In [1]:
```
import numpy
import urllib
import scipy.optimize
import random
from sklearn.decomposition import PCA
from collections import defaultdict

### PCA on beer reviews ###

def parseData(fname):
  for l in urllib.urlopen(fname):
    yield eval(l)

print "Reading data..."
data = list(parseData("http://jmcauley.ucsd.edu/cse190/data/beer/beer_50000.json"))
print "done"
```

```
Reading data...
done
```

In [2]:
```
import numpy
import scipy.optimize
import random
from math import exp
from math import log

def feature(datum):
  feat = [1, datum['review/taste'], datum['review/appearance'], \
          datum['review/aroma'], datum['review/palate'], \
          datum['review/overall']]
  return feat

# first shuffle the data
rand_data = numpy.copy(data)
numpy.random.shuffle(rand_data)
X = [feature(d) for d in rand_data]
y = [d['beer/ABV'] >= 6.5 for d in rand_data]

def inner(x,y):
  return sum([x[i]*y[i] for i in range(len(x))])

def sigmoid(x):
  return 1.0 / (1 + exp(-x))

####################################################
# Logistic regression by gradient ascent           #
####################################################

# NEGATIVE Log-likelihood
def f(theta, X, y, lam):
```

```
  loglikelihood = 0
  for i in range(len(X)):
    logit = inner(X[i], theta)
    loglikelihood -= log(1 + exp(-logit))
    if not y[i]:
      loglikelihood -= logit
  for k in range(len(theta)):
    loglikelihood -= lam * theta[k]*theta[k]
  # for debugging
  # print("ll =" + str(loglikelihood))
  return -loglikelihood

# NEGATIVE Derivative of log-likelihood
def fprime(theta, X, y, lam):
  dl = [0]*len(theta)
  for i in range(len(X)):
    logit = inner(X[i], theta)
    for k in range(len(theta)):
      dl[k] += X[i][k] * (1 - sigmoid(logit))
      if not y[i]:
        dl[k] -= X[i][k]
  for k in range(len(theta)):
    dl[k] -= lam*2*theta[k]
  return numpy.array([-x for x in dl])
```

In [3]:

```
# 1. The code currently does not perform any train/test splits.
# Split the data into training, validation, and test sets,
# via 1/3, 1/3, 1/3 splits. Use random splits of the data
# (i.e., each should be a random, non- overlapping sample of the da
ta;
# this can be obtained by first shuffling the data). After training
on
# the training set, report the accuracy of the classifier on the
# validation and test sets (1 mark).


# Split the data into training, validation, and test sets,
# via 1/3, 1/3, 1/3 splits
length = int(len(rand_data)/3)

X_train = X[:length]
y_train = y[:length]

X_validation = X[length:2*length]
y_validation = y[length:2*length]

X_test = X[2*length:]
y_test = y[2*length:]

##################################################
# Train                                          #
##################################################
```

```
def train(lam):
  theta,_,_ = scipy.optimize.fmin_l_bfgs_b(f, [0]*len(X[0]), fprime
, pgtol = 10, args = (X_train, y_train, lam))
  return theta

# train on training set
lam = 1.0
theta = train(lam)

#################################################
# Predict                                       #
#################################################

def performance(theta, set_x, set_y):
  scores = [inner(theta,x) for x in set_x]
  predictions = [s > 0 for s in scores]
  correct = [(a==b) for (a,b) in zip(predictions,set_y)]
  acc = sum(correct) * 1.0 / len(correct)
  return acc

# accuracy
acc_train = performance(theta, X_train, y_train)
print("lambda = " + str(lam) + ":\taccuracy_train =\t" + str(acc_tr
ain))

acc_test = performance(theta, X_test, y_test)
print("lambda = " + str(lam) + ":\taccuracy_tests =\t" + str(acc_te
st))

acc_validate = performance(theta, X_validation, y_validation)
print("lambda = " + str(lam) + ":\taccuracy_validation =\t" + str(a
cc_validate))
```

```
lambda = 1.0:   accuracy_train =        0.7205088203528142
lambda = 1.0:   accuracy_tests =        0.7143628509719222
lambda = 1.0:   accuracy_validation =   0.7147485899435977
```

In [4]:

```
# 2. Report the number of Positives, Negatives, True Positives,
# True Negatives, False Positives, and False Negatives using
# the test set of the classifier you trained above (1 mark).
def P(y_data):
    posit = [(a==1) for a in y_data]
    p = sum(posit) * 1.0
    return p

def N(y_data):
    negat = [(a==0) for a in y_data]
    n = sum(negat) * 1.0
    return n

def TP(theta, X_data, y_data):
    scores = [inner(theta,x) for x in X_data]
    predictions = [s > 0 for s in scores]
```

```
    correct = [((a==1) and (b==1)) for (a,b) in zip(predictions,y_d
ata)]
    tp = sum(correct) * 1.0
    return tp
def TN(theta, X_data, y_data):
    scores = [inner(theta,x) for x in X_data]
    predictions = [s > 0 for s in scores]
    correct = [((a==0) and (b==0)) for (a,b) in zip(predictions,y_d
ata)]
    tn = sum(correct) * 1.0
    return tn
def FP(theta, X_data, y_data):
    scores = [inner(theta,x) for x in X_data]
    predictions = [s > 0 for s in scores]
    correct = [((a==1) and (b==0)) for (a,b) in zip(predictions,y_d
ata)]
    fp = sum(correct) * 1.0
    return fp
def FN(theta, X_data, y_data):
    scores = [inner(theta,x) for x in X_data]
    predictions = [s > 0 for s in scores]
    correct = [((a==0) and (b==1)) for (a,b) in zip(predictions,y_d
ata)]
    fn = sum(correct) * 1.0
    return fn

# Positeves
p = P(y_test)
# Negatives
n = N(y_test)
# True Positives
tp = TP(theta, X_test, y_test)
# True Negatives
tn = TN(theta, X_test, y_test)
# False Positives
fp = FP(theta, X_test, y_test)
# False Negatives
fn = FN(theta, X_test, y_test)
print ("Positives\tNegatives\tTrue Positives\tTrue Negatives\tFalse
Positives\tFalse Negatives")
print (str(p)+'\t\t'+str(n)+'\t\t'+str(tp)+'\t\t'+str(tn)+'\t\t'+st
r(fp)+'\t\t'+str(fn))
```

```
Positives       Negatives       True Positives  True Negatives
False Positives False Negatives
10341.0         6327.0          9111.0          2796.0
3531.0          1230.0
```

In [ ]:
```
# 3. (Hard)Describehowyouwouldmodifythecodestubprovidedifyouwantedt
oassigngreaterimportance
# to False Positives compared to False Negatives. Suggest specific
modifications
# that would make to the code if you wanted to assign 10 times as m
```

```
uch importance
# to False Positives as compared to False Negatives.

# Answer
# So the basic idea is that: the loss function can be the contribut
ed both by FP and FN,
# and loss function also equals to -loglikelihood. so we should mod
ify the function
# f(theta, X, y, lam) which represents -loglikelihood.
# the log likelihood equation: ∑i-log(1+e-Xi.θ)+∑y=0-Xi.θ-λ||θ||2
# FP means modle prediction is positive but it is negative in lable
.
# if we want to assign more importance on FP, we should multiple a
factor
# to the second part of the equation above
def f(theta, X, y, lam):
  loglikelihood = 0
  for i in range(len(X)):
    logit = inner(X[i], theta)
    loglikelihood -= log(1 + exp(-logit))
    ##########################################
    if not y[i]:                              #
        if logit > 0:                         #
            loglikelihood -= logit * factor  # Say: factor can be 1
0.
        else:                                 #
            loglikelihood -= logit            #
    ##########################################
  for k in range(len(theta)):
    loglikelihood -= lam * theta[k]*theta[k]
  # for debugging
  # print("ll =" + str(loglikelihood))
  return -loglikelihood

# NEGATIVE Derivative of log-likelihood
def fprime(theta, X, y, lam):
  dl = [0]*len(theta)
  for i in range(len(X)):
    logit = inner(X[i], theta)
    for k in range(len(theta)):
      dl[k] += X[i][k] * (1 - sigmoid(logit))
    ##########################################
      if not y[i]:                            #
            if logit > 0:                     #
                dl[k] -= X[i][k]*factor      # we add the factor he
re accordingly
            else:                             #
                dl[k] -= X[i][k]              #
    ##########################################
  for k in range(len(theta)):
    dl[k] -= lam*2*theta[k]
  return numpy.array([-x for x in dl])
```

In [5]:
```
# 4. Implement a training/validation/test pipeline so that you can
select
# the best model based on its perfor- mance on the validation set.
# Try models with λ ∈ {0, 0.01, 0.1, 1, 100}. Report the performanc
e on the
# training/validation/test sets for the best value of λ (1 mark).

LAM = [0, 0.01, 0.1, 1, 100]
for lam in LAM:
    theta = train(lam)
    print str(lam) + '\t' + str(performance(theta, X_validation, y_
validation))
```

```
0       0.7158286331453259
0.01    0.7172086883475339
0.1     0.7155886235449418
1       0.7147485899435977
100     0.6665066602664107
```

In [6]:
```
# the best value of λ is 0.01
lam = 0.01
theta = train(lam)
print "training\t" + '\t' + str(performance(theta, X_train, y_train
))
print "testing   \t" + '\t' + str(performance(theta, X_test, y_test
))
print "validation\t" + '\t' + str(performance(theta, X_validation,
y_validation))
```

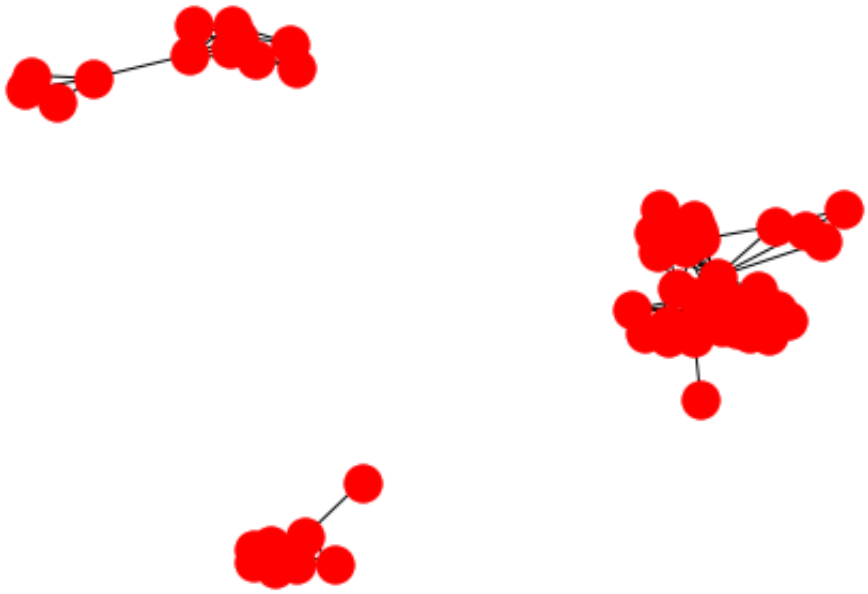
```
training                0.7203288131525261
testing                 0.7170626349892009
validation              0.7172086883475339
```

In [7]:
```
# 5. How many connected components are in the graph, and how many
# nodes are in the largest connected component (1 mark)?
### Network visualization ###
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

edges = set()
nodes = set()
for edge in urllib.urlopen("http://jmcauley.ucsd.edu/cse255/data/facebook/egonet.txt", 'r'):
    x,y = edge.split()
    x,y = int(x),int(y)
    edges.add((x,y))
    #edges.add((y,x))
    nodes.add(x)
    nodes.add(y)

#print len(edges)
G = nx.Graph()
for e in edges:
  G.add_edge(e[0],e[1])
#print len(G.edges())
nx.draw(G)
plt.show()
plt.clf()

components = sorted(nx.connected_components(G), key = len, reverse=True)
print "there are " + str(len(components)) + " components in the graph"
print "there are " + str(len(components[0])) + " nodes in the largest components"
```



```
there are 3 components in the graph
there are 40 nodes in the largest components

<Figure size 432x288 with 0 Axes>
```

In [8]:
```
# 6. What is the normalized-cut cost of the 50/50 split you found a
bove (1 mark)?
sorted_comp = sorted(components[0])
c1 = sorted_comp[:len(sorted_comp)/2]
c2 = sorted_comp[len(sorted_comp)/2:]

def norm_cut(c1, c2, G):
    s1 = sum([G.degree(v) for v in c1]) * 1.0
    s2 = sum([G.degree(v) for v in c2]) * 1.0
    return nx.cut_size(G, c1, c2) * (1/s1 + 1/s2) / 2.0

nc_0 = norm_cut(c1, c2, G)
nc_0
```

Out[8]:
```
0.4224058769513316
```

In [42]:
```
# 7. What are the elements of the split, and what is its normalized
cut cost (1 mark)?
#
A = numpy.copy(c1).tolist()
B = numpy.copy(c2).tolist()
res = nc_0
cur_min = (res,A,B)

# function to find the best partition in each step
def find_min(A,B,cur_min):
    res = (0,[],[])
    for i in range(len(A)):
        B.append(A[i])
        A.remove(A[i])
        if norm_cut(A,B,G) < cur_min[0]:
            a = numpy.copy(A).tolist()
            b = numpy.copy(B).tolist()
            res = (norm_cut(A,B,G), a, b)
        tmp = B[len(B)-1]
        B.remove(tmp)
        A.insert(i,tmp)
    return res


for i in range(200):
    ab = find_min(cur_min[1],cur_min[2],cur_min)
    ba = find_min(cur_min[2],cur_min[1],cur_min)
    if ab[0] == 0 and ba[0] == 0:
        #print "end at " + str(i) + " iteration"
        break
    if ab[0] == 0:
        cur_min = ba
    elif ba[0] == 0:
        cur_min = ab
    elif ab[0] < ba[0]:
        cur_min = ab
    else:
        cur_min = ba

print "The minimum cost is " + str(cur_min[0]) + "\t"
print "cluster A: " + str(cur_min[1])
print "cluster B: " + str(cur_min[2])
#print nx.cut_size(G, cur_min[1], cur_min[2])
#print norm_cut(cur_min[1], cur_min[2],G)
#print
```

```
The minimum cost is 0.0981704596162
cluster A: [825, 861, 863, 864, 876, 878, 882, 884, 886, 888, 889,
893, 804, 729]
cluster B: [697, 703, 708, 713, 719, 745, 747, 753, 769, 772, 774,
798, 800, 803, 805, 810, 811, 819, 890, 880, 869, 840, 830, 828, 8
23, 856]
```

In [41]:
```
# 8. Re-implement your greedy algorithm above so that it maximizes
# the modularity, rather than the normal- ized cut cost. Report
# modularity values for the 50/50 split you find (1 mark).


# get the Network modularity
def max_modul(c1, c2, G):
    n1 = 0
    n2 = 0
    a1 = 0
    a2 = 0
    for edge in G.edges():
        if (edge[0] in c1) and (edge[1] in c1):
            n1 = n1 + 1
            a1 = a1 + 2
        elif (edge[0] in c2) and (edge[1] in c2):
            n2 = n2 + 1
            a2 = a2 + 2
        else:
            a1 += 1
            a2 += 1

    #print n1
    e1 = n1*1.0 / len(G.edges())
    e2 = n2*1.0 / len(G.edges())
    a1 = a1*1.0 / (2 * len(G.edges()))
    a2 = a2*1.0 / (2 * len(G.edges()))
    q = (e1-a1*a1) + (e2-a2*a2)
    return q

# function to find the best partition in each step
def find_max(A,B,G,cur_max):
    res = (0,[],[], 1000)
    for i in range(len(A)):
        B.append(A[i])
        A.remove(A[i])
        modul = max_modul(A,B,G)
        if modul > cur_max[0] or (modul == cur_max[0] and B[len(B)-
1] < cur_max[3]):
            a = numpy.copy(A).tolist()
            b = numpy.copy(B).tolist()
            res = (modul, a, b, B[len(B)-1])
            cur_max = (modul, a, b, B[len(B)-1])
        tmp = B[len(B)-1]
        B.remove(tmp)
        A.insert(i,tmp)
    return res


g = G.subgraph(components[0])

C = numpy.copy(c1).tolist()
D = numpy.copy(c2).tolist()
```

```
res = max_modul(C,D,g)

# last parameter for check the node id when there is a equal result
# appears for moving two different nodes
cur_max = (0,C,D,1000)

# interate 200 times to make sure that we will get a point that can
get
# maximum result
for i in range(200):
    ba = find_max(cur_max[2],cur_max[1],g,cur_max)
    ab = find_max(cur_max[1],cur_max[2],g,cur_max)
    if ab[0] == 0 and ba[0] == 0:
        print "end at " + str(i) + " iteration"
        break
    if ab[0] == 0:
        cur_max = ba
    elif ba[0] == 0:
        cur_max = ab
    elif ab[0] > ba[0]:
        cur_max = ab
    else:
        cur_max = ba



print "The maximum Network modularity is " + str(cur_max[0]) + "\t"
print "cluster A: " + str(sorted((cur_max[1])))
print "cluster B: " + str(sorted((cur_max[2])))
```

```
end at 11 iteration
The maximum Network modularity is 0.338016528926
cluster A: [697, 703, 708, 713, 719, 745, 747, 772, 774, 800, 803,
805, 810, 819, 823, 828, 830, 840, 880]
cluster B: [729, 753, 769, 798, 804, 811, 825, 856, 861, 863, 864,
869, 876, 878, 882, 884, 886, 888, 889, 890, 893]
```