Course: Data mining & Knowledge Engineering

Final Project Laboratory

Type: **Group of 2 students**

Duration: **2 weeks**

Deadline: **23:59 Mon, Mar 14th, 2022**

Student 1: **Nguyen Vu Anh Thu** - *Student ID:* **1759038**

Student 2: **Quach Hoang Minh** - *Student ID:* **1859033**

Distributed optimization algorithms

The goal of this notebook is to work on distributed optimization algorithms, which are the foundation for large scale analytics and machine learning. Specifically, we will focus on the details of stochastic gradient descent (SGD). To do so, we will work on a simple regression problem, where we will apply SGD to minimize a loss function, as defined for the problem at hand. The emphasis of this laboratory is **not** on the machine learning part: even if you've never worked on regression problems, this shouldn't prevent you from being successful in developing the Notebook.

Next, an outline of the steps we will follow in this Notebook:

- Brief introduction to linear regression
- Implementation of serial algorithms: from Gradient Descent, to Stochastic Gradient Descent
- Implementation of distributed algorithms with Apache Spark

Initialization code

In [72]:
```
%matplotlib inline
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy import stats
from mpl_toolkits.mplot3d import axes3d
from pyspark import SparkContext, SparkConf
from sklearn.datasets import make_regression
```

A simple example: linear regression

Let's see briefly how to use gradient descent in a simple least squares regression setting. Asssume we have an output variable y which we think depends linearly on the input vector x. That is, we have:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix}$$

We approximate y_i by:

$$f_\theta(x_i) = \theta_1 + \theta_2 x_i$$

Define the loss function for the simple linear least squares regression as follows:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(f_\theta(x_i) - y_i)^2$$

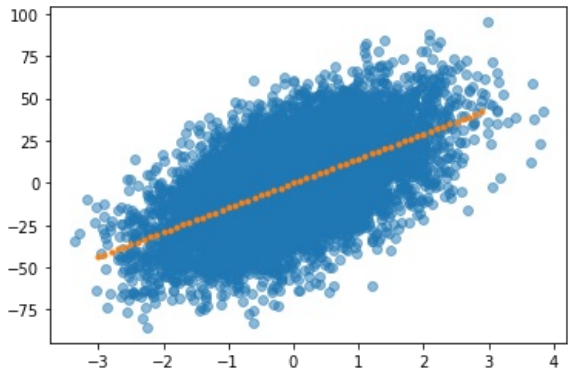
Now, let's use scikit learn to create a regression problem. A few notes are in order:

- The call to `make_regression` essentially generates samples for a regression problem
- The call to `stats.linegress` calculates a linear least-squares regression for two sets of measurements

This means we have a sort of "basline" to experiment with our SGD implementation.

In [73]:
```
x, y = make_regression(n_samples = 10000,
                       n_features=1,
                       n_informative=1,
                       noise=20,
                       random_state=2017)

x = x.flatten()
slope, intercept,_,_,_ = stats.linregress(x,y)
best_fit = np.vectorize(lambda x: x * slope + intercept)
plt.plot(x, y, 'o', alpha=0.5)
grid = np.arange(-3,3,0.1)
plt.plot(grid,best_fit(grid), '.')
```

Out[73]:
```
[<matplotlib.lines.Line2D at 0x7fe201fa1198>]
```



Batch gradient descent

Before delving into SGD, let's take a simpler approach. Assume that we have a vector of paramters θ and a loss function $J(\theta)$, which we want to minimize. The loss function we defined above has the form:

$$J(\theta) = \sum_{i=1}^{m} J_i(\theta)$$

where J_i is associated with the i-th observation in our data set, such as the one we generated above. The batch gradient descent algorithm, starts with some initial feasible parameter θ (which we can either fix or assign randomly) and then repeatedly performs the update:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)}) = \theta^{(t)} - \eta \sum_{i=1}^{m} \nabla_\theta J_i(\theta^{(t)})$$

where t is an iteration index, and η is a constant controlling step-size and is called the learning rate. Note that in order to make a **single update**, we need to calculate the gradient **using the entire dataset**. This can be very inefficient for large datasets, and it is the goal of this Notebook to insist on this aspect.

In code, the main loop for batch gradient descent looks like this:

```
for i in range(n_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

For a given number of iterations (also called epochs) n_e, we first evaluate the gradient vector of the loss function using **ALL** examples in the data set, and then we update the parameters with a given learning rate. Batch gradient descent is guaranteed to converge to the global minimum for convex loss surfaces and to a local minimum for non-convex surfaces.

In [74]:
```
def computeGradient(X,y,theta):
    '''
    Compute the gradient values based on fomula Grad = X*(X.T*theta - y)

    Arguments:
    X             -- input.
    Y             -- output.
    theta         -- weights.

    Return:
    Grad          -- Gradient values.
    '''
    Grad = X.dot(X.T.dot(theta)-y)/y.size

    return Grad
def updateWeights(w, Grad, lr):
    '''
    Update weights based on fomula w = w - lr*Grad
```

```
    Arguments:
    w             -- old weights.
    Grad          -- current gradient value.
    lr            -- learning rate.

    Return:
    w             -- updated (new) weights
    '''

    w = w - lr * Grad

    return w

def computeloss(X, y, theta):
    '''
    Compute the value of loss function given updated weight via loss = (X.T*theta - y)^2

    Arguments:
    X             -- input.
    y             -- output.
    theta         -- (updated) weights.

    Return:
    loss          -- value of loss function
    '''
    temp = np.dot(X.T, theta) - y
    loss =  0.5*np.dot(temp, temp.T)/y.size
    return loss



def isConverged(w1, w2, eps):
    '''
    Check whether or not convergence is reached by checking the relative
    difference between the current weight and the previous weight is less
    than convergence tolerance value. In measuring convergence, L2 norm is calculated.
    '''

    RelDiff = np.linalg.norm(w1 - w2)/len(w1)

    return (RelDiff < eps)
```

NOTE: Who computes the gradient?

Given a loss function $J(\theta)$, the gradient with respect to paramters θ must be derived manually. In other words, given the expression of $J(\theta)$, pencil and paper are required to derive the analytical form of its gradient. Then this expression can be plugged into our code. Recently, machine learning libraries have adopted the techniques of **automatic differentiation**, which eliminate this tedious and error prone step. Given a loss function $J(\theta)$, such libraries automatically compute the gradient.

For the linear regression case, let's derive the update step for gradient descent. Recall that we have defined:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(f_\theta(x_i) - y_i)^2$$

$$f_\theta(x_i) = \theta_1 + \theta_2 x_i$$

So we have that:

$$\nabla_\theta J(\theta) = \frac{1}{2}\sum_{i=1}^{m}\nabla_\theta(\theta_1 + \theta_2 x_i - y_i)^2$$

If we explicit the partial derivatives of the gradient, we have:

$$\frac{\partial J(\theta)}{\partial \theta_1} = \sum_{i=1}^{m}(\theta_1 + \theta_2 x_i - y_i)$$

$$\frac{\partial J(\theta)}{\partial \theta_2} = \sum_{i=1}^{m}(\theta_1 + \theta_2 x_i - y_i)x_i$$

So now we can explicit the update rules for the two model parameters:

$$\theta_1^{(t+1)} = \theta_1^{(t)} - \eta\sum_{i=1}^{m}(\theta_1^{(t)} + \theta_2^{(t)} x_i - y_i)$$

$$\theta_2^{(t+1)} = \theta_2^{(t)} - \eta\sum_{i=1}^{m}(\theta_1^{(t)} + \theta_2^{(t)} x_i - y_i)x_i$$

Using matrix notation

Now, before writing some code, let's see how can we simplify the above expressions using matrices. Note that this is not only useful for working on a more compact notation, but it helps reason about efficient computations using libraries such as numpy, which we will use extensively.

We firstly expressed our prediction as: $f_\theta(x_i) = \theta_1 + \theta_2 x_i$. Let's introduce a surrogate dimension for our input set x, such that:

$$x = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots & \dots \\ 1 & x_m \end{bmatrix}$$

where we define, with a small abuse of notation, $x_i = \begin{bmatrix} 1 & x_i \end{bmatrix}$. Also, let's define vector $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$.

Then, we can rewrite $f_\theta(x_i) = x_i\theta = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \theta_1 + \theta_2 x_i$.

Let's use this notation to rewrite our gradients in matrix form.

$$\nabla_\theta J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \nabla_\theta (x_i\theta - y_i)^2$$

$$= \sum_{i=1}^{m} (x_i\theta - y_i) x_i^T =$$

$$= x^T(x\theta - y) =$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_m \end{bmatrix} \left(\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots & \dots \\ 1 & x_m \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix} \right)$$

Numpy arrays

With the work we did above, we can now cast everything into numpy arrays, which are efficient, and for which an efficient implementation of vector and matrix operations exists. Specifically, above we used the traditional matrix notation, where we manipulate column vectors. Hence, we express matrix operations (namely matrix products) using the traditional "row-by-column" approach.

In numpy, we avoid this formalism by using dot product operations. So, given two column vectors:

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_m \end{bmatrix} b = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{bmatrix}$$

we have that $ab^T = a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_m b_m$, where $\cdot$ is the symbol we use for dot product.

Question 1. (1.0pt) Implement your own version of Gradient Descent, as a serial algorithm. Follow the guidelines below:

- Define a function to perform gradient descent. The function should accept as inputs: the training data x and y, the initial guess for the parameters (θ_1 and θ_2), the learning rate. Additional arguments include the definition of the maximum number of iterations before the algorithm stops, and a second stop condition on the marginal improvment on the loss.
- Keep track of the values of the loss, for each iteration.
- Keep track of the gradient values, for each iteration.

Once the $\nabla ient_d escent$ function is defined, you can generate input data according to the cell above, that use scikitlearn.
The output of your cell should contain the following information:

- The values of the paramters obtained through Gradient Descent optimization

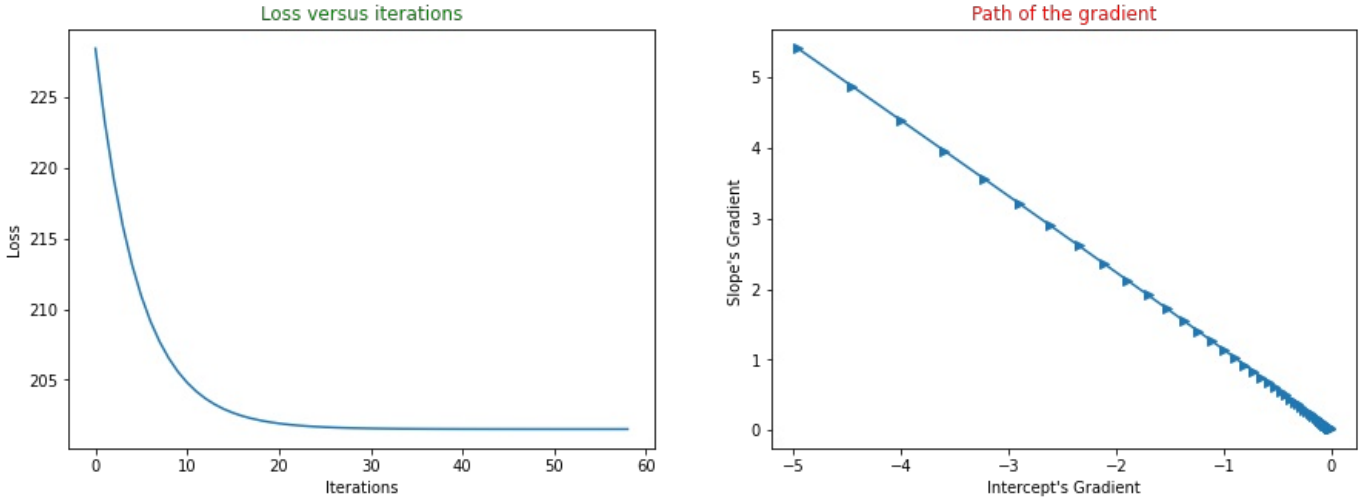
- The values of the paramters obtained with the above cell, using scikitlearn
- A plot of the loss versus iterations
- A plot of the path the gradient takes from its initial to its final position

In [75]:
```
def BGD(xy, theta_init, lr, eps, maxIter):

    '''
    An implementation of Batch Gradient Descent (BGD) algorithm that computes gradient
    values using all traning data, update weight values in the opposite direction
    of the gradient of the objective function and compute the value of loss function
    respect to updated weights. BGD algorithm will end before maxIter if the relative
    difference between the current weight and the previous weight is less than covergence tolerance (eps).

    Arguments:
    xy            -- training data for BGD.
    theta_init    -- initial weights.
    lr            -- learning rate.
    eps           -- covergence tolerance.
    maxIter       -- number of iterations that BGD should be run

    Return:
    thetas        -- A list of weight values, saving for each iteration.
    GDs           -- A list of gradient values, saving for each iteration.
    losses        -- A list of values of the loss function, saving for each iteration.
    '''

    # x, y are input and output data matrice respectively
    x, y = xy

    # X is extended input matrix by adding "1" column
    X = np.vstack((np.ones(len(x)),x.T))




    # history vectors
    thetas = [theta_init]
    GDs = []
    losses = [computeloss(X, y, theta_init)]


    for i in range(maxIter):

        # current 'updated' gradient values (newGrad)
        newGrad = computeGradient(X, y, thetas[-1]) #TODO
        GDs.append(newGrad)

        # updated weights (newtheta)
        newtheta = updateWeights(thetas[-1], GDs[-1], lr) #TODO
        thetas.append(newtheta)

        # current value of loss function (newloss) respect to updated weights
        newloss = computeloss(X, y, thetas[-1])
        losses.append(newloss)

        if isConverged(thetas[-1], thetas[-2], eps): #TODO
            break

    return np.array(thetas), np.array(GDs), np.array(losses)
```

In [76]:
```
# Parameter configuration
initialWeights = np.array([-5, 20], dtype=float)
stepSize = 1e-1
convergenceTol = 1e-3
numIterations = 100
data =  [x,y]



# implement Batch gradient descent with configed parameters
BGDWeights, BGDs, BGDlosses = BGD(data,initialWeights,stepSize,convergenceTol, numIterations) #TODO
[interceptBGD, slopeBGD] = BGDWeights[-1]

# The values of the paramters obtained through Batch Gradient Descent optimizationand scikitlearn
print("Values of the paramters using batch gradient descent:")
print("Slope: ", slopeBGD)
print("Intercept: ", interceptBGD)
print("Values of the paramters using scikitlearn:")
print("Slope: ", slope)
print("Intercept: ", intercept)

# Plot the loss function respect to iterations
fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(len(BGDlosses)), BGDlosses)
```

```
plt.title('Loss versus iterations', color='g', fontsize=12)
plt.xlabel("Iterations")
plt.ylabel("Loss")

# Plot the path the gradient takes from its initial to its final position
plt.subplot(1,2,2)
plt.plot(BGDs[:,0],BGDs[:,1],'->')
plt.title('Path of the gradient', color='r', fontsize=12)
plt.xlabel("Intercept's Gradient")
plt.ylabel("Slope's Gradient ")
plt.show()
```

```
Values of the paramters using batch gradient descent:
Slope:  14.528543728225225
Intercept:  -0.11672983262043739
Values of the paramters using scikitlearn:
Slope:  14.515169479593917
Intercept:  -0.10682356416090799
```



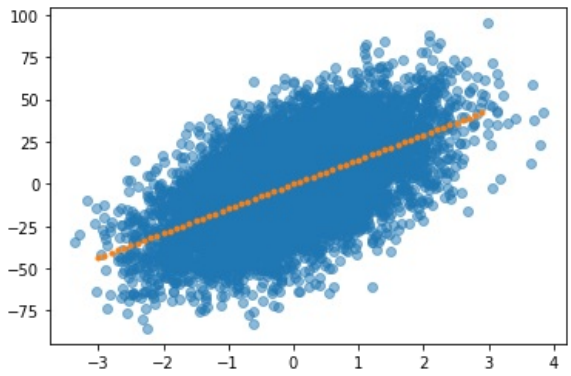
Comment:

Comment on the difference #TODO

Question 2. (1.0pt) Plot the regression line, along with the training data, given the coefficients θ that you have obtained with Gradient Descent.

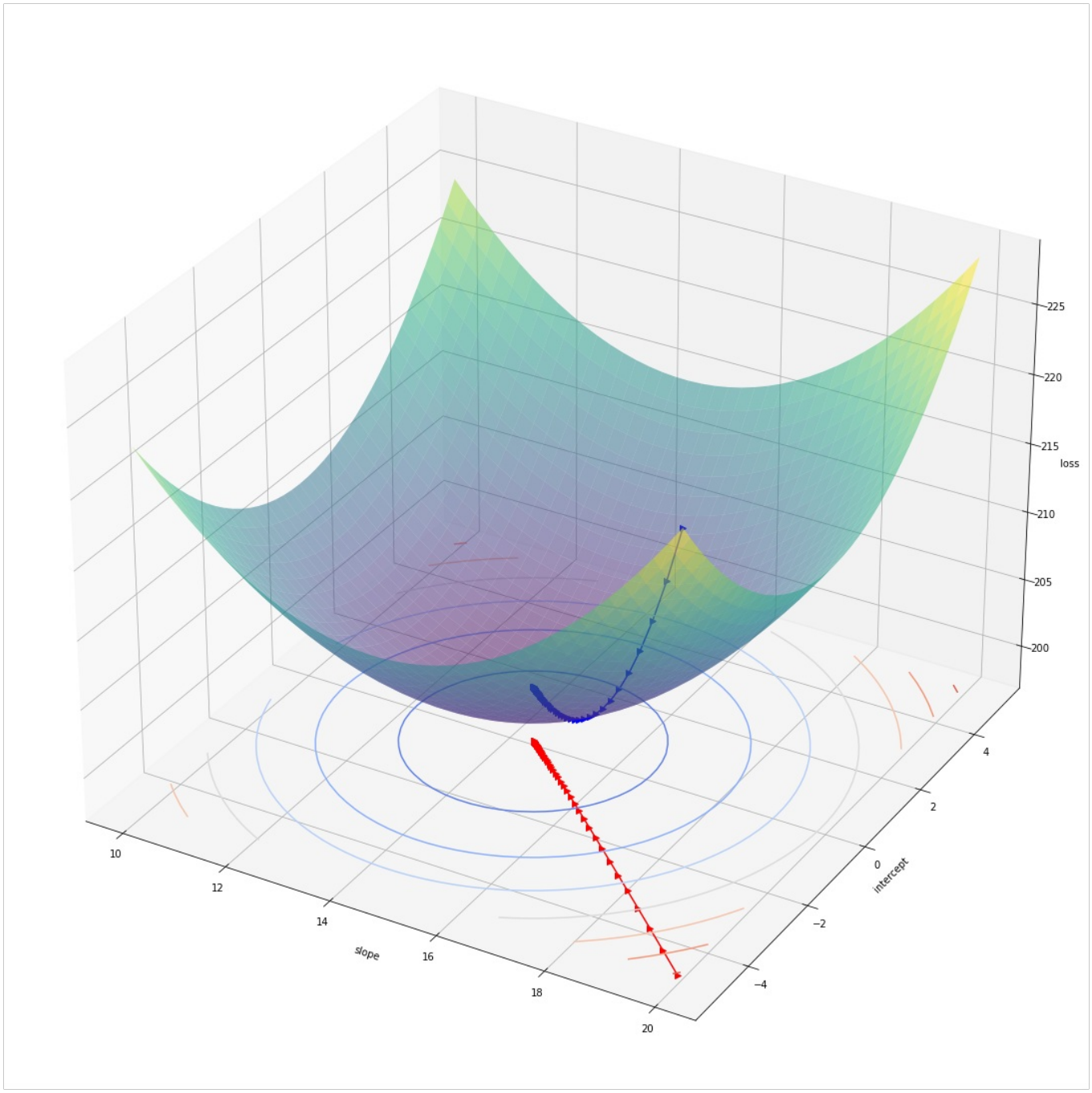
In [77]:
```
best_fit = np.vectorize(lambda x: x * slopeBGD + interceptBGD) #TODO khong hieu lam
plt.plot(x, y, 'o', alpha=0.5)
grid = np.arange(-3,3,0.1)
plt.plot(grid,best_fit(grid), '.')
```

Out[77]:
```
[<matplotlib.lines.Line2D at 0x7fe201dc8780>]
```



Question 3. (1.0pt) Plot a 3D surface representing: on the x,y axes the parameter values, on the z axis the loss value. Additionally, plot the trajectory of the loss function on the 3D surface, using the history you collected in the gradient_descent function you designed. Finally, plot a contour projection of the 3D surface, along with the corresponding projection of the trajectory followed by your Gradient Descent algorithm.

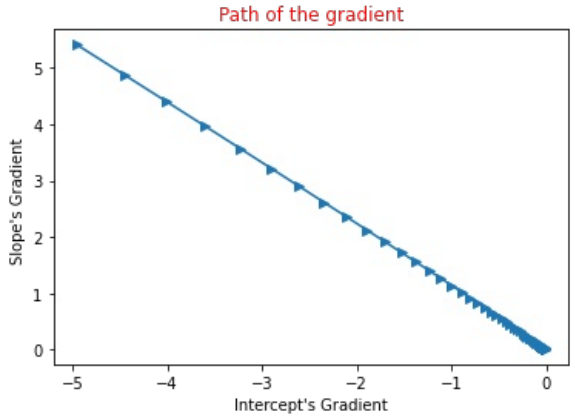
In [78]:
```
fig = plt.figure(figsize=(30,20))
ax = plt.axes(projection='3d')

s = np.linspace(10, 20, 40)
i = np.linspace(-5, 5, 40)

S, I = np.meshgrid(s, i)

X = np.vstack((np.ones(len(x)),x.T))
l = np.array([computeloss(X,y,[i, s]) for i, s in zip(np.ravel(I), np.ravel(S))]) #TODO
L = l.reshape(S.shape)

ax.plot_surface(S, I, L, rstride=1, cstride=1, alpha=0.5, cmap='viridis', edgecolor='none')
cset = ax.contour(S, I, L, zdir='z', offset=0.98*min(BGDlosses) , cmap='coolwarm')
plt.plot(BGDWeights[:,1],BGDWeights[:,0], BGDlosses, "b->")
plt.plot(BGDWeights[:,1],BGDWeights[:,0], 0.98*min(BGDlosses), "r->")

ax.set_xlabel('slope')
ax.set_ylabel('intercept')
ax.set_zlabel('loss')
plt.show()
```



Question 4. (1.0pt) Plot the path the gradient takes from its initial to its final position. This is a two dimensional plot (because our parameter vector has size 2), with a point for each gradient value, and a line connecting the points.

In [79]:
```
# Plot the path the gradient takes from its initial to its final position
```

```
plt.plot(BGDs[:,0],BGDs[:,1], '->') #TODO
plt.title('Path of the gradient',color='r',fontsize=12)
plt.xlabel("Intercept's Gradient")
plt.ylabel("Slope's Gradient ")
plt.show()
```



Stochastic Gradient Descent

The gradient descent algorithm makes intuitive sense as it always proceeds in the direction of steepest descent (the gradient of J) and guarantees that we find a local minimum (global under certain assumptions on J). When we have very large data sets, however, computing $\nabla_\theta J(\theta)$ can be computationally challenging: as noted above, we must process every data point before making a single step (hence the name "batch").

An alternative approach to alleviate such computational costs is the Stochastic Gradient Descent method: essentially, the idea is to update the parameters θ sequentially (one data point at the time), with every observation x_i, y_i. Following the same notation we used for Gradient Descent, the following expression defines how to update parameters, while processing one data point at the time:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J_i(\theta^{(t)})$$

The stochastic gradient approach allows us to start making progress on the minimization problem one step at the time. It is computationally cheaper, but it results in a larger variance of the loss function in comparison with batch gradient descent.

Generally, the stochastic gradient descent method will get close to the optimal θ much faster than the batch method, but will never fully converge to the local (or global) minimum. Thus the stochastic gradient descent method is useful when we are satisfied with an **approximation** for the solution to our optimization problem.

A full recipe for stochastic gradient descent follows:

```
for i in range(n_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

The reshuffling of the data is done to avoid a bias in the optimization algorithm by providing the data examples in a particular order.

Question 5. (1.0pt) Implement your own version of Stochastic Gradient Descent, as a serial algorithm. Follow the guidelines below:

- Define a function to perform gradient descent. The function should accept as inputs: the training data x and y, the initial guess for the parameters (θ_1 and θ_2), the learning rate. Additional arguments include the definition of the maximum number of iterations before the algorithm stops, and a second stop condition on the marginal improvment on the loss.
- Keep track of the values of the loss, for each iteration.
- Keep track of the gradient values, for each iteration.

Once the $\nabla ient_descent$ function is defined, you can generate input data according to the cell above, that use scikitlearn.
The output of your cell should contain the following information:

- The values of the paramters obtained through Gradient Descent optimization
- The values of the paramters obtained with the above cell, using scikitlearn
- A plot of the loss versus iterations
- A plot of the path the gradient takes from its initial to its final position

In [80]:
```
def SGD(xy, theta_init, lr, eps, maxIter):
    '''
```

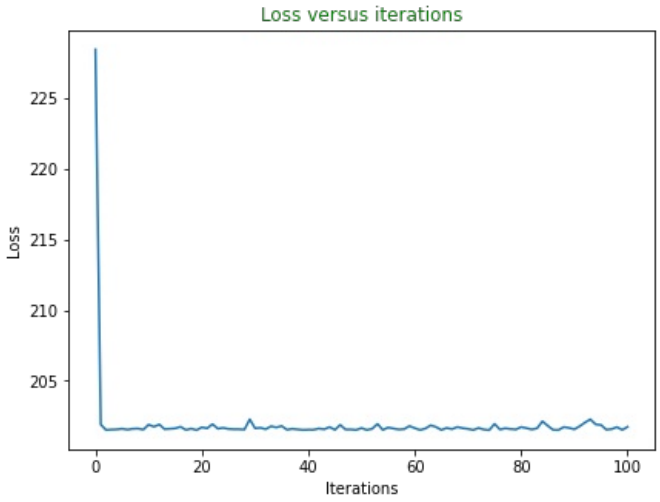
```
    An implementation of Stochastic Gradient Descent (SGD) algorithm that computes gradient
    values using each point of traning data, update weight values in the opposite direction
    of the gradient of the objective function and compute the value of loss function respect
    to updated weights. SGD algorithm will end before maxIter if the relative difference between
    the current weight and the previous weight is less than covergence tolerance (eps).

    Arguments:
    xy            -- training data for SGD.
    theta_init    -- initial weights.
    lr            -- learning rate.
    eps           -- covergence tolerance.
    maxIter       -- number of iterations that SGD should be run

    Return:
    thetas        -- A list of weight values, saving for each iteration.
    GDs           -- A list of gradient values, saving for each iteration.
    losses        -- A list of values of the loss function, saving for each iteration.
    '''
    # x, y are input and output data matrice respectively
    x, y = xy
    # X is extended input matrix by adding "1" column
    X = np.vstack((np.ones(len(x)),x.T))
    # xy is training data matrix (combine x and y)
    xy = np.vstack((x,y)).T

    # history vectors
    thetas = [theta_init]
    oldtheta = theta_init
    GDs = []
    losses = [computeloss(X, y, theta_init)]

    for i in range(maxIter):

        # shuffe training data after each epoch
        np.random.shuffle(xy)
        # accumulate gradient values
        accGrad = 0

        # xs, ys are input and output data sample (data point) in training data
        for xs, ys in xy:
            Xs= np.array([1, xs])
            # current 'updated' gradient values (newGrad)
            newGrad = computeGradient(Xs,ys,oldtheta) #TODO check lai
            # updated weights (newtheta)
            newtheta = updateWeights(oldtheta,newGrad, lr) #TODO
            # current value of loss function (newloss) respect to updated weights
            newloss = computeloss(Xs, ys, newtheta) #TODO
            accGrad += newGrad
            oldtheta = newtheta

        thetas.append(newtheta)
        # gradient value after each epoch is the mean of gradient value at each data point
        GDs.append(accGrad/len(x))
        losses.append(computeloss(X, y, thetas[-1]))

        if isConverged(thetas[-1], thetas[-2], eps): #TODO
            break
    return np.array(thetas), np.array(GDs), np.array(losses)
```

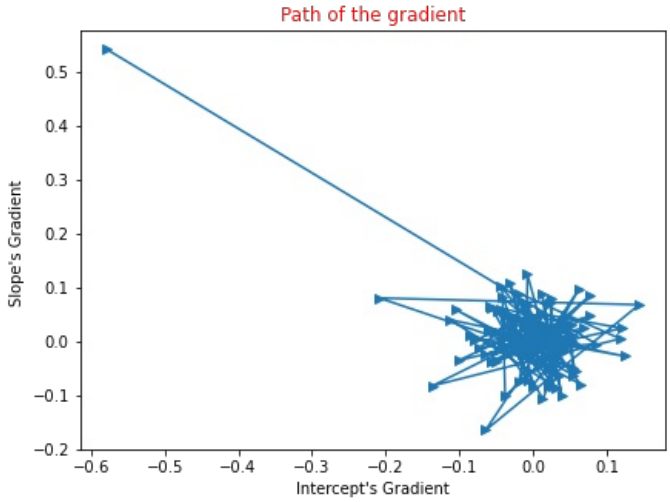
In [81]:

```
# Parameter configuration
initialWeights = np.array([-5, 20], dtype=float)
stepSize = 1e-3
convergenceTol = 1e-3
numIterations = 100
data = [x, y]

# implement Stochastic gradient descent with configed parameters
SGDWeights, SGDs, SGDlosses = SGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
[interceptSGD, slopeSGD] = SGDWeights[-1]

# The values of the paramters obtained through Gradient Descent optimizationand scikitlearn
print("Values of the paramters using Stochastic gradient descent:")
print("Slope: ", slopeSGD)
print("Intercept: ", interceptSGD)
print("Values of the paramters using scikitlearn:")
print("Slope: ", slope)
print("Intercept: ", intercept)

# Plot the loss function respect to iterations
fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(len(SGDlosses)), SGDlosses)
plt.title('Loss versus iterations',color='g',fontsize=12)
plt.xlabel("Iterations")
plt.ylabel("Loss")

# Plot the path the gradient takes from its initial to its final position
plt.subplot(1,2,2)
```

```
plt.plot(SGDs[:,0],SGDs[:,1],'->')
plt.title('Path of the gradient',color='r',fontsize=12)
plt.xlabel("Intercept's Gradient")
plt.ylabel("Slope's Gradient ")
plt.show()
```

```
Values of the paramters using Stochastic gradient descent:
Slope:  13.909128679524278
Intercept:  -0.4495138179460024
Values of the paramters using scikitlearn:
Slope:  14.515169479593917
Intercept:  -0.10682356416090799
```





Comment:

TODO

Mini-batch Stochastic Gradient Descent

Mini-batch gradient descent is a trade-off between stochastic gradient descent and batch gradient descent. In mini-batch gradient descent, the cost function (and therefore gradient) is averaged over a small number of samples, which is what we call the mini-batch, and that we denote by mb. This is opposed to the SGD batch size of 1 sample, and the BGD size of all the training samples.

Let's use the notation we introduced above to rewrite the gradients in matrix form for the mini-batch variant:

$$\nabla_\theta J(\theta) = \frac{1}{2mb}\sum_{i=1}^{mb}\nabla_\theta(x_i\theta - y_i)^2$$

$$= \sum_{i=1}^{mb}(x_i\theta - y_i)x_i^T =$$

$$= x_{mb}^T(x_{mb}\theta - y_{mb})$$

What's the benefit of doing it this way? First, it smooths out some of the noise in SGD, but not all of it, thereby still allowing the "kick" out of local minimums of the cost function. Second, the mini-batch size is still small, thereby keeping the performance benefits of SGD.

Question 6. (1.0pt) Implement your own version of Mini-batch Stochastic Gradient Descent, as a serial algorithm. Follow the guidelines below:

- Define a function to extract mini-batches from the training data.
- Define a function to perform gradient descent. The function should accept as inputs: the training data x and y, the initial guess for the parameters (θ_1 and θ_2), the learning rate. Additional arguments include the definition of the maximum number of iterations before the algorithm stops, and a second stop condition on the marginal improvment on the loss.
- Keep track of the values of the loss, for each iteration.
- Keep track of the gradient values, for each iteration.

Once the $\nabla ient_descent$ function is defined, you can generate input data according to the cell above, that use scikitlearn.
The output of your cell should contain the following information:

- The values of the paramters obtained through Gradient Descent optimization

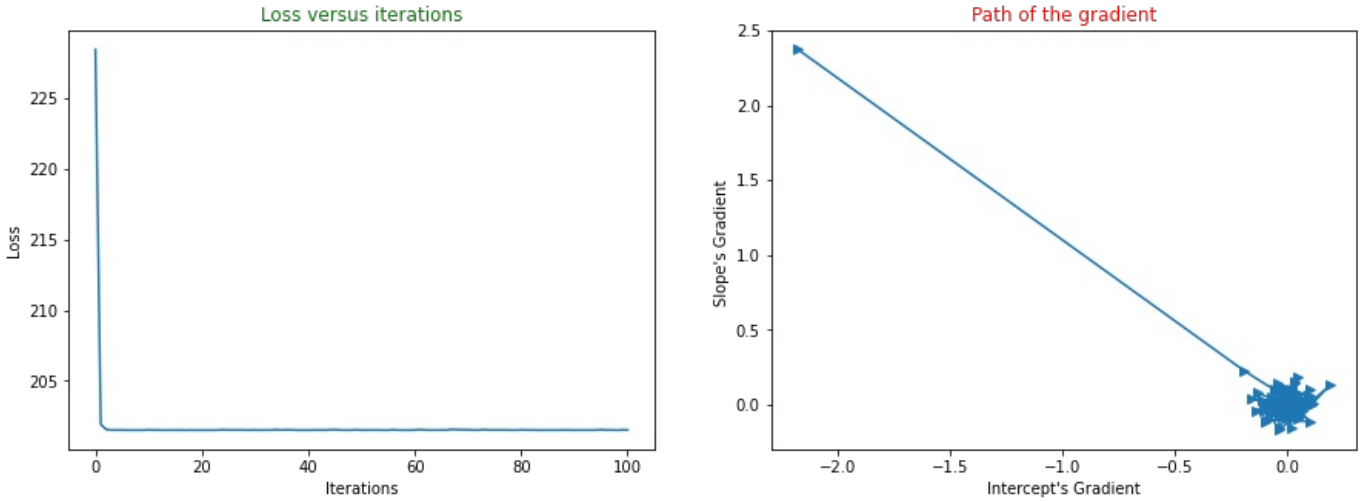
- The values of the paramters obtained with the above cell, using scikitlearn
- A plot of the loss versus iterations
- A plot of the path the gradient takes from its initial to its final position

In [82]:
```
def MBSGD(xy, theta_init, batch_size, lr, eps, maxIter):

    '''
    An implementation of Stochastic Gradient Descent (SGD) algorithm that computes gradient
    values using groups of data point in traning data, update weight values in the opposite
    direction of the gradient of the objective function and compute the value of loss function
    respect to updated weights. MBSGD algorithm will end before maxIter if the relative difference
    between the current weight and the previous weight is less than covergence tolerance (eps).

    Arguments:
    xy            -- training data for SGD.
    theta_init    -- initial weights.
    lr            -- learning rate.
    eps           -- covergence tolerance.
    maxIter       -- number of iterations that SGD should be run

    Return:
    thetas        -- A list of weight values, saving for each iteration.
    GDs           -- A list of gradient values, saving for each iteration.
    losses        -- A list of values of the loss function, saving for each iteration.
    '''

    # x, y are input and output data matrice respectively
    x, y = xy
    # X is extended input matrix by adding "1" column
    X = np.vstack((np.ones(len(x)),x.T))
    # xy is training data matrix (combine x and y)
    xy = np.vstack((x,y)).T

    # history vectors
    thetas = [theta_init]
    oldtheta = theta_init
    GDs = []
    losses = [computeloss(X, y, theta_init)]

    for i in range(maxIter):

        # shuffe training data after each epoch
        np.random.shuffle(xy)
        # accumulate gradient values
        accGrad = 0
        # number of batch in training data given batch size
        Nb_batches = int(len(xy)/batch_size)

        for bid in range(Nb_batches):
            batch = xy[bid*batch_size:(bid+1)*batch_size]
            xb = batch[:,0]
            yb = batch[:,1]
            Xb= np.vstack((np.ones(len(xb)),xb.T))

            # current 'updated' gradient values (newGrad)
            newGrad = computeGradient(Xb,yb,oldtheta) #TODO
            # updated weights (newtheta)
            newtheta = updateWeights(oldtheta,newGrad,lr) #TODO
            accGrad += newGrad
            oldtheta = newtheta

        thetas.append(newtheta)
        # gradient value after each epoch is the mean of gradient value at each minibatch of data points
        GDs.append(accGrad/Nb_batches)
        losses.append(computeloss(X, y, thetas[-1]))

        if isConverged(thetas[-1], thetas[-2], eps): #TODO
            break
    return np.array(thetas), np.array(GDs), np.array(losses)
```

In [83]:
```
# Parameter configuration
initialWeights = np.array([-5, 20], dtype=float)
stepSize = 1e-2
batchSize = 50
convergenceTol = 1e-3
numIterations = 100
data = [x, y]

# implement Stochastic gradient descent with configed parameters
MBSGDWeights, MBSGDs, MBSGDlosses = MBSGD(data, initialWeights,batchSize, stepSize, convergenceTol, numIterations
[interceptMBSGD, slopeMBSGD] = MBSGDWeights[-1]

# The values of the paramters obtained through Mini-batch Stochastic Gradient Descent optimizationand scikitlearn
print("Values of the paramters using mini-batch stochastic gradient descent:")
print("Slope: ", slopeMBSGD)
```

```
print("Intercept: ", interceptMBSGD)
print("Values of the paramters using scikitlearn:")
print("Slope: ", slope)
print("Intercept: ", intercept)

# Plot the loss function respect to iterations
fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(len(MBSGDlosses)), MBSGDlosses)
plt.title('Loss versus iterations',color='g',fontsize=12)
plt.xlabel("Iterations")
plt.ylabel("Loss")

# Plot the path the gradient takes from its initial to its final position
plt.subplot(1,2,2)
plt.plot(MBSGDs[:,0],MBSGDs[:,1],'->')
plt.title('Path of the gradient',color='r',fontsize=12)
plt.xlabel("Intercept's Gradient")
plt.ylabel("Slope's Gradient ")
plt.show()
```

```
Values of the paramters using mini-batch stochastic gradient descent:
Slope:  14.478286982488433
Intercept:  -0.23542772147590352
Values of the paramters using scikitlearn:
Slope:  14.515169479593917
Intercept:  -0.10682356416090799
```



Comment:

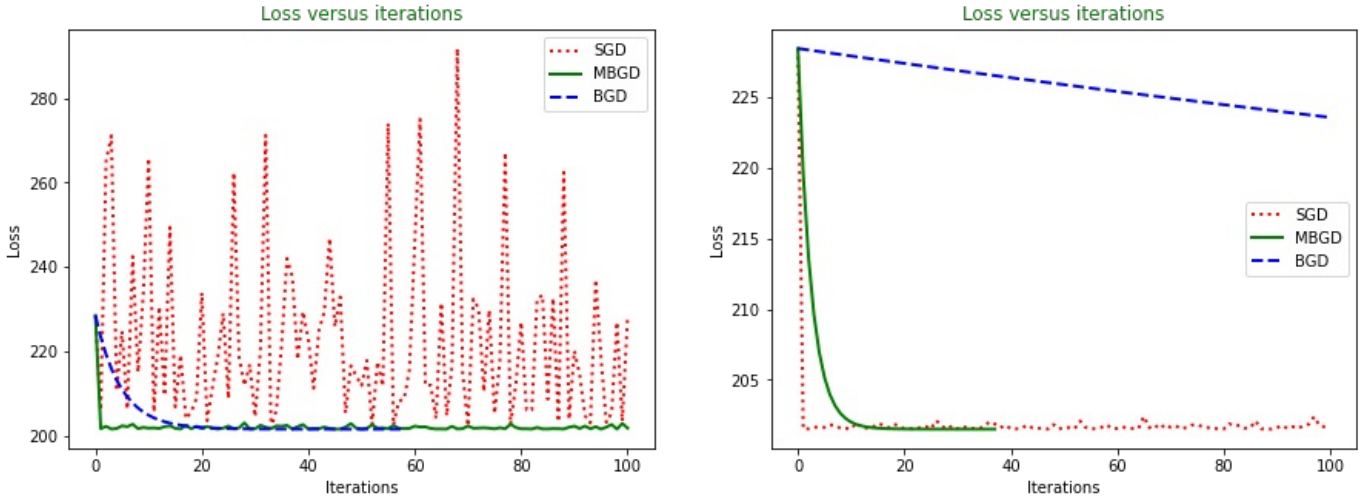
TODO

Question 7. (1.0pt) Compare the loss rate of the three approachs, Gradient Descent, Stochastic Gradient Descent, Mini-batch Stochastic Gradient Descent, by plotting in the same figure, the loss rate as a function of iterations. Comment the behavior of the three algorithms.

In [84]:

```
# Parameter configuration
initialWeights = np.array([-5, 20], dtype=float)
batchSize = 50
convergenceTol = 1e-3
numIterations = 100
data = [x, y]

stepSize = 1e-1
# implement Stochastic gradient descent with configed parameters
BGDWeights, BGDs, BGDlosses = BGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
[interceptBGD, slopeBGD] = BGDWeights[-1]

# implement Stochastic gradient descent with configed parameters
SGDWeights, SGDs, SGDlosses = SGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
[interceptSGD, slopeSGD] = SGDWeights[-1]

# implement Stochastic gradient descent with configed parameters
MBSGDWeights, MBSGDs, MBSGDlosses = MBSGD(data, initialWeights,batchSize, stepSize, convergenceTol, numIterations
[interceptMBSGD, slopeMBSGD] = MBSGDWeights[-1]

# Plot loss rate versus iteration of the three approachs: Gradient Descent, Stochastic Gradient Descent, Mini-bat
fig = plt.figure(figsize=(15,5))
```

```
plt.subplot(1,2,1)
plt.plot(np.arange(len(SGDlosses)), SGDlosses, color='r', linewidth=2, linestyle='dotted', label="SGD")
plt.plot(np.arange(len(MBSGDlosses)), MBSGDlosses , color='g', linewidth=2, linestyle='solid', label="MBGD")
plt.plot(np.arange(len(BGDlosses)), BGDlosses, color='b', linewidth=2, linestyle='dashed', label="BGD")
plt.legend()
plt.title('Loss versus iterations',color='g',fontsize=12)
plt.xlabel("Iterations")
plt.ylabel("Loss")

stepSize = 1e-3
# implement Stochastic gradient descent with configed parameters
BGDWeights, BGDs, BGDlosses = BGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
[interceptBGD, slopeBGD] = BGDWeights[-1]

# implement Stochastic gradient descent with configed parameters
SGDWeights, SGDs, SGDlosses = SGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
[interceptSGD, slopeSGD] = SGDWeights[-1]

# implement Stochastic gradient descent with configed parameters
MBSGDWeights, MBSGDs, MBSGDlosses = MBSGD(data, initialWeights,batchSize, stepSize, convergenceTol, numIterations
[interceptMBSGD, slopeMBSGD] = MBSGDWeights[-1]

# Plot loss rate versus iteration of the three approachs: Gradient Descent, Stochastic Gradient Descent, Mini-bat
plt.subplot(1,2,2)
plt.plot(np.arange(len(SGDlosses)), SGDlosses, color='r', linewidth=2, linestyle='dotted', label="SGD")
plt.plot(np.arange(len(MBSGDlosses)), MBSGDlosses , color='g', linewidth=2, linestyle='solid', label="MBGD")
plt.plot(np.arange(len(BGDlosses)), BGDlosses, color='b', linewidth=2, linestyle='dashed', label="BGD")
plt.legend()
plt.title('Loss versus iterations',color='g',fontsize=12)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()
```



Comment:

TODO

Distributed Mini-Batch Stochastic Gradient Descent

We're now ready to study the problem of distributed, mini-batch stochastic gradient descent. Clearly, so far we've worked with very small datasets: as a consequence it is hard to appreciate the computational bottleneck of serial implementations. But before we delve into an experimental setting, and try to spice things up with larger datasets, let's focus on the problem per se.

The problem statement is as follows: we need to design a distributed version of the algorithm we examined above, that is the Mini-Batch Stochastic Gradient Descent. In the interest of time, and also because in this laboratory we want to work with Apache Spark, we will look at **synchronous** distributed algorithms. Don'y worry, things will be much more clear in the next section!

The below reference constitutes the base for those of you interested in distributed optimization algorithms, an important element of large-scale machine learning. The reference is a research paper that was published in 2010, in a very important conference called NIPS. I suggest to have a look at section 1 only!

References

- Parallelized Stochastic Gradient Descent: http://martin.zinkevich.org/publications/nips2010.pdf

Some useful hints to design your distibuted algorithm

In this Notebook we will use Apache Spark, which is simply a fantastic tool! As we've learned in class (and this is actually true for Hadoop MapReduce as well as for any other distributed computing framework embracing the Bulk Synchrnous Parallel (BSP) programming model), Apache Spark features a synchronization barrier that really helps in dealing with distributed computations. To simplify the discussion (a much more profound treatment of the subject can be found in this nice book https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.466.8142&rep=rep1&type=pdf), in Apache Spark workers operate synchronously, in that the result of their computation, when aggregated or distributed, is processed only when **all** workers are done. So on the one hand, this simplifies the programming model, while on the other hand, this might introduce "stragglers", that is slow workers that penalize the performance of your distributed algorithm.

So what are the basic steps you should follow to design and implement your distributed algorithm?

Algorithm pseudo-code

In what follows, we outline the algorithm pseudo-code, to help you think about your algorithm implementation.

```
for i in range(iterations)

  # Randomly partition the input, giving T examples to each machine.
  mini_batches = sample(input_data)

  # Initialize random paramters
  params = initialize

  # This part executes in parallel, on each of the k worker machines
  # This is essentially the same code as for the serial implementation
  # Indeed, we're working on an embarassingly parallel formulation
  for data in mini_batch:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad

  # Here we aggregate from all machines, and compute the update for iteration i
  # The update is simply the average of all updates coming from each of the k worker machines
  Aggregate from all computers params = average(params_k)
```

Input data and parameters

As for the input data, we want to use the same procedure we used from the beginning of the Notebook: we use scikit learn to generate our x and y. Then, you have to find ways to **partition** this data across the machines you have in your cluster. The method `sc.parallelize` comes handy here.

Once you have a parallel collection, that takes the form of an `RDD`, you'll have to `sample` from it, to create random mini batches, each of which will be used **independently** by each worker machine to process its own data. Be careful that here we want to **keep the data partitioning** of our original data. You should be wise enough to lookup for the PySpark API to get the details of the methods we suggested here: `sample` and `mapPartitions`.

Last but not least, we have to take care about the parameters! The initial parameter values, as well as any updates to them, must be shipped to all worker machines. There are various ways to do this, exploiting the great flexibility of Apache Spark. This data could be serialized and shipped along with the code each task should execute. However, a much more efficient method to do this is to exploit what we have learned in class, namely `broadcast variables`. You might argue that for our simple Notebook, since the parameter vector is of size 2, this is not a big deal. However, for different scenarios, like big models, having an efficient primitive to ship a lot of data to workers is a big plus: remember that broadcast variables are sent to workers using a protocol very similar to BitTorrent, a famous peer-to-peer content distribution system.

Driver vs. workers

Now, to clarify even further what we expect from our Apache Spark program, here's some additional information about the roles of each component invovled in the execution of the pseudo-code outlined above.

Essentially, the `driver` machine (the one where the `SparkContext` is created), is the one responsible for executing the "serial" part of the algorithm: prepare and dispatch data, and wait for the contributions coming from all machines.

Instead, the `worker` machines receive their task to execute, proceed independently (some faster than others), and send their updates to the driver. The driver will **wait for all updates from workers** before proceeding with the final aggregation.

Additional hints

As you might have heard in class, Apache Spark has a lot of surprises. On of such cool methods allow you to define the way data is aggregated. Indeed, you surely realized that (in the old Hadoop MapReduce jargon) we're talking about a "map only job": workers "map" on their minibatch, computing partial gradient updates, then send everything (more or less at the same time) to the driver machine. At the scale of this Notebook, this is rather harmless. Instead, at larger scale, this could create "congestion" in the driver: a nice solution to this would be to have the possibility to aggregate data along a tree-structure, which would involve worker machines as well as the driver.

One final note. In case you wonder how to produce an `iterable` in python with a singleton in it, just `return (variable, )`.

Question 8. (2.0pts) Implement your own version of Mini-Batch, Stochastic Gradient Descent, as a **distributed** algorithm. The output of your cell should contain the following information:

- The values of the paramters obtained through Gradient Descent optimization
- The values of the paramters obtained with the above cell, using scikitlearn
- A plot of the loss versus iterations
- A plot of the path the gradient takes from its initial to its final position

In [85]:
```
# conf = SparkConf().setAppName("final").setMaster("local[*]")
# sc = SparkContext(conf=conf)
def distributed_mini_batch_SGD(xy, theta, lr, eps, max_iter):

    '''
    An implementation of distributed mini-batch Stochastic Gradient Descent (DMBSGD) algorithm
    that computes gradient values using groups of data point in traning data, update weight values in
    the opposite direction of the gradient of the objective function and compute the value of loss
    function respect to updated weights. These operations are implemented on each partition of cluster
    and then take average values. DMBSGD algorithm will end before max_Iter if the relative difference
    between the current weight and the previous weight is less than covergence tolerance (eps).

    Arguments:
    xy            -- training data for DMBSGD.
    theta         -- initial weights.
    lr            -- learning rate.
    eps           -- covergence tolerance.
    max_iter      -- number of iterations that DMBSGD should be run

    Return:
    thetas        -- A list of weight values, saving for each iteration.
    GDs           -- A list of gradient values, saving for each iteration.
    losses        -- A list of values of the loss function, saving for each iteration.
    '''
    # create read-only variables (weight and learning rate) cached on each machine
    bcTheta = sc.broadcast(theta)
    bcLr = sc.broadcast(lr)

    def mini_batch_stochastic_gradient_descent(data):

        '''
        An implementation of mini-batch Stochastic Gradient Descent (MBSGD) algorithm that computes
        gradient values using groups of data point in traning data, update weight values in the opposite directio
        of the gradient of the objective function and compute the value of loss function respect to updated weigh

        Arguments:
        data          -- training data for MBSGD.

        Return:
        newtheta      -- weight values.
        newloss       -- gradient values.
        newGrad       -- values of the loss function.
        1             -- count for taking average later
        '''

        xy = np.array(list(data))
        x = xy[:,0]
        y = xy[:,1]

        # x matrix with "1" column
        X = np.vstack((np.ones(len(x)),x.T))

        # previous weights
        oldtheta = bcTheta.value
        # current gradient values
        newGrad = computeGradient(X,y,oldtheta) #TODO
        # updated weights
        newtheta = updateWeights(oldtheta,newGrad,bcLr.value) #TODO
        # current values of loss function respect to updated weight
        newloss = computeloss(X, y, newtheta) #TODO

        return ([newtheta, newloss, newGrad, 1], )

    def stochastic_gradient_descent(data):

        '''
        An implementation of Stochastic Gradient Descent (SGD) algorithm that computes gradient values
        using each data point in traning data,  update weight values in the opposite direction of the
        gradient of the objective function and compute the value of loss function respect to updated weights.

        Arguments:
        data          -- training data for SGD.
```

```
        Return:
        newtheta      -- weight values.
        newloss       -- gradient values.
        newGrad       -- values of the loss function.
        1             -- count for taking average later
        '''

        # accumulate gradient values
        accGrad = 0
        # accumulate loss values
        accLoss = 0
        # previous weights
        oldtheta =  bcTheta.value
        for xs, ys in data:
            Xs= np.array([1, xs])
            # current gradient values
            newGrad = computeGradient(Xs,ys,oldtheta) #TODO
            # updated weights
            newtheta = updateWeights(oldtheta,newGrad,bcLr.value)
            # current value of loss function respect to updated weights
            newloss = computeloss(Xs, ys, newtheta) #TODO
            accGrad += newGrad
            accLoss += newloss
            oldtheta = newtheta

        return ([newtheta, accLoss/len(x), accGrad/len(x), 1], )

    # x, y are input and output data matrice respectively
    x, y = xy

    # X is extended input matrix by adding "1" column
    X = np.vstack((np.ones(len(x)),x.T))

    # data is training data matrix (combine x and y)
    data = np.vstack((x,y)).T

    # create parallelized collection whose form is RDD (dataRDD) based on training data (data)
    # divide the dataset into 6 partions using the number of partitions parameter
    Nb_partitions = 6
    dataRDD = sc.parallelize(data, Nb_partitions)

    #history vectors
    X = np.vstack((np.ones(len(x)),np.array(x).T))
    losses=[computeloss(X, np.array(y), bcTheta.value)]
    GDs=[]
    thetas=[bcTheta.value]

    # fraction of the input data set that should be used for one iteration of DMBSGD
    miniBatchFraction = 0.7

    # Before reaching stop condition (max_iter), at each itaration, 3 below steps will be implemented:
    # 1. A mini-batch data will be selected from data in each partition via 'sample' action given
    # miniBatchFraction parameter (the probability that a data point in partition will be selected into mini-batc
    # 2. Using 'mapPartitions' transformation to apply minibatch stochastic gradient descent algorithm
    # to computes gradient values, loss value, weights for each partition.
    # 3. Using 'reduce' action to take average gradient values, loss value, weights from the values in each
    # partition at step 2
    for i in range(max_iter):

        batchRDD = dataRDD.sample(False, miniBatchFraction, 42 + i)
        GradRDD = batchRDD.mapPartitions(mini_batch_stochastic_gradient_descent)
        res = GradRDD.reduce(lambda res1, res2: [res1[0]+res2[0], res1[1]+res2[1], res1[2]+res2[2], res1[3]+res2[

        thetas.append(res[0]/res[3])
        losses.append(res[1]/res[3])
        GDs.append(res[2]/res[3])

        # update broadcast weight for next iteration
        bcTheta = sc.broadcast(thetas[-1])

        if isConverged(thetas[-1], thetas[-2], eps): #TODO
            break

    return np.array(thetas), np.array(GDs), np.array(losses)
```

Comment:

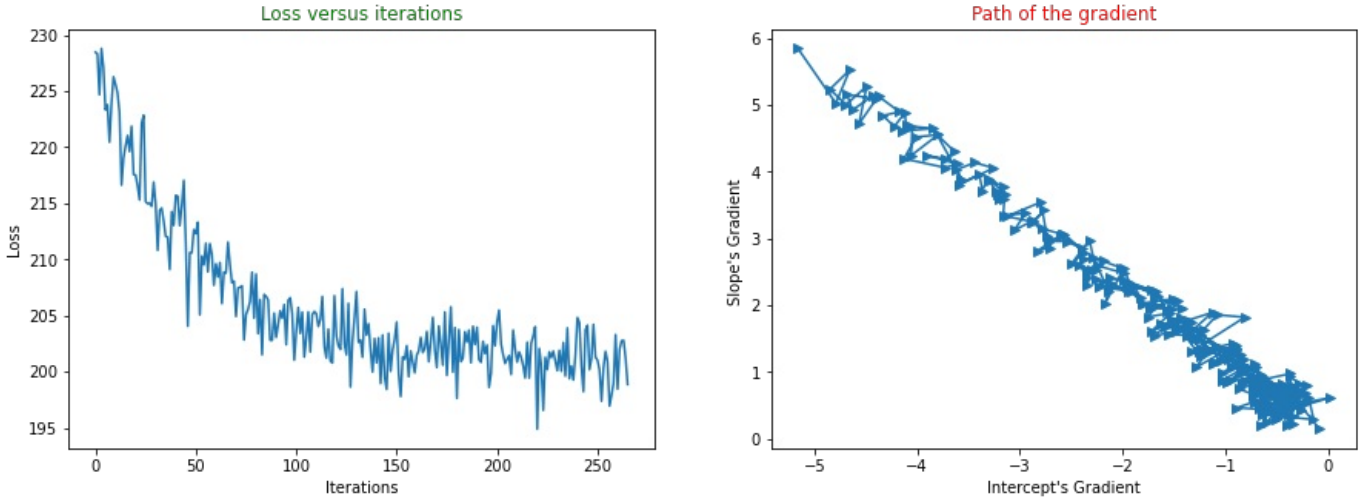
TODO

In [86]:

```
# Parameter configuration
initialWeights = np.array([-5, 20], dtype=float)
stepSize = 1e-2
convergenceTol = 1e-3
numIterations = 1000
```

```
data = [x, y]

# implement Distributed Mini-batch Stochastic gradient descent with configed parameters
DMBSGDWeights, DMBSGDs, DMBSGDlosses = distributed_mini_batch_SGD(data, initialWeights, stepSize, convergenceTol,
[interceptDMBSGD, slopeDMBSGD] = DMBSGDWeights[-1]

# The values of the paramters obtained through Distributed Mini-batch Stochastic Gradient Descent optimizationand
print("Values of the paramters using distributed mini-batch stochastic gradient descent:")
print("Slope: ", slopeDMBSGD)
print("Intercept: ", interceptDMBSGD)
print("Values of the paramters using scikitlearn:")
print("Slope: ", slope)
print("Intercept: ", intercept)

# Plot the loss function respect to iterations
fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(len(DMBSGDlosses)), DMBSGDlosses)
plt.title('Loss versus iterations',color='g',fontsize=12)
plt.xlabel("Iterations")
plt.ylabel("Loss")

# Plot the path the gradient takes from its initial to its final position
plt.subplot(1,2,2)
plt.plot(DMBSGDs[:,0],DMBSGDs[:,1],'->')
plt.title('Path of the gradient',color='r',fontsize=12)
plt.xlabel("Intercept's Gradient")
plt.ylabel("Slope's Gradient ")
plt.show()
```

```
Values of the paramters using distributed mini-batch stochastic gradient descent:
Slope:  14.855116522289881
Intercept:  -0.575557594861977
Values of the paramters using scikitlearn:
Slope:  14.515169479593917
Intercept:  -0.10682356416090799
```



Comment:

TODO

Question 9 (BONUS) (1.5pt). Inspecting your input data, and its partitioning.
Answer the following questions:

- What is the best way to partition your data, given your cluster configuration? How many partitions did you chose
- Given the number of partitions from the first point, how many RDD blocks is your data broken into?
- What would be different if you were to concieve the same algorithm digesting input data stored as a CSV file on HDFS? Argue about how to "ship" data to worker machines, partitions, blocks, etc...

[HINT] The questions above are best answered by looking at the Spark Master Web UI.

- What is the best way to partition your data, given your cluster configuration? How many partitions did you chose

**Ans: The best way to partition my data is running on yarn cluster, my cluster configuration is local mode. We choose divide the dataset into

6 partions

- Given the number of partitions from the first point, how many RDD blocks is your data broken into?

**Ans: 6 RDD blocks

- What would be different if you were to concieve the same algorithm digesting input data stored as a CSV file on HDFS? Argue about how to "ship" data to worker machines, partitions, blocks, etc...

Ans: ...

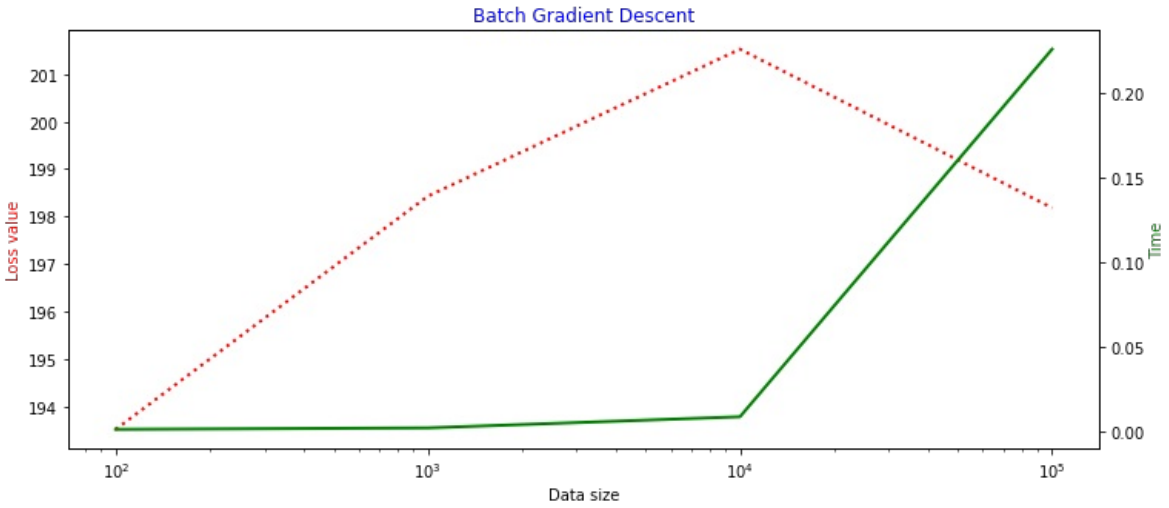
Question 10. (1.0pt) Comparison of serial vs. distributed algorithms.

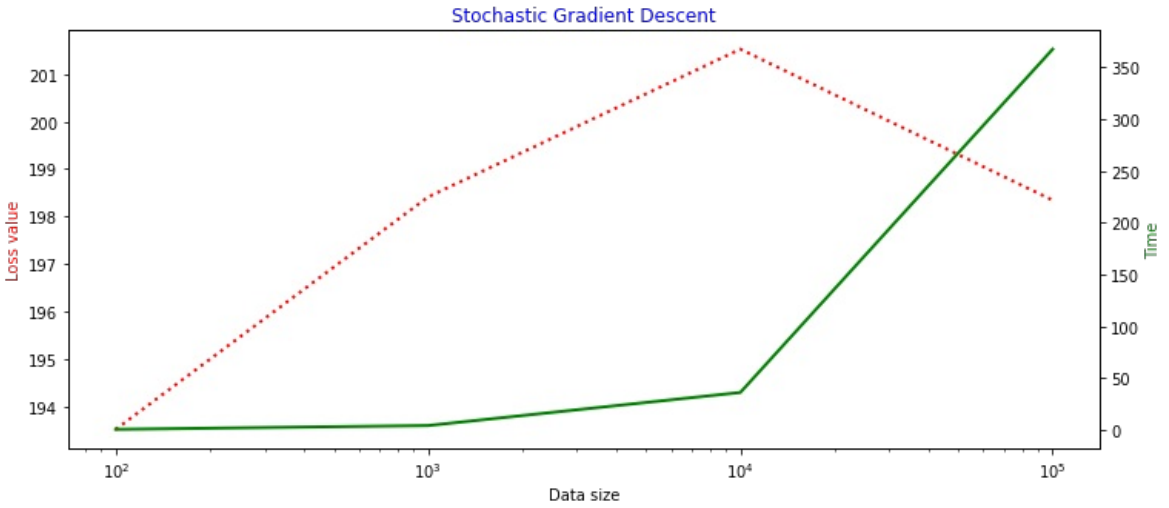
Given all the implementation effort you did so far, this question is about an experimental study on scalability. Given input data sizes in the range $[10^2, 10^3, 10^4, 10^5]$, collect the run-time and the loss rate at convergence for all variants of the serial algorithms and for the distributed one. Produce a plot with 2 y-axis, one for the run-time, one for the loss rate, and one x-axis with the input data size.

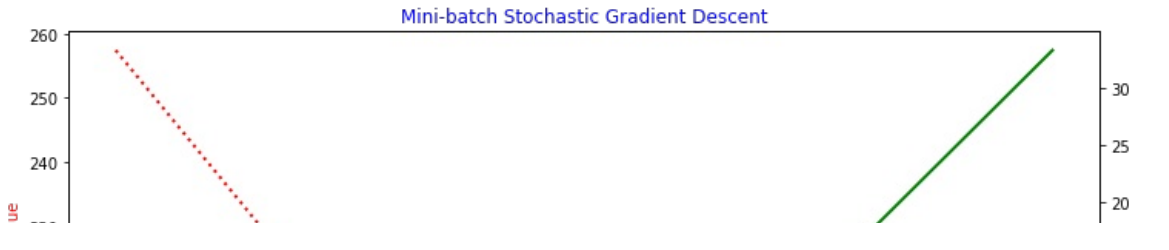
Discuss your results.

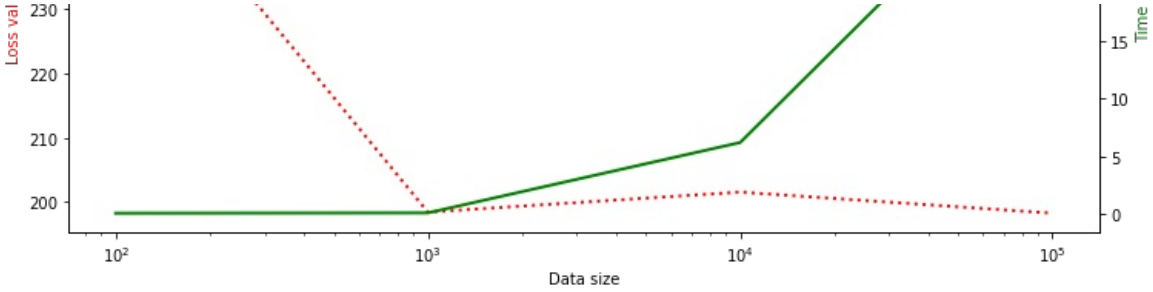
In [87]:
```
import time

def implement_algo(data_size, algo_name):
    '''
    An implementation of selected algorithm (algo_name)
    that computes loss function and running time given trainning data size.

    Arguments:
    data_size     -- size training data.
    algo_name     -- name of selected algorithm.

    Return:
    runtime       -- total running time.
    losses        -- the value of loss function.
    None          -- if algo_name is not eligible
    '''

    # create training data given data size
    x, y = make_regression(n_samples = data_size,
                           n_features=1,
                           n_informative=1,
                           noise=20,
                           random_state=2017)
    x = x.flatten()

    # parameter configuration
    initialWeights = np.array([-5, 20], dtype=float)
    batchSize = 50
    convergenceTol = 1e-2
    numIterations = 100
    data = [x, y]

    # implement selected algorithm
    if(algo_name == "Batch Gradient Descent"):
        before = time.time()
        stepSize = 1e-1
        _, _, losses = BGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
        after = time.time()
        runtime = after-before
    elif(algo_name == "Stochastic Gradient Descent"):
        before = time.time()
        stepSize = 1e-3
        _, _, losses = SGD(data, initialWeights, stepSize, convergenceTol, numIterations) #TODO
        after = time.time()
        runtime = after-before
    elif(algo_name == "Mini-batch Stochastic Gradient Descent"):
        before = time.time()
        stepSize = 1e-2
        _, _, losses = MBSGD(data, initialWeights,batchSize, stepSize, convergenceTol, numIterations) #TODO
        after = time.time()
        runtime = after-before
    elif(algo_name == "Distributed mini-batch Stochastic Gradient Descent"):
        before = time.time()
        stepSize = 1e-2
        _, _, losses = distributed_mini_batch_SGD(data, initialWeights, stepSize, convergenceTol, numIterations)
        after = time.time()
        runtime = after-before
    else:
        print('Do not match any algorithms')
        return None

    return runtime, losses[-1]
```

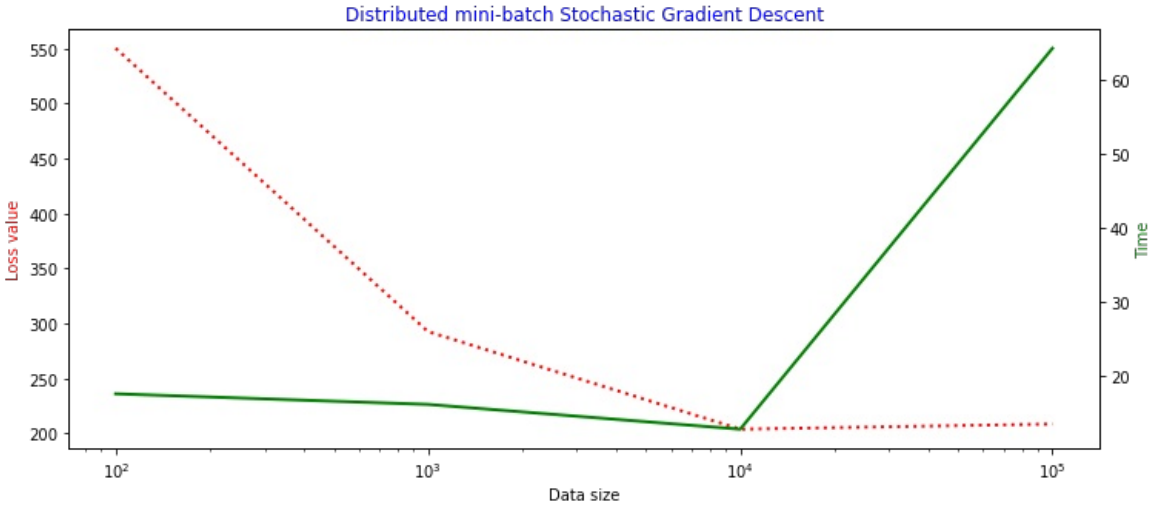
```python
def plot_algo(algo_name):

    '''
    Plot loss function and running time of selected algorithm (algo_name).

    Arguments:
    algo_name     -- name of selected algorithm.
    '''

    runtimes=[]
    losses=[]
    data_sizes = [100,1000,10000,100000]
    # implement selected algorithm (algo_name) given data size
    for size in data_sizes:
        runtime_algo, loss_algo = implement_algo(size, algo_name)
        runtimes.append(runtime_algo)
        losses.append(loss_algo)

    # plot the loss function and running time respect to data size
    fig = plt.figure(figsize=(12,5))
    plt.plot(data_sizes, losses, color='r', linewidth=2, linestyle='dotted')
    plt.xlabel("Data size")
    plt.ylabel("Loss value", color='r')
    plt.xscale("log")
    plt.twinx()
    plt.plot(data_sizes, runtimes , color='g', linewidth=2, linestyle='solid')
    plt.ylabel("Time", color='g')
    plt.title(algo_name,fontsize=12,color='b')
    plt.show()


plot_algo("Batch Gradient Descent")
plot_algo("Stochastic Gradient Descent")
plot_algo("Mini-batch Stochastic Gradient Descent")
plot_algo("Distributed mini-batch Stochastic Gradient Descent")
```











Comment:

TODO

Submission Instruction:

1. After you finished your work, clear all the outputs.
2. Run all cells again from the beginning to the end.
3. Make sure there are no error outputs or warnings. If there are questions that you don't want to answer, please remove the code cell and just leave the question cell.
4. Export your notebook as pdf format or html format. Open and check it again to make sure that it's readable.
5. Zip both the .ipynb file and the pdf/html file in one .zip file and name it as **SID1_SID2.zip**.
6. Remember to unzip it again to check the zip process was successful.
7. Submit it on Moodle.