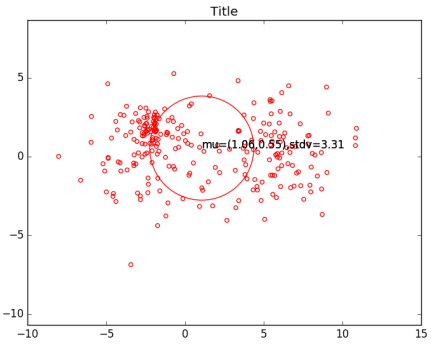
6.036 Project 3

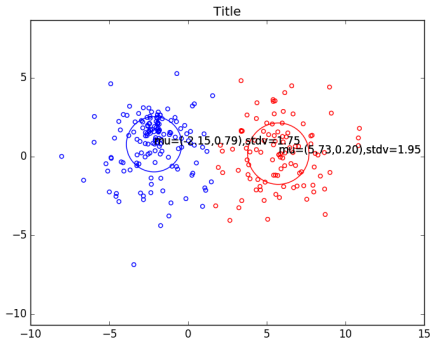
Kevin

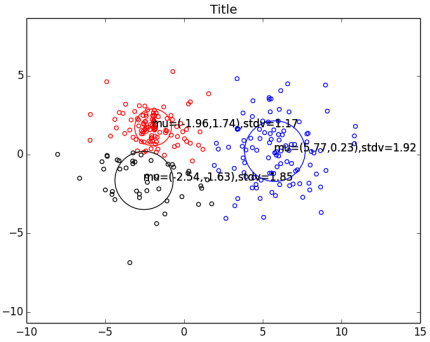
Due MAY 3RD, 2016

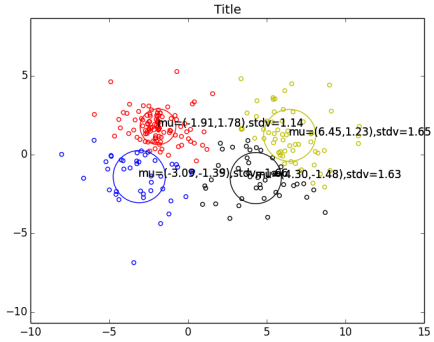
I Warmup

a. Here are the plots that I got:

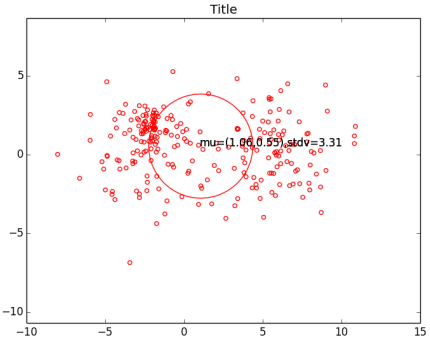


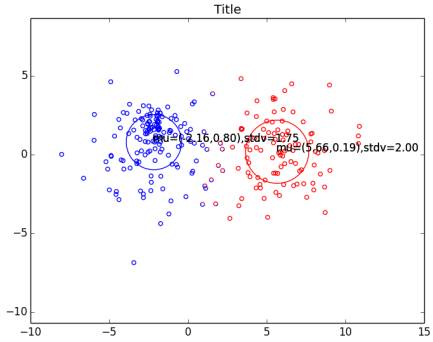


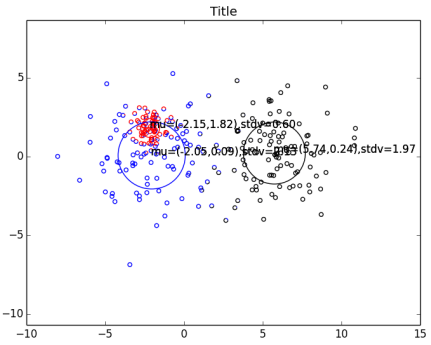


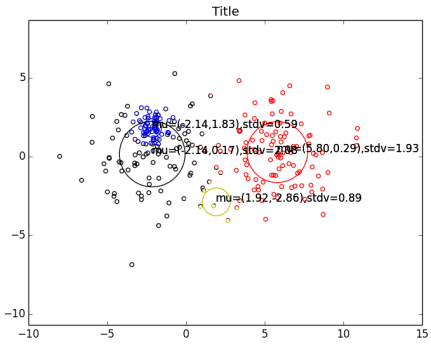


d. These are the plots:









They differ because the algorithm we use to generate these plots are different. They work for the similar purpose, due to different nature, it is inevitable that the plots are different.

e. The best cluster score was achieved when K=3. The score was -1169.26072362 after running a few times. Looking at the graphs, it shows that three does divide in a pretty accurate way then the other three. Two was close, but three is a bit more exact.

II Mixture models

a. We know that some values of X are unknown and left to zero. Thus, when calculating the posterier, we shouldn't be having the zeroes included. We should only have the known values. This is essentially what this formula does. It reshapes Mu to have the values of X that are known. Through these values, we are able to find the values of the probability that a user is in a cluster.

f. So, we know we have the probability that a user is in a certain cluster through the posterier matrix. Thus, getting a certain row vector gives us the probability a unique user is in each of the clusters.

We then move on to the Mu matrix. This gives us the mean value of each cluster on each movie. Thus, getting a column vector gives us each mean rating of a movie by certain clusters.

Thus, if we multiply these two vectors we get the expected rating of a user for each of the unknown rating values.

g All the times I ran, I had a final LL value of around 1 million 3 hundred thousand. The exact value was -1381601.14519.

f. The rmse value I got was: 0.483859051544.

Here is the code for my project 3:

```
import random as ra
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as pat
import math as math
from numpy import linalg as LA
from scipy.misc import logsumexp

#-------------------------------------------------------------------------
# Utility Functions - There is no need to edit code in this section.
#-------------------------------------------------------------------------

# Reads a data matrix from file.
# Output: X: data matrix.
def readData(file):
    X = []
```

```
    with open(file,"r") as f:
        for line in f:
            X.append(map(float,line.split(" ")))
    return np.array(X)


# plot 2D toy data
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
#        Label: n*K matrix, each row corresponds to the soft counts for all mixtures for a
#        title: a string represents the title for the plot
def plot2D(X,K,Mu,P,Var,Label,title):
    r=0.25
    color=["r","b","k","y","m","c"]
    n,d = np.shape(X)
    per= Label/(1.0*np.tile(np.reshape(np.sum(Label,axis=1),(n,1)),(1,K)))
    fig=plt.figure()
    plt.title(title)
    ax=plt.gcf().gca()
    ax.set_xlim((-20,20))
    ax.set_ylim((-20,20))
    for i in xrange(len(X)):
        angle=0
        for j in xrange(K):
            cir=pat.Arc((X[i,0],X[i,1]),r,r,0,angle,angle+per[i,j]*360,edgecolor=color[j])
            ax.add_patch(cir)
            angle+=per[i,j]*360
    for j in xrange(K):
        sigma = np.sqrt(Var[j])
        circle=plt.Circle((Mu[j,0],Mu[j,1]),sigma,color=color[j],fill=False)
        ax.add_artist(circle)
        text=plt.text(Mu[j,0],Mu[j,1],"mu=("+str("%.2f" %Mu[j,0])+","+str("%.2f" %Mu[j,1])+
        ax.add_artist(text)
    plt.axis('equal')
    plt.show()

#-----------------------------------------------------------------------------------------



#-----------------------------------------------------------------------------------------
# K-means methods - There is no need to edit code in this section.
#-----------------------------------------------------------------------------------------

# initialization for k means model for toy data
# input: X: n*d data matrix;
#        K: number of mixtures;
#        fixedmeans: is an optional variable which is
#        used to control whether Mu is generated from a deterministic way
#        or randomized way
# output: Mu: K*d matrix, each row corresponds to a mixture mean;
#         P: K*1 matrix, each entry corresponds to the weight for a mixture;
#         Var: K*1 matrix, each entry corresponds to the variance for a mixture;
def init(X,K,fixedmeans=False):
```

```
    n, d = np.shape(X)
    P=np.ones((K,1))/float(K)

    if (fixedmeans):
        assert(d==2 and K==3)
        Mu = np.array([[4.33,-2.106],[3.75,2.651],[-1.765,2.648]])
    else:
        # select K random points as initial means
        rnd = np.random.rand(n,1)
        ind = sorted(range(n),key = lambda i: rnd[i])
        Mu = np.zeros((K,d))
        for i in range(K):
            Mu[i,:] = np.copy(X[ind[i],:])

    Var=np.mean( (X-np.tile(np.mean(X,axis=0),(n,1)))**2 )*np.ones((K,1))
    return (Mu,P,Var)


# K Means method
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
# output: Mu: K*d matrix, each row corresponds to a mixture mean;
#         P: K*1 matrix, each entry corresponds to the weight for a mixture;
#         Var: K*1 matrix, each entry corresponds to the variance for a mixture;
#         post: n*K matrix, each row corresponds to the soft counts for all mixtures for an
def kMeans(X, K, Mu, P, Var):
    prevCost=-1.0; curCost=0.0
    n=len(X)
    d=len(X[0])
    while abs(prevCost-curCost)>1e-4:
        post=np.zeros((n,K))
        prevCost=curCost
        #E step
        for i in xrange(n):
            post[i,np.argmin(np.sum(np.square(np.tile(X[i,],(K,1))-Mu),axis=1))]=1
        #M step
        n_hat=np.sum(post,axis=0)
        P=n_hat/float(n)
        curCost = 0
        for i in xrange(K):
            Mu[i,:]= np.dot(post[:,i],X)/float(n_hat[i])
            # summed squared distance of points in the cluster from the mean
            sse = np.dot(post[:,i],np.sum((X-np.tile(Mu[i,:],(n,1)))**2,axis=1))
            curCost += sse
            Var[i]=sse/float(d*n_hat[i])
        print curCost
    # return a mixture model retrofitted from the K-means solution
    return (Mu,P,Var,post)
#-------------------------------------------------------------------------------



#-------------------------------------------------------------------------------
# PART 1 - EM algorithm for a Gaussian mixture model
```

```
#-----------------------------------------------------------------------------

# E step of EM algorithm
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
# output:post: n*K matrix, each row corresponds to the soft counts for all mixtures for an
#        LL: a Loglikelihood value
def Gaussian(P, var, point, d):
    return P * 1.0/(2.0 * math.pi * var)**(d/2.0) * math.e**(-1.0/(2 * var) * (LA.norm(poi

def Estep(X,K,Mu,P,Var):
    n,d = np.shape(X) # n data points of dimension d
    post = np.zeros((n,K)) # posterior probabilities to compute
    LL = 0.0    # the LogLikelihood

    #Write your code here
    for i in xrange(n):
        p = 0
        s=0
        for j in xrange(K):
            var = Var[j]
            point = X[i] - Mu[j]
            prob = Gaussian(P[j], var, point, d)
            post[i][j] = prob
            p += prob[0]
            s += prob[0]

        for j in xrange(K):
            post[i][j] *= 1/p

        LL += np.log(s)
    return (post,LL)


# M step of EM algorithm
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
#        post: n*K matrix, each row corresponds to the soft counts for all mixtures for an
# output:Mu: updated Mu, K*d matrix, each row corresponds to a mixture mean;
#        P: updated P, K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: updated Var, K*1 matrix, each entry corresponds to the variance for a mixture
def Mstep(X,K,Mu,P,Var,post):
    n,d = np.shape(X) # n data points of dimension d

    for j in xrange(K):
        new = 0.0
        for i in xrange(n):
            new += post[i][j]
        P[j] = new
    P /= n
```

```
    for j in xrange(K):
        new = 0.0
        sum = 0.0
        for i in xrange(n):
            new += post[i][j] * X[i]
            sum += post[i][j]
        Mu[j] = new/sum

    for j in xrange(K):
        new = 0.0
        sum = 0.0
        for i in xrange(n):
            sum += post[i][j]
            point = X[i] - Mu[j]
            new += post[i][j] * (LA.norm(point))**2
        Var[j] = 1/(d*sum) * new

    return (Mu,P,Var)


# Mixture of Gaussians
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
# output: Mu: updated Mu, K*d matrix, each row corresponds to a mixture mean;
#         P: updated P, K*1 matrix, each entry corresponds to the weight for a mixture;
#         Var: updated Var, K*1 matrix, each entry corresponds to the variance for a mixtu
#         post: updated post, n*K matrix, each row corresponds to the soft counts for all
#         LL: Numpy array for Loglikelihood values
def mixGauss(X,K,Mu,P,Var):
    n,d = np.shape(X) # n data points of dimension d
    post = np.zeros((n,K)) # posterior probabilities

    #Write your code here
    #Use function Estep and Mstep as two subroutines

    L=[]

    for c in xrange(2):
        post, ll = Estep(X, K, Mu, P, Var)
        L.append(ll)
        Mu, P, Var = Mstep(X, K, Mu, P, Var, post)

    i = 0
    while np.abs(L[i+1] - L[i]) > 10**(-6) * np.abs(L[i+1]):
        post, ll = Estep(X, K, Mu, P, Var)
        if (ll - L[i+1]) > 10**(-6) * np.abs(ll):
            L.append(ll)
            i += 1
        else:
            break
        Mu, P, Var = Mstep(X, K, Mu, P, Var, post)
    LL = np.asarray(L)
    return (Mu,P,Var,post, LL)
```

```
# Bayesian Information Criterion (BIC) for selecting the number of mixture components
# input:  n*d data matrix X, a list of K's to try
# output: the highest scoring choice of K
def BICmix(X,Kset):
    n,d = np.shape(X)

    #Write your code here
    finalscore = float("-inf")
    maxK = float("-inf")
    for K in Kset:
        Mu, P, Var = init(X, K)
        (Mu,P,Var,post,LL) = mixGauss(X,K, Mu, P, Var)
        ll = LL[-1]

        parameters = K * (d+2.0)-1.0
        score = ll - 0.5 * parameters * math.log(n)
        if score > finalscore:
            finalscore = score
            maxK = K
    return maxK

#-------------------------------------------------------------------------------



#-------------------------------------------------------------------------------
# PART 2 - Mixture models for matrix completion
#-------------------------------------------------------------------------------

# RMSE criteria
# input: X: n*d data matrix;
#        Y: n*d data matrix;
# output: RMSE
def rmse(X,Y):
    return np.sqrt(np.mean((X-Y)*(X-Y)))


# E step of EM algorithm with missing data
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
# output:post: n*K matrix, each row corresponds to the soft counts for all mixtures for an
#        LL: a Loglikelihood value
def Cu(X):
    n,d = np.shape(X)
    final = []
    for i in xrange(n):
        Cu = []
        for j in xrange(d):
            if X[i][j] != 0:
                Cu.append(X[i][j])
        final.append(np.array(Cu))
    return np.array(final)
```

```
def Cu_index(X):
    n,d = np.shape(X)
    final = []
    for i in xrange(n):
        Cu = []
        for j in xrange(d):
            if X[i][j] != 0:
                Cu.append(j)
        final.append(Cu)
    return np.array(final)

def get_Mu_Cu(index, Mu):
    return [Mu[i] for i in index]

def compare_matrix(X):
    n,d = np.shape(X)
    final = []
    for i in xrange(n):
        row = []
        for j in xrange(d):
            if X[i][j] != 0:
                row.append(1.0)
            else:
                row.append(0.0)
        final.append(np.array(row))
    return np.array(final)

def function(x, mu, var, p, d):
    point = (x - mu)
    return math.log(p) - d/2.0*(math.log(2.0 * math.pi * var)) - (1.0/(2.0 * var) * (LA.nor

def Estep_part2(X,K,Mu,P,Var):
    n,d = np.shape(X) # n data points of dimension d
    post = np.zeros((n,K)) # posterior probabilities to compute
    LL = 0.0    # the LogLikelihood
    X_Cu = Cu(X)
    index = Cu_index(X)

        #Write your code here
    for i in xrange(n):
        p = []
        length = len(X_Cu[i])

        for j in xrange(K):
            var = Var[j]
            Muc = get_Mu_Cu(index[i], Mu[j])
            prob = function(X_Cu[i], Muc, var, P[j], length)
            p.append(prob)

        for j in xrange(K):
            var = Var[j]
            Muc = get_Mu_Cu(index[i], Mu[j])
            prob = function(X_Cu[i], Muc, var, P[j], length)
            prob = prob - logsumexp(p)
            post[i][j] = prob
```

```
        LL = LL + logsumexp(p)

    return (np.exp(post),LL)


# M step of EM algorithm
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
#        post: n*K matrix, each row corresponds to the soft counts for all mixtures for an
# output:Mu: updated Mu, K*d matrix, each row corresponds to a mixture mean;
#        P: updated P, K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: updated Var, K*1 matrix, each entry corresponds to the variance for a mixture
def Mstep_part2(X,K,Mu,P,Var,post, minVariance=0.25):
    n,d = np.shape(X) # n data points of dimension d

    for j in xrange(K):
        new = 0.0
        for i in xrange(n):
            new += post[i][j]
        P[j] = new
    P /= n

    indices = compare_matrix(X)

    for k in xrange(K):
        for i in xrange(d):
            new = 0.0
            sum = 0.0
            index = indices[:,i]
            column = X[:,i]
            pcolumn = post[:,k]
            for u in xrange(n):
                new += index[u] * column[u] * pcolumn[u]
                sum += index[u] * pcolumn[u]
            if sum >= 1:
                Mu[k][i] = new/sum
            else:
                continue

    XCu = Cu(X)
    index = Cu_index(X)
    for j in xrange(K):
        new = 0.0
        sum = 0.0
        for i in xrange(n):
            length = len(XCu[i])
            sum += length*post[i][j]
            Muc = get_Mu_Cu(index[i], Mu[j])
            point = XCu[i] - Muc
            new += post[i][j] * (LA.norm(point))*(LA.norm(point))

        var = new/sum
        if var <= minVariance:
            Var[j] = minVariance
```

```
        else:
            Var[j] = var

    return (Mu,P,Var)

# mixture of Guassians
# input: X: n*d data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
# output: Mu: updated Mu, K*d matrix, each row corresponds to a mixture mean;
#         P: updated P, K*1 matrix, each entry corresponds to the weight for a mixture;
#         Var: updated Var, K*1 matrix, each entry corresponds to the variance for a mixtu
#         post: updated post, n*K matrix, each row corresponds to the soft counts for all
#         LL: Numpy array for Loglikelihood values
def mixGauss_part2(X,K,Mu,P,Var):
    n,d = np.shape(X) # n data points of dimension d
    post = np.zeros((n,K)) # posterior probs tbd

    #Write your code here
    #Use function Estep and Mstep as two subroutines
    L= []

    for c in [0,1]:
        post, ll = Estep_part2(X, K, Mu, P, Var)
        L.append(ll)
        Mu, P, Var = Mstep_part2(X, K, Mu, P, Var, post)

    i = 0
    while np.abs(L[i+1] - L[i]) > 10**(-6) * np.abs(L[i+1]):
        post, ll = Estep_part2(X, K, Mu, P, Var)
        if np.abs(ll - L[i+1]) > 10**(-6) * np.abs(ll):
            L.append(ll)
            i += 1
        else:
            break
        Mu, P, Var = Mstep_part2(X, K, Mu, P, Var, post)

    LL = np.array(L)

    return (Mu,P,Var,post,LL)


# fill incomplete Matrix
# input: X: n*d incomplete data matrix;
#        K: number of mixtures;
#        Mu: K*d matrix, each row corresponds to a mixture mean;
#        P: K*1 matrix, each entry corresponds to the weight for a mixture;
#        Var: K*1 matrix, each entry corresponds to the variance for a mixture;
# output: Xnew: n*d data matrix with unrevealed entries filled
def fillMatrix(X,K,Mu,P,Var):
    n,d = np.shape(X)
    Xnew = np.copy(X)

    post, LL = Estep_part2(X, K, Mu, P, Var)
```

```
    for i in xrange(n):
        for j in xrange(d):
            if Xnew[i][j] == 0:
                probability = post[i]
                means = Mu[:, j]

                new = np.dot(probability, means)
                Xnew[i][j] = new
    return Xnew

#-------------------------------------,-------------------------------------
```