Optimization Algorithms

The goal of this notebook is to work on 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 Batch Gradient Descent, to Stochastic Gradient Descent, and Mini-batch Stochastic Gradient Descent

Initialization code

```
%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 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:

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

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:

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

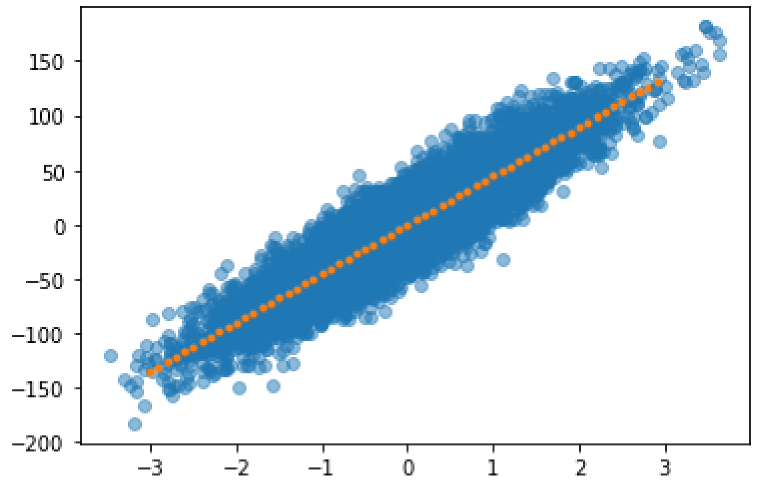
Now, let's use scikit learn to create a regression problem. A few notes are in order:

- The call to `make_regression` essentially generates samples for a regression problem
- The call to `stats.linregress` 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.

```
x, y = make_regression(n_samples = 10000,
                       n_features=1,
                       n_informative=1,
                       noise=20,
                       random_state=2021)
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), '.')
```

```
[<matplotlib.lines.Line2D at 0x7fba69661d50>]
```



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

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.

$$\nabla_\theta J(\theta) = \frac{1}{2}\sum_{i=1}^{m} \nabla_\theta (x_i\theta - y_i)^2$$
$$= \sum_{i=1}^{m}(x_i\theta - y_i)x_i^T =$$
$$= x^T(x\theta - y) =$$
$$= \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_m \end{bmatrix} \left(\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdots & \cdots \\ 1 & x_m \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_m \end{bmatrix} \right)$$

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:

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

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. 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 ```gradient_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:

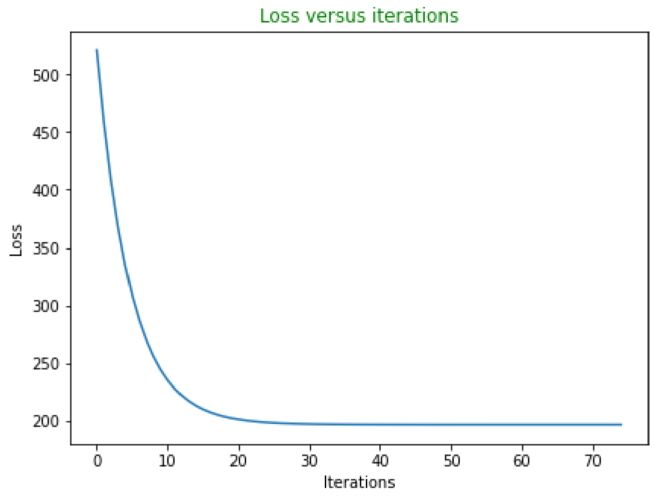
- The values of the paramters obtained through Gradient Descent optimization
- The values of the paramters obtained with the above cell, using scikitlearn

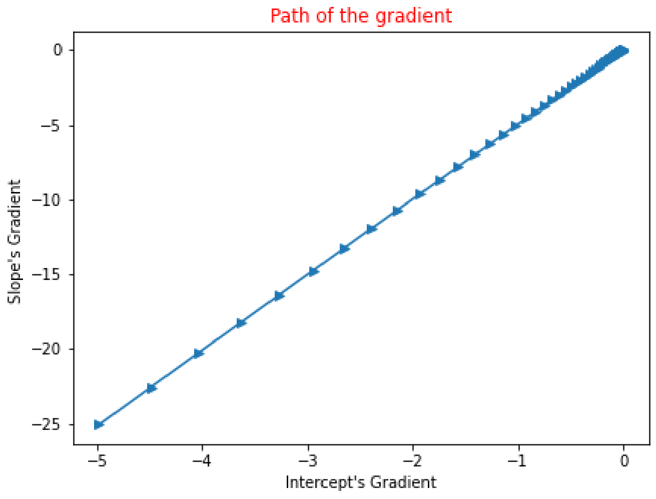
- A plot of the loss versus iterations
- A plot of the path the gradient takes from its initial to its final position

```
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.
    '''
    N = X.shape[0]
    return 1/N * X.T.dot(X.dot(theta) - y)

def updateWeights(w, Grad, eta):

    '''
    Update weights based on fomula w = w - eta*Grad

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

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

    w = w - eta * Grad
    return w

def computeloss(X, y, theta):

    '''
    Compute the value of loss function given updated weight via loss = (X.T*theta - y)

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

    Return:
    loss           -- value of loss function
    '''
```

```
        loss = np.sum((X.dot(theta) - y) ** 2) / 2.0
        loss = loss / X.shape[0]

        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)# use np.linalg.norm

        return (RelDiff < eps)


    def BGD(xy, theta_init, eta, 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 coverge

        Arguments:
        xy              -- training data for BGD.
        theta_init      -- initial weights.
        eta             -- 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 = xy[0]
        X = X.reshape(X.shape[0], 1)
        y = xy[1]
        one = np.ones((X.shape[0],1))
        Xbar = np.concatenate((one, X), axis = 1)

        thetas = [theta_init]
        GDs = []
        losses = [computeloss(Xbar, y, theta_init)]
        for epoch in range(maxIter):
            grad = computeGradient(Xbar, y, thetas[-1])
            theta_new = updateWeights(thetas[-1], grad, eta)
            if isConverged(thetas[-1], theta_new, eps):
                break

            losses.append(computeloss(Xbar, y, theta_new))
            GDs.append(grad)
            thetas.append(theta_new)

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


    # 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, numI
    [interceptBGD, slopeBGD] = BGDWeights[-1]

    # The values of the paramters obtained through Batch Gradient Descent optimizationand
    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:  44.864252418137994
Intercept:  -0.014436202294262027
Values of the paramters using scikitlearn:
Slope:  44.873771053464
Intercept:  -0.0123734058920343
```





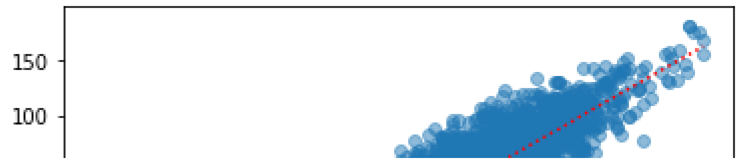
Comments:

Tính toán nhanh

Nếu lượng dữ liệu quá lớn sẽ không thể tính do tràn số

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

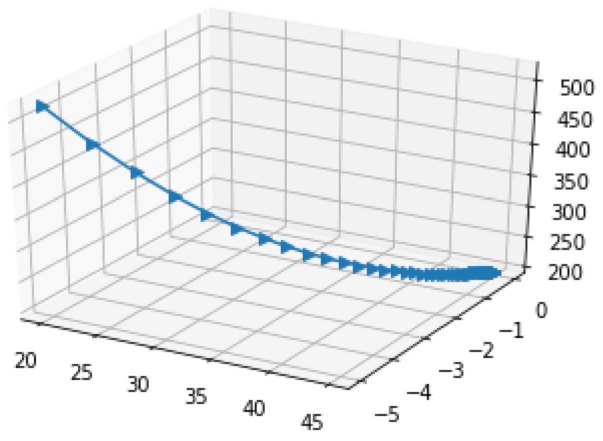
```
x_min = x[np.argmin(x)]
x_max = x[np.argmax(x)]
regression_line_x = [x_min, x_max]
regression_line_y = [interceptBGD + slopeBGD * rlx for rlx in regression_line_x]
plt.plot(x,y, 'o', alpha=0.5)
plt.plot(regression_line_x, regression_line_y, color = 'r', linestyle = 'dotted')
plt.show()
```



Question 3. 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.

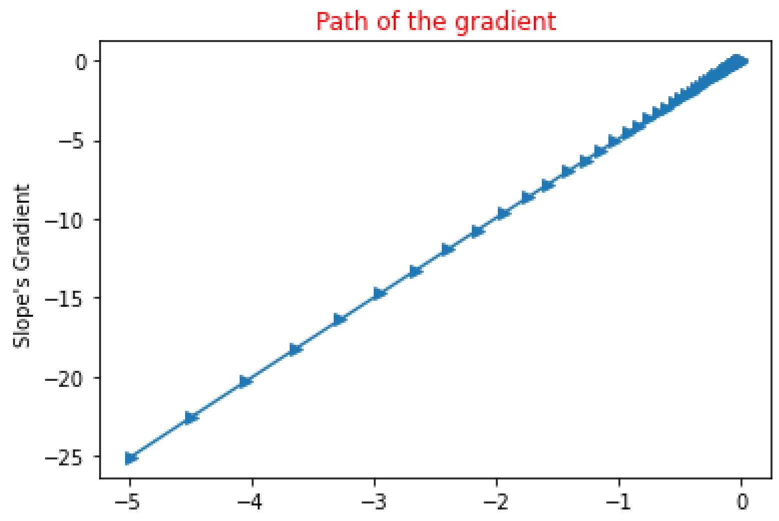
```
#hint:
# figsize=(30,20)
# use plt.axes(projection='3d'), plot_surface with cmap='viridis', and remember to set
ax = plt.axes(projection="3d")

theta_3D_y = BGDWeights[:, 0]
theta_3D_x = BGDWeights[:, 1]
loss_3D_z = BGDlosses

ax.plot(theta_3D_x, theta_3D_y, loss_3D_z, '->')
plt.show()
```



Question 4. 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.

```
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()
```



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 sample in data:
    params_grad = evaluate_gradient(loss_function, sample, 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. 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 ```gradient_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

```
def computeGradient(x, y,theta):

    '''
    Compute the gradient values based on fomula Grad = theta1 + theta2 * x - y =>

    Arguments:
    x              -- input.
    y              -- output.
    theta          -- weights.

    Return:
    Grad           -- Gradient values.
    '''
    t = theta[0] + theta[1] * x[1] - y
    return np.array([t, t * x[1]])

def updateWeights(w, Grad, eta):

    '''
    Update weights based on fomula w = w - eta*Grad

    Arguments:
```

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

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

    return w - eta * Grad

def computeloss(X, y, theta):

    '''
    Compute the value of loss function given updated weight via loss = (X.T*theta - y)

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

    Return:
    loss          -- value of loss function
    '''

    loss = np.sum((X.dot(theta) - y) ** 2) / 2.0
    loss = loss / X.shape[0]

    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)# use np.linalg.norm

    return (RelDiff < eps)


def SGD(xy, theta_init, eta, eps, maxIter):
    '''
    An implementation of Stochastic Gradient Descent (SGD) algorithm that computes gra
    values using each point of traning data, update weight values in the opposite dire
    of the gradient of the objective function and compute the value of loss function r
    to updated weights. SGD algorithm will end before maxIter if the relative differen
    the current weight and the previous weight is less than covergence tolerance (eps)

    Arguments:
    xy            -- training data for SGD
    xy              -- training data for SGD.
    theta_init      -- initial weights.
    eta             -- 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 = xy[0]
    X = X.reshape(X.shape[0], 1)
    y = xy[1]
    one = np.ones((X.shape[0],1))
    Xbar = np.concatenate((one, X), axis = 1)
    trainData = np.c_[ Xbar, y]

    thetas = [theta_init]
    GDs = []
    losses = [computeloss(Xbar, y, theta_init)]
    N = trainData.shape[0]

    for epoch in range(maxIter):
        np.random.shuffle(trainData)
        Xbar = trainData[:, :2]
        y = trainData[:, 2]

        theta = thetas[-1]
        grad = []
        grads = []
        for idx in range(N):
            grad = computeGradient(Xbar[idx], y[idx], theta)
            theta = updateWeights(theta, grad, eta)
            grads.append(grad)

        if isConverged(theta[-1], theta, eps):
            break

        GDs.append(np.average(grads, axis = 0))
        thetas.append(theta)
        losses.append(computeloss(Xbar, y, theta))

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


# 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, numI
[interceptSGD, slopeSGD] = SGDWeights[-1]

# The values of the paramters obtained through Gradient Descent optimizationand scikit
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:
```

Comments:

Tính toán chậm hơn nhiều so với BGD

It move quicky to "near converged", only after the first interation but not achive completely converge

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. 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 ```gradient_descent``` function is defined, you can generate input data according to the cell above, that use scikitlearn.

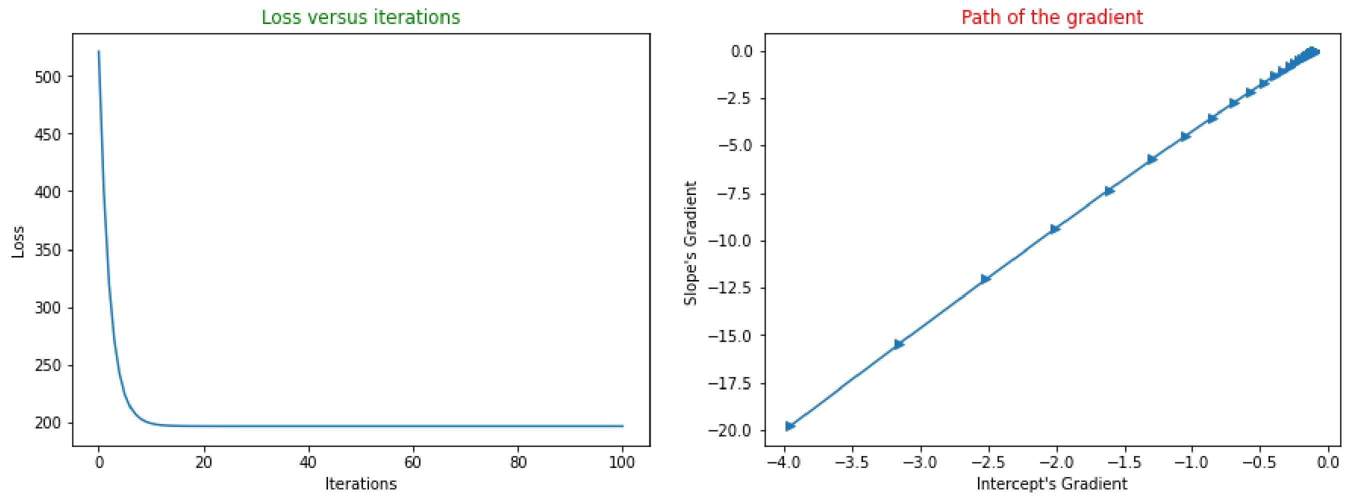
The output of your cell should contain the following information:

- 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

```
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.
    '''
    N = X.shape[0]
    return 1/N * X.T.dot(X.dot(theta) - y)

def updateWeights(w, Grad, eta):

    '''
    Update weights based on fomula w = w - eta*Grad

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

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

    w = w - eta * Grad
    return w

def computeloss(X, y, theta):

    '''
    Compute the value of loss function given updated weight via loss = (X.T*theta - y)

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

    Return:
    loss          -- value of loss function
```

```
        '''

        loss = np.sum((X.dot(theta) - y) ** 2) / 2.0
        loss = loss / X.shape[0]

        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)# use np.linalg.norm

        return (RelDiff < eps)

    def extractMiniBatchData(train_data, batch_size):
      return np.split(train_data, batch_size)


    def MBSGD(xy, theta_init, batch_size, eta, eps, maxIter):

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

        Arguments:
        xy             -- training data for SGD.
        theta_init     -- initial weights.
        eta            -- 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 = xy[0]
        X = X.reshape(X.shape[0], 1)
        y = xy[1]
        one = np.ones((X.shape[0],1))
        Xbar = np.concatenate((one, X), axis = 1)
        trainData = np.c_[ Xbar, y]
        thetas = [theta_init]
        GDs = []
        losses = [computeloss(Xbar, y, theta_init)]

        mini_batch_data_train = extractMiniBatchData(trainData, batch_size)

        for epoch in range(maxIter):

            theta = thetas[-1]

            grads_min_train_data = []
            losses_min_train_data = []
            theta_min_train_data = []
            for mini_train_data_idx in range(batch_size):
                mini_train_data = np.array(mini_batch_data_train[mini_train_data_idx])
                Xbar = mini_train_data[:, :2]
                y = mini_train_data[:, 2]

                grad = computeGradient(Xbar, y, theta)
                theta = updateWeights(theta, grad, eta)
                grads_min_train_data.append(grad)
                losses_min_train_data.append(computeloss(Xbar, y, theta))
                theta_min_train_data.append(theta)

            if isConverged(theta[-1], theta, eps):
                break

            GDs.append(np.average(grads_min_train_data, axis = 0))
            thetas.append(np.average(theta_min_train_data, axis = 0))
            losses.append(np.average(losses_min_train_data))

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


    # 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, co
    [interceptMBSGD, slopeMBSGD] = MBSGDWeights[-1]

    # The values of the paramters obtained through Mini-batch Stochastic Gradient Descent
    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:  44.818547200482925
Intercept:  -0.1143932063166192
Values of the paramters using scikitlearn:
Slope:  44.873771053464
Intercept:  -0.0123734058920343
```



Comments:

Caculate quite quickly than SGD

Not affected by noise as SGD

Question 7. 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

```
# 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, numI
[interceptBGD, slopeBGD] = BGDWeights[-1]

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

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

# Plot loss rate versus iteration of the three approachs: Gradient Descent, Stochastic
fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(len(SGDlosses)), SGDlosses, color='r', linewidth=2, linestyle='dott
plt.plot(np.arange(len(MBSGDlosses)), MBSGDlosses , color='g', linewidth=2, linestyle=
plt.plot(np.arange(len(BGDlosses)), BGDlosses, color='b', linewidth=2, linestyle='dash
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, numI
[interceptBGD, slopeBGD] = BGDWeights[-1]

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

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

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

Comments:

BGD:

- Hội tụ chậm hơn SGD, MiniGD
- Không bị ảnh hưởng bởi nhiễu
- Trường hợp learning rate thấp thì tốc độ hội tụ chậm

SGD:

- Hội tụ nhanh mặc dù learning rate thấp
- Bị ảnh hưởng bởi nhiễu.

MiniGD:

- Hội tụ nhanh
- không bị ảnh hưởng bởi nhiễu
- Trường hợp learning rate thấp thì tốc độ hội tụ chậm