Bayesian Models for Machine Learning
Problem Set #4

Si Kai Lee sl3950@columbia.edu

December 11, 2016

Problem 1

b

As K increases, the log likelihood increases. As we are running maximum likelihood-EM, increasing the number of clusters leads to an increase log likelihood as each point becomes closer to a cluster which leads to overfitting. If we carry out model selection using log likelihood as the criteria, we would select the model with the same number of clusters as points.

c

The number of clusters increases as the K defined increases. However, at $K = 8$ and $K = 10$ the clusters become more arbitrary which suggests overfitting.

In [1]:
```
# Use gammaln for stability
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
from scipy.special import digamma, gammaln, multigammaln
from scipy.stats import multivariate_normal, wishart
from sklearn.covariance import empirical_covariance
```

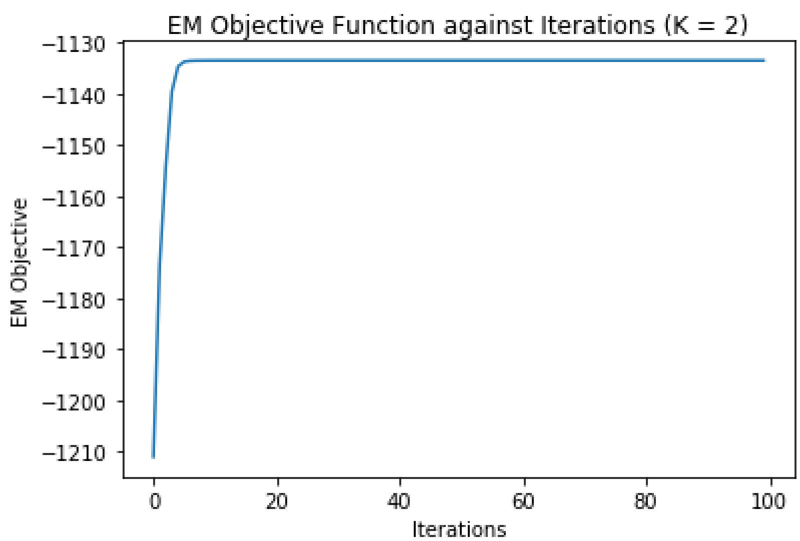
In [2]:
```
# Load data
data = loadmat('hw4_data_mat/data.mat')
X = data['X']
d = X.shape[0]
num = X.shape[1]
```

In [3]:
```
def EM_GMM(X, k):
    # Initialise
    pi = np.ones(k)
    mu = np.random.rand(d, k)
    lamda = [np.identity(d) for i in range(k)]
    LL = []

    for a in range(100):
        # E-Step
        c = np.empty((k, num))
        for i in range(k):
            c[i, :] = map(lambda j: pi[i] * multivariate_normal.pdf(X[:,
 j], mu[:, i], np.linalg.inv(lamda[i])), range(num))
        for j in range(num):
            c[:, j] = c[:, j] / float(np.sum(c[:, j]))

        # M-Step
        n = np.sum(c, axis=1)
        for i in range(k):
            mu[:, i] = 1/float(n[i]) * np.dot(X, c[i, :].T)
            x_minus_mu_j = X.T - mu[:, i]
            Sigma = 1/float(n[i]) * sum(map(lambda j: c[i, j] *
(np.dot(x_minus_mu_j[j].reshape((d, 1)), x_minus_mu_j[j].reshape((1,
d)))), range(num)))
            lamda[i] = np.linalg.inv(Sigma)
        pi = n / float(250)

        # Calculate log-likelihood
        LL_t = 0
        for i in range(num):
            LL_t += np.log(sum(map(lambda j: pi[j] * multivariate_normal.
(X[:, i], mu[:, j], np.linalg.inv(lamda[j])), range(k))))
        LL.append(LL_t)

    return LL, c
```

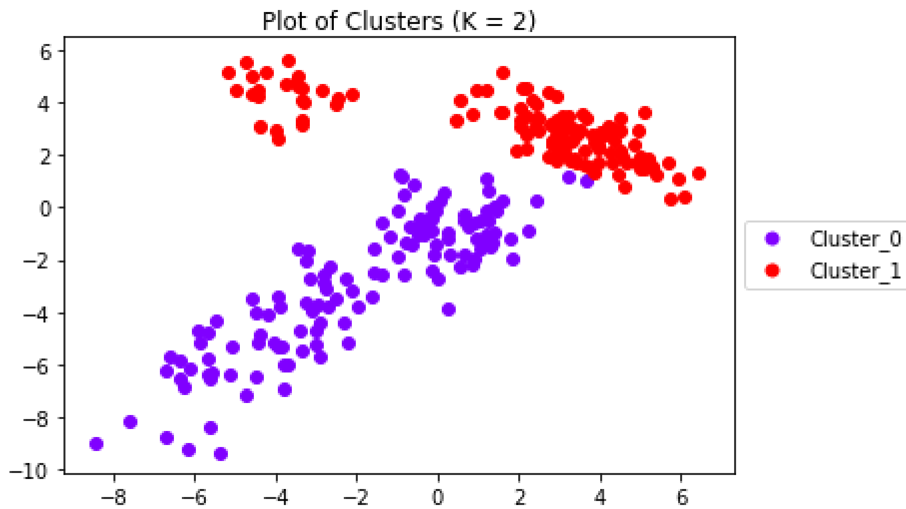
In [4]:
```
def plot_clusters(X, c, k):
    cluster = {}
    for i in range(k):
        cluster[i] = [[], []]
    for i in range(250):
        assignment = np.argmax(c[:, i])
        cluster[assignment][0].append(X[:, i][0])
        cluster[assignment][1].append(X[:, i][1])
    color = iter(plt.cm.rainbow(np.linspace(0,1,k)))
    for i in range(k):
        plt.scatter(cluster[i][0], cluster[i][1], label='Cluster_' + str(
), c=next(color), marker='o')
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.title('Plot of Clusters (K = ' + str(k) + ')')
```

In [5]:
```
L_2, c_2 = EM_GMM(X, 2)
plt.plot(range(100), L_2)
plt.xlabel('Iterations')
plt.ylabel('EM Objective')
plt.title('EM Objective Function against Iterations (K = 2)')
```

Out[5]: Text(0.5,1,u'EM Objective Function against Iterations (K = 2)')

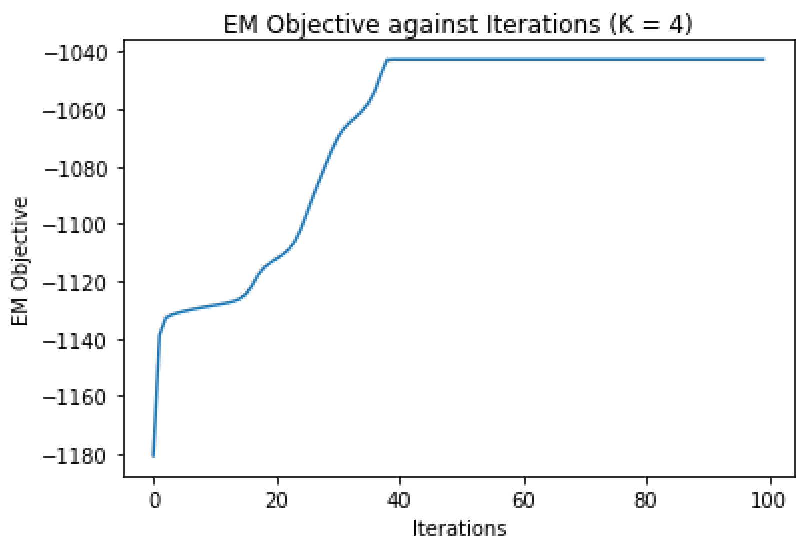


In [6]:
```
plot_clusters(X, c_2, 2)
```

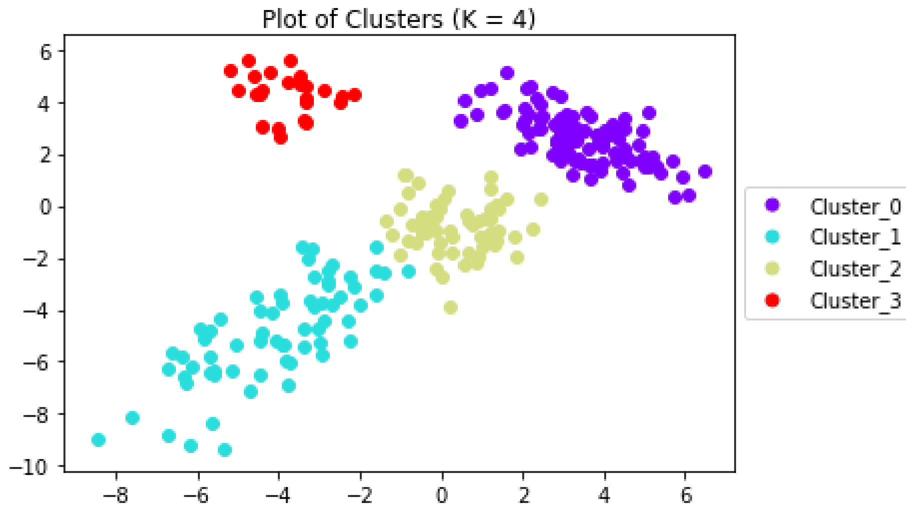


In [7]:
```
L_4, c_4 = EM_GMM(X, 4)
plt.plot(range(100), L_4)
plt.xlabel('Iterations')
plt.ylabel('EM Objective')
plt.title('EM Objective against Iterations (K = 4)')
```

Out[7]:
```
Text(0.5,1,u'EM Objective against Iterations (K = 4)')
```

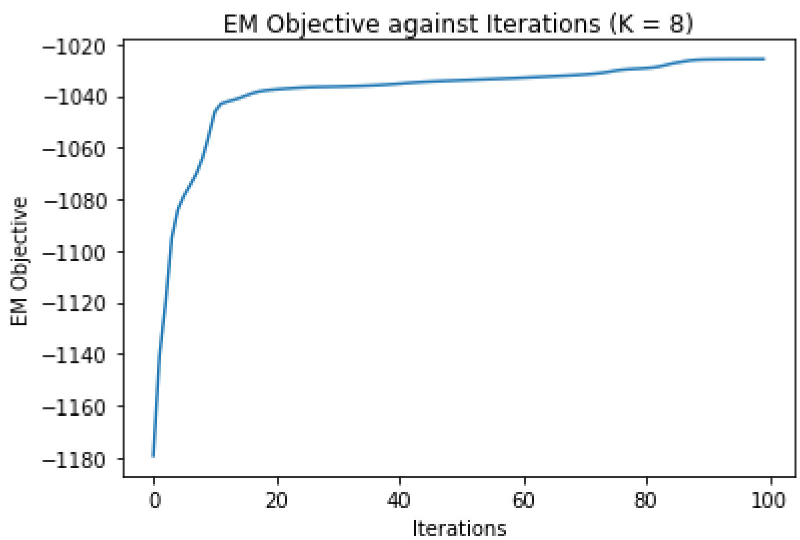


In [8]:
```
plot_clusters(X, c_4, 4)
```

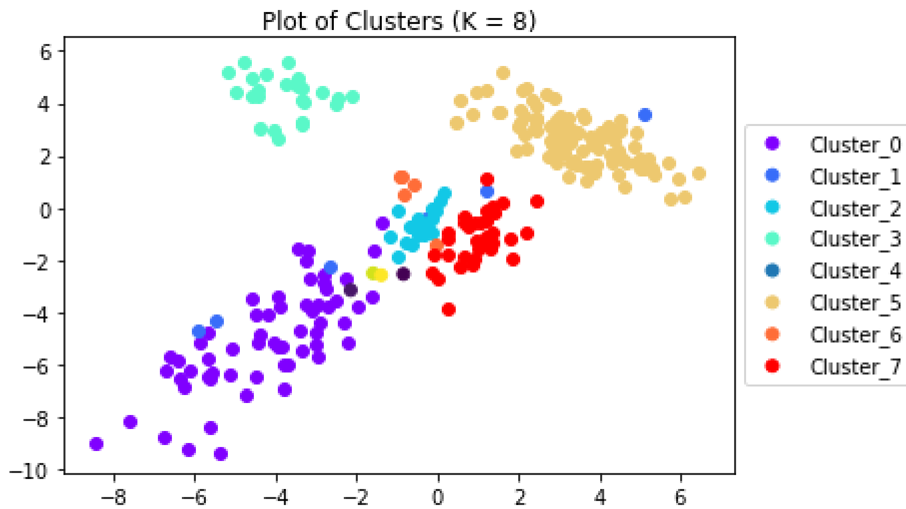


In [9]:
```
L_8, c_8 = EM_GMM(X, 8)
plt.plot(range(100), L_8)
plt.xlabel('Iterations')
plt.ylabel('EM Objective')
plt.title('EM Objective against Iterations (K = 8)')
```

Out[9]:
```
Text(0.5,1,u'EM Objective against Iterations (K = 8)')
```

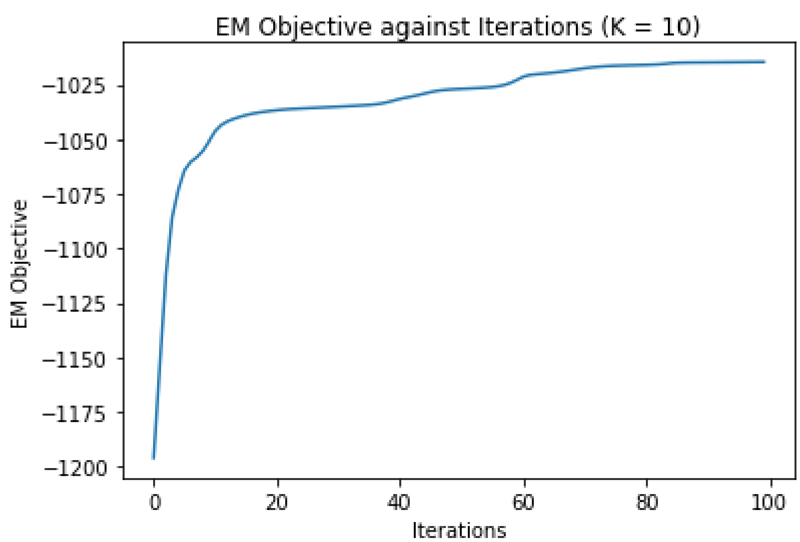


In [10]:
```
plot_clusters(X, c_8, 8)
```

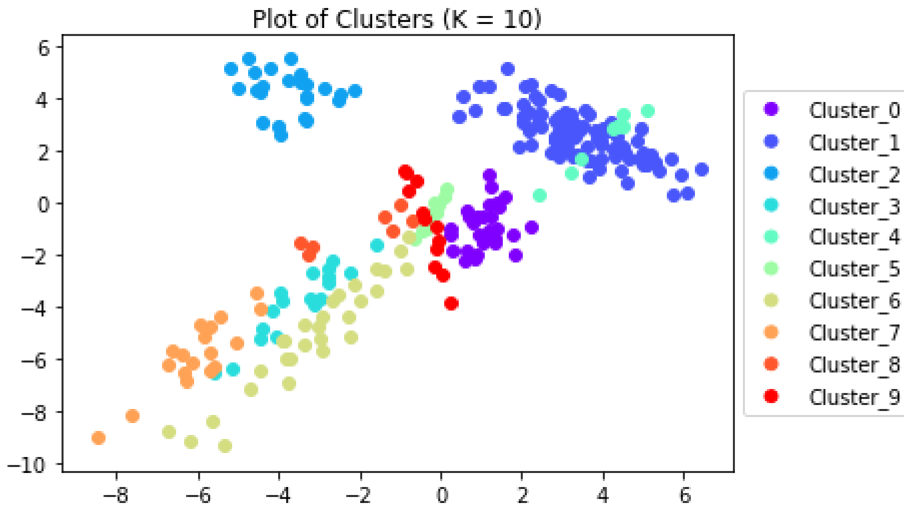


In [11]:
```
L_10, c_10 = EM_GMM(X, 10)
plt.plot(range(100), L_10)
plt.xlabel('Iterations')
plt.ylabel('EM Objective')
plt.title('EM Objective against Iterations (K = 10)')
```

Out[11]:
```
Text(0.5,1,u'EM Objective against Iterations (K = 10)')
```



In [12]:
```
plot_clusters(X, c_10, 10)
```



In [ ]:
```

```

Problem 2

b

The variational objective function peaks at $K = 4$. The phenomenon might indicate that log likelihood might be a possible criteria to use for model selection in VI-GMM.

b

The number of clusters increases with K till $K = 4$ and stays at 4 for higher values of K.

In [1]:
```
# Use gammaln for stability
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
from scipy.special import digamma, gammaln, multigammaln
from scipy.stats import wishart
from sklearn.covariance import empirical_covariance
```

In [2]:
```
# Load data
data = loadmat('hw4_data_mat/data.mat')
X = data['X']
num = X.shape[1]
np.random.seed(3950)
```

In [3]:
```
# Set k
# k = 2
```

In [4]:
```
# Set prior parameters
d = 2
c_0 = 10
m_0 = 0
a_0 = d

# Calculate empirical covariance
A = empirical_covariance(X.T)
B_0 = 2.0/10 * A
```

In [5]:
```
# t1 of q(c)
def t1(a_j, B_j, k):
    t1_1 = sum(map(lambda k: digamma(0.5 * (1 - k + a_j)), range(1,
d+1)))
    t1_2 = np.linalg.slogdet(B_j)
    return t1_1 - t1_2[0] * t1_2[1]
```

In [6]:
```
# t2 of q(c)
def t2(X, idx, m_j, a_j, B_j):
    return np.dot(np.dot((X[:, idx] - m_j).T, a_j * np.linalg.inv(B_j)),
 (X[:, idx] - m_j))
```

In [7]:
```
# t3 of q(c)
def t3(a_j, B_j, Sigma_j):
    return np.trace(np.dot(a_j * np.linalg.inv(B_j), Sigma_j))
```

In [8]:
```
# t4 of q(c)
def t4(alpha, i):
    return digamma(alpha[i]) - digamma(sum(alpha))
```

In [9]:
```
def update_q_c(X, alpha, m, Sigma, a, B, k):
    q_c = np.empty((k ,num))
    for i in range(k):
        # Calculate t1 and t3 first as reusable
        q_t1 = t1(a[i], B[i], k)
        q_t3 = t3(a[i], B[i], Sigma[i])
        q_t4 = t4(alpha, i)
        q_c[i, :] = map(lambda j: np.exp(0.5 * (q_t1 - t2(X, j, m[i], a[i
 B[i]) - q_t3) + q_t4), range(num))
    for j in range(num):
        q_c[:, j] = q_c[:, j] / float(np.sum(q_c[:, j]))
    return q_c
```

In [10]:
```
def cal_n(q_c):
    # Returns a k-length vector
    return np.sum(q_c, axis=1)
```

In [11]:
```
def update_q_pi(alpha_0, n):
    return alpha_0 + n
```

In [12]:
```
def update_q_mu(X, c_0, n, a, B, q_c, k):
    Sigma = map(lambda j: np.linalg.inv(1.0/c_0 * np.identity(d) + n[j]*a
j]*np.linalg.inv(B[j])), range(k))
    m = map(lambda j: np.dot(Sigma[j], a[j]*np.dot(np.linalg.inv(B[j]), n
dot(X, q_c[j, :].T))), range(k))
    return Sigma, m
```

In [13]:
```
def update_q_lambda(X, a_0, n, B, B_0, m, Sigma, q_c, k):
    a = a_0 + n
    x_minus_m = []
    for i in range(k):
        x_minus_m.append(X.T - m[i])
    for i in range(k):
        B_2 = sum(map(lambda j: q_c[i, j] * (np.dot(x_minus_m[i][j].resh
ape((d, 1)), x_minus_m[i][j].reshape((1, d))) + Sigma[i]), range(num)))
        B[i] = B_0 + B_2
    return a, B
```

In [14]:
```
def cal_E_ln_p_x_i_mu_j_lambda_j(X, E_ln_lambda_j, E_lambda_j, m_j, Sigm
a_j):
    x_minus_m = X.T - m_j
    E_x_m_T_lambda_x_m = map(lambda i: -np.dot(np.dot(x_minus_m[i].resha
pe((1, 2)), E_lambda_j), x_minus_m[i].reshape((2, 1))), range(num))
    E_x_m_T_lambda_x_m -= np.trace(np.dot(E_lambda_j, Sigma_j))
    return np.array(0.5 * E_x_m_T_lambda_x_m + 0.5 * E_ln_lambda_j).resh
ape((250))
```

In [15]:
```
def cal_E_ln_pi(alpha, k):
    return map(lambda i: digamma(alpha[i]) - digamma(sum(alpha)), range(k
```

In [16]:
```
def cal_L1(X, alpha, E_ln_lambda, E_lambda, m, Sigma, c, k):
    t2 = np.empty((k, num))
    for i in range(k):
        t2[i, :] = cal_E_ln_p_x_i_mu_j_lambda_j(X, E_ln_lambda[i], E_lam
bda[i], m[i], Sigma[i])
    t3 = np.array(cal_E_ln_pi(alpha, k)).reshape((1, k))
    t23 = t2 + t3.T
    L1 = 0
    for j in range(num):
        L1 += sum(map(lambda i: c[i, j] * t23[i, j], range(k)))
    return L1
```

In [17]:
```
def cal_E_ln_lambda(a, B, k):
    E_ln_lambda = []
    for i in range(k):
        t1 = np.linalg.slogdet(B[i])
        t2 = sum(map(lambda j: digamma(0.5 * (a[i] + 1 - j)), range(1, d+
)))
        E_ln_lambda.append(-t1[0]*t1[1] + t2)
    return E_ln_lambda
```

In [18]:
```
def cal_E_lambda(a, B):
    return map(lambda a_B: a_B[0] * np.linalg.inv(a_B[1]), zip(a, B))
```

In [19]:
```
def cal_E_ln_p_mu(m):
    return map(lambda mu: -0.5*(np.dot(np.dot(mu.reshape((1, 2)), 1/floa
t(c_0) * np.identity(d)), mu.reshape((2, 1)))), m)
```

In [20]:
```
def cal_E_ln_p_lambda(E_ln_lambda, E_lambda, B_0):
    return map(lambda lbda: -0.5*(lbda[0] + np.trace(np.dot(B_0,
lbda[1]))), zip(E_ln_lambda, E_lambda))
```

In [21]:
```
def cal_L2(E_ln_p_mu, E_ln_p_lambda):
    return sum(E_ln_p_mu + E_ln_p_lambda)
```

In [22]:
```
def cal_L3(c, k):
    L3 = 0
    for j in range(num):
        L3 += sum(map(lambda i: c[i, j] * np.log(c[i, j]), range(k)))
    return L3
```

In [23]:
```
def cal_L4(alpha, k):
    sum_alpha = sum(alpha)
    t1 = sum(map(lambda i: gammaln(alpha[i]), range(k)))
    t2 = gammaln(sum_alpha)
    t3 = (k - sum_alpha) * digamma(sum_alpha)
    t4 = sum(map(lambda i: (alpha[i]-1) * digamma(alpha[i]), range(k)))
    return t1 - t2 - t3 - t4
```

In [24]:
```
def cal_L5(Sigma, k):
    return sum(map(lambda i: 0.5 * np.log(np.linalg.det(2 * np.pi * np.e
xp(1) * Sigma[i])), range(k)))
```

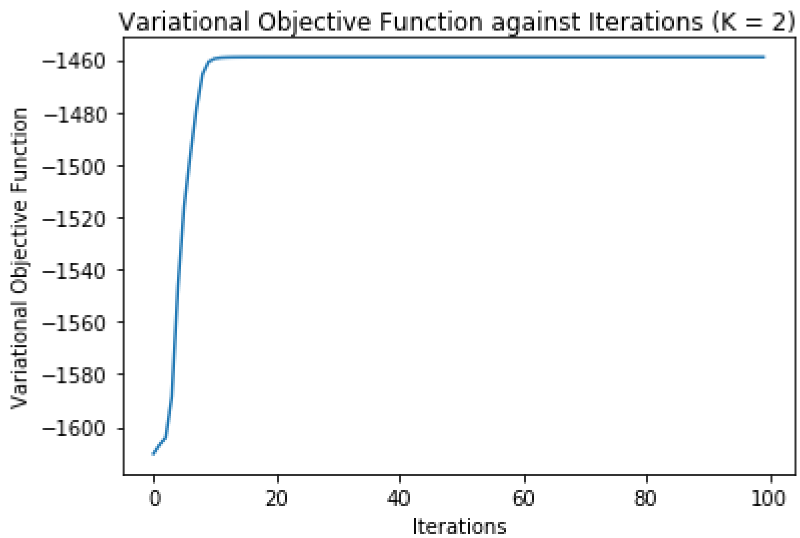
In [25]:
```
def cal_L6(a, B, E_ln_lambda, E_lambda, k):
    return sum(map(lambda i: -0.5 * a[i] * np.log(np.linalg.det(B[i])) +
 0.5 * a[i] * d * np.log(2) + multigammaln(a[i]/2, d) - 0.5 * (a[i] - d
- 1) *  E_ln_lambda[i] - np.trace(np.dot(B[i], E_lambda[i])), range(k)))
```

In [26]:
```
def VI(X, k):
    # Initialise variables
    alpha_0 = np.ones(k)
    alpha = alpha_0
    m = np.random.uniform(-1, 1, k)
    Sigma = [10*np.identity(d)] * k
    a = [a_0] * k
    B = [B_0] * k
    L = []

    # Run VI
    for i in range(100):
        # Update hyperparameters
        c = update_q_c(X, alpha, m, Sigma, a, B, k)
        n = cal_n(c)
        alpha = update_q_pi(alpha_0, n)
        Sigma, m = update_q_mu(X, c_0, n, a, B, c, k)
        a, B = update_q_lambda(X, a_0, n, B, B_0, m, Sigma, c, k)

        # Calculate likelihood
        E_ln_lambda = cal_E_ln_lambda(a, B, k)
        E_lambda = cal_E_lambda(a, B)
        L1 = cal_L1(X, alpha, E_ln_lambda, E_lambda, m, Sigma, c, k)
        E_ln_p_mu = cal_E_ln_p_mu(m)
        E_ln_p_lambda = cal_E_ln_p_lambda(E_ln_lambda, E_lambda, B_0)
        L2 = cal_L2(E_ln_p_mu, E_ln_p_lambda)
        L3 = cal_L3(c, k)
        L4 = cal_L4(alpha, k)
        L5 = cal_L5(Sigma, k)
        L6 = cal_L6(a, B, E_ln_lambda, E_lambda, k)
        LL = L1 + L2 - L3 + L4 + L5 + L6
        L.append(LL.flatten()[0])

    return L, c
```

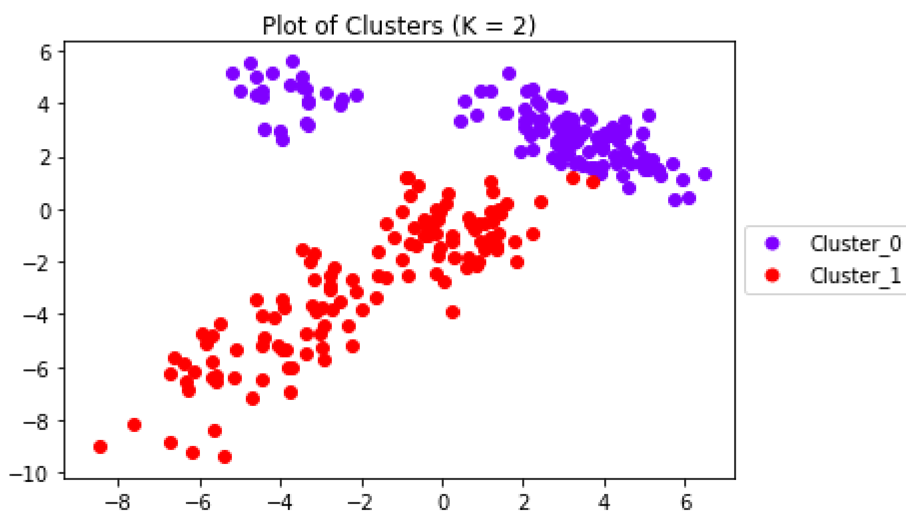
In [27]:
```
def plot_clusters(X, c, k):
    cluster = {}
    for i in range(k):
        cluster[i] = [[], []]
    for i in range(250):
        assignment = np.argmax(c[:, i])
        cluster[assignment][0].append(X[:, i][0])
        cluster[assignment][1].append(X[:, i][1])
    color = iter(plt.cm.rainbow(np.linspace(0,1,k)))
    for i in range(k):
        plt.scatter(cluster[i][0], cluster[i][1], label='Cluster_' + str(
), c=next(color), marker='o')
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.title('Plot of Clusters (K = ' + str(k) + ')')
```

In [28]:
```
L_2, c_2 = VI(X, 2)
plt.plot(range(100), L_2)
plt.xlabel('Iterations')
plt.ylabel('Variational Objective Function')
plt.title('Variational Objective Function against Iterations (K = 2)')
```

Out[28]:
```
Text(0.5,1,u'Variational Objective Function against Iterations (K =
 2)')
```

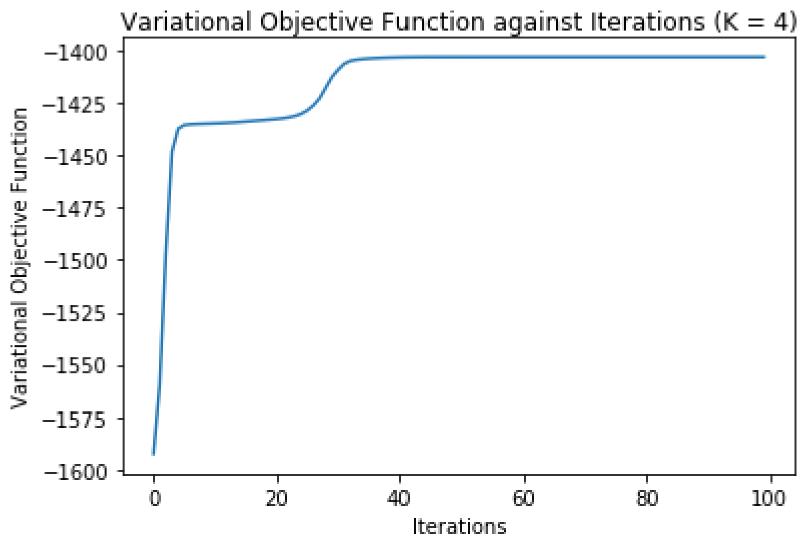


In [29]:
```
plot_clusters(X, c_2, 2)
```

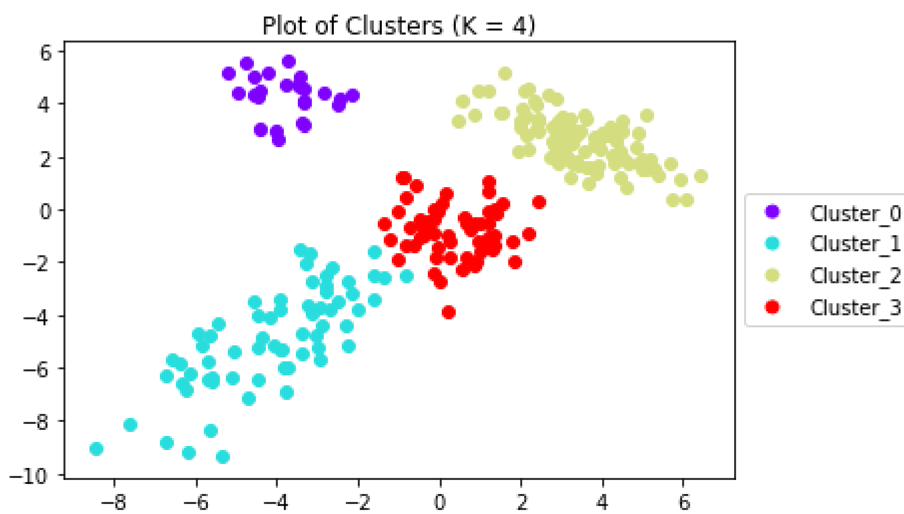


In [30]:
```
L_4, c_4 = VI(X, 4)
plt.plot(range(100), L_4)
plt.xlabel('Iterations')
plt.ylabel('Variational Objective Function')
plt.title('Variational Objective Function against Iterations (K = 4)')
```

Out[30]:
```
Text(0.5,1,u'Variational Objective Function against Iterations (K =
 4)')
```

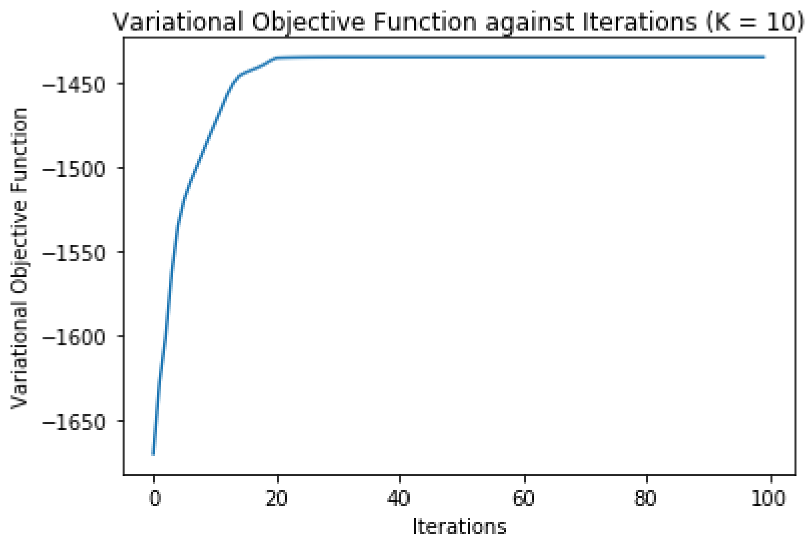


In [31]:
```
plot_clusters(X, c_4, 4)
```

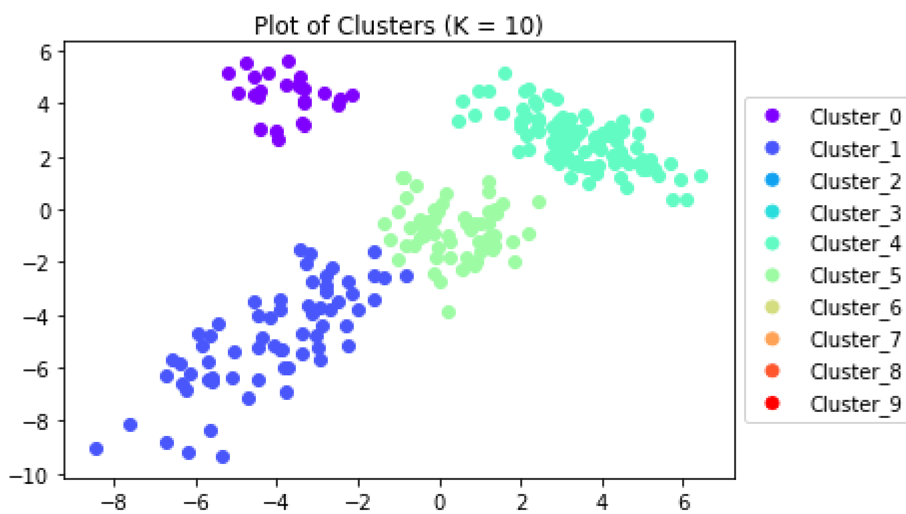


In [32]:
```
L_10, c_10 = VI(X, 10)
plt.plot(range(100), L_10)
plt.xlabel('Iterations')
plt.ylabel('Variational Objective Function')
plt.title('Variational Objective Function against Iterations (K = 10)')
```

Out[32]:
```
Text(0.5,1,u'Variational Objective Function against Iterations (K = 1
0)')
```



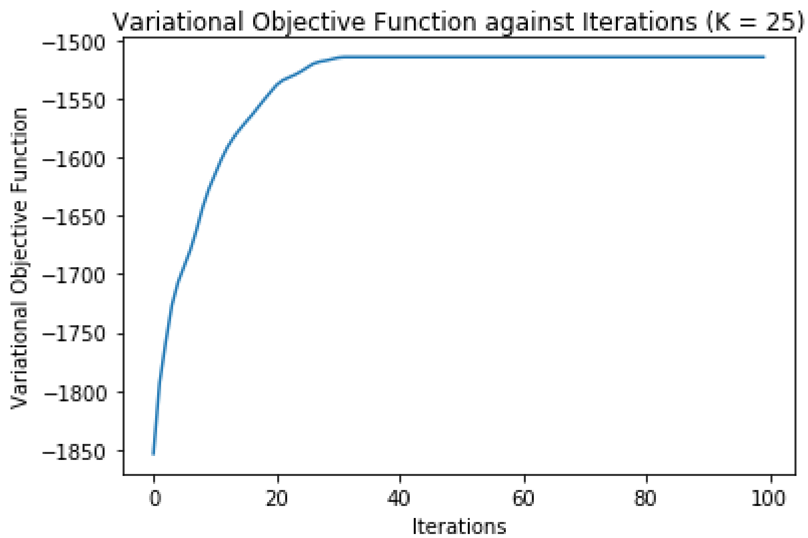
In [33]:
```
plot_clusters(X, c_10, 10)
```



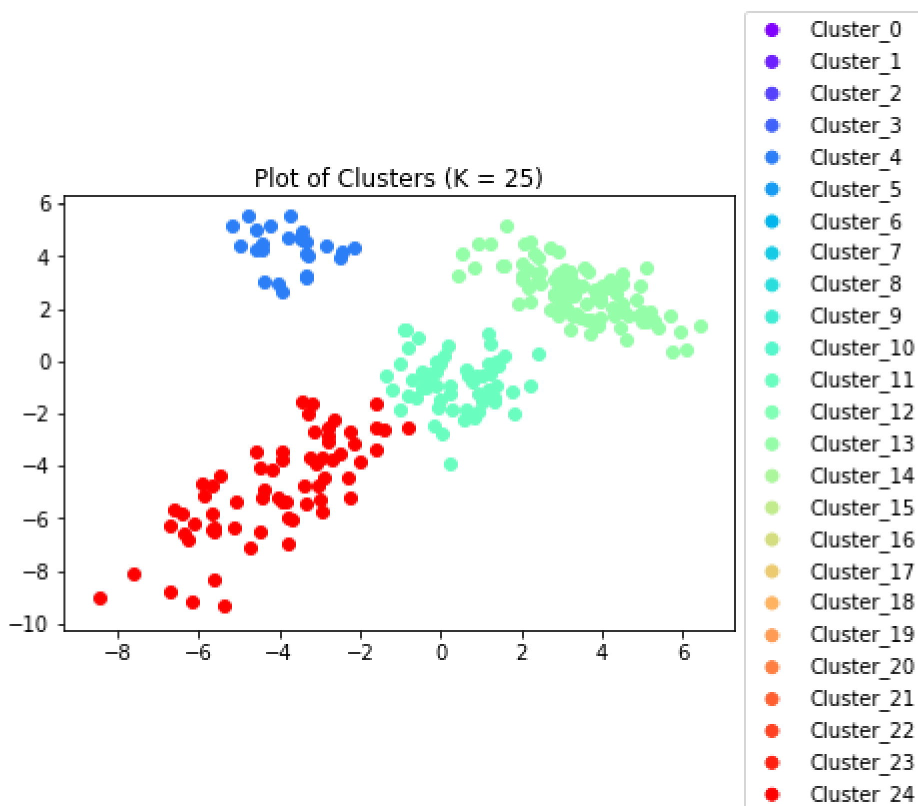
Loading [Contrib]/a11y/accessibility-menu.js

In [34]:
```
L_25, c_25 = VI(X, 25)
plt.plot(range(100), L_25)
plt.xlabel('Iterations')
plt.ylabel('Variational Objective Function')
plt.title('Variational Objective Function against Iterations (K = 25)')
```

Out[34]:
```
Text(0.5,1,u'Variational Objective Function against Iterations (K = 2
5)')
```



In [35]:
```
plot_clusters(X, c_25, 25)
```



In [ ]:

Loading [Contrib]/a11y/accessibility-menu.js

Problem 3

In [1]:
```
# Use gammaln for stability
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
from scipy.special import digamma, gammaln, multigammaln
from scipy.stats import multivariate_normal, wishart
from sklearn.covariance import empirical_covariance
```

In [2]:
```
# Load data
data = loadmat('hw4_data_mat/data.mat')
X = data['X']
d = X.shape[0]
num = X.shape[1]
np.random.seed(3950)
```

In [3]:
```
# Set prior parameters
c_0 = 0.1
a_0 = d
alpha_0 = 1

# Calculate empirical mean
sum_X = np.sum(X, axis=1)
m_0 = sum_X / float(num)

# Calculate empirical covariance
A = empirical_covariance(X.T)
B_0 = c_0 * d * A
```

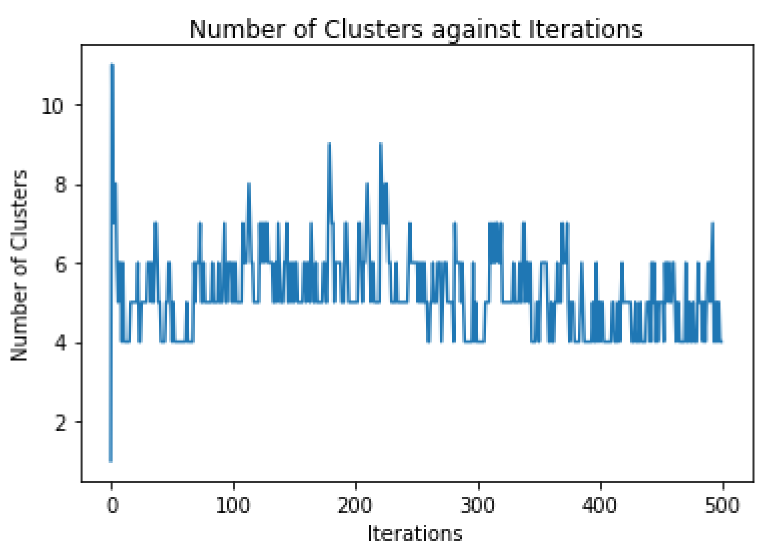
In [ ]:
```
def p_x(sample):
    phi_n_t1 = (c_0 / (np.pi * (1 + c_0)))**(0.5 * d)
    x_minus_m = (sample - m_0).reshape((d,1))
    phi_n_t2 = (np.linalg.det(B_0 + c_0/(c_0+1) * np.dot(x_minus_m.T, x_
minus_m)))**(-0.5*(a+1))/np.linalg.det(B_0)**(-0.5*a)
    phi_n_t3 = np.exp(multigammaln(0.5*(a+1), d)- multigammaln(0.5*a,
d))
    return alpha_0/float(alpha_0 + num - 1) * phi_n_t1 * phi_n_t2 * phi_
n_t3
```

```python
# Initialisation
c = [0] * num
n = {0: range(num)}
theta = {}
lamda = wishart.rvs(a_0, np.linalg.inv(B_0))
covariance = np.linalg.inv(lamda)
theta[0] = [np.random.multivariate_normal(m_0, 1/float(c_0) *
covariance), covariance]

a = a_0
B = B_0
m = m_0

num_clusters = []
largest_six = []

p_x_all = map(lambda i: p_x(X[:, i]), range(num))

for iter in range(500):
    counts_clusters = [len(n[i]) for i in n]
    counts_clusters.sort(reverse=True)
    if len(n.keys()) < 6:
        largest_six.append(counts_clusters)
    else:
        largest_six.append(counts_clusters[:6])
    num_clusters.append(len(n.keys()))

    # 1
    for sample in range(num):
        # a) and b)
        phi = []
        for cluster in n:
            if c[sample] == cluster:
                n[cluster].remove(sample)
            phi_j = multivariate_normal.pdf(X[:, sample], theta[cluster]
[0], theta[cluster][1]) * len(n[cluster]) / float(alpha_0 + num - 1)
            phi.append(phi_j)
        phi.append(p_x_all[sample])

        # c)
        phi = np.array(phi) / sum(phi)
        idx_n = len(phi)
        c[sample] = int(np.random.choice(idx_n, 1, p = phi))
        # Add point to new cluster
        try:
            n[c[sample]].append(sample)
        except KeyError:
            n[c[sample]] = [sample]

        # d)
        if c[sample] == idx_n - 1:
            c_j = 1 + c_0
            m_j = c_0/(c_j) * m_0 + 1/(c_j) * X[:, sample]
            a_j = a_0 + 1
            x_bar_minus_m = np.array(X[:, sample] - m_0).reshape((d,1))
            B_j = B_0 + c_0/(c_j) * np.dot(x_bar_minus_m, x_bar_minus_m.T
```

```
            lamda_j = wishart.rvs(a_j, np.linalg.inv(B_j))
            covariance_j = np.linalg.inv(lamda_j)
            theta[idx_n - 1] = [np.random.multivariate_normal(m_j, 1/flo
at(c_j) * covariance_j), covariance_j]

        # Housekeeping
        # Remove all clusters with 0 entries
        n = { k : v for k,v in n.iteritems() if len(v) > 0}
        # Theta
        exist_c = n.keys()
        theta_n = {}
        for i in range(len(exist_c)):
            theta_n[i] = theta[exist_c[i]]
        theta = theta_n
        # Reindex clusters
        c_n = []
        n = {}
        for i in range(num):
            for j in range(len(exist_c)):
                if c[i] == exist_c[j]:
                    c_n.append(j)
                    try:
                        n[j].append(i)
                    except KeyError:
                        n[j] = [i]
        c = c_n

    # 2
    for cluster in n:
        s_j = len(n[cluster])
        c_j = s_j + c_0
        sum_j = np.sum(X[:, n[cluster]], axis=1)
        m_j = c_0/(c_j) * m_0 + 1/(c_j) * sum_j
        a_j = a_0 + s_j
        x_bar = 1/float(s_j) * sum_j
        x_minus_m_bar = X[:, n[cluster]].T - x_bar.T
        x_bar_minus_m = np.array(x_bar - m_0).reshape((d,1))
        B_j = B_0 + np.dot(x_minus_m_bar.T, x_minus_m_bar) + s_j * c_0/(c
 * np.dot(x_bar_minus_m, x_bar_minus_m.T)
        lamda_j = wishart.rvs(a_j, np.linalg.inv(B_j))
        covariance_j = np.linalg.inv(lamda_j)
        theta[cluster] = [np.random.multivariate_normal(m_j, 1/(c_j) * c
ovariance_j), covariance_j]

    # print 'Iteration ' + str(iter) + ' Done!'
```

In [ ]:
```
iterations = range(500)
plt.plot(iterations, num_clusters)
plt.xlabel('Iterations')
plt.ylabel('Number of Clusters')
plt.title('Number of Clusters against Iterations')
```

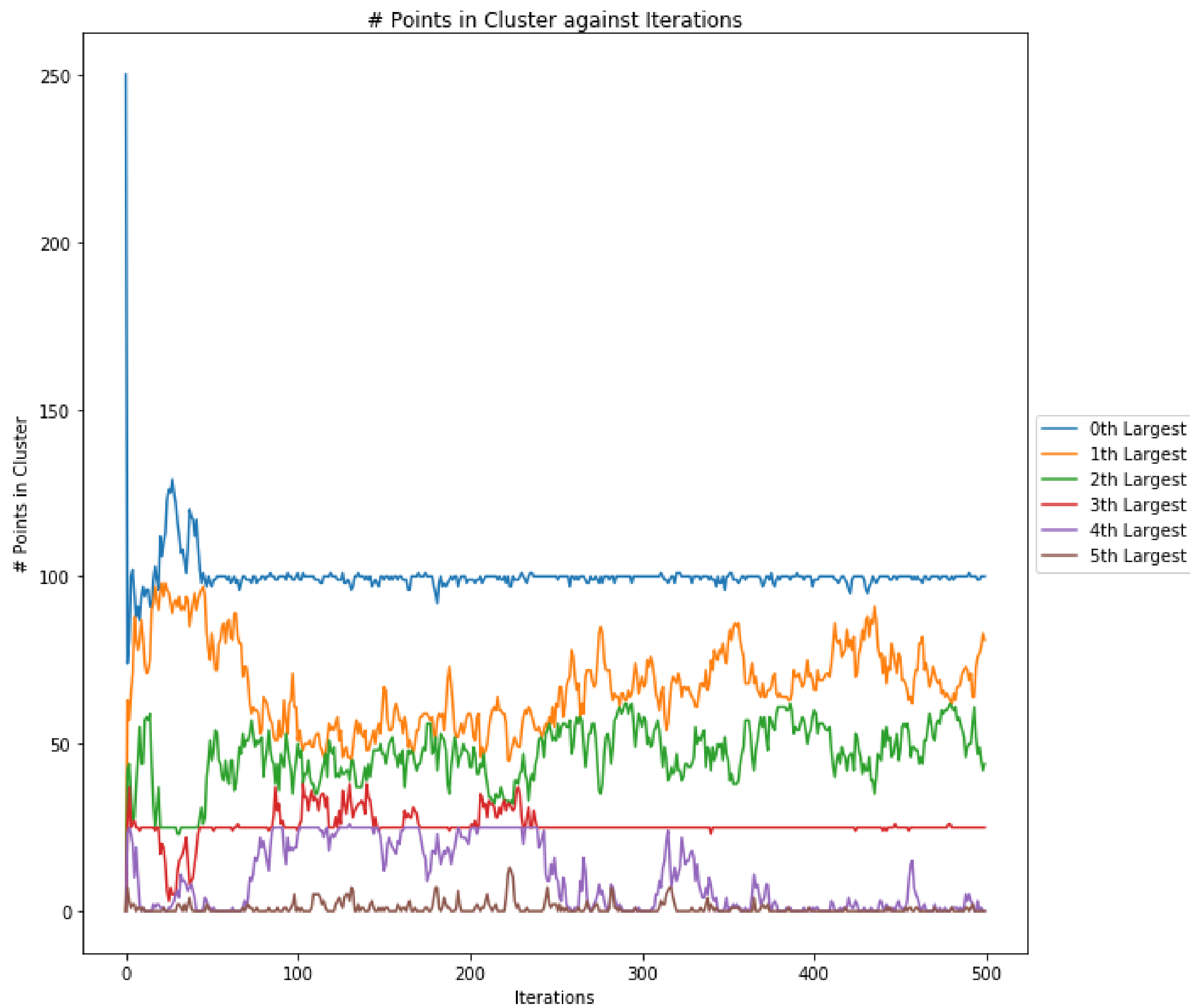
Out[ ]: Text(0.5,1,u'Number of Clusters against Iterations')



In [ ]:
```
largest_six_split = {0:[], 1:[], 2:[], 3:[], 4:[], 5:[]}
for i in largest_six:
    for j in range(len(i)):
        largest_six_split[j].append(i[j])
    if len(i) < 6:
        for k in range(len(i), 6):
            largest_six_split[k].append(0)
```

In [ ]:
```
plt.figure(figsize=(10,10))
for i in largest_six_split:
    plt.plot(iterations, largest_six_split[i], label=str(i)+'th
Largest')
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xlabel('Iterations')
plt.ylabel('# Points in Cluster')
plt.title('# Points in Cluster against Iterations')
```

Out[ ]:
```
Text(0.5,1,u'# Points in Cluster against Iterations')
```



In [ ]: