DATSCIW261 ASSIGNMENT 6

MIDS UC Berkeley, Machine Learning at Scale

AUTHOR : Rajesh Thallam
EMAIL : rajesh.thallam@ischool.berkeley.edu
WEEK : 7
DATE : 17-Oct-15

HW6.0

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

Mathematical Optimization

In mathematics, computer science and economics, mathematical optimization is the selection of a best element with regard to some criteria from some set of available alternatives. An optimization problem consists of maximizing or minimizing an objective function by choosing input values from an allowed set and computing the value of the function.

Length of Stay in Health Care

Healthcare cost depends on the healthcare value provided to the patient which is measured in terms of the patient outcomes achieved per dollar expended. Health outcome is measured along patient's wellness, readmission rate, and sustainability of recovery. This cost includes total costs of all resources – manpower, drugs, device, space and equipment – during patient's care duration i.e. hospital inpatients Length of Stay (LOS). The value of health care delivered to patients can be increased either by improving outcomes at same costs or by reducing the total costs involved in patients care while maintaining the quality of outcomes.

The primary objective function is to minimize LOS which would bring reduction of total costs incurred during patient's stay at hospital. The decision variables could be combination of the predictor variables listed: age, gender, ethnicity, diagnosis stats, location, admission and discharge dates, observed and average LOS, specialty treatments received, patient medical history, family medical history, physician comments and recommendations, comorbidities, costs of the treatment, discharge destination, discharge method and patient condition on admission and discharge.

HW6.1

Optimization Theory:

For unconstrained univariate optimization what are the first order Necessary Conditions for Optimality (FOC). What are the second order optimality conditions (SOC)? Give a mathematical defintion.

For univariate optimization problem with no constraints with $f(x)$ as objective function, at a point x* in the domain of *f* where a minimum occurs, the following are true

- **First Order Condition: For a point to be maximum or minimum, necessary condition is to have a zero value for the gradient function (first derivative) at the point.**

$$\frac{\partial f(x*)}{\partial x} = 0$$

- **Second Order Condition: If a point has a value of 0 at the first derivative, and a negative value at the second derivative function, then it is a local maximum. or if a positive value at the second derivative function, then it is a local minimum.**

  If x* is a local maximum,

$$\frac{\partial f^2(x*)}{\partial x^2} < 0$$

  If x* is a local minimum,

$$\frac{\partial f^2(x*)}{\partial x^2} > 0$$

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.

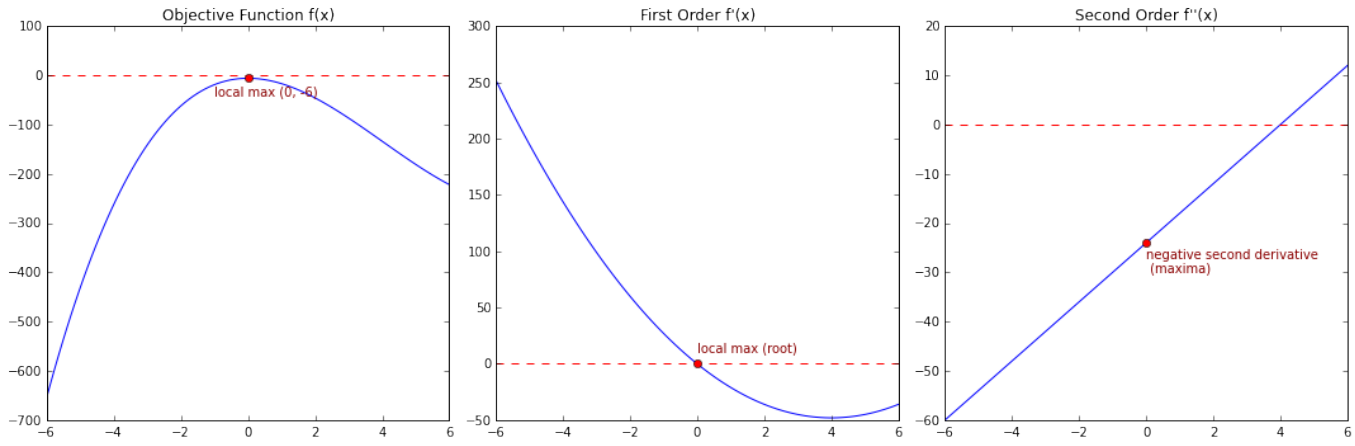
For the given univariate objective function $f(x)$, following are the firt order and second order derivative functions

Objective function: $f(x) = x^3 - 12x^2 - 6$
First Order: $f'(x) = 3x^2 - 24x$
Second Order: $f''(x) = 6x - 24$

Plots
Plotting the functions over domain [-6, 6]

In [149]:

```
# making matplotlib plots inline
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 5))

# objective function
ax = plt.subplot(131)
X = np.linspace(-6, 6, 100)
y = [(x*x*x - 12*x*x - 6) for x in X]
line = plt.plot(X, y)
ax.set_title("Objective Function f(x)")
# annotation
plt.axhline(y=0., color='r', ls='dashed')
plt.annotate(
    'local max (0, -6)',
    xy = (0, -6),
    xytext = (-1, -40),
    color = "darkred"
)
plt.plot(0,-6,'o',color='r')

# first order derivative
ax = plt.subplot(132)
y = [(3*x*x - 24*x) for x in X]
line = plt.plot(X, y)
ax.set_title("First Order f'(x)")
plt.axhline(y=0., color='r', ls='dashed')
plt.plot(0,0,'o',color='r')
plt.annotate(
    'local max (root)',
    xy = (0, -6),
    xytext = (0, 10),
    color = "darkred"
)

# second order derivative
ax = plt.subplot(133)
y = [(6*x - 24) for x in X]
line = plt.plot(X, y)
ax.set_title("Second Order f''(x)")
plt.axhline(y=0., color='r', ls='dashed')
plt.plot(0,-24,'o',color='r')
plt.annotate(
    'negative second derivative\n (maxima)',
    xy = (0, -24),
    xytext = (0, -30),
    color = "darkred"
)

plt.tight_layout()
plt.show()
```



From the graphs, it appears we have local maximum at (x, y) = (0, -6)

- **Necessary condition for FOC for optimality suggests minimum/maxiumum point when $f'(x = x*) = 0$. As per the plot, f'(x) = 0 at x = 0 (i.e. root).**
- **Sufficient condition SOC declares candidate point x* as optimal**
  - **Maximum when $f''(x = x*) < 0$**
  - **Minimum when $f''(x = x*) > 0$**

  In the second order plot, $f''(0) < 0$ confirming a local maximum at x = 0.

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 the unconstrained multi-variate optimization problem $f(x_1, x_2, \ldots, x_n)$, to find the optimal value following necessary and sufficient conditions must be satisfied

- **First Order Condition: states that the gradient at a specific point x = x* is the vector whose elements are the respective partial derivatives evaluated at x = x*, such that**

$$\Delta f(x = x*) = \left(\frac{\partial f}{\partial x1} + \frac{\partial f}{\partial x2} + \ldots + \frac{\partial f}{\partial xn}\right) = (0, 0, \ldots, 0)$$

- **Second Order Condition: Hessian matrix (H) of second order deratives should satisfy**
  - **If $H(x*)$ is negative definite then x* is a local maximum.**
  - **If $H(x*)$ is positive definite then x* is a local minimum.**

  Hessian matrix is a square matrix of second-order partial derivatives of a scalar-valued function, or scalar field.

In the current context of problem, we have only one variable and Hessian matrix is partial second derivative of x

$$H = \left[\frac{\partial^2 f}{\partial x^2}\right]$$
$$H = \left[6x - 24\right]$$

HW6.2

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

In Newton-Raphson method, gradient function is approximated by tangent slope at an arbitrary point (here starting with x = 1).

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

Pluggin $x = 1$ into objective function

$$f(x) = x^3 + 2x - 4$$
$$f(1) = 1^3 + 2 - 4 = -1$$

Calculating tangent slope

$$f'(x) = 3x^2 + 2$$
$$f'(1) = 3 + 2 = 5$$

Plugging in the point and slope into a linear equation (second order derivative)

$$y = mx + b$$
$$-1 = 5 * 1 + b$$
$$b = -6$$

Solve for $y = 0$ to get x

$$0 = 5x - 6$$
$$x = 1.2$$

Putting this into python code below

$$x_{t2} = x_{t1} - \frac{f(x_{t1})}{f'(x_{t1})}$$

In [154]:
```
def f(x):
    return float(x)**3 + 2*float(x) - 4

def f1(x):
    return 3*float(x)**2 + 2

def newton_raphson(f, f1, x0, _max = 10000):
    x_t1 = x0

    while True:
        x = x_t1 - f(x_t1)/f1(x_t1)
        _max -= 1
        if abs(x - x_t1) < 0.001 or _max <= 0:
            return x
        else:
            x_t1 = x

# second try value
print "x_t2 = {}".format(newton_raphson(f, f1, 1.0, 1.0))
```

```
x_t2 = 1.2
```

HW6.3

Convex optimization

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

The optimization function $min_x f(x)$ is called a convex optimization problem if:

- **The objective function f is convex**
- **The functions defining the inequality constraints f_i are convex**
- **The functions defining the equality constraints f_i are affine**

First Order Condition: The necessary condition for optimality for a given x* is a global minimum of $f(x)$ if $f'(x) = 0$ i.e. funtion is globally above the tangent at y

Second Order Condition: The second order optimality conditions are not neessary. The first order condition is sufficient to confirm the global minimum is not a local minimum. This is convenient to machine learning because there is no need to test the second order equation to confirm global minimum after gradient descent.

For convex optimization problems, only first order condition is needed for optimization.

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

HW6.4

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

$$0.5 * \sum_i weight_i * (WX_i - y_i)^2$$

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.

Let cost function $J(W)$ be defined as

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

Expanding cost function

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

Calculating gradient descent

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

Rearranging terms

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

where $(WX_i - y_i)$ is error term and $w_i X_i$ is weighted input i and combined gives the weighted error

---

HW6.5

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

Weight each example as follows: weight(x)= abs(1/x)

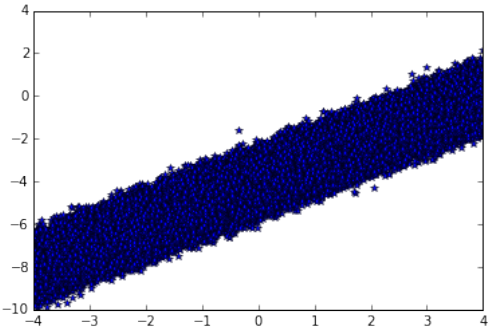
Sample 1% of the data in MapReduce and use the sampled dataset to train a (weighted if available in SciKit-Learn) linear regression model locally using [SciKit-Learn] (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html). Plot the resulting weighted linear regression model versus the original model that you used to generate the data. Comment on your findings.

Data Generation

```
Sizes: 1000000 points
True model: y = 1.0 * x - 4
Noise: Normal Distributed mean = 0, var = 0.5
```

In [99]:
```
%matplotlib inline
import numpy as np
import pylab

size = 1000000
x = np.random.uniform(-4, 4, size)
y = x * 1.0 - 4 + np.random.normal(0,0.5,size)
data = zip(y,x)

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

In [74]:
```
pylab.plot(x, y,'*')
pylab.show()
```



Gradient Descent

In [94]:
```
%%writefile GradientDescentWOLS.py
#!/usr/bin/python
from mrjob.job import MRJob
from mrjob.step import MRStep

# Mapper: calculate partial gradient for each example
class MRJobGradientDescentWOLS(MRJob):

    # run before the mapper processes any input
    def read_weightsfile(self):
        self.weights = [ float(x) for line in open('weights.txt').read().strip().split('\n') for x in line.split(',') ]
        # Read weights file
        with open('weights.txt', 'r')  as f:
            for line in f:
                val = line.strip().split(',')
        self.weights = [float(x) for x in val]
        self.partial_Gradient = [0] * len(self.weights)
        self.partial_count = 0

    # calculate partial gradient for each example
    def partial_gradient(self, _, line):
        y_i, x_i = (map(float,line.split(',')))

        w_i = 1 / abs(x_i)
        intercept, slope = self.weights
        y_hat = slope * x_i + intercept

        gd_intercept = w_i * (y_hat - y_i)
        gd_slope = w_i * (y_hat - y_i) * x_i

        self.partial_Gradient[0] += gd_intercept
        self.partial_Gradient[1] += gd_slope
        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__':
    MRJobGradientDescentWOLS.run()
```

```
Overwriting GradientDescentWOLS.py
```

In [97]:
```
%%writefile driver.py
#!/usr/bin/python
from GradientDescentWOLS import MRJobGradientDescentWOLS
import numpy as np
import sys
import time

start_time = time.time()

learning_rate = 0.1
stop_criteria = 0.00000005

# create a mrjob instance for batch gradient descent update over all data
mr_job = MRJobGradientDescentWOLS(args=['LinearRegression.csv', '--file', 'weights.txt', '--no-strict-protocol'])

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

# update centroids iteratively
i = 0

# write coefficients for plotting
f_coeff = open(output_fname,'w')

while(1):
    coeff = "iteration={}\tweights=[{}, {}]".format(i, weights[0], weights[1])
    print coeff
    f_coeff.write(coeff + '\n')

    #Save weights from previous iteration
    weights_old = weights

    wth 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*np.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

f_coeff.close()

print "Final weights\n"
print weights

end_time = time.time()
print "Time taken to determine gradient descent of weighted linear regression model = {:.2f} seconds".format(end_time - st
art_time)
```

```
Overwriting driver.py
```

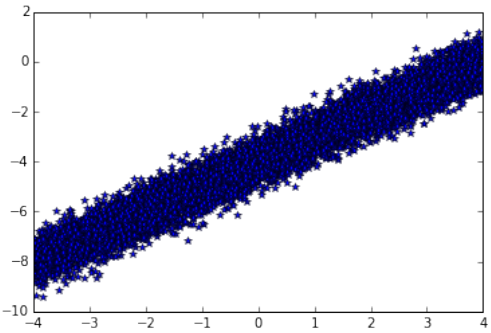
In [91]:
```
!./driver.py
```

```
iteration=0     weights=[-0.117121327441, 2.11593563767]
iteration=1     weights=[-1.38056533283, 1.89276430707]
iteration=2     weights=[-2.22967689629, 1.71423794932]
iteration=3     weights=[-2.80033113947, 1.57142200244]
iteration=4     weights=[-3.18384476502, 1.45717105371]
iteration=5     weights=[-3.44158836671, 1.36577019406]
iteration=6     weights=[-3.61480684171, 1.29264838557]
iteration=7     weights=[-3.73121933841, 1.23414934299]
iteration=8     weights=[-3.80945484431, 1.18734836233]
iteration=9     weights=[-3.86203318498, 1.14990586464]
iteration=10    weights=[-3.89736844536, 1.11995028363]
iteration=11    weights=[-3.92111539645, 1.09598440979]
iteration=12    weights=[-3.93707437929, 1.07681048754]
iteration=13    weights=[-3.94779944406, 1.0614703067]
iteration=14    weights=[-3.95500705781, 1.04919728422]
iteration=15    weights=[-3.9598507814, 1.03937813484]
iteration=16    weights=[-3.96310586957, 1.0315222106]
iteration=17    weights=[-3.96529333333, 1.02523697425]
iteration=18    weights=[-3.96676331757, 1.02020837872]
iteration=19    weights=[-3.96775113569, 1.01618517117]
iteration=20    weights=[-3.96841492822, 1.01296633623]
iteration=21    weights=[-3.9688609716, 1.01039105045]
iteration=22    weights=[-3.96916068709, 1.00833064575]
iteration=23    weights=[-3.96936207164, 1.0066821799]
iteration=24    weights=[-3.96949738018, 1.00536329272]
iteration=25    weights=[-3.96958828832, 1.00430809084]
iteration=26    weights=[-3.96964936212, 1.00346385523]
iteration=27    weights=[-3.96969038978, 1.00278840722]
iteration=28    weights=[-3.9697179487, 1.00224800102]
iteration=29    weights=[-3.96973645863, 1.0018156377]
iteration=30    weights=[-3.96974888933, 1.0014697163]
iteration=31    weights=[-3.96975723624, 1.00119295452]
Final weights

[-3.96976284  1.00097153]
```

SciKitLearn OLS

In [103]:
```
# sample 1% of data
i = np.random.choice(np.arange(1000000), 10000, replace=False)
x_sample = x[i].reshape((10000,1))
y_sample = y[i].reshape((10000,1))
plt.plot(x_sample, y_sample,'*')
```

Out[103]:
```
[<matplotlib.lines.Line2D at 0x7f611a663050>]
```



In [104]:
```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(x_sample, y_sample)

print "Slope: {}".format(lr.coef_[0][0])
print "Intercept: {}".format(lr.intercept_[0])
```

```
Slope: 0.998412067332
Intercept: -4.00277490897
```

PolyFit WOLS

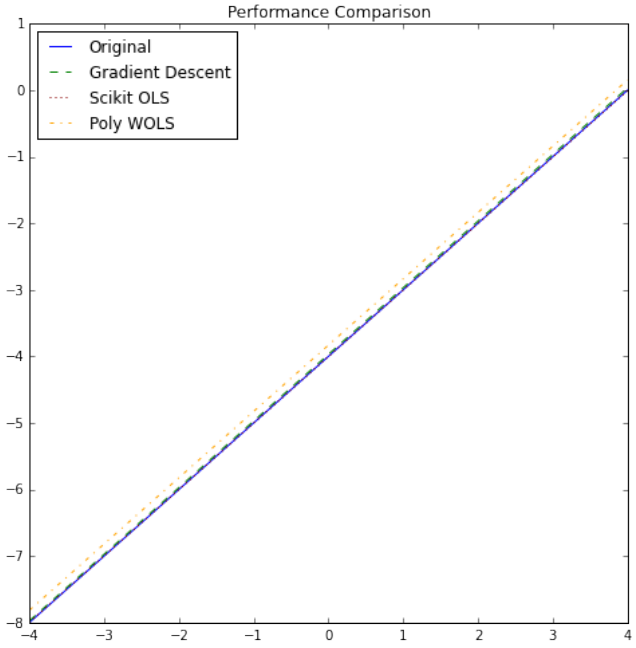
In [106]:
```
coeff = np.polyfit(x_sample.flatten(), y_sample.flatten(), 1, w = abs(1/x_sample).flatten())
print "Slope: {}".format(coeff[0])
print "Intercept: {}".format(coeff[1])
```

```
Slope: 0.99602305286
Intercept: -3.83046625209
```

Performance Comparison

In [157]:
```
plt.figure(figsize=(8, 8))

# Uniform dataset
plt.subplot(1,1,1)

ls = np.linspace(-4, 4, 10)

# plot Original Line
y_o = 1.0 * ls - 4
plt.plot(ls, y_o, color='blue', label="Original", linestyle='-')

# plot gradient descent
m = 1.00097153
b = -3.96976284
y_g = m * ls + b
plt.plot(ls, y_g, label='Gradient Descent', linestyle='--', color='green')

# plot Scikit Learn OLS line
y_s = lr.coef_[0] * ls + lr.intercept_
plt.plot(ls, y_s, label="Scikit OLS", linestyle=':',color='darkred')

# plot polyfit weighted WOLS line
y_p = coeff[0] * ls + coeff[1]
plt.plot(ls, y_p, label="Poly WOLS", linestyle='-.',color='orange')

plt.legend(loc=2)
plt.title("Performance Comparison")
```

Out[157]:
```
<matplotlib.text.Text at 0x7f611a287150>
```



Report

Comparing the four approaches on the same plot, following are the findings:

- **Weighted OLS does not perform as well as OLS due to the reason that noise around x=0 is amplified with the weights assigned.**
- **Gradient Descent has good performance due to more data compared to other approaches SciKitLearn OLS and Polynomial weighted OLS.**

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.

**** -- END OF ASSIGNMENT 6 -- ****