# DATASCI W261: Machine Learning at Scale

**Nick Hamlin** (nickhamlin@gmail.com)
**Tigi Thomas** (tgthomas@berkeley.edu)
**Rock Baek** (rockb1017@gmail.com)
**Hussein Danish** (husseindanish@gmail.com)

Time of Submission: 10:30 AM EST, Saturday, Feb 27, 2016
W261-3, Spring 2016
Week 6 Homework

## Submission Notes:

- For each problem, we've included a summary of the question as posed in the instructions. cases, we have not included the full text to keep the final submission as uncluttered as pos reference, we've included a link to the original instructions in the "Useful Reference" below.
- Some aspects of this notebook don't always render nicely into PDF form. In these situation reference the complete rendered notebook on Github (https://github.com/nickhamlin/mids_261_homework/blob/master/HW6/MIDS-W261-2015-Week06-Hamlin-Thomas-Baek-Danish.ipynb)
- We've found that the Latex equations render beautifully on a local ipython server, but the q varies depending on which browser one uses when rendering a notebook online. Chrome is perfect, but seems to work the best of the browsers (Firefox renders way too small, IE rend too big). For highest quality results, please run the notebook locally.

## Useful References:

- **Original Assignment Instructions (https://www.dropbox.com/sh/5bex8l871t0bg3a/AABXfLW7xv9OUAzY01fMa29za/HW Questions.txt?dl=0)**
- Linear Regression in MRJob (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressio

In [1]:
```
#Use this to make sure we reload the MrJob code when we make changes
%load_ext autoreload
%autoreload 2
#Render matplotlib charts in notebook
%matplotlib inline

#Import some modules we know we'll use frequently
import numpy as np
import pylab as plt
```

# HW 6.0

## HW 6.0 - Problem Statement

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.

## HW 6.0 - Response

Mathematical optimization is the selection of a best element (with regard to some criteria) from some set of available alternatives. In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function.

*Optimization for inventory warehousing*

A common supply chain problem revolves around minimizing the cost of inventory that is staying in a warehouse. In this situation, a company will have a number of different products that each take up different amounts of space in the warehouse. The goal will be to find a combination of product quantities such that the space is best utilized. For example, let us consider a simplified scenario where our warehouse contains 100 different locations. Let us further say that we only carry two different products and that their quantities are denoted x1 and x2. Product 1 takes up two locations whereas product 2 takes up three locations. In this situation, our objective function is equal to 2x1 + 3x2 and this function is constrained to be less than or equal to 100. The decision variables here are the product quantities x1 and x2.

Many such projects have been deployed in the real word, however in reality there are often more variables and constraints to take into account such as the cost of the product, how often it is received, how often it is ordered, etc... In addition, inventory might be seasonal and can change and the frequency with which it is handled can change every few months. However, we have worked on many implementations with variations of this setup and they have had varying degress of success depending on the situation at the client.

# HW6.1

## HW 6.1 Problem Statement

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.

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?

## HW 6.1 Univariate optimiality conditions

For an unconstrained univariate optimization, we have the first-order condition that the first derivative of the objective function will be equal to zero for values of x that represent a maximur minimum of the objective function. That is:

$$f'(x = x*) = 0 \qquad \text{(First-order conditi}$$

When we find a root based on this first-order condition, we can invoke the second-order conditions for optimality to determine whether or not the root that we've found represents a loc maximum or local minimum. If the second derivative of the objective function is negative, then we've found a local max, and if the second derivative is positive, then we've found a local min. Mathematically speaking:

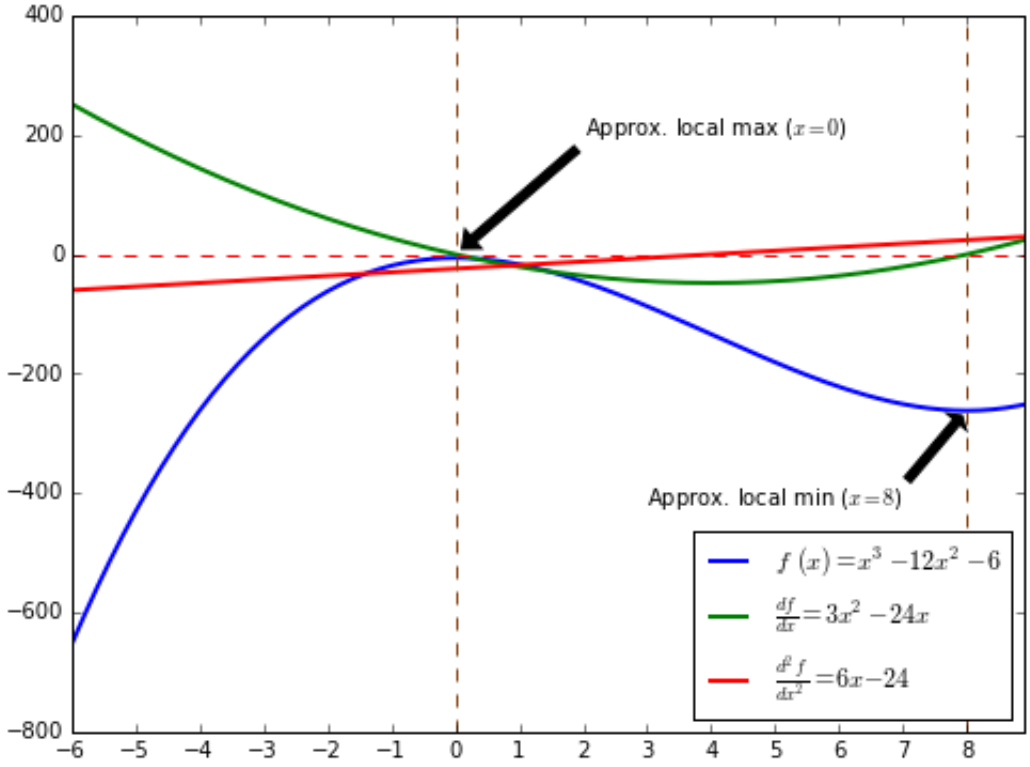
$$\begin{aligned} &\text{if } f''(x = x*) < 0, \text{ then maximum} \quad \text{(Second-order conditic} \\ &\text{if } f''(x = x*) > 0, \text{ then minimum} \end{aligned}$$

We can see these conditions more clearly in action with the visual example below.

## HW 6.1 Visual Example

To demonstrate, we'll plot the function $f(x) = x^3 - 12x - 6$ along with its first and second derivatives on the same set of axes.

In [10]:
```
# HW 6.1 - Plotting a visual example
def f(x):
    """The function in question"""
    return x**3-12*x**2-6

def dev1f(x):
    """The first derivative of our function"""
    return 3*x**2-24*x

def dev2f(x):
    """The second derivative of our function"""
    return 6*x-24

def plot_6_1():

    #Create vectors for X, Y, Y', and Y"
    x=np.arange(-6,9,.1)
    y0=f(x)
    y1=dev1f(x)
    y2=dev2f(x)

    #Plot each vector on the same coordinate plane
    fg = plt.figure(figsize=(8,6))
    plt.xticks(np.arange(min(x), max(x)+1, 1.0))
    plt.plot(x,y0, color="blue", label='$f\,(x)=x^3-12x^2-6$', linewidt
    plt.plot(x,y1, color="green", label='$\\frac{df}{dx} = 3x^2-24x$',
    plt.plot(x,y2, color="red", label='$\\frac{d^2f}{dx^2} = 6x-24$', l
    plt.legend(loc='lower right')

    #Plot dotted lines at each root and where x=0
    plt.axhline(0, color ="red", ls = '--', )
    plt.axvline(0, color ="saddlebrown", ls = '--')
    plt.axvline(8, color = "saddlebrown", ls = '--')

    #Add arrow annotations of critical points
    plt.annotate('Approx. local max ($x=0$)', xy=(0, 0), xytext=(2, 200
                arrowprops=dict(facecolor='black', shrink=0.05),
                )
    plt.annotate('Approx. local min ($x=8$)', xy=(7.97, -264), xytext=(
        arrowprops=dict(facecolor='black', shrink=0.05),
        )

    #Display results
    plt.show()

plot_6_1()
```




Here, we can invoke the first-order condition and look for spots where the first derivative (shown in green) crosses the x-axis (shown in red dashes). These two roots (0 and 8) represent the locations of the local extrema of the objective function (shown in blue). Based on the second-order conditions, we know that 0 represents a local maximum because the second derivative (shown in red) is negative, while 8 represents a local minimum because the second derivative is positive.

## HW 6.1 Multivariate optimiality conditions

In the multivariate case, we have similar first and second-order conditions for optimality, but we must express them in many dimensions. Instead of setting the simple first derivative of a univari objective function equal to 0 to find its extrema, we set the gradient of the multivariate objective function equal to zero to find extreme points. This gives us:

$$\nabla f(X = x') = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right] = (0, 0, \ldots 0) \quad \text{(Multivariate I}$$

Similarly, we can calculate the Hessian matrix, the matrix of partial second derivatives to evalua the second order conditions. If the values in the hessian matrix for a particular variable are posit then we've found a local minimum in that dimension, and vice versa.

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \quad \text{(Hessian Ma}$$

# HW6.2

## HW 6.2 Problem Statement

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)

## HW 6.2 Implementation

By setting up python functions for the function of which we want to approximate roots, as well as its first derivative, we can then create another function to run a single iteration of the Newton-Raphson method and output the result directly.

In [4]:
```
# HW 6.2 - Example Newton-Raphson Calculation
from __future__ import division

def f(x):
    """Calculate the value of our function"""
    return x**3+2*x-4

def dev1f(x):
    """Calculate the value of our function's derivative"""
    return 3*x**2+2

def iterate_nr_method(xt1):
    """Calculate a single iteration of the Newton-Raphson method for ou
    xt2=xt1-(f(xt1)/dev1f(xt1))
    return xt2

iterate_nr_method(1)
```

Out[4]:
```
1.2
```

# HW6.3

## HW 6.3 Problem Statement

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.

## HW 6.3 - What makes an optimization problem convex?

Our optimization problem is convex if the loss function we have is convex, that is, we can assume that there is a single global solution that minimizes our loss. A loss function is convex if the tangent lines at any point on the loss function are always below the loss function.

## HW 6.3 - What are the first order necessary conditions for optimality in convex optimization.

The first-order conditions for convex optimization are similar to those described earlier. Specifically, we must have a loss function that is differentiable. We will find the optimal solution when the first derivative of this loss function is equal to zero:

$$f'(x) = 0(\text{for all } x \in R).$$

## HW 6.3 - What are the second order optimality conditions for convex optimization?

The second order conditions for convex optimization is that our function can be differentiated twice and that the second derivative will be positive when the first derivative is equal to zero. The fact that this second derivative is positive indicates that we have found a global minimum. To summarize: $f''(x) >= 0(\text{for all } x \in R)$

## HW 6.3 - Are both necessary to determine the maximum or minimum of candidate optimal solutions?

If we know that the function is convex, we know that our local extremum will be a minimum and that it will be a global minimum. Therefore, if we know definitively in advance that our function is convex, invoking the second order condition doesn't tell us anything we don't already know. However, in practice, this is a useful confirmatory step.

## HW 6.3 - Fill in the BLANKS

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

## HW 6.4 - Problem Statement

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.

## HW 6.4 Derivation

Let us formulate the objective function above as follows:

$$J(W) = 0.5 * \sum_i w_i (WX_i - y_i)^2 \tag{1}$$

To get the gradient, we must perform the partial derivative with respect to W. First, we will apply the square term to simplify the terms for the derivatives:

$$J(W) = 0.5 * \sum_i w_i ((WX_i)^2 + y_i^2 - 2WX_i y_i) \tag{2}$$

This further becomes

$$J(W) = 0.5 * \sum_i w_i (W^2 X_i^2 + y_i^2 - 2WX_i y_i) \tag{3}$$

The gradient for this is thus:

$$\frac{\partial J(W)}{\partial W} = \frac{\partial}{\partial W}(0.5 * \sum_i w_i (W^2 X_i^2 + y_i^2 - 2WX_i y_i)) \tag{4}$$

Then applying the partial derivative with respect to W, we get:

$$\frac{\partial J(W)}{\partial W} = 0.5 * \sum_i w_i (2WX_i^2 - 2X_i y_i) \tag{5}$$

Bringing out the constant term (2) :

$$\frac{\partial J(W)}{\partial W} = 0.5 * 2 * \sum_i w_i(WX_i^2 - X_i y_i) \tag{6}$$

Finally, factoring out Xi:

$$\frac{\partial J(W)}{\partial W} = \sum_i w_i X_i(WX_i - y_i) \tag{7}$$

To find the value of W which minimizes J(W), we set the gradient to 0 and get:

$$0 = \sum_i w_i X_i(WX_i - y_i) \tag{8}$$

# HW 6.5

## HW 6.5 Problem Statement

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

- Generate one million datapoints just like in this notebook (http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressio
- Weight each example like this: $weight(x) = |(1/x)|$
- Sample 1% of the data in MapReduce and use the sampled dataset to train a (weighted if a 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 the data. Comment on your findings.

In [21]:
```
size = 1000000 #Create 1m random points
x = np.random.uniform(-4, 4, size)
y = x * 1.0 - 4 + np.random.normal(0,0.5,size)
data = zip(y,x) #NOTICE THAT THE Y-VALUE COMES FIRST!
np.savetxt('LinearRegression.csv',data,delimiter = ",")
```

In [22]:

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

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

def weight_point(x):
    """Define a simple function to calculate our weight value, given an
    return abs(1/x)

# This MrJob calculates the gradient of the entire training set
class MrJobBatchGDUpdate_LinearRegression(MRJob):
    # run before the mapper processes any input
    def read_weightsfile(self):
        # Read weights file
        # NOTE - THIS NOMENCLATURE CAN BE CONFUSING!
        # self.weights represents our estimate of the model parameters
        # NOT the weights that we are adding to each point based on the
        # weighting approach in the problem statement.
        with open('weights.txt', 'r') as f:
            self.weights = [float(v) for v in f.readline().split(',')]
        # Initialize 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] #Y_hat=beta_0+beta

        # Update partial gradient vector with gradient from current exa
        #NOTE THAT THESE NEXT TWO LINES ARE WHERE WE ADD THE WEIGHTS
        #THAT MAKE THIS A WOLS MODEL
        self.partial_Gradient[0]+=(D[0]-y_hat)*weight_point(D[1]) #Upda
        self.partial_Gradient[1]+=(D[0]-y_hat)*D[1]*weight_point(D[1])
        self.partial_count+=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):
        total_gradient = [0]*2
        total_count = 0
        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 None, [v/total_count for v in total_gradient]

    def steps(self):
```

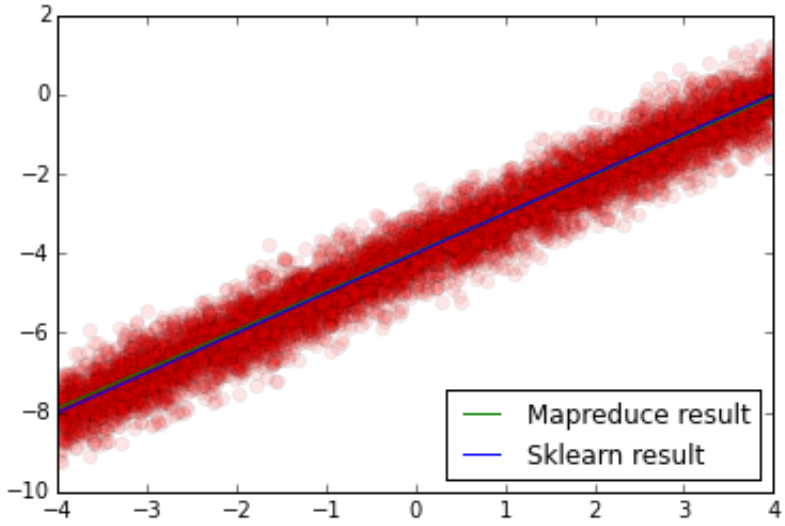
```
        return [MRStep(mapper_init=self.read_weightsfile,
                       mapper=self.partial_gradient,
                       mapper_final=self.partial_gradient_emit,
                       reducer=self.gradient_accumulater)]

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

```
Overwriting MrJobBatchGDUpdate_LinearRegression.py
```

In [1]:
```
from numpy import random,array
from MrJobBatchGDUpdate_LinearRegression import MrJobBatchGDUpdate_Line

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 all da
mr_job = MrJobBatchGDUpdate_LinearRegression(args=['LinearRegression.cs
                                                   'weights.txt','--no-
# Update centroids iteratively
i = 0
while(1):
    print "iteration ="+str(i)+"  weights =",weights
    # 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+=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\n"
print weights
```

```
iteration =0  weights = [ 2.84159495 -1.800999  ]
iteration =1  weights = [ 1.06730747 -1.5214152 ]
iteration =2  weights = [-0.24526872 -1.26965372]
iteration =3  weights = [-1.21628526 -1.04296816]
iteration =4  weights = [-1.93462624 -0.8388774 ]
iteration =5  weights = [-2.46604513 -0.65514151]
iteration =6  weights = [-2.85918416 -0.48973941]
iteration =7  weights = [-3.15002727 -0.34084836]
iteration =8  weights = [-3.36519423 -0.20682511]
iteration =9  weights = [-3.52437751 -0.08618864]
iteration =10  weights = [-3.64214504  0.02239546]
iteration =11  weights = [-3.72927373  0.12012933]
iteration =12  weights = [-3.79373606  0.20809568]
iteration =13  weights = [-3.84142986  0.28726959]
iteration =14  weights = [-3.87671822  0.35852909]
iteration =15  weights = [-3.90282885  0.4226647 ]
iteration =16  weights = [-3.92214958  0.48038818]
iteration =17  weights = [-3.93644688  0.53234027]
iteration =18  weights = [-3.94702757  0.57909777]
iteration =19  weights = [-3.95485844  0.6211799 ]
iteration =20  weights = [-3.96065473  0.65905402]
iteration =21  weights = [-3.96494558  0.69314082]
iteration =22  weights = [-3.96812246  0.72381895]
iteration =23  weights = [-3.970475    0.75142922]
iteration =24  weights = [-3.97221749  0.77627838]
iteration =25  weights = [-3.97350846  0.79864252]
iteration =26  weights = [-3.97446522  0.81877014]
iteration =27  weights = [-3.97517458  0.83688487]
iteration =28  weights = [-3.97570075  0.85318802]
iteration =29  weights = [-3.97609126  0.86786073]
iteration =30  weights = [-3.9763813   0.88106606]
iteration =31  weights = [-3.97659689  0.89295075]
iteration =32  weights = [-3.97675731  0.90364688]
iteration =33  weights = [-3.97687682  0.91327331]
iteration =34  weights = [-3.97696598  0.921937  ]
iteration =35  weights = [-3.97703261  0.92973426]
iteration =36  weights = [-3.97708252  0.93675171]
iteration =37  weights = [-3.97711998  0.94306736]
iteration =38  weights = [-3.97714819  0.94875139]
iteration =39  weights = [-3.97716951  0.95386696]
iteration =40  weights = [-3.97718567  0.95847092]
iteration =41  weights = [-3.97719799  0.96261445]
iteration =42  weights = [-3.97720743  0.96634359]
iteration =43  weights = [-3.9772147   0.96969977]
iteration =44  weights = [-3.97722035  0.97272031]
iteration =45  weights = [-3.97722476  0.97543876]
iteration =46  weights = [-3.97722823  0.97788535]
Final weights

[-3.97723099  0.98008725]
```

In [12]:
```
from sklearn.linear_model import LinearRegression
import pandas as pd
import random as rand #avoid namespace collisions

#Randomly sample 1% of our data
n = 1000000 #number of records in file
s = int(n*0.01) #desired sample size
filename = "LinearRegression.csv"
skip = sorted(rand.sample(xrange(n),n-s))
df = pd.read_csv(filename, skiprows=skip, header=None)

#Set up background stuff to plot mapreduce data
floor_x=min(df[1])
ceiling_x=max(df[1])
step_x=(ceiling_x-floor_x)/s
pred_x=np.arange(floor_x,ceiling_x,step_x)

def get_line(x,m,b):
    """Quick function to calculate Y values given X, slope, and interce
    return m*x+b

mr_intercept=weights[0]
mr_slope=weights[1]
pred_y_mr=get_line(pred_x,mr_slope,mr_intercept)

#Build Sklearn model
sklearn_model=LinearRegression()
sklearn_model.fit(df[1].reshape(-1,1),df[0])
sk_slope=sklearn_model.coef_
sk_intercept=sklearn_model.intercept_

pred_y_sk=get_line(pred_x,sk_slope,sk_intercept)

#Actually make the plots
plt.plot(df[1],df[0],'ro',alpha=0.1) #Plot data points, with alpha turn
plt.plot(pred_x,pred_y_mr,color="green", label="Mapreduce result") #Plo
plt.plot(pred_x,pred_y_sk, color="blue", label="Sklearn result") #Plot
plt.legend(loc="lower right")
plt.show()
```




It's a little difficult to see amid the visual noise of all the individual data points (even after reducing their opacity), but if we look closely we can see that our mapreduce and sklearn results are very similar. Mapreduce produced a slightly higher intercept and lower slope than the sklearn version of our calculation.

# HW6.6

## HW 6.6 Problem Statement

Improve this notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/0t7985e40fovlkw/EM-GMM-MapReduce%20Design%201.ipynb) 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 with similar graphics (or possibly better graphics)

**NOTE**: We had integrated our version of the notebook into this main notebook before the announcement was made that we were allowed to separate them. We've left it included here, since it also makes it possible to submit the entire assignment in a single document. It's also available separately here (https://github.com/nickhamlin/mids_261_homework/blob/master/HW6/MIDS-W261-2015-HWK-Week06-Hamlin-Thomas-Baek-Danish-GMM-EXAMPLE.ipynb)

# Expectation Step:

Given priors $\boldsymbol{x}^{(i)}$ , mean vector $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$ , calculate the probability of that each data point belongs to a class

$$p(\omega_k|\boldsymbol{x}^{(i)}, \theta) = \frac{\pi_k N(\boldsymbol{x}^{(i)}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K}[\pi_j N(\boldsymbol{x}^{(i)}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)]} \tag{1}$$

## Maximization Step:

Given probabilities, update priors, mean vector, and covariance matrix

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{n_k}\sum_{i=1}^{n} p(\omega_k|\boldsymbol{x}^{(i)}, \theta)\boldsymbol{x}^{(i)} \tag{2}$$

$$\hat{\boldsymbol{\Sigma}}_k = \frac{1}{n_k}\sum_{i=1}^{n} p(\omega_k|\boldsymbol{x}^{(i)}, \theta)(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(i)})(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(i)})^T \tag{3}$$

$$\hat{\pi}_k = \frac{n_k}{n} \text{ where } n_k = \sum_{i=1}^{n} p(\omega_k|\boldsymbol{x}^{(i)}, \theta) \tag{4}$$

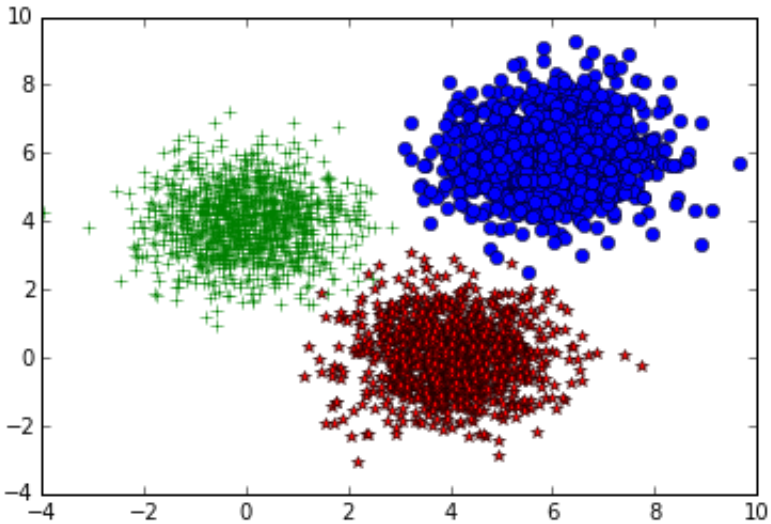
## Data Generation

In [3]:
```
import json

#Generate 3 clusters of 1000 points each
size1 = size2 = size3 = 1000

#First cluster is centered at (4,0)
samples1 = np.random.multivariate_normal([4, 0], [[1, 0],[0, 1]], size1
data = samples1

#Second cluster is centered at (6,6)
samples2 = np.random.multivariate_normal([6, 6], [[1, 0],[0, 1]], size2
data = np.append(data,samples2, axis=0)

#Third cluster is centered at (0,4)
samples3 = np.random.multivariate_normal([0, 4], [[1, 0],[0, 1]], size3
data = np.append(data,samples3, axis=0)

#Randomize ordering of data and save to file
data = data[np.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

As a starting point, it's useful to be able to see the actual clusters of data we're trying to "discover" below.

In [4]:
```
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

Here suppose we know there are 3 components

In [5]:

```
%%writefile mr_GMixEmInitialize.py
from __future__ import division
import json
from math import pi, sqrt, exp, pow
from random import sample

from numpy import mat, zeros, shape, random, array, zeros_like, dot, li
from mrjob.job import MRJob

class MrGMixEmInit(MRJob):
    DEFAULT_PROTOCOL = 'json'

    def __init__(self, *args, **kwargs):
        super(MrGMixEmInit, self).__init__(*args, **kwargs)
        self.numMappers = 1 #We only need one mapper to initialize our
        self.count = 0 #Keep track of how many example data points we'v
        self.jsonOut= None #Initialize somewhere to store our eventual


    def configure_options(self):
        """
        Set up infrastructure to enable us to pass important parameters
        when we call it
        """
        super(MrGMixEmInit, self).configure_options()

        #Number of clusters we want (defaults to 3)
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')

        #Path to wherever we want our output files to be saved
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt is s

    def mapper(self, _, xjIn):
        """
        Extract a random starting 2*K points from our input data and em
        """
        if self.count <= 2*self.options.k:
            self.count += 1
            #Emit records with a static key
            yield (1,xjIn)

    def reducer(self, key, xjIn):
        """
        Accumulate data points mapped to 0 from 1st mapper and
        pull out k of them as our starting point
        """

        #Randomly choose K starting centroids
        centroid_possibilities=[json.loads(xj) for xj in xjIn]
        centroids=sample(centroid_possibilities,self.options.k)
```

```
        centroids = sample(centroid_possibilities,self.options.k)

        #use the covariance of the selected centers as the starting gue

        #Calculate the mean of our starting centroids
        mean = array(centroids[0])
        for i in range(1,self.options.k):
            mean+=array(centroids[i])
        mean/=self.options.k #This is the starting point for equation 2

        #Accumulate the deviations
        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]+=xmm[i]*xmm[i]

        cov/=self.options.k

        #Calculate inverse of covariances - This is equivalent to equat
        covInv = linalg.inv(cov)
        cov_1 = [covInv.tolist()]*self.options.k

        #It might be useful to examine our intermediate outputs, so we'
        #their own file, just in case.
        jDebug = json.dumps([centroids,mean.tolist(),cov.tolist(),covIn
        debugPath = self.options.pathName + '/debug.txt'
        with open(debugPath,'w+') as f:
            f.write(jDebug)

        #We also need a starting guess at the phi's - prior probabiliti
        #initialize them all with the same number (1/k) to represent eq
        phi = zeros(self.options.k,dtype=float)
        for i in range(self.options.k):
            phi[i] = 1/self.options.k #These represent the starting val


        #Form output object
        outputList = [phi.tolist(), centroids, cov_1]
        self.jsonOut  = json.dumps(outputList)

        #Write new parameters to file
        fullPath = self.options.pathName + '/intermediateResults.txt'
        with open(fullPath,'w+') as f:
            f.write(self.jsonOut)

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 [6]:

```
%%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, linalg
import json

def gauss(x, mu, P_1):
    """
    Compute a Gaussian given:
    x: Vector of values
    mu: vector of means
    P_1: Inverted covariance matrix (which we calculate in the reducer)
    """
    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)

        #Load previous iteration results from disk
        fullPath = self.options.pathName + '/intermediateResults.txt'
        with open(fullPath,'r') as f:
            inputJson = f.read()
        inputList = json.loads(inputJson)

        #Store previous iteration components in their own variables
        self.phi = array(inputList[0])           #prior class probabili
        self.means = array(inputList[1])         #current means list
        self.cov_1 = array(inputList[2])         #inverse covariance ma

        #Create destinations for next iteration's results
        self.new_phi = zeros_like(self.phi)        #partial weighted su
        self.new_means = zeros_like(self.means)
        self.new_cov = zeros_like(self.cov_1)

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


    def configure_options(self):
        """
        Set up infrastructure to enable us to pass important parameters
        when we call it
        """
        super(MrGMixEm, self).configure_options()
```

```
        super(MRKmeans, self).configure_options()

        #Number of clusters we want (defaults to 3)
        self.add_passthrough_option(
            '--k', dest='k', default=3, type='int',
            help='k: number of densities in mixture')

        #Path to wherever we want our output files to be saved
        self.add_passthrough_option(
            '--pathName', dest='pathName', default="", type='str',
            help='pathName: pathname where intermediateResults.txt is s

    def mapper(self, key, val):
        """accumulate partial sums for each mapper"""
        x = array(json.loads(val)) #Load each record

        #Calculate probabilty of each point being from each
        #centroid (that is, estimate class assignments)
        #based on the mean vector and covariance matrix
        #This is the execution of the expecation step shown in Equation
        wtVect = zeros_like(self.phi) #This is an array of priors
        for i in range(self.options.k):
            wtVect[i] = self.phi[i]*gauss(x,self.means[i],self.cov_1[i]

        #Normalize elements of probability vector
        wtVect/=sum(wtVect)

        #Accumulate to update our est of probability densities.
        self.count += 1 #increment count
        #Accumulate weights for new prior estimates
        #These are partial sums that will be completed in the reducer
        self.new_phi+=wtVect
        for i in range(self.options.k):
            #Accumulate weighted x's for new mean calculation
            self.new_means[i]+=wtVect[i]*x
            #Accumulate weighted squares for new covariance 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
        #We don't yield anything here, since we do that via mapper_fina

    def mapper_final(self):
        """
        Aggregate mapper results
        (running totals for priors, centroids, and covariance matrix)
        into a list of lists and emit to the reducer
        """
        out = [self.count, (self.new_phi).tolist(), (self.new_means).to
        jOut = json.dumps(out)
        yield 1,jOut
```

```
    def reducer(self, key, xs):
        #Accumulate partial sums emitted by the mapper
        first = True
        #xs gives us a list of partial stats, including counts, posteri

        #Each stat is an array that stores information for K components
        for val in xs:
            temp = json.loads(val)
            if first:  #For the first record that arrives, initialize t
                totCount = temp[0]
                totPhi = array(temp[1])
                totMeans = array(temp[2])
                totCov = array(temp[3])
                first = False
            else:
                #For each subsequent record that arrives, update the cu
                totCount+=temp[0]
                totPhi+=array(temp[1])
                totMeans+=array(temp[2])
                totCov+=array(temp[3])

        #Finish calculation of new probability parameters. array divide
        newPhi = totPhi/totCount
        #Initialize these to something we already know to make sure we
        newMeans = totMeans
        newCov_1 = totCov

        #For each centroid, calculate new mean and covariance
        #This is the Maximization step shown in Equations 2-4 above.
        for i in range(self.options.k):
            newMeans[i,:] = totMeans[i,:]/totPhi[i]
            tempCov = totCov[i,:,:]/totPhi[i]
            #Invert the covariance matrix here to avoid doing a matrix
            #with every input data point.
            newCov_1[i,:,:] = linalg.inv(tempCov)

        #Compile our new parameters together and write to file
        #This enables us to pass them to the next iteration of the algo
        outputList = [newPhi.tolist(), newMeans.tolist(), newCov_1.toli
        jsonOut = json.dumps(outputList)
        fullPath = self.options.pathName + '/intermediateResults.txt'
        with open(fullPath,'w') as f:
            f.write(jsonOut)


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

```
Overwriting mr_GMixEmIterate.py
```

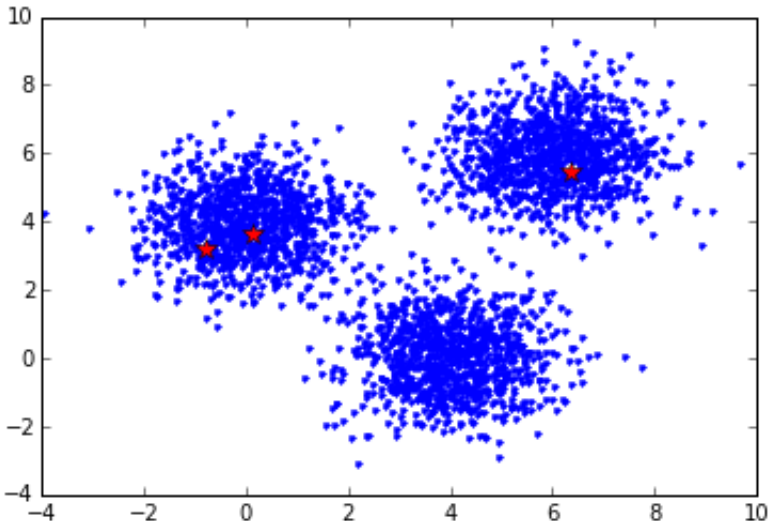
# Driver

In [9]:

```
from mr_GMixEmInitialize import MrGMixEmInit
from mr_GMixEmIterate import MrGMixEm
import json
import os
from math import sqrt

#Load original data points for plotting
filePath = 'data.txt'
original_data=[]
with open(filePath,'r') as input_data:
    for i in input_data.readlines():
        original_data.append(eval(i.strip()))
original_data=np.array(original_data)

def plot_iteration(means):
    """Plot both our original data points, as well as the current locat
    plt.plot(original_data[:, 0], original_data[:, 1],'.', color = 'blu
    plt.plot(means[0][0], means[0][1],'*',markersize =10,color = 'red')
    plt.plot(means[1][0], means[1][1],'*',markersize =10,color = 'red')
    plt.plot(means[2][0], means[2][1],'*',markersize =10,color = 'red')
    plt.show()

def dist(x,y):
    """Calculate the 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)


#INITIALIZE STARTING CENTROIDS

#If you don't pass the current dir as the --pathName, all the files wil
#temp folder and not be accessible to the driver function, and everythi
#no apparent reason.
pwd=os.getcwd()
mrJob = MrGMixEmInit(args=[filePath,'--no-strict-protocols','--pathName
with mrJob.make_runner() as runner:
    runner.run()

#pull out the centroid values to compare with values after one iteratic
emPath = "intermediateResults.txt"
with open(emPath) as fileIn:
    paramJson = fileIn.read()

#Initialize the "previous step delta" to something really high to start
#this will get updated with every iteration of the algorithm, which we'
#to have converged once delta falls below a certain threshold.
delta = 10
iter_num = 0 #Which iteration are we on?

#BEGIN ITERATIONS
while delta > 0.02:
```

```
    print "Iteration " + str(iter_num)
    iter_num = iter_num + 1
    #Parse old centroid values from file
    oldMeans = json.loads(paramJson)[1]

    #Run one iteration of our MapReduce job
    mrJob2 = MrGMixEm(args=[filePath,'--no-strict-protocols','--pathNam
    with mrJob2.make_runner() as runner:
        runner.run()

    #Compare load new centroids into memory
    with open(emPath) as fileIn:
        paramJson = fileIn.read()

    newParam = json.loads(paramJson)
    k_means = len(newParam[1])
    newMeans = newParam[1]

    #Calculate how much centroids have moved in this iteration
    delta = 0.0
    for i in range(k_means):
        delta += dist(newMeans[i],oldMeans[i])

    #Display mean locations and plot for this iteration
    for n,mean in enumerate(oldMeans):
        print 'Mean {0}: {1}'.format(str(n),str(mean))
    plot_iteration(oldMeans)
    print "----------------------------------------"

#Display mean locations and plot for final iteration
print "Iteration " + str(iter_num) + ": MODEL HAS CONVERGED!"
for n,mean in enumerate(newMeans):
    print 'Mean {0}: {1}'.format(str(n),str(mean))
plot_iteration(newMeans)
```

```
Iteration 0
Mean 0: [-0.7711333296788728, 3.2043728958077144]
Mean 1: [0.13937949139759165, 3.654284598341057]
Mean 2: [6.370756548444613, 5.464212291594231]
```



# HW6.7

## HW 6.7 Problem Statement

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

As a unit test use the dataset in the following slides:

https://www.dropbox.com/s/maoj9jidxj1xf5l/MIDS-Live-Lecture-06-EM-Bernouilli-MM-Systems-Test.pdf?dl=0 (https://www.dropbox.com/s/maoj9jidxj1xf5l/MIDS-Live-Lecture-06-EM-Bernouilli-MM-Systems-Test.pdf?dl=0)

Cross-check that you get the same cluster assignments and cluster Bernouilli models as presented in the slides after 25 iterations. Dont forget the smoothing.

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.

## HW 6.7 Unit Test Setup

Unfortunately, our unit test data is structured differently than our main data. Therefore, to facilitate testing, we can manually translate the unit test data into the term-document matrix format used in the main tweet corpus.

In [32]:
```
# HW 6.7 - Example Unit Test data
# This version isn't actually used, but is helpful for visual spot chec
# to make sure that the test matrix is created properly.
%%writefile unittest.txt
id,cluster,word_count,hot,chocolate,cocoa,beans,ghana,africa,harvest,bu
1,1,4,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,hot chocolate cocoa beans
2,1,3,0,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,cocoa ghana africa
3,1,3,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,beans harvest ghana
4,1,2,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,cocoa butter
5,1,2,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,butter truffles
6,1,2,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,sweet chocolate
7,1,2,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,sweet sugar
8,1,3,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,sugar cane brazil
9,1,3,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,sweet sugar beet
10,1,3,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,sweet cake icing
11,1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,cake black forest
```

```
Overwriting unittest.txt
```

This is the same matrix as above, but without the additional visual cues. This is what we'll use in our testing. In this format, we don't much care what the actual words are, just where they co-occur. This enables us to use a words position in the feature vector as it's index throughout the process.

In [2]:
```
%%writefile unittest.txt
1,None,4,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2,None,3,0,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0
3,None,3,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4,None,2,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0
5,None,2,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0
6,1,2,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0
7,0,2,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0
8,None,3,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0
9,None,3,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0
10,None,3,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0
11,None,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1
```

```
Overwriting unittest.txt
```

## HW 6.7 Unit Test Implementation

Here we create an MRJob definition that accepts two files containing the current list of class prior probabilities (alpha) and word-class conditional probabilies (q). We'll need to initialize these when we run the job (more on this below).

In [8]:

```
%%writefile mrbmm.py
from __future__ import division
from math import log, exp
import json

import numpy as np
from scipy.misc import logsumexp

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


class MRBMM(MRJob):

    def __init__(self, *args, **kwargs):
        super(MRBMM, self).__init__(*args, **kwargs)
        #Initializing these values here makes them available to the cla
        self.k = 0 #Number of clusters to create
        self.r=[] #List of document class assignments
        self.alphas=[] #List of priors
        self.epsilon = 0.0001 #smoothing
        self.q={} #Dictionary of word-class conditional probabilities

    def steps(self):
        return [
            MRStep(
                mapper_init=self.mapper_init
                ,mapper=self.mapper
                ,mapper_final=self.mapper_final
                ,reducer_init=self.reducer_init
                ,reducer=self.reducer
                ,reducer_final=self.reducer_final
            )
        ]

    def mapper_init(self):
        """
        Load existing model parameters into memory as a list with len=k
        """
        self.alphas=[map(float,s.split('\n')[0].split(',')) for s in op
        self.k=len(self.alphas)

        with open('q.txt','r') as f:
            self.q = json.load(f)
        self.r={}


    def mapper(self,_,line):
        """Given each document, emit pairs of the format
        Key: (current cluster assignment, word index)
        Value: (word presence in document,document class probabilities)
        line=line.split(',')
        line_id,cluster,total_words=int(line[0]),(line[1]),float(line[2
        probs=[0.0]*self.k #for each doc, what are the probabilies per
```

```
        probs [k] before #for each doc, what are the probabilities per
        line_id-=1 #the file is one-indexed, but python is zero-indexed
        features=(map(float,line[3:])) #Convert point to floats

        #This gives us an index that we can easily zip with the feature
        fake_words=[i for i in range(len(features))]
        #Update R
        for word, count in zip(fake_words, features):
            for k in range(self.k):
                if count>0:
                    update_value=self.q[str(word)][str(k)]
                else:
                    update_value=1-self.q[str(word)][str(k)]
                probs[k] += log(update_value + self.epsilon)

        #Calculate equation 1A from the slide
        complete_log=[log(alpha)+prob for alpha, prob in zip(self.alpha
        log_sum=logsumexp(complete_log) #This is the "denominator"
        log_output=[j-log_sum for j in complete_log] #subtracting logs
        self.r[line_id] = [exp(i) for i in log_output]

        # Now, emit situations where a word appears in a class - to be
        for word, count in zip(self.q.keys(),features):
            for k in range(self.k):
                if count>0:
                    #print (k,word), (1,self.r[line_id][k])
                    yield (k,word), (1,self.r[line_id][k])


        #Finally, emit class prob for each doc -> to be aggregated into
        for k in range(self.k):
            #print (k,'*'),(1,self.r[line_id][k])
            yield (k,'*'),(1,self.r[line_id][k])
        #print ""

    def mapper_final(self):
        """Print results of expectation step for each document so we ca
        for i in self.r.iteritems():
            #print i[0],"{0:.2f},{1:0.2f}".format(i[1][0],i[1][1])
            print i[0],"{0:.2f},{1:0.2f},{2:0.2f},{3:0.2f}".format(i[1]

        print ""

    def reducer_init(self):
        """Reload class priors and word-class conditional probabilities
        self.alphas=[map(float,s.split('\n')[0].split(',')) for s in op
        self.k=len(self.alphas)

        with open('q.txt','r') as f:
            self.q = json.load(f)

    def reducer(self, key, value):
        """Aggregate pairs emitted by the mapper and calculate
        updated class priors and word-class conditional probabilities""
        k, word = key #k is the cluster for the pair, not to be confuse
```

```
        # aggregate new alphas
        if word == '*':
            n=0
            total_r=0
            for count, r in value:
                n += count
                total_r += r
            self.alphas[k] = total_r/n
            return

        #recalculate q as words come in
        total_r=0
        total_r_count=0
        for count, r in value:
            total_r += r
            total_r_count += r*count

        #Add Laplace Smoothing when we calculate Q
        self.q[unicode(word)][unicode(k)] = (total_r_count+self.epsilon
        #print self.q
        #print ""

    def reducer_final(self):
        """Make sure new parameters persist on disk to the next iterati
        #Wipe old files
        open('alphas.txt','w').close()
        open('q.txt','w').close()

        #Write new files
        with open('alphas.txt','w+') as f:
            f.writelines(','.join(str(j) for j in self.alphas))

        with open('q.txt','w+') as f:
            json.dump(self.q,f)

        print "=================="
        print "ALPHAS"
        print self.alphas
        print ""

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

```
Overwriting mrbmm.py
```

In [10]:
```
### HW 6.7 - Unit Test Driver Code
from __future__ import division
import json

from numpy import random

from mrbmm import MRBMM


def run_bmm_mrjob(alphas,q,k,iterations,source):

    #Set up job and save model parameters to file
    mr_job=MRBMM(args=[source,'--file', 'alphas.txt','--file', 'q.txt']
    with open('alphas.txt','w+') as f:
        f.writelines(','.join(str(j) for j in alphas))

    with open('q.txt','w+') as f:
        json.dump(q,f)

    i=0 #Track which iteration we're on
    while(1):
        print "RESULTS FOR ITERATION "+str(i)
        output=[] #Initialze destination for our final results
        with mr_job.make_runner() as runner:
            runner.run() #stream output
        i+=1
        print ""
        if i==iterations:
            break
```

## HW 6.7 - Initializing and Running the Unit Test

It's probably not how we'd implement this in a large scale situation, but since our data is relatively small, we can use in-memory operations to initialize our starting values of alpha and q. First, we define two helper functions to compute an initial posting list and q vector, which we'll then call when we set up th

In [4]:
```
#HW 6.7 - Define helper functions for initializing unit test

def word_lookup(data):
    """Create an initial postings list for our small dataset"""
    results = {}
    for i in range(len(data)):
        if r[0][i] is not None:
            for word,value in enumerate(data[i]):
                if word not in results and value>0:
                    results[word] = [i]
                elif value>0:
                    if i not in results[word]:
                        results[word].append(i)
    return results


def initialize_q(num_clusters,data,word_lookup):
    """Initialize the vector of word-class conditional probabilities fo
    q={}
    for k in range(num_clusters): #For each cluster
        for idx,i in enumerate(data): #for ech doc we see
            features=i[3:]
            for word,value in enumerate(features):
                if word not in q.keys():
                    try:
                        doc_word_count=sum([r[k][j] for j in word_looku
                        corpus_word_count=sum([x for x in r[k] if x is
                        q[word] = {k: doc_word_count/corpus_word_count}
                    except KeyError:
                        q[word] = {k: 0.0}
                else:
                    try:
                        doc_word_count=sum([r[k][j] for j in word_looku
                        corpus_word_count=sum([x for x in r[k] if x is
                        q[word][k] = doc_word_count/corpus_word_count
                    except KeyError:
                        q[word][k] = 0.0

    return q
```

In [43]:
```
#### HW 6.7 - INITIALIZE UNIT TEST AND RUN ######
from __future__ import division
import numpy as np

#Initialize starting alphas and r
data=[map(eval,s.split('\n')[0].split(',')) for s in open('unittest.txt
k=2
vocab_size=len(data[0])-3
starting_alphas=[0]*k
one_r=[None]*len(data)
#This contains two lists, one for each class.  Each lists has one eleme
r=[]
for i in range(k):
    r.append(one_r[:])

#Manually set cluster assignments like the example in the book
r[0][5] = 1.0
r[1][6] = 1.0
r[0][6] = 0.0
r[1][5] = 0.0

#Calculate initial class probablities
for i in data:
    if i[1] is not None:
        test=i[1]
        starting_alphas[test]+=1
starting_alphas=[i/sum(starting_alphas) for i in starting_alphas]

#Calculate initial word-class conditional probabilities
features=[j[3:] for j in data]
word_lookup_results=word_lookup(features)
q=initialize_q(k,data,word_lookup_results)

#Run the job!
print "STARTING JOB:"
iterations=25
source='unittest.txt'
run_bmm_mrjob(starting_alphas,q,k,iterations,source) #Run the jobs
print "DONE"
```

```
-protocols)
WARNING:mrjob.runner:

STARTING JOB:
RESULTS FOR ITERATION 0
0 1.00,0.00
1 0.50,0.50
2 0.50,0.50
3 0.50,0.50
4 0.50,0.50
5 1.00,0.00
6 0.00,1.00
7 0.00,1.00
8 0.00,1.00
```

```
9 0.50,0.50
10 0.50,0.50

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

WARNING:mrjob.runner:
```

## HW 6.7 Unit Test Discussion

As shown above, our model converges to the same results shown in the slides after 25 iterations. That is, it fixes the manual disturbances we made to documents 6 and 7, and settles on a final hard assignment (which is interesting, since Bernoulli Mixture Models don't always do this).

## HW 6.7 Full Test Setup

Having passed the unit test, we can now rerun our job on the full dataset. Ideally, we'd configure our code to be able to easily switch back and forth. In the interest of time, instead we've used recycled but slightly modified versions of the unit test code below. Also, in practice we'd rewrite the initialization process into a separate mapreduce job (as shown in 6.6), but to save implementation time here we've recycled the in-memory initializer from the unit test here as well.

In [34]:

```
# HW 6.7 - Full Test MRjob setup
%%writefile mrbmm.py
from __future__ import division
from math import log, exp
import json

import numpy as np
from scipy.misc import logsumexp

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


class MRBMM(MRJob):

    def __init__(self, *args, **kwargs):
        super(MRBMM, self).__init__(*args, **kwargs)
        #Initializing these values here makes them available to the cla
        self.k = 0 #Number of clusters to create
        self.r=[] #List of document class assignments
        self.alphas=[] #List of priors
        self.epsilon = 0.0001 #smoothing
        self.q={} #Dictionary of word-class conditional probabilities

    def steps(self):
        return [
            MRStep(
                mapper_init=self.mapper_init
                ,mapper=self.mapper
                ,mapper_final=self.mapper_final
                ,reducer_init=self.reducer_init
                ,reducer=self.reducer
                ,reducer_final=self.reducer_final
            )
        ]

    def mapper_init(self):
        """
        Load existing model parameters into memory as a list with len=k
        """
        self.alphas=[map(float,s.split('\n')[0].split(',')) for s in op
        self.k=len(self.alphas)

        with open('q.txt','r') as f:
            self.q = json.load(f)
        self.r={}


    def mapper(self,_,line):

        line=line.split(',')
        line_id,real_cluster,total_words=int(line[0]),int(line[1]),floa
        probs=[0.0]*self.k #for each doc, what are the probabilies per
        line_id-=1 #the file is one-indexed, but python is zero-indexed
```

```
        line_id = i #the line is the indexed, but python is zero indexed
        features=(map(float,line[3:])) #Convert point to floats

        #This gives us an index that we can easily zip with the feature
        fake_words=[i for i in range(len(features))]
        #Update R
        for word, count in zip(fake_words, features):
            for k in range(self.k):
                if count>0:
                    update_value=self.q[str(word)][str(k)]
                else:
                    update_value=1-self.q[str(word)][str(k)]
                probs[k] += log(update_value + self.epsilon)

        #Calculate equation 1A from the slide
        complete_log=[log(alpha)+prob for alpha, prob in zip(self.alpha
        log_sum=logsumexp(complete_log) #This is the "denominator"
        log_output=[j-log_sum for j in complete_log] #subtracting logs
        self.r[line_id] = [exp(i) for i in log_output]

        # Now, emit situations where a word appears in a class - to be
        for word, count in zip(self.q.keys(),features):
            for k in range(self.k):
                if count>0:
                    #print (k,word), (1,self.r[line_id][k])
                    yield (k,word), (1,self.r[line_id][k])


        #Finally, emit class prob for each doc -> to be aggregated intc
        for k in range(self.k):
            #print (k,'*'),(1,self.r[line_id][k])
            yield (k,'*'),(1,self.r[line_id][k])
        #print ""

    def mapper_final(self):
        """Print intermediate results (optional, but useful for debuggi
        for i in self.r.iteritems():
            #print i[0],"{0:.2f},{1:0.2f}".format(i[1][0],i[1][1])
            #print i[0],"{0:.2f},{1:0.2f},{2:0.2f},{3:0.2f}".format(i[1
            pass
        print ""

    def reducer_init(self):
        self.alphas=[map(float,s.split('\n')[0].split(',')) for s in op
        self.k=len(self.alphas)

        with open('q.txt','r') as f:
            self.q = json.load(f)

    def reducer(self, key, value):
        k, word = key #k is the cluster for the pair, not to be confuse
        # aggregate new alphas
        if word == '*':
            n=0
            total r=0
```

```
            for count, r in value:
                n += count
                total_r += r
            self.alphas[k] = total_r/n
            return

        #recalculate q as words come in
        total_r=0
        total_r_count=0
        for count, r in value:
            total_r += r
            total_r_count += r*count

        #Add Laplace Smoothing when we calculate Q
        self.q[unicode(word)][unicode(k)] = (total_r_count+self.epsilon
        #print self.q
        #print ""

    def reducer_final(self):
        #Wipe old files
        open('alphas.txt','w').close()
        open('q.txt','w').close()

        #Write new files
        with open('alphas.txt','w+') as f:
            f.writelines(','.join(str(j) for j in self.alphas))

        with open('q.txt','w+') as f:
            json.dump(self.q,f)

        print "=================="
        print "ALPHAS"
        print self.alphas
        print ""

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

```
Overwriting mrbmm.py
```

In [35]:
```
### HW 6.7 - Full Test Driver Code
from __future__ import division
import json

from numpy import random

from mrbmm import MRBMM


def run_bmm_mrjob(alphas,q,k,iterations,source):

    #Set up job and save model parameters to file
    mr_job=MRBMM(args=[source,'--file', 'alphas.txt','--file', 'q.txt']
    with open('alphas.txt','w+') as f:
        f.writelines(','.join(str(j) for j in alphas))

    with open('q.txt','w+') as f:
        json.dump(q,f)

    i=0 #Track which iteration we're on
    while(1):
        print "RESULTS FOR ITERATION "+str(i)
        output=[] #Initialze destination for our final results
        with mr_job.make_runner() as runner:
            runner.run() #stream output
        i+=1
        print ""
        if i==iterations:
            break
```

In [5]:
```
#### HW 6.7 - Initialize Full Test ######
# Since the data is larger here and we're still using the in-memory int
# We separate this step out from the actual job execution for the full
from __future__ import division
import numpy as np

#Initialize starting alphas and r
k=4
full_data=[map(eval,s.split('\n')[0].split(',')) for s in open('topUser
print "data loaded"
full_vocab_size=len(full_data[0])-3
full_starting_alphas=[0]*k
one_r=[None]*len(full_data)
print "basic stats complete"
#This contains two lists, one for each class.  Each lists has one eleme
r=[]
for i in range(k):
    r.append(one_r[:])
print "r calculated"
for i in full_data:
    if i[1] is not None:
        test=i[1]
        full_starting_alphas[test]+=1
full_starting_alphas=[i/sum(full_starting_alphas) for i in full_startin
print "full starting alphas done:"
print full_starting_alphas
full_features=[j[3:] for j in full_data]
print "full feature matrix done"
full_word_lookup_results=word_lookup(full_features)
print "full posting list done"
full_q=initialize_q(k,full_data,full_word_lookup_results)
print "full q vector computed"
print "INITIALIZING DONE"
```

```
data loaded
basic stats complete
r calculated
full starting alphas done
full feature matrix done
full posting list done
full q vector computed
INITIALIZING DONE
```

In [36]:
```
# HW 6.7 - Running the main job to calculate parameters
source='topUsers_Apr-Jul_2014_1000-words.txt'
iterations=6
run_bmm_mrjob(full_starting_alphas,full_q,k,iterations,source)
```

```
WARNING:mrjob.runner:
WARNING:mrjob.runner:PLEASE NOTE: Starting in mrjob v0.5.0, protocol
s will be strict by default. It's recommended you run your job with
--strict-protocols or set up mrjob.conf as described at https://pyth
onhosted.org/mrjob/whats-new.html#ready-for-strict-protocols (http
s://pythonhosted.org/mrjob/whats-new.html#ready-for-strict-protocol
s)
WARNING:mrjob.runner:

RESULTS FOR ITERATION 0

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

WARNING:mrjob.runner:
WARNING:mrjob.runner:PLEASE NOTE: Starting in mrjob v0.5.0, protocol
s will be strict by default. It's recommended you run your job with
--strict-protocols or set up mrjob.conf as described at https://pyth
onhosted.org/mrjob/whats-new.html#ready-for-strict-protocols (http
s://pythonhosted.org/mrjob/whats-new.html#ready-for-strict-protocol
s)
WARNING:mrjob.runner:


ALPHAS
[0.7519999999997505, 0.09099999999998093, 0.0539999999999789, 0.1029
9999999999468]


RESULTS FOR ITERATION 1

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

WARNING:mrjob.runner:
WARNING:mrjob.runner:PLEASE NOTE: Starting in mrjob v0.5.0, protocol
s will be strict by default. It's recommended you run your job with
--strict-protocols or set up mrjob.conf as described at https://pyth
onhosted.org/mrjob/whats-new.html#ready-for-strict-protocols (http
s://pythonhosted.org/mrjob/whats-new.html#ready-for-strict-protocol
s)
WARNING:mrjob.runner:


ALPHAS
[4.754667905962165e-44, 0.0018355696308464497, 0.997176386888497, 0.
0009880434806567079]


RESULTS FOR ITERATION 2

==================
```

```
WARNING:mrjob.runner:
WARNING:mrjob.runner:PLEASE NOTE: Starting in mrjob v0.5.0, protocol
s will be strict by default. It's recommended you run your job with
--strict-protocols or set up mrjob.conf as described at https://pyth
onhosted.org/mrjob/whats-new.html#ready-for-strict-protocols (http
s://pythonhosted.org/mrjob/whats-new.html#ready-for-strict-protocol
s)
WARNING:mrjob.runner:


ALPHAS
[0.001999609399494976, 0.2781197580119392, 0.6307326584863198, 0.089
147974102246]


RESULTS FOR ITERATION 3

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

WARNING:mrjob.runner:
WARNING:mrjob.runner:PLEASE NOTE: Starting in mrjob v0.5.0, protocol
s will be strict by default. It's recommended you run your job with
--strict-protocols or set up mrjob.conf as described at https://pyth
onhosted.org/mrjob/whats-new.html#ready-for-strict-protocols (http
s://pythonhosted.org/mrjob/whats-new.html#ready-for-strict-protocol
s)
WARNING:mrjob.runner:


ALPHAS
[0.05930211014487631, 0.883, 1.6998402464064356e-41, 0.0576978898551
23585]


RESULTS FOR ITERATION 4

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

WARNING:mrjob.runner:
WARNING:mrjob.runner:PLEASE NOTE: Starting in mrjob v0.5.0, protocol
s will be strict by default. It's recommended you run your job with
--strict-protocols or set up mrjob.conf as described at https://pyth
onhosted.org/mrjob/whats-new.html#ready-for-strict-protocols (http
s://pythonhosted.org/mrjob/whats-new.html#ready-for-strict-protocol
s)
WARNING:mrjob.runner:
```

```
ALPHAS
[0.04847784096975346, 0.000999999992100129, 0.03552047768153301, 0.9
150016813566136]


RESULTS FOR ITERATION 5

==================
ALPHAS
[1.512954231175285e-133, 0.18422507112720848, 2.338279533567523e-44,
0.8157749288727915]
```

## HW 6.7 - Evaluating our results

Unlike in our unit test, here we need an extra closing job to evaluate our results. This job takes the final parameters produced above and makes one final pass through the dataset to calculate the most likely cluster for each document given the model parameters. The predicted cluster can then be compared to the actual cluster in the reducer. So we can best see what's going on here, we also output the intermediate values for each cluster, including the vector of document-class probabilities and the predicted and true classes.

In [38]:

```
# HW 6.7 - Final job for evaluating results
%%writefile mrbmm_final.py
from __future__ import division
from math import log, exp
import json

import numpy as np
from scipy.misc import logsumexp

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


class MRBMM_Final(MRJob):

    def __init__(self, *args, **kwargs):
        super(MRBMM_Final, self).__init__(*args, **kwargs)
        #Initializing these values here makes them available to the cla
        self.k = 0 #Number of clusters to create
        self.r=[] #List of document class assignments
        self.alphas=[] #List of priors
        self.epsilon = 0.0001 #smoothing
        self.q={} #Dictionary of word-class conditional probabilities

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

    def mapper_init(self):
        """
        Load existing model parameters into memory as a list with len=k
        """
        self.alphas=[map(float,s.split('\n')[0].split(',')) for s in op
        self.k=len(self.alphas)

        with open('q.txt','r') as f:
            self.q = json.load(f)
        self.r={}


    def mapper(self,_,line):

        line=line.split(',')
        line_id,real_cluster,total_words=int(line[0]),int(line[1]),floa
        probs=[0.0]*self.k #for each doc, what are the probabilies per
        #line_id-=1 #the file is one-indexed, but python is zero-indexe
        features=(map(float,line[3:])) #Convert point to floats
```

```
        features=map(float,line[1:])
        #This gives us an index that we can easily zip with the feature
        fake_words=[i for i in range(len(features))]
        #Update R
        for word, count in zip(fake_words, features):
            for k in range(self.k):
                if count>0:
                    update_value=self.q[str(word)][str(k)]
                else:
                    update_value=1-self.q[str(word)][str(k)]
                probs[k] += log(update_value + self.epsilon)

        #Calculate equation 1A from the slide
        complete_log=[log(alpha)+prob for alpha, prob in zip(self.alpha
        log_sum=logsumexp(complete_log) #This is the "denominator"
        log_output=[j-log_sum for j in complete_log] #subtracting logs
        self.r[line_id] = [exp(i) for i in log_output]

        #Find the cluster id of the most likely cluster - use this as c
        predicted_cluster=np.argmax(self.r[line_id])
        print self.r[line_id]
        print int(real_cluster),int(predicted_cluster)
        print ""
        yield int(real_cluster),int(predicted_cluster)

    def reducer_init(self):
        self.right=0
        self.wrong=0
        self.total=0

    def reducer(self, key, value):
        for i in value:
            self.total+=1
            if i==key:
                self.right+=1
            else:
                self.wrong+=1

    def reducer_final(self):
        print "OVERALL ACCURACY: {0}".format(self.right/self.total)

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

```
Overwriting mrbmm_final.py
```

In [39]:
```
# HW 6.7 - Run final results
from mrbmm_final import MRBMM_Final
def run_bmm_final_mrjob(source):

    #Set up job and save model parameters to file
    mr_job=MRBMM_Final(args=[source,'--file', 'alphas.txt','--file', 'q

    print "RESULTS FOR ITERATION "
    output=[] #Initialze destination for our final results
    with mr_job.make_runner() as runner:
        runner.run() #stream output

source='topUsers_Apr-Jul_2014_1000-words.txt'
run_bmm_final_mrjob(source)
```

```
[6.470373663451782e-90, 0.0, 1.0, 0.0]
2 2

[6.4703736634510465e-90, 0.0, 1.0, 0.0]
2 2

[0.0, 1.0, 0.0, 0.0]
0 1

[0.0, 1.0, 0.0, 0.0]
3 1

[0.0, 1.0, 0.0, 9.886164325311807e-279]
3 1

[0.0, 1.0, 0.0, 2.6717375297969583e-273]
0 1

[0.0, 1.0, 0.0, 3.4666902413370546e-308]
```

## HW 6.7 - Full Results Discussion

Ultimately our model did not perform particularly well. While we haven't included the complete results of the same dataset run through k-means for brevity (they're available in full here (https://github.com/nickhamlin/mids_261_homework/blob/master/HW4/MIDS-W261-2015-HWK-Week04-Hamlin-Thomas-Baek-Danish.ipynb)), it's clear that our k-means implementation was more effective.

This doesn't match what we'd have expected. Theoretically, BMM should have produced a better result because it's not making hard choices about which tweet belongs in which cluster. We know that this is especially relevant in this particular dataset because it's often difficult to distinguish between the "robot" and "cyborg" voices when they week.

The fact that BMM performs as poorly as it does here suggests a bug in the implementation. We're particularly suspicious of the calculation of the updated class priors based on the results for each iteration shown above, which indicate that the highest probabilty cluster tends to swing dramatically from iteration to iteration. That said, the algorithm worked properly on the unit test, so it's also possible that it needs more iterations to fully converge or the unit test dataset was too small to reveal the bug.

## End of Submission