In [216]:

```
import pandas as pd
import numpy as np
import json
import matplotlib.pyplot as plt
import networkx as nx

pd.set_option('display.max_columns', None)

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
from sklearn.pipeline import Pipeline

from networkx.algorithms.components import number_connected_components
from networkx.algorithms.cuts import normalized_cut_size
```

Tasks - Diagnostics:

- We'll start by building a classifier that predicts whether a beer is highly alcoholic (ABV greater than 7 percent).
- First, randomly shuffle the data and split it into 50%/50% train/test fractions.

In [100]:

```
#Load beer dataset
def parseData(fname):
    for l in open(fname):
        yield eval(l)

beer = list(parseData("data/beer_50000.json"))
beer = pd.DataFrame(beer)
beer.head(1)
```

Out[100]:

	review/appearance	beer/style	review/palate	review/taste	beer/name	review/timeUnix	beer/A
0	2.5	Hefeweizen	1.5	1.5	Sausa Weizen	1234817823	

Problem 1

- We'll use the style of the beer to predict its ABV.
- Construct a one-hot encoding of the beer style, for those categories that appear in more than 1,000 reviews.

In [101]:

```
#Filtering beer style categoreis more frequent than 1000 reviews.
beer_style_l = list(beer['beer/style'].value_counts(dropna=False)[beer['beer/style'].value
_counts()>1000].index)
```

In [102]:

```
#One-hot-Econding beer styles.
for i in beer_style_l:
    beer[i] = beer['beer/style'] == i
beer.head(1)
```

Out[102]:

	review/appearance	beer/style	review/palate	review/taste	beer/name	review/timeUnix	beer/A
0	2.5	Hefeweizen	1.5	1.5	Sausa Weizen	1234817823	

- Train a logistic regressor using this one-hot encoding to predict whether beers have an ABV greater than 7 percent (i.e., d['beer/ABV'] > 7).
- Train the classifier on the training set and report its performance in terms of the accuracy and Balanced Error Rate (BER) on the test set, using a regularization constant of C = 10.
- For all experiments use the class weight='balanced' option

In [103]:

```
#Creating the parameter abv_gt_7: true if beer/ABV > 7 false otherwise
beer['abv_gt_7'] = beer['beer/ABV'] > 7
```

In [104]:

```
#Train-Test-Split
X = beer[beer_style_l]
y = beer['abv_gt_7']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
```

In [105]:

```
#Fit the Model
logreg = LogisticRegression(C=10, class_weight='balanced')
logreg.fit(X_train, y_train)
```

Out[105]:

```
LogisticRegression(C=10, class_weight='balanced')
```

In [106]:

```
#Find the accuracy of the model on the test data
y_pred = logreg.predict(X_test)
print('Accuracy of logistic regression classifier on the test set: '+ str(logreg.score(X_t
est, y_test)))
```

```
Accuracy of logistic regression classifier on the test set: 0.84996
```

In [107]:

```
#Find the Balanced Error Rate of the model on the test data.
cm = confusion_matrix(y_test, y_pred)
ber = 0.5*((cm[0][1]/(cm[0][0]+cm[0][1]))+(cm[1][0]/(cm[1][0]+cm[1][1])))
print("Balanced Error Rate: " + str(ber))
```

```
Balanced Error Rate: 0.15950823088547797
```

Problem 2

- Extend your model to include two additional features: (1) a vector of five ratings (review/aroma, review/overall, etc.); and (2) the review length (in characters).
- The length feature should be scaled to be between 0 and 1 by dividing by the maximum length.
- Using the same value of C from the previous question, report the BER of the new classifier

In [108]:

```
#Feature engineer parameter revieww_length
beer['review_length'] = [len(x) for x in beer['review/text'].str.split()]
beer['review_length'] /= max(beer['review_length'])
```

In [109]:

```
beer.columns
```

Out[109]:

```
Index(['review/appearance', 'beer/style', 'review/palate', 'review/taste',
       'beer/name', 'review/timeUnix', 'beer/ABV', 'beer/beerId',
       'beer/brewerId', 'review/timeStruct', 'review/overall', 'review/text',
       'user/profileName', 'review/aroma', 'user/gender', 'user/birthdayRaw',
       'user/birthdayUnix', 'user/ageInSeconds',
       'American Double / Imperial Stout', 'American IPA',
       'American Double / Imperial IPA', 'Scotch Ale / Wee Heavy',
       'Russian Imperial Stout', 'American Pale Ale (APA)', 'American Porte
r',
       'Rauchbier', 'Rye Beer', 'Czech Pilsener', 'Fruit / Vegetable Beer',
       'English Pale Ale', 'Old Ale', 'abv_gt_7', 'review_length'],
      dtype='object')
```

In [110]:

```
#Feature selection
col = beer_style_l +['review/appearance', 'review/palate', 'review/taste', 'review/overal
l', 'review/aroma', 'review_length']
```

In [111]:

```
#Train-Test-Split
X = beer[col]
y = beer['abv_gt_7']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
```

In [112]:

```
#Fit the Model
logreg = LogisticRegression(C=10, class_weight='balanced',max_iter=1000)
logreg.fit(X_train, y_train)
```

Out[112]:

```
LogisticRegression(C=10, class_weight='balanced', max_iter=1000)
```

In [113]:

```
#Find the accuracy of the model on the test data
y_pred = logreg.predict(X_test)
print('Accuracy of logistic regression classifier on the test set: '+ str(logreg.score(X_t
est, y_test)))
```

```
Accuracy of logistic regression classifier on the test set: 0.86212
```

In [114]:

```
#Find the Balanced Error Rate of the model on the test data.
cm = confusion_matrix(y_test, y_pred)
ber = 0.5*((cm[0][1]/(cm[0][0]+cm[0][1]))+(cm[1][0]/(cm[1][0]+cm[1][1])))
print("Balanced Error Rate: " + str(ber))
```

```
Balanced Error Rate: 0.1422613250925131
```

Problem 3

- Implement a complete regularization pipeline with the balanced classifier.
- Split your test data from above in half so that you have 50%/25%/25% train/validation/test fractions. Consider values of C in the range {10^−6, 10^−5, 10^−4, 10^−3}.
- Report (or plot) the train, validation, and test BER for each value of C. Based on these values, which classifier would you select (in terms of generalization performance) and why.

In [147]:

```
#Train-Test-Validation-Split
X = beer[col]
y = beer['abv_gt_7']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.5, random_
state=1)
```

In [148]:

```
#Cross Validation
def find_ber(x, y, c):
    logreg = LogisticRegression(C=c, class_weight='balanced', max_iter=1000)
    logreg.fit(X_train, y_train)

    y_pred = logreg.predict(x)
    cm = confusion_matrix(y, y_pred)
    ber = 0.5*((cm[0][1]/(cm[0][0]+cm[0][1]))+(cm[1][0]/(cm[1][0]+cm[1][1])))

    return ber


train_ber_l = []
test_ber_l = []
val_ber_l = []

c_range = [1e-6, 1e-5, 1e-4, 1e-3]

for c in c_range:
    train_ber_l.append(find_ber(X_train, y_train,c))
    test_ber_l.append(find_ber(X_test,y_test,c))
    val_ber_l.append(find_ber(X_val,y_val,c))
```

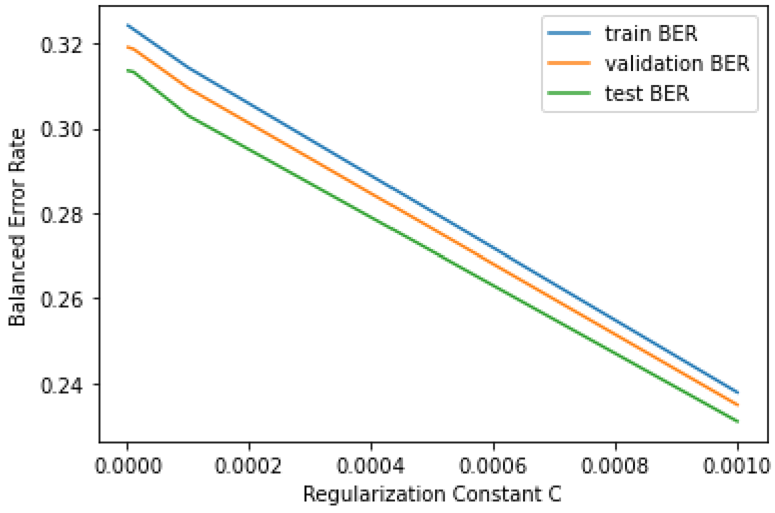
In [149]:

```
#List of computed BER (4 in each set, 12 total)
print(train_ber_l)
print(test_ber_l)
print(val_ber_l)
```

```
[0.32418952875267754, 0.3232863733728786, 0.3142319059976932, 0.2377664668808
7]
[0.31353128599342034, 0.31323323090170563, 0.30290010489865693, 0.23092492561
203143]
[0.3190482228677, 0.3186307293963302, 0.3093905063241834, 0.2348309055670827]
```

In [150]:

```
#Plot the result
plt.xlabel('Regularization Constant C')
plt.ylabel('Balanced Error Rate')
plt.plot(c_range, train_ber_l, label="train BER")
plt.plot(c_range, val_ber_l, label="validation BER")
plt.plot(c_range, test_ber_l, label="test BER")
plt.legend()
plt.show()
```



From the graph above, I would select C = 10^-3 because all training set, validation set and test set data shows a negative linear relationship between C and BER. That is to say, the balanced error rate decreases with increasing C. Furthermore, since all three plots show the same trend, we cannot find patterns of overfitting. Therefore, if we want to minimize the BER, we chooses the largest c among the range, which is 10^-3.

Problem 4

An ablation study measures the marginal benefit of various features by re-training the model with one feature ‘ablated’ (i.e., deleted) at a time. Considering each of the three features in your classifier above (i.e., beer style, ratings, and length), report the BER with only the other two features and the third deleted.

In [151]:

```
#Feature selection
ratings = ['review/appearance', 'review/palate', 'review/taste', 'review/overall', 'revie
w/aroma']
length = ['review_length']
```

First let's use only beer style and ratings as our features

In [152]:

```
#Train-Test-Validation-Split
X = beer[beer_style_l+ratings]
y = beer['abv_gt_7']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.5, random_
state=1)
```

In [153]:

```
#Find the BER for each c
train_ber_l = []
test_ber_l = []
val_ber_l = []

c_range = [1e-6, 1e-5, 1e-4, 1e-3]

for c in c_range:
    train_ber_l.append(find_ber(X_train, y_train,c))
    test_ber_l.append(find_ber(X_test,y_test,c))
    val_ber_l.append(find_ber(X_val,y_val,c))
```

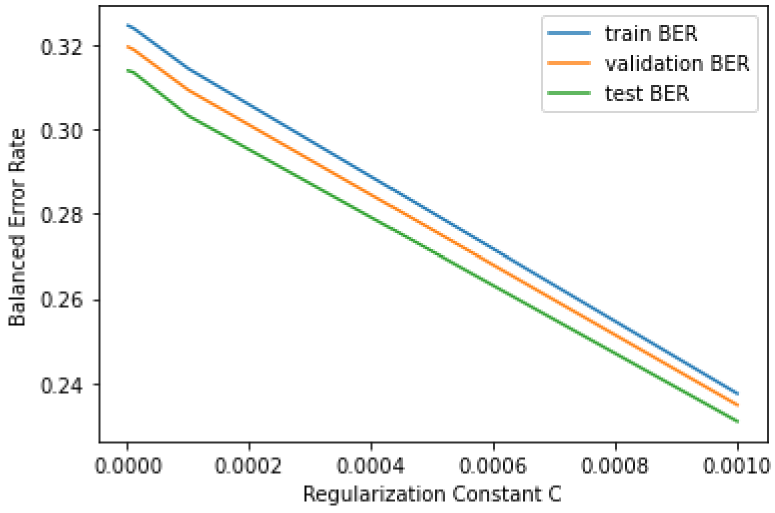
In [154]:

```
#List of computed BER (4 in each set, 12 total)
print(train_ber_l)
print(test_ber_l)
print(val_ber_l)
```

```
[0.32459965603888613, 0.32395601623002146, 0.3143807155215027, 0.237531152166
74905]
[0.3138979245405383, 0.31349106394819326, 0.3032110714050654, 0.2309456712798
4209]
[0.31953176558795054, 0.31889952849989095, 0.30929438911245105, 0.23483090556
70827]
```

In [155]:

```
#Plot the result
plt.xlabel('Regularization Constant C')
plt.ylabel('Balanced Error Rate')
plt.plot(c_range, train_ber_l, label="train BER")
plt.plot(c_range, val_ber_l, label="validation BER")
plt.plot(c_range, test_ber_l, label="test BER")
plt.legend()
plt.show()
```



Next let's use only beer style and length as our features

In [156]:

```
#Train-Test-Validation-Split
X = beer[beer_style_l+length]
y = beer['abv_gt_7']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.5, random_
state=1)
```

In [157]:

```
#Find the BER for each c
train_ber_l = []
test_ber_l = []
val_ber_l = []

c_range = [1e-6, 1e-5, 1e-4, 1e-3]

for c in c_range:
    train_ber_l.append(find_ber(X_train, y_train,c))
    test_ber_l.append(find_ber(X_test,y_test,c))
    val_ber_l.append(find_ber(X_val,y_val,c))
```

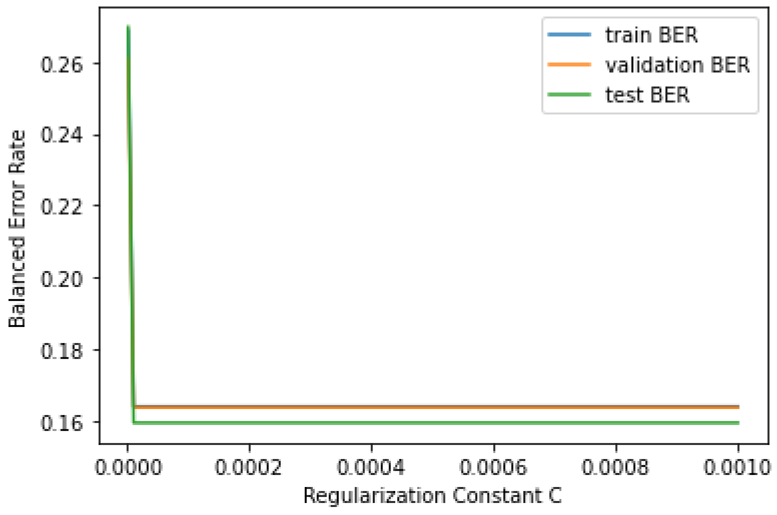
In [158]:

```
#List of computed BER (4 in each set, 12 total)
print(train_ber_l)
print(test_ber_l)
print(val_ber_l)
```

```
[0.2687268289668809, 0.1641062675070028, 0.1641062675070028, 0.16410626750700
28]
[0.26978907433517896, 0.15950823088547797, 0.15950823088547797, 0.15950823088
547797]
[0.2610092129832204, 0.1637920202994615, 0.1637920202994615, 0.16379202029946
15]
```

In [159]:

```
#Plot the result
plt.xlabel('Regularization Constant C')
plt.ylabel('Balanced Error Rate')
plt.plot(c_range, train_ber_l, label="train BER")
plt.plot(c_range, val_ber_l, label="validation BER")
plt.plot(c_range, test_ber_l, label="test BER")
plt.legend()
plt.show()
```



Lastly let's use only ratings and length as our features

In [160]:

```
#Train-Test-Validation-Split
X = beer[ratings+length]
y = beer['abv_gt_7']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.5, random_
state=1)
```

In [161]:

```
#Find the BER for each c
train_ber_l = []
test_ber_l = []
val_ber_l = []

c_range = [1e-6, 1e-5, 1e-4, 1e-3]

for c in c_range:
    train_ber_l.append(find_ber(X_train, y_train,c))
    test_ber_l.append(find_ber(X_test,y_test,c))
    val_ber_l.append(find_ber(X_val,y_val,c))
```

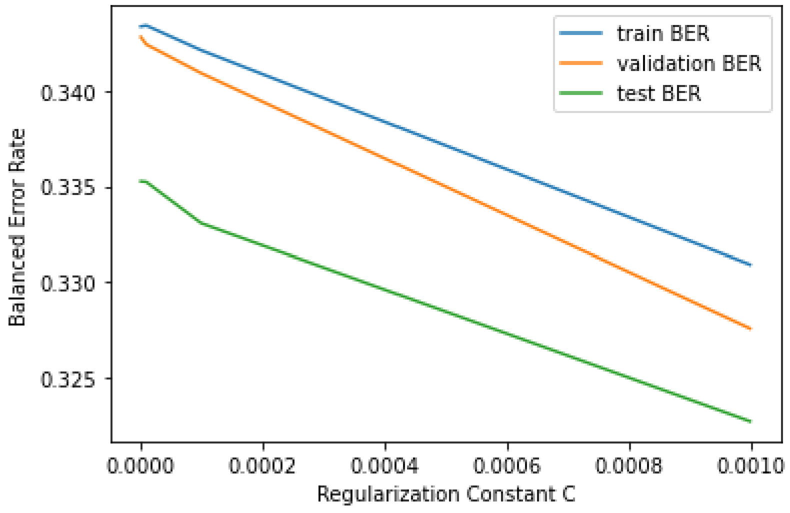
In [162]:

```
#List of computed BER (4 in each set, 12 total)
print(train_ber_l)
print(test_ber_l)
print(val_ber_l)
```

```
[0.34338492750041194, 0.34343513140550336, 0.3421375535508321, 0.330897542840
66564]
[0.33525386074596447, 0.3352227422442485, 0.33306015740447686, 0.322679302869
2113]
[0.34282694230362865, 0.3424424734566991, 0.34093318770103853, 0.327551221655
2888]
```

In [163]:

```
#Plot the result
plt.xlabel('Regularization Constant C')
plt.ylabel('Balanced Error Rate')
plt.plot(c_range, train_ber_l, label="train BER")
plt.plot(c_range, val_ber_l, label="validation BER")
plt.plot(c_range, test_ber_l, label="test BER")
plt.legend()
plt.show()
```



Task (Community Detection):

Download the Facebook ego-network data.

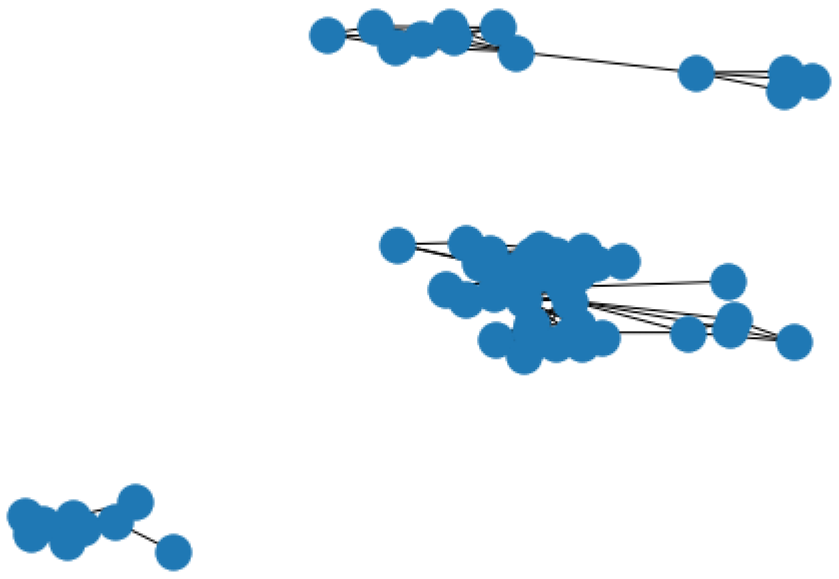
Problem 6

- How many connected components are in the graph, and how many nodes are in the largest connected component

In [181]:

```
#Load Data
edges = set()
nodes = set()
for edge in open("data/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)

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



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

In [184]:

```
print("There are " + str(number_connected_components(G)) + "connected components in the gr
aph.")
```

```
There are 3connected components in the graph.
```

In [193]:

```
print("There are " + str(len(sorted(nx.connected_components(G))[0])) + " nodes in the larg
est connected component.")
```

```
There are 40 nodes in the largest connected component.
```

Next we’ll implement a ‘greedy’ version of normalized cuts, using just the largest connected component found above. First, split it into two equal halves, just by taking the 50% of nodes with the lowest and 50% with the highest IDs.

In [283]:

```
#Split into lower half and higher half
largest_component = sorted(nx.connected_components(G))[0]
half_index = int(len(largest_component)/2)
lower_half = sorted(largest_component)[:half_index]
higher_half = sorted(largest_component)[half_index:]
```

Problem 7

What is the normalized-cut cost of the 50/50 split you found above.

In [284]:

```
print("The normalized-cut cost of the 50/50 split is " +str(0.5*normalized_cut_size(G, hig
her_half, lower_half)))
```

```
The normalized-cut cost of the 50/50 split is 0.4224058769513316
```

Now we’ll implement our greedy algorithm as follows: during each step, we’ll move one node from one cluster to the other, choosing whichever move minimizes the resulting normalized cut cost (in case of a tie, pick the node with the lower ID). Repeat this until the cost can’t be reduced any further.

In [286]:

```
normalized_cut_cost = float('inf')
normalized_cut_cost_prev = float('inf')

cluster_1 = list(largest_component)
cluster_2 = []

while normalized_cut_cost <= normalized_cut_cost_prev:
    normalized_cut_cost_temp = float('inf')
    for n1 in cluster_1:
        temp_1 = list(cluster_1)
        temp_1.remove(n1)
        temp_2 = cluster_2.copy()
        temp_2.append(n1)
        ncc = 0.5*normalized_cut_size(G, temp_1, temp_2)
        if ncc == normalized_cut_cost_temp:
            minimizing_node = min(minimizing_node, n1)
            normalized_cut_cost_temp = ncc
        if ncc < normalized_cut_cost_temp:
            minimizing_node = n1
            normalized_cut_cost_temp = ncc
    cluster_1.remove(minimizing_node)
    cluster_2.append(minimizing_node)

    normalized_cut_cost_prev = normalized_cut_cost
    normalized_cut_cost = 0.5*normalized_cut_size(G, cluster_1, cluster_2)

    print(cluster_1,cluster_2)
    print(normalized_cut_cost, normalized_cut_cost_prev)
```

```
[769, 772, 774, 800, 803, 804, 805, 810, 811, 819, 823, 825, 697, 828, 830, 7
03, 708, 840, 713, 719, 856, 729, 861, 863, 864, 869, 745, 747, 876, 878, 88
0, 753, 882, 884, 886, 888, 889, 890, 893] [798]
0.5011389521640092 inf
[772, 774, 800, 803, 804, 805, 810, 811, 819, 823, 825, 697, 828, 830, 703, 7
08, 840, 713, 719, 856, 729, 861, 863, 864, 869, 745, 747, 876, 878, 880, 75
3, 882, 884, 886, 888, 889, 890, 893] [798, 769]
0.43595748513781307 0.5011389521640092
[772, 774, 800, 803, 804, 805, 810, 811, 819, 823, 825, 697, 828, 830, 703, 7
08, 840, 713, 719, 856, 729, 861, 863, 864, 745, 747, 876, 878, 880, 753, 88
2, 884, 886, 888, 889, 890, 893] [798, 769, 869]
0.4125515689461183 0.43595748513781307
[772, 774, 800, 803, 804, 805, 810, 811, 819, 823, 825, 697, 828, 830, 703, 7
08, 840, 713, 719, 856, 729, 861, 863, 864, 745, 747, 876, 878, 880, 753, 88
2, 884, 886, 888, 889, 893] [798, 769, 869, 890]
0.36048192771084336 0.4125515689461183
[772, 774, 800, 803, 804, 805, 810, 819, 823, 825, 697, 828, 830, 703, 708, 8
40, 713, 719, 856, 729, 861, 863, 864, 745, 747, 876, 878, 880, 753, 882, 88
4, 886, 888, 889, 893] [798, 769, 869, 890, 811]
0.30330882352941174 0.36048192771084336
[772, 774, 800, 803, 804, 805, 810, 819, 823, 825, 697, 828, 830, 703, 840, 7
13, 719, 856, 729, 861, 863, 864, 745, 747, 876, 878, 880, 753, 882, 884, 88
6, 888, 889, 893] [798, 769, 869, 890, 811, 708]
0.31003705830158146 0.30330882352941174
```

Problem 8

What are the elements of the split, and what is its normalized cut cost?

- Element in cluster 1: [772, 774, 800, 803, 804, 805, 810, 819, 823, 825, 697, 828, 830, 703, 708, 840, 713, 719, 856, 729, 861, 863, 864, 745, 747, 876, 878, 880, 753, 882, 884, 886, 888, 889, 893]
- Element in cluster 2: [798, 769, 869, 890, 811]
- Normalized cut cost: 0.303.

```
In [ ]:
```