In [1]:
```
import nltk
import tagutils; reload(tagutils)
from tagutils import *
from IPython.core.display import HTML
from nltk.corpus import brown
import random as pyrand
from tagutils import *
```

Training and Test Sentences

In previous worksheets, we looked at applying taggers to the analysis of texts.

We already noted that taggers are often trained.

Let's look in more detail at how taggers work.

We start off by defining a `training_set` and a `test_set`.

In [2]:
```
sents = list(brown.tagged_sents())
pyrand.shuffle(sents)
training_set = sents[:-1000]
test_set = sents[-1000:]
```

In [38]:
```
HTML(mstags(test_set[:5]))
```

Out[38]:

`` So you're looking for a woman who married a man who might have lived here a year ago and might have been poisoned .

Five days later , on receiving it , Meredith sat drumming his dactyls on his writing table .

The Medical Museum

The combination proved quite irresistible last night .

Bullets began to snap past him .

Unigram Tagging

There are a variety of tools for keeping statistics.

`FreqDist` counts the number of occurrences of its arguments and returns them in order of decreasing frequency.

In [39]:
```
fd = nltk.FreqDist([w[0] for s in training_set for w in s])
fd.items()[:10]
```

Out[39]:
```
[('the', 61543),
 (',', 57205),
 ('.', 48490),
 ('of', 35404),
 ('and', 27399),
 ('to', 25268),
 ('a', 21449),
 ('in', 19152),
 ('that', 10051),
 ('is', 9826)]
```

`ConditionalFreqDist` computes statistics over pairs `(A,B)` and can be used to estimate $P(B|A)$.

Note that `cfd[word]` is essentially a posterior probability distribution.

In [4]:
```
cfd = nltk.ConditionalFreqDist([w for s in training_set for w in s])
cfd["frequent"].items()[:10]
```

Out[4]:
```
[('JJ', 32), ('VB', 2)]
```

For a unigram model, we compute

$$\text{tag}(\text{word}) = \arg\max P(\text{tag}|\text{word})$$

In [5]:
```
likely_tags = dict((w, cfd[w].max()) for w in fd.keys())
likely_tags.items()[:10]
```

Out[5]:
```
[('Ranavan', 'NP-TL'),
 ('fawn', 'NN'),
 ('gai', 'FW-JJ'),
 ('mid-week', 'NN'),
 ('1,800', 'CD'),
 ('deferment', 'NN'),
 ('Debts', 'NNS-TL'),
 ('Poetry', 'NN'),
 ('woods', 'NNS'),
 ('clotted', 'VBN')]
```

We can now construct a simple tagger. This tagger just assigns the most frequent tag from the training set to each word.

In [6]:
```
baseline_tagger = nltk.UnigramTagger(model=likely_tags)
```

We can now evaluate performance on the test set; the error rate of this "baseline tagger" is about 10.3%.

In [7]:
```
baseline_tagger.evaluate(test_set)
```

Out[7]:
```
0.8969654199011997
```

Note that performance on the training set is, as usual, better than on the test set. We've already seen this phenomenon in other pattern recognition systems.

In [8]:
```
baseline_tagger.evaluate(training_set[:1000])
```

Out[8]: 0.9264958945867545

n-Gram Taggers

Unigram taggers can also be trained and evaluated directly.

In [9]:
```
unigram_tagger = nltk.UnigramTagger(training_set)
```

In [10]:
```
unigram_tagger.evaluate(test_set)
```

Out[10]: 0.8969654199011997

A generalization of unigram taggers are n-gram taggers.

A bigram tagger computes tags as:

$$\mathrm{T(W)} = \arg\max_{W} P(T|W, W_{\mathrm{prev}})$$

In [11]:
```
bigram_tagger = nltk.BigramTagger(training_set)
```

In [12]:
```
bigram_tagger.evaluate(test_set)
```

Out[12]: 0.32364149611856036

In [13]:
```
bigram_tagger.evaluate(training_set[:1000])
```

Out[13]: 0.8145926545061213

In [14]:
```
trigram_tagger = nltk.TrigramTagger(training_set)
```

In [15]:
```
trigram_tagger.evaluate(test_set)
```

Out[15]: 0.16287932251235004

In [16]:
```
trigram_tagger.evaluate(training_set[:1000])
```

Out[16]: 0.7679826933477556

Note that for bigram and trigram taggers, performance in the test set is really bad.

The reason is that a lot of the contexts (word sequences) that the bigram and trigram taggers are based on don't exist in the training set.

Backoff Models

To deal with lack of training data for longer contexts, we can use *backoff models*.

That is, we first try to assign tags using a complicated model, but if the context is missing, we revert to a smaller, simpler model.

This improves the error rate from 10.3% to 7.8%

In [17]:
```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(training_set, backoff=t0)
t2 = nltk.BigramTagger(training_set, backoff=t1)
t2.evaluate(test_set)
```

Out[17]: 0.9229828275699835

More context does not actually help.

In [18]:
```
t3 = nltk.TrigramTagger(training_set,backoff=t2)
t3.evaluate(test_set)
```

Out[18]: 0.9222300635144672

Intrinsic Error Rate

We already talked about intrinsic error rates in the context of Bayesian classification.

For n-gram taggers, we can compute the error rates more directly.

Consider $P(T|W_n, W_{n-1}, W_{n-2})$.

The rate of correct tagging for a given context is given by $\max_T P(T|W_n, W_{n-1}, W_{n-2})$.

The intrinsic error rate for a context is therefore $1 - \max_T P(T|W_n, W_{n-1}, W_{n-2})$.

If we sum this over all contexts and their probabilities, we get the overall intrinsic error rate.

In [19]:
```
cfd = nltk.ConditionalFreqDist(
           ((x[1], y[1], z[0]), z[1])
           for sent in training_set
           for x, y, z in nltk.trigrams(sent))
```

In [20]:
```
cfd.conditions()[30000:30003]
```

Out[20]: [('ABN', 'CC', 'bear'), ('ABN', 'CC', 'dispensing'), ('ABN', 'CC', 'had')]

In [21]:
```
[cfd[cfd.conditions()[i]].items() for i in range(30000,30005)]
```

Out[21]:
```
[[('VB', 1)], [('VBG', 1)], [('HVD', 1)], [('IN', 1)], [('RB', 1)]]
```

In [22]:
```
ambiguous_contexts = [c for c in cfd.conditions() if len(cfd[c]) > 1]
```

In [23]:
```
print ambiguous_contexts[5000]
print cfd[ambiguous_contexts[5000]].items()
```

```
('NNS', ',', 'leaves')
[('NNS', 1), ('VBZ', 1)]
```

In [24]:
```
sum(cfd[c].N() for c in ambiguous_contexts) *1.0 / cfd.N()
```

Out[24]:
```
0.11594425738142042
```

In [25]:
```
def words(s): return [w for w,t in s]
bigram_tagger.tag(words(test_set[0]))
```

Out[25]:
```
[('``', '``'),
 ('So', 'RB'),
 ("you're", 'PPSS+BER'),
 ('looking', 'VBG'),
 ('for', 'IN'),
 ('a', 'AT'),
 ('woman', 'NN'),
 ('who', 'WPS'),
 ('married', None),
 ('a', None),
 ('man', None),
 ('who', None),
 ('might', None),
 ('have', None),
 ('lived', None),
 ('here', None),
 ('a', None),
 ('year', None),
 ('ago', None),
 ('and', None),
 ('might', None),
 ('have', None),
 ('been', None),
 ('poisoned', None),
 ('.', None)]
```

In [26]:
```
predicted_tags = [tag for sent in test_set for (word, tag) in t2.tag(words(sent))]
true_tags = [tag for sent in test_set for (word, tag) in sent]
confusion = nltk.ConfusionMatrix(true_tags, predicted_tags)
```

In [27]:
```
with open("confusion.txt","w") as stream: stream.write(str(confusion))
```

In [28]:
```
!gvim -c ':set nowrap' confusion.txt
```

In [29]:
```
from collections import Counter
conf = Counter(zip(predicted_tags,true_tags))
conf.most_common(10)
```

Out[29]:
```
[(('NN', 'NN'), 2707),
 (('IN', 'IN'), 2076),
 (('AT', 'AT'), 1847),
 ((',', ','), 1126),
 (('.', '.'), 1052),
 (('JJ', 'JJ'), 1021),
 (('NNS', 'NNS'), 913),
 (('CC', 'CC'), 678),
 (('RB', 'RB'), 574),
 (('NP', 'NP'), 572)]
```

In [30]:
```
[((u,v),n) for (u,v),n in conf.most_common() if u!=v][:10]
```

Out[30]:
```
[(('TO', 'IN'), 124),
 (('NN', 'JJ'), 99),
 (('NN', 'NP'), 79),
 (('VBN', 'VBD'), 70),
 (('IN', 'TO'), 69),
 (('NN', 'VB'), 67),
 (('NN', 'NNS'), 67),
 (('VBD', 'VBN'), 66),
 (('NP', 'NP-TL'), 33),
 (('JJ', 'NN'), 31)]
```

In [31]:
```

```