HW 6.0

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

Mathematical optimization is the selection of a best element element (with regard to some criteria) from some set of available alternatives.

One optimization problem that I worked on was the following: on some days, we're missing the price of a bond or a similar financial instrument, but we do have prices of similar ones. What is the most likely price today?

I'm actually currently working on this problem.

HW 6.1

- 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 $[-6, 6]$

If f is the function that we're trying to optimize, the FOC is that $f'(x) = 0$ at maximum and minimum, and the SOC is that $f''(x) < 0$, then x is a local maximum, and if $f''(x) > 0$, then x is a local minimum.

- 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.

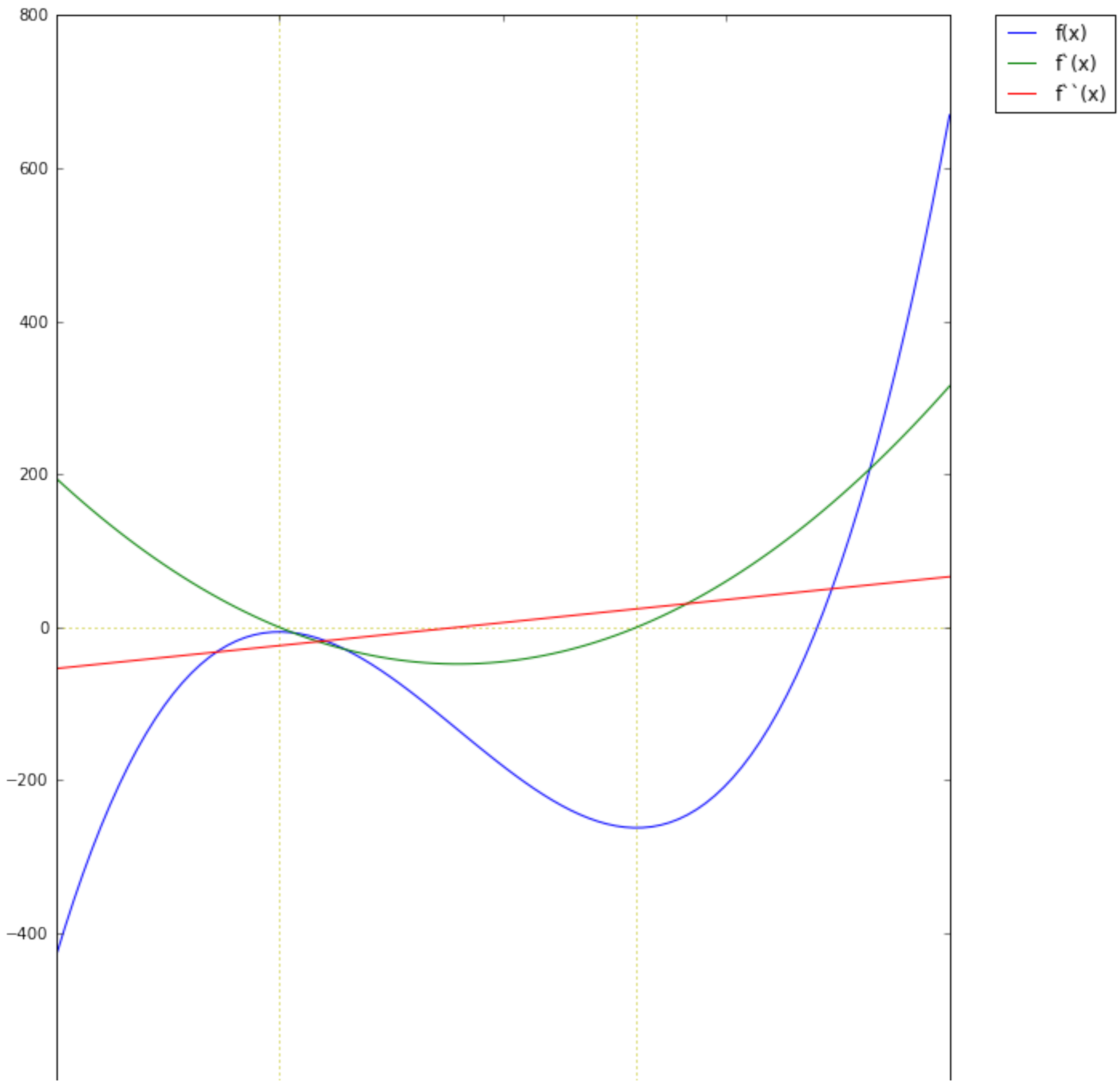
In [19]:
```
%matplotlib inline
from matplotlib import pyplot as py
import numpy as np
```

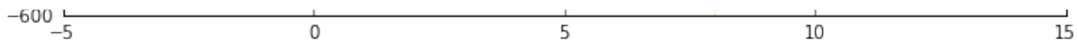
```
plot = py.figure(figsize = (10,12))

x = np.linspace(-5, 15, 100)
f = x**3 - 12*x**2 - 6
f1 = 3*x**2 - 24*x
f2 = 6*x - 24

py.plot([0,0], [-600, 800], color='y', linestyle = 'dotted')
py.plot([8,8], [-600, 800], color='y', linestyle = 'dotted')
py.plot([-5, 15], [0,0], color='y', linestyle = 'dotted')
py.plot(x, f, label = 'f(x)')
py.plot(x, f1, label = 'f`(x)')
py.plot(x, f2, label = 'f``(x)')
py.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

Out[19]: <matplotlib.legend.Legend at 0x10af36ad0>





- 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?

For unconstrained multi-variate optimization, the FOC is that the gradient function is 0, and the Hessian is positive definite. The Hessian matrix is the matrix that describes the second-order derivatives of the function in all dimensions: If the function has variables x_i, x_j, then $H_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

HW 6.2

- 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)

With to the Newton-Raphson method, when searching for the solution of $f(x)$, we calculate the $n + 1$-th approximation ($x_{t=n+1}$) from the n-th approximation ($x_{t=n}$) as

$$x_{t=n+1} = x_{t=n} - \frac{f(x)}{f'(x)},$$

so, if we have $x_{t=1} = 1$, then, for our function $f(x)$, which has $f'(x) = 3x^2 + 2x$,

$$\begin{aligned} x_{t=2} &= x_{t=1} - \frac{1^3 + 2 \times 1 - 4}{3 \times 1 + 2 \times 1} \\ &= 1 - \frac{-1}{5} \\ &= 1.2 \end{aligned}$$

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?

What makes an optimization problem convex is finding the minimum for a convex function on a convex set.

The first order necessary conditions in convex optimization is:

Supose f is differentiable (that is, its gradient ∇f exists at each point in its domain $\mathcal{S}$, which is open). Then f is convex if and only if its domain $\mathcal{S}$ is convex and

$f(y) \geq f(x) + \nabla f(y)^T(y - x)$ for all $x, y \in \mathcal{S}$.

The second order necessary condition for convex optimization is:

$f(y) \geq f(x) + \nabla f(y)^T \bullet (y - x)$

for all $x, y \in \mathbb{R}^n$.

Fill in the BLANKS here:

Convex minimization, a subfield of optimization, studies the problem of minimizing convex functions over convex sets. The convex 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:

$$0.5 * \sum_{i=1}^{n}(weight_i * (W * X_i - y_i)^2)$$

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.

If the input variables X are in an n-dimensional space, then we can define this objective function as

$J(X) = 0.5 \times \sum_{i=1}^{n}(w_i(Wx_i - y_i)^2$.

The gradient of this objective function is the sum of all partial derivatives of these components. That is,

$$\begin{aligned}\nabla J(X) &= \nabla\left(0.5 \times \sum_{i=1}^{n}(w_i(Wx_i - y_i)^2\right)\\ &= \sum_{i=1}^{n} 0.5 \frac{\partial}{\partial W}(w_i(Wx_i - y_i)^2\\ &= \sum_{i=1}^{n} \frac{\partial}{\partial W}(w_i(Wx_i - y_i)\end{aligned}$$

HW 6.5

Write a MapReduce job in MRJob to do the training at scale of a weighted OLS model using gradient descent.
Generate one million datapoints just like in the following notebook:
http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipyn
http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipyr

Weight each example as follows:

$$weight(x) = abs(1/x)$$

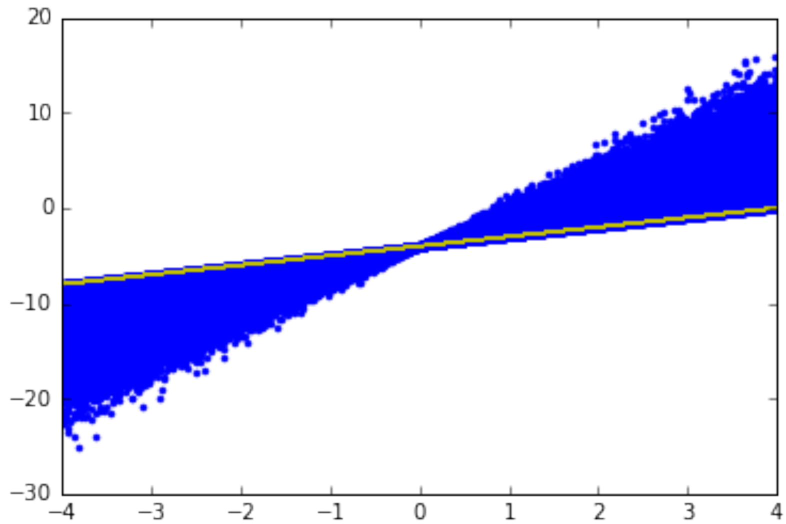
Sample 1% of the data in MapReduce and use the sampled dataset to train a (weighted if available in SciKit-Learn) linear regression model locally using SciKit-Learn (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html))

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

In [4]:
```
%matplotlib inline
import numpy as np
import pylab
size = 1000000
x = np.random.uniform(-4, 4, size)
y = x-4
for i in range(len(x)):
    if x[i] < 0:
        y[i] -= abs(np.random.normal(0,abs(x[i]),1))
    else:
        y[i] += abs(np.random.normal(0,abs(x[i]),1))

data = zip(y,x)
np.savetxt('LinearRegression.csv',data,delimiter = ",")
```

In [7]:
```
pylab.plot(x, y,'.')
pylab.plot(x,x-4,color='y',linestyle='dotted')
pylab.show()
```



In [8]:
```
%writefile MrJobBatchGDUpdate_LinearRegression.py
rom mrjob.job import MRJob

 This MrJob calculates the gradient of the entire training set
     Mapper: calculate partial gradient for each example

lass MrJobBatchGDUpdate_LinearRegression(MRJob):
   # run before the mapper processes any input
   def read_weightsfile(self):
       # Read weights file
       with open('weights.txt', 'r') as f:
           self.weights = [float(v) for v in f.readline().split(',')]
       # Initialze gradient for this iteration
       self.partial_Gradient = [0]*len(self.weights)
       self.partial_count = 0

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

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

```
    # Accumulate partial gradient from mapper and emit total 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 [self.mr(mapper_init=self.read_weightsfile,
                        mapper=self.partial_gradient,
                        mapper_final=self.partial_gradient_emit,
                        reducer=self.gradient_accumulater)]

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

```
Overwriting MrJobBatchGDUpdate_LinearRegression.py
```

Driver Code:

In [1]:

```
from numpy import random,array
from MrJobBatchGDUpdate_LinearRegression import MrJobBatchGDUpdate_Linear

learning_rate = 0.05
stop_criteria = 0.000005

# Generate random values as inital weights
weights = array([random.uniform(-4.2,-3.8),random.uniform(0.9, 1.1)])
# 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 data
mr_job = MrJobBatchGDUpdate_LinearRegression(args=['LinearRegression.csv'

# 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 = 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 = [-3.9577672   1.02193788]
iteration =1  weights = [-3.9602182   1.22898852]
iteration =2  weights = [-3.962505    1.38081511]
iteration =3  weights = [-3.9646469   1.49214677]
iteration =4  weights = [-3.9666593   1.57378414]
iteration =5  weights = [-3.96855466  1.63364718]
iteration =6  weights = [-3.9703432   1.67754345]
iteration =7  weights = [-3.97203348  1.70973155]
iteration =8  weights = [-3.97363277  1.73333426]
iteration =9  weights = [-3.97514734  1.75064144]
iteration =10  weights = [-3.97658271  1.76333223]
iteration =11  weights = [-3.97794375  1.7726379 ]
iteration =12  weights = [-3.97923487  1.77946134]
iteration =13  weights = [-3.98046006  1.7844646 ]
iteration =14  weights = [-3.98162298  1.78813316]
iteration =15  weights = [-3.98272702  1.79082303]
Final weights

[-3.98377532  1.79279525]
```

HW6.5.1 (OPTIONAL)

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

HW6.6 Clean up notebook for GMM via EM

Using the following notebook as a starting point:

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

Improve this notebook as follows:

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

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

$$p(\omega_k|\mathbf{x^{(i)}},\theta) = \frac{\pi_\mathbf{k}\mathcal{N}(\mathbf{x^{(i)}}|\mu_\mathbf{k},\sum_\mathbf{k})}{\sum_\mathbf{j=1}^\mathbf{K}\pi_\mathbf{j}\mathcal{N}(\mathbf{x^{(i)}}|\mu_\mathbf{j},\sum_\mathbf{j})} \quad (1)$$

M Step: Given probabilities, update priors, mean and covariance

$$\hat{\mu}_\mathbf{k} = \frac{\mathbf{1}}{\mathbf{n_k}}\sum_{\mathbf{i=1}}^{\mathbf{n}}\mathbf{p}(\omega_\mathbf{k}|\mathbf{x^{(i)}},\theta)\mathbf{x^{(i)}} \quad (2)$$

$$\hat{\sum_k} = \frac{1}{n_k}\sum_{i=1}^{n}p(\omega_k|\mathbf{x^{(i)}},\theta)(\mathbf{x^{(i)}}-\hat{\mu}_\mathbf{k})(\mathbf{x^{(i)}}-\hat{\mu}_\mathbf{k})^\mathbf{T} \quad (3)$$

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

Data Generation

In [ ]:
```
%matplotlib inline
import numpy as np
import pylab
import json
size1 = size2 = size3 = 1000
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)
# Randomlize data
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")
```

In [ ]:
```
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()
```

Initialization

Here suppose we know there are 3 components

In [ ]:
```
%%writefile mr_GMixEmInitialize.py
from mrjob.job import MRJob

from numpy import mat, zeros, shape, random, array, zeros_like, dot, lina
from random import sample
import json
from math import pi, sqrt, exp, pow


class MrGMixEmInit(MRJob):
    DEFAULT_PROTOCOL = 'json'

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

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


    def configure_options(self):
```

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

    def mapper(self, key, xjIn):
        #something simple to grab random starting point
        #collect the first 2k
        if self.count <= 2*self.options.k:
            self.count += 1
            yield (1,xjIn)

    def reducer(self, key, xjIn):
        #accumulate data points mapped to 0 from 1st mapper and pull out
        cent = []
        for xj in xjIn:
            x = json.loads(xj)
            cent.append(x)
            yield 1, xj
        index = sample(range(len(cent)), self.options.k)
        cent2 = []
        for i in index:
            cent2.append(cent[i])



        #use the covariance of the selected centers as the starting guess
        #first, calculate mean of centers
        # equation(2)

        mean = array(cent2[0])
        for i in range(1,self.options.k):
            mean = mean + array(cent2[i])
        mean = mean/float(self.options.k)



        #then accumulate the deviations
        cov = zeros((len(mean),len(mean)),dtype=float)
        for x in cent2:
            xmm = array(x) - mean
            for i in range(len(mean)):
                cov[i,i] = cov[i,i] + xmm[i]*xmm[i]

        cov = cov/(float(self.options.k))
        covInv = linalg.inv(cov)
```

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

        jDebug = json.dumps([cent2,mean.tolist(),cov.tolist(),covInv.toli
        debugPath = self.options.pathName + 'debug.txt'
        fileOut = open(debugPath,'w')
        fileOut.write(jDebug)
        fileOut.close()

        #also need a starting guess at the phi's - prior probabilities
        #initialize them all with the same number - 1/k - equally probabl

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

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

        #form output object
        outputList = [phi.tolist(), cent2, cov_1]

        jsonOut  = json.dumps(outputList)

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

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

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 [ ]:
```
iteFile mr_GMixEmIterate.py
 mrjob.job import MRJob

 math import sqrt, exp, pow,pi
 numpy import zeros, shape, random, array, zeros_like, dot, linalg
rt json

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

s MrGMixEm(MRJob):
DEFAULT_PROTOCOL = 'json'

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

    fullPath = self.options.pathName + 'intermediateResults.txt'
    fileIn = open(fullPath)
    inputJson = fileIn.read()
    fileIn.close()
    inputList = json.loads(inputJson)
    temp = inputList[0]
    self.phi = array(temp)          #prior class probabilities
    temp = inputList[1]
    self.means = array(temp)        #current means list
    temp = inputList[2]
    self.cov_1 = array(temp)        #inverse covariance matrices for w, c
    #accumulate partial sums
    #sum of weights - by cluster
    self.new_phi = zeros_like(self.phi)       #partial weighted sum of we
    self.new_means = zeros_like(self.means)
    self.new_cov = zeros_like(self.cov_1)

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


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

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

```
    def mapper(self, key, val):
        #accumulate partial sums for each mapper
        xList = json.loads(val)
        x = array(xList)
        wtVect = zeros_like(self.phi)
        for i in range(self.options.k):
            wtVect[i] = self.phi[i]*gauss(x,self.means[i],self.cov_1[i])
        wtSum = sum(wtVect)
        wtVect = wtVect/wtSum
        #accumulate to update est of probability densities.
        #increment count
        self.count += 1
        #accumulate weights for phi est
        self.new_phi = self.new_phi + wtVect
        for i in range(self.options.k):
            #accumulate weighted x's for mean calc
            self.new_means[i] = self.new_means[i] + wtVect[i]*x
            #accumulate weighted squares for cov estimate
            xmm = x - self.means[i]
            covInc = zeros_like(self.new_cov[i])

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


    def mapper_final(self):

        out = [self.count, (self.new_phi).tolist(), (self.new_means).tolist(),
        jOut = json.dumps(out)

        yield 1,jOut


    def reducer(self, key, xs):
        #accumulate partial sums
        first = True
        #accumulate partial sums
        #xs us a list of paritial stats, including count, phi, mean, and covar
        #Each stats is k-length array, storing info for k components
        for val in xs:
            if first:
                temp = json.loads(val)
                #totCount, totPhi, totMeans, and totCov are all arrays
                totCount = temp[0]
                totPhi = array(temp[1])
                totMeans = array(temp[2])
```

```
                totCov = array(temp[3])
                first = False
            else:
                temp = json.loads(val)
                #cumulative sum of four arrays
                totCount = totCount + temp[0]
                totPhi = totPhi + array(temp[1])
                totMeans = totMeans + array(temp[2])
                totCov = totCov + array(temp[3])
        #finish calculation of new probability parameters. array divided by ar
        newPhi = totPhi/totCount
        #initialize these to something handy to get the right size arrays
        newMeans = totMeans
        newCov_1 = totCov
        for i in range(self.options.k):
            newMeans[i,:] = totMeans[i,:]/totPhi[i]
            tempCov = totCov[i,:,:]/totPhi[i]
            #almost done.  just need to invert the cov matrix.  invert here to
            #with every input data point.
            newCov_1[i,:,:] = linalg.inv(tempCov)

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

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

_name__ == '__main__':
MrGMixEm.run()
```

Driver

In [ ]:

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

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()
```

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

#first run the initializer to get starting centroids
filePath = 'data.txt'
mrJob = MrGMixEmInit(args=[filePath])
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()

delta = 10
iter_num = 0
#Begin iteration on change in centroids
while delta > 0.02:
    print "Iteration" + str(iter_num)
    iter_num = iter_num + 1
    #parse old centroid values
    oldParam = json.loads(paramJson)
    #run one iteration
    oldMeans = oldParam[1]
    mrJob2 = MrGMixEm(args=[filePath])
    with mrJob2.make_runner() as runner:
        runner.run()

    #compare new centroids to old ones
    fileIn = open(emPath)
    paramJson = fileIn.read()
    fileIn.close()
    newParam = json.loads(paramJson)

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

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

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

```
print newMeans
plot_iteration(newMeans)
```

In [ ]:

In [ ]:

In [20]:

```
## HW6.6 Clean up notebook for GMM via EM

Using the following notebook as a starting point:

http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/0t7985e40fovlkw/EM-GMM-

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 comm
-- Comment the code
-- Rename/Reorganize the code to make it more readable
-- Rerun the examples similar graphics (or possibly better graphics)


## HW6.7  Implement Bernoulli Mixture Model via EM
Implement the EM clustering algorithm to determine Bernoulli Mixture Mode

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

https://www.dropbox.com/s/maoj9jidxj1xf5l/MIDS-Live-Lecture-06-EM-Bernoui

Cross-check that you get the same cluster assignments and cluster Bernoui

As a full test: use the same dataset from HW 4.5, the Tweet Dataset.
Using this data, you will implement a 1000-dimensional EM-based Bernoulli
by their 1000-dimensional word stripes/vectors using K = 4.  Use the same

Repeat this experiment using your KMeans MRJob implementation fron HW4.
Report the rand index score using the class code as ground truth label fo

Here is some more information on the Tweet Dataset.

Here you will use a different dataset consisting of word-frequency distri
for 1,000 Twitter users. These Twitter users use language in very differe
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/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
by their 1000-dimensional word stripes/vectors using several
centroid initializations and values of K.



## HW6.8 (OPTIONAL) 1 Million songs
Predict the year of the song. Ask Jimi
```

```
  File "<ipython-input-20-8484db9086f4>", line 2
    * Taking x=1 as the first approximation(xt1) of a root of $X^3 + 2
x -4 = 0$, use the Newton-Raphson method to calculate the second appro
ximation (denoted as xt2) of this root. (Hint the solution is xt2=1.2)
    ^
SyntaxError: invalid syntax
```

In [ ]: