MIDS-W261-2016-HWK-Week06-Group12-Krunic-Cordell-Dunmire

HW6.0.

In mathematics, computer science, economics, or management science what is mathematical optimization? Give an example of a optimization problem that you have worked with directly or that your organization has worked on. Please describe the objective function and the decision variables. Was the project successful (deployed in the real world)? Describe.

Mathematical optimization is the process of finding the optimal solution or best member for a particular problem using mathematical or numerical principles. One example in Telecommunications is develop predictive models for customer churn. This involves minimizing a cost-function (e.g. the objective function) using various input variables like previous billing history, data plan, and other important features.

HW6.1

Optimization theory: For unconstrained univariate optimization what are the first order Necessary Conditions for Optimality (FOC). What are the second order optimality conditions (SOC)? Give a mathematical defintion. Also in python, plot the univartiate function

X^3 -12x^2-6 defined over the real domain -6 to +6.

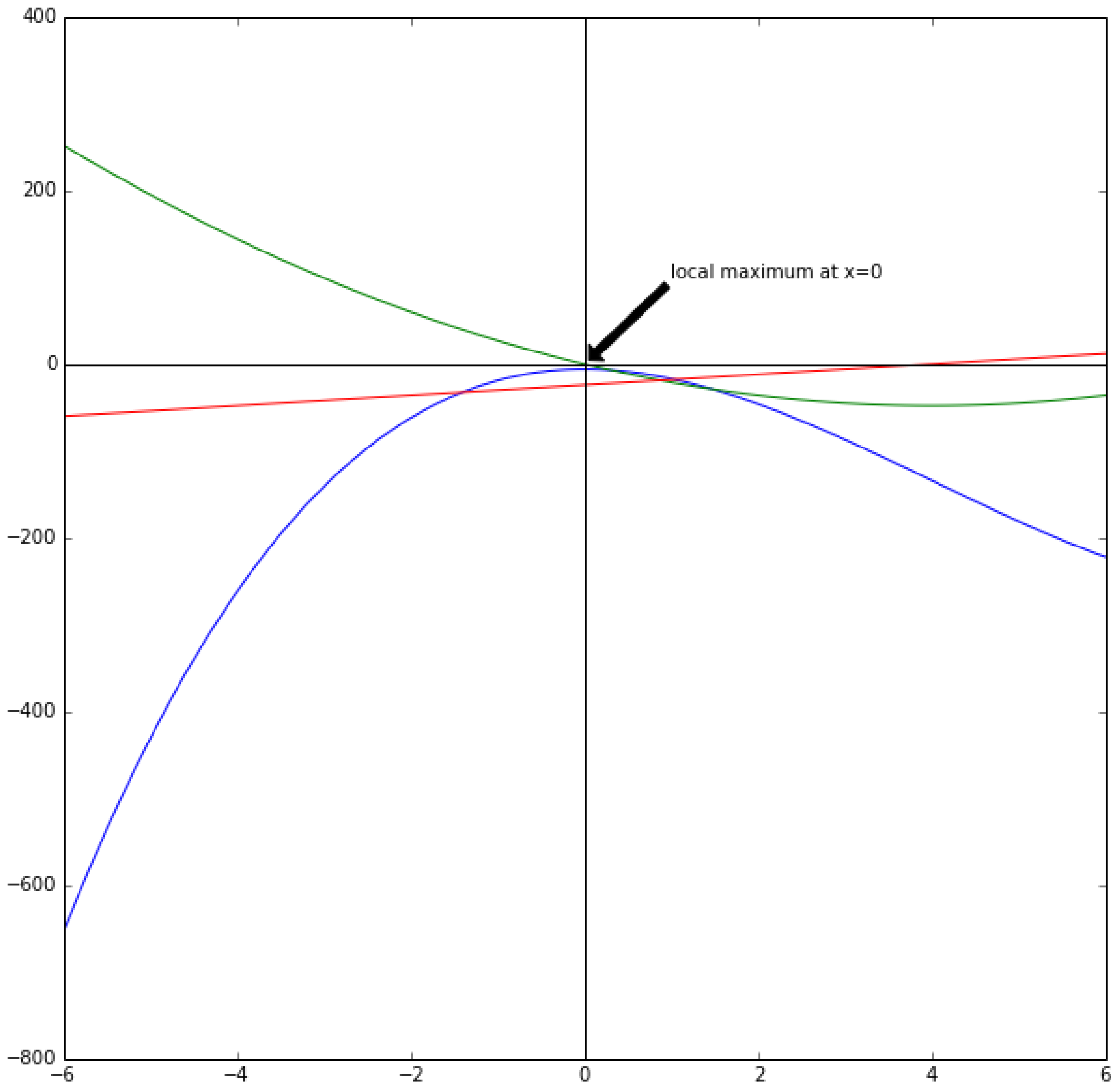
Also plot its corresponding first and second derivative functions. Eyeballing these graphs, identify candidate optimal points and then classify them as local minimums or maximums. Highlight and label these points in your graphs. Justify your responses using the FOC and SOC.

A: Unconstrained univariate optimization

First Order :

Second Order : *[Math Processing Error]* for a minimum and *[Math Processing Error]* for a maximum

In [2]:
```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
from numpy import power, square, linspace

x = linspace(-6,6,100) # 100 linearly spaced numbers from -6 to +6
y = power(x,3)-(12*square(x))-6   # computing the values of x^3 - 1
2x^2 - 6
y_prime = 3*square(x)-24*x # first derivative 3x^2 - 24x
y_doubleprime = 6*x-24 # second derivative

# compose plot
plt.figure(figsize=(10,10))
plt.plot(x,y)
plt.plot(x,y_prime) # first derivative
plt.plot(x,y_doubleprime) # second derivative

# plot x,y axes for reference
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')

plt.annotate('local maximum at x=0', xy=(0, 0), xytext=(1, 100),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )

plt.show() # show the plot
```



x = 0 would be a good candidate for an optimization. In the first graph above, the graph of the actual function, this appears to be supported (by eyeball). This seems to be supported by our conditions. The first order condition states that *[Math Processing Error]* should be 0, which would indicate an optimal point. The second condition states that *[Math Processing Error]* for a local maximum, which is clearly demonstrated in the second graph.

B

For unconstrained multi-variate optimization what are the first order Necessary Conditions for Optimality (FOC). What are the second order optimality conditions (SOC)? Give a mathematical defintion. What is the Hessian matrix in this context?

The FOC for multivariate is where the gradient *[Math Processing Error]*: *[Math Processing Error]* where *[Math Processing Error]*

The SOC for a local maxmimum is when the Hessian evaluated at that point is negative definite. The Hessian is given by:
[Math Processing Error]

HW6.2

Taking x=1 as the first approximation(xt1) of a root of X^3 + 2x -4 = 0, use the Newton-Raphson method to calculate the second approximation (denoted as xt2) of this root. (Hint the solution is xt2=1.2)

On Paper Iteration function is *[Math Processing Error]*

Derivative of function = 3X + 2

Step 0:

[Math Processing Error] = 1

Step 1:

[Math Processing Error]

In [3]:
```
#Programmatically

# equation f(x) = x^3 + 2x - 4
def f(x):
    return x**3+2*x-4

# first derivative of equation f'(x) = 3x^2 + 2
def df(x):
    return 3*x**2+2

# computing where the tangent line at x = 0
def dx(f, x):
    return abs(0-f(x))

# starting with x0 as the first guess, iterate
# do this only one iteration to get the second estimate
def newtons_method(f, df, x0, e):
    delta = dx(f, x0)
    while delta > e:
        x0 = x0 - f(x0)/df(x0)
        delta = dx(f, x0)
        break
    print 'Next approximation is at: ', x0

newtons_method(f, df, 1.0, 0.000001)
```

```
Next approximation is at:  1.2
```

HW6.3 Convex optimization

What makes an optimization problem convex? What are the first order Necessary Conditions for Optimality in convex optimization. What are the second order optimality conditions for convex optimization? Are both necessary to determine the maximum or minimum of candidate optimal solutions?

==================

An optimization problem is convex if its objective function *[Math Processing Error]* is convex, the inequality constraints *[Math Processing Error]* are convex and the quality constraints *[Math Processing Error]* are affine.

The first order necessary conditions for optimality in convex optimization is
[Math Processing Error]

The function is globally above the tangent at y.

The second order optimality condititions for convex optimization is:
[Math Processing Error]

The function is flat or curved upwards in every direction.

Both FOC and SOC are not necessary to determine a maximum or minimum candidate optimal solutions.

Fill in the BLANKS here:

*Convex minimization, a subfield of optimization, studies the problem of minimizing **convex** functions over **convex** sets. The **convexity** property can make optimization in some sense "easier" than the general case - for example, any local minimum must be a global minimum.*

HW 6.4

The learning objective function for weighted ordinary least squares (WOLS) (aka weight linear regression) is defined as follows:

0.5 *sumOverTrainingExample i (weight_i* (W * X_i - y_i)^2)

Where training set consists of input variables X (in vector form) and a target variable y, and W is the vector of coefficients for the linear regression model.

Derive the gradient for this weighted OLS by hand; showing each step and also explaining each step.

===

Approach A

Solution:

Since we wish to minimize this objective function with respect to *[Math Processing Error]*, simply have to compute the partial derivative of *[Math Processing Error]*. The gradient is the combined vector of the partials.

Step 1

Here we re-define the objective function without the absolute value to make it continuous. This is allowed since we are dealing with real numbers. Making this change will let the following computations resolve easily.
[Math Processing Error]

Step 2

We take a parital derivative with respect to *[Math Processing Error]*. Notice we can pass it to the inside of the sum since it is not a function of the interior.
[Math Processing Error]

Step 3

We resolve the interior derivative and notice that all of the terms *[Math Processing Error]* fall-out and we are left with a simplified expression inside the sum.
[Math Processing Error][Math Processing Error][Math Processing Error][Math Processing Error]

Thus our combined gradient for the weighted OLS, denoted *[Math Processing Error]* is:
[Math Processing Error]

where *[Math Processing Error]* is the quantity described above.

Approach B

This is our learning objective function for a weighted ordinary least squares.

Cost = *[Math Processing Error]*

We would like to minimize this and so we will find the gradient.

So in keeping with gradient descent we will take the partial derivative with respect to W:

[Math Processing Error]

With this we apply the chain rule and get the following result

[Math Processing Error]

Which we can then simplify to:

[Math Processing Error]

HW 6.5

Write a MapReduce job in MRJob to do the training at scale of a weighted OLS model using gradient descent.

Generate one million datapoints just like in the following notebook: http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipynb (http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipynb)

Weight each example as follows:

weight(x)= abs(1/x)

Sample 1% of the data in MapReduce and use the sampled dataset to train a (weighted if available in SciKit-Learn) linear regression model locally using SciKit-Learn (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html))

Plot the resulting weighted linear regression model versus the original model that you used to generate the data. Comment on your findings.

In [1]:
```
# generate 10^6 data points and save to disk
import numpy as np
size = 1000000
x = np.random.uniform(-4, 4, size)
y = x * 1.0 - 4 + np.random.normal(0,0.5,size)
data = zip(y,x)
np.savetxt('LinearRegression.csv',data,delimiter = ",")
```

In [2]:

```
%%writefile MrJobBatchGDUpdate_LinearRegression.py
from mrjob.job import MRJob, MRStep

# This MrJob calculates the gradient of the entire training set
#     Mapper: calculate partial gradient for each example
#
class MrJobBatchGDUpdate_LinearRegression(MRJob):
    # run before the mapper processes any input
    def read_weightsfile(self):
        # Read weights file
        with open('weights.txt', 'r') as f:
            self.weights = [float(v) for v in f.readline().split
(',')]
        # Initialze gradient for this iteration
        self.partial_Gradient = [0]*len(self.weights)
        self.partial_count = 0

    # Calculate partial gradient for each example
    def partial_gradient(self, _, line):
        D = (map(float,line.split(',')))
        # y_hat is the predicted value given current weights
        y_hat = self.weights[0]+self.weights[1]*D[1]
        # Update partial gradient vector with gradient form current
example
        self.partial_Gradient =  [self.partial_Gradient[0] + D[0]-y
_hat,
                                  self.partial_Gradient[1]+ (D[0]-y
_hat) *D[1]]
        self.partial_count = self.partial_count + 1
        #yield None, (D[0]-y_hat,(D[0]-y_hat)*D[1],1)

    # Finally emit in-memory partial gradient and partial count
    def partial_gradient_emit(self):
        yield None, (self.partial_Gradient,self.partial_count)

    # Accumulate partial gradient from mapper and emit total gradie
nt
    # Output: key = None, Value = gradient vector
    def gradient_accumulater(self, _, partial_Gradient_Record):
        total_gradient = [0]*2
        total_count = 0
        for partial_Gradient,partial_count in partial_Gradient_Reco
rd:
            total_count = total_count + partial_count
            total_gradient[0] = total_gradient[0] + partial_Gradien
t[0]
            total_gradient[1] = total_gradient[1] + partial_Gradien
t[1]
        yield None, [v/total_count for v in total_gradient]

    def steps(self):
        return [MRStep(mapper_init=self.read_weightsfile,
                       mapper=self.partial_gradient,
                       mapper_final=self.partial_gradient_emit,
                       reducer=self.gradient_accumulater)]

if __name__ == '__main__':
    MrJobBatchGDUpdate_LinearRegression.run()
```

```
Writing MrJobBatchGDUpdate_LinearRegression.py
```

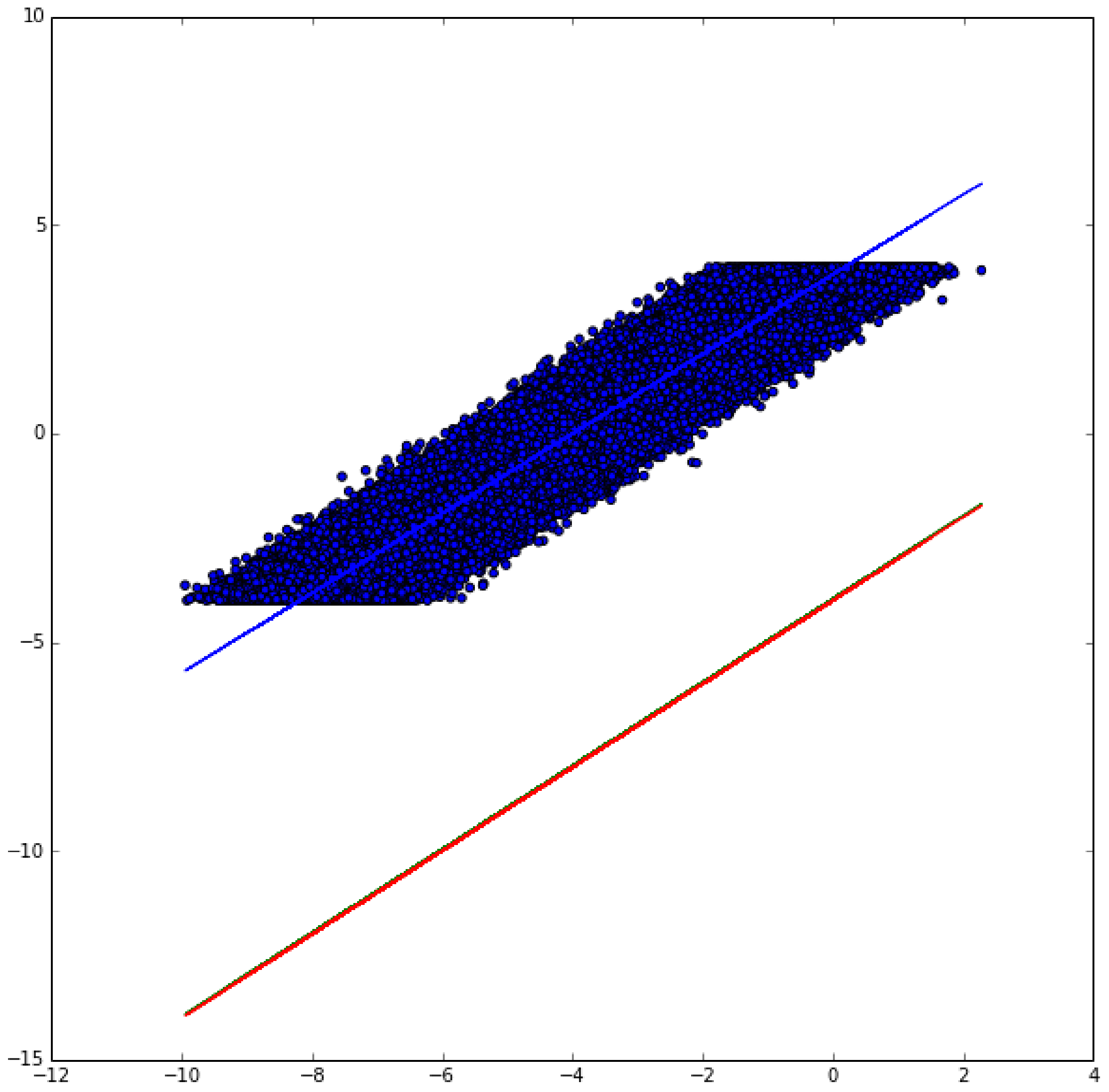
In [2]:
```
from numpy import random,array
from MrJobBatchGDUpdate_LinearRegression import MrJobBatchGDUpdate_
LinearRegression

learning_rate = 0.05
stop_criteria = 0.000005

# Generate random values as inital weights
weights = array([random.uniform(-3,3),random.uniform(-3,3)])
# Write the weights to the files
with open('weights.txt', 'w+') as f:
    f.writelines(','.join(str(j) for j in weights))

# create a mrjob instance for batch gradient descent update over al
l data
mr_job = MrJobBatchGDUpdate_LinearRegression(args=['LinearRegressio
n.csv',
                                                    '--file', 'weigh
ts.txt',
                                                    '--strict-protoc
ols'])
# Update centroids iteratively
i = 0
while(1):
    print 'iteration: {0:4d} weights: {1:3.10f} {2:3.10f}'.format
(i, weights[0], weights[1])
    # Save weights from previous iteration
    weights_old = weights
    with mr_job.make_runner() as runner:
        runner.run()
        # stream_output: get access of the output
        for line in runner.stream_output():
            # value is the gradient value
            key,value =  mr_job.parse_output_line(line)
            # Update weights
            weights = weights + learning_rate*array(value)
    i = i + 1
    # Write the updated weights to file
    with open('weights.txt', 'w+') as f:
        f.writelines(','.join(str(j) for j in weights))
    # Stop if weights get converged
    if(sum((weights_old-weights)**2)<stop_criteria):
        break

print 'Final weights: {0:3.10f} {1:3.10f}'.format(weights[0], weigh
ts[1])
```

```
iteration:    0 weights: -2.8007643516 -0.2380593320
iteration:    1 weights: -2.8604548401 0.0919394606
iteration:    2 weights: -2.9172403907 0.3339342020
iteration:    3 weights: -2.9712450261 0.5113969438
iteration:    4 weights: -3.0225922286 0.6415392189
iteration:    5 weights: -3.0714034578 0.7369818689
iteration:    6 weights: -3.1177971435 0.8069794891
iteration:    7 weights: -3.1618880264 0.8583182849
iteration:    8 weights: -3.2037867466 0.8959744488
iteration:    9 weights: -3.2435996124 0.9235969320
iteration:   10 weights: -3.2814284967 0.9438614504
iteration:   11 weights: -3.3173708240 0.9587300679
iteration:   12 weights: -3.3515196209 0.9696415434
iteration:   13 weights: -3.3839636094 0.9776509072
iteration:   14 weights: -3.4147873302 0.9835318097
iteration:   15 weights: -3.4440712832 0.9878515709
iteration:   16 weights: -3.4718920803 0.9910262125
iteration:   17 weights: -3.4983226033 0.9933608121
iteration:   18 weights: -3.5234321631 0.9950790932
iteration:   19 weights: -3.5472866594 0.9963451241
iteration:   20 weights: -3.5699487361 0.9972792278
iteration:   21 weights: -3.5914779343 0.9979696495
iteration:   22 weights: -3.6119308391 0.9984811117
iteration:   23 weights: -3.6313612220 0.9988610875
iteration:   24 weights: -3.6498201775 0.9991444009
iteration:   25 weights: -3.6673562534 0.9993565997
iteration:   26 weights: -3.6840155768 0.9995164294
iteration:   27 weights: -3.6998419725 0.9996376464
iteration:   28 weights: -3.7148770777 0.9997303489
iteration:   29 weights: -3.7291604499 0.9998019515
iteration:   30 weights: -3.7427296708 0.9998579008
iteration:   31 weights: -3.7556204442 0.9999021995
iteration:   32 weights: -3.7678666896 0.9999377917
iteration:   33 weights: -3.7795006313 0.9999668440
iteration:   34 weights: -3.7905528829 0.9999909532
iteration:   35 weights: -3.8010525278 1.0000112974
iteration:   36 weights: -3.8110271953 1.0000287475
iteration:   37 weights: -3.8205031337 1.0000439489
iteration:   38 weights: -3.8295052788 1.0000573810
iteration:   39 weights: -3.8380573198 1.0000694015
iteration:   40 weights: -3.8461817618 1.0000802784
iteration:   41 weights: -3.8538999842 1.0000902136
iteration:   42 weights: -3.8612322980 1.0000993602
iteration:   43 weights: -3.8681979982 1.0001078356
iteration:   44 weights: -3.8748154155 1.0001157303
iteration:   45 weights: -3.8811019638 1.0001231152
iteration:   46 weights: -3.8870741865 1.0001300465
iteration:   47 weights: -3.8927477997 1.0001365695
iteration:   48 weights: -3.8981377338 1.0001427209
iteration:   49 weights: -3.9032581728 1.0001485315
iteration:   50 weights: -3.9081225911 1.0001540272
iteration:   51 weights: -3.9127437899 1.0001592302
iteration:   52 weights: -3.9171339300 1.0001641599
iteration:   53 weights: -3.9213045643 1.0001688336
```

```
iteration:   54 weights: -3.9252666680 1.0001732665
iteration:   55 weights: -3.9290306675 1.0001774726
iteration:   56 weights: -3.9326064681 1.0001814646
iteration:   57 weights: -3.9360034797 1.0001852542
iteration:   58 weights: -3.9392306416 1.0001888523
iteration:   59 weights: -3.9422964462 1.0001922690
iteration:   60 weights: -3.9452089615 1.0001955138
iteration:   61 weights: -3.9479758517 1.0001985955
iteration:   62 weights: -3.9506043982 1.0002015226
iteration:   63 weights: -3.9531015181 1.0002043029
iteration:   64 weights: -3.9554737826 1.0002069438
iteration:   65 weights: -3.9577274346 1.0002094525
Final weights: -3.9598684046 1.0002118355
```

In [3]:
```
%matplotlib inline
# construct a linear model with scikit learn
from sklearn.linear_model import LinearRegression
import matplotlib
import matplotlib.pyplot as plt
import random
from numpy import loadtxt, hsplit, asarray

def mr_predict(X, coefs):
    y = []
    for x in X:
        y.append(coefs[0] + x * coefs[1])
    return asarray(y)

def actual(X):
    y = []
    for x in X:
        y.append(x - 4.0)
    return asarray(y)

# read the original generated data set
data = loadtxt('LinearRegression.csv', delimiter = ",")

# generate a 1% random sample from the data set
random_sample = [ data[i] for i in sorted(random.sample(xrange(len
(data)), len(data)/100)) ]

# split the data into an X array and Y array
train_data = hsplit(asarray(data),2)

# create a sample weight vector weight(x) = 1/abs(x)
w = [abs(1.0/x) for x,y in random_sample]

# create an instance of the scikit linear regression and train it
lr = LinearRegression()
model = lr.fit(train_data[0], train_data[1], w)

# plot the results
plt.figure(figsize=(10,10))
plt.plot(train_data[0], model.predict(train_data[0]))
plt.plot(train_data[0], mr_predict(train_data[0], weights))
plt.plot(train_data[0], actual(train_data[0]))
plt.scatter(train_data[0], train_data[1])

plt.show()
```

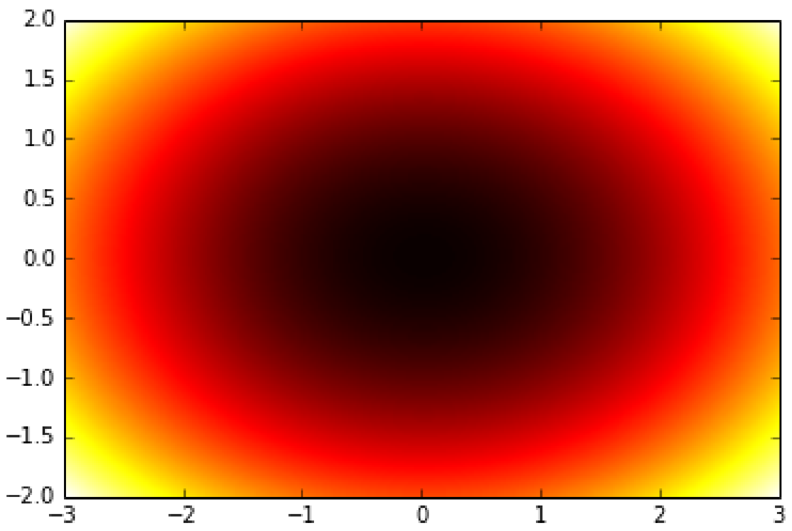
```
/Library/Python/2.7/site-packages/ipykernel/__main__.py:35: Deprec
ationWarning: The n_jobs parameter in fit is deprecated and will b
e removed in 0.17. It has been moved from the fit method to the Li
nearRegression class constructor.
```

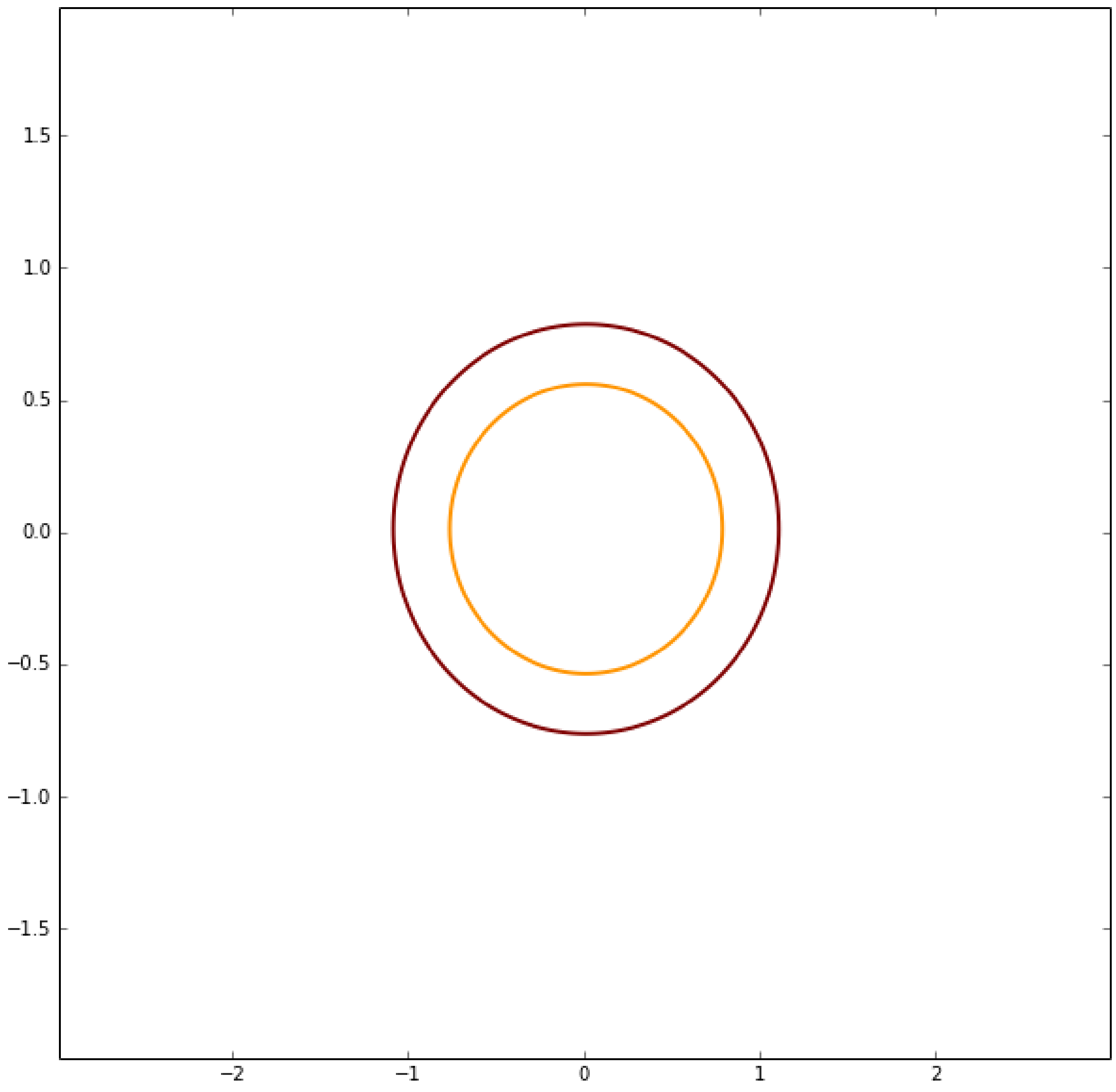


HW6.5.1 (OPTIONAL)

Using MRJob and in Python, plot the error surface for the weighted linear regression model using a heatmap and contour plot. Also plot the current model in the original domain space. (Plot them side by side if possible) Plot the path to convergence (during training) for the weighted linear regression model in plot error space and in the original domain space. Make sure to label your plots with iteration numbers, function, model space versus original domain space, etc. Comment on convergence and on the mean squared error using your weighted OLS algorithm on the weighted dataset versus using the weighted OLS algorithm on the uniformly weighted dataset.

In [6]:
```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

x = np.linspace(-3,3,30)
y = x
z = np.square(x)+(2*np.square(y))

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z = np.square(X)+(2*np.square(Y))
# compose plot

im = plt.imshow(Z, interpolation='bilinear', origin='lower', cmap=c
m.hot, extent=(-3, 3, -2, 2))
levels = np.arange(-1.2, 1.6, 0.6)
plt.figure(figsize=(10,10))
CS = plt.contour(Z, levels,
                 origin='lower',
                 linewidths=2,
                 extent=(-3, 3, -2, 2))

plt.show() # show the plot
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2
−1
0
1
2

HW6.6 Clean up notebook for GMM via EM

Using the following notebook as a starting point:

http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/0t7985e40fovlkw/EM-GMM-MapReduce%20Design%201.ipynb (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/0t7985e40fovlkw/EM-GMM-MapReduce%20Design%201.ipynb)

Improve this notebook as follows: -- Add in equations into the notebook (not images of equations) -- Number the equations -- Make sure the equation notation matches the code and the code and comments refer to the equations numbers -- Comment the code -- Rename/Reorganize the code to make it more readable -- Rerun the examples similar graphics (or possibly better graphics)

Introduction

This is a map-reduce version of expectation maximization algorithm for a mixture of Gaussians model. There are two mrJob MR packages, mr_GMixEmIterate and mr_GMixEmInitialize. The driver calls the mrJob packages and manages the iteration.

E Step

Given priors *[Math Processing Error]*, mean vector *[Math Processing Error]* and covariance matrix *[Math Processing Error]*: calculate the probablility that each of the data points belongs to a class *[Math Processing Error]*:
[Math Processing Error]

M Step

Given probabilities: update priors, mean and covariance
[Math Processing Error][Math Processing Error][Math Processing Error]

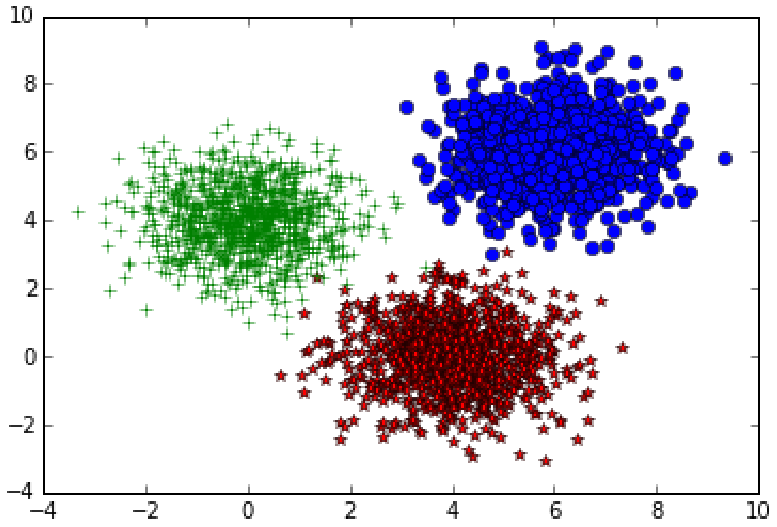
We can compute the Guassian probability as:
[Math Processing Error]

Data Generation

In [31]:
```
%matplotlib inline
from numpy import random, append
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import json

# generate 3 clusters the size of 1000 points each
size1 = size2 = size3 = 1000

# cluster 1 is centered about (4,0)
samples1 = random.multivariate_normal([4, 0], [[1, 0],[0, 1]], size
1)
data = samples1

# cluster 2 is centered about (6,6)
samples2 = random.multivariate_normal([6, 6], [[1, 0],[0, 1]], size
2)
data = append(data,samples2, axis=0)

# cluster 3 is centered about (0,4)
samples3 = random.multivariate_normal([0, 4], [[1, 0],[0, 1]], size
3)
data = append(data,samples3, axis=0)

# we've appended all the generated data for the three clusters into
a single list
# now randomize the list - shaken, not stirred. Save the result to
a JSON file.
data = data[random.permutation(size1+size2+size3),]
with open("data.txt", "w") as f:
    for row in data.tolist():
        json.dump(row, f)
        f.write("\n")
```

Data Visualization

In [32]:
```
plt.plot(samples1[:, 0], samples1[:, 1],'*', color = 'red')
plt.plot(samples2[:, 0], samples2[:, 1],'o',color = 'blue')
plt.plot(samples3[:, 0], samples3[:, 1],'+',color = 'green')
plt.show()
```



Initialization

If we didn't already know, from the visualization we conclude there are 3 clusters, so we'll set k = 3.

In [33]:

```
%%writefile mr_GMixEmInitialize.py
from mrjob.job import MRJob, MRStep

from numpy import mat, zeros, shape, random, array, zeros_like, do
t, linalg
from numpy import mean as npmean
from random import sample
import json
from math import pi, sqrt, exp, pow


class MrGMixEmInit(MRJob):
    DEFAULT_PROTOCOL = 'json'

    def __init__(self, *args, **kwargs):
        super(MrGMixEmInit, self).__init__(*args, **kwargs)

        self.numMappers = 1     #number of mappers
        self.count = 0

    def configure_options(self):
        super(MrGMixEmInit, self).configure_options()
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt
is stored')

    # each line is a data point of the cluster x_j
    # take the first 2*k lines and emit them; ignore the rest
    def mapper(self, key, xjIn):
        #something simple to grab random starting point
        #collect the first 2*k
        if self.count <= 2*self.options.k:
            self.count += 1
            yield (1,xjIn)

    # every key is the same = 1, so we should have 2*k items show u
p at the reducer
    # all we want is to accumulate the points into a list from whic
h we will draw k randomly
    # the yield just throws the first k*2 samples into the aether
    def reducer(self, key, xjIn):
        #accumulate data points mapped to 0 from 1st mapper and pul
l out k of them as starting point
        initial_point_set = []
        for xj in xjIn:
            x = json.loads(xj)
            initial_point_set.append(x)
            yield 1, xj

        # take k random samples
        centroids = [ initial_point_set[i] for i in sample(xrange(l
en(initial_point_set)), self.options.k) ]

        # use the covariance of the selected centers as the startin
g guess for covariances
        # first, calculate mean of centers, the mu vector from Equa
tion 2
        mean = npmean(centroids, axis=0)

        # then accumulate the deviations along the diagonal of a sq
uare matrix
        # this is the initialization of the Sigma matrix from Equat
ion 3
        cov = zeros((len(mean),len(mean)),dtype=float)
        for x in centroids:
            xmm = array(x) - mean
            for i in range(len(mean)):
                cov[i,i] = cov[i,i] + xmm[i]*xmm[i]

        # This implements the 1/n_k from Equation 3 and then invert
s the matrix
        cov = cov/(float(self.options.k))
        covInv = linalg.inv(cov)

        cov_1 = [covInv.tolist()]*self.options.k

        # dump our initialization results to a file for debug inspe
ction
        jDebug = json.dumps([centroids,mean.tolist(),cov.tolist(),c
ovInv.tolist(),cov_1])
        with open(self.options.pathName + "debug.txt", 'w') as debu
gfile:
            debugfile.write(jDebug)

        #also need a starting guess at the pi's - prior probabiliti
es - Equation 4
        #initialize them all with the same number - 1/k - equally p
robably for each cluster

        pi = zeros(self.options.k,dtype=float)

        for i in range(self.options.k):
            pi[i] = 1.0/float(self.options.k)

        #form output object
        outputList = [pi.tolist(), centroids, cov_1]

        jsonOut  = json.dumps(outputList)

        #write new parameters to file
        fullPath = self.options.pathName + 'intermediateResults.tx
t'
        fileOut = open(fullPath,'w')
        fileOut.write(jsonOut)
        fileOut.close()

def steps(self):
    return [MRStep(mapper=self.mapper,
                   reducer=self.reducer)]

if __name__ == '__main__':
    MrGMixEmInit.run()
```

```
Overwriting mr_GMixEmInitialize.py
```

Iteration

Mapper - each mapper needs k vector means and covariance matrices to make probability calculations. Can also accumulate partial sum (sum restricted to the mapper's input) of quantities required for update. Then it emits partial sum as single output from combiner.

Emit (dummy_key, partial_sum_for_all_k's)

Reducer –the iterator pulls in the partial sum for all k's from all the mappers and combines in a single reducer. In this case the reducer emits a single (json'd python object) with the new means and covariances.

In [34]:

```
%%writefile mr_GMixEmIterate.py
from mrjob.job import MRJob

from math import sqrt, exp, pow,pi
from numpy import zeros, shape, random, array, zeros_like, dot, lin
alg
import json

# compute the guassian probability for a point x as in Equation 5
def gauss(x, mu, P_1):
    xtemp = x - mu
    n = len(x)
    p = exp(- 0.5*dot(xtemp,dot(P_1,xtemp)))
    detP = 1/linalg.det(P_1)
    p = p/(pow(2.0*pi,n/2.0)*sqrt(detP))
    return p

class MrGMixEm(MRJob):
    DEFAULT_PROTOCOL = 'json'

    def __init__(self, *args, **kwargs):
        super(MrGMixEm, self).__init__(*args, **kwargs)

        with open(self.options.pathName + 'intermediateResults.tx
t', 'r') as fileIn:
            inputJson = fileIn.read()

        inputList = json.loads(inputJson)
        self.phi =   inputList[0]  # prior class probabilities (Eq
n. 3)
        self.means = inputList[1]  # means (Eqn. 2)
        self.cov_1 = inputList[2]  # inverse covariance matrices us
ed for Eqn. 4

        #sum of weights - by cluster
        self.new_phi = zeros_like(self.phi)        #partial weighte
d sum of weights
        self.new_means = zeros_like(self.means)
        self.new_cov = zeros_like(self.cov_1)

        self.numMappers = 1             #number of mappers
        self.count = 0                  #passes through mapper


    def configure_options(self):
        super(MrGMixEm, self).configure_options()

        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt
is stored')
```

```
    # each item is a point in the set of data points
    def mapper(self, _, val):
        #accumulate partial sums for each mapper
        xList = json.loads(val)
        x = array(xList)
        wtVect = zeros_like(self.phi)

        # for each cluster compute the weights from the guassian pr
obabilities for this point
        # as shown in Equation 4
        for i in range(self.options.k):
            wtVect[i] = self.phi[i]*gauss(x,self.means[i],self.cov_
1[i])
        wtSum = sum(wtVect)
        wtVect = wtVect/wtSum
        #accumulate to update est of probability densities.
        #increment count
        self.count += 1
        #accumulate weights for phi est
        self.new_phi = self.new_phi + wtVect
        for i in range(self.options.k):
            #accumulate weighted x's for mean calc
            self.new_means[i] = self.new_means[i] + wtVect[i]*x
            #accumulate weighted squares for cov estimate
            xmm = x - self.means[i]
            covInc = zeros_like(self.new_cov[i])

            for l in range(len(xmm)):
                for m in range(len(xmm)):
                    covInc[l][m] = xmm[l]*xmm[m]
            self.new_cov[i] = self.new_cov[i] + wtVect[i]*covInc
        #dummy yield - real output passes to mapper_final in self


    def mapper_final(self):

        out = [self.count, (self.new_phi).tolist(), (self.new_mean
s).tolist(), (self.new_cov).tolist()]
        jOut = json.dumps(out)

        yield 1,jOut



    def reducer(self, key, xs):
        #accumulate partial sums
        first = True
        #accumulate partial sums
        #xs us a list of paritial stats, including count, phi, mea
n, and covariance.
        #Each stats is k-length array, storing info for k component
s
        for val in xs:
            if first:
                temp = json.loads(val)
                #totCount, totPhi, totMeans, and totCov are all arr
ays
                totCount = temp[0]
                totPhi = array(temp[1])
                totMeans = array(temp[2])
                totCov = array(temp[3])
                first = False
            else:
                temp = json.loads(val)
                #cumulative sum of four arrays
                totCount = totCount + temp[0]
                totPhi = totPhi + array(temp[1])
                totMeans = totMeans + array(temp[2])
                totCov = totCov + array(temp[3])
        #finish calculation of new probability parameters. array di
vided by array
        newPhi = totPhi/totCount
        #initialize these to something handy to get the right size
arrays
        newMeans = totMeans
        newCov_1 = totCov
        for i in range(self.options.k):
            newMeans[i,:] = totMeans[i,:]/totPhi[i]
            tempCov = totCov[i,:,:]/totPhi[i]
            #almost done.  just need to invert the cov matrix.  inv
ert here to save doing a matrix inversion
            #with every input data point.
            newCov_1[i,:,:] = linalg.inv(tempCov)

        outputList = [newPhi.tolist(), newMeans.tolist(), newCov_1.
tolist()]
        jsonOut = json.dumps(outputList)

        #write new parameters to file
        with open(self.options.pathName + 'intermediateResults.tx
t','w') as fileOut:
            fileOut.write(jsonOut)

if __name__ == '__main__':
    MrGMixEm.run()
```

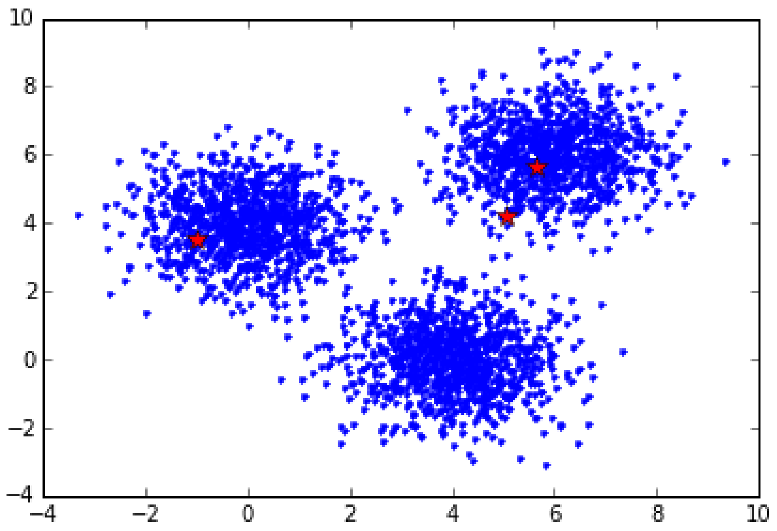
```
Overwriting mr_GMixEmIterate.py
```

Driver

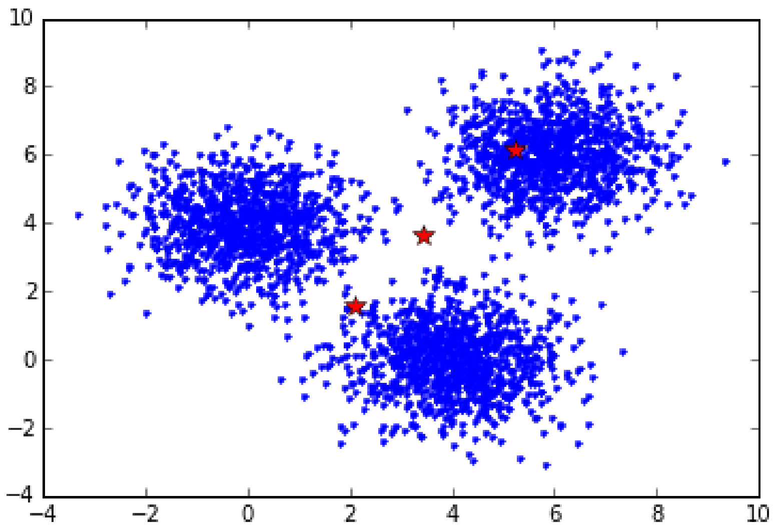
In [35]:

```
from mr_GMixEmInitialize import MrGMixEmInit
from mr_GMixEmIterate import MrGMixEm
import json
from math import sqrt
import matplotlib.pyplot as plt

def plot_iteration(means):
    plt.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    plt.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    plt.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    plt.plot(means[0][0], means[0][1],'*',markersize =10,color = 'r
ed')
    plt.plot(means[1][0], means[1][1],'*',markersize =10,color = 'r
ed')
    plt.plot(means[2][0], means[2][1],'*',markersize =10,color = 'r
ed')
    plt.show()

def dist(x,y):
    #euclidean distance between two lists
    sum = 0.0
    for i in range(len(x)):
        temp = x[i] - y[i]
        sum += temp * temp
    return sqrt(sum)

# first run the initializer to get starting centroids
filePath = 'data.txt'
mrJob = MrGMixEmInit(args=[filePath,
                           '--k','3',
                           '--pathName','/Users/rcordell/Documents/
MIDS/W261/week06/HW6/',
                           '--strict-protocols'])

# after running this MRJob we will have an initialized mu, sigma, a
nd priors (pi's)
# as indicated in Equations 2, 3 and 4
with mrJob.make_runner() as runner:
    runner.run()

# pull out the centroid values to compare with values after one ite
ration
emPath = "/Users/rcordell/Documents/MIDS/W261/week06/HW6/intermedia
teResults.txt"
with open(emPath, "r") as fileIn:
    paramJson = fileIn.read()

delta = 10
iter_num = 0
#Begin iteration on change in centroids
while delta > 0.02:
    print "Iteration" + str(iter_num)
    iter_num = iter_num + 1
    # parse old centroid values
    oldParam = json.loads(paramJson)
    # run one iteration
    oldMeans = oldParam[1]
    mrJob2 = MrGMixEm(args=[filePath,
                           '--k','3',
                           '--pathName','/Users/rcordell/Documents/
MIDS/W261/week06/HW6/',
                           '--strict-protocols'])
    with mrJob2.make_runner() as runner:
        runner.run()

    #compare new centroids to old ones
    with open(emPath, 'r') as fileIn:
        paramJson = fileIn.read()

    newParam = json.loads(paramJson)

    k_means = len(newParam[1])
    newMeans = newParam[1]

    delta = 0.0
    for i in range(k_means):
        delta += dist(newMeans[i],oldMeans[i])

    print oldMeans
    plot_iteration(oldMeans)
print "Iteration" + str(iter_num)
print newMeans
plot_iteration(newMeans)
```

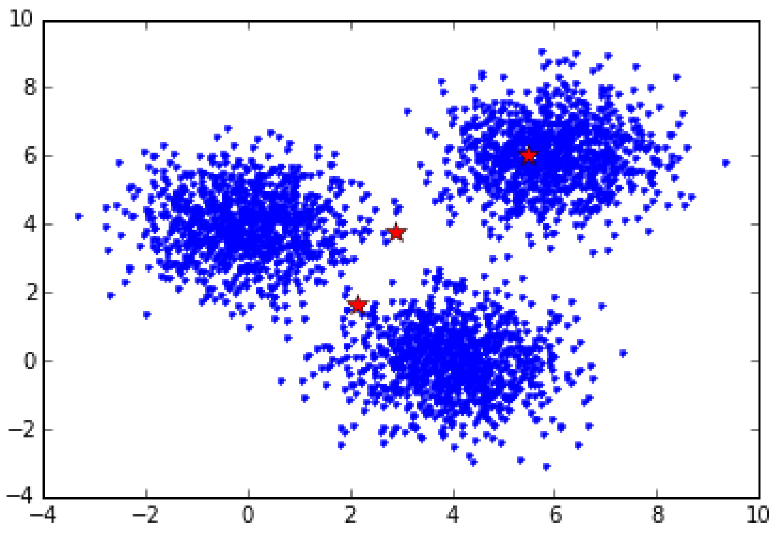
```
Iteration0
[[-0.9888830377827774, 3.5024720410779446], [5.632531279247575, 5.
639369810591645], [5.075913188878569, 4.220503792169227]]
```



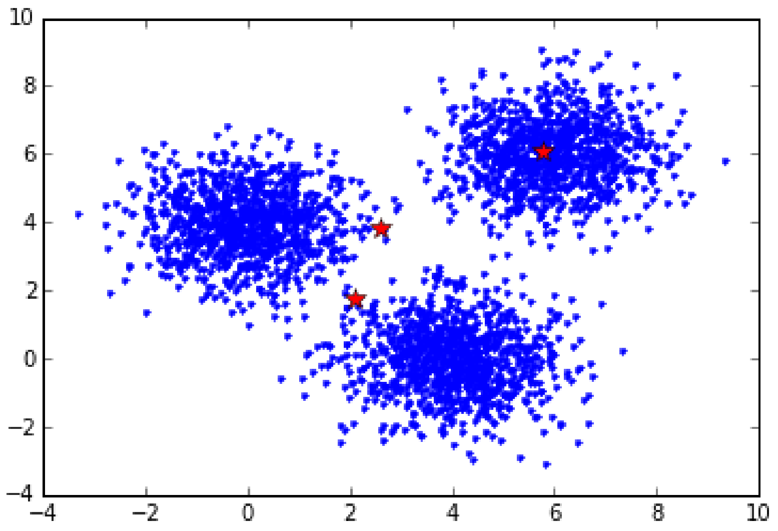
```
Iteration1
[[2.1134726192285447, 1.5984239730818919], [5.222596281352015, 6.1
52235142254203], [3.445601287149526, 3.6438840919083004]]
```



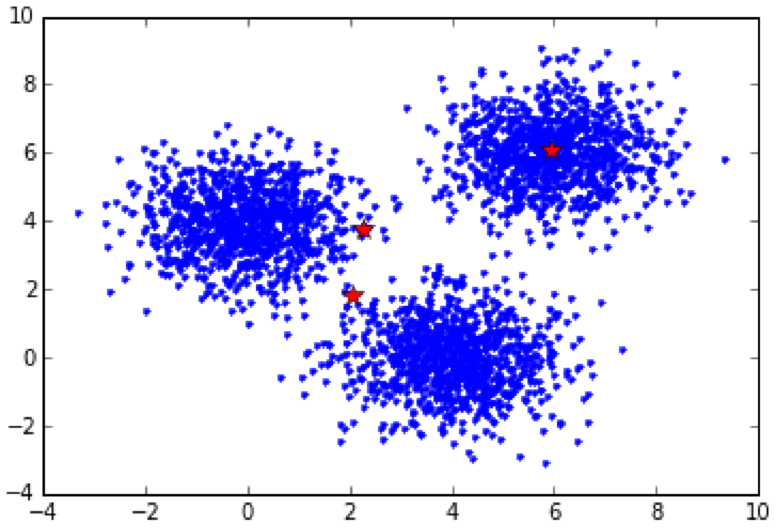
```
Iteration2
[[2.131596475369682, 1.661140557555178], [5.4837768644688385, 6.04
9774821653279], [2.910295021840567, 3.7601323152738266]]
```



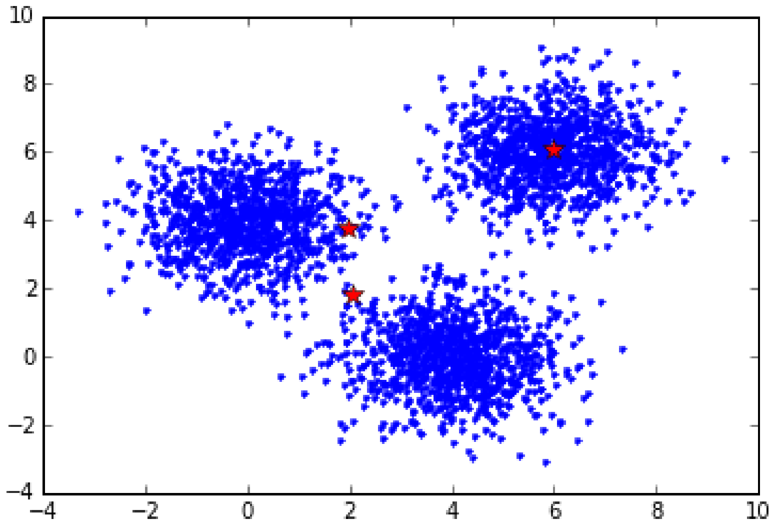
```
Iteration3
[[2.083253366788591, 1.7677842578288032], [5.785236540853202, 6.07
2051510371967], [2.6098113769859705, 3.800229802771744]]
```



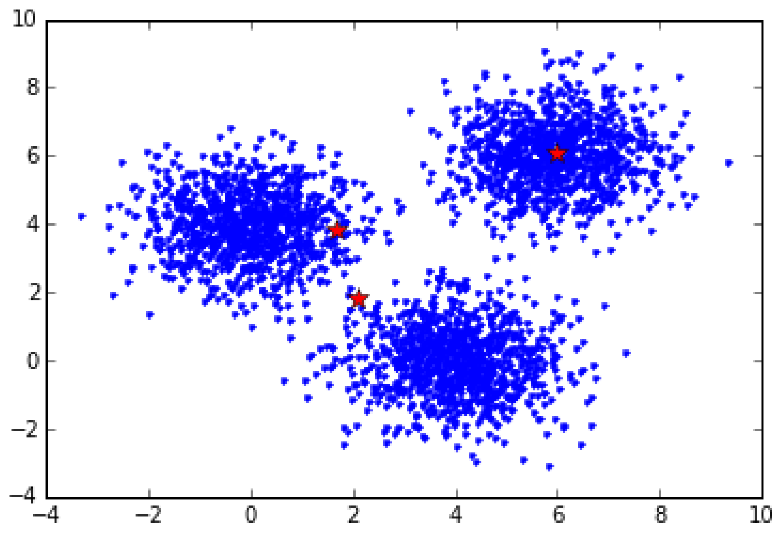
```
Iteration4
[[2.0675741991606476, 1.8259298892325253], [5.939818996154481, 6.0
87072107579678], [2.2711420860537626, 3.777028136097353]]
```



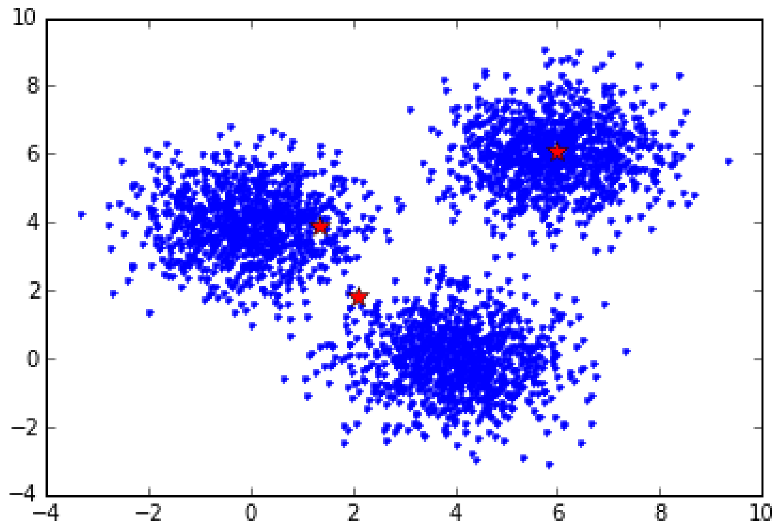
```
Iteration5
[[2.070488688634239, 1.843955946566191], [5.968825428781317, 6.083
479494842308], [1.9670284650977687, 3.770791187955294]]
```



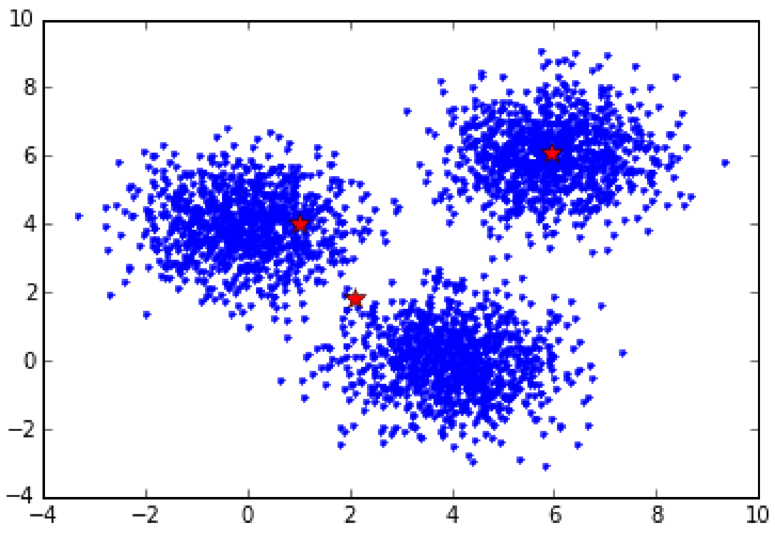
```
Iteration6
[[2.078349939360448, 1.846450505677145], [5.970849287961664, 6.076
809575640579], [1.67219677025096, 3.8066599126057308]]
```



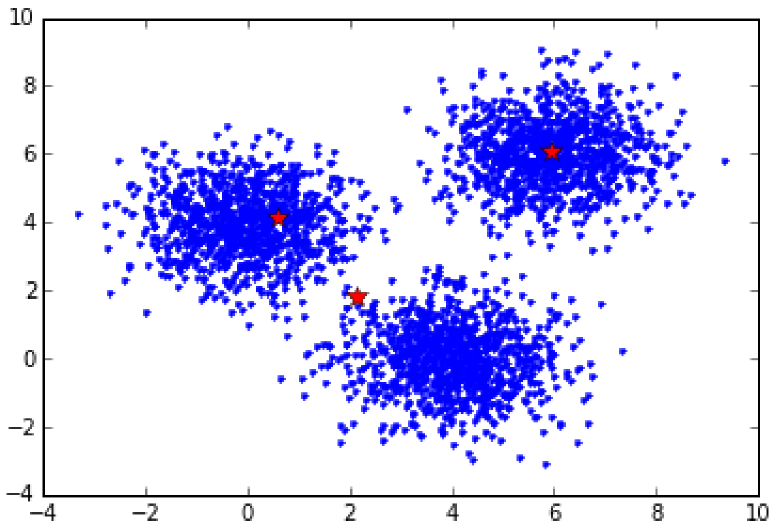
```
Iteration7
[[2.0914902525132035, 1.8401548826022853], [5.96764360650507, 6.07
0057580615657], [1.3485665363784753, 3.8801736210234155]]
```



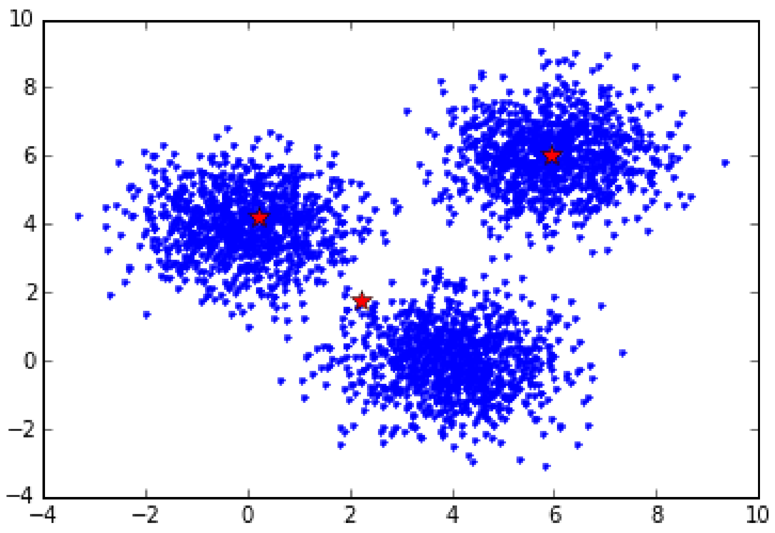
```
Iteration8
[[2.112773501397907, 1.825210733070598], [5.963895525249326, 6.063
678183547147], [0.9889553422116729, 3.989698222521716]]
```



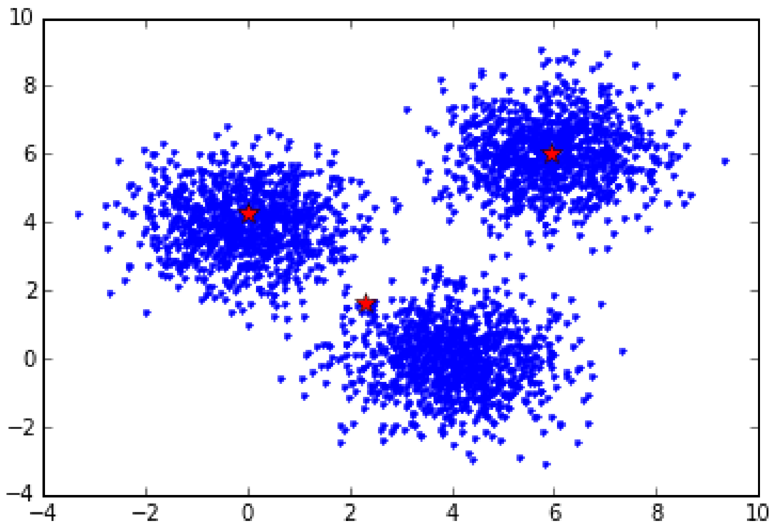
```
Iteration9
[[2.14735878252047, 1.7975726630999755], [5.960626813537449, 6.057
361247156723], [0.5993309768504114, 4.119984527444593]]
```



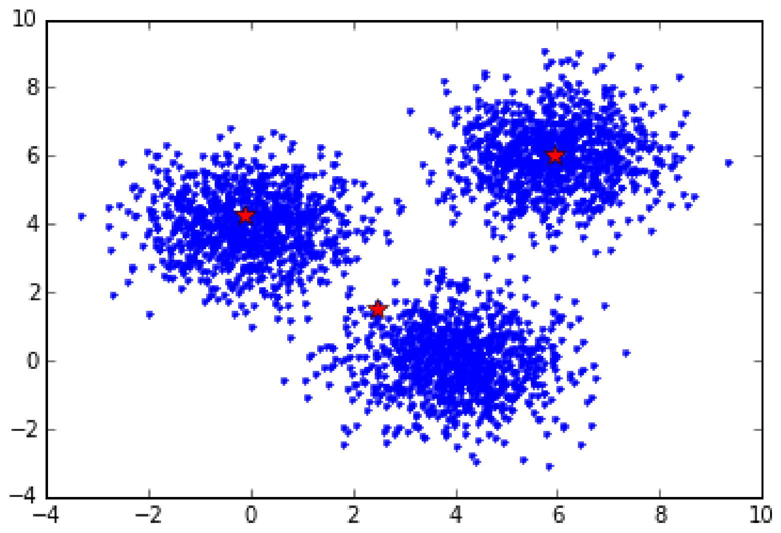
```
Iteration10
[[2.205510792193527, 1.7458071884196904], [5.957383931336121, 6.05
0174075392001], [0.22671612653934622, 4.228041979744838]]
```



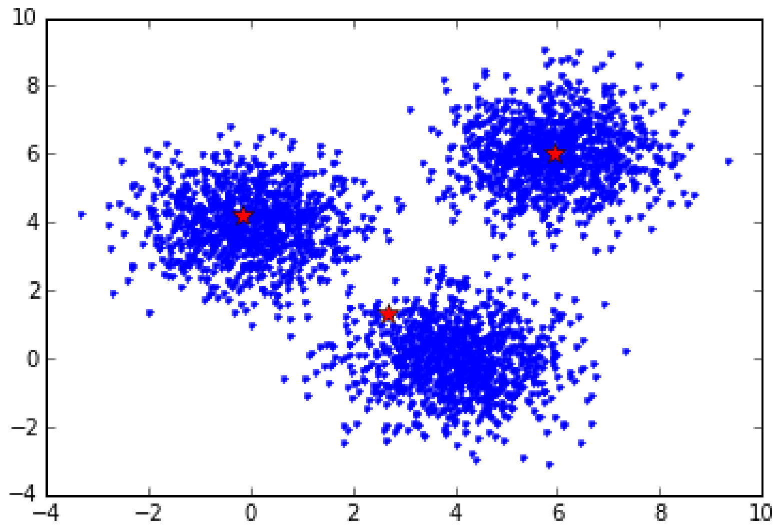
```
Iteration11
[[2.3083159323802636, 1.6437744891035384], [5.951885837953377, 6.0
4255126121449], [-0.014567079636599008, 4.259462181090834]]
```



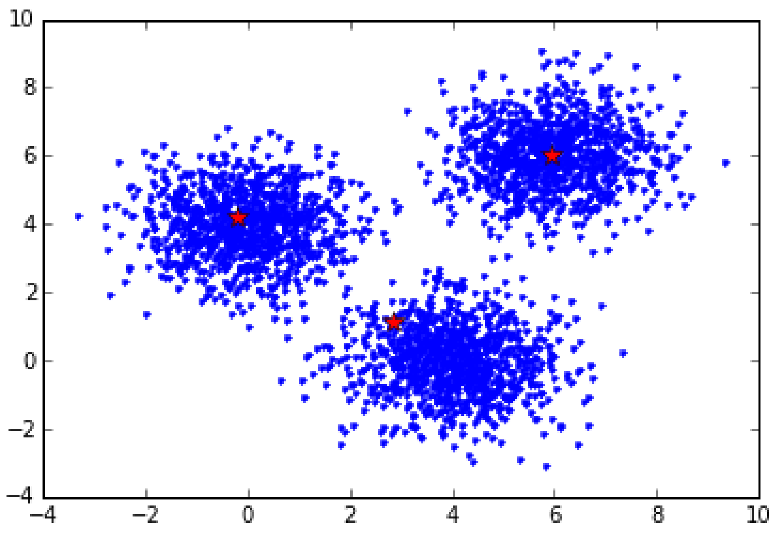
```
Iteration12
[[2.4735498717000888, 1.4783817464263793], [5.945961540159406, 6.0
387412836675125], [-0.12030991620243672, 4.2465128947898645]]
```



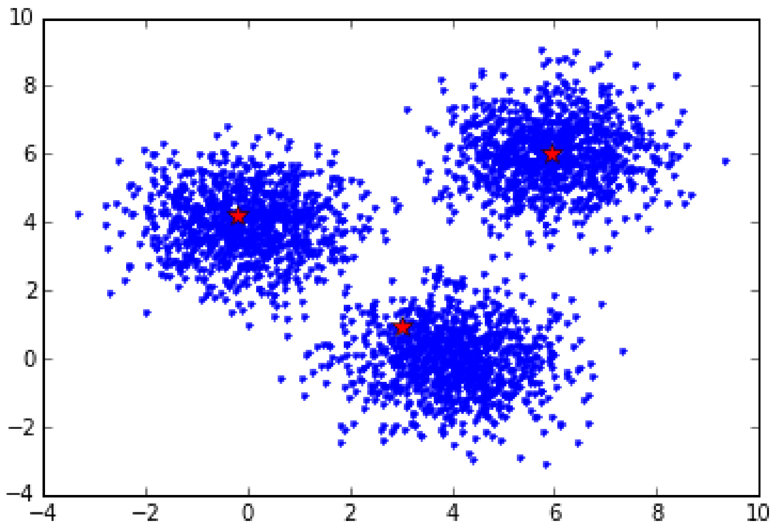
```
Iteration13
[[2.6688437443708484, 1.2876839441073875], [5.943841179322851, 6.0
37861790822221], [-0.163638545665176, 4.2293281819005015]]
```



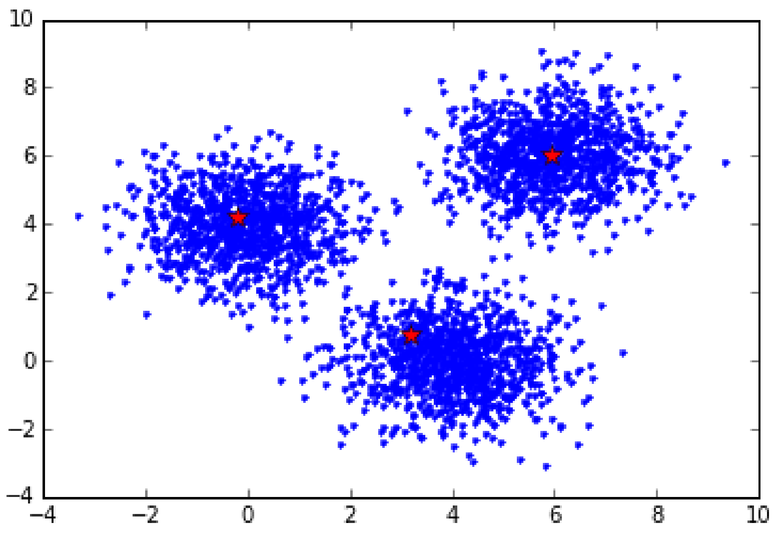
```
Iteration14
[[2.854789193084156, 1.1083590745481107], [5.943280585268021, 6.03
776550739148], [-0.18397010317963458, 4.215344328255406]]
```



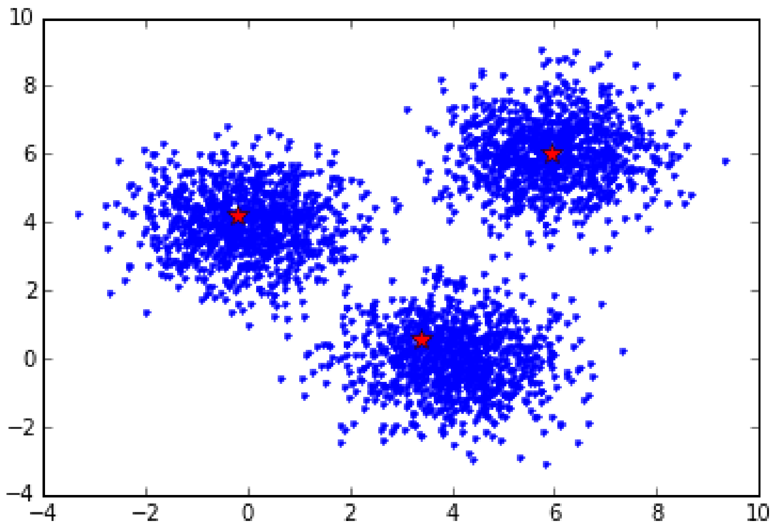
```
Iteration15
[[3.0282376070585277, 0.9416281734925245], [5.9430514696090695, 6.
037791114222845], [-0.1956802250357914, 4.205194022858775]]
```



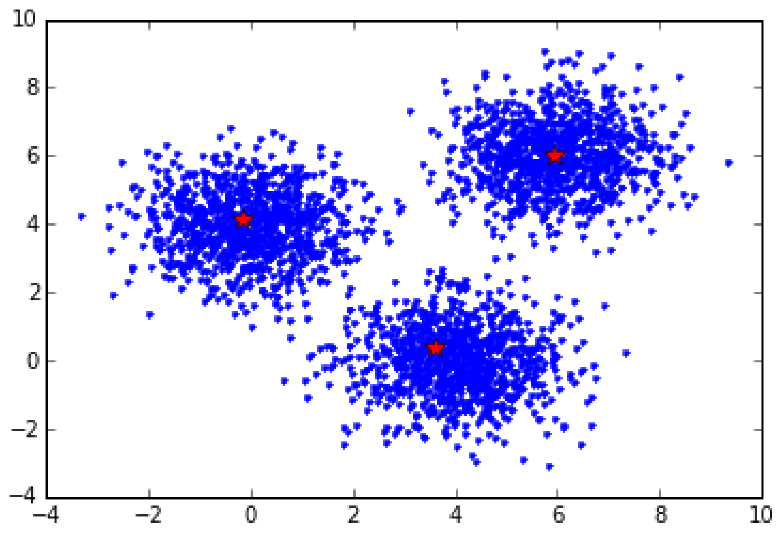
```
Iteration16
[[3.2032230424231827, 0.7732844351437782], [5.942869162241926, 6.0
37787721822767], [-0.2021925573106376, 4.19598643721064]]
```



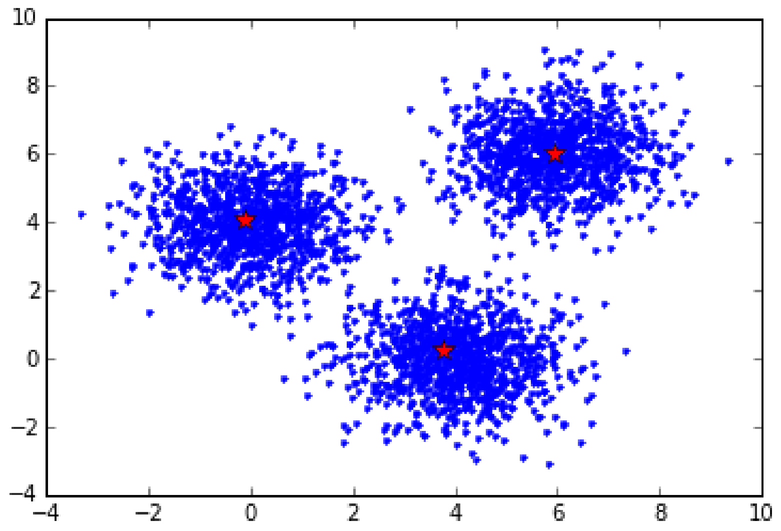
```
Iteration17
[[3.390262824645587, 0.5929055472428831], [5.942654666393197, 6.03
7739440968037], [-0.2002006741731183, 4.181988342004799]]
```



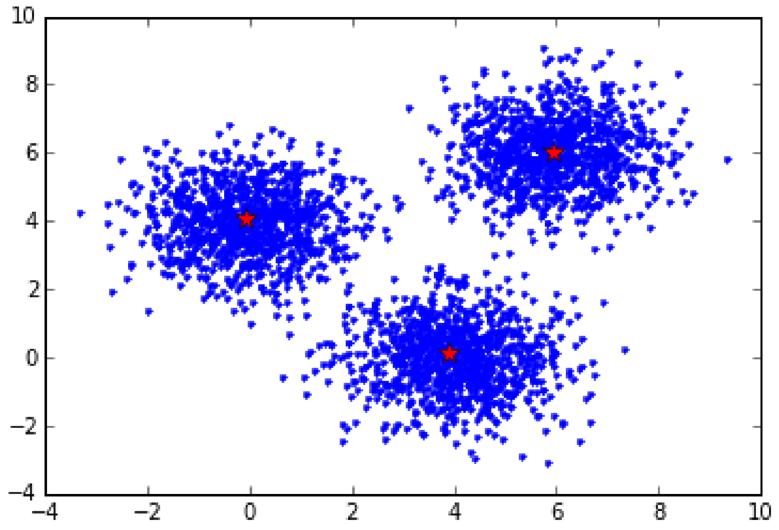
```
Iteration18
[[3.5840599725961924, 0.40521443878529934], [5.942328843446679, 6.
037618475394503], [-0.18031892271025274, 4.1535538815881345]]
```



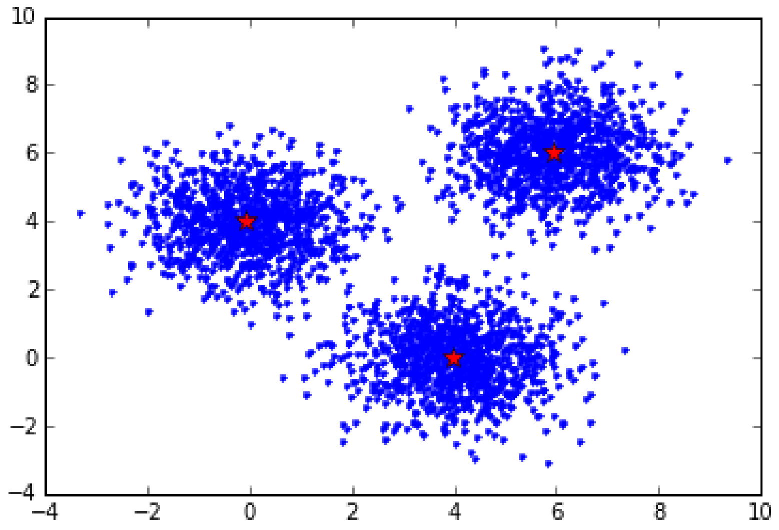
```
Iteration19
[[3.76219322155042, 0.2318737588833673], [5.941797441200421, 6.037
326560381442], [-0.13974667084368111, 4.107533898412953]]
```



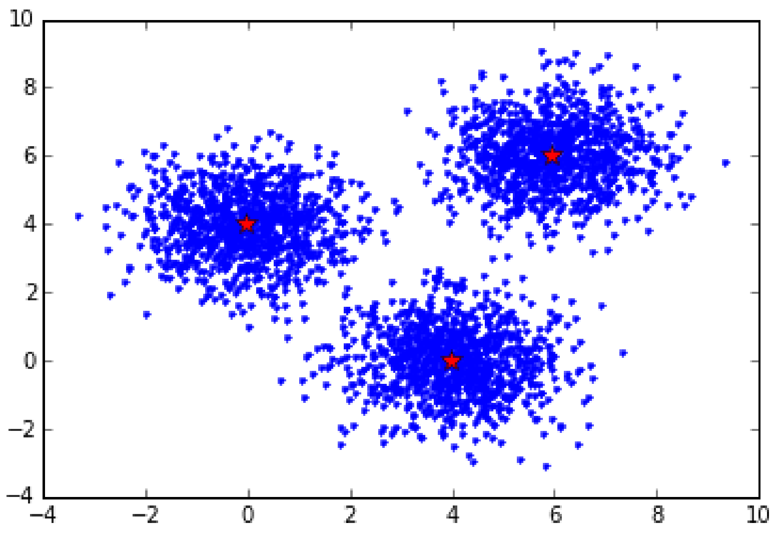
```
Iteration20
[[3.897401755241948, 0.10043079082977767], [5.941236119821501, 6.03
6838372536276], [-0.09395925422676217, 4.057884392126545]]
```



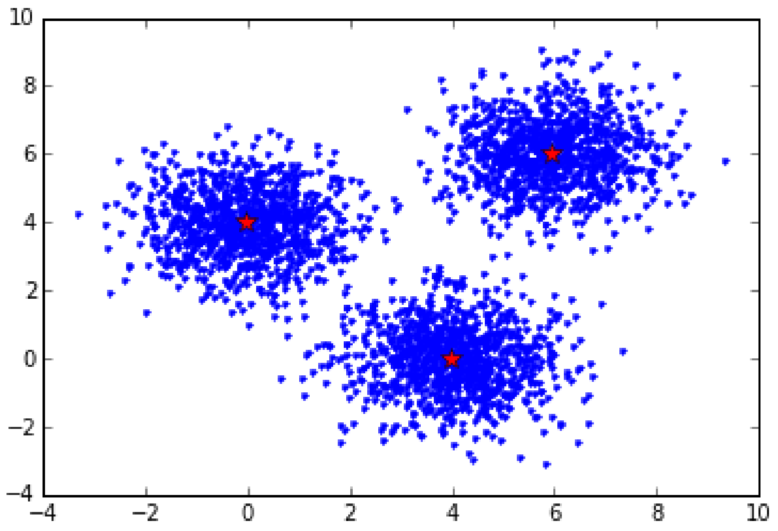
```
Iteration21
[[3.969042488719904, 0.033032650513515l9], [5.94104457470251, 6.03
6325607513039], [-0.06328446233358243, 4.0229130434661995]]
```



```
Iteration22
[[3.9908720565250375, 0.014350690359509782], [5.941045829858317,
6.035931826167468], [-0.0523985139712314, 4.008726451036319]]
```



```
Iteration23
[[3.9955784865857953, 0.010696308006288914], [5.94104856140416, 6.
035796925690169], [-0.049614568570987116, 4.0048343853125505]]
```



HW6.7 Implement Bernoulli Mixture Model via EM

Version 1

Implement the EM clustering algorithm to determine Bernoulli Mixture Model for discrete data in MRJob.

As a unit test:

As a test: use the same dataset from HW 4.5, the Tweet Dataset. Using this data, you will implement a 1000-dimensional EM-based Bernoulli Mixture Model algorithm in MrJob on the users by their 1000-dimensional word stripes/vectors using K = 4. Repeat this experiment using your KMeans MRJob implementation fron HW4. Report the rand index score using the class code as ground truth label for both algorithms and comment on your findings.

Here is some more information on the Tweet Dataset.

Here you will use a different dataset consisting of word-frequency distributions for 1,000 Twitter users. These Twitter users use language in very different ways, and were classified by hand according to the criteria:

0: Human, where only basic human-human communication is observed.

1: Cyborg, where language is primarily borrowed from other sources (e.g., jobs listings, classifieds postings, advertisements, etc...).

2: Robot, where language is formulaically derived from unrelated sources (e.g., weather/seismology, police/fire event logs, etc...).

3: Spammer, where language is replicated to high multiplicity (e.g., celebrity obsessions, personal promotion, etc...)

Check out the preprints of recent research, which spawned this dataset:

http://arxiv.org/abs/1505.04342 (http://arxiv.org/abs/1505.04342) http://arxiv.org/abs/1508.01843 (http://arxiv.org/abs/1508.01843)

The main data lie in the accompanying file:

topUsers_Apr-Jul_2014_1000-words.txt

and are of the form:

USERID,CODE,TOTAL,WORD1_COUNT,WORD2_COUNT,... . .

where

USERID = unique user identifier CODE = 0/1/2/3 class code TOTAL = sum of the word counts

Using this data, you will implement a 1000-dimensional K-means algorithm in MrJob on the users by their 1000-dimensional word stripes/vectors using several centroid initializations and values of K.

IR Book 16.3 Concept Code

This code implements the Bernoulli EM algorithm as described in the IR Book, example 16.3. This exercise is useful to ensure an understanding of the algorithm and to check functionality. There are a few bits that aren't explained will in the book but become apparent when working with this code.

- During the computation of the *[Math Processing Error]* classifications it is best to use log(probabilities) in order to prevent underflow
- The use of *[Math Processing Error]* as a smoothing parameter is crucial to the good behavior of the algorithm and to avoid either a divide by zero or a log(0) problem. In the IR book *[Math Processing Error]* is set to 0.0001. Setting to smaller values causes the algorithm to take more iterations to converge to the same solution as in the book.

In a Bernoulli Mixture Model a document is a vector of Booleans indicating the presence of a term.

The conditional probability of a document given a set of parameters is given by:
[Math Processing Error]

This is the sum for each class of the product of the probabilities of the terms in a document with 1 minus the probabilities of the terms not in the document.

The probability that a document from cluster *[Math Processing Error]* containts term *[Math Processing Error]* is given by:
[Math Processing Error]

The prior *[Math Processing Error]* of cluster *[Math Processing Error]* is the probability document *[Math Processing Error]* is in *[Math Processing Error]* if we have no other information about it.

When we don't know the classifications of the documents we can use Expectation Maximization iteratively to arrive at classifications, *[Math Processing Error]*.

E Step
[Math Processing Error]

The actual computation as implemented by taking the sum of the log probabilities as opposed to the products of the probabilities themselves. This is necessary because once you are dealing with many terms, multiplying a lot of small numbers results in numeric underflow. Also, in order to prevent taking the log(0), which will happen in the first iteration, a very small number, *[Math Processing Error]*, is added to the probability before taking the log. So we end up with:
[Math Processing Error]

In the code below, a single pass is made to calculate the *[Math Processing Error]* class soft assignments so that the terms are calculated only once so it may not be obvious what's going on.

*Note: the [Math Processing Error] values are important for the algorithm to get past the first iteration. If you take a straight calculation of the $r*{1,1}*[Math Processing Error]*\epsilon$ term this is avoided and you'll see that the result is very close to 1_

M Step

With smoothing *[Math Processing Error]* the M-Step equation becomes:
[Math Processing Error]

[Math Processing Error] if term is an element of document n and 0 otherwise.

[Math Processing Error] is the smoothing factor, set to 0.00001 as in example 16.3 in the IR book.

Finally, priors are updated per iteration as:
[Math Processing Error]

In [36]:
```
# save this to a file - it will be handy for MR testing later
with open('test.txt','w') as outfile:
    outfile.write('hot chocolate cocoa beans\n')
    outfile.write('cocoa ghana africa\n')
    outfile.write('beans harvest ghana\n')
    outfile.write('cocoa butter\n')
    outfile.write('butter truffles\n')
    outfile.write('sweet chocolate\n')
    outfile.write('sweet sugar\n')
    outfile.write('sugar cane brazil\n')
    outfile.write('sweet sugar beet\n')
    outfile.write('sweet cake icing\n')
    outfile.write('cake black forest\n')
```

In [37]:

```
import re
from math import log
import numpy as np

# read the documents. Each document consists of a list of words.
documents = []
with open('test.txt', 'r') as docfile:
    for line in docfile.readlines():
        documents.append(re.split(' ', line.strip()))

classes = 2
r = [[None] * len(documents), [None] * len(documents)]

# set our initial conditions (r_6,1 = 1.0 and r_7,1 = 0.0 and the c
onverse for the other class)
r[0][5] = r[1][6] = 1.0
r[0][6] = r[1][5] = 0.0

# initialize the priors
alpha = [0.0] * classes

# delta is to keep the arithmetic well behaved - it is not the smoo
thing factor
delta = 1.0E-12

# epsilon is for smoothing in Equation 2 and 3 (16.16 in the IR Boo
k)
epsilon = 0.0001

# conditional term probabilities
qm = {}

# compute alphas - Equation 3
def compute_alphas(alphas):
    for k in range(classes):
        alphas[k] = sum([x for x in r[k] if x is not None])/len([x
for x in r[k] if x is not None])


# compute inverted postings list
# this is handy for the computation of the qm's
def compute_postings():
    postings = {}
    for i in range(len(documents)):
        if r[0][i] is not None:
            for word in documents[i]:
                if word not in postings:
                    postings[word] = [i]
                else:
                    if i not in postings[word]:
                        postings[word].append(i)
    return postings

# compute qm's - Equation 2
```

```
def compute_next_qms(qm):
    for k in range(classes):
        for i in range(len(r[k])):
            if r[k][i] is not None:
                for word in documents[i]:
                    # use smoothing value epsilon
                    if word not in qm:
                        qm[word] = {k: sum([r[k][j]+epsilon for j i
n postings[word]]) / \
                                    sum([x+epsilon for x in r[k] if
x is not None])}
                    else:
                        qm[word][k] = sum([r[k][j]+epsilon for j in
postings[word]]) / \
                                    sum([x+epsilon for x in r[k] if
x is not None])


# compute next iteration of r's - Equation 1a
# note - need to do log(probability)
def compute_next_r(rs):
    for i in range(len(documents)):
        vocab_words_in_doc = []
        p = np.zeros((classes, 2))

        # find all vocab words in the doc. Note: there may be none.
        for word in documents[i]:
            if word in qm:
                vocab_words_in_doc.append(word)
                for k in range(classes):
                    # prevent math errors by not taking log(0)
                    p[k][0] += np.log(qm[word][k] + delta)
        if len(vocab_words_in_doc) > 0:
            # find all vocab words not in doc for all classes at th
e same time
            for word in qm:
                if word not in vocab_words_in_doc:
                    for k in range(classes):
                        # prevent math errors by not taking log(0)
                        p[k][1] += np.log(1-qm[word][k] + delta)
            # compute the denominator of Equation 1a for all classe
s
            denom = 0.0
            for k in range(classes):
                denom += alpha[k]*np.exp(p[k][0]+p[k][1])

            # compute the new r of Equation 1a for all classes
            for k in range(classes):
                rs[k][i] = alpha[k]*np.exp(p[k][0]+p[k][1])/denom

        else:
            # set to prior in case of no information
            for k in range(classes):
                rs[k][i] = alpha[k]
```

```
# iterate
for _ in range(25):
    compute_alphas(alpha)
    postings = compute_postings()
    compute_next_qms(qm)
    compute_next_r(r)
```

In [38]:
```
# if you take out rounding you can see that these values are not ex
act
print 'Soft classifications\n',np.around(r, 2)
print '\nPriors (alpha)',np.around(alpha,2)
```

```
Soft classifications
[[ 1.  1.  1.  1.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.  1.]]

Priors (alpha) [ 0.45  0.55]
```

Bernoulli Mixture Model MapReduce Implementation

- Initialize the classification matrix and the priors
- Implement the EM algorithm as MapReduce iterations

The first stage MR creates the data structures and initializes them. The first data structure is the soft classification, *[Math Processing Error]*. This is represented by r and has an entry per document and per class. The entry for the document is a list of indexes of the word count positions that have a value > 0 in the input file. The second data structure is the postings of which terms are in which document. The postings are nice to have when computing the iterations. The postings are computed using map reduce, the r document structure is built as a dictionary of lists where the doc id is the key.

In [88]:

```
%%writefile MRJob_BernoulliMixtureInit.py
from mrjob.job import MRJob, MRStep
import json
import re

# This MrJob calculates the gradient of the entire training set
#     Mapper: calculate partial gradient for each example
#
class MRJob_BernoulliMixtureInit(MRJob):
    DEFAULT_PROTOCOL = 'json'

    def __init__(self, *args, **kwargs):
        super(MRJob_BernoulliMixtureInit, self).__init__(*args, **k
wargs)

        self.count = 0

    def configure_options(self):
        super(MRJob_BernoulliMixtureInit, self).configure_options()
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt
is stored')

    # mapper init creates the initial r_n,k store
    def map_postings_init(self):
        self.r = {}

    # mapper to read a line from document file and emit binary post
ings data
    # we don't care about term frequencies because this is bernoull
i, so
    # if a word count is > 0 emit True otherwise False
    def map_postings(self, _, line):
        value_list = re.split(',', line.strip())
        doc = value_list[0]
        # add entry to the r dictionary for this document
        # initialize soft assignments to 1/k

        r_class = [1.0/float(self.options.k)]*4
        self.r[doc] = {'class' : r_class}
        self.r[doc]['term_idx'] = []

        # words appear starting in position 3 (from 0)
        for i in range(3,len(value_list)):
            # emit the position(word), id(doc) as key, value
            if int(value_list[i]) > 0:
                # if this term count > 0 then add term position to
doc index
                self.r[doc]['term_idx'].append(i-3)
                yield i-3, int(value_list[0])
    def map_postings_final(self):
        with open(self.options.pathName+'r.txt','w') as r_file:
            for doc in self.r:
                priors = '\t'.join(str(x) for x in self.r[doc]['cla
ss'])
                r_file.write('{0}\t{1}\n'.format(doc, priors))

        # write out a different format of the documents that makes
it a
        # bit easier to perform the EM computations
        with open(self.options.pathName+'document_vectors.txt','w')
as vec_file:
            for doc in self.r:
                terms  = '\t'.join(str(x) for x in self.r[doc]['ter
m_idx'])
                vec_file.write('{0}\t{1}\n'.format(doc, terms))

    # postings reducer initialization of the postings dictionary
    def reduce_postings_init(self):
        self.postings = {}

    # reduce postings list
    def reduce_postings(self, term_idx, docs):
        if term_idx not in self.postings:
            self.postings[term_idx] = []
        for doc in docs:
            if doc not in self.postings[term_idx]:
                self.postings[term_idx].append(doc)

    # reducer final is to write the postings list to disk
    def reduce_postings_final(self):
        with open(self.options.pathName+'postings.txt','w') as post
ings_file:
            for word in self.postings:
                p = '\t'.join([str(x) for x in self.postings[wor
d]])
                postings_file.write('{0}\t{1}\n'.format(word,p))

    def steps(self):
        return [MRStep(mapper_init=self.map_postings_init,
                       mapper=self.map_postings,
                       mapper_final=self.map_postings_final,
                       reducer_init=self.reduce_postings_init,
                       reducer=self.reduce_postings,
                       reducer_final=self.reduce_postings_final,
                       jobconf = {
                        'mapred.map.tasks' : 1,
                        'mapred.reduce.tasks' : 1
                     })]


if __name__ == '__main__':
    MRJob_BernoulliMixtureInit.run()
```

```
Overwriting MRJob_BernoulliMixtureInit.py
```

```
In [89]: !python MRJob_BernoulliMixtureInit.py --pathName '/Users/rcordell/D
ocuments/MIDS/W261/week06/HW6/' \
  --strict-protocols \
  --quiet \
  --k 4 \
  -r local 'topUsers_test.txt'
```

This MR code is where the EM calculations are made and the intermediate results saved to disk.

In [193]:

```
%%writefile MRJob_BernoulliMixtureModelQM.py
from mrjob.job import MRJob, MRStep
import json
import re

# This MrJob calculates the gradient of the entire training set
#     Mapper: calculate partial gradient for each example
#
class MRJob_BernoulliMixtureModelQM(MRJob):
    DEFAULT_PROTOCOL = 'json'

    def configure_options(self):
        super(MRJob_BernoulliMixtureModelQM, self).configure_option
s()
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt
is stored')

    # load up the postings list
    def qm_mapper_init(self):
        self.postings = {}
        with open(self.options.pathName+'postings.txt','r') as post
ings_file:
            for line in postings_file.readlines():
                p = re.split('\t',line.strip())
                word = int(p[0].strip())
                if word not in self.postings:
                    self.postings[word] = []
                for doc in p[1:]:
                    self.postings[word].append(int(doc.strip()))
        # load the r_nk values
        self.r = {}
        with open(self.options.pathName+'r.txt','r') as r_file:
            for line in r_file.readlines():
                vals = re.split('\t', line.strip())
                doc = int(vals[0].strip())
                self.r[doc] = [float(x.strip()) for x in vals[1:]]

    # examine the document term index and emit word, (r_nk, class)
    def qm_mapper(self, _, line):
        r_line = re.split('\t', line.strip())
        doc = int(r_line[0].strip())
        # word index vector
        words = [int(x.strip()) for x in r_line[1:]]
        for word in words:
            #the word should be in the postings list; this is
            #just defensive programming
            if word in self.postings:
                # emit all the document r_nk that have that word
                for document in self.postings[word]:
                    for k in range(len(self.r[document])):
                        yield (word,k), self.r[document][k]

    def qm_mapper_final(self):
        alpha = [0.0]*4
        for k in range(self.options.k):
            for doc in self.r:
                alpha[k] += self.r[doc][k]
        with open(self.options.pathName+'alpha.txt','w') as alpha_f
ile:
            for k in alpha:
                alpha_file.write('{0}\t'.format(k/len(self.r)))



    # the driver should intercept the output and write the results
to
    # the qm.txt file
    def qm_reduce(self, word_class, r_nk):
        yield (word_class), sum(r_nk)

    def steps(self):
        return [MRStep(mapper_init=self.qm_mapper_init,
                       mapper=self.qm_mapper,
                       mapper_final=self.qm_mapper_final,
                       reducer=self.qm_reduce,
                       jobconf = {
                            'stream.num.map.output.key.fields': 2,
                            'mapred.text.key.partitioner.options':
'-k1n,2n',
                            'mapred.text.key.comparator.options':
'-k2,2n'
                        }
                    )
                ]

if __name__ == '__main__':
    MRJob_BernoulliMixtureModelQM.run()
```

```
Overwriting MRJob_BernoulliMixtureModelQM.py
```

In [ ]:

```
!python MRJob_BernoulliMixtureModelQM.py --pathName '/Users/rcordel
l/Documents/MIDS/W261/week06/HW6/' \
  --strict-protocols \
  --quiet \
  --k 4 \
  -r local 'document_vectors.txt'
```

In [136]:

```
%%writefile MRJob_BernoulliMixtureModelRnk.py
from mrjob.job import MRJob, MRStep
import json
import re
import numpy as np

# This MrJob calculates the gradient of the entire training set
#     Mapper: calculate partial gradient for each example
#
class MRJob_BernoulliMixtureModelRnk(MRJob):
    DEFAULT_PROTOCOL = 'json'


    def configure_options(self):
        super(MRJob_BernoulliMixtureModelRnk, self).configure_optio
ns()
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt
is stored')


    # read in the qm and alpha values
    def rnk_mapper_init(self):
        self.qm = {}
        with open(self.options.pathName+'qm.txt','r') as qm_file:
            for line in qm_file.readlines():
                vals = re.split('\t',line.strip())
                self.qm[int(vals[0])]=[float(x.strip()) for x in va
ls[1:]]
        self.delta = 1E-12

    # for this we want the document word vector file
    def rnk_mapper(self, _, line):
        r_line = re.split('\t', line.strip())
        doc = int(r_line[0].strip())
        words = [int(x.strip()) for x in r_line[1:]]
        for i in range(max(self.qm.keys())):
            if i in words:
                if i in self.qm:
                    for k in range(len(self.qm[i])):
                        yield (doc,k), (np.log(self.qm[i][k]+1E-1
2))
                else:
                    for k in range(len(self.qm[i])):
                        yield (doc,k), (np.log(1.0-self.qm[i][k]+1E
-12))


    # the reducer outputs the sum of the log probabilities for each
term for each document for each class
```

```
    def rnk_reducer(self, doc_class, score):
        yield doc_class, sum(score)

    # load up the alpha values in preparation to calculate the new
classifications
    def mapper_init(self):
        self.alpha = []
        with open(self.options.pathName+'alpha.txt','r') as alpha_f
ile:
            for line in alpha_file.readlines():
                self.alpha = [(float(x.strip())) for x in re.split
('\t',line.strip())]
        self.r = {}

    def mapper(self, doc_class, log_prob):
        if doc_class[0] not in self.r:
            self.r[doc_class[0]] = []
        self.r[doc_class[0]].append(log_prob)

    def mapper_final(self):
        for doc in self.r:
            print doc, r[doc]


    def reducer(self, doc_class, log_prob):
        pass

        # now we have all the log probabilities for every class in
every document
#        for doc in r:
#            a = [np.exp(x) for x in r[doc]]
#            denom = 0.0
#            for i in range(len(a)):
#                a[i] = self.alpha[i]*a[i]
#                denom += a[i]
#            for k in range(len(r[doc])):
#                r[doc][k] = a[k]/denom
#                yield (doc,k), r[doc][k]

    def mapper_final(self):
        pass

    def steps(self):
        return [MRStep(mapper_init=self.rnk_mapper_init,
                       mapper=self.rnk_mapper,
                       reducer=self.rnk_reducer,
                       jobconf = {
                            'stream.num.map.output.key.fields': 2,
                            'mapred.text.key.partitioner.options':
'-k1n,2n',
                            'mapred.text.key.comparator.options':
'-k2,2n'
                        }),
                MRStep(mapper_init=self.mapper_init,
                              mapper=self.mapper,
                              mapper_final=self.mapper_final,
#                              reducer_init=self.reducer_init,
#                              reducer=self.reducer,
                              jobconf = {
                                   'mapred.map.tasks' : 1,
                                   'mapred.reduce.tasks' : 1
                                   }
                           )
                   ]


if __name__ == '__main__':
    MRJob_BernoulliMixtureModelRnk.run()
```

```
Overwriting MRJob_BernoulliMixtureModelRnk.py
```

In [137]:

```
!python MRJob_BernoulliMixtureModelRnk.py --pathName '/Users/rcordell/Documents/MIDS/W261/week06/HW6/' \
  --strict-protocols \
  --quiet \
  --k 4 \
  -r local 'document_vectors.txt'
```

This is the driver for the MRJob steps that invokes the initialization MR job and then iteratively drives the EM MR job.

```
In [ ]:
```

```
from mrjob.job import MRJob
from MRJob_BernoulliMixtureInit import MRJob_BernoulliMixtureInit
from MRJob_BernoulliMixtureModelQM import MRJob_BernoulliMixtureMod
elQM
from MRJob_BernoulliMixtureModelRnk import MRJob_BernoulliMixtureMo
delRnk
import json

# need to make this an argument passed to the program
filePath = 'topUsers_test.txt'
doc_vectors = 'document_vectors.txt'
qm_file = 'qm.txt'
pathName = '/Users/rcordell/Documents/MIDS/W261/week06/HW6/'

# initialize documents and postings
mrJob = MRJob_BernoulliMixtureInit(args=[filePath,
                                        '--k','4',
                                        '--pathName',pathName,
                                        '--strict-protocols'])

# after running this MRJob we will have an initialized r_n,k and po
stings
with mrJob.make_runner() as runner:
    runner.run()

#Begin iteration on change in centroids
for iteration in range(1):
    print "Iteration" + str(iteration+1)

    # run one iteration
    mrJob_M = MRJob_BernoulliMixtureModelQM(args=[doc_vectors,
                                            '--k','4',
                                            '--pathName',pathName,
                                            '--strict-protocols'])
    with mrJob_M.make_runner() as runner:
        runner.run()
        qm = {}
        for line in runner.stream_output():
            key, value = mrJob_M.parse_output_line(line)
            if key[0] not in qm:
                qm[key[0]]=[]
            qm[key[0]].append(value)
        with open (pathName+qm_file,'w') as qmFile:
            for word in qm:
                qmFile.write('{0}\t{1}\n'.format(word, '\t'.join([s
tr(x) for x in qm[word]])))

    mrJob_E = MRJob_BernoulliMixtureModelQM(args=[doc_vectors,
                                            '--k','4',
                                            '--pathName',pathName,
                                            '--strict-protocols'])
    with mrJob2.make_runner() as runner:
        runner.run()
```

Question 6.7

Version 2

Implement the EM clustering algorithm to determine Bernoulli Mixture Model for discrete data in MRJob.

Solution:

We will accomplish this using a similar framework to the KMeans implementation in the last homework. This includes a `driver` file that manages the job, and subordinate task MRJob scripts that execute different portions of the algorithm. We will detail each portion below.

Driver

The driver will manage the workflow and algorithm iterations. In particular, it has wrappers for the MRJob scripts making extracting and handling results straightforward. The convergence criteria is a maximum-delta threshold where each iteration must produce a significant change in the number of elements assigned to a particular class. This is set to 10% end-over-end in the driver.

In [1]:

```
%%writefile driver.py
from __future__ import division

from mrjob.job import MRJob
from mrjob.step import MRStep

from init_alphas import initAlphas
from conditional_list import conditionalList
from bernoulli import Bernoulli

import cPickle as pickle
from collections import defaultdict
import pdb

# Storage files
centroidFile = './.clusters'
scoreFile = './.scores'
trackerFile = './.tracker'
dataFile = 'topUsers_Apr-Jul_2014_1000-words.txt'
continueFile = './.continueFile'


def getName(obj, namespace):
        return [name for name in namespace if namespace[name] is ob
j]


def extractValues(job, runner):
        output = defaultdict(int)
        for line in runner.stream_output():
                key, value = job.parse_output_line(line)
                output[key] = value

        return output


def dumpToFile(variable, filename):
        with open(filename, 'w') as f:
                pickle.dump(variable, f)


def dumpToTracker(variable, filename):
        with open(filename, 'a') as f:
                f.write('dumping...' + '\n')
                f.write(str(variable) + '\n')
                f.write('dump complete.' + '\n')


def runJob(method, args, dFile=None):
        job = method(args=args)

        methodName = getName(method, globals())[0]
        print '\n\t' + 'Running ' + methodName + '...'
        with job.make_runner() as runner:

                # Start
                runner.run()

                result = extractValues(job, runner)
                dumpToTracker(result, trackerFile)

                print '\t' + 'Complete: ' + methodName

                if dFile:
                        dumpToFile(result, dFile)

                else:
                        return result


if __name__ == '__main__':

        # Clear files
        open(centroidFile, 'w').close()
        open(scoreFile, 'w').close()
        open(trackerFile, 'w').close()
        open(continueFile, 'w').close()


        print '\n' + 'Iteration Setup.' + '\n'
        print '-------------'


        # Step 1: Create initial clusters
        runJob(initAlphas, args=[dataFile, '--clusters=' + centroid
File])

        # Step 2: Create priors
        runJob(conditionalList, args=[dataFile, '--clusters=' + cen
troidFile], dFile=scoreFile)

        # Step 3: Apply Bernoulli
        runJob(Bernoulli, args=[dataFile, '--clusters=' + centroidF
ile, '--scores=' + scoreFile, \
                '--continuing=' + continueFile])


        # Loop initializations
        priorError = 1
        threshold = 0.1
        maxIter = 10

        centroidArg = '--clusters=' + centroidFile
        scoreArg = '--scores=' + scoreFile
        continueArg = '--continuing=' + continueFile
        # Pre
        priorDist = defaultdict(int)
        with open(dataFile, 'r') as f:
                for line in f.readlines():

                        line = [int(x) for x in line.split(',')]
                        label = line[1]

                        priorDist[label] += 1


        for i in range(maxIter):

                """ This loop is the iteration portion of the EM al
gorithm. """

                print '\n' + 'Iteration ' + str(i) + '.' + '\n'
                print '-------------'

                # Step 1: Create initial clusters
                runJob(initAlphas, args=[continueFile, centroidAr
g])

                # Step 2: Create priors
                runJob(conditionalList, args=[continueFile, centroi
dArg], dFile=scoreFile)

                # Step 3: Apply Bernoulli
                runJob(Bernoulli, args=[continueFile, centroidArg,
scoreArg, continueArg])

                # Step 4: Check thresholds

                print '\t' + 'Checking Delta.' + '\n'

                newDist = defaultdict(int)

                # Post
                with open(continueFile, 'r') as f:
                        for line in f.readlines():

                                line = [int(x) for x in line.split
(',')]
                                label = line[1]

                                newDist[label] += 1

                # Check
                maxDiff = 0
                for key in priorDist.keys():
                        delta = abs(newDist[key] - priorDist[key])
/ priorDist[key]
                        maxDiff = max(maxDiff, delta)
                printDiff = round(maxDiff, 3)
                print '\t' + 'Max Iteration Delta: ' + str(printDif
f)


                if maxDiff <= threshold:
                        break

                else:
                        priorDist = newDist


        print '\n' + 'Results' + '\n'
        print '-------------'

        priorDist = defaultdict(int)
        laterDist = defaultdict(int)

        with open(dataFile, 'r') as f:
                for line in f.readlines():

                        line = [int(x) for x in line.split(',')]
                        label = line[1]

                        priorDist[label] += 1

        with open(continueFile, 'r') as f:
                for line in f.readlines():

                        line = [int(x) for x in line.split(',')]
                        label = line[1]

                        laterDist[label] += 1

        for key in priorDist.keys():

                print '\n' + 'Class: ' + str(key)

                print '\t' + 'Initial: ' + str(priorDist[key])
                print '\t' + 'Final: ' + str(laterDist[key]) + '\n'
```

```
Writing driver.py
```

Initialization

Here we initialize the cluster centers using a simple prior calculation. The prior for each center is the proportion of elements that fall into that label.

In [2]:

```
%%writefile init_alphas.py
from __future__ import division

from mrjob.job import MRJob
from mrjob.step import MRStep

import numpy as np
from collections import defaultdict
import string
import random
import cPickle as pickle

class initAlphas(MRJob):

        """ This class executes the MRJob necessary for one iterati
on of gradient descent. """

        def configure_options(self):

                """ This function defines the arguments to the job.
"""

                super(initAlphas, self).configure_options()

                self.add_file_option('--clusters',
                        help='File to read the updated coefficient
s.')


        def load_options(self, args):

                """ This function initializes arguments defined in
'configure_options'. """

                super(initAlphas, self).load_options(args)

                if self.options.clusters is None:
                        self.option_parser.error('You must specify
a --clusters for Bernoulli EM.')

                self.clusters = self.options.clusters


        def mapper_get_labels(self, _, line):

                """ This computes the prior initialization. """

                # Parse data
                line = [int(x) for x in line.split(',')]
                label = line[1]

                yield label, 1
        def reducer_count_labels(self, label, counts):

                """ This function aggregates the total counts per l
abel. """

                yield None, (label, sum(counts))


        def reducer_prior_labels(self, _, labelCount):

                """ This function calculates the priors for each la
bel. """

                # Assign
                aggregatedLabel = defaultdict(int)
                priorLabel = defaultdict(int)

                for k, v in labelCount:
                        aggregatedLabel[k] += v


                # Write
                with open(self.clusters, 'w') as f:

                        pickle.dump(aggregatedLabel, f)

                        # Normalize
                        total = sum(aggregatedLabel.values())
                        for k, v in aggregatedLabel.iteritems():
                                priorLabel[k] = v / total


                        # Write prior
                        pickle.dump(priorLabel, f)


        def steps(self):

                """ This defines the run-order for the MapReduce jo
b. """

                return [MRStep(mapper=self.mapper_get_labels,
                                                reducer=self.reduce
r_count_labels),

                                MRStep(reducer=self.reducer_prior_l
abels)]


if __name__ == '__main__':
        initAlphas.run()
```

```
Writing init_alphas.py
```

Conditional Probabilities

Next, we compute the conditional probability for each dimension and each class. The results are initialized with the priors, which is the iterating mechanism used in this algorithm.

In [3]:

```
%%writefile conditional_list.py
from __future__ import division

from mrjob.job import MRJob
from mrjob.step import MRStep

import numpy as np
from collections import defaultdict
import string
import random
import cPickle as pickle

class conditionalList(MRJob):

        """ This class executes the MRJob necessary for one iterati
on of gradient descent. """

        def configure_options(self):

                """ This function defines the arguments to the job.
"""

                super(conditionalList, self).configure_options()

                self.add_file_option('--clusters',
                        help='File to read the updated coefficient
s.')


        def load_options(self, args):

                """ This function initializes arguments defined in
'configure_options'. """

                super(conditionalList, self).load_options(args)

                if self.options.clusters is None:
                        self.option_parser.error('You must specify
a --clusters for Bernoulli EM.')

                self.clusters = self.options.clusters

                # Get aggregates and prior
                with open(self.clusters, 'r') as f:

                        self.aggregatedLabel = pickle.load(f)


        def mapper_normalize(self, _, line):

                """ This computes a normalized vector for each elem
ent. """

                # Parse data
                line = [int(x) for x in line.split(',')]

                custID = line[0]
                label = line[1]
                body = line[3:]

                conditionalPrior = map(lambda dim: 1 if dim > 0 els
e 0, body)

                yield label, conditionalPrior


        def reducer_aggregate_conditionals(self, label, conditional
Lists):

                """ This function aggregates the conditional lists
for each label. """

                prior = self.aggregatedLabel[label]

                conditionalPriorAgg = [(sum(count) + 1) / (prior +
2) for count in zip(*conditionalLists)]

                yield label, conditionalPriorAgg


        def steps(self):

                """ This defines the run-order for the MapReduce jo
b. """

                return [MRStep(mapper=self.mapper_normalize,
                                                reducer=self.reduce
r_aggregate_conditionals)]


if __name__ == '__main__':
        conditionalList.run()
```

```
Writing conditional_list.py
```

Updating Labels

Finally, we update the labels given the Bernoulli criteron and conditional probabilities computed in the previous step.

In [6]:

```
%%writefile bernoulli.py
from __future__ import division

from mrjob.job import MRJob
from mrjob.step import MRStep

import numpy as np
from collections import defaultdict
import string
import random
import cPickle as pickle
import operator

class Bernoulli(MRJob):

        """ This class executes the MRJob necessary for one iterati
on of gradient descent. """

        def configure_options(self):

                """ This function defines the arguments to the job.
"""

                super(Bernoulli, self).configure_options()

                self.add_file_option('--clusters',
                        help='File to read the updated coefficient
s.')

                self.add_file_option('--scores',
                        help='File to read the updated prior probab
ilities.')

                self.add_file_option('--continuing',
                        help='File to write the updated data.')


        def load_options(self, args):

                """ This function initializes arguments defined in
'configure_options'. """

                super(Bernoulli, self).load_options(args)

                if self.options.clusters is None:
                        self.option_parser.error('You must specify
a --clusters for Bernoulli EM.')

                if self.options.scores is None:
                        self.option_parser.error('You must specify
a --scores for Bernoulli EM.')

                if self.options.continuing is None:
                        self.option_parser.error('You must specify
a --continuing for Bernoulli EM.')

                self.clusters = self.options.clusters
                self.scores = self.options.scores
                self.continuing = self.options.continuing

                # Get aggregates and prior
                with open(self.clusters, 'r') as f:

                        self.aggregatedLabel = pickle.load(f)
                        self.priorLabel = pickle.load(f)

                # Get conditionals
                with open(self.scores, 'r') as f:

                        self.priors = pickle.load(f)


        def mapper_normalize(self, _, line):

                """ Assign a label to each custID. """

                # Parse data
                line = [int(x) for x in line.split(',')]

                custID = line[0]
                label = line[1]
                body = line[3:]

                dimCheck = map(lambda dim: 1 if dim > 0 else 0, bod
y)

                # Score
                labelScore = defaultdict(int)
                for label, value in self.priorLabel.iteritems():

                        # Prior
                        labelScore[label] += np.log(self.priorLabel
[label])

                        # Conditional
                        for index, dim in enumerate(dimCheck):

                                if dim == 1:
                                        labelScore[label] += np.log
(self.priors[label][index])

                                else:
                                        labelScore[label] += np.log
(1 - self.priors[label][index])


                # ArgMax
                bestLabel = max(labelScore.iteritems(), key=operato
r.itemgetter(1))[0]

                yield None, (custID, bestLabel, body)


        def reducer_write(self, _, values):

                """ This clears the continue file and writes the da
ta. """

                open(self.continuing, 'w').close()

                # Write to new file
                with open(self.continuing, 'a') as f:

                        for valueTuple in values:

                                # Unpack
                                custID, bestLabel, body = valueTupl
e

                                # Write
                                bodyStr = ','.join([str(x) for x in
body])

                                lineStr = str(custID) + ',' + str(b
estLabel) + ',' + str(123) + ',' \
                                        + bodyStr + '\n'

                                f.write(lineStr)


        def steps(self):

                """ This defines the run-order for the MapReduce jo
b. """

                return [MRStep(mapper=self.mapper_normalize,
                                                reducer=self.reduce
r_write)]


if __name__ == '__main__':
        Bernoulli.run()
```

```
Overwriting bernoulli.py
```

Submit Process

Below we execute a run on the Twitter unit-test data using this algorithm.

In [7]:

```
!python driver.py
```

```
Iteration Setup.

-------------

        Running initAlphas...
No handlers could be found for logger "mrjob.runner"
        Complete: initAlphas

        Running conditionalList...
        Complete: conditionalList

        Running Bernoulli...
        Complete: Bernoulli

Iteration 0.

-------------

        Running initAlphas...
        Complete: initAlphas

        Running conditionalList...
        Complete: conditionalList

        Running Bernoulli...
        Complete: Bernoulli
        Checking Delta.

        Max Iteration Delta: 0.815

Iteration 1.

-------------

        Running initAlphas...
        Complete: initAlphas

        Running conditionalList...
        Complete: conditionalList

        Running Bernoulli...
        Complete: Bernoulli
        Checking Delta.

        Max Iteration Delta: 0.72

Iteration 2.

-------------

        Running initAlphas...
        Complete: initAlphas

        Running conditionalList...
        Complete: conditionalList

        Running Bernoulli...
        Complete: Bernoulli
        Checking Delta.

        Max Iteration Delta: 0.093

Results

-------------

Class: 0
        Initial: 752
        Final: 719


Class: 1
        Initial: 91
        Final: 47


Class: 2
        Initial: 54
        Final: 103


Class: 3
        Initial: 103
        Final: 131
```

In [ ]: