Florian Diebold
Markus Fuchs <mfuchs13@googlemail.com>

In [2]:
```
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 *
```

Evaluation Framework

In [3]:
```
sents = list(brown.tagged_sents())
n = len(sents)
test = sorted(list(set(range(0,n,10))))
training = sorted(list(set(range(n))-set(test)))
training_set = [sents[i] for i in training]
test_set = [sents[i] for i in test]
```

In [4]:
```
print len(training_set)
print len(test_set)
```

```
51606
5734
```

In [5]:
```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(training_set, backoff=t0)
t2 = nltk.BigramTagger(training_set, backoff=t1)
t2.evaluate(test_set)
```

Out[5]: 0.9234475055210225

Wordnet-Based Improvements

In [6]:
```
import nltk.tag.api
#help(nltk.tag.api.TaggerI)
```

Your homework consists of implementing new taggers based on Wordnet. With regular taggers, we have a problem of sparsity; that is, we don't know what tag to assign to a word if we have never seen it in a context.

However, for many words, Wordnet may give us useful information to help with tagging. You need to work out some ideas, implement them, and test them.

There are different implementation strategies, but a simple one might be:

- write classes that map token sequences to other token sequences using WordNet; for example, you might map an input sentence to some collection of hyponyms
- then, apply the regular NLTK n-gram taggers to the modified output sequences
- use backoff (as above) when the WordNet mapping fails for some reason (you can't find the word, or maybe the mapping would be ambiguous and you don't know how to handle it)

This may not be the best strategy, but it's a good way of getting started.

Another strategy is to use use WordNet to generate a cloud of related words around a given word, and then see whether you can find bigrams in an existing model for any of the related words.

Implement your model(s) so that they conform to the NLTK tagging APIs, perform evaluations on the training and test sets defined above, and be ready to present your results (idea, evaluation, results) in the exercises.

In [7]:
```
from nltk.corpus import wordnet as wn
```

In [8]:
```
def tag_fits(wn_tag, brown_tag):
    if wn_tag == wn.NOUN:
        return brown_tag.startswith("N")
    elif wn_tag == wn.ADJ or wn_tag == wn.ADJ_SAT:
        return brown_tag.startswith("JJ")
    elif wn_tag == wn.ADV:
        return brown_tag.startswith("RB")
    elif wn_tag == wn.VERB:
        return brown_tag.startswith("VB")
```

In [9]:
```
from nltk.corpus import wordnet as wn
from nltk.tag import ContextTagger
from nltk.probability import FreqDist, ConditionalFreqDist

class SimpleWordnetTagger(ContextTagger):
    def __init__(self, n, train=None, cutoff=0, backoff=None, verbose=False):
        ContextTagger.__init__(self, None, backoff)
        self._n = n
        self.enabled = True

        if train:
            self._train(train, cutoff, verbose)

    def _train(self, tagged_corpus, cutoff=0, verbose=False):
        # This is almost completely copied from ContextTagger,
        # except only tags that fit the respective wordnet tag
        # are used for training (see tag_fits above)

        token_count = hit_count = 0

        useful_contexts = set()

        fd = ConditionalFreqDist()
        for sentence in tagged_corpus:
            tokens, tags = zip(*sentence)
            for index, (token, tag) in enumerate(sentence):
                token_count += 1
                context = self.context(tokens, index, tags[:index])
                if context is None: continue
                # don't use this tag if it doesn't fit the established wordnet POS
                if not tag_fits(context[1], tag): continue
                fd[context].inc(tag)
                if (self.backoff is None or
                    tag != self.backoff.tag_one(tokens, index, tags[:index])):
                    useful_contexts.add(context)

        for context in useful_contexts:
            best_tag = fd[context].max()
            hits = fd[context][best_tag]
            if hits > cutoff:
                self._context_to_tag[context] = best_tag
                hit_count += hits

        if verbose:
            size = len(self._context_to_tag)
            backoff = 100 - (hit_count * 100.0)/ token_count
            pruning = 100 - (size * 100.0) / len(fd.conditions())
            print "[Trained Wordnet tagger: ",
            print "size=%d, backoff=%.2f%%, pruning=%.2f%%]" % (size, backoff, pruni

    def context(self, tokens, index, history):
        # Look for a Wordnet synset and use its POS for tagging
        if not self.enabled: return None
        tag_context = tuple(history[max(0, index-self._n+1):index])
        ss = wn.synsets(tokens[index])
        if len(ss) == 0: return None
        s = ss[0]
        return (tag_context, s.pos)
```

In [10]:
```
t0 = nltk.DefaultTagger('NN')
wt1 = SimpleWordnetTagger(1, training_set, backoff=t0, verbose=True)
wt2 = SimpleWordnetTagger(2, training_set, backoff=t0, verbose=True)
t1 = nltk.UnigramTagger(training_set, backoff=wt2, verbose=True)
t3 = nltk.BigramTagger(training_set, backoff=t1, verbose=True)

print t3.evaluate(test_set)
```

```
[Trained Wordnet tagger:  size=5, backoff=80.82%, pruning=0.00%]
[Trained Wordnet tagger:  size=769, backoff=77.52%, pruning=2.66%]
[Trained Unigram tagger: size=35636, backoff=14.91%, pruning=33.23%]
[Trained Unigram tagger: size=21958, backoff=74.31%, pruning=87.49%]
0.925451274312
```