Christian Maxeiner, christian.maxeiner@hotmail.de
Christian Stahl, stahlchr@gmail.com

In [29]:
```
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 [30]:
```
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 [31]:
```
print len(training_set)
print len(test_set)
print test_set[0]
print size(training_set)
```

```
51606
5734
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'),
('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'),
('investigation', 'NN'), ('of', 'IN'), ("Atlanta's", 'NP$'), ('recent', 'JJ'),
('primary', 'NN'), ('election', 'NN'), ('produced', 'VBD'), ('``', '``'), ('no',
'AT'), ('evidence', 'NN'), ("''", "''"), ('that', 'CS'), ('any', 'DTI'),
('irregularities', 'NNS'), ('took', 'VBD'), ('place', 'NN'), ('.', '.')]
51606
```

In [32]:
```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(training_set, backoff=t0)
t2 = nltk.BigramTagger(training_set, backoff=t1)
t2.evaluate(test_set)
```

Out[32]:
```
0.9234475055210225
```

In [33]:
```
print t2.tag(['house'])
```

```
[('house', 'NN')]
```

Classifier-Based Tagging

In [34]:
```
import nltk.tag.api
help(nltk.tag.api.TaggerI)
```

```
Help on class TaggerI in module nltk.tag.api:

class TaggerI(__builtin__.object)
 |  A processing interface for assigning a tag to each token in a list.
 |  Tags are case sensitive strings that identify some property of each
 |  token, such as its part of speech or its sense.
 |
 |  Some taggers require specific types for their tokens.  This is
 |  generally indicated by the use of a sub-interface to ``TaggerI``.
 |  For example, featureset taggers, which are subclassed from
 |  ``FeaturesetTagger``, require that each token be a ``featureset``.
 |
 |  Subclasses must define:
 |    - either ``tag()`` or ``batch_tag()`` (or both)
 |
 |  Methods defined here:
 |
 |  batch_tag(self, sentences)
 |      Apply ``self.tag()`` to each element of *sentences*.  I.e.:
 |
 |          return [self.tag(sent) for sent in sentences]
 |
 |  evaluate(self, gold)
 |      Score the accuracy of the tagger against the gold standard.
 |      Strip the tags from the gold standard text, retag it using
 |      the tagger, then compute the accuracy score.
 |
 |      :type gold: list(list(tuple(str, str)))
 |      :param gold: The list of tagged sentences to score the tagger on.
 |      :rtype: float
 |
 |  tag(self, tokens)
 |      Determine the most appropriate tag sequence for the given
 |      token sequence, and return a corresponding list of tagged
 |      tokens.  A tagged token is encoded as a tuple ``(token, tag)``.
 |
 |      :rtype: list(tuple(str, str))
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Implement a new tagger based on classifiers.

When applying a classifier, you need to transform the input into a feature vector. In this case, we are trying to predict $P(t_n| < \text{input words} >)$. How do we do this?

For a simple unigram tagger, we are estimating $P(t_n|w_n)$. If $w_n \in V = \{1, \ldots, N\}$, where V is a vocabulary of size N representing each word as an integer, then the input feature vector might be a binary vector $\vec{x} = (x_1 \ldots x_N)$ where

$x_i = \delta_{i,w_n}$

For a simple bigram tagger, we are estimating something like $P(t_n|w_n t_{n-1})$, which we could similarly represent as a concatenation of two large binary input vectors.

However, such a brute force approach may not work very well because we have a very high dimensional input vector and classifiers often need a lot of training data. We are free to preprocess the data in any form we like in order to get better feature vectors.

Here are some ideas:

- use the posterior probabilities for tags returned by a unigram and bigram tagger as feature vectors
- use possible grammatical categories and semantic categories from Wordnet as feature vectors
- use simple features like capitalization, word length, and position in sentence
- provide information about word frequency in input
- "hash" the large range of possible words V down to a much smaller vocabulary
- same as before, but do the hasing somewhat more intelligently: leave all the stop words alone, but has down the content words
- do the "hashing" in some way that's informed by Wordnet

Note that in order to be able to tag using the algorithms we have described, you can use tags assigned to previous words, but you cannot use tags assigned to subsequent words.

Try to beat the bigram-with-backoff tagger above, using the same evaluation paradigm. Your tagger should implement the standard NLTK tagging API.

Two classifiers to try are logistic regression and decision tree classifiers. You can use implementations from the `sklearn` package.

First, import some useful stuff.

In [35]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from nltk.corpus import wordnet
from collections import Counter,defaultdict
```

Because the NLTK unigram and bigram taggers only return single tags withot probabilities, create a general tagger class that returns probabilities.

In [36]:
```
class ProbCounterTagger:
    def __init__(self):
        self.counts = defaultdict(Counter)
    def addTrainingData(self, td):
        for token,tag in td:
            self.counts[token].update([tag])
    def train(self):
        self.probs = {}
        for token,tagcounts in self.counts.items():
            n = sum(tagcounts.values())
            self.probs[token] = {tag:count*1.0/n for tag,count
                in tagcounts.items()}
    def getProbs(self, token):
        if token in self.probs:
            return self.probs[token]
        return {}
```

Define some methods to create different bigrams from sentences.

In [37]:
```
def make_bigrams(sent):
    sent = ['']+sent
    return zip(sent,sent[1:])

def make_tagged_bigrams(tagged_sent):
    return zip(make_bigrams([w for w,t in tagged_sent]),
        [t for w,t in tagged_sent])

def make_bigrams_with_tag(tagged_sent):
    shifted_tags = [''] + [t for w,t in tagged_sent]
    return zip(shifted_tags, [w for w,t in tagged_sent])

def make_tagged_bigrams_with_tag(tagged_sent):
    return zip(make_bigrams_with_tag(tagged_sent),
        [t for w,t in tagged_sent])

print make_bigrams(['Hi','here','I','am','.'])
print make_tagged_bigrams(training_set[0][:5])
print make_tagged_bigrams_with_tag(training_set[0][:5])
```

```
[('', 'Hi'), ('Hi', 'here'), ('here', 'I'), ('I', 'am'), ('am', '.')]
[(('', 'The'), 'AT'), (('The', 'jury'), 'NN'), (('jury', 'further'), 'RBR'),
(('further', 'said'), 'VBD'), (('said', 'in'), 'IN')]
[(('', 'The'), 'AT'), (('AT', 'jury'), 'NN'), (('NN', 'further'), 'RBR'),
(('RBR', 'said'), 'VBD'), (('VBD', 'in'), 'IN')]
```

Define a general classifier tagger class which trains on the probabilities returned by Unigram and Bigram Taggers.

Explanation of the attributes:

trainingset: A list of tagged sentences to train.

use_also_bigrams: If true, not only the probabilities returned from a Unigram Tagger are used as training data, but also the probabilities returned by a Bigram Tagger.

combine_ngram_tagger_results: Only used if use_also_bigrams is true. If this is true, the probabilities of the Unigram and Bigram Tagger are not not concatenated into one big feature vector, but the two featurevectors are added component-wise.

use_prev_tag: Only used if use_also_bigrams is true. If this is true, bigrams are built like (previous tag + actual word), and not like (previous word + actual word).

sklearn_classifier: A classifier-class from sklearn which should be used for training and tagging.

In [38]:
```
class ClassifierTagger1(nltk.tag.api.TaggerI):
    def __init__(self, trainingset, use_also_bigrams,
        combine_ngram_tagger_results, use_prev_tag,
        sklearn_classifier):
        self.use_also_bigrams = use_also_bigrams
        self.combine_ngram_tagger_results=combine_ngram_tagger_results
        self.use_prev_tag = use_prev_tag

        # train a probability tagger with unigrams
        self.unigram_tagger = ProbCounterTagger()
        for sent in trainingset:
            self.unigram_tagger.addTrainingData(sent)
        self.unigram_tagger.train()

        # eventually also train a probability tagger with bigrams
        if(use_also_bigrams):
            self.bigram_tagger = ProbCounterTagger()
            for sent in trainingset:
                if self.use_prev_tag:
                    self.bigram_tagger.addTrainingData(
                        make_tagged_bigrams_with_tag(sent))
                else:
                    self.bigram_tagger.addTrainingData(
                        make_tagged_bigrams(sent))
            self.bigram_tagger.train()

        # create list of possible tags
        tagged_words = []
        for sent in trainingset:
            tagged_words += sent
        self.taglist = list(set([t for w,t in tagged_words]))

        # train a classifier with the tagger probabilities
        classifier_training = []
        classifier_target = []
        for sent in trainingset:
            bigs = make_tagged_bigrams(sent)
            for i in range(len(bigs)):
                ((prev,word),tag) = bigs[i]
                if self.use_prev_tag:
                    if i == 0:
                        fv = self.getFeatureVector('', word)
                    else:
                        fv = self.getFeatureVector(bigs[i-1][1], word)
                else:
                    fv = self.getFeatureVector(prev,word)
                classifier_training.append(fv)
                classifier_target.append(self.taglist.index(tag))
        self.classifier = sklearn_classifier()
        self.classifier.fit(classifier_training, classifier_target)

    # A feature vector is created from Unigram Tagger probabilities:
    # In a feature vector as long as the list with possible tags,
    # the according places are filled with the probabilities returned
    # from the Unigram Tagger.
    def getFeatureVectorUnigram(self, token):
        probs = self.unigram_tagger.getProbs(token)
        fv = zeros(len(self.taglist))
        for tag in probs:
            fv[(array(self.taglist))==tag] = probs[tag]
        return fv
```

First try how the classfier-based tagger performs with logistic regression.

Test this tagger and compare its performance to the normal unigram and bigram tagger.

In [39]:
```
ct11 = ClassifierTagger1(training_set[:1000], False, False,
    False, LogisticRegression)
ct12 = ClassifierTagger1(training_set[:1000], True, False,
    False, LogisticRegression)
ct13 = ClassifierTagger1(training_set[:1000], True, True,
    False, LogisticRegression)
dt = nltk.DefaultTagger('NN')
ut = nltk.UnigramTagger(training_set[:1000])
bt = nltk.BigramTagger(training_set[:1000], backoff=ut)
ut2 = nltk.UnigramTagger(training_set[:1000], backoff=dt)
bt2 = nltk.BigramTagger(training_set[:1000], backoff=ut2)
```

In [40]:
```
print 'Classifier Tagger (LR with Unigram Tagger probabilities):'
print str(ct11.evaluate(test_set[:500]))
print 'Classifier Tagger (LR with concatenated Unigram and Bigram'
print 'Tagger probabilities):'
print str(ct12.evaluate(test_set[:500]))
print 'Classifier Tagger (LR with added Unigram and Bigram Tagger'
print 'probabilities):'
print str(ct13.evaluate(test_set[:500]))
print 'Unigram Tagger:'
print str(ut.evaluate(test_set[:500]))
print 'Bigram Tagger with Unigram backoff:'
print str(bt.evaluate(test_set[:500]))
print 'Bigram Tagger with Unigram and Default (NN) Tagger backoff:'
print str(bt2.evaluate(test_set[:500]))
```

```
Classifier Tagger (LR with Unigram Tagger probabilities):
0.778871740287
Classifier Tagger (LR with concatenated Unigram and Bigram
Tagger probabilities):
0.784903317367
Classifier Tagger (LR with added Unigram and Bigram Tagger
probabilities):
0.741706581515
Unigram Tagger:
0.736029803087
Bigram Tagger with Unigram backoff:
0.740819584886
Bigram Tagger with Unigram and Default (NN) Tagger backoff:
0.782685825794
```

Seems like concatenating the probabilities from Unigram and Bigram Tagger is a better idea than adding them.

All classifier taggers perform better than the normal Unigram and Bigram Taggers.

However, when the Default Tagger is added as last backoff, then only the Classifier Tagger with concatenated feature vectors is still a little bit better.

Now check how the performance changes when decision trees are used instead of logistic regression.

In [41]:
```
ct14 = ClassifierTagger1(training_set[:1000], False, False,
    False, DecisionTreeClassifier)
ct15 = ClassifierTagger1(training_set[:1000], True, False,
    False, DecisionTreeClassifier)
ct16 = ClassifierTagger1(training_set[:1000], True, True,
    False, DecisionTreeClassifier)
```

In [42]:
```
print 'Classifier Tagger (DT with Unigram Tagger probabilities):'
print str(ct14.evaluate(test_set[:500]))
print 'Classifier Tagger (DT with concatenated Unigram and'
print 'Bigram Tagger probabilities):'
print str(ct15.evaluate(test_set[:500]))
print 'Classifier Tagger (DT with added Unigram and Bigram Tagger'
print 'probabilities):'
print str(ct16.evaluate(test_set[:500]))
print 'Unigram Tagger:'
print str(ut.evaluate(test_set[:500]))
print 'Bigram Tagger with Unigram backoff:'
print str(bt.evaluate(test_set[:500]))
print 'Bigram Tagger with Unigram and Default (NN) Tagger backoff:'
print str(bt2.evaluate(test_set[:500]))
```

```
Classifier Tagger (DT with Unigram Tagger probabilities):
0.73824729466
Classifier Tagger (DT with concatenated Unigram and
Bigram Tagger probabilities):
0.390189817279
Classifier Tagger (DT with added Unigram and Bigram Tagger
probabilities):
0.460262551002
Unigram Tagger:
0.736029803087
Bigram Tagger with Unigram backoff:
0.740819584886
Bigram Tagger with Unigram and Default (NN) Tagger backoff:
0.782685825794
```

The Decision Tree Classifier does not work very well with Unigram+Bigram probabilities. When only Unigram Tagger probabilities are used, it is slightly better than the normal Unigram Tagger (but loses against the normal bigram tagger with unigram backoff).

Check if there is any difference when creating bigrams differently: Instead of (previous word, actual word), use (previous tag, actual word).

In [43]:
```
ct17 = ClassifierTagger1(training_set[:1000], True, False,
    True, LogisticRegression)

print 'Bigram Tagger with Unigram and Default (NN) Tagger backoff:'
print str(bt2.evaluate(test_set[:500]))
print 'Classifier Tagger (LR with concatenated Unigram and'
print 'Bigram (word,word) Tagger probabilities):'
print str(ct12.evaluate(test_set[:500]))
print 'Classifier Tagger (LR with concatenated Unigram and'
print 'Bigram (tag,word) Tagger probabilities):'
print str(ct17.evaluate(test_set[:500]))
```

```
Bigram Tagger with Unigram and Default (NN) Tagger backoff:
0.782685825794
Classifier Tagger (LR with concatenated Unigram and
Bigram (word,word) Tagger probabilities):
0.784903317367
Classifier Tagger (LR with concatenated Unigram and
Bigram (tag,word) Tagger probabilities):
0.739843888593
```

The performance goes down. Probably because (tag,word)-bigrams are less specific than (word,word)-bigrams.

Create another classifier with an arbitrary 'getFeatures'-method so that some other features can be tested in a simple way.

In [44]:
```
class ClassifierTagger2(nltk.tag.api.TaggerI):
    def __init__(self, trainingset, getFeatures_method,
        sklearn_classifier):
        self.getFeatures = getFeatures_method
        self.classifier = sklearn_classifier()

        # create list of possible tags
        tagged_words = []
        for sent in trainingset:
            tagged_words += sent
        self.taglist = list(set([t for w,t in tagged_words]))

        # extract features and train classifier
        classifier_training = []
        classifier_target = []
        for sent in trainingset:
            for i in range(len(sent)):
                classifier_training.append(self.getFeatures(
                    sent[i][0], (i+1.0)/(len(sent)+0.0)))
                classifier_target.append(self.taglist.index(
                    sent[i][1]))
        self.classifier.fit(classifier_training, classifier_target)

    def tag(self, sent):
        result = []
        for i in range(len(sent)):
            result.append(self.taglist[int(self.classifier.predict(
                self.getFeatures(sent[i],
                (i+1.0)/(len(sent)+0.0)))[0])])
        return result

    # evaluation method does not work, implement own
    def evaluate(self, gold):
        correct = 0
        total = 0
        for sent in gold:
            words = [w for w,t in sent]
            tags = [t for w,t in sent]
            pred = self.tag(words)
            for i in range(len(tags)):
                if(tags[i]==pred[i]):
                    correct = correct+1
                total = total+1
        return (1.0*correct)/(1.0*total)
```

First try a very simple feature "vector" just containing one feature: The index of the word in the big list of possible words.

In [45]:
```
tswordlist = []
for sent in training_set:
    for w,t in sent:
        tswordlist.append(w)
tswordlist = list(set(tswordlist))
print len(tswordlist)

def getWordIndex(word, _):
    fv = []
    if word in tswordlist:
        fv.append(tswordlist.index(word))
    else:
        fv.append(len(tswordlist))
    return fv
```

```
53369
```

In [46]:
```
ct27 = ClassifierTagger2(training_set[:1000],
    getWordIndex, LogisticRegression)
ct28 = ClassifierTagger2(training_set[:1000],
    getWordIndex, DecisionTreeClassifier)
print 'Word index with logistic regression:'
print str(ct27.evaluate(test_set[:500]))
print 'Word index with decision trees:'
print str(ct28.evaluate(test_set[:500]))
```

```
Word index with logistic regression:
0.15016852936
Word index with decision trees:
0.764502394891
```

The decision tree performs much better than the logistic regression classifier. This is because the features (the index of the word) have really nothing to do with the type of the word (which we want to predict). The tree can still perform good because it just grows very big.

Now try a another simple feature vector, containing:

Word length, position of the word in the sentence (relative to sentence length), first letter capital or not, relation of #vowels to wordlength.

In [47]:
```
# returns the number of vowels in the given word
def getNumberOfVowels(word):
    num_vowels = 0
    for c in word.lower():
        if(c=='a' or c=='e' or c=='i' or c=='o' or c=='u'):
            num_vowels = num_vowels + 1
    return num_vowels

def getSimpleFeatures(word, sentpos):
    fv = []
    fv.append(sentpos) # position of the word in the sentence
    fv.append(len(word)) # length of the word
    if(word[0].isupper()): # is the first letter upper case?
        fv.append(1)
    else:
        fv.append(0)
    nv = getNumberOfVowels(word)
    # number of vowels / word length
    fv.append((nv+0.0)/(len(word)+0.0))
    return fv
```

In [48]:
```
ct21 = ClassifierTagger2(training_set[:1000],
    getSimpleFeatures, LogisticRegression)
ct22 = ClassifierTagger2(training_set[:1000],
    getSimpleFeatures, DecisionTreeClassifier)
print 'Simple features with logistic regression:'
print str(ct21.evaluate(test_set[:500]))
print 'Simple features with decision trees:'
print str(ct22.evaluate(test_set[:500]))
```

```
Simple features with logistic regression:
0.352492460529
Simple features with decision trees:
0.352315061203
```

As expected, logistic regression now performs better and the decision tree performs worse.

But now they are both quite bad. There is too little information (for solving the tagging problem) in the extracted features.

One meaningful feature should be the ending of the word. Check if this improves the performance.

In [49]:
```
# returns the sum of the ASCII values of the last three characters
# of the word
def getWordEnding(word):
    if(len(word) >= 3):
        return ord(word[-1]) + ord(word[-2]) + ord(word[-3])
    elif(len(word) == 2):
        return ord(word[0]) + ord(word[1])
    elif(len(word) == 1):
        return ord(word[0])
    else:
        return 0

def getBetterFeatures(word, sentpos):
    fv = getSimpleFeatures(word, sentpos)
    fv.append(getWordEnding(word))
    return fv
```

In [50]:
```
ct29 = ClassifierTagger2(training_set[:1000],
    getBetterFeatures, LogisticRegression)
ct210 = ClassifierTagger2(training_set[:1000],
    getBetterFeatures, DecisionTreeClassifier)
print 'Better features with logistic regression:'
print str(ct29.evaluate(test_set[:500]))
print 'Better features with decision trees:'
print str(ct210.evaluate(test_set[:500]))
```

```
Better features with logistic regression:
0.423895689196
Better features with decision trees:
0.660280290935
```

As expected, the performance is improved with one more meaningful feature, the ending of the word.

Now add a simple hash of the word to the feature vector: Just the sum of all the ASCII values of the characters of the word.

In [51]:
```
# returns a simple hash for a given word: the sum of
# all its ASCII characters
def getSimpleHash(word):
    h = 0
    for c in word:
        h += ord(c)
    return h

def getSimpleFeaturesWithHash(word, sentpos):
    fv = getSimpleFeatures(word, sentpos)
    # very simple hash: sum of all ASCII values
    fv.append(getSimpleHash(word))
    return fv
```

In [52]:
```
ct23 = ClassifierTagger2(training_set[:1000],
    getSimpleFeaturesWithHash, LogisticRegression)
ct24 = ClassifierTagger2(training_set[:1000],
    getSimpleFeaturesWithHash, DecisionTreeClassifier)
print 'Simple features with hash with logistic regression:'
print str(ct23.evaluate(test_set[:500]))
print 'Simple features with hash with decision trees:'
print str(ct24.evaluate(test_set[:500]))
```

```
Simple features with hash with logistic regression:
0.390278516942
Simple features with hash with decision trees:
0.62755011531
```

Now the decision tree classifier is better. The reason is the same as with the word index: the very different hashes for different words lead to a very big tree. This is not a nice decision tree, but it performs better than the logistic regression.

To check this assumption, now use an additional hash function (found in the internet) that produces even more different hashes. The decision tree should be as good as with the word index, and logistic regression should be really bad.

In [53]:
```
def getOtherHash(word):
    h = 5381
    for c in word:
        h = ((h << 5) + h) + ord(c)
    return h

def getSimpleFeaturesWith2Hashes(word, sentpos):
    fv = getSimpleFeatures(word, sentpos)
    # very simple hash: sum of all ASCII values
    fv.append(getSimpleHash(word))
    # more complicated hash
    fv.append(getOtherHash(word))
    return fv
```

In [54]:
```
ct25 = ClassifierTagger2(training_set[:1000],
    getSimpleFeaturesWith2Hashes, LogisticRegression)
ct26 = ClassifierTagger2(training_set[:1000],
    getSimpleFeaturesWith2Hashes, DecisionTreeClassifier)
print 'Simple features with hash with logistic regression:'
print str(ct25.evaluate(test_set[:500]))
print 'Simple features with hash with decision trees:'
print str(ct26.evaluate(test_set[:500]))
```

```
Simple features with hash with logistic regression:
0.0416888415824
Simple features with hash with decision trees:
0.737005499379
```

Now the logistic regression works even worse and the decision tree works even better.

In general, classifiers need feature vectors that contain information that is in some way related to what should be predicted. If there is no relation at all, decision trees are still good, but then they are nothing more than a big index structure for all the possible words.