In [1]:
```
import numpy
import urllib.request
import scipy.optimize
import random
import matplotlib.pyplot as plt
from math import exp
from math import log
import networkx as nx
```

In [2]:
```
def parseData(fname):
    for l in urllib.request.urlopen(fname):
        yield eval(l)


print("Reading data...")
data = list(parseData("http://jmcauley.ucsd.edu/cse258/data/beer/beer_50
000.json"))
print('Done')
```

```
Reading data...
Done
```

In [3]:
```
# Q1:
data1 = [d for d in data]
random.shuffle(data1)
train = data1[:int(len(data) / 3)]
validation = data1[int(len(data) / 3):2 * int(len(data) / 3)]
test = data1[2 * int(len(data) / 3):]


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


X_train = [feature(d) for d in train]
Y_train = [d["beer/ABV"] >= 6.5 for d in train]
X_validate = [feature(d) for d in validation]
Y_validate = [d["beer/ABV"] >= 6.5 for d in validation]
X_test = [feature(d) for d in test]
Y_test = [d["beer/ABV"] >= 6.5 for d in test]


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


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


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]
    return -loglikelihood


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])
```

```
##################################################
# Train                                          #
##################################################
def train(lam):
    theta, _, _ = scipy.optimize.fmin_l_bfgs_b(f, [0] * len(X_train[0]),
 fprime, pgtol=10,
                                              args=(X_train, Y_train, l
am))

    return theta


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


##################################################
# Validation pipeline                            #
##################################################

lam = 1.0

theta = train(lam)
acc_validation = performance(theta, X_validate, Y_validate)
acc_test = performance(theta, X_test, Y_test)
print("lambda = " + str(lam) + ":\tvalidation\t accuracy=" + str(acc_val
idation))
print("lambda = " + str(lam) + ":\ttest\t accuracy=" + str(acc_test))

# validation     accuracy=0.7209288371534861
# test   accuracy=0.7162227021838253
```

```
lambda = 1.0:   validation       accuracy=0.7209288371534861
lambda = 1.0:   test     accuracy=0.7162227021838253
```

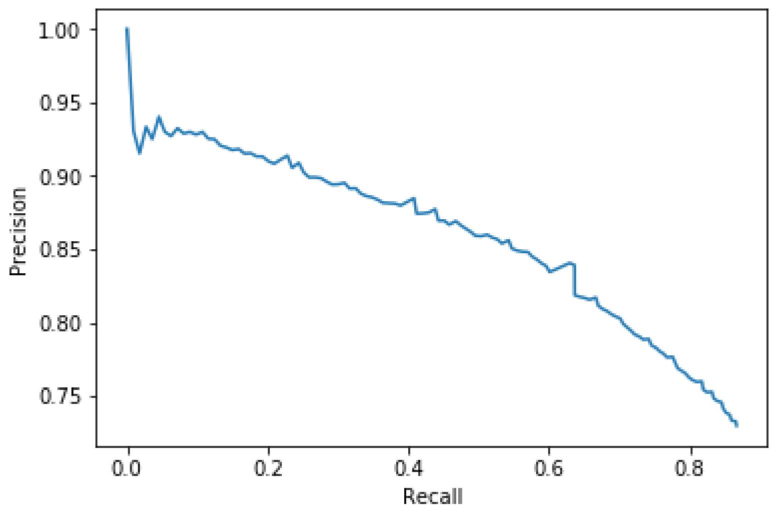
In [4]:
```
# Q2:
Positives = sum([y > 0 for y in Y_test])
print("Positives:\t", Positives)
Negatives = sum([y == 0 for y in Y_test])
print("Negatives:\t", Negatives)
Y_pred = [inner(theta, x) for x in X_test]
TP = sum([Y_test[i] > 0 and Y_pred[i] > 0 for i in range(len(Y_test))])
print("True Positives:\t", TP)
TN = sum([Y_test[i] == 0 and Y_pred[i] <= 0 for i in range(len(Y_test
))])
print("True Negatives:\t", TN)
FP = sum([Y_test[i] == 0 and Y_pred[i] > 0 for i in range(len(Y_test))])
print("False Positives:\t", FP)
FN = sum([Y_test[i] > 0 and Y_pred[i] <= 0 for i in range(len(Y_test))])
print("False Negatives:\t", FN)

# Positives:      10412
# Negatives:      6256
# True Positives:         9024
# True Negatives:         2914
# False Positives:        3342
# False Negatives:        1388
```

```
Positives:       10412
Negatives:       6256
True Positives:  9024
True Negatives:  2914
False Positives:         3342
False Negatives:         1388
```

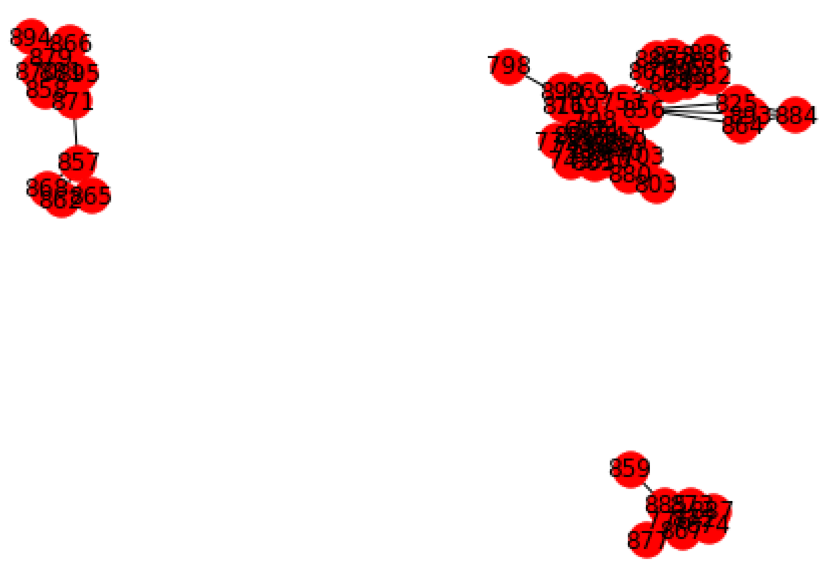
In [6]:
```
# Q3:
Precision = TP / (TP + FP)
print("Precision:\t", Precision)
Recall = TP / (TP + FN)
print("Recall:\t", Recall)

y = [(Y_pred[i], Y_test[i]) for i in range(len(Y_test))]
sort_Y = sorted(y)
Y_100 = sort_Y[len(Y_pred) - 100:]
TP100 = sum([y[0] > 0 and y[1] > 0 for y in Y_100])
FP100 = sum([y[0] > 0 and y[1] == 0 for y in Y_100])
Precision100 = TP100 / (TP100 + FP100)
Recall100 = TP100 / (TP + FN)
print("Precision@100:\t", Precision100)
print("Recall@100:\t", Recall100)

# Precision:      0.7297428432799612
# Recall:         0.866692278140607
# Precision@100:          0.94
# Recall@100:     0.009028044563964657
```

```
Precision:       0.7297428432799612
Recall:  0.866692278140607
Precision@100:   0.94
Recall@100:      0.009028044563964657
```

In [12]:
```
# Q4:
from sklearn.metrics import precision_recall_curve
precision = []
recall = []
for k in range(1, 16668,100):
    Y_k = sort_Y[len(Y_pred) - k:]
    TP_k = sum([y[0] > 0 and y[1] > 0 for y in Y_k])
    FP_k = sum([y[0] > 0 and y[1] == 0 for y in Y_k])
    precision.append(TP_k / (TP_k + FP_k))
    recall.append(TP_k / (TP + FN))

plt.plot(recall,precision)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.show()
```



In [8]:
```
# Q5:
G = nx.karate_club_graph()

edges = set()
nodes = set()
for edge in urllib.request.urlopen("http://jmcauley.ucsd.edu/cse158/data/facebook/egonet.txt"):
    x, y = edge.split()
    x, y = int(x), int(y)
    edges.add((x, y))
    edges.add((y, x))
    nodes.add(x)
    nodes.add(y)

G = nx.Graph()
for e in edges:
    G.add_edge(e[0], e[1])

nx.draw(G, with_labels=True)
plt.show()
plt.clf()

print("number of connected components:\t", nx.number_connected_components(G))
connected_components = nx.connected_components(G)
number, C = max([(len(C), C) for C in connected_components])
print(number, "\tnodes are in the largest connected component")

# Number of connected components: 3
# 40 nodes are in the largest connected component
```




```
number of connected components:   3
40      nodes are in the largest connected component

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

In [9]:
```
# Q6:
sorted_C = sorted(C)
low = sorted_C[:int(len(sorted_C) / 2)]
high = sorted_C[int(len(sorted_C) / 2):]
cut = 0
for edge in edges:
    if ((edge[0] in low and edge[1] in high) or (edge[0] in low and edge
[1] in high)):
        cut += 1
degree_high = sum([G.degree(v) for v in high])
degree_low = sum([G.degree(v) for v in low])
normalized = 1 / 2 * (cut / degree_high + cut / degree_low)
print("normalized cut cost:\t", normalized)

# normalized cut cost: 0.42240587695133147
```

```
normalized cut cost:     0.42240587695133147
```

In [10]:
```
# Q7:
def cut_cost(node, high, low):
    half1 = [v for v in high]
    half2 = [v for v in low]
    if node in high:
        half1.remove(node)
        half2.append(node)
    else:
        half1.append(node)
        half2.remove(node)
    return nx.algorithms.cuts.normalized_cut_size(G, half1, half2) / 2

while True:
    temp = min([(cut_cost(node, high, low), node) for node in sorted_C])
    if normalized >= temp[0]:
        normalized = temp[0]
        if temp[1] in high:
            high.remove(temp[1])
            low.append(temp[1])
        else:
            low.remove(temp[1])
            high.append(temp[1])
    else:
        break

print("minimum normalized cut cost:\t", normalized)
print("split1:\t", high)
print("split2:\t", low)

# minimum normalized cut cost:   0.09817045961624274
#split1:          [825, 861, 863, 864, 876, 878, 882, 884, 886, 888, 889,
 893, 729, 804]
#split2:          [697, 703, 708, 713, 719, 745, 747, 753, 769, 772, 774,
 798, 800, 803,
#           805, 810, 811, 819, 828, 823, 830, 840, 880, 890, 869, 856]
```

```
minimum normalized cut cost:     0.09817045961624274
split1:  [825, 861, 863, 864, 876, 878, 882, 884, 886, 888, 889, 893, 7
29, 804]
split2:  [697, 703, 708, 713, 719, 745, 747, 753, 769, 772, 774, 798, 8
00, 803, 805, 810, 811, 819, 828, 823, 830, 840, 880, 890, 869, 856]
```

In [11]:

```
# Q8:
edge_num = sum([G.degree(v) for v in C])
low = sorted_C[:int(len(sorted_C) / 2)]
high = sorted_C[int(len(sorted_C) / 2):]

def modularity_Q(high, low):
    ekk = sum(
        [(edge[0] in high and edge[1] in high) or (edge[0] in low and ed
ge[1] in low) for edge in edges]) / edge_num
    ak1 = sum([(edge[0] in high) + (edge[1] in high) for edge in edges])
 / (edge_num * 2)
    ak2 = sum([(edge[0] in low) + (edge[1] in low) for edge in edges]) /
 (edge_num * 2)
    return ekk - ak1 ** 2 - ak2 ** 2

def greedy_modularity(node, high, low):
    half1 = [v for v in high]
    half2 = [v for v in low]
    if node in high:
        half1.remove(node)
        half2.append(node)
    else:
        half1.append(node)
        half2.remove(node)
    return modularity_Q(half1, half2)


modularity = 0
while True:
    temp = max([(greedy_modularity(node, high, low), node) for node in s
orted_C])
    if modularity <= temp[0]:
        modularity = temp[0]
        if temp[1] in high:
            high.remove(temp[1])
            low.append(temp[1])
        else:
            low.remove(temp[1])
            high.append(temp[1])
    else:
        break

print("maximum modularity:\t", modularity)
print("split1:\t", high)
print("split2:\t", low)

# maximum modularity:    0.3380165289256197
# split1:        [825, 856, 861, 863, 864, 869, 876, 878, 882, 884, 886,
 888, 889, 890, 893, 804, 729, 753, 811, 769, 798]
# split2:        [697, 703, 708, 713, 719, 745, 747, 772, 774, 800, 803,
 805, 810, 819, 828, 823, 830, 840, 880]
```

```
maximum modularity:      0.3380165289256197
split1:  [825, 856, 861, 863, 864, 869, 876, 878, 882, 884, 886, 888, 8
89, 890, 893, 804, 729, 753, 811, 769, 798]
split2:  [697, 703, 708, 713, 719, 745, 747, 772, 774, 800, 803, 805, 8
10, 819, 828, 823, 830, 840, 880]
```