DATASCI W261: Machine Learning at Scale

- **Sayantan Satpati**
- **sayantan.satpati@ischool.berkeley.edu**

- **W261**
- **Week-6**
- **Assignment-6**
- **Date of Submission: 17-OCT-2015**

=== Week 6 ASSIGNMENTS ===

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.

**** In mathematics, computer science and operations research, mathematical optimization (alternatively, optimization or mathematical programming) 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. The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. More generally, optimization includes finding "best available" values of some objective function given a defined domain (or a set of constraints), including a variety of different types of objective functions and different types of domains.

Optimization Problem at Work: An Example

Optimization of the Ranking of Search Results on ebay's Search Result Page (SRP).

Objective Function: Function of a large number of factors that affect the Search Ranking; it is carried out in 2 or more stages (Multi Round Ranking). A less expensive ranking function is applied on the entire search results, which is then narrowed down by a more expensive function on much smaller dataset.

Decision Variables: Price, Shipping, Item Location, Item Category Demand Data for the item searched, User Profile, etc

Reference: https://en.wikipedia.org/wiki/Mathematical_optimization (https://en.wikipedia.org/wiki/Mathematical_optimization)

HW6.1

Optimization theory: For unconstrained univariate optimization what are the first order Necessary Conditions for Optimality (FOC). What are the second order optimality conditions (SOC)? Give a mathematical defintion. Also in python, plot the univartiate function X^3 -12x^2-6 defined over the real domain -6 to +6.

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

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?

Univariate Optimization

FOC

- Consider the function, y = f(x).
- The necessary condition for a relative extremum (maximum or minimum) is that the first-order derivative be zero, i.e. f'(x) = 0.

SOC

- If the first-order condition is satisfied at x=x0,
  1. f(x0) is a local maximum if f''(x0) < 0
  2. f(x0) is a local minimum if f''(x0) > 0

PLOTS

Optimization Function

$$f(x) = X^3 - 12x^2 - 6$$

FOC

$$f'(x) = 3x^2 - 24x$$

SOC

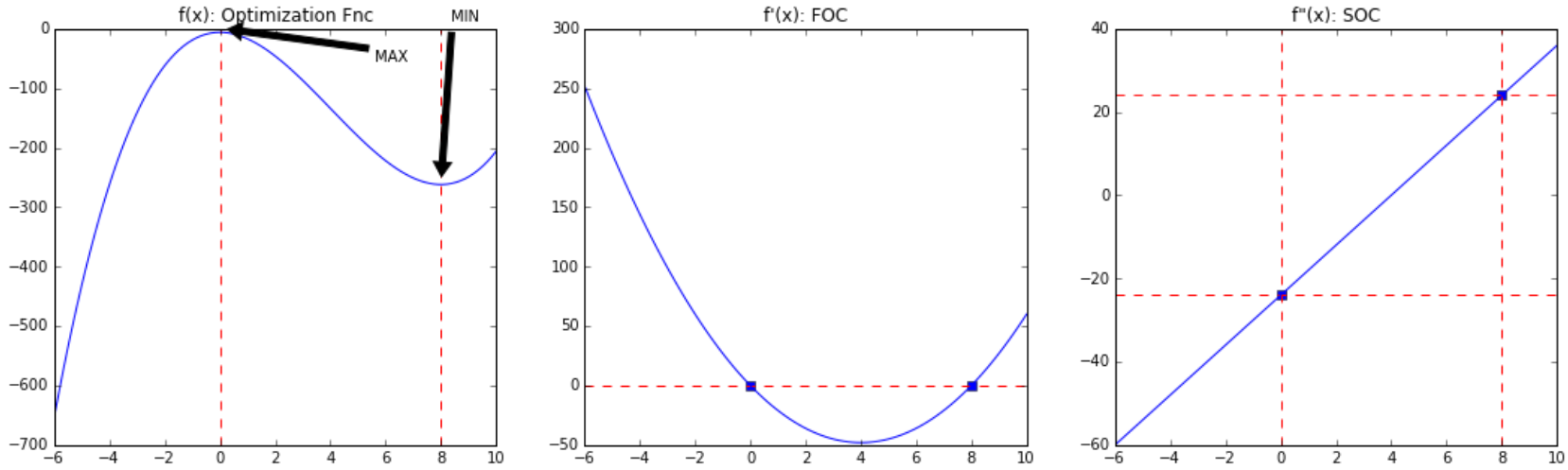
$$f\ ''\ (x) = 6x - 24$$

- At x=0, f'(x) = 0 and f"(x) < 0. Therefore, we get maxima at x = 0
- At x=8, f'(x) = 0 and f"(x) > 0. Therefore, we get minima at x = 8

In [125]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(18,5))
plt.subplot(131)
x = np.linspace(-6,10,100)
y = map(lambda x: x*x*x - 12*x*x - 6, x)
plt.title('f(x): Optimization Fnc')
#plt.annotate('max', xy=(0, 0))
plt.annotate('MAX', xy=(0,1),  xycoords='data',
                xytext=(0.8, 0.95), textcoords='axes fraction',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='right', verticalalignment='top',
                )
plt.annotate('MIN', xy=(8,-262),  xycoords='data',
                xytext=(0.9, 0.01), textcoords='axes fraction',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='left', verticalalignment='bottom',
                )
#plt.annotate('min', xy=(8, -262))
plt.axvline(x=0,color='r',ls='--')
plt.axvline(x=8,color='r',ls='--')
plt.plot(x, y)

plt.subplot(132)
y = map(lambda x: 3*x*x - 24*x, x)
plt.title("f'(x): FOC")
plt.plot([0,8], [0,0], 'bs')
plt.axhline(color='r', ls='--')
plt.plot(x, y)

plt.subplot(133)
y = map(lambda x: 6*x -24, x)
plt.title('f"(x): SOC')
plt.plot([0,8], [-24,24], 'bs')
plt.axhline(y=-24,color='r',ls='--')
plt.axhline(y=24,color='r',ls='--')
plt.axvline(x=0,color='r',ls='--')
plt.axvline(x=8,color='r',ls='--')
```

```
plt.plot(x, y)
```

```
Out[125]: [<matplotlib.lines.Line2D at 0x124d985d0>]
```



Multivariate Optimization

FOC

- Partial Derivative of the multivariate optimization function is set to 0 in order to find the critical or stationary points.

$$\frac{\partial f}{\partial x} = 0$$

$$\frac{\partial f}{\partial y} = 0$$

SOC

Let f be a function of n variables with continuous partial derivatives of first and second order, defined on the set S. Suppose that x^* is a stationary point of f in the interior of S (so that $f_i' = 0$ for all i).

- Take the second Partial Derivatives and put them in a matrix called the Hessian Matrix H. It's used in a second derivative test to find extreme values of functions of more than one variable.

Taking Example of n=2

$\displaystyle \left[\begin{array}{cc} f'*{11}(x^*) & f*{12}(x^) \ f'_{21}(x^) & f'_{22}(x^*)
\end{array} \right]$

1. if $H(x^*)$ is negative definite then x* is a local maximizer
2. if $H(x^*)$ is negative semidefinite, but neither negative definite nor positive semidefinite, then x* is not a local minimizer, but might be a local maximizer
3. if $H(x^*)$ is positive definite then x* is a local minimizer
4. if $H(x^*)$ is positive semidefinite, but neither positive definite nor negative semidefinite, then x* is not a local maximizer, but might be a local minimizer
5. if $H(x^*)$ is neither positive semidefinite nor negative semidefinite then x* is neither a local maximizer nor a local minimizer.

Example

Consider the function $f(x, y) = x^3 + y^3 - 3xy$. The first-order conditions for an optimum are $3x^2 - 3y = 0$ and $3y^2 - 3x = 0$

Thus the stationary points satisfy $y = x^2 = y^4$, so that either (x, y) = (0, 0) or $y^3 = 1$. So there are two stationary points: (0, 0), and (1, 1).

Now, the Hessian of f at any point (x, y) is

H(x, y):

$$\begin{bmatrix} 6x & -3 \\ -3 & 6y \end{bmatrix}$$

Thus |H(0, 0)| = −9, so that (0, 0) is neither a local maximizer nor a local minimizer (i.e. is a saddle point).

We have $f''_{11}(1, 1) = 6 > 0$ and |H(1, 1)| = 36 − 9 > 0, so that (1, 1) is a local minimizer.

Refer: https://www.economics.utoronto.ca/osborne/MathTutorial/LON.HTM (https://www.economics.utoronto.ca/osborne/MathTutorial/LON.HTM)

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)

Second Approximation is 1.2

In [128]:
```
def newton_raphson(x):
    f_x = x*x*x + 2*x -4
    f1_x = 3*x*x + 2
    return x - (f_x * 1.0 /f1_x)

# Initial Value of x=1
x1 = 1
for i in xrange(10):
    x1 = newton_raphson(x1)
    print "[{0}] Approximation: {1}".format(i+1, x1)
```

```
[1] Approximation: 1.2
[2] Approximation: 1.17974683544
[3] Approximation: 1.17950905701
[4] Approximation: 1.1795090246
[5] Approximation: 1.1795090246
[6] Approximation: 1.1795090246
[7] Approximation: 1.1795090246
[8] Approximation: 1.1795090246
[9] Approximation: 1.1795090246
[10] Approximation: 1.1795090246
```

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 following makes an optimization problem convex:

1. if a local minimum exists, then it is a global minimum.
2. the set of all (global) minima is convex.
3. for each strictly convex function, if the function has a minimum, then the minimum is unique.

Intuitively, this means that if we take any two elements in set C, and draw a line segment between these two elements, then every point on that line segment also belongs to C.

First Order

First Order Condition for Convexity:

Suppose a function f : R n → R is differentiable (i.e., the gradient ∇xf(x) exists at all points x in the domain of f). Then f is convex if and only if D(f) is a convex set and for all x, y ∈ D(f), f(y) ≥ f(x) + ∇xf(x) T(y − x). The function ***f(x) + ∇xf(x)T(y − x)*** is called the first-order approximation to the function f at the point x. Intuitively, this can be thought of as approximating f with its tangent line at the point x. The first order condition for convexity says that f is convex if and only if the tangent line is a global underestimator of the function f. In other words, if we take our function and draw a tangent line at any point, then every point on this line will lie below the corresponding point on f.

Second Order

Suppose a function f : R n → R is twice differentiable (i.e., the Hessian ∇^2x f(x) is defined for all points x in the domain of f). Then f is convex if and only if D(f) is a convex set and its Hessian is positive semidefinite: i.e., for any x ∈ D(f), **∇^2x f(x) >> 0** Here, the notation '>>' when used in conjunction with matrices refers to positive semidefi- niteness, rather than componentwise inequality.

In one dimension, this is equivalent to the condition that the second derivative f′′(x) always be non-negative (i.e., the function always has positive non-negative). Again analogous to both the definition and the first order conditions for convexity, f is strictly convex if its Hessian is positive definite, concave if the Hessian is negative semidefi- nite, and strictly concave if the Hessian is negative definite.

It is not necessary to compoute the 2nd order derivative, since due to the covex nature of the function, f"(x) is always non-negative.

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

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

HW6.4

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

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

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

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

Lets define some notations:

- w = weight vector of size i
- X = Design Matrix
- y = Output Vector
- n = Num of Training Examples
- p = Num of features

Optimization Problem

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

After taking the first order partial derivative on weights:

$\displaystyle w*j* = *w_j - 0.5 2 1/i * \sum*{i=1}^{n} (wX - y) *w_i* x_j $

⇔

$\displaystyle w*j* = *w_j - 1/i * \sum*{i=1}^{n} (wX - y) *w_i* x_j $

Steps for running Gradient Descent:

```
Initialize w vector of length j to 1 {np.ones(i)}.

While Not Converged (or num_iteration < MAX_ITERATIONS):
    Calculate hypothesis => wX = np.dot(X,w)
    Calculate loss => hypothesis - y
    Calculate the Gradient => np.dot(np.dot(loss, x_j), w_i)
    Finally update the weights => w = w - 1/i * gradient
```

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 (http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kritdm3mo1daolj/MrJobLinearRegressionGD.ipynb)

Weight each example as follows:

weight(x)= 1/x

Sample 1% of the data in MapReduce and use the sampled dataset to train a weighted 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))

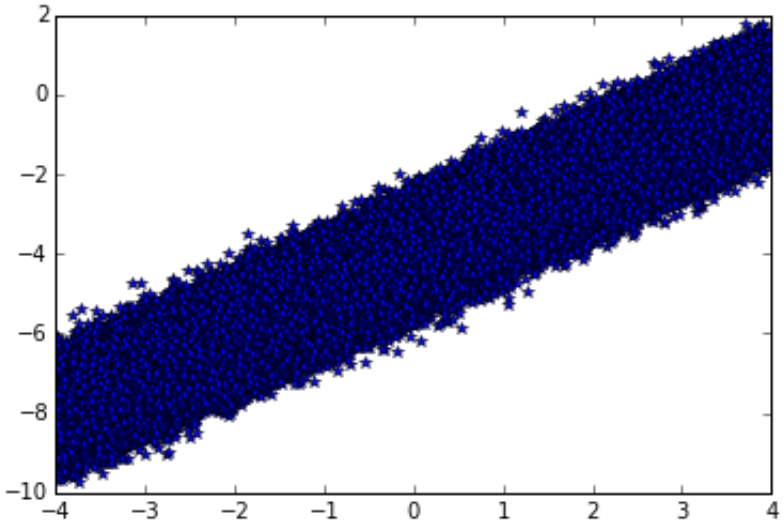
Plot the resulting weighted linear regression model versus the original model that you used to generate the data. Comment on your findings.

Generate 1 million datapoints / Sample 1%

In [12]:
```
!rm -f LinearRegression.csv
import numpy as np
import random
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)
print data[:2]
data_sample = random.sample(data, size/100)
print "Length of Sampled Data: %d" %(len(data_sample))
np.savetxt('LinearRegression.csv', data, delimiter = ",")
np.savetxt('LinearRegression_Sample.csv', data_sample, delimiter = ",")
!head -n 5 LinearRegression.csv
```

```
[(-1.1745142980270935, 2.756885626648609), (-4.9092846515895205, -0.48284897935925652)]
Length of Sampled Data: 10000
-1.1745142980270935,2.756885626648609
-4.909284651589520543e+00,-4.828489793592565249e-01
-3.115156445013957054e+00,6.548877899007843340e-01
-3.316521387617382910e+00,4.620116129601967714e-01
-2.647259824231611436e+00,1.475351889157416885e+00
```

In [13]:
```
%matplotlib inline
pylab.plot(x, y,'*')
pylab.show()
```



In [14]:
```
%%writefile MrJobBatchGDUpdate_LinearRegression.py
from mrjob.job import MRJob
from mrjob.step import MRStep

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

    # Calculate partial gradient for each example
    def partial_gradient(self, _, line):
        D = (map(float,line.split(',')))
        # y_hat is the predicted value given current weights
        y_hat = self.weights[0]+self.weights[1]*D[1]
        # Update parial gradient vector with gradient form current example
        # Weight for Weighted Linear Regression
        w = abs(1.0/D[1])
        self.partial_Gradient =  [self.partial_Gradient[0]+ (D[0]-y_hat)*w, self.partial_Gradi
ent[1]+(D[0]-y_hat)*D[1]*w]
        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 [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 [15]:
```
!chmod a+x MrJobBatchGDUpdate_LinearRegression.py
```

In [16]:
```
from numpy import random,array
from MrJobBatchGDUpdate_LinearRegression import MrJobBatchGDUpdate_LinearRegression

learning_rate = 0.05
stop_criteria = 0.000005

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

# create a mrjob instance for batch gradient descent update over all data
mr_job = MrJobBatchGDUpdate_LinearRegression(args=['--file', 'weights.txt',
                                                   '--no-strict-protocol',
                                                  '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 = [ 0.75490465  1.76135779]
iteration =1  weights = [ 0.51712274  1.55814631]
iteration =2  weights = [ 0.29124087  1.40911474]
iteration =3  weights = [ 0.07666113  1.29982049]
iteration =4  weights = [-0.12718374  1.21967076]
iteration =5  weights = [-0.32083204  1.16089635]
iteration =6  weights = [-0.50479477  1.117799  ]
iteration =7  weights = [-0.67955704  1.08619938]
iteration =8  weights = [-0.84557949  1.06303223]
iteration =9  weights = [-1.00329957  1.04604935]
iteration =10  weights = [-1.15313273  1.03360184]
iteration =11  weights = [-1.29547356  1.02448035]
iteration =12  weights = [-1.43069686  1.01779792]
iteration =13  weights = [-1.55915863  1.01290401]
iteration =14  weights = [-1.68119705  1.00932151]
iteration =15  weights = [-1.79713336  1.00670053]
iteration =16  weights = [-1.90727271  1.00478443]
iteration =17  weights = [-2.01190499  1.003385  ]
iteration =18  weights = [-2.11130558  1.00236424]
iteration =19  weights = [-2.20573609  1.00162093]
iteration =20  weights = [-2.29544503  1.00108083]
iteration =21  weights = [-2.38066849  1.00068954]
iteration =22  weights = [-2.46163077  1.00040714]
iteration =23  weights = [-2.53854491  1.00020437]
iteration =24  weights = [-2.61161333  1.0000598 ]
iteration =25  weights = [-2.68102833  0.9999577 ]
iteration =26  weights = [-2.74697257  0.99988656]
iteration =27  weights = [-2.80961959  0.99983793]
iteration =28  weights = [-2.86913426  0.99980564]
iteration =29  weights = [-2.92567319  0.99978516]
iteration =30  weights = [-2.97938518  0.99977319]
iteration =31  weights = [-3.03041157  0.9997673 ]
iteration =32  weights = [-3.07888663  0.99976573]
iteration =33  weights = [-3.12493795  0.9997672 ]
iteration =34  weights = [-3.1686867   0.99977075]
iteration =35  weights = [-3.21024801  0.99977571]
iteration =36  weights = [-3.24973125  0.99978159]
iteration =37  weights = [-3.28724034  0.99978803]
iteration =38  weights = [-3.32287397  0.99979477]
iteration =39  weights = [-3.35672591  0.99980164]
```

```
iteration =40  weights = [-3.38888527  0.9998085 ]
iteration =41  weights = [-3.41943665  0.99981526]
iteration =42  weights = [-3.44846047  0.99982187]
iteration =43  weights = [-3.47603309  0.99982828]
iteration =44  weights = [-3.50222709  0.99983447]
iteration =45  weights = [-3.52711138  0.99984041]
iteration =46  weights = [-3.55075146  0.99984612]
iteration =47  weights = [-3.57320954  0.99985158]
iteration =48  weights = [-3.59454471  0.99985679]
iteration =49  weights = [-3.61481313  0.99986176]
iteration =50  weights = [-3.63406812  0.9998665 ]
iteration =51  weights = [-3.65236037  0.99987101]
iteration =52  weights = [-3.669738    0.99987531]
iteration =53  weights = [-3.68624675  0.9998794 ]
iteration =54  weights = [-3.70193006  0.99988328]
iteration =55  weights = [-3.71682921  0.99988698]
iteration =56  weights = [-3.7309834   0.99989049]
iteration =57  weights = [-3.74442989  0.99989383]
iteration =58  weights = [-3.75720404  0.99989701]
iteration =59  weights = [-3.76933949  0.99990002]
iteration =60  weights = [-3.78086817  0.99990289]
iteration =61  weights = [-3.79182042  0.99990561]
iteration =62  weights = [-3.80222505  0.9999082 ]
iteration =63  weights = [-3.81210945  0.99991066]
iteration =64  weights = [-3.82149963  0.99991299]
iteration =65  weights = [-3.8304203   0.99991521]
iteration =66  weights = [-3.83889494  0.99991732]
iteration =67  weights = [-3.84694585  0.99991932]
iteration =68  weights = [-3.85459421  0.99992122]
iteration =69  weights = [-3.86186015  0.99992303]
iteration =70  weights = [-3.8687628   0.99992475]
iteration =71  weights = [-3.87532031  0.99992638]
iteration =72  weights = [-3.88154995  0.99992793]
iteration =73  weights = [-3.88746811  0.9999294 ]
iteration =74  weights = [-3.89309036  0.9999308 ]
iteration =75  weights = [-3.89843149  0.99993213]
iteration =76  weights = [-3.90350557  0.99993339]
iteration =77  weights = [-3.90832595  0.99993459]
iteration =78  weights = [-3.91290531  0.99993573]
iteration =79  weights = [-3.91725569  0.99993681]
```

```
iteration =80  weights = [-3.92138856  0.99993784]
iteration =81  weights = [-3.92531479  0.99993882]
iteration =82  weights = [-3.92904471  0.99993975]
iteration =83  weights = [-3.93258813  0.99994063]
iteration =84  weights = [-3.93595437  0.99994147]
iteration =85  weights = [-3.93915231  0.99994226]
iteration =86  weights = [-3.94219035  0.99994302]
iteration =87  weights = [-3.94507649  0.99994373]
iteration =88  weights = [-3.94781832  0.99994442]
iteration =89  weights = [-3.95042306  0.99994506]
iteration =90  weights = [-3.95289756  0.99994568]
iteration =91  weights = [-3.95524834  0.99994627]
Final weights

[-3.95748158  0.99994682]
```

Weighted Linear Regression using sklearn

In [96]:
```
from scipy import stats
import statsmodels.api as sm

data = np.loadtxt('LinearRegression_Sample.csv', delimiter=',')
#print data.shape
#print data

X = data[:,1]
y = data[:,0]

weights = np.absolute(np.reciprocal(X))
#print weights.shape
#print weights

X = sm.add_constant(X)

mod_wls = sm.WLS(y, X, weights=weights)
res_wls = mod_wls.fit()
print(res_wls.summary())
```

WLS Regression Results

Dep. Variable:	y	R-squared:	0.822
Model:	WLS	Adj. R-squared:	0.822
Method:	Least Squares	F-statistic:	4.629e+04
Date:	Sat, 17 Oct 2015	Prob (F-statistic):	0.00
Time:	08:16:44	Log-Likelihood:	-10011.
No. Observations:	10000	AIC:	2.003e+04
Df Residuals:	9998	BIC:	2.004e+04
Df Model:	1		

	coef	std err	t	P>|t|	[95.0% Conf.	Int.]
const	-4.0597	0.004	-970.367	0.000	-4.068	-4.051
x1	0.9999	0.005	215.157	0.000	0.991	1.009

Omnibus:	3882.143	Durbin-Watson:	1.978
Prob(Omnibus):	0.000	Jarque-Bera (JB):	1482977.381
Skew:	-0.516	Prob(JB):	0.00
Kurtosis:	62.650	Cond. No.	1.11

Plotting the 2 Regressions

The 2 models have been plotted below

- Model 1: Weighted Linear Regression using Gradient Descent (mrjob). Coefficients are:

```
beta_0 = -3.95748158
beta_1 = 0.99994682
```

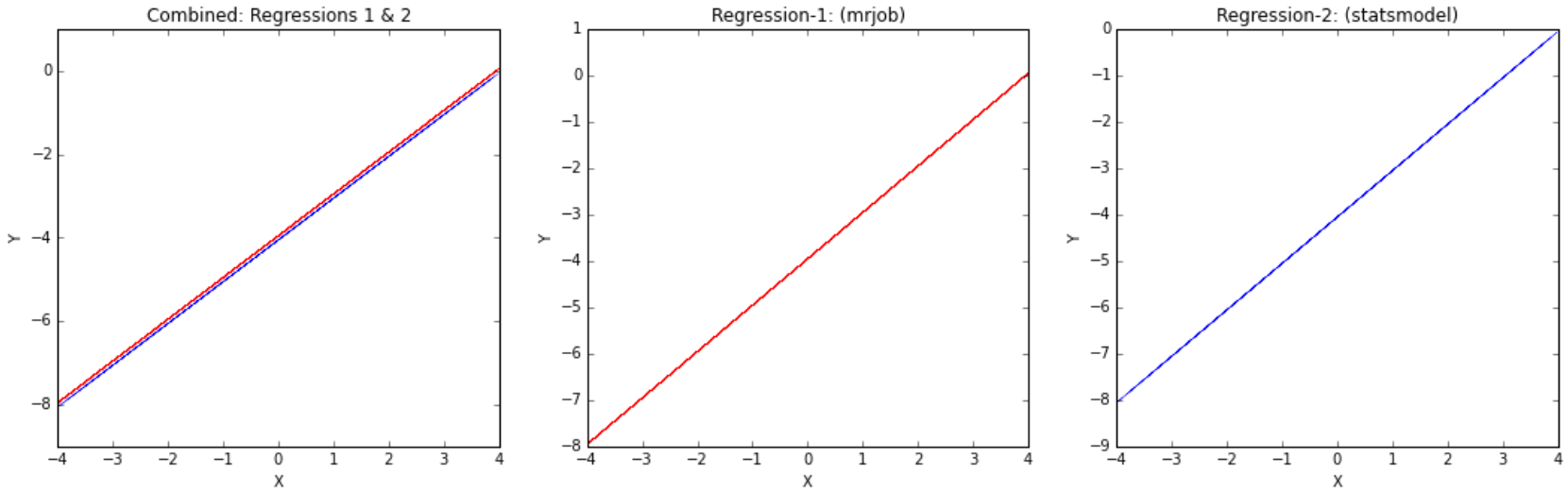
- Model 2: Weighted Linear Regression using statsmodel. Coefficients are:

```
beta_0 = -4.0597
beta_1 = 0.9999
```

The coefficients are very similar and therefore the 2 plotted lines look similar too. This is because we have no heteroskedasticity in our dataset generated using a normal distribution with mean=0 and sd=0.5

In [132]:
```
d1 = np.loadtxt('LinearRegression.csv', delimiter=',')
d2 = np.loadtxt('LinearRegression_Sample.csv', delimiter=',')

# First Equation using mrjob
X1 = d1[:,1]
y1 = d1[:,0]
y1_hat = -3.95748158 + 0.99994682 * X1

# Second Equation using statsmodel
X2 = d2[:,1]
y2 = d2[:,0]
y2_hat = -4.0597 + 0.9999 * X2

plt.figure(figsize=(18,5))
plt.subplot(131)
plt.title("Combined: Regressions 1 & 2")
plt.xlabel('X')
plt.ylabel('Y')
plt.plot(X1, y1_hat, 'r--', lw=0.05)
plt.plot(X2, y2_hat, 'b--', lw=0.05)

plt.subplot(132)
plt.title("Regression-1: (mrjob)")
plt.xlabel('X')
plt.ylabel('Y')
plt.plot(X1, y1_hat, 'r--', lw=0.05)

plt.subplot(133)
plt.title("Regression-2: (statsmodel)")
plt.xlabel('X')
plt.ylabel('Y')
plt.plot(X2, y2_hat, 'b--', lw=0.05)
```

```
Out[132]: [<matplotlib.lines.Line2D at 0x132cc13d0>]
```



HW6.5.1 (Optional)

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

```
In [ ]:
```