NLP Lab Session Week 5
September 29, 2017

POS taggers in NLTK

Getting Started

For this lab sessioin, download the examples: LabWeek5POStags.ipynb or LabWeek5POStags.py

Save the file in a folder where you keep materials for this class. Open your command prompt or terminal window and use the cd command to change directory to your class materials folder. Type at the prompt:

```
$ jupyter notebook
```

Start your nlp session by:

```
>>> import nltk
```

Reading Tagged Corpora

The NLTK corpus readers have additional methods (aka functions) that can give the additional tag information from reading a tagged corpus. Both the Brown corpus and the Penn Treebank corpus have text in which each token has been tagged with a POS tag. (These were manually assigned by annotators.)

The tagged_sents function gives a list of sentences, each sentence is a list of (word, tag) tuples. We'll first look at the Brown corpus, which is described in Chapter 2 of the NLTK book.

```
import nltk
from nltk.corpus import brown
brown.tagged_sents()[:2]
```

The tagged_words function just gives a list of all the (word, tag) tuples, ignoring the sentence structure.

```
brown.tagged_words()[:50]
```

We said that each of these is what Python calls a tuple, which is a pair (or triple, etc.) in which you can't change the elements.

```
>>> wordtag = brown.tagged_words()[0]
>>> wordtag
('The', 'AT')
```

```
>>> type(wordtag)
<type 'tuple'>
>>> wordtag[0]
'The'
>>> wordtag[1]
'AT'
```

The Brown corpus is organized into different types of text, which can be selected by the categories argument, and it also allows you to map the tags to a simplified tag set, described in table 5.1 in the NLTK book.

```
brown.categories()
brown_humor_tagged = brown.tagged_words(categories='humor', tagset='universal')
brown_humor_tagged[:50]
```

Other tagged corpora also come with the tagged_words method. Note that the chat corpus is tagged with Penn Treebank POS tags.

```
nltk.corpus.nps_chat.tagged_words()[:50]
```

Penn Treebank

In this class, we will mostly use the Penn Treebank tag set, as it is the most widely used. The Treebank has the tagged_words and tagged_sents methods, as well as the words method that we used before to get the tokens.

```
from nltk.corpus import Treebank
# the .raw() and .words() functions still get the text as strings and as tokens
treebank_text = treebank.raw()
print(treebank_text[:50], '\n')
treebank_tokens = treebank.words()
print(treebank_tokens[:20])

# but we also have functions to get words with tags and sentences with tagged words
treebank_tagged_words = treebank.tagged_words()[:50]
len(treebank.tagged_words())
treebank_tagged_words[:50]

treebank_tagged = treebank.tagged_sents()[:2]
len(treebank.tagged_sents())
treebank_tagged[:2]
```

The NLTK has almost 4,000 sentences of tagged data from Penn Treebank, while the actual Treebank has much more. This will limit the accuracy of the POS taggers (and

later parsers) that we can define in lab, but also make the running times short enough for labs.

Let's look at the frequencies of the tags in this portion of Penn Treebank. To do that, we use the NLTK Frequency Distribution for all the tags from the (word, tag) pairs in the Treebank.

```
tag_fd = nltk.FreqDist(tag for (word, tag) in treebank_tagged_words)
print(tag_fd.keys(), '\n')
for tag,freq in tag_fd.most_common():
   print (tag, freq)
```

We see that NN, the tag of single nouns, is the most frequent tag; it has 13,166 occurrences of the 100,676 words, or about 13%. The tags IN, for prepositions except to, NNP, for single proper nouns, and DT, for determiners, are close behind at 10%, 9% and 8%, respectively. The next tag in the list is –NONE-, which is the tag of those empty elements, which come from the grammar syntactic constructs.

This is a very detailed look at the POS tags. We could also approximate classes of tags by using the first letter of the POS tag as the key in a frequency distribution. For example, this will group all the nouns, which all start with "N", all the verbs, "V", adjectives "J", and adverbs "R". The prepositions will be split between "I" for the ones that are tagged "IN" and "T" for the ones that are tagged "TO".

```
# use the first letter of the POS tag to get classes of tags
tag_classes_fd = nltk.FreqDist(tag[0] for (word, tag) in treebank_tagged_words)
print(tag_classes_fd.keys(), '\n')
for tag,freq in tag_classes_fd.most_common():
   print (tag, freq)
```

POS Tagging

The process of classifying words into their parts of speech and labeling them accordingly is known as part-of-speech tagging, POS-tagging, or simply tagging. Parts of speech are also known as word classes or lexical categories. The collection of tags used for a particular task is known as a tagset.

We will use the tagged sentences and words from the Penn Treebank that we defined in the previous section.

We separate our tagged data into a training set, where we'll learn the probabilities of the words and their tags, and a test set to evaluate how out taggers perform. This allows us to test the tagger's accuracy on similar, but not the same, data that it was trained on. The training set is the first 90% of the sentences and the test set is the remaining 10%.

```
size = int(len(treebank_tagged) * 0.9)
treebank_train = treebank_tagged[:size]
treebank_test = treebank_tagged[size:]
```

In the NLTK, a number of POS taggers are included in the tag module, including one that we can use that has been trained on all of Penn Treebank. But for instructional purposes, we will develop a sequence of N-gram taggers whose performance improves.

To introduce the N-gram taggers in NLTK, we start with a default tagger that just tags everything with the most frequent tag: NN. We create the tagger and run it on text. Note that this simple tagger doesn't actually use the training set.

```
# creates the tagger
t0 = nltk.DefaultTagger('NN')
# show the effect of the tagger by tagging the first 50 words
t0.tag(treebank_tokens[:50])
```

The NLTK includes a function for taggers that computes tagging accuracy by comparing the result of a tagger with the original "gold standard" tagged text. Here we use the NLTK function "evaluate" to apply the default tagger (to the untagged text) and compare it with the gold standard tagged text in the test set.

```
t0.evaluate(treebank_test)
```

The evaluate function first takes the tagged text and removes the tags, so that only tokens are left. Then it runs the tagger, in this case t0, to tag all the text. Then it compares the tags predicted by the tagger with the "gold standard" tags already given. It reports the accuracy, which is the percentage of words with correct tags.

Other simple taggers described in the NLTK book are the Regular Expression Tagger and the Lookup Tagger.

Next we train a Unigram tagger. It tags each word with the most frequent tag in that word has in the corpus. For example, if the word "bank" occurs 30 times with the tag "NN" and 10 times with the tag "VB", we'll just tag it with "NN".

```
t1 = nltk.UnigramTagger(treebank_tagged)
t1.tag(treebank_tokens[:50])
```

Train the tagger on the training set and evaluate on the test set.

```
t1 = nltk.UnigramTagger(treebank_train)
t1.evaluate(treebank_test)
```

In the lecture slides, this Unigram Tagger is what Chris Manning called their baseline tagger and they got about 90% accuracy. Why isn't ours quite as good?

Finally, NLTK has a Bigram tagger that can be trained using 2 tag-word sequences. But there will be unknown frequencies in the test data for the bigram tagger, and unknown words for the unigram tagger, so we can use the backoff tagger capability of NLTK to create a combined tagger. This tagger uses bigram frequencies to tag as much as possible. If a word doesn't occur in a bigram, it uses the unigram tagger to tag that word. If the word is unknown to the unigram tagger, then we use the default tagger to tag it as 'NN'.

```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(treebank_train, backoff=t0)
t2 = nltk.BigramTagger(treebank_train, backoff=t1)
t2.evaluate(treebank_test)
```

This accuracy is not bad, especially on only part of Penn Treebank! We know that HMM and other feature techniques can raise the accuracy to between 95 and 98%.

But note that this good performance is also on a test set taken from the Penn Treebank, where there may not be very many unknown words. More modern text or text on different topics than the Wall Street Journal will have more difficulty with unknown words. We know from the lectures that combining this with a regular expression tagger or a classifier tagger can improve performance on unknown words.

Let's use this tagger to tag some example text. Define some example text, tokenize it, and apply the tagger.

```
text = "Three Calgarians have found a rather unusual way of leaving snow and ice behind. They set off this week on foot and by camels on a grueling trek across the burning Arabian desert."
```

In previous labs, we applied the function "nltk.word_tokenize" directly to multi-sentence text for simplicity. But this function is actually trained to tokenize individual sentences and will work better if we first use the sentence splitter, aka tokenizer, to produce a list of text strings for individual sentences.

```
>>> textsplit = nltk.sent_tokenize(text)
>>> textsplit
```

After producing the list of sentence texts, apply the word tokenizer to each sentence.

```
>>> tokentext = [nltk.word_tokenize(sent) for sent in textsplit]
>>> tokentext
```

Now apply the t2 bigram POS tagger to each sentence of tokens in the list.

```
>>> taggedtext = [t2.tag(tokens) for tokens in tokentext]
```

```
>>> taggedtext
```

We observe that this text has quite a few words that appear to be unknown to this tagger from the data it was trained on. Examples of this are "Calgarians" and "camels". In both cases, these two words are tagged as NN instead of the correct tags of NNPS and NNS, respectively. This points out the benefit of adding sequence information such as an HMM tagger would use and lexical information, such as a Maximum Entropy tagger could use if you defined such features. In the NLTK, another strategy would be to use a Regular Expression tagger as a backoff tagger that could take into account word features.

Stanford POS Tagger

One of the problems with training our own POS tagger is that we don't have all the Penn Treebank data. But NLTK also provides some taggers that come pre-trained on the larger amount of data. One of these is the Stanford POS tagger, which was trained using a maximum entropy classifier. This is described in the nltk.tag module:

http://www.nltk.org/_modules/nltk/tag.html

This tagger is available in the module:
'taggers/maxent_treebank_pos_tagger/english.pickle' and it is used for the standard nltk.pos_tag function.

```
>>> taggedtextStanford = [nltk.pos_tag(tokens) for tokens in tokentext]
>>> taggedtextStanford
```

Lab Exercise

Choose a corpus from any of the NLTK books or Gutenberg. Tokenize the corpus and run a POS tagger, either the Bigram tagger t2 that we defined in lab, or the Stanford tagger, nltk.pos_tag, on the tokens.

Post the frequencies of either the POS tags or the classes of POS tags from your corpus.

Note that you should

- split the raw text into sentences,
- tokenize the list of sentences
- then run the tagger on the list of sentences, where the result is a list of lists of (word, tag) pairs
- flatten the list to get just one list of (word, tag)
- put either the tags or the first letter of the tags into a Frequency Distribution and print those.

Report the corpus name, tokenizer, pos tagger and frequencies of the tags or classes of tags in the Discussion for this week. Note any significant differences in the frequent tags

or tag classes between your corpus and the Penn Treebank tags that we ran in the first part of the lab.