is a contiguous sequence of n tokens in a sentence. The following code returns a list of bigrams and a list of trigrams. Each n-gram is represented as a tuple in python (if you are not familiar with python tuples read the python tuple doc page)

```
bigram_tuples = list(nltk.bigrams(tokens))
trigram_tuples = list(nltk.trigrams(tokens))
```

We can calculate the count of each n-gram using the following code:

```
count = {item : bigram_tuples.count(item) for item in set(bigram_tuples)}
```

Or we can find all the distinct n-grams that contain the word "on":

```
ngrams = [item for item in set(bigram_tuples) if "on" in item]
```

If you find it hard to understand the examples above, read about list/dict comprehensions here. List/dict comprehensions are a way of executing iterations in one line that may be very useful and convenient. Besides making coding easier, learning them will also help you understand code written by other python programmers.

Default POS Tagger (Non-statistical)

The most naïve way of tagging parts-of-speech is to assign the same tag to all the tokens. This is exactly what the NLTK default tagger does. Although inaccurate and arbitrary, it sets a baseline for taggers, and can be used as a default tagger when more sophisticated methods fail.

In NLTK, it's easy to create a default tagger by indicating the default tag in the constructor.

```
default_tagger = nltk.DefaultTagger('NN')
tagged_sentence = default_tagger.tag(tokens)
```

Now we have our first tagger. NLTK can help if you need to understand the meaning of a tag.

```
# Show the description of the tag 'NN'
nltk.help.upenn_tagset('NN')
```

Regular Expression POS Tagger (Non-statistical)

A regular expression tagger maintains a list of regular expressions paired with a tag (see the Wikipedia article for more information about regular expressions: http://en.wikipedia.org/wiki/Regular_expression). The tagger tries to match each token to one of the regular expressions in its list; the token receives the tag that is paired with the first matching regular expression. "None" is given to a token that does not match any regular expression.

To create a Regular Expression Tagger in NLTK, we provide a list of pattern-tag pairs to the appropriate constructor. Example:

```
patterns = [(r'.*ing$', 'VBG'),(r'.*ed$', 'VBD'),(r'.*es$', 'VBZ'),(r'.*ed$', 'VB')]
regexp_tagger = nltk.RegexpTagger(patterns)
regexp_tagger.tag(tokens)
```

N-gram HMM Tagger (Statistical)

Although there are many different kinds of statistical taggers, we will only work with Hidden Markov Model (HMM) taggers in this assignment.

Like every statistical tagger, n-gram taggers use a set of tagged sentences, known as the training data, to create a model that is used to tag new sentences. In NLTK, a sentence of the training data must be formatted as a list of tuples, where each tuple is a pair or word-tag (see example below).

```
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL')]
```

NLTK already provides corpora formatted this way. In particular, we are going to use the Brown corpus.

```
# import the corpus from NLTK and build the training set from sentences in "news"
from nltk.corpus import brown
training = brown.tagged_sents(categories='news')

# Create Unigram, Bigram, Trigram taggers based on the training set.
unigram_tagger = nltk.UnigramTagger(training)
bigram_tagger = nltk.BigramTagger(training)
trigram_tagger = nltk.TrigramTagger(training)
```

Although we could also build 4-gram, 5-gram, etc. taggers, trigram taggers are the most popular model. This is because a trigram model is an excellent compromise between computational complexity and performance.

Combination of Taggers

A tagger fails when it cannot find a best tag sequence for a given sentence. For example, one situation when an n-gram tagger will fail is when it encounters an OOV (out of vocabulary) word not seen in the training data: the tagger will tag the word as "NONE". One way to handle tagger failure is to fall back to an alternative tagger if the primary one fails. This is called "using back off." One can easily set a hierarchy of taggers in NLTK as follows.

```
default_tagger = nltk.DefaultTagger('NN')
bigram_tagger = nltk.BigramTagger(training, backoff=default_tagger)
trigram_tagger = nltk.TrigramTagger(training, backoff=bigram_tagger)
```

Tagging Low Frequency Words

Low frequency words are another common source of tagger failure, because an n-gram that contains a low frequency word and is found in the test data might not be found in the training data. One method to resolve this tagger failure is to group low frequency words. For example, we could substitute the token "_RARE_" for all words with frequency lower than 0.05% in the training data. Any words in the development data that were not found in the training data could then be treated instead as the token "_RARE_", thereby allowing the algorithm to assign a tag. If we wanted to add another group we could substitute the string "_NUMBER_" for those rare words that represent a numeral. When tagging the test data, we could substitute "_NUMBER_" for all tokens that were unseen in the training data and represent a numeral. We will use this technique later in this assignment.