In [ ]:

```
import numpy
import scipy.optimize
import random
from sklearn.decomposition import PCA
from collections import defaultdict
from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import pairwise_distances




list1 = [1,2,3,4]
list2 = list1
list2.remove(1)
print list1

### PCA on beer reviews ###

def parseData():
  for l in open("/Users/yansong/Documents/CSE255/beer_50000.json"):
    yield eval(l)


print "Reading data..."
data = list(parseData())
print "done"

X = [[x['review/overall'], x['review/taste'], x['review/aroma'], x['review/appearance'], x['review/palate']] for x in data]

pca = PCA(n_components=5)
pca.fit(X)

Xdata = numpy.matrix(X)

def problemB():
  basis = pca.components_
  basis_transpose = basis.transpose()

  XNewData = numpy.dot(X, basis_transpose)
  #print type(XNewData)
  b3 = 0.0
  b4 = 0.0
  for i in range(50000):
    b3 += XNewData[i][3]
    b4 += XNewData[i][4]
  b3 /= 50000
  b4 /= 50000
  reError = 0.0
  #print XNewData[0][3]
  for i in range(50000):
    reError += pow(XNewData[i][3] - b3, 2)
    reError += pow(XNewData[i][4] - b4, 2)
  print reError

#NewData = pca.fit_transform(X)

#print pca.components_
#print X
#print NewData

def problemC():
  kmeansX = X[:500]
  kmeans = AgglomerativeClustering(n_clusters = 2,linkage ='ward')
  kmeans.fit(kmeansX)
  label = kmeans.labels_
  total0 = 0
  total1 = 0
  total0_array = [0.0 for i in range(len(kmeansX[0]))]
  total1_array = [0.0 for i in range(len(kmeansX[0]))]
  for i in range(500):
    if (label[i]):
      total1 += 1
      for j in range(5):
        total1_array[j] += kmeansX[i][j]
    else :
      total0 += 1
      for j in range(5):
        total0_array[j] += kmeansX[i][j]
  print "total0 is ", total0
  print "total1 is ", total1

  for i in range(5):
    total0_array[i] /= float(total0)
    print total0_array[i],
  print

  for i in range(5):
    total1_array[i] /= float(total1)
    print total1_array[i],
  print

  reconstruction_error = 0.0
  for i in range(500):
      if (label[i] == 0):
          reconstruction_error += calError(kmeansX[i], total0_array)
      else:
          reconstruction_error += calError(kmeansX[i], total1_array)
  print "the total reconstruction error is ", reconstruction_error


def calError(point1, point2):
  error = 0.0
  for i in range(5):
    error += pow(point1[i] - point2[i], 2)
  print point1, point2, error
  return error


print "the ans of the third problem"
problemC()

def problemD():
  kmeansX = X[:500]
  kmeans = KMeans(n_clusters = 2)
  kmeans.fit(kmeansX)
  labels = kmeans.labels_
  centers = kmeans.cluster_centers_
  reError = 0.0
  for i in range(500):
    if (labels[i] == 0):
      reError += calError(kmeansX[i], centers[0])
    else :
      reError += calError(kmeansX[i], centers[1])
    print reError
  print reError




def problemA():
  newData = []
  for feature in range(5):
    total = 0.0
    for i in X:
      total += i[feature]
    total /= 50000
    newData.append(total)
  error = 0.0
  for data in X:
    tmpError = 0.0
    for j in range(5):
      tmpError += pow(abs(data[j] - newData[j]),2)
    error += tmpError
  print error
```

```
#problemC()

def normalized_cut_cost(first, second, edges, largest_connected_component):
  edge_num = 0
  for n1 in first:
    for n2 in second:
      if not ((n1, n2) in edges): continue
      edge_num += 1
  degree_first = sum(list(largest_connected_component.degree(first).values()))
  degree_second = sum(list(largest_connected_component.degree(second).values()))
  #print degree_first, degree_second
  result = (float(edge_num) / float(degree_first) + float(edge_num) / float(degree_second)) / 2.0
  return result


### Network visualization ###
import networkx as nx
import matplotlib.pyplot as plt
'''
# Karate club
G = nx.karate_club_graph()
nx.draw(G)
plt.show()
plt.clf()
'''
edges = set()
nodes = set()
for edge in open("/Users/yansong/Documents/CSE255/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])
print nx.number_connected_components(G)
connected_components = nx.connected_components(G);
#print list(connected_components[0])
counter = 0
nodes = []
for i in connected_components:
  if counter == 0:
    for j in i:
      nodes.append(j)
  counter += 1
print nodes
largest_connected_component = G.subgraph(nodes)
nx.draw(largest_connected_component)
#nx.draw(G)
#plt.show()
#plt.clf()
nodes.sort()
print len(nodes)
first = []
second = []
for i in range(len(nodes)):
  if i < len(nodes) / 2:
    first.append(nodes[i])
  else:
    second.append(nodes[i])
print first
print second
edge_num = 0
for n1 in first:
  for n2 in second:
    if not ((n1, n2) in edges): continue
    edge_num += 1
degree_first = sum(list(largest_connected_component.degree(first).values()))
degree_second = sum(list(largest_connected_component.degree(second).values()))
print degree_first, degree_second
result = (float(edge_num) / float(degree_first) + float(edge_num) / float(degree_second)) / 2.0
print result

print "test", normalized_cut_cost(first, second, edges, largest_connected_component)

minimize_cost = result
while len(first) > 0 and len(second) > 0:
  partial_min = minimize_cost
  node = 0
  for n1 in first:
    tmp_first = list(first)
    tmp_first.remove(n1)
    tmp_second = list(second)
    tmp_second.append(n1)
    tmp_first.sort()
    tmp_second.sort()
    result = normalized_cut_cost(tmp_first, tmp_second, edges, largest_connected_component)
    if result < partial_min:
      partial_min = result
      node = n1
  for n2 in second:
    tmp_first = list(first)
    tmp_first.append(n2)
    tmp_second = list(second)
    tmp_second.remove(n2)
    tmp_first.sort()
    tmp_second.sort()
    result = normalized_cut_cost(tmp_first, tmp_second, edges, largest
_connected_component)
    if result < partial_min:
      partial_min = result
      node = n2
  if partial_min < minimize_cost:
    minimize_cost = partial_min
    if first.count(node) == 1:
      first.remove(node)
      second.append(node)
    else:
      first.append(node)
      second.remove(node)
  else:
    break
print minimize_cost
print "first is", first
print "second is", second
print normalized_cut_cost(first, second, edges, largest_connected_comp
onent)

one = []
two = []
three = []
four = []
for i in range(len(nodes)):
  if i < 10:
    one.append(nodes[i])
  elif i < 20:
    two.append(nodes[i])
  elif i < 30:
    three.append(nodes[i])
  else:
    four.append(nodes[i])
community4 = [one, two, three, four]
print "initial community 4", community4


def normarlized4(community4, edges, largest_connected_component):
  edge_list = []
  for i in range(4):
    edge_num = 0
    for j in range(4):
      if i == j:
        continue
      else:
        for n1 in community4[i]:
          for n2 in community4[j]:
            if not((n1, n2) in edges): continue
            edge_num += 1
    edge_list.append(edge_num)
  degree = []
  for i in range(4):
    tmp = sum(list(largest_connected_component.degree(community4[i]).v
alues()))
    degree.append(tmp)
  #print degree_first, degree_second
  total = 0.0
  for i in range(4):
    total += float(edge_list[i]) / float(degree[i])
  result = total / 4.0
  return result

def larger_than_zero(community):
  for i in range(4):
    if len(community[i]) <= 0:
      return False
  return True


minimize_cost = normarlized4(community4, edges, largest_connected_comp
onent)
larger_than_zero(community4)
#print "test", community4
#print "the last problem", minimize_cost
while (larger_than_zero(community4)):
  partial_min = minimize_cost
  pos_from = 0
  pos_to = 0
  node = 0
  for i in range(4):
    for j in range(4):
      if i == j:
        continue
      else:
        for n1 in community4[i]:
          tmp = []
          for k in range(4):
            tmptmp = list(community4[k])
            tmp.append(tmptmp)
          #tmp = list(community4)
          tmp[i].remove(n1)
          tmp[j].append(n1)
          result = normarlized4(tmp, edges, largest_connected_componen
t)
          if result < partial_min:
            #print "partial min is", partial_min
            #print "delete from", i
            #print "to", j
            #print "the node to be deleted", n1
            partial_min = result
            pos_from = i
            pos_to = j
            node = n1
  if partial_min < minimize_cost:
    #print community4
    #print "the node to be deleted", node
    #print "from which ", pos_from
    #print pos_to
    minimize_cost = partial_min
    community4[pos_from].remove(node)
    community4[pos_to].append(node)
    for i in range(4):
      community4[i].sort()
  else:
    break

print minimize_cost
for i in range(4):
  community4[i].sort()
print community4

'''
### Find all 3 and 4-cliques in the graph ###
cliques3 = set()
cliques4 = set()
for n1 in nodes:
  for n2 in nodes:
    if not ((n1,n2) in edges): continue
    for n3 in nodes:
      if not ((n1,n3) in edges): continue
      if not ((n2,n3) in edges): continue
      clique = [n1,n2,n3]
      clique.sort()
      cliques3.add(tuple(clique))
      for n4 in nodes:
        if not ((n1,n4) in edges): continue
        if not ((n2,n4) in edges): continue
        if not ((n3,n4) in edges): continue
        clique = [n1,n2,n3,n4]
        clique.sort()
        cliques4.add(tuple(clique))
'''
```