HW 2 - A53252914

In [52]:
```
import numpy
import urllib
import scipy.optimize
import random
import ast
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
from sklearn.metrics import mean_squared_error, accuracy_score, confusion_matr
ix
from math import sqrt
from sklearn import svm
from sklearn import preprocessing
import numpy as np
```

In [53]:
```
def parseDatafromFile(fname):
  for l in open(fname):
    yield ast.literal_eval(l)
```

In [54]:
```
data = list(parseDatafromFile("C:/Users/ashak/Desktop/CSE258/beer_50000.json"
))
```

Code stub

In [55]:
```
def inner(x,y):
    return sum([x[i]*y[i] for i in range(len(x))])

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

# NEGATIVE Log-likelihood
def f(theta, X, y, lam):
  loglikelihood = 0
  for i in range(len(X)):
    logit = inner(X[i], theta)
    loglikelihood -= np.log(1 + np.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])
def ypred(y):
    return np.around(y)
```

In [56]:
```
def feature(datum):
    feat = [1, datum['review/taste'], datum['review/appearance'], datum['revie
w/aroma'], datum['review/palate'], datum['review/overall']]
    return feat

X = [feature(d) for d in data]
y = [d['beer/ABV'] >= 6.5 for d in data]
```

In [63]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33334, s
huffle = True)
X_trainfin, X_val, y_trainfin, y_val = train_test_split(X_train, y_train, test
_size=0.5, shuffle = True)
```

In [64]:
```
print(len(X_trainfin),len(X_test),len(X_val))
```

```
16666 16667 16667
```

In [65]:
```
###################################################
# Train                                           #
###################################################

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

1) Accuracy on Train, Validation and Test with lambda = 1

In [69]:
```
theta_opt = train(1)
print('Training accuracy:')
print(accuracy_score(y_trainfin, ypred(sigmoid(np.dot(X_trainfin, theta_opt
)))))
print('Validation accuracy:')
print(accuracy_score(y_val, ypred(sigmoid(np.dot(X_val, theta_opt)))))
print('Test accuracy:')
test_pred = ypred(sigmoid(np.dot(X_test, theta_opt)));
print(accuracy_score(y_test,test_pred))
```

```
Training accuracy:
0.7207488299531981
Validation accuracy:
0.7155456890862183
Test accuracy:
0.7139857202855943
```

2) Reporting +ves, -ves, False +ves, False -ves using test set, lamda = 1

In [70]:
```
tn, fp, fn, tp = confusion_matrix(y_test,test_pred).ravel()
print("True positive :",tp)
print("False positive :",fp)
print("False negative :",fn)
print("True negative :", tn)
print("Total postives :", tp+fp)
print("Total negatives :",tn+fn)
print("Total:", tp+fp+tn+fn)
```

```
True positive : 9041
False positive : 3497
False negative : 1270
True negative : 2859
Total postives : 12538
Total negatives : 4129
Total: 16667
```

4) a) Finding optimum lambda using Validation set

In [66]:
```
lamSet = [0,0.01, 0.1, 1, 100]
max_accuracy = -100
opt_lam = 1
for lam in lamSet:
    theta = train(lam)
    accuracy = accuracy_score(y_val, ypred(sigmoid(np.dot(X_val, theta))))
    print("Accuracy for lambda " , lam, " = ", accuracy)
    if(accuracy > max_accuracy):
        max_accuracy = accuracy
        opt_lam = lam
print("Optimal lambda = ", opt_lam)
```

```
Accuracy for lambda  0  =  0.7169856602867942
Accuracy for lambda  0.01  =  0.7148857022859543
Accuracy for lambda  0.1  =  0.7150056998860023
Accuracy for lambda  1  =  0.7155456890862183
Accuracy for lambda  100  =  0.6630467390652187
Optimal lambda =  0
```

4) b) Calculating Train, Validation and Test accuracy using the optimum lambda

In [67]:
```
theta_opt = train(opt_lam)
print('Training accuracy:')
print(accuracy_score(y_trainfin, ypred(sigmoid(np.dot(X_trainfin, theta_opt
)))))
print('Validation accuracy:')
print(accuracy_score(y_val, ypred(sigmoid(np.dot(X_val, theta_opt)))))
print('Test accuracy:')
test_pred = ypred(sigmoid(np.dot(X_test, theta_opt)));
print(accuracy_score(y_test,test_pred))
```

```
Training accuracy:
0.7236289451578063
Validation accuracy:
0.7169856602867942
Test accuracy:
0.7144057118857623
```

4) c) Reporting +ves, -ves, False +ves, False -ves using test set, Using optimum lambda

In [68]:
```
confusion_matrix(y_test,test_pred)
tn, fp, fn, tp = confusion_matrix(y_test,test_pred).ravel()
print("True positive :",tp)
print("False positive :",fp)
print("False negative :",fn)
print("True negative :", tn)
print("Total postives :", tp+fp)
print("Total negatives :",tn+fn)
```

```
True positive : 8956
False positive : 3405
False negative : 1355
True negative : 2951
Total postives : 12361
Total negatives : 4306
```

3) Code stub modification giving higher weight to false positives

- In logistic regression, We have to find the parameter theta that maximises the likelihood of the output variable y given x.
  - $L(\theta) = \prod_{y=1} P_{y|x} \prod_{y=0}(1 - P_{y|x})$
  - $P_{y|x} = \sigma(X.\theta)$
  - $\sigma(X.\theta) = \frac{1}{1+e^{X.\theta}}$
- Finding the theta that maxmises the log likelihood is the same as finding theta that minimises -ve log likelihood which becomes the cost function where we try to minimise the cost.
  - $LL(\theta) = \sum_{i=1}^{n} y_i(-log(1 + e^{X_i.\theta})) - (1 - y_i) * (log(1 + e^{X_i.\theta}) + (X_i.\theta))$
  - $NLL(\theta) = -(\sum_{i=1}^{n} y_i(-log(1 + e^{X_i.\theta})) - (1 - y_i) * (log(1 + e^{X_i.\theta}) + (X_i.\theta)))$
- Since Fp and Fn are equally weighed in this model, This can be rewritten as,
  - $NLL(\theta) = -(\sum_{i=1}^{n} c_1 y_i(-log(1 + e^{X_i.\theta})) - c_2(1 - y_i) * (log(1 + e^{X_i.\theta}) + (X_i.\theta)))$
  - Where, $c_1 = 1, c_2 = 1$
- So, I penalised the False positives by increasing the factor of penalisation when true y = 0 in the negative log likelihood function which is the cost function we try to minimise in logistic regression.
  - $NLL(\theta) = -(\sum_{i=1}^{n} 1 * y_i(-log(1 + e^{X_i.\theta})) - 10 * (1 - y_i) * (log(1 + e^{X_i.\theta}) + (X_i.\theta)))$
- I scale the -ve log likelihood function in in the term that comes into effect when true y = 0 by 10. This reduced the number of false +ves greatly which is what was expected.

In [46]:
```
def f_fp(theta, X, y, lam):
  loglikelihood = 0
  for i in range(len(X)):
    logit = inner(X[i], theta)
    #loglikelihood -= np.log(1 + np.exp(-logit))
    if y[i]:
        loglikelihood -= np.log(1 + np.exp(-logit))
    if not y[i]:
        loglikelihood -= 10* logit
        loglikelihood -= 10* np.log(1 + np.exp(-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_fp(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 y[i]:
            dl[k] += X[i][k] * (1 - sigmoid(logit))
      if not y[i]:
            dl[k] -=10* X[i][k]
            dl[k] +=10* X[i][k] * (1 - sigmoid(logit))
  for k in range(len(theta)):
    dl[k] -= lam*2*theta[k]
  return numpy.array([-x for x in dl])

def train_fp(lam):
    theta,_,_ = scipy.optimize.fmin_l_bfgs_b(f_fp, [0]*len(X[0]), fprime_fp, p
gtol = 10, args = (X_trainfin, y_trainfin, lam))
    return theta
```

In [71]:
```
theta_opt = train_fp(1)
print('Training accuracy:')
print(accuracy_score(y_trainfin, ypred(sigmoid(np.dot(X_trainfin, theta_opt
)))))
print('Validation accuracy:')
print(accuracy_score(y_val, ypred(sigmoid(np.dot(X_val, theta_opt)))))
print('Test accuracy:')
test_pred = ypred(sigmoid(np.dot(X_test, theta_opt)));
print(accuracy_score(y_test,test_pred))
```

```
Training accuracy:
0.43051722068882753
Validation accuracy:
0.4386512269754605
Test accuracy:
0.4400311993760125
```

In [72]:
```
confusion_matrix(y_test,test_pred)
tn, fp, fn, tp = confusion_matrix(y_test,test_pred).ravel()
print("True positive :",tp)
print("False positive :",fp)
print("False negative :",fn)
print("True negative :", tn)
print("Total postives :", tp+fp)
print("Total negatives :",tn+fn)
```

```
True positive : 1043
False positive : 65
False negative : 9268
True negative : 6291
Total postives : 1108
Total negatives : 15559
```

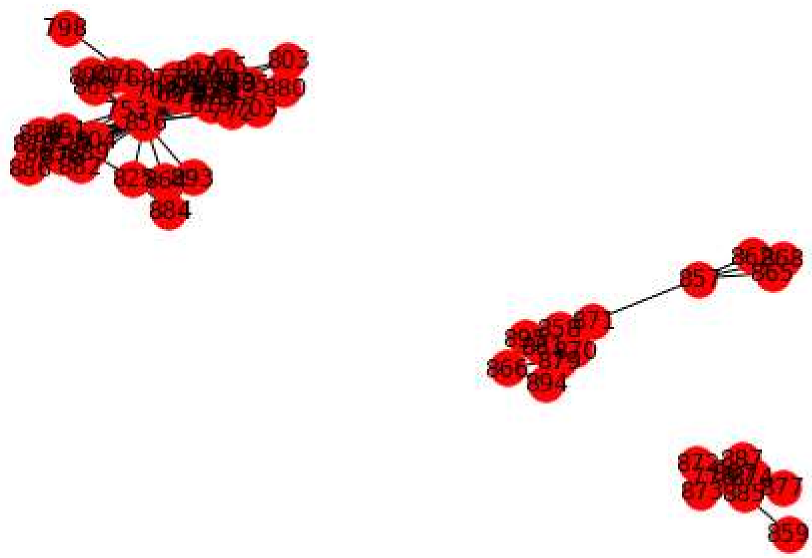
- It can be seen that FP decrease drastically as the model is trying its best not to predict false positives as we have given a high penalising factor if it does so.

Community Detection - FB egonet

In [22]:
```
import networkx as nx
```

In [23]:
```
g = nx.read_edgelist('C:/Users/ashak/Desktop/CSE258/egonet.txt', create_using
= nx.Graph(),nodetype = int)
print(nx.info(g))
nx.draw(g,with_labels = True)
plt.show()
```

```
Name:
Type: Graph
Number of nodes: 61
Number of edges: 270
Average degree:   8.8525
```



In [24]:
```
Gc = max(nx.connected_component_subgraphs(g), key=len)
```

5) Num of connected Components and num of nodes in largest connected component

In [25]:
```
print("Number of connected components :")
nx.number_connected_components(g)
```

```
Number of connected components :
```

Out[25]: 3

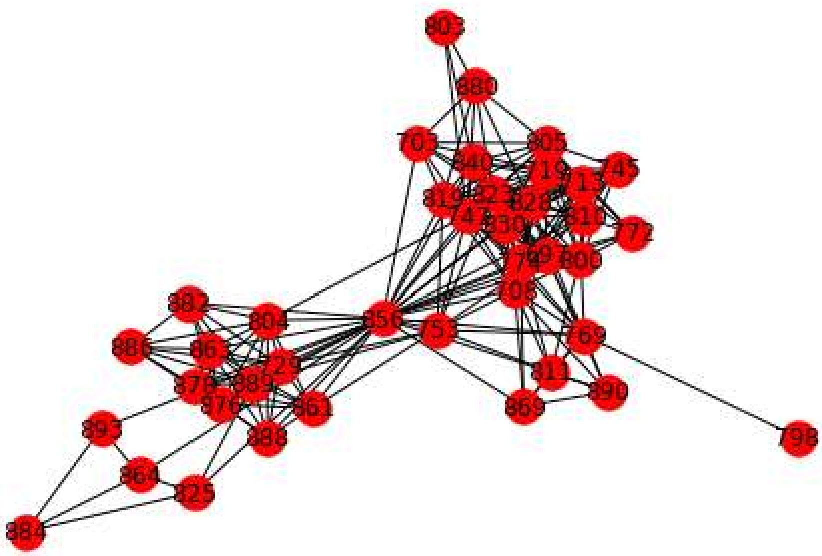
In [26]:
```
print('Number of nodes in the largest Connected component: ')
nx.number_of_nodes(Gc)
```

```
Number of nodes in the largest Connected component:
```

Out[26]: 40

In [27]:
```
nx.draw(Gc, with_labels = True)
print(nx.info(Gc))
plt.show()
```

```
Name: 
Type: Graph
Number of nodes: 40
Number of edges: 220
Average degree:  11.0000
```



In [28]:
```
lst = sorted(list(Gc.nodes()))
```

6) Calculating the normalised cut cost - 50:50 split

In [29]:
```
S = sorted(list(set(lst[:20])))
T = sorted(list(set(lst[20:])))
nx_in = 1/2 * nx.normalized_cut_size(Gc, S, T)
S1 = S
T1 = T
print("Community 1: ",S)
print("Community 2: ",T)
print("Normalised cut cost - 50/50 split: ", nx_in)
```

```
Community 1:  [697, 703, 708, 713, 719, 729, 745, 747, 753, 769, 772, 774, 79
8, 800, 803, 804, 805, 810, 811, 819]
Community 2:  [823, 825, 828, 830, 840, 856, 861, 863, 864, 869, 876, 878, 88
0, 882, 884, 886, 888, 889, 890, 893]
Normalised cut cost - 50/50 split:  0.4224058769513316
```

7) Finding the split that minimises normalised cut cost

In [30]:

```
%%time
cut_cost = float('inf');
cut_cost_past = 0;
nx_iter = float('inf');
opt_cost = 0;
T_1 = []
S_1 = []
x = 0;
while((cut_cost_past - cut_cost) > 0 or x==0):
    if x != 0:
        cut_cost_past = cut_cost;

    for i in S:
        Smod = [];
        Smod = [s for s in S if s!=i];
        Tmod = [];
        Tmod = Tmod + T;
        Tmod.append(i);

        nx_iter1 = 1/2 * nx.normalized_cut_size(Gc, Smod, Tmod)
        for j in T:
            tmod = [];
            tmod = [t for t in T if t!=j];
            smod = [];
            smod = smod + S;
            smod.append(j);

            nx_iter2 = 1/2 * nx.normalized_cut_size(Gc, smod, tmod)
            if (nx_iter1 < cut_cost or nx_iter2 < cut_cost):
                if nx_iter1 < nx_iter2:
                    if x != 0:
                        cut_cost_past = cut_cost;
                    cut_cost = nx_iter1;
                    opt_i = i
                    S_opt = Smod;
                    T_opt = Tmod;
                elif nx_iter1 >= nx_iter2:
                    if x != 0:
                        cut_cost_past = cut_cost;
                    cut_cost = nx_iter2;
                    opt_i = j
                    S_opt = smod;
                    T_opt = tmod;
            x = x+1;
    S_1 = S;
    T_1 = T;
    S = S_opt
    T = T_opt

print("Community 1: ",S_opt,len(S_opt))
print("Community 2: ",T_opt,len(T_opt))
print('Optimum cut cost : ', cut_cost)
```

```
Community 1:  [697, 703, 708, 713, 719, 745, 747, 753, 769, 772, 774, 798, 80
0, 803, 805, 810, 811, 819, 828, 823, 830, 840, 880, 890, 869, 856] 26
Community 2:  [825, 861, 863, 864, 876, 878, 882, 884, 886, 888, 889, 893, 72
9, 804] 14
Optimum cut cost :  0.09817045961624274
Wall time: 914 ms
```

In [31]:
```
from itertools import product
def get_modularity(network, community_dict):
    Q = 0
    G = network.copy()
    nx.set_edge_attributes(G, {e:1 for e in G.edges}, 'weight')
    A = nx.to_scipy_sparse_matrix(G).astype(float)
    if type(G) == nx.Graph:
        out_degree = in_degree = dict(nx.degree(G))
        M = 2.*(G.number_of_edges())
    nodes = list(G)
    Q = np.sum([A[i,j] - in_degree[nodes[i]]*out_degree[nodes[j]]/M for i, j i
n product(range(len(nodes)),range(len(nodes))) \
                if community_dict[nodes[i]] == community_dict[nodes[j]]])
    return Q / M
```

In [49]:
```
S = sorted(list(set(lst[:20])))
T = sorted(list(set(lst[20:])))
s1 = [1]*20;
t1 = [2]*20;
s1.extend(t1)
comm_dict = dict(zip(list(map(int, lst)),s1))
```

In [50]:
```
get_modularity(Gc,comm_dict)
```

Out[50]:
```
0.07681818181818181
```

8) Maximum modularity Split

In [51]:
```
%%time
cut_mod = -100;
cut_mod_past = -100;
opt_mod = 0;
T_1m = []
S_1m = []
x = 0;
while(cut_mod - cut_mod_past > 0 or x==0):
    if x != 0:
        cut_mod_past = cut_mod;
    for i in S:
        Smod = []
        Smod = [s for s in S if s!=i]
        Tmod = []
        Tmod = Tmod + T
        Tmod.append(i)
        S1 = [];
        T1 = [];
        S1 = [1]*len(Smod);
        T1 = [2]*len(Tmod);
        S1 = S1 + T1
        Smod_1 = [];
        Smod_1 = Smod
        Smod_1 = Smod_1 + Tmod
        comm_dict1 = dict(zip(list(map(int, Smod_1)),S1))
        nx_iter1_mod = get_modularity(Gc,comm_dict1)
        for j in T:
            tmod = []
            tmod = [t for t in T if t!=j]
            smod = []
            smod = smod + S
            smod.append(j)
            s1 = [];
            t1 = [];
            s1 = [1]*len(smod);
            t1 = [2]*len(tmod);
            s1 = s1 + t1
            smod_1 = [];
            smod_1 = smod
            smod_1 = smod_1 + tmod
            comm_dict2 = dict(zip(list(map(int, smod_1)),s1))
            nx_iter2_mod = get_modularity(Gc,comm_dict2)
            if (nx_iter1_mod > cut_mod or nx_iter2_mod > cut_mod):
                if(nx_iter1_mod >= nx_iter2_mod):
                    if x != 0:
                        cut_mod_past = cut_mod;
                    cut_mod = nx_iter1_mod;
                    opt_im = i
                    S_optm = Smod;
                    T_optm = Tmod;
                elif (nx_iter1_mod < nx_iter2_mod):
                    if x != 0:
                        cut_mod_past = cut_mod;
                    cut_mod = nx_iter2_mod;
                    opt_im = j
                    S_optm = smod;
```

```
                    T_optm = tmod;
            x = x+1;
    S_1m = S;
    T_1m = T;
    S = S_optm
    T = T_optm


print("Community 1: ",sorted(S_optm),len(S_optm))
print("Community 2: ",sorted(T_optm),len(T_optm))
print('Max mod : ', cut_mod)
```

```
Community 1:  [697, 703, 708, 713, 719, 745, 747, 772, 774, 800, 803, 805, 81
0, 819, 823, 828, 830, 840, 880] 19
Community 2:  [729, 753, 769, 798, 804, 811, 825, 856, 861, 863, 864, 869, 87
6, 878, 882, 884, 886, 888, 889, 890, 893] 21
Max mod :  0.33801652892561984
Wall time: 4min 20s
```