In [2]:
```
import nltk
from nltk.corpus import names
from pylab import *
import random as pyrandom
```

Text Classification

Let's start with a very simple text classification problem: guessing the gender of a name from the name itself.

You can probably make a pretty good guess about the gender of names like: "Bilama" or "Telek".

If we want to generalize to new names, we need to extract properties of names that occur in new, previously unseen names. We call these properties *features*.

In NLTK, they are represented as hash tables.

In [3]:
```
def gender_features(w):
    return dict(last_letter=w[-1],
                first_letter=w[0],
                length=len(w))
gender_features("Petra")
```

Out[3]:
```
{'first_letter': 'P', 'last_letter': 'a', 'length': 5}
```

For the training data, we read male and female names from NLTK corpora.

In [4]:
```
male = [(name,'male') for name in names.words('male.txt')]
female = [(name,'female') for name in names.words('female.txt')]
nlist = male+female
print len(nlist),len(set([x for x,y in nlist]))
```

```
7944 7579
```

In [5]:
```
pyrandom.shuffle(nlist)
nlist[:5]
```

Out[5]:
```
[('Celine', 'female'),
 ('Brunella', 'female'),
 ('Heywood', 'male'),
 ('Brittan', 'female'),
 ('Leticia', 'female')]
```

We extract features and then split the data into a training set and a test set.

Typesetting math: 91%

In [6]:
```
featuresets = [(gender_features(n),g) for n,g in nlist]
training_set = featuresets[500:]
test_set = featuresets[:500]
```

In [7]:
```
featuresets[:5]
```

Out[7]:
```
[({'first_letter': 'C', 'last_letter': 'e', 'length': 6}, 'female'),
 ({'first_letter': 'B', 'last_letter': 'a', 'length': 8}, 'female'),
 ({'first_letter': 'H', 'last_letter': 'd', 'length': 7}, 'male'),
 ({'first_letter': 'B', 'last_letter': 'n', 'length': 7}, 'female'),
 ({'first_letter': 'L', 'last_letter': 'a', 'length': 7}, 'female')]
```

Once we have features and corresponding labels, we can train a classifier...

In [8]:
```
classifier = nltk.NaiveBayesClassifier.train(training_set)
```

... and evaluate its performance on the test set.

In [9]:
```
nltk.classify.accuracy(classifier,test_set)
```

Out[9]:
```
0.794
```

Classiifers also give us information about how informative features are.

In [10]:
```
classifier.show_most_informative_features(5)
```

```
Most Informative Features
             last_letter = 'a'            female : male   =     40.2 : 1.0
             last_letter = 'k'              male : female =     31.7 : 1.0
             last_letter = 'f'              male : female =     15.3 : 1.0
             last_letter = 'p'              male : female =     11.9 : 1.0
             last_letter = 'v'              male : female =     11.2 : 1.0
```

Naive Bayesian classifiers assume a very simple statistical model of the posterior probability P(c|x) for input features x = (x_1,...,x_n)

- We assume that each feature is generated independently P(x|c) = \prod_i(x_i|c)
- We use Bayes formula to turn that equation into a posterior probability P(c|x)

Here, the different P(x_i|c) are modeled via empirical distributions; that is, we count how often x_i is true given that the class is c.

Another Classifier

Typesetting math: 91%

There are many different classifiers available in NLTK; they give different performance on different tasks. There is no single best classifier, so you need a bit of experimentation.

In [11]:
```
classifier = nltk.MaxentClassifier.train(training_set,algorithm="IIS",max_ite
nltk.classify.accuracy(classifier,test_set)
```

```
  ==> Training (10 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -0.69315        0.370
             2          -0.47663        0.732
             3          -0.42013        0.775
             4          -0.39252        0.778
             5          -0.37730        0.780
             6          -0.36812        0.782
             7          -0.36224        0.781
             8          -0.35830        0.781
             9          -0.35559        0.780
         Final          -0.35366        0.781
```

Out[11]: 0.806

The maximum entropy classifier can be derived in different ways. In practice, it amounts to the same classifier as logistic regression:

P(c|x) = \sigma(w \cdot x)

where \sigma(x) = \frac{1}{1+e^{-x}}

The term "MaxEnt" is frequently used in NLP. It refers to the fact that we are thinking of the problem as follows:

- assume that we have a set of binary feature functions of documents x_i(d)
- each binary feature function is true or false
- for each binary feature function, we have a posterior P(c|x_i)
- we want to find an overall posterior probability P(c|d) that is...
  - consistent with the individual posteriors
  - otherwise a "maximum entropy distribution"

"Traditional" Classifier

The NLTK classifiers take features in the form of hash tables; this is convenient for NLP tasks, but somewhat inefficient.

Classifiers in other machine learning libraries tend to take input data in a different format.

A common format is two matrices, one for inputs (each row representing an input vectors), and one for outputs (containing integer classes or indicator functions).

Typesetting math: 91%

In [12]:
```
xs = zeros((len(training_set),26))
ys = zeros(len(training_set))
```

For coding the inputs, we use a "unary code".

In [13]:
```
for i,(f,c) in enumerate(training_set):
    ll = f["last_letter"].lower()
    if ll==" " : continue
    xs[i,ord(ll)-ord("a")] = 1
    if c=="female": ys[i] = 1
```

LogisticRegression is the same as *MaxentClassifier*, but the sklearn implementation is much faster.

In [14]:
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(xs,ys)
```

Out[14]:
```
LogisticRegression(C=1.0, dual=False, fit_intercept=True,
intercept_scaling=1,
          penalty='l2', scale_C=False, tol=0.0001)
```

In [15]:
```
xs = zeros((len(test_set),26))
ys = zeros(len(test_set))
for i,(f,c) in enumerate(test_set):
    ll = f["last_letter"].lower()
    if ll==" " : continue
    xs[i,ord(ll)-ord("a")] = 1
    if c=="female": ys[i] = 1
```

In [16]:
```
1.0-sum(lr.predict(xs)!=ys)*1.0/len(ys)
```

Out[16]:
```
0.77400000000000002
```

Bigger Feature Set

There is no single right feature set, and different feature sets give different amounts of performance for different classifiers.

Typesetting math: 91%

In [17]:
```
def more_features(w):
    features = {}
    features["first"] = w[0].lower()
    features["last"] = w[-1].lower()
    features["last2"] = w[-2:].lower()
    for c in [chr(i) for i in range(ord("a"),ord("z")+1)]:
        features["nr_"+c] = name.lower().count(c)
        features["has_"+c] = (c in name.lower())
    return features
```

In [18]:
```
featuresets = [(more_features(n),g) for n,g in nlist]
training_set = featuresets[500:]
test_set = featuresets[:500]
classifier = nltk.NaiveBayesClassifier.train(training_set)
nltk.classify.accuracy(classifier,test_set)
```

Out[18]:
```
0.792
```

Usually, you should split the training data into three sets:

- the training set
- the feature evaluation set
- the test set

If you don't, you risk that you get a good result on the test set by accident, a result that doesn't generalize.

Other approaches are resampling methods and cross-validation.

What are some of the tradeoffs in choosing a feature set?

Decision Trees

Decision trees are another common classifier.

In [19]:
```
def simple_features(w):
    return {'fl':w[0].lower(),'ll': w[-1].lower(),'l':len(w)}
simple_features("Petra")
```

Out[19]:
```
{'fl': 'p', 'l': 5, 'll': 'a'}
```

In [20]:
```
featuresets = [(simple_features(n),g) for n,g in nlist]
training_set = featuresets[500:]
test_set = featuresets[:500]
```

Typesetting math: 91%

In [21]:
```
classifier = nltk.DecisionTreeClassifier.train(training_set,depth_cutoff=2)
nltk.classify.accuracy(classifier,test_set)
```

Out[21]: 0.754

Typesetting math: 91%

In [22]:
```
print classifier.pseudocode()
```

```
if ll == ' ': return 'female'
if ll == 'a': return 'female'
if ll == 'b':
  if fl == 'a': return 'male'
  if fl == 'b': return 'female'
  if fl == 'c': return 'male'
  if fl == 'd': return 'female'
  if fl == 'g': return 'male'
  if fl == 'h': return 'male'
  if fl == 'j': return 'male'
  if fl == 'k': return 'male'
  if fl == 'l': return 'female'
  if fl == 'm': return 'female'
  if fl == 'r': return 'male'
  if fl == 's': return 'female'
  if fl == 't': return 'male'
  if fl == 'w': return 'male'
  if fl == 'z': return 'male'
if ll == 'c': return 'male'
if ll == 'd': return 'male'
if ll == 'e':
  if fl == 'a': return 'female'
  if fl == 'b': return 'female'
  if fl == 'c': return 'female'
  if fl == 'd': return 'female'
  if fl == 'e': return 'female'
  if fl == 'f': return 'female'
  if fl == 'g': return 'female'
  if fl == 'h': return 'female'
  if fl == 'i': return 'female'
  if fl == 'j': return 'female'
  if fl == 'k': return 'female'
  if fl == 'l': return 'female'
  if fl == 'm': return 'female'
  if fl == 'n': return 'female'
  if fl == 'o': return 'female'
  if fl == 'p': return 'female'
  if fl == 'q': return 'female'
  if fl == 'r': return 'female'
  if fl == 's': return 'female'
  if fl == 't': return 'female'
  if fl == 'u': return 'female'
  if fl == 'v': return 'female'
  if fl == 'w': return 'male'
  if fl == 'y': return 'female'
  if fl == 'z': return 'male'
if ll == 'f':
  if fl == 'a': return 'male'
  if fl == 'b': return 'male'
  if fl == 'c': return 'male'
  if fl == 'g': return 'male'
  if fl == 'j': return 'male'
  if fl == 'l': return 'male'
  if fl == 'o': return 'male'
  if fl == 'p': return 'male'
```

Typesetting math: 91%

Decision trees are classifiers that classify as a nested sequence of if-then statements.

Variables can be binary, categorical, or numeric.

For numerical variables, they divide the feature space into axis-parallel rectangles and associated probabilities.

Decision trees are generally grown as follows:

- take a set of data
- consider splits along every possible feature and value
- pick the best split according to the minimal impurity of the corresponding label set
- split according to that feature and value
- repeat the process on each subset (branch)
- stop if a minimum impurity or set size is reached

A better way of doing this is to split like the above, into small terminal nodes (deliberate overfitting), then start merging terminal nodes back together again, based on cross-validated error ("pruning"). This is what CART does, and leads to better overall performance.

Document Classification

In [23]:
```
from nltk.corpus import movie_reviews
```

In [24]:
```
documents = [(list(movie_reviews.words(fileid)),category)
                     for category in movie_reviews.categories()
                         for fileid in movie_reviews.fileids(category)]
```

In [25]:
```
pyrandom.shuffle(documents)
```

In [26]:
```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
```

In [27]:
```
word_features = all_words.keys()[:2000]
```

In [28]:
```
def document_features(document):
    document_words = set(document)
    features = {}
    for w in word_features:
        features[w] = (w in document_words)
    return features
```

Typesetting math: 91%

In [29]:
```
print document_features(documents[0][0])
```


```
{'limited': False, 'four': False, 'woods': False, 'woody': False,
'captain': False, 'hate': False, 'consider': False, 'relationships':
False, 'whose': False, 'buddy': True, 'themes': False, 'presents': False,
'edward': False, 'under': False, 'lord': False, 'worth': True, 'rescue':
False, 'every': True, 'jack': True, 'bringing': False, 'school': False,
'skills': True, 'ups': False, 'enjoy': True, 'force': False, 'tired':
False, 'miller': False, 'direct': False, 'second': False, 'street':
False, 'even': True, '+': False, 'above': False, 'new': True, 'poorly':
False, 'ever': False, 'disney': False, 'told': True, 'hero': False,
'mel': False, 'human': False, 'men': False, 'here': True, 'studio':
False, 'cult': False, '100': False, 'kids': False, 'daughter': False,
'leaves': False, 'changed': True, 'credit': False, 'military': False,
'changes': False, 'fantastic': False, 'julie': False, 'explained': False,
'julia': False, 'highly': False, 'brought': False, 'moral': False,
'actions': False, 'total': False, 'sarah': False, 'plot': False, 'would':
False, 'army': False, 'hospital': False, 'music': False, 'therefore':
False, 'recommend': True, 'strike': False, 'survive': False, 'type':
False, 'until': False, 'speaking': False, 'successful': False, 'brings':
False, 'wars': False, 'award': False, 'hurt': False, 'phone': False,
'adult': False, 'excellent': True, '90': False, 'hold': False, 'must':
False, 'shoot': False, 'word': False, 'room': False, '1997': False,
'1996': False, '1999': False, '1998': True, 'blade': False, 'movies':
False, 'era': False, 'ms': False, 'mr': False, 'my': True, 'example':
False, 'give': False, 'climax': False, 'laughs': False, 'want': False,
'times': False, 'end': False, 'thing': False, 'provide': False, 'travel':
False, 'sitting': False, 'feature': False, 'machine': False, 'how': True,
'amazing': False, 'writers': False, 'answer': False, 'beach': False,
'badly': False, 'elizabeth': False, 'beauty': False, 'mess': False,
'after': True, 'wrong': False, 'president': False, 'law': False, 'danny':
False, 'attempt': False, 'third': False, 'appreciate': False, 'lost':
False, 'green': False, 'ultimate': False, 'keeps': False, 'worst': False,
'order': True, 'office': False, 'over': False, 'before': False, 'fit':
False, 'personal': False, ',': True, 'writing': False, 'better': True,
'production': False, 'compelling': False, 'hidden': False, 'then': True,
'them': True, 'safe': False, 'break': True, 'band': False, 'effects':
False, 'they': True, 'one': True, 'alex': False, 'rocky': False, 'debut':
False, 'l': False, 'grows': False, 'each': False, 'went': False, 'side':
False, 'mean': False, 'meets': False, 'series': True, 'truman': False,
'sounds': False, 'driving': False, 'god': False, 'cheesy': False,
'content': False, 're': True, 'got': False, 'turning': False, 'little':
True, 'free': False, 'standard': False, 'masterpiece': False, 'struggle':
False, 'wanted': False, 'created': True, 'starts': False, 'days': False,
'creates': False, 'isn': False, 'uses': True, 'onto': False, 'already':
True, 'features': False, 'fantasy': False, 'another': True, 'wasn':
False, 'comic': False, 'toy': False, 'top': False, 'girls': False,
'fiction': True, 'needed': False, 'master': False, 'too': True, 'tom':
False, 'hollywood': False, 'john': False, 'carrey': False, 'urban':
False, 'murder': False, 'serve': False, 'took': False, 'japanese': False,
'predictable': False, 'somewhat': False, 'helen': False, 'wasted': False,
'begins': False, 'trek': False, 'target': False, 'roles': False,
'likely': False, 'project': False, 'matter': False, 'silly': False,
'williams': False, 'feeling': False, 'powers': False, 'screenplay':
False, 'fashion': False, 'sees': False, 'modern': False, 'mind': True,
'talking': False, 'manner': False, 'seen': True, 'seem': False, 'tells':
e, 'ray': False, 'forced': False, 'strength': False, 'genuine':
```


Typesetting math: 91%

In [30]:
```
featuresets = [(document_features(d),c) for d,c in documents]
training_set = featuresets[:100]
test_set = featuresets[100:]
```

In [31]:
```
classifier = nltk.NaiveBayesClassifier.train(training_set)
```

In [32]:
```
nltk.classify.accuracy(classifier,test_set)
```

Out[32]: 0.7094736842105264

In [33]:
```
classifier.show_most_informative_features(5)
```

```
Most Informative Features
                powerful = True              pos : neg    =      6.8 : 1.0
                  change = True              pos : neg    =      6.8 : 1.0
               obviously = True              neg : pos    =      6.5 : 1.0
                     due = True              pos : neg    =      6.1 : 1.0
               perfectly = True              pos : neg    =      6.1 : 1.0
```

Parts of Speech Tagging

In [34]:
```
from nltk.corpus import brown
```

In [35]:
```
suffixes = nltk.FreqDist()
for word in brown.words():
    word = word.lower()
    suffixes.inc(word[-1:])
    suffixes.inc(word[-2:])
    suffixes.inc(word[-3:])
```

In [36]:
```
common = suffixes.keys()[:100]
print common
```

```
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the', 'y', 'r',
'to', 'in', 'f', 'o', 'ed', 'nd', 'is', 'on', 'l', 'g', 'and', 'ng',
'er', 'as', 'ing', 'h', 'at', 'es', 'or', 're', 'it', '``', 'an', "''",
'm', ';', 'i', 'ly', 'ion', 'en', 'al', '?', 'nt', 'be', 'hat', 'st',
'his', 'th', 'll', 'le', 'ce', 'by', 'ts', 'me', 've', "'", 'se', 'ut',
'was', 'for', 'ent', 'ch', 'k', 'w', 'ld', '`', 'rs', 'ted', 'ere',
'her', 'ne', 'ns', 'ith', 'ad', 'ry', ')', '(', 'te', '--', 'ay', 'ty',
'ot', 'p', 'nce', "'s", 'ter', 'om', 'ss', ':', 'we', 'are', 'c', 'ers',
'uld', 'had', 'so', 'ey']
```

Typesetting math: 91%

In [37]:
```
def pos_features(w):
    features = {}
    for s in common:
        features[s] = w.lower().endswith(s)
    return features
```

In [38]:
```
tagged_words = brown.tagged_words(categories='news')
```

In [39]:
```
featuresets = [(pos_features(w),c) for w,c in tagged_words]
n = len(featuresets)
print n
```

```
100554
```

In [40]:
```
training_set = featuresets[n//10:]
test_set = featuresets[:n//10]
```

In [41]:
```
classifier = nltk.DecisionTreeClassifier.train(training_set)
```

In [42]:
```
nltk.classify.accuracy(classifier,test_set)
```

Out[42]:
```
0.6270512182993535
```

In [43]:
```
classifier.classify(pos_features('cats'))
```

Out[43]:
```
'NNS'
```

In [44]:
```
print classifier.pseudocode(depth=4)
```

```
if the == False:
  if , == False:
    if s == False:
      if . == False: return '.'
      if . == True: return '.'
    if s == True:
      if is == False: return 'PP$'
      if is == True: return 'BEZ'
  if , == True: return ','
if the == True: return 'AT'
```

In [44]:

Typesetting math: 91%