Python Tutorial on Implementation of Gradient Desecent Optimization algorithms from Scratch

Example : Implementation of Gradient Descent Algorithms from scratch in Python

Table of Contents

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

1.Create Data

In [15]:

```
# generate random data with some noise

X = 2 * np.random.rand(100, 1)
print(X.shape)
# print(X)
y = 8 +(3*X) + (np.random.randn(100, 1))
print(y.shape)

## For printing together input  and target
X_y=np.c_[X,y]
print("Input and target")
print(X_y)
```

```
(100, 1)
(100, 1)
Input and target
[[ 0.04497682  9.10423587]
 [ 0.44369684  9.91643706]
 [ 1.14757428 11.57207686]
 [ 1.98332104 12.02308413]
 [ 0.03038902  9.2510614 ]
 [ 1.06406009 10.54856201]
 [ 0.76146702 11.78215315]
 [ 0.92371199  9.46087616]
 [ 1.14317242 11.56766946]
 [ 1.55185131 12.12205791]
 [ 0.0186962   7.2777673 ]
 [ 1.1789661  11.50784111]
 [ 0.12408403  7.41553359]
 [ 1.19483023 12.57702559]
 [ 1.59064174 14.23886903]
 [ 1.40627822 12.4646239 ]
 [ 0.41236386  9.66891799]
 [ 0.01554289  7.50275661]
 [ 1.24460193 12.63655415]
 [ 0.59811763 11.0568907 ]
 [ 0.41969231  9.65690536]
 [ 1.36654913 12.13159462]
 [ 1.64921642 13.65952935]
 [ 0.35949318  6.70617922]
 [ 0.02194567  8.71344024]
 [ 0.19417357  7.8774343 ]
 [ 0.43027749  8.26604118]
 [ 0.01570807  7.77699381]
 [ 1.8942345  14.27457547]
 [ 1.84577681 13.78644959]
 [ 1.0891461  11.9357483 ]
 [ 0.94948277 10.53574713]
 [ 1.8847801  13.73178354]
 [ 1.26666735 11.90012728]
 [ 0.40107006  7.84937596]
 [ 1.44085866 11.71417226]
 [ 1.61521082 13.30733268]
 [ 1.50202438 11.5954719 ]
```

```
[ 0.88315884 11.46431646]
[ 0.47911189  9.47640518]
[ 0.30588616 10.13541128]
[ 1.52129161 12.84197568]
[ 0.25075942  7.88154753]
[ 0.90823295 10.79025615]
[ 1.63317685 12.44083037]
[ 1.68238394 14.15681462]
[ 1.36562465 12.50388206]
[ 0.77130311 12.89670151]
[ 1.10717202 10.36614913]
[ 0.23961462  7.03771564]
[ 0.55449143  8.97155405]
[ 1.96119551 13.11300052]
[ 1.02226458 10.56326872]
[ 0.13509331  9.68292579]
[ 0.27385538  7.45643523]
[ 0.68252181 10.8104117 ]
[ 0.47555176  9.06428252]
[ 1.09838153 11.87228133]
[ 1.08531632 11.92645865]
[ 0.75247426  8.81934867]
[ 1.99022675 15.21941186]
[ 1.99980551 12.91520817]
[ 1.5864137  12.94750455]
[ 1.66776477 13.05471928]
[ 0.39009362  9.5872106 ]
[ 0.06842432  6.52626061]
[ 0.49160231  8.96437506]
[ 1.89122417 13.97621451]
[ 0.29577358  9.39384895]
[ 1.21440789 11.27633599]
[ 1.79309467 11.8779532 ]
[ 1.63039219 12.65629986]
[ 0.81381444 11.28511091]
[ 1.0785014  11.64870513]
[ 0.65317283 10.92945199]
[ 1.39720473 10.49816875]
[ 0.72273399 10.0385813 ]
[ 1.56516876 11.22598596]
[ 1.79435949 13.69041692]
[ 0.92515284 10.69258089]
```

```
 [ 0.92666572 11.16743518]
 [ 1.79482058 14.3119088 ]
 [ 0.83466322 10.25302971]
 [ 0.89994989  9.90721344]
 [ 0.05247682  7.3231602 ]
 [ 1.34408815 12.47571037]
 [ 1.32256478 11.96633382]
 [ 0.9896771  11.92459708]
 [ 0.38734407  9.25595385]
 [ 0.62419774  9.34182876]
 [ 1.03598244  9.13251898]
 [ 1.52120598 12.7770783 ]
 [ 0.16201475  7.62316748]
 [ 1.5294854  12.93170731]
 [ 0.77389001 11.13815733]
 [ 1.43854668 14.43331059]
 [ 0.77970708  9.08175759]
 [ 0.42117791  9.99767758]
 [ 0.38189651  8.82041598]
 [ 0.45969414 11.29653945]]
```

In [16]:

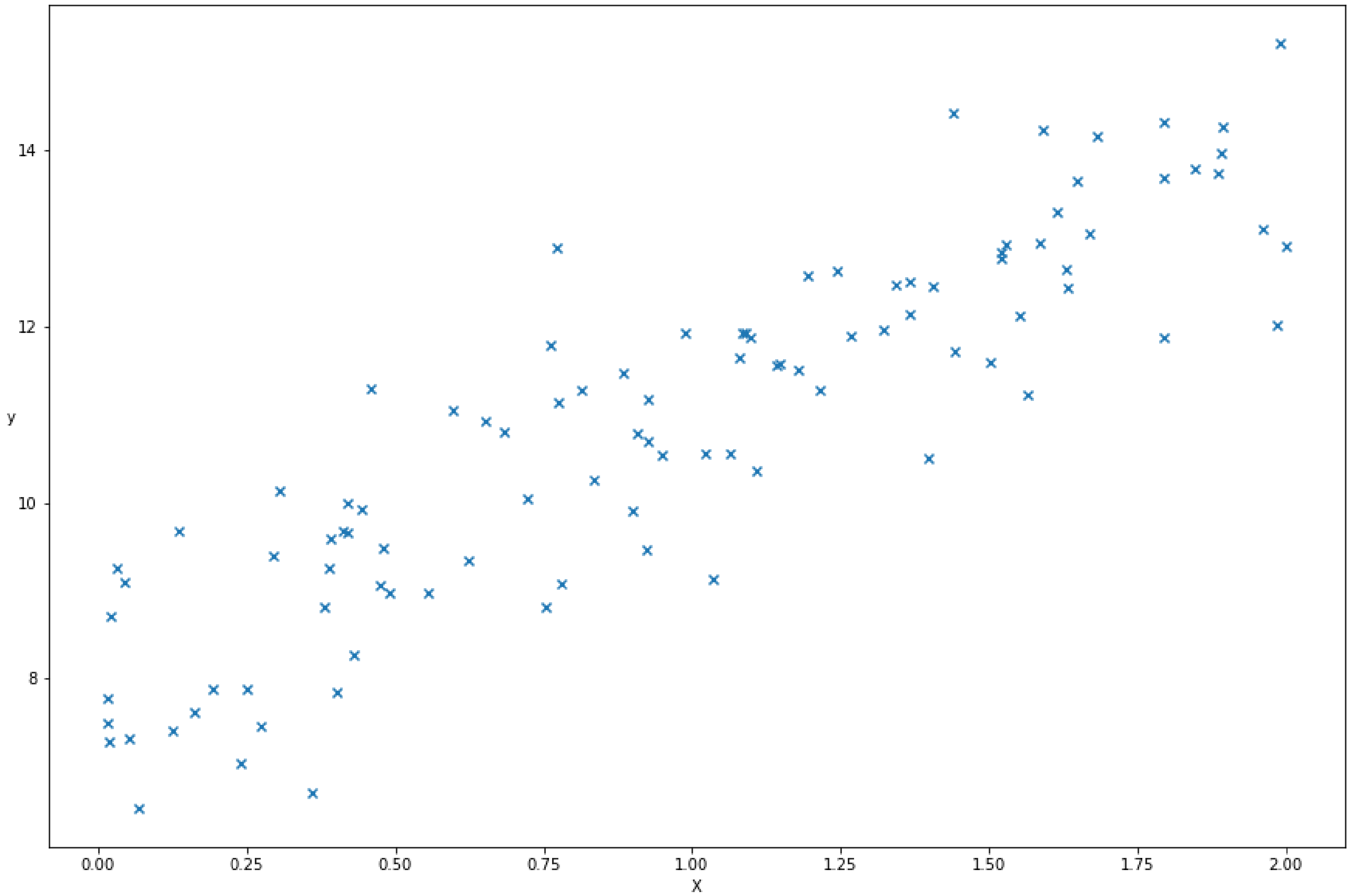
```
# X=2*np.random.rand(100,1)
# print(X)
# print(X.shape)
# y=8+3*X+np.random.rand(100,1)
# print(y.shape)
```

In [17]:

```python
# Scatter plot to see the relation between X and y
plt.figure(figsize=(15,10))
plt.scatter(X, y, marker='x')
plt.xlabel('X')
plt.ylabel('y', rotation=0)
```

Out[17]:

```
Text(0, 0.5, 'y')
```



14
12
10
8
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
X

2. Solve using Numpy Library

$$\nabla_w \hat{L}(f_w) = \nabla_w \frac{1}{n} ||Xw - y||_2^2 = 0$$

In [18]:

```
# # Understanding numpy functions
# import numpy as np
# X = np.arange(6)
# print(X)
# X = X.reshape((6, 1))
# print(X)
# X1=np.ones_like(X)             # Return an array of ones with the same shape and type as a given array.
# print(X1)
# z=np.c_[X1,X]                  # Translates slice objects to concatenation along the second axis.
# print(z)
```

In [19]:

```
# Adding bais unit to every vector in X
B=np.ones_like(X)
X_b=np.c_[B,X]
print(X_b)
print(X_b.shape)

#y=y.reshape((6,2))
theta=np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
print(theta)
```

```
[[1.         0.04497682]
 [1.         0.44369684]
 [1.         1.14757428]
 [1.         1.98332104]
 [1.         0.03038902]
 [1.         1.06406009]
 [1.         0.76146702]
 [1.         0.92371199]
 [1.         1.14317242]
 [1.         1.55185131]
 [1.         0.0186962 ]
 [1.         1.1789661 ]
 [1.         0.12408403]
 [1.         1.19483023]
 [1.         1.59064174]
 [1.         1.40627822]
 [1.         0.41236386]
 [1.         0.01554289]
 [1.         1.24460193]
 [1.         0.59811763]
 [1.         0.41969231]
 [1.         1.36654913]
 [1.         1.64921642]
 [1.         0.35949318]
 [1.         0.02194567]
 [1.         0.19417357]
 [1.         0.43027749]
 [1.         0.01570807]
 [1.         1.8942345 ]
 [1.         1.84577681]
 [1.         1.0891461 ]
 [1.         0.94948277]
 [1.         1.8847801 ]
 [1.         1.26666735]
 [1.         0.40107006]
 [1.         1.44085866]
 [1.         1.61521082]
 [1.         1.50202438]
 [1.         0.88315884]
 [1.         0.47911189]
 [1.         0.30588616]
```

```
 [1.         1.52129161]
 [1.         0.25075942]
 [1.         0.90823295]
 [1.         1.63317685]
 [1.         1.68238394]
 [1.         1.36562465]
 [1.         0.77130311]
 [1.         1.10717202]
 [1.         0.23961462]
 [1.         0.55449143]
 [1.         1.96119551]
 [1.         1.02226458]
 [1.         0.13509331]
 [1.         0.27385538]
 [1.         0.68252181]
 [1.         0.47555176]
 [1.         1.09838153]
 [1.         1.08531632]
 [1.         0.75247426]
 [1.         1.99022675]
 [1.         1.99980551]
 [1.         1.5864137 ]
 [1.         1.66776477]
 [1.         0.39009362]
 [1.         0.06842432]
 [1.         0.49160231]
 [1.         1.89122417]
 [1.         0.29577358]
 [1.         1.21440789]
 [1.         1.79309467]
 [1.         1.63039219]
 [1.         0.81381444]
 [1.         1.0785014 ]
 [1.         0.65317283]
 [1.         1.39720473]
 [1.         0.72273399]
 [1.         1.56516876]
 [1.         1.79435949]
 [1.         0.92515284]
 [1.         0.92666572]
 [1.         1.79482058]
 [1.         0.83466322]
```

```
 [1.         0.89994989]
 [1.         0.05247682]
 [1.         1.34408815]
 [1.         1.32256478]
 [1.         0.9896771 ]
 [1.         0.38734407]
 [1.         0.62419774]
 [1.         1.03598244]
 [1.         1.52120598]
 [1.         0.16201475]
 [1.         1.5294854 ]
 [1.         0.77389001]
 [1.         1.43854668]
 [1.         0.77970708]
 [1.         0.42117791]
 [1.         0.38189651]
 [1.         0.45969414]]
(100, 2)
[[7.87752609]
 [3.11459601]]
```

In [20]:

```
# Adding bais unit to every vector in X
B= np.ones_like(X)                # Return an array of ones with the same shape and type as a given array.
X_b = np.c_[B, X]                 # Translates slice objects to concatenation along the second axis.
# X_b = X
print(X_b)
theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)   # Compute the (multiplicative) inverse of a matrix.

# Values are very close to are thetas 8 and 3
print(theta)

```

```
[[1.         0.04497682]
 [1.         0.44369684]
 [1.         1.14757428]
 [1.         1.98332104]
 [1.         0.03038902]
 [1.         1.06406009]
 [1.         0.76146702]
 [1.         0.92371199]
 [1.         1.14317242]
 [1.         1.55185131]
 [1.         0.0186962 ]
 [1.         1.1789661 ]
 [1.         0.12408403]
 [1.         1.19483023]
 [1.         1.59064174]
 [1.         1.40627822]
 [1.         0.41236386]
 [1.         0.01554289]
 [1.         1.24460193]
 [1.         0.59811763]
 [1.         0.41969231]
 [1.         1.36654913]
 [1.         1.64921642]
 [1.         0.35949318]
 [1.         0.02194567]
 [1.         0.19417357]
 [1.         0.43027749]
 [1.         0.01570807]
 [1.         1.8942345 ]
 [1.         1.84577681]
 [1.         1.0891461 ]
 [1.         0.94948277]
 [1.         1.8847801 ]
 [1.         1.26666735]
 [1.         0.40107006]
 [1.         1.44085866]
 [1.         1.61521082]
 [1.         1.50202438]
 [1.         0.88315884]
 [1.         0.47911189]
 [1.         0.30588616]
```

```
 [1.         1.52129161]
 [1.         0.25075942]
 [1.         0.90823295]
 [1.         1.63317685]
 [1.         1.68238394]
 [1.         1.36562465]
 [1.         0.77130311]
 [1.         1.10717202]
 [1.         0.23961462]
 [1.         0.55449143]
 [1.         1.96119551]
 [1.         1.02226458]
 [1.         0.13509331]
 [1.         0.27385538]
 [1.         0.68252181]
 [1.         0.47555176]
 [1.         1.09838153]
 [1.         1.08531632]
 [1.         0.75247426]
 [1.         1.99022675]
 [1.         1.99980551]
 [1.         1.5864137 ]
 [1.         1.66776477]
 [1.         0.39009362]
 [1.         0.06842432]
 [1.         0.49160231]
 [1.         1.89122417]
 [1.         0.29577358]
 [1.         1.21440789]
 [1.         1.79309467]
 [1.         1.63039219]
 [1.         0.81381444]
 [1.         1.0785014 ]
 [1.         0.65317283]
 [1.         1.39720473]
 [1.         0.72273399]
 [1.         1.56516876]
 [1.         1.79435949]
 [1.         0.92515284]
 [1.         0.92666572]
 [1.         1.79482058]
 [1.         0.83466322]
```

```
 [1.         0.89994989]
 [1.         0.05247682]
 [1.         1.34408815]
 [1.         1.32256478]
 [1.         0.9896771 ]
 [1.         0.38734407]
 [1.         0.62419774]
 [1.         1.03598244]
 [1.         1.52120598]
 [1.         0.16201475]
 [1.         1.5294854 ]
 [1.         0.77389001]
 [1.         1.43854668]
 [1.         0.77970708]
 [1.         0.42117791]
 [1.         0.38189651]
 [1.         0.45969414]]
[[7.87752609]
 [3.11459601]]
```

3. Batch Gradient Decent Algorithm

Below mentioned **Cost Function** which stands for **Mean Squared Error(MSE)** is for **Linear Regression** and **Gradient** is derived from the it.

Cost

$$J(\theta) = 1/m \sum_{i=1}^{m} (h(\theta)^{(i)} - y^{(i)})^2$$

Gradient

$$\frac{\partial J(\theta)}{\partial \theta_j} = 2/m \sum_{i=1}^{m} (h(\theta^{(i)}) - y^{(i)}).\, X_j^{(i)}$$

Below mentioned **Cost Function** which stands for **Mean Squared Error(MSE)** is for **Linear Regression** and **Gradient** is derived from the it.

Cost

$$J(\theta) = 1/2m \sum_{i=1}^{m} (h(\theta)^{(i)} - y^{(i)})^2$$

Gradient

$$\frac{\partial J(\theta)}{\partial \theta_j} = 1/m \sum_{i=1}^{m} (h(\theta^{(i)}) - y^{(i)}).\, X_j^{(i)}$$

Function for computing Cost function

In [21]:

```
def cal_cost(theta, X, y):
    '''
    Calculates the cost for given X and y.


    Parameters
    ----------
    theta: Vector of thetas
    X: Row of X's
    y: Actual y's

    Returns
    -------
    Calculated cost
    '''
    m = len(y)
    predictions = np.dot(X, theta)
    cost = (1/ m) * np.sum(np.square(predictions - y))

    return cost
```

Function for Batch gradient Decent Algorithm

In [22]:

```python
def gradient_descent(X, y, theta, learning_rate, iterations):
    '''
    Parameters
    ----------
    X: Matrix of X with bais units added to every vector of X
    y: Vector of y
    theta: Vector initialized randomly
    learning_rate: learning rate that will be used as step size
    iterations = Number of iterations

    Returns
    -------
    The final theta vector, array of cost and theta history over no of iterations
    '''

    m = len(y)
    cost_history = np.zeros(iterations)
    theta_history = np.zeros((iterations, 2))

    for i in range(iterations):
      #Forward propagation
        predictions = np.dot(X, theta)
      #Compute Cost (Value of Loss function)
        cost_history[i] = cal_cost(theta, X, y)
      # Backpropogation
        grad =(2/m) * (X.T.dot((predictions - y)))
      # Update Parameters
        theta = theta - (learning_rate *  grad)
        theta_history[i, :] = theta.T


    return theta, cost_history, theta_history
```

Calling function

In [23]:

```
# Using gradient descent to find out the thetas between our X and y relation
import time
lr = 0.3
n_iter = 100

theta = np.random.randn(2,1)
print("Initial value of weights")
print(theta)
X_b = np.c_[np.ones((len(X),1)),X]

## Calling BGD function
t10 = time.time()
theta,cost_history,theta_history = gradient_descent(X_b, y, theta, lr, n_iter)
t11 = time.time()
time_BGD = t11 - t10
print("Processing Time of MBG Algorithm")
print(time_BGD)


# Predicted value by NNs
y_predicted= np.dot(X_b, theta)
print("Optimal value of weights")
print('{:<10}{:.3f}'.format('Theta0:',theta[0][0]))
print('{:<10}{:.3f}'.format('Theta1:',theta[1][0]))
print("Minimum  value of cost function")
print('{:<10}{:.3f}'.format('Cost/MSE:',cost_history[-1]))

# print(y_predicted)

predicted__actual = np.c_[y_predicted, y]
print("Comparing predicted vs actual")
print(predicted__actual)
```

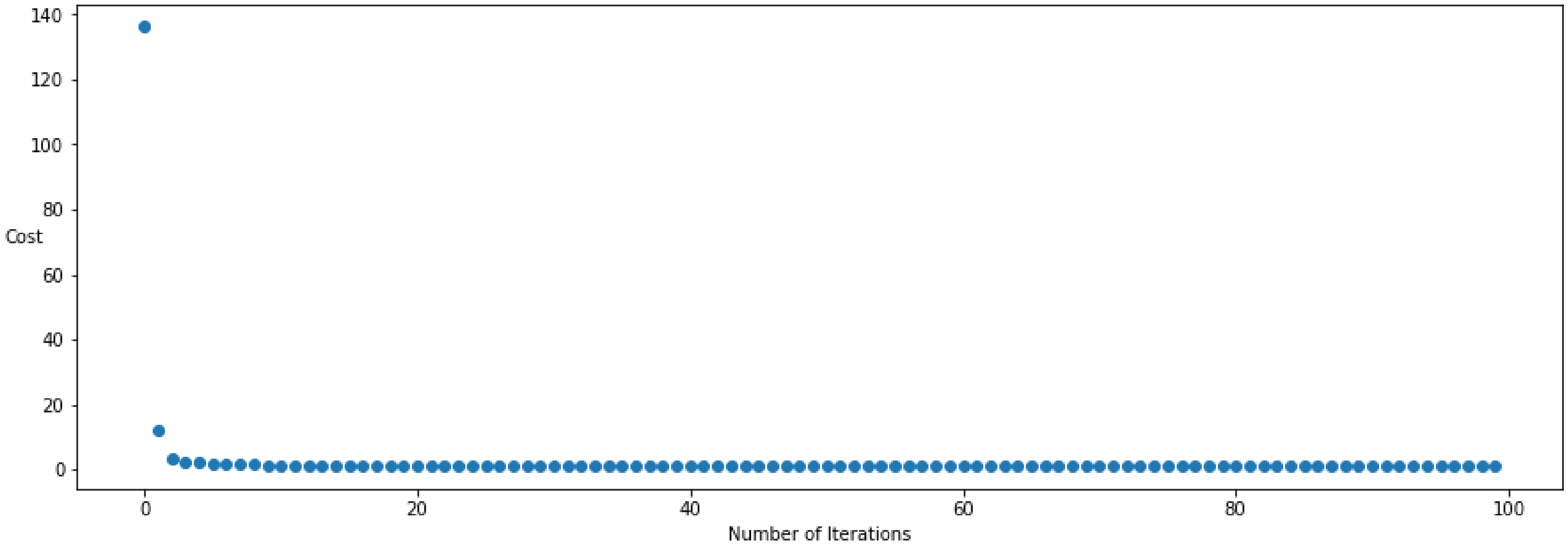
```
[11.45175653 11.57207686]
[14.0548556  12.02308413]
[ 7.97206146  9.2510614 ]
[11.19163502 10.54856201]
[10.24914889 11.78215315]
[10.75449303  9.46087616]
[11.43804608 11.56766946]
[12.71095756 12.12205791]
[ 7.93564186  7.2777673 ]
[11.5495326  11.50784111]
[ 8.26389315  7.41553359]
[11.59894458 12.57702559]
[12.83177803 14.23886903]
[12.25754129 12.4646239 ]
[ 9.16179784  9.66891799]
[ 7.92582023  7.50275661]
[11.75396842 12.63655415]
[ 9.74036482 11.0568907 ]
[ 9.18462378  9.65690536]
```

In [24]:

```python
# plotting the cost history vs number of iterations

plt.figure(figsize=(15,5))
plt.scatter(range(n_iter), cost_history)
plt.xlabel('Number of Iterations')
plt.ylabel('Cost', rotation=0)

```

Out[24]:

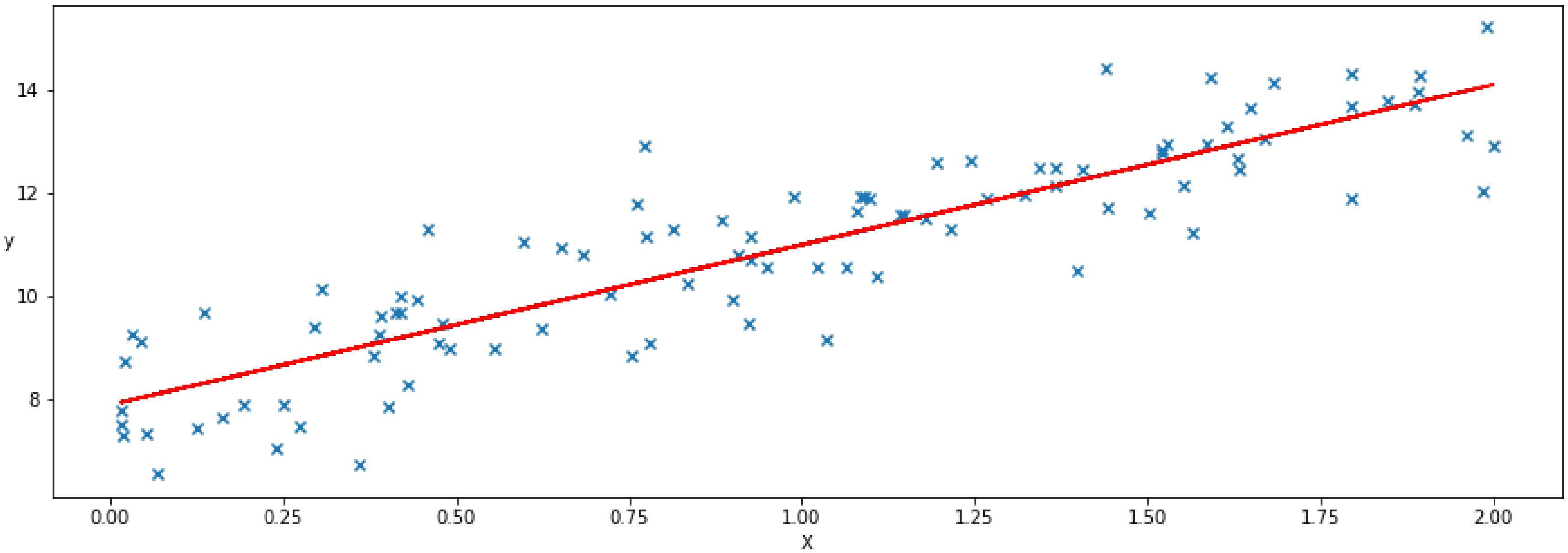
```
Text(0, 0.5, 'Cost')
```



In [25]:

```python
# plot prediction line
plt.figure(figsize=(15,5))
plt.scatter(X, y, marker='x')
plt.plot(X, y_predicted, 'red')
plt.xlabel('X')
plt.ylabel('y', rotation=0)
```

Out[25]: Text(0, 0.5, 'y')



4. Stochastic Gradient Descent (SGD) Algorithm

In [26]:

```
def stochastic_gradient_descent(X, y, theta, learning_rate,iterations):
    '''
    Parameters
    ----------
    X: Matrix of X with bais units added to every vector of X
    y: Vector of y
    theta: Vector initialized randomly
    learning_rate: learning rate that will be used as step size
    iterations = Number of iterations

    Returns
    -------
    The final theta vector, array of cost and theta history over no of iterations
    '''
    m = len(y)
    theta_history = np.zeros((iterations, 2))
    cost_history = np.zeros(iterations)
#     theta_history = []

    for i in range(iterations):
        cost_per_iteration = .0
        for j in range(m):
             # #Step : Shuffle (X,y)
            X_rand_idx = np.random.randint(m)
            X_inner = X[X_rand_idx].reshape(1, X.shape[1])
            y_inner = y[X_rand_idx].reshape(1, 1)

            # Forward Propgation
            predictions = np.dot(X_inner, theta)

            # Computing cost value
            cost_per_iteration += cal_cost(theta, X_inner, y_inner)

            # Backpropagation
            grad= (2/m) * (X_inner.T.dot((predictions - y_inner)))

            # Weight updation
            theta = theta - (learning_rate * grad)

        cost_history[i] = cost_per_iteration

```

```python

    return theta, cost_history, theta_history
```

In [31]:

```
# Using stochastic gradient descent to find out the thetas between our X and y relation
import time
lr = 0.3
n_iter = 100
theta = np.random.randn(2,1)
print("Initial value of weights")
print(theta)

X_b = np.c_[np.ones((len(X),1)),X]

## calling SGD
t20 = time.time()
theta,cost_history,theta_history = stochastic_gradient_descent(X_b, y, theta, lr, n_iter)
t21 = time.time()
time_SGD = t21 - t20
print("Processing Time of SGD Algorithm")
print(time_SGD)
# theta,cost_history,theta_history = stochastic_gradient_descent(X_b, y, theta, lr, n_iter)
print(theta)


# Predicted value by NNs
y_predicted= np.dot(X_b, theta)

print("Optimal value of weights")
print('{:<10}{:.3f}'.format('Theta0:',theta[0][0]))
print('{:<10}{:.3f}'.format('Theta1:',theta[1][0]))
print("Minimum  value of cost function")
print('{:<10}{:.3f}'.format('Cost/MSE:',cost_history[-1]))


# print(y_predicted)
predicted__actual = np.c_[y_predicted, y]
print("Comparing predicted vs actual")
print(predicted__actual)
```

```
[ 9.26269015  8.26604118]
[ 7.9709022   7.77699381]
[13.82434371 14.27457547]
[13.67335075 13.78644959]
[11.31570832 11.9357483 ]
[10.88052087 10.53574713]
[13.79488402 13.73178354]
[11.86886012 11.90012728]
[ 9.17168056  7.84937596]
[12.41163588 11.71417226]
[12.95491287 13.30733268]
[12.60222674 11.5954719 ]
[10.67385716 11.46431646]
[ 9.41485695  9.47640518]
[ 8.87508987 10.13541128]
[12.66226294 12.84197568]
[ 8.70331633  7.88154753]
[10.75198745 10.79025615]
[13.01089457 12.44083037]
```

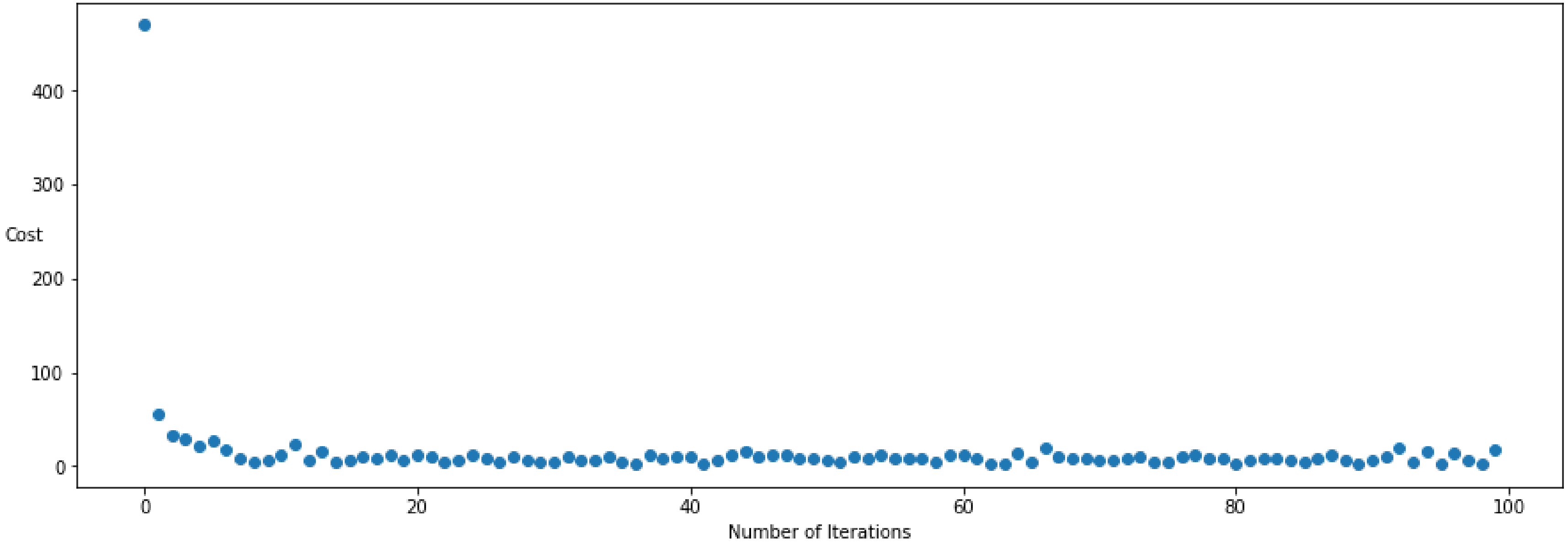
In [32]:

```python
# def stochastic_gradient_descent(X, y, theta, learning_rate,iterations):
#     '''
#     Parameters
#     ----------
#     X: Matrix of X with bais units added to every vector of X
#     y: Vector of y
#     theta: Vector initialized randomly
#     learning_rate: Learning rate that will be used as step size
#     iterations = Number of iterations

#     Returns
#     -------
#     The final theta vector, array of cost and theta history over no of iterations
#     '''

#     m = len(y)
#     theta_history = np.zeros((iterations, 2))
#     cost_history = np.zeros(iterations)
#     theta_history = []

#     for i in range(iterations):
#         cost_per_iteration = .0
#         for j in range(m):
#              # #Step : Shuffle (X,y)
#             X_rand_idx = np.random.randint(m)
#             X_inner = X[X_rand_idx].reshape(1, X.shape[1])
#             y_inner = y[X_rand_idx].reshape(1, 1)

#             predictions = np.dot(X_inner, theta)

#             theta = theta - (2/m) * learning_rate * (X_inner.T.dot((predictions - y_inner)))
#             cost_per_iteration += cal_cost(theta, X_inner, y_inner)
#         cost_history[i] = cost_per_iteration

#     return theta, cost_history, theta_history
```

In [20]:

```
# plotting the cost history vs number of iterations

plt.figure(figsize=(15,5))
plt.scatter(range(n_iter), cost_history)
plt.xlabel('Number of Iterations')
plt.ylabel('Cost', rotation=0)
```

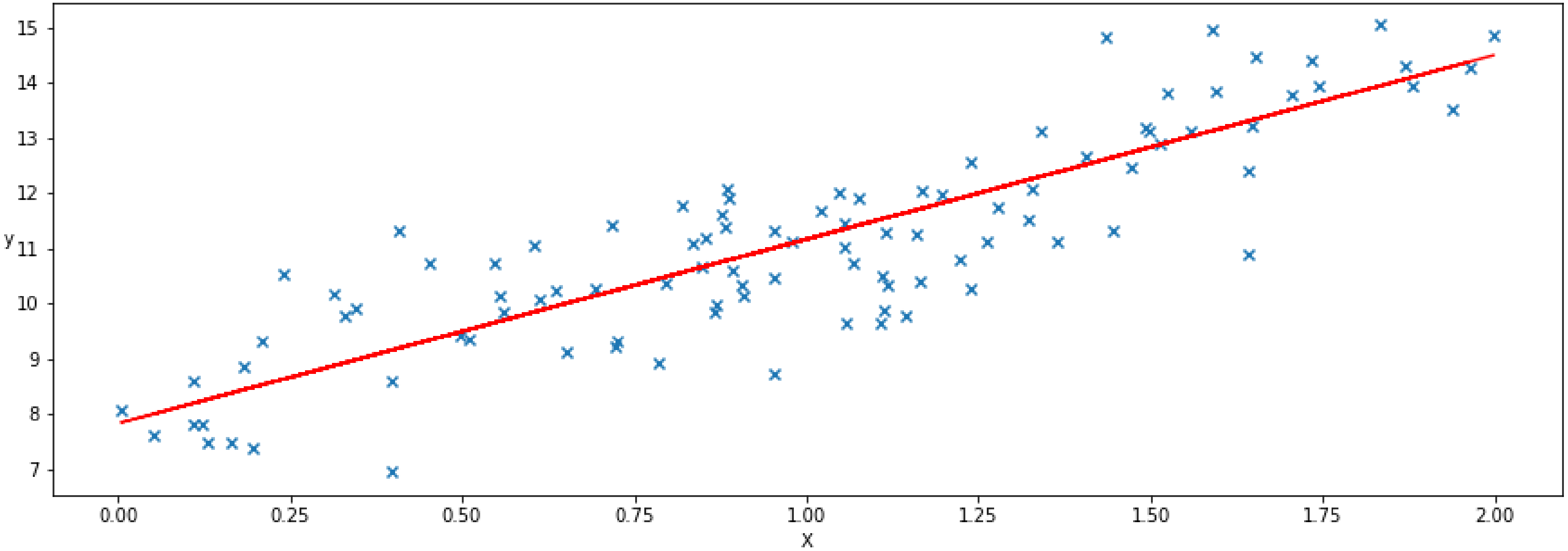
Out[20]: Text(0, 0.5, 'Cost')



In [20]:

```python
# plot prediction line
plt.figure(figsize=(15,5))
plt.scatter(X, y, marker='x')
plt.plot(X, np.dot(X_b, theta), 'red')
plt.xlabel('X')
plt.ylabel('y', rotation=0)
```

Out[20]: Text(0, 0.5, 'y')



5. Mini-batch Gradient Descent (MBGD) Algorithm

In [27]:

```python
def mini_batch_gradient_descent(X, y, theta, learning_rate, iterations):
    '''
    Parameters
    ----------
    X: Matrix of X with bais units added to every vector of X
    y: Vector of y
    theta: Vector initialized randomly
    learning_rate: learning rate that will be used as step size
    iterations = Number of iterations

    Returns
    -------
    The final theta vector, array of cost and theta history over no of iterations
    '''

    m = len(y)
    cost_history = np.zeros(iterations)
    # print(cost_history)
    theta_history = np.zeros((iterations, 2))
    # print(theta_history)
    # Batch size
    batch_size=10
    for i in range(iterations):
        cost_per_iteration = .0
        # #Step 1: Shuffle (X,y)
        rand_indicies = np.random.permutation(m)
        X = X[rand_indicies]
        y = y[rand_indicies]

        for j in range(0, m, batch_size):
            X_inner = X[i: i+batch_size]
            y_inner = y[i: i+batch_size]
            # forward propagation
            predictions = np.dot(X_inner, theta)
            # # weight updataion equation
            # theta = theta - (2/m) * learning_rate * (X_inner.T.dot((predictions - y_inner)))
            # cost_per_iteration += cal_cost(theta, X_inner, y_inner)
            # Computing cost value
            cost_per_iteration += cal_cost(theta, X_inner, y_inner)
            # Backpropagation
            grad= (2/m) * (X_inner.T.dot((predictions - y_inner)))
```

```
            # Weight updation
            theta = theta - (learning_rate * grad)
        cost_history[i] = cost_per_iteration

    return theta, cost_history, theta_history
```

In [28]:

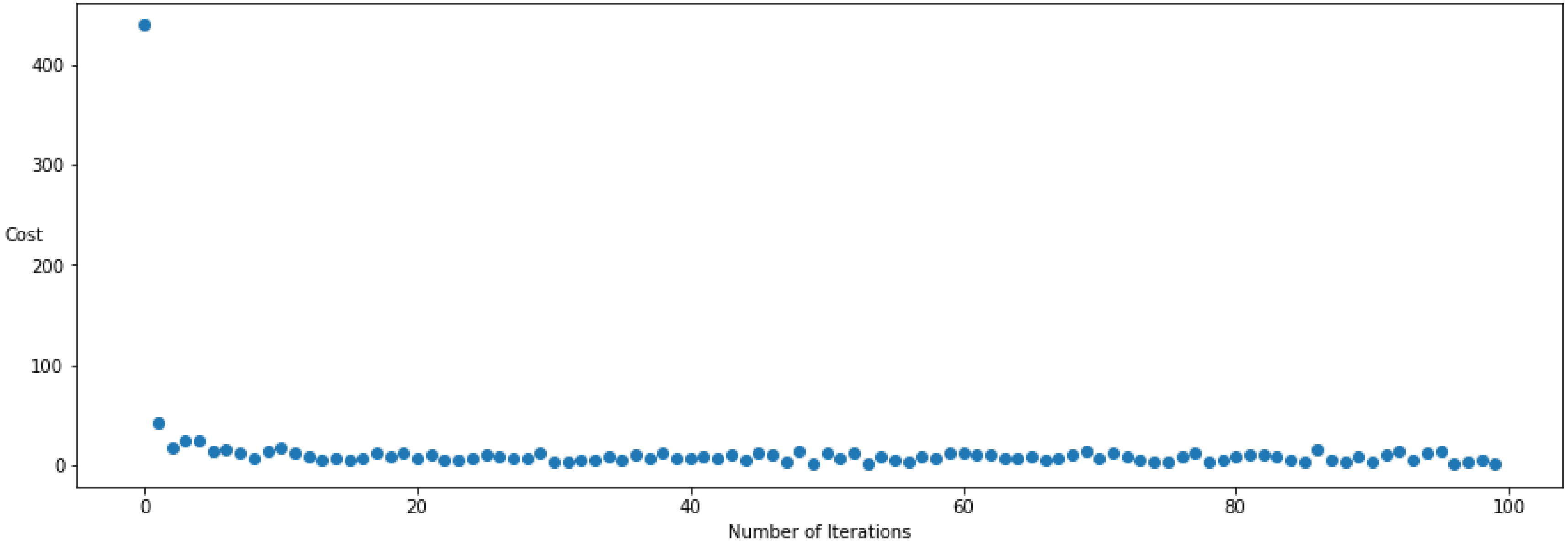
```python
# Using mini batch gradient descent to find out the thetas between our X and y relation
lr = 0.3
n_iter = 100
theta = np.random.randn(2,1)
print("Initial value of weights")
print(theta)

# print(X_b)
X_b = np.c_[np.ones((len(X),1)),X]
# print(X_b)

## calling MBGD
t30 = time.time()
theta,cost_history,theta_history = mini_batch_gradient_descent(X_b, y, theta, lr, n_iter)
t31 = time.time()
time_MBGD = t31 - t30
print("Processing Time of MBGD Algorithm")
print(time_MBGD)
# theta,cost_history,theta_history = mini_gradient_descent(X_b, y, theta, lr, n_iter)
# print(theta)

# Predicted value by NNs
y_predicted= np.dot(X_b, theta)

print("Optimal value of weights")
print('{:<10}{:.3f}'.format('Theta0:',theta[0][0]))
print('{:<10}{:.3f}'.format('Theta1:',theta[1][0]))
print("Minimum  value of cost function")
print('{:<10}{:.3f}'.format('Cost/MSE:',cost_history[-1]))

# print(y_predicted)
predicted__actual = np.c_[y_predicted, y]
print("Comparing predicted vs actual")
print(predicted__actual)
```

```
[12.12343891 12.13159462]
[13.04414139 13.65952935]
[ 8.84326137  6.70617922]
[ 7.74380331  8.71344024]
[ 8.30478315  7.8774343 ]
[ 9.07381964  8.26604118]
[ 7.72348624  7.77699381]
[13.84221304 14.27457547]
[13.68437689 13.78644959]
[11.21988316 11.9357483 ]
[10.76497248 10.53574713]
[13.8114182  13.73178354]
[11.79810449 11.90012728]
[ 8.97868538  7.84937596]
[12.36547953 11.71417226]
[12.93337851 13.30733268]
[12.5647082  11.5954719 ]
[10.5489425  11.46431646]
[ 9.23288283  9.47640518]
[ 8.66865284 10.13541128]
```

In [29]:

```
# plotting the cost history vs number of iterations

plt.figure(figsize=(15,5))
plt.scatter(range(n_iter), cost_history)
plt.xlabel('Number of Iterations')
plt.ylabel('Cost', rotation=0)
```

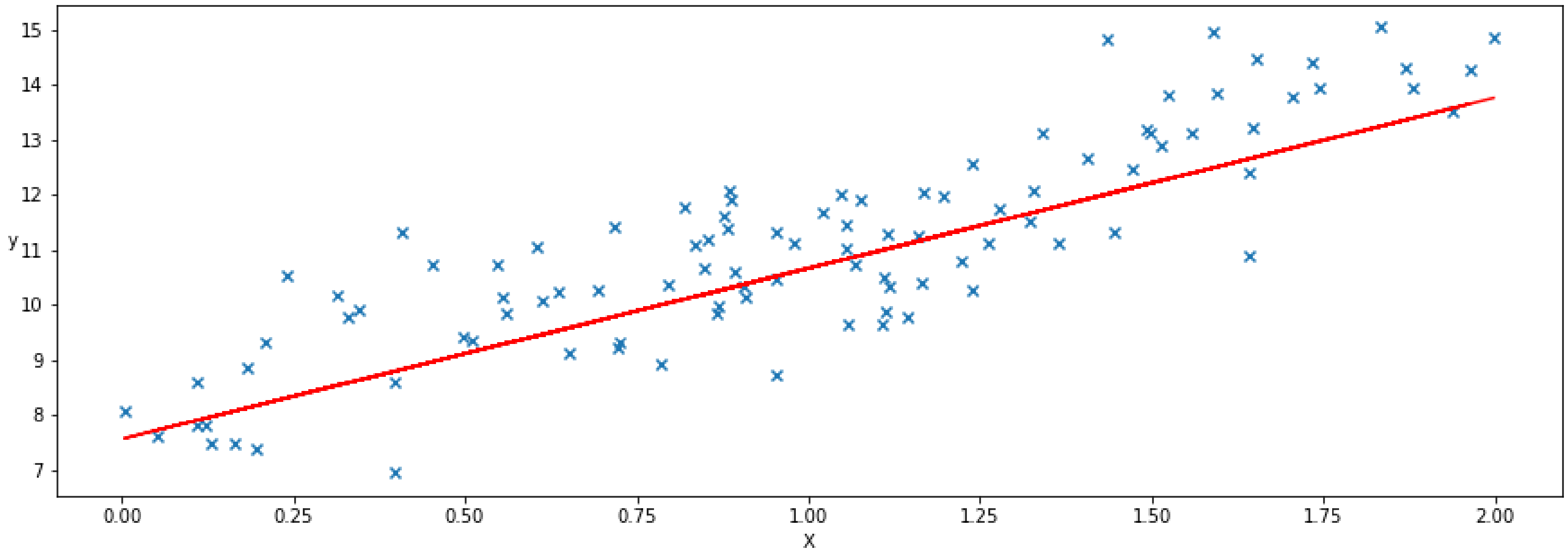
Out[29]: Text(0, 0.5, 'Cost')



In [30]:

```python
# plot prediction line

plt.figure(figsize=(15,5))
plt.scatter(X, y, marker='x')
plt.plot(X, np.dot(X_b, theta), 'red')
plt.xlabel('X')
plt.ylabel('y', rotation=0)
```

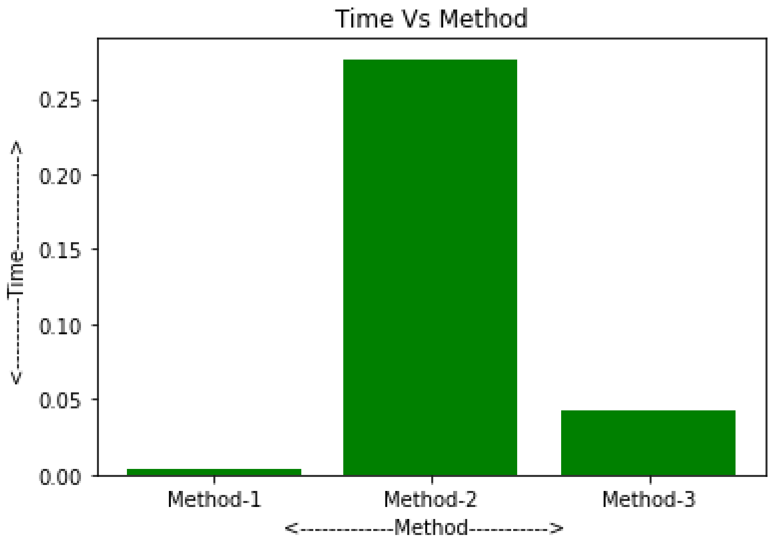
Out[30]: Text(0, 0.5, 'y')



Comparison of Processing Speed of above three Grandient Decent Optimization Algorithms Implementation

In [33]:

```
import matplotlib.pyplot as plt
%matplotlib inline
# plt.style.use('ggplot')

x = ['Method-1', 'Method-2', 'Method-3']
Processing_Times = [time_BGD, time_SGD, time_MBGD]


x_pos = [i for i, _ in enumerate(x)]

plt.bar(x_pos, Processing_Times, color='green')
plt.xlabel("<-------------Method----------->  ")
plt.ylabel("<----------Time-------------> ")
plt.title(" Time Vs Method ")

plt.xticks(x_pos, x)

plt.show()
```



In [ ]:

```
 
```