Homework 6

Alex Smith
June 16, 2016
MIDS261 - Machine Learning at Scale
Professor Shanahan
Due: June 24, 2016

Useful resources

The following resources were particularly useful.

- 6.7 Async
- Wikipedia article, Hessian matrix (https://en.wikipedia.org/wiki/Hessian_matrix)
- UWash Chapter on Optimality (https://www.math.washington.edu/~burke/crs/516/notes/ch1.pdf)
- Hessian Matrix (https://en.wikipedia.org/wiki/Hessian_matrix)

Libraries

The following libraries must be installed before running the below code. They can all be installed through Pip (https://github.com/pypa/pip).

- Scikit Learn (http://scikit-learn.org/stable/)
- Numpy (http://www.numpy.org/)
- Regular Expression (https://docs.python.org/2/library/re.html)
- Pretty Table (https://pypi.python.org/pypi/PrettyTable)
- Random (https://docs.python.org/2/library/random.html)
- Datetime (https://docs.python.org/2/library/datetime.html)
- Matplotlib (http://matplotlib.org/)

In [5]:
```
%%javascript
// get rid of right lines in MatJax formulas
$(".MathJax").find("nobr > span > span").css({"border-left-color": "#ee
e"})
```

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 "is the selection of a best element (with regard to some criteria) from some set of available alternatives" (Async 6.7). In optimization, we want to choose the maximum (or minimum) of some variable given some variables that we can manipulate. We can test the different levers and measure the outcome. I have a slightly *unorthodox* example from my work because inputs are not necessarily quantitative and the reaching the optimum is measured somewhat subjectively. However, I believe the principles are the same. I am trying to improve the responsivness of my department, as measured by the percentage of workloads that we complete same-day. We know that 100% is not ideal because there are some workloads that we should wait on so that we can do more research. Some of the levers we have at our disposal: training backfills for when people are out, retraining current employees, reporting on daily numbers, having management review work that we wait on, and simplifying decision-making processes. We are currently in the process of deploying this to the real world. For example, we have begun simplifying decision-making processes. In the past, employees used a plethora of outdated decision-trees and emails from previous management to guide decisions. We replaced all of this with a simple set of guiding principles and empowered employees to make good decisions. This reduced the time that employees spent on each workload. We also began manually reviewing all the word we were choosing to wait on. We would send back work to the employee if it was inappropriate to wait. We callibrate regular with key decision makers. Now, we see that our responsivness is likely too high because our customer outcomes on some transactions are less than ideal. To solve this, we are reviewing work that we should wait on and helping folks identify this work. In some ways this work is subjective, but it is in every sense about achieving the optimal level of same-day completion for work, balancing between getting back to customers quickly and doing all the work that is necessary to set a transaction on the correct path.

HW6.1 Optimization theory:

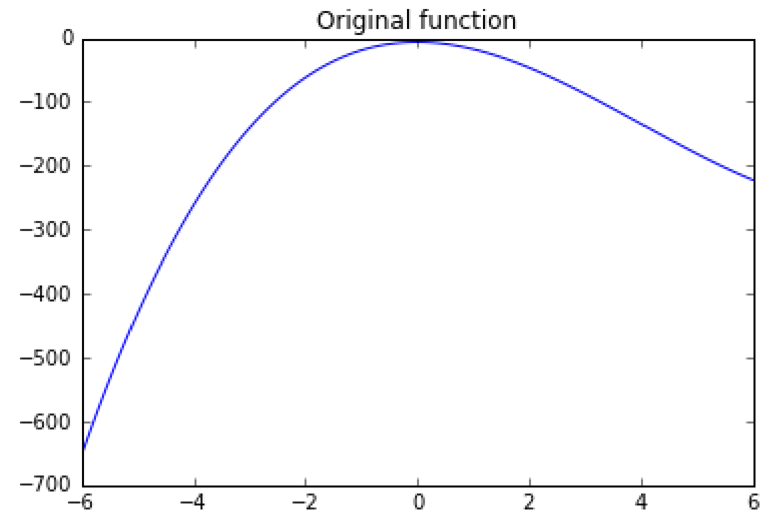
In python, plot the univartiate function:
X^3 -12x^2-6
defined over the real domain -6 to +6.

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.

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?

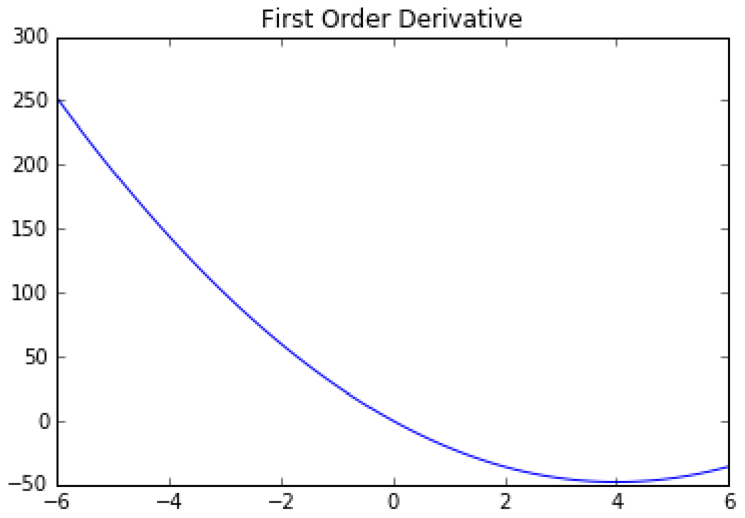
In [2]:
```
# create the function that we can use to
# generate the points for plotting
def sample_function(xlist):
    """function that takes a list of numbers
    and returns a list of y-values f"""


    # write the function
    def actual_function(x):
        y = (x**3) - (12 * (x**2)) - 6
        return y

    # create an array to store
    # the y-values
    ylist = []

    # loop through each x, generate
    # the corresponding y value and
    # append it to the array
    for x in xlist:
        y = actual_function(x)
        ylist.append(y)

    # return the y list
    return ylist
```

In [3]:
```
%matplotlib inline
# import numpy to help us generate our values
# import matplotlib to help us plot
import numpy as np
import matplotlib.pylab as plt

# generate our x-values
xlist = list(np.arange(-6,6.1,0.1))

# generate our y-value
ylist = sample_function(xlist)

# plot the graph
plt.plot(xlist,ylist)
plt.title("Original function")
plt.show()
```



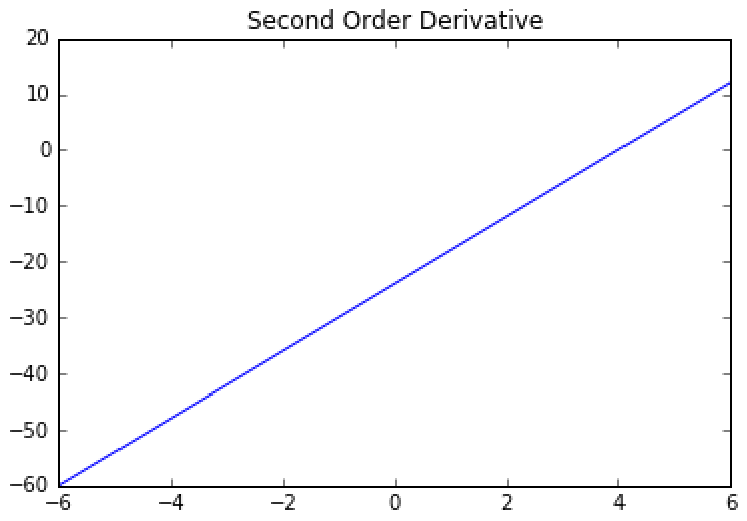
In [4]:
```
# create the first derivative of the function that
# we can use to generate the points for plotting
def sample_1stderivative(xlist):
    """function that takes a list of numbers
    and returns a list of y-values f"""


    # write the function
    def actual_function(x):
        y = (3 * (x**2)) - (24 * x)
        return y

    # create an array to store
    # the y-values
    ylist = []

    # loop through each x, generate
    # the corresponding y value and
    # append it to the array
    for x in xlist:
        y = actual_function(x)
        ylist.append(y)

    # return the y list
    return ylist
```

In [5]:
```
# plot the first order derivative

%matplotlib inline
# import numpy to help us generate our values
# import matplotlib to help us plot
import numpy as np
import matplotlib.pylab as plt

# generate our x-values
xlist = list(np.arange(-6,6.1,0.1))

# generate our y-value
ylist = sample_1stderivative(xlist)

# plot the graph
plt.plot(xlist,ylist)
plt.title("First Order Derivative")
plt.show()
```



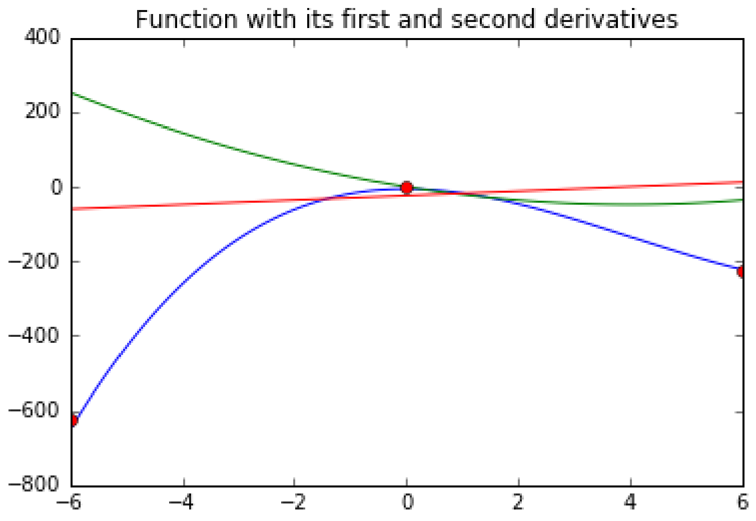
In [6]:
```
# create the second derivative of the function that
# we can use to generate the points for plotting
def sample_2ndderivative(xlist):
    """function that takes a list of numbers
    and returns a list of y-values f"""


    # write the function
    def actual_function(x):
        y = (6 * x) - 24
        return y

    # create an array to store
    # the y-values
    ylist = []

    # loop through each x, generate
    # the corresponding y value and
    # append it to the array
    for x in xlist:
        y = actual_function(x)
        ylist.append(y)

    # return the y list
    return ylist
```

In [7]:
```
# plot the second order derivative

%matplotlib inline
# import numpy to help us generate our values
# import matplotlib to help us plot
import numpy as np
import matplotlib.pylab as plt

# generate our x-values
xlist = list(np.arange(-6,6.1,0.1))

# generate our y-value
ylist = sample_2ndderivative(xlist)

# plot the graph
plt.plot(xlist,ylist)
plt.title("Second Order Derivative")
plt.show()
```



Plot the graph and its derivatives on the same plot

In [11]:
```
%matplotlib inline
# import numpy to help us generate our values
# import matplotlib to help us plot
import numpy as np
import matplotlib.pylab as plt

# generate the x values
xlist = list(np.arange(-6,6.1,0.1))

# generate the y-values for the original,
# the first order, and 2nd order derivatives
ylist = sample_function(xlist)
ylist_1 = sample_1stderivative(xlist)
ylist_2 = sample_2ndderivative(xlist)

# plot each graph
plt.plot(xlist,ylist,xlist,ylist_1,xlist,ylist_2)

# plot the eyeballed extrema on the range -6,6
plt.plot(0,0,'ro')
plt.plot(6,-225,'ro')
plt.plot(-6,-625,'ro')

# show the plot
plt.title("Function with its first and second derivatives")
plt.show()
```



In the graph above, we have labelled the local extrema with red dots. The maximum is at (0,0). We can determine this using first-order and second-order derivatives. The first order derivative of the function crosses zero when x=0. This identifies that there is a local extrema at the point where x=0 on the original function. The second order derivative is negative at x=0. This identifies that the local extrema identified by the first order derivative is actually a local maximum value. By looking at the graph of the original function, we can tell that this is true.
In addition to using the first- and second-order derivatives to identify extrema, we should also test the bounds of our function. If this function would continue unbounded then these values would not be local extrema. However, since our function is bounded, we want to test if these are locally for the function defined in this interval. We find that they are because the neighboring points are all higher than these bounds, implying that these points are local minimums.

For unconstrained multi-variate optimization, the **first order necessary conditions for optimality (FOC)** are that when the first order derivative is zero the objective function has reached a local extrema. Mathematically, we can define this as: (1) given the problem, *P*, that is to maximize or minimize f(x), then (2) if $\overline{x}$ is is the solution to *P*, the first order derivative of $\overline{x}$ equals 0.

For unconstrained multi-variate optimization, the **second order necessary conditions for optimality (SOC)** are that when first order derivative is zero AND (1) the second order derivative being positive means that we've found local a minimum or (2) the second order derivative being negative means that we've found a local maximum. Mathematically, we can define this as: (1) given the problem, *P*, that is to maximize or minimize f(x), then if $\overline{x}$ is the **minimum** solution to *P*, then the first order derivative of $\overline{x}$ equals 0 AND the second order derivative of $\overline{x}$ is positive AND (2) given the problem, *P*, that is to maximize or minimize f(x), then if $\overline{x}$ is the **maximum** solution to *P*, then the first order derivative of $\overline{x}$ equals 0 AND the second order derivative of $\overline{x}$ is negative.

In this context, the **Hessian matrix** is simply the matrix of the second order partial derivatives. When optimizing linear regression, we can think of the first order derivative matrix as telling us where to go and the Hessian matrix as telling us how far to go.

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)*

Let us define the function, f, of which we are finding the roots

$$f(x) = x^3 + 2x - 4$$

Then the derivative of f with respect to x is:

$$\frac{df}{dx} = 3x^2 + 2$$

Let us start with x=1 as the first approximation of the root of f, $xt1$.

$$xt1 = 1$$
$$f(xt1) = (1)^3 + 2(1) - 4 = 1^3 + 2 - 4 = 1 + 2 - 4 = 3 - 4 = -1$$
$$f'(xt1) = 3(1)^2 + 2 = 3(1) + 2 = 3 + 2 = 5$$

Let us find the tangent line to $xt1$.
Remember that the formula for the tangent line is y = mx + b where m is the slope and b is the y-intercept.

$$y = mx + b$$
$$m = f'(xt1) = 5$$

We know that the tangent line must pass through (xt1,f(xt1)).

$$-1 = 5(1) + b$$
$$-1 = 5 + b$$
$$-6 = b$$

Therefore, the equation of the tangent line to $xt1$ is:

$$y = 5x - 6$$

We can solve for the next closest root by finding where this tangent line intersects the x-axis.

$$y = 0 = 5x - 6$$
$$0 = 5x - 6$$
$$6 = 5x$$
$$\frac{6}{5} = x$$
$$x = 1.2$$
$$xt2 = 1.2$$

The second approximation of the root using the Newton-Rasphson method is 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?
Fill in the BLANKS here:
Convex minimization, a subfield of optimization, studies the problem of minimizing BLANK functions over BLANK sets. The BLANK property can make optimization in some sense "easier" than the general case - for example, any local minimum must be a global minimum.

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.

Objective function:

$$f = 0.5 * \sum_i weight_i * (W * X_i - y_i)^2$$

Exapand the objective function

$$f = 0.5 * \sum_i weight_i * (W * X_i - y_i) * (W * X_i - y_i)$$
$$f = 0.5 * \sum_i weight_i * (W^2 X_i^2 - WX_i y_i - WX_i y_i + y_i^2)$$
$$f = 0.5 * \sum_i weight_i * (W^2 X_i^2 - 2WX_i y_i + y_i^2)$$
$$f = 0.5 * \sum_i W^2 X_i^2 * weight_i - 2WX_i y_i * weight_i + y_i^2 * weight_i$$

Calculate the partial derivatives with respect to W, the coefficients

We are attempting to choose the coefficients that will minimize this function.

$$\frac{\partial f}{\partial W} = 0.5 * \sum_i 2WX_i^2 * weight_i - 2X_i y_i * weight_i + 0$$
$$\frac{\partial f}{\partial W} = 0.5 * \sum_i 2WX_i^2 * weight_i - 2X_i y_i * weight_i$$
$$\frac{\partial f}{\partial W} = 0.5 * \sum_i 2X_i * weight_i * (WX_i - y_i)$$
$$\frac{\partial f}{\partial W} = \sum_i 0.5 * 2X_i * weight_i * (WX_i - y_i)$$
$$\frac{\partial f}{\partial W} = \sum_i X_i * weight_i * (WX_i - y_i)$$

JUST FOR FUN: Calculate the partial derivatives with respect to x and y

I need some more practice taking partial derivatives so I also take the partial derivatives with respect to x and y.

With respect to x:

$$\frac{\partial f}{\partial x} = 0.5 * \sum_i 2W^2 X_i * weight_i - 2Wy_i * weight_i + 0$$
$$\frac{\partial f}{\partial x} = 0.5 * \sum_i 2W^2 X_i * weight_i - 2Wy_i * weight_i$$
$$\frac{\partial f}{\partial x} = \sum_i 0.5 * 2W^2 X_i * weight_i - 0.5 * 2Wy_i * weight_i$$
$$\frac{\partial f}{\partial x} = \sum_i W^2 X_i * weight_i - Wy_i * weight_i$$

With respect to y:

$$\frac{\partial f}{\partial y} = 0.5 * \sum_i 0 - 2WX_i * weight_i + 2y_i * weight_i$$

$$\frac{\partial f}{\partial y} = 0.5 * \sum_i -2WX_i * weight_i + 2y_i * weight_i$$

$$\frac{\partial f}{\partial y} = \sum_i 0.5 * -2WX_i * weight_i + 0.5 * 2y_i * weight_i$$

$$\frac{\partial f}{\partial y} = \sum_i -WX_i * weight_i + y_i * weight_i$$

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 this notebook (http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipynb)
Weight each example as follows:
weight(x)= abs(1/x)

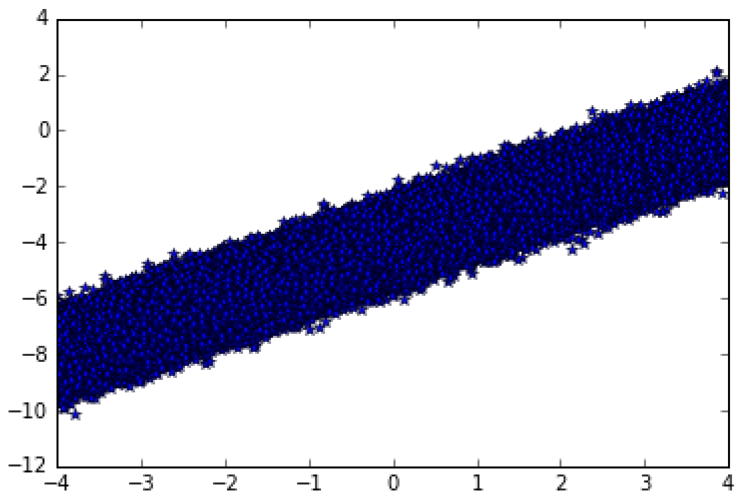
Sample 1% of the data in MapReduce and use the sampled dataset to train a (weighted if available in SciKit-Learn) linear regression model locally using SciKit-Learn (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

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

Generate 1,000,000 datapoints

Size: 1,000,000 points
True model: y = 1.0 • x - 4.0
Noise: Normal Distributed mean = 0, var = 0.5
Domain: -4,4

In [8]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# set our size and domain
SIZE = 1000000
LOWER = -4.0
UPPER = 4.0

# set the normal distribution
MEAN = 0
VAR = 0.5

# generate the x- and y-values
x = np.random.uniform(LOWER,UPPER,SIZE)
y = x * 1.0 - 4.0 + np.random.normal(MEAN,VAR,SIZE)

# generate the weigths for the x and y coordinates
w = abs(1.0/x)

# plot the points
plt.plot(x,y,"*")
plt.show()
```



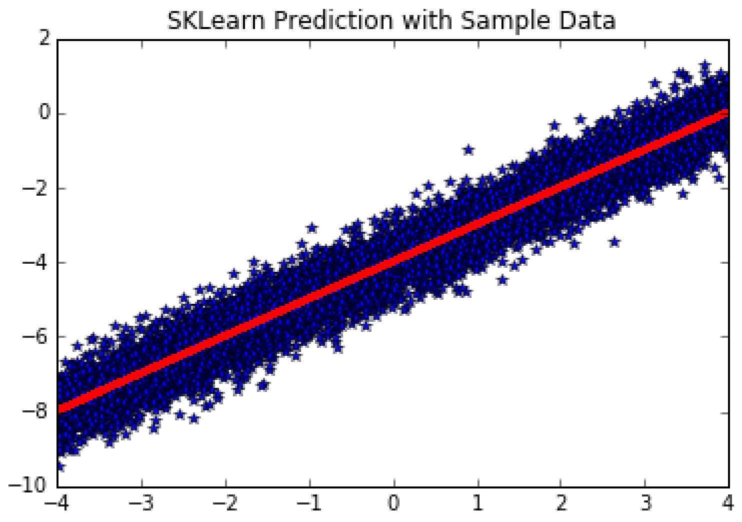
Generate the random sample of 1% from the data

In [9]:
```
import numpy as np

# zip the x and y values together
# so that we might randomly sample them
yxw = np.array(zip(y,x,w))

# save the file as data
np.savetxt('data.txt',yxw,delimiter = ",")

# set the sample of 1%
SAMPLE_SIZE = 0.01 * SIZE

# grab the random sample
yxw_sample = yxw[np.random.randint(0,len(yxw),size=SAMPLE_SIZE),:]
y_sample = yxw_sample[:,0]
x_sample = yxw_sample[:,1]
w_sample = yxw_sample[:,2]

# save the sameple as well
np.savetxt('data_sample.txt',yxw_sample,delimiter = ",")
```

```
/Users/Alex/miniconda2/envs/main/lib/python2.7/site-packages/ipykernel/
__main__.py:14: VisibleDeprecationWarning: using a non-integer number i
nstead of an integer will result in an error in the future
```

In [12]:
```
from sklearn.linear_model import LinearRegression

# use sklearn to generate a weighted
# linear regression from the sample
# data
sk_linear = LinearRegression().fit(x_sample.reshape(-1,1),\
                                   y_sample,\
                                   w_sample)

# print out the attributes predicted
# by the model, coefficient and intercept
print "Coefficient:",sk_linear.coef_[0]
print "Intercept:",sk_linear.intercept_

# plot the model compared to the sample
plt.plot(x_sample,y_sample,"*")
plt.plot(x_sample,sk_linear.predict(x_sample.reshape(-1,1)),color='red',
         linewidth=3)
plt.title("SKLearn Prediction with Sample Data")
plt.show()
```

```
Coefficient: 0.999179308085
Intercept: -3.97843423919
```



MRJob Class to calculate weighted OLS

This MRJob class calculates the gradient of the entire training set.

- Mapper: emits the partial gradient for each example
- Reducer: accumulates the gradient and emits the total gradient

Inspired by class notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipynb).

```
%%writefile MRGDUpdate_LinearRegression.py

# import the MRJob class
from mrjob.job import MRJob

class MRGDUpdate_LinearRegression(MRJob):

    # before processing input, we read in
    # the existing coefficients file
    def read_coef_file(self):

        # open the coefficients file
        with open('coefs.txt', 'r') as myfile:

            # set the coefficients for this instances
            # by reading in each line from the
            # file
            self.coefs = \
            [float(v) for v in myfile.readline().split(',')]

        # initialze gradient for this iteration
        self.partial_Gradient = [0]*len(self.coefs)
        self.partial_count = 0

    # calculate partial gradient for each example
    def partial_gradient(self, _, line):

        # set the points as D
        D = (map(float,line.split(',')))

        # grab the weight
        wgt_xi = D[2]

        # y_hat is the predicted value given
        # current coefficients
        y_hat = self.coefs[0] + self.coefs[1]*D[1]

        # store the difference between each
        # predicted value and the actual value
        self.partial_Gradient =  [self.partial_Gradient[0] + \
                                  D[0]-y_hat, self.partial_Gradient[1] +
 \
                                  (D[0]-y_hat) * D[1] * wgt_xi]

        # increment the counter for the
        # partial gradients we've calculated
        # so far
        self.partial_count = self.partial_count + 1

        # we don't yield anything because we leave
        # that to our mapper final
        # 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 gradient
    # output: key = None, Value = gradient vector
    def gradient_accumulater(self, _, partial_Gradient_Record):

        # initalize an array to hold the gradient we accumulate
        total_gradient = [0]*2

        # initalize a count
        total_count = 0

        # loop through each input in this reducer,
        # cumulatively sum the gradient and
        # increment the count variable
        for partial_Gradient,partial_count in partial_Gradient_Record:
            total_count = total_count + partial_count
            total_gradient[0] = total_gradient[0] + partial_Gradient[0]
            total_gradient[1] = total_gradient[1] + partial_Gradient[1]

        # yield the average of the gradients of the gradients
        yield None, [v/total_count for v in total_gradient]

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

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

```
Overwriting MRGDUpdate_LinearRegression.py
```

Get the cloud ready for our analysis

Spin up a cluster

In [1]:
```
# create the cluster
!mrjob create-cluster \
--max-hours-idle 1 \
--aws-region=us-west-1 -c ~/.mrjob.conf
```

```
Unexpected option hadoop from /Users/Alex/.mrjob.conf
Using s3://mrjob-f8c316b67324528f/tmp/ as our temp dir on S3
Creating persistent cluster to run several jobs in...
Creating temp directory /var/folders/k8/fy2j66nj4xsczx6cbcxhjlvm0000gn/
T/no_script.Alex.20160624.170118.208751
Copying local files to s3://mrjob-f8c316b67324528f/tmp/no_script.Alex.2
0160624.170118.208751/files/...
j-TTYM4ZTIQGX6
```

Move the data files up to the cloud

In [24]:
```
!aws s3 cp data.txt s3://aks-w261-hw6/data.txt
!aws s3 cp data_sample.txt s3://aks-w261-hw6/data_sample.txt
```

```
upload: ./data.txt to s3://aks-w261-hw6/data.txt
upload: ./data_sample.txt to s3://aks-w261-hw6/data_sample.txt
```

MRJob Runner to calculate the coefficients

```
%reload_ext autoreload
%autoreload 2

from numpy import random,array
from MRGDUpdate_LinearRegression import MRGDUpdate_LinearRegression

# set the learning rate and the stop criterion
# how quickly we'll move and when to stop getting
# better and better approximations
learning_rate = 0.05
stop_criteria = 0.00001

# generate random values as inital coefficients
coefficients = array([random.uniform(-3,3),random.uniform(-3,3)])

# write the coefficients to the files
with open('coefs.txt', 'w+') as f:
    f.writelines(','.join(str(j) for j in coefficients))

# move the coefficients file to the cloud
!aws s3 cp coefs.txt s3://aks-w261-hw6/coefs.txt --quiet

# create a mrjob instance for the gradient
# descent update over the test data, that runs
# locally

# run the job on our entire generated data
# in the coud
mr_job = \
MRGDUpdate_LinearRegression(args=['-r', 'emr',\
                                  's3://aks-w261-hw6/data.txt',\
                                  '--file=s3://aks-w261-hw6/coefs.txt',\
                                  '--cluster-id=j-TTYM4ZTIQGX6',\
                                  '--aws-region=us-west-1',\
                                  '--output-dir=s3://aks-w261-hw6/out_6_
5',\
                                  '--no-output'])

# test the job on our sample generated data
# in the coud
# mr_job = \
# MRGDUpdate_LinearRegression(args=['-r', 'emr',\
#                                    's3://aks-w261-hw6/data_sample.tx
t',\
#                                    '--file=s3://aks-w261-hw6/coefs.tx
t',\
#                                    '--cluster-id=j-3RGX32QSDBOPN',\
#                                    '--aws-region=us-west-1',\
#                                    '--output-dir=s3://aks-w261-hw6/out_
6_5',\
#                                    '--no-output'])

# test on the same file from the example notebook
# mr_job = \
# MRGDUpdate_LinearRegression(args=['LinearRegression.csv',\
#                                   '--file=coefs.txt'])
# update centroids iteratively
# keep track of the iteration where on by initalizing
# a counter,i
i = 0

# initalize the condition to keep iterating until
# we've met our stop criterion
not_stopping = True

# while we have yet to decide to stop
while(not_stopping):

    # prin the iteration number with the current
    # coefficients
    print "iteration = "+str(i)+"  coefficients =",coefficients

    # save the coefficients from previous iteration
    coefficients_old = coefficients

    # remove any output from a previous run
    !aws s3 rm --recursive s3://aks-w261-hw6/out_6_5 --quiet

    # run the MRJob that we created to iteratively
    # update
    with mr_job.make_runner() as runner:
        runner.run()

        # loop through each line of output
        for line in runner.stream_output():

            # grab the key,value from the output
            # and store it
            key,value =  mr_job.parse_output_line(line)

            # update the coefficients
            coefficients = coefficients + learning_rate*array(value)

    # remove any output from a previous run
    !aws s3 rm --recursive s3://aks-w261-hw6/out_6_5 --quiet

    # update our counter
    i = i + 1

    # write the updated coefficients to file
    with open('coefs.txt', 'w+') as f:
        f.writelines(','.join(str(j) for j in coefficients))

    # move the coefficients file to the cloud
    !aws s3 cp coefs.txt s3://aks-w261-hw6/coefs.txt --quiet

    # stop if coefficients coverge
    if(sum((coefficients_old-coefficients)**2)<stop_criteria):

        # set the stopping variable to false
        # and break out
        not_stopping = False
        break
```

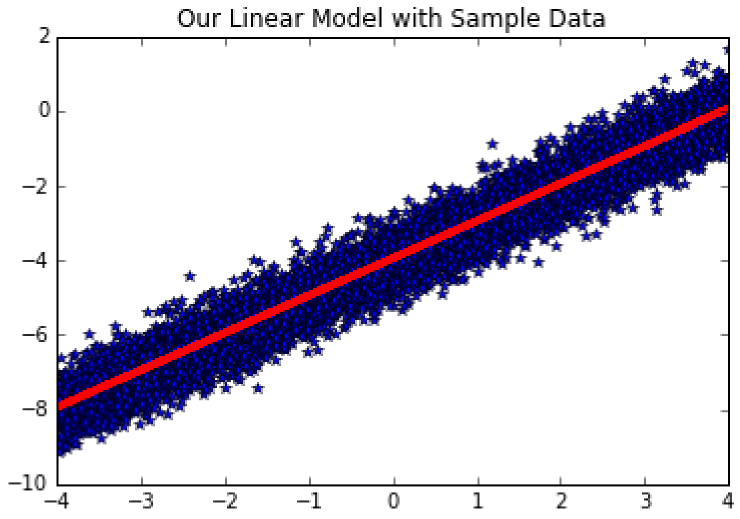
```
# print the final coefficients
print "Final coefficients\n"
print coefficients
```

```
iteration = 0  coefficients = [ 0.63602093  0.21051394]
iteration = 1  coefficients = [ 0.40427824  0.28973606]
iteration = 2  coefficients = [ 0.18412415  0.36102035]
iteration = 3  coefficients = [-0.02502093  0.42516166]
iteration = 4  coefficients = [-0.22370757  0.48287528]
iteration = 5  coefficients = [-0.41245881  0.53480489]
iteration = 6  coefficients = [-0.59177153  0.58152974]
iteration = 7  coefficients = [-0.76211775  0.6235711 ]
iteration = 8  coefficients = [-0.92394588  0.66139802]
iteration = 9  coefficients = [-1.07768191  0.69543263]
iteration = 10  coefficients = [-1.22373051  0.72605478]
iteration = 11  coefficients = [-1.36247612  0.7536063 ]
iteration = 12  coefficients = [-1.49428393  0.77839477]
iteration = 13  coefficients = [-1.6195009  0.800697 ]
iteration = 14  coefficients = [-1.73845661  0.82076208]
iteration = 15  coefficients = [-1.85146416  0.83881415]
iteration = 16  coefficients = [-1.958821    0.85505492]
iteration = 17  coefficients = [-2.0608097   0.86966588]
iteration = 18  coefficients = [-2.1576987   0.88281036]
iteration = 19  coefficients = [-2.249743    0.89463533]
iteration = 20  coefficients = [-2.33718487  0.90527305]
iteration = 21  coefficients = [-2.42025445  0.91484252]
iteration = 22  coefficients = [-2.49917037  0.92345083]
iteration = 23  coefficients = [-2.57414034  0.93119435]
iteration = 24  coefficients = [-2.64536167  0.93815978]
iteration = 25  coefficients = [-2.7130218   0.94442514]
iteration = 26  coefficients = [-2.77729881  0.95006065]
iteration = 27  coefficients = [-2.83836187  0.95512949]
iteration = 28  coefficients = [-2.89637168  0.95968849]
iteration = 29  coefficients = [-2.95148091  0.9637888 ]
iteration = 30  coefficients = [-3.00383461  0.96747645]
iteration = 31  coefficients = [-3.05357055  0.97079286]
iteration = 32  coefficients = [-3.10081964  0.97377528]
iteration = 33  coefficients = [-3.14570622  0.97645725]
iteration = 34  coefficients = [-3.18834842  0.97886893]
iteration = 35  coefficients = [-3.22885846  0.98103746]
iteration = 36  coefficients = [-3.26734296  0.98298727]
iteration = 37  coefficients = [-3.3039032   0.98474033]
iteration = 38  coefficients = [-3.3386354   0.98631642]
iteration = 39  coefficients = [-3.37163096  0.98773331]
iteration = 40  coefficients = [-3.40297671  0.98900701]
iteration = 41  coefficients = [-3.43275515  0.99015193]
iteration = 42  coefficients = [-3.46104465  0.99118102]
iteration = 43  coefficients = [-3.48791965  0.99210593]
iteration = 44  coefficients = [-3.51345089  0.99293714]
iteration = 45  coefficients = [-3.53770555  0.99368409]
iteration = 46  coefficients = [-3.56074747  0.99435527]
iteration = 47  coefficients = [-3.58263727  0.99495831]
iteration = 48  coefficients = [-3.60343258  0.99550008]
iteration = 49  coefficients = [-3.62318811  0.99598675]
iteration = 50  coefficients = [-3.64195585  0.99642389]
iteration = 51  coefficients = [-3.6597852   0.99681649]
iteration = 52  coefficients = [-3.67672307  0.99716905]
iteration = 53  coefficients = [-3.69281404  0.99748561]
iteration = 54  coefficients = [-3.70810046  0.99776981]
iteration = 55  coefficients = [-3.72262256  0.99802493]
iteration = 56  coefficients = [-3.73641854  0.9982539 ]
iteration = 57  coefficients = [-3.74952472  0.99845938]
iteration = 58  coefficients = [-3.76197559  0.99864374]
iteration = 59  coefficients = [-3.77380391  0.99880912]
iteration = 60  coefficients = [-3.78504081  0.99895746]
iteration = 61  coefficients = [-3.79571586  0.99909047]
iteration = 62  coefficients = [-3.80585716  0.99920973]
iteration = 63  coefficients = [-3.81549139  0.99931662]
iteration = 64  coefficients = [-3.82464391  0.99941241]
iteration = 65  coefficients = [-3.8333388   0.99949823]
iteration = 66  coefficients = [-3.84159894  0.99957509]
iteration = 67  coefficients = [-3.84944608  0.99964391]
iteration = 68  coefficients = [-3.85690086  0.99970551]
iteration = 69  coefficients = [-3.86398289  0.99976064]
iteration = 70  coefficients = [-3.87071083  0.99980994]
iteration = 71  coefficients = [-3.87710236  0.99985403]
iteration = 72  coefficients = [-3.88317432  0.99989344]
iteration = 73  coefficients = [-3.88894268  0.99992865]
iteration = 74  coefficients = [-3.89442263  0.99996009]
iteration = 75  coefficients = [-3.89962857  0.99998816]
iteration = 76  coefficients = [-3.90457422  1.00001319]
iteration = 77  coefficients = [-3.90927258  1.00003551]
iteration = 78  coefficients = [-3.91373603  1.0000554 ]
iteration = 79  coefficients = [-3.9179763   1.00007311]
iteration = 80  coefficients = [-3.92200456  1.00008887]
iteration = 81  coefficients = [-3.9258314   1.00010288]
iteration = 82  coefficients = [-3.9294669   1.00011533]
iteration = 83  coefficients = [-3.93292063  1.00012638]
iteration = 84  coefficients = [-3.93620167  1.00013618]
Final coefficients

[-3.93931866  1.00014486]
```

Plot the data with the line we calculated

We choose to plot only a sample because we don't want to overwhelm pyplot.

In [11]:
```
# import libraries to help us plot
from matplotlib import pyplot as plt

# generate our linear data
y_inter = coefficients[0]
slope = coefficients[1]
y_pred = x_sample * slope + y_inter


# plot the model compared to the sample
plt.plot(x_sample,y_sample,"*")
plt.plot(x_sample,y_pred,color='red',
         linewidth=3)
plt.title("Our Linear Model with Sample Data")
plt.show()
```



Comments

For this full data set, I used a 20 node cluster on Amazon web services. I had 1 master node, m1.medium, and 19 slave instances, m3.xlarge. Both methods are very accurate. The SKLearn method is especially useful when the data set is smaller and can be handled in memory. The upload, download time from AWS is too lengthy is dealing with small data set. For large data sets, it makes a lot of sense to parallelize.

HW6.6 Clean up notebook for GMM via EM *(Optional)*

Using this notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/0t7985e40fovlkw/EM-GMM-MapReduce%20Design%201.ipynb) as a starting point.

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 algo 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: Probability of class step

Given priors, mean vector and covariance matrix, calculate the probability of that each data point belongs to a class.

$$p(\omega_k|x^{(i)}, \theta) = \frac{\pi_k \mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(x^{(i)}|\mu_j, \Sigma_j)} \text{ Equation 6.6.1}$$

M-Step: Update priors, means, and covariance

Given probabilities, update priors, mean and covariance.

$$\hat{\mu}_k = \frac{1}{n_k} \sum_{i=1}^{n} p(\omega_k | x^{(i)}, \theta) x^{(i)} \text{Equation 6.6.2}$$

$$\hat{\Sigma}_k = \frac{1}{n_k} \sum_{i=1}^{n} p(\omega_k | x^{(i)}, \theta)(x^{(i)} - \hat{\mu}_k)(x^{(i)} - \hat{\mu}_k)^T \text{Equation 6.6.3}$$

$$\hat{\pi}_k = \frac{n_k}{n} \text{ where } n_k = \sum_{i=1}^{n} p(\omega_k | x^{(i)}, \theta) \text{ Equation 6.6.4}$$

Data Generation

We generate that we want to group.

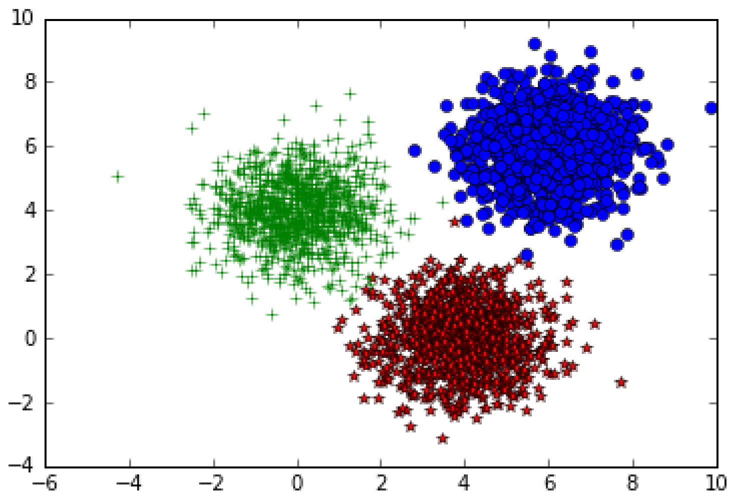
In [73]:
```
# open any plots in the notebook rather than
# generating a new notebook
%matplotlib inline

# import libraries to help us manipulate
# and generate the data
import numpy as np
import pylab
import json

# set the amount of data we want to generate
size1 = size2 = size3 = 1000

# create each of the 3 samples and as we create
# each sample append it to our list of data
samples1 = np.random.multivariate_normal([4, 0], [[1, 0],[0, 1]], size1)
data = samples1
samples2 = np.random.multivariate_normal([6, 6], [[1, 0],[0, 1]], size2)
data = np.append(data,samples2, axis=0)
samples3 = np.random.multivariate_normal([0, 4], [[1, 0],[0, 1]], size3)
data = np.append(data,samples3, axis=0)

# randomize the data within the list
data = data[np.random.permutation(size1+size2+size3),]

# write the data we generated to a file
# store locally, called data.txt
with open("data6_6.txt", "w") as f:
    for row in data.tolist():
        json.dump(row, f)
        f.write("\n")

# when completed let the user know
print "data6_6.txt file generated"
```

```
data6_6.txt file generated
```

Data Visualization

In [74]:

```
# import the matplotlib library
# to help us generate pretty visualization
import matplotlib.pylab

# plot each of the samples in a different color
pylab.plot(samples1[:, 0], \
           samples1[:, 1],'*', \
           color = 'red')
pylab.plot(samples2[:, 0], \
           samples2[:, 1],'o', \
           color = 'blue')
pylab.plot(samples3[:, 0], \
           samples3[:, 1],'+', \
           color = 'green')
pylab.show()
```



Initalization

We initalize our mixture model with k, 3, randomly based center points.

- **Mapper:** sends the first 2,000 points to the reducer
- **Reducer:** grabs k clusters and initalizes the probabilities, means, and covariances

```
%%writefile mr_GMixEmInitialize.py

# import the MRJob class
from mrjob.job import MRJob

# import the libraries to help us manipulate
# the data
from numpy import mat, zeros, shape, random, array, zeros_like, dot, lin
alg
from random import sample
import json
from math import pi, sqrt, exp, pow


class MrGMixEmInit(MRJob):
    DEFAULT_PROTOCOL = 'json'

    # initalize the class and set the number
    # of mappers
    def __init__(self, *args, **kwargs):

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

        self.numMappers = 1
        self.count = 0


    # configure the options like the number of
    # of clusters we're attempting to calculate
    def configure_options(self):

        super(MrGMixEmInit, self).configure_options()

        # grab the number of the clusters passed in
        # as an argument
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')

        # grab the path where we'll be storing
        # intermediate results
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt is st
ored')


    # simple mapper that grabs the first
    # 2,000 points
    def mapper(self, key, xjIn):

        # if we have to reach 2,000 points
        # for each cluster
        if self.count <= 2*self.options.k:

            # increment the count by 1
            # and yield 1 for the key
            # and the point
            self.count += 1
            yield (1,xjIn)


    # we yielded 1 for the key for every
    # line outputted by the mapper, this
    # leads to a single reducer; our
    # reducer
    def reducer(self, key, xjIn):

        # initalize an empty list of potential
        # centroids
        cent = []

        # for each point that we outputted
        # from the mapper
        for xj in xjIn:

            # load the point in because
            # it was transferred using
            # json protoccol
            x = json.loads(xj)

            # add the point to the list
            # of potential centroids
            cent.append(x)

            # yield the key of 1, and
            # the point
            yield 1, xj

        # grab k-sample indexes
        index = sample(range(len(cent)), self.options.k)

        # initalize a second array to hold
        # the actual center points
        cent2 = []

        # add the points we got from the sample
        # index to our second list
        for i in index:
            cent2.append(cent[i])



        # use the covariance of the selected centers as
        # the starting guess for covariances
        # first, calculate mean of centers

        # grab the mean of the first centroid
        mean = array(cent2[0])

        # loop through k centroids, skipping
        # the first centroid
        for i in range(1,self.options.k):
            # add the mean of this next
            # centroid to the mean
            mean = mean + array(cent2[i])

        # calculate the mean of the centroids
        # by dividing the new total mean by
        # the number of clusters
        mean = mean/float(self.options.k)



        # gather the covariances

        # create a covariance array, initalize it
        # to zeroes, create it as a 2d array
        cov = zeros((len(mean),len(mean)),dtype=float)

        # loop through each of the centroids
        for x in cent2:

            # calculate the difference between
            # part of the point and the mean
            xmm = array(x) - mean

            # loop through each value in the
            # mean
            for i in range(len(mean)):

                # set the covariance as the existing
                # covariance plus the the new
                # difference to the mean squared
                cov[i,i] = cov[i,i] + xmm[i]*xmm[i]

        # set the covariance as the covariance sum of
        # differences to the mean divided by the number
        # of clustsers
        cov = cov/(float(self.options.k))

        # inverse the covariance matrix
        covInv = linalg.inv(cov)

        # send the inverse covariance matrix to a
        # list and multiply it by the number
        # of clusters
        cov_1 = [covInv.tolist()]*self.options.k

        # write all of this to a debug text file
        # so that the user go back in and see
        # what's up
        jDebug = json.dumps([cent2,mean.tolist(),cov.tolist(),covInv.tol
ist(),cov_1])
        debugPath = self.options.pathName + 'debug.txt'
        fileOut = open(debugPath,'w')
        fileOut.write(jDebug)
        fileOut.close()

        # we also need a starting guess at
        # the phi's - prior probabilities. we
        # initialize them all with the same number
        # - 1/k - equally probably for each cluster

        # create an array of zeroes for the k clusters
        phi = zeros(self.options.k,dtype=float)

        # loop through each cluster
        for i in range(self.options.k):

            # set the phi value equal to 1 divided
            # by the number of clusters
            phi[i] = 1.0/float(self.options.k)

        # form output object by converting the
        # probabilities to a list
        outputList = [phi.tolist(), cent2, cov_1]

        # dump the output list
        jsonOut  = json.dumps(outputList)

        # write the parameters we've initalized
        # to a file
        fullPath = self.options.pathName + 'intermediateResults.txt'
        fileOut = open(fullPath,'w')
        fileOut.write(jsonOut)
        fileOut.close()

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.

```
%%writefile mr_GMixEmIterate.py

# import the MRJob class
from mrjob.job import MRJob

# import libraries to help us
# manipulate and deal with the data
from math import sqrt, exp, pow, pi
from numpy import zeros, shape, random, array, zeros_like, dot, linalg
import json

# define our Gaussian distribution
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

# define our MRJob class
class MrGMixEm(MRJob):
    DEFAULT_PROTOCOL = 'json'

    # initalize the class and set the options
    # like the number of mappers
    def __init__(self, *args, **kwargs):
        super(MrGMixEm, self).__init__(*args, **kwargs)

        # set the path where we'll read our
        # intermediate results and open the file
        fullPath = self.options.pathName + 'intermediateResults.txt'
        fileIn = open(fullPath)

        # import the file as a json file,
        # and then go ahead and close it
        inputJson = fileIn.read()
        fileIn.close()

        # convert the information we just
        # read in into a json format
        inputList = json.loads(inputJson)

        # set the class probabilities
        temp = inputList[0]
        self.phi = array(temp)

        # set the means
        temp = inputList[1]
        self.means = array(temp)

        # set the covariance matrices
        temp = inputList[2]
        self.cov_1 = array(temp)

        # initiliaze numpy arrays to hold
        # the probabilities, means, and covariances
        self.new_phi = zeros_like(self.phi)
        self.new_means = zeros_like(self.means)
        self.new_cov = zeros_like(self.cov_1)

        # set number of mappers
        self.numMappers = 1

        # count variable availabe to each
        # mapper and reducers
        self.count = 0


    # configure the options for the job by reading in
    # the arguments passed into the program
    def configure_options(self):
        super(MrGMixEm, self).configure_options()

        # set the number of centroids
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')

        # set the path to store intermediate results
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt is st
ored')


    # accumulates the partial sums and stores them
    # for the mapper final to output
    def mapper(self, key, val):

        # remember that in the initalization step
        # we yielded 1 as the key from the reducer
        # this means that all our values are read
        # into our mapper here

        # load in all the values and convert it
        # from a json to alist
        xList = json.loads(val)
        x = array(xList)

        # initalize a zeroes arrays for holding
        # the probabilities
        wtVect = zeros_like(self.phi)

        # loop through eah of the centroids
        # possibilities
        for i in range(self.options.k):

            # set the probability as the existing
            # probabilities multiplied by the
            # gaussian distribution with the means
            # covariance from when this job was
            # initalized
            wtVect[i] = self.phi[i] * \
            gauss(x, self.means[i], self.cov_1[i])

        # calculate sum of the probabilities and
        # normalize the probabilities
        wtSum = sum(wtVect)
        wtVect = wtVect/wtSum

        # accumulate to update the estimate of
        # probability densities

        # increment count
        self.count += 1

        # set the new probabilities as the existing
        # new probabilities summed with the new
        # weighted vector probabilities we just
        # calculated
        self.new_phi = self.new_phi + wtVect

        # loop through each of the centroids
        for i in range(self.options.k):

            # accumulate weighted x's for
            # calculating the mean
            self.new_means[i] = \
            self.new_means[i] + wtVect[i]*x

            #accumulate weighted squares for
            # estimating the covariance
            xmm = x - self.means[i]
            covInc = zeros_like(self.new_cov[i])

            # loop through each of the differences
            # from the mean
            for l in range(len(xmm)):

                # loop through each of the differences
                # from the mean
                for m in range(len(xmm)):

                    # set the covariance matrix as the
                    # matrix multiplication
                    covInc[l][m] = xmm[l]*xmm[m]

            # set the new covariance values
            self.new_cov[i] = self.new_cov[i] + \
            wtVect[i]*covInc

        # this mapper doesn't actually yield anything
        # we wait for the mapper final to
        # actually yield information


    # our mapper final method outputs the count
    # and probabilities, means, and covariances
    def mapper_final(self):
        # describe the output and convert it to
        # a json format
        out = [self.count, (self.new_phi).tolist(), \
               (self.new_means).tolist(), \
               (self.new_cov).tolist()]
        jOut = json.dumps(out)

        # yield the key,1, and the output
        # we just converted to a json
        yield 1,jOut


    # our reducer accumulates the partial sums
    # and outputs the new centroids
    def reducer(self, key, xs):

        # remember that our mapper final outputted
        # one thing, 1,jOut. we call in the entire
        # dataset into our reducer

        # set a dummy variable, first, to true
        first = True

        # accumulate partial sums
        # xs us a list of paritial stats, including count,
        # phi, mean, and covariance; each stats is
        # k-length array, storing info for k components

        # for each of the partial stats outputted
        # by the mapper final
        for val in xs:

            # as long as first is true
            if first:

                # load in the values as a json
                temp = json.loads(val)

                # initalize a set of arrays that
                # hold the values from this
                # first json
                totCount = temp[0]
                totPhi = array(temp[1])
                totMeans = array(temp[2])
                totCov = array(temp[3])

                # set the first to positive once
                # we've worked through our first
                # record
                first = False

            else:

                # load in the values a sjon
                temp = json.loads(val)

                # sum through each of statistics
                # performing a cumulative sum
                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
        # by dividing the probabilities by the total counts
        newPhi = totPhi/totCount

        # initalize arrays to hold the new means and
        # covariances; here we just initalize to get
        # the right sizes
        newMeans = totMeans
        newCov_1 = totCov

        # loop through each centroid
        for i in range(self.options.k):

            # set the new mean and covariance as
            # the total mean and covariance
            # divided by the total probability
            newMeans[i,:] = totMeans[i,:]/totPhi[i]
            tempCov = totCov[i,:,:]/totPhi[i]

            # invert the covariance matrix
            newCov_1[i,:,:] = linalg.inv(tempCov)

        # generate the output list based on the
        # new statistics we calculated
        outputList = [newPhi.tolist(), \
                      newMeans.tolist(), \
                      newCov_1.tolist()]

        # convert the output to a json
        jsonOut = json.dumps(outputList)

        # write the new parameters to file
        fullPath = self.options.pathName + 'intermediateResults.txt'
        fileOut = open(fullPath,'w')
        fileOut.write(jsonOut)
        fileOut.close()

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

```
Overwriting mr_GMixEmIterate.py
```

Driver

```
# import the MRJob classes that we just
# wrote
from mr_GMixEmInitialize import MrGMixEmInit
from mr_GMixEmIterate import MrGMixEm

# import libraries to help us deal with
# and manipulate our data
import json
from math import sqrt

# function to plot the samples we've
# previously generated
def plot_iteration(means):
    pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    pylab.plot(means[0][0], means[0][1],'*',markersize =10,color =
'red')
    pylab.plot(means[1][0], means[1][1],'*',markersize =10,color =
'red')
    pylab.plot(means[2][0], means[2][1],'*',markersize =10,color =
'red')
    pylab.show()

# function to calculate the euclidean
# distance between the two lists
def dist(x,y):

    # initalize a sum value at 0
    sum = 0.0

    # loop through each element in
    # the lists
    for i in range(len(x)):

        # calculate the difference
        # between lists
        temp = x[i] - y[i]

        # add this difference
        # to the sum, cumulatively
        sum += temp * temp

    # return the square root of the
    # sum
    return sqrt(sum)

# run our data initalizer to get some
# starting centroids
filePath = 'data6_6.txt'
mrJob = MrGMixEmInit(args=[filePath])
with mrJob.make_runner() as runner:
    runner.run()

# pull out the centroid values to compare with
# values after one iteration
emPath = "intermediateResults.txt"
```

```
fileIn = open(emPath)
paramJson = fileIn.read()
fileIn.close()

# set the delta and the number of iterations
# we are currently on
delta = 10
iter_num = 0

# begin iteration on change in centroids
# continue it while the delta, difference
# between the lists of centroids
# is greater than 0.02
while delta > 0.02:

    # print the iteration number we're on
    # and increment the interation counter
    print "Iteration" + str(iter_num)
    iter_num = iter_num + 1

    # parse old centroid values
    oldParam = json.loads(paramJson)
    oldMeans = oldParam[1]

    # load in our iteration class
    # and run it
    mrJob2 = MrGMixEm(args=[filePath])
    with mrJob2.make_runner() as runner:
        runner.run()

    # load in the new centroids from
    # the file outputted the job
    fileIn = open(emPath)
    paramJson = fileIn.read()
    fileIn.close()
    newParam = json.loads(paramJson)

    # grab the number of centroids
    # and the new means
    k_means = len(newParam[1])
    newMeans = newParam[1]

    # initalize the difference
    # to zero
    delta = 0.0

    # loop through each value,
    # comparing it to the previous
    # centroids and adding the difference
    # to the delta
    for i in range(k_means):
        delta += dist(newMeans[i],oldMeans[i])

    # print the previous means
    print oldMeans

    # plot th eold mean
    plot_iteration(oldMeans)

# print the last iteration once we're done
# with the new means and a plot with the
# centroids
print "Iteration" + str(iter_num)
print newMeans
plot_iteration(newMeans)
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-81-e9a3e90bfd4d> in <module>()
     53 # values after one iteration
     54 emPath = "intermediateResults.txt"
---> 55 fileIn = open(emPath)
     56 paramJson = fileIn.read()
     57 fileIn.close()

IOError: [Errno 2] No such file or directory: 'intermediateResults.txt'
```