Gradient Descent — Intro and Implementation in python

Introduction

Gradient Descent is an optimization algorithm in machine learning used to minimize a function by iteratively moving towards the minimum value of the function.

We basically use this algorithm when we have to find the least possible values that can satisfy a given cost function.

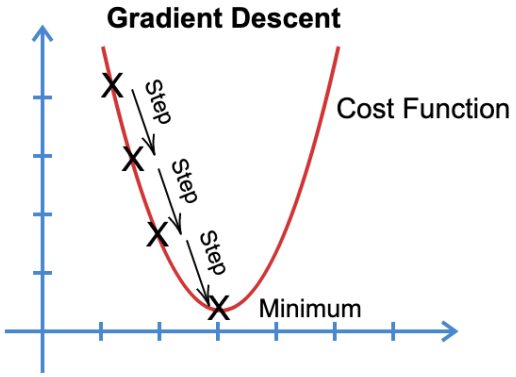
In machine learning, more often that not we try to minimize **loss functions** (like **Mean Squared Error**). By minimizing the loss function , we can improve our model, and Gradient Descent is one of the most popular algorithms used for this purpose.

WHAT IS GRADIENT DESCENT?

Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible.
You start by defining the initial parameter's values and from there gradient descent uses calculus to iteratively adjust the values so they minimize the given cost-function.

WHAT IS COST FUNCTION?

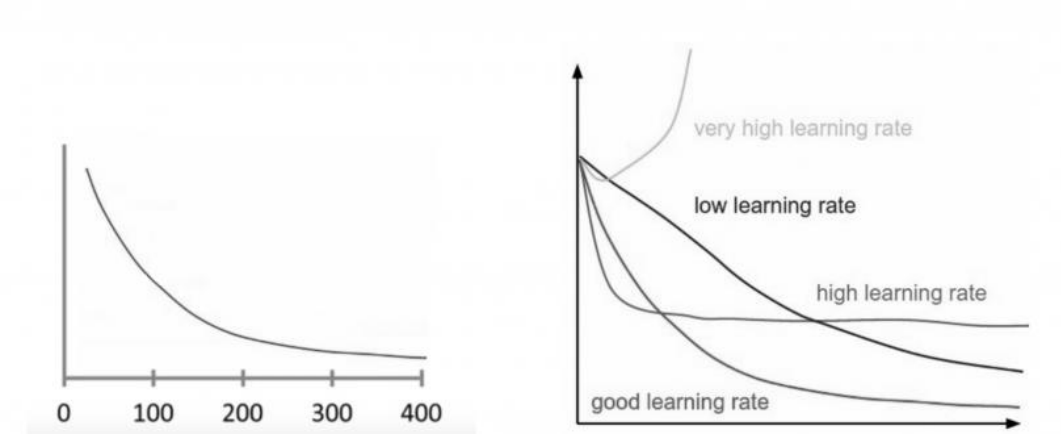
*It is a **function** that measures the performance of a model for any given data. **Cost Function** quantifies the error between predicted values and expected values and presents it in the form of a single real number.*



We first take a point in the cost function and start moving in steps towards the minimum point. The size of that step, or how quickly we have to converge to the minimum point is defined by **Learning Rate.** We can cover more area with higher learning rate but at the risk of overshooting the minima. On the other hand, small steps/smaller learning rates will consume a lot of time to reach the lowest point.

HOW TO MAKE SURE IT WORKS PROPERLY

A good way to make sure gradient descent runs properly is by plotting the cost function as the optimization runs. Put the number of iterations on the x-axis and the value of the cost-function on the y-axis. This helps you see the value of your cost function after each iteration of gradient descent, and provides a way to easily spot how appropriate your learning rate is. You can just try different values for it and plot them all together. The left image below shows such a plot, while the image on the right illustrates the difference between good and bad learning rates.



If gradient descent is working properly, the cost function should decrease after every iteration. When gradient descent can't decrease the cost-function anymore and remains more or less on the same level, it has converged. The number of iterations gradient descent needs to converge

can sometimes vary a lot. It can take 50 iterations, 60,000 or maybe even 3 million, making the number of iterations to convergence hard to estimate in advance.
There are some algorithms that can automatically tell you if gradient descent has converged, but you must define a threshold for the convergence beforehand, which is also pretty hard to estimate. For this reason, simple plots are the preferred convergence test.
Another advantage of monitoring gradient descent via plots is it allows us to easily spot if it doesn't work properly, for example if the cost function is increasing. Most of the time the reason for an increasing cost-function when using gradient descent is a learning rate that's too high.

Example:

Cost function **$f(x) = x^3-4x^2+6$**

Derivative of f(x) [x_derivative], **$f'(x) = 3x^2-8x$** (This will give the slope of any point x along f(x))

Start with any value of x. Lets say 0.5 and learning_rate = 0.05

Iterate over a number of times and keep calculating value of x.

x = x - (x_derivative * learning_rate)

x = 0.5 - (-3.25*0.05) = 0.6625

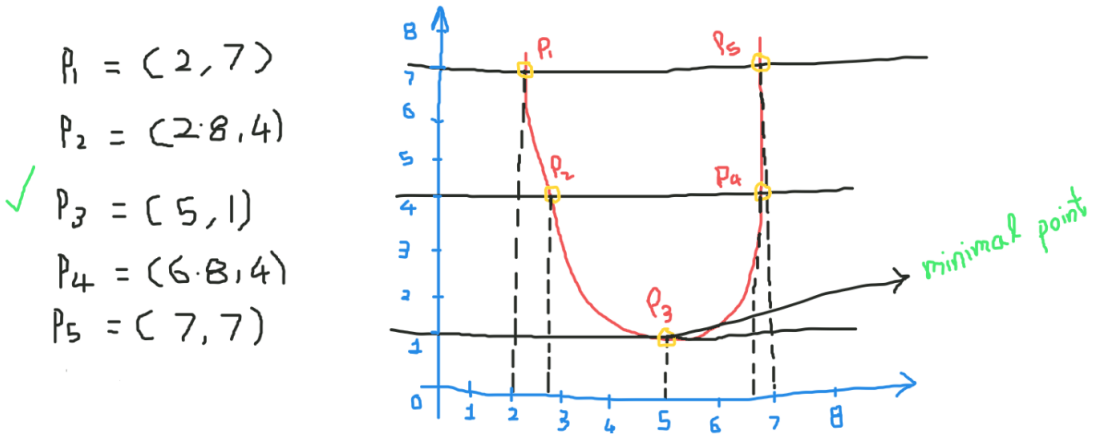
Use x = 0.6625 in second iteration

x = 0.6625+(3.983*0.05) = 0.86165

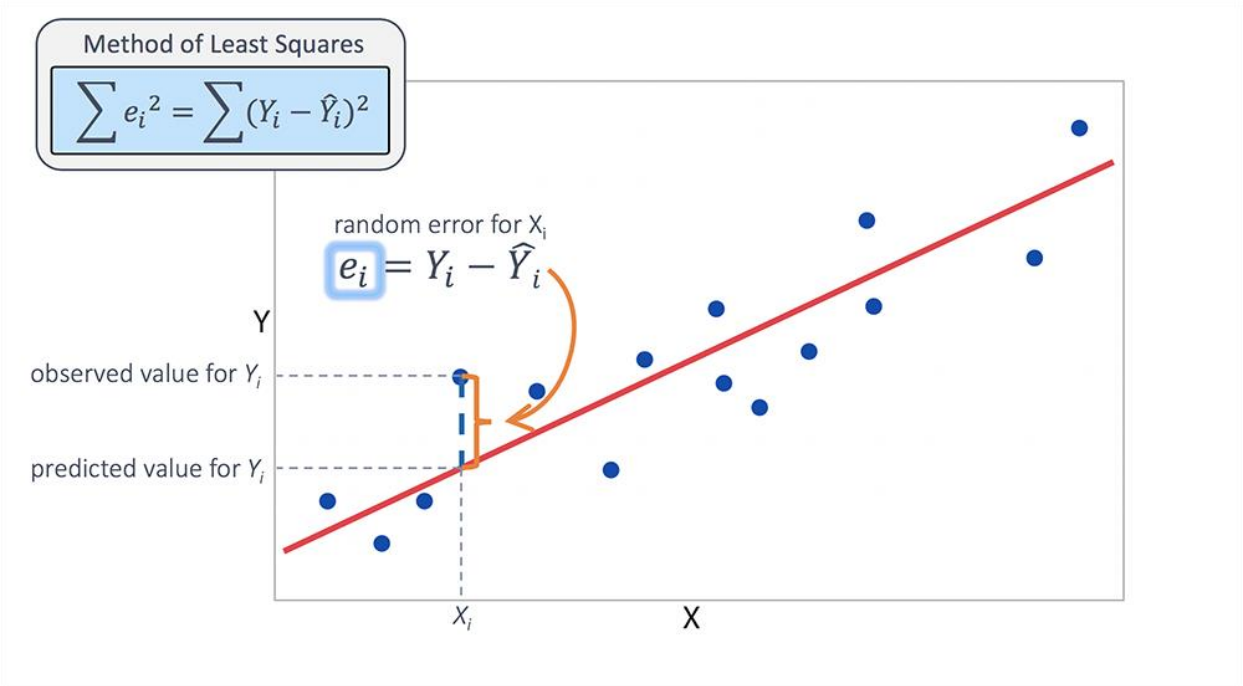
and so on until we stop seeing any change in the value of x.

The number of iterations can be fixed and given by the user. Or, if you have a precision in mind (~0.001). You can stop calculating once you reach this value of precision.

we have five points here at different positions



In an Algorithm, our main Motive is to **minimize our loss** that indicates that our model has performed well. For analyzing this we will use **linear regression.**



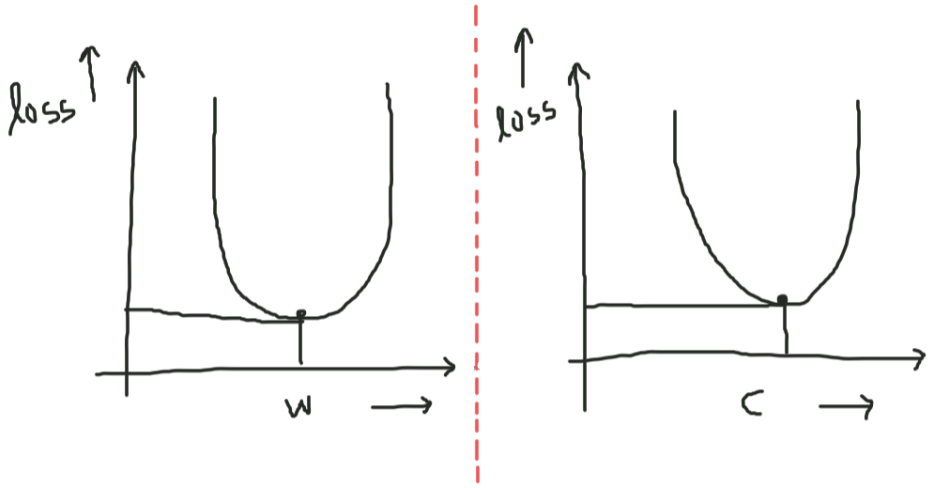
Since linear regression uses a line to predict the continuous output-

let the line be y = w * x + c

Here we need to find the w and c such that we get the best fit line that minimizes the error. So our aim is to find the optimal w and c values

We start **w and c with some random values** and we start updating these values w.r.t to the loss i.e we **update these weights till our slope is equal to or close to zero.**

We will take the loss function on the y-axis and w and c on the x-axis. Check the figure below-



For reaching the minimal w value in the first graph follow these steps-

1. Start calculating the loss for the given set of x _values with the w and c.

2. Plot the point and now update the weight as-

w_new =w_old – learning_rate * slope at (w_old,loss)

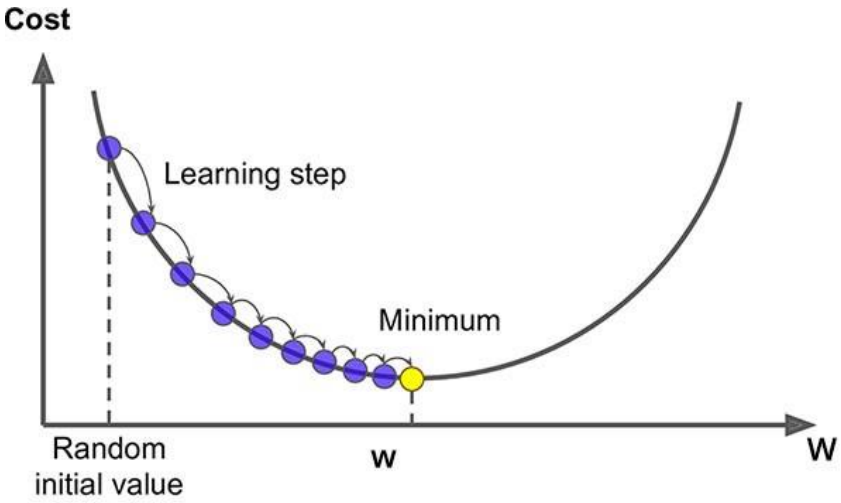
Repeat these steps until you reach the minimal values!

*** we are subtracting slope here because we want to move to the bottom of the hill or in the direction of steepest descent**

*** as we subtract our parameters gets updated and we will get a slope less than the previous one which is what we want to move to a point where the slope is equal to or close to zero**

***we will talk about learning rate later!**

The same applies to graph 2 also i.e loss vs c



Now the question is **why the learning rate is put in the equation?** We cannot travel all the points present in between the starting point and the minimal point-

We need to skip some points-

- We can take elephant steps in the initial phase.
- But while moving near minima, we need to take baby steps, because we may cross over the minima and move to a point where slope increases.So in order to **control the step size** and the **shift in the graph learning_rate was introduced.** Even without the learning rate, we get the minimum value but what we are concerned about is that we want our algorithm to be faster !!!

  Here is a sample algorithm of how linear regression works using Gradient Descent. Here we use mean square error as a loss function-

  1. initializing model parameters with zeros m=0 , c=0
  2. initialize learning rate with any value in the range of (0,1) exclusive

lr = 0.01

You know the mean square error equation i.e-

$$M.S.E = \frac{1}{N} * \Sigma(Yoriginal - Ypred)^2$$

Now substitute **(w *x +c)** in place of **Ypred** and differentiate w.r.t **w**

$$\frac{d(loss)}{dm} = \frac{1}{N} \times \frac{d}{dw}\left((Y_{original} - (w \times x + c)\right)^2$$

$$= \frac{2}{N} \times \left(Y_{original} - (w \times x + c)\right) \times (-x)$$

$$= -\frac{2}{N} \times x \times (Y_{original} - Y_{pred})$$

1. Here m in the L.H.S is noting but slope (w). Same with respect to c also-

$$\frac{d(loss)}{dc} = \frac{1}{N} \times \frac{d}{dc}\left((Y_{original} - Y_{pred})\right)^2$$

$$= \frac{1}{N} \times \frac{d}{dc}\left(Y_{original} - (w \times x + c)\right)^2$$

$$= -\frac{2}{N} \times 1 \times (Y_{original} - Y_{pred})$$

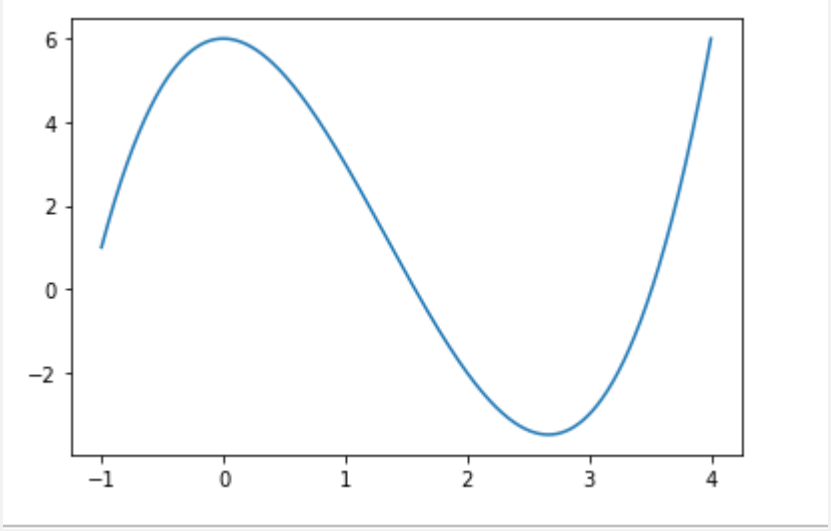
Python Implementation

We will implement a simple form of Gradient Descent using python. Let's take the polynomial function in the above section and treat it as Cost function and attempt to find a local minimum value for that function.

Cost function **$f(x) = x^3 - 4x^2 + 6$**

Let's import required libraries first and create f(x). Also generate 1000 values from -1 to 4 as **x** and plot the curve of **f(x)**.

```
# Importing required libraries
import numpy as np
import matplotlib.pyplot as pltf_x = lambda x: (x**3)-4*(x**2)+6
x = np.linspace(-1,4,1000)#Plot the curve
plt.plot(x, f_x(x))
plt.show()
```



$f(x) = x^3 - 4x^2 + 6$

Let's find out the derivative of f(x).

$d\,f(x)/dx = 3x^2 - 8x$. Let's create a lambda function in python for the derivative.

```
f_x_derivative = lambda x: 3*(x**2)-8*x
```

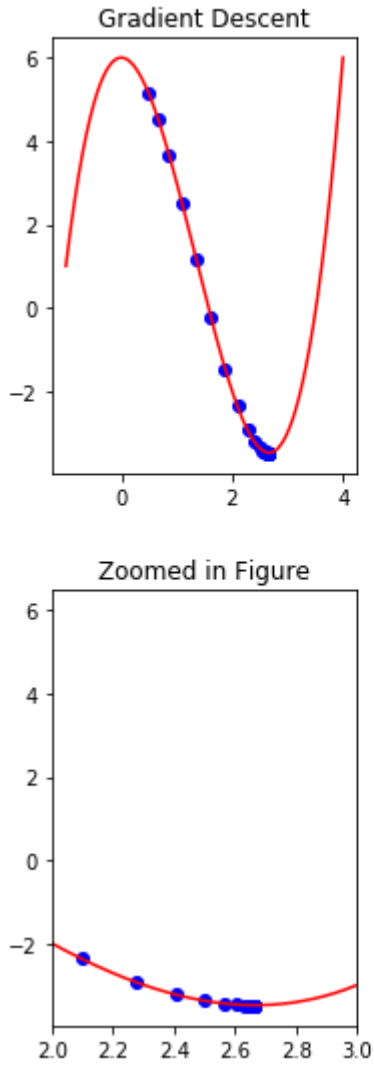
Let’s create a function to plot gradient descent and also a function to calculate gradient descent by passing a fixed number of iterations as one of the inputs.

```
def plot_gradient(x, y, x_vis, y_vis):
   plt.subplot(1,2,2)
   plt.scatter(x_vis, y_vis, c = "b")
   plt.plot(x, f_x(x), c = "r")
   plt.title("Gradient Descent")
   plt.show()plt.subplot(1,2,1)
   plt.scatter(x_vis, y_vis, c = "b")
   plt.plot(x,f_x(x), c = "r")
   plt.xlim([2.0,3.0])
   plt.title("Zoomed in Figure")
   plt.show()

def gradient_iterations(x_start, iterations, learning_rate):

   # These x and y value lists will be used later for visualization.
   x_grad = [x_start]
   y_grad = [f_x(x_start)]
   # Keep looping until number of iterations
   for i in range(iterations):

      # Get the Slope value from the derivative function for x_start
      # Since we need negative descent (towards minimum), we use '-' of derivative
      x_start_derivative = - f_x_derivative(x_start)

      # calculate x_start by adding the previous value to
      # the product of the derivative and the learning rate calculated above.
      x_start += (learning_rate * x_start_derivative)

      x_grad.append(x_start)
      y_grad.append(f_x(x_start))print ("Local minimum occurs at:
{:.2f}".format(x_start))
   print ("Number of steps: ",len(x_grad)-1)
   plot_gradient(x, f_x(x) ,x_grad, y_grad)
```

Now that we have defined these functions let's call gradient_iterations functions by passing **x_start = 0.5, iterations = 1000, learning_rate = 0.05**

```
gradient_iteration(0.5, 1000, 0.05)
```

```
Local minimum occurs at: 2.67
Number of steps:  1000
```



gradient_iteration(0.5, 1000, 0.05)

We are able to find the Local minimum at **2.67** and as we have given the number of iterations as 1000, Algorithm has taken 1000 steps. It might have reached the value 2.67 at a much earlier iteration. But since we don't know at what point will our algorithm reach the local minimum

with the given learning rate, we give a high value of iteration just to be sure that we find our local minimum.

This doesn't sound to be very optimal because of the unnecessary number of loop iterations even after it has found the local minimum.

Let's take another approach of fixing the number of iterations by using precision.

In this approach , Since we know the dataset, we can define the level of precision that we want and stop the algorithm once we reach that level of precision.

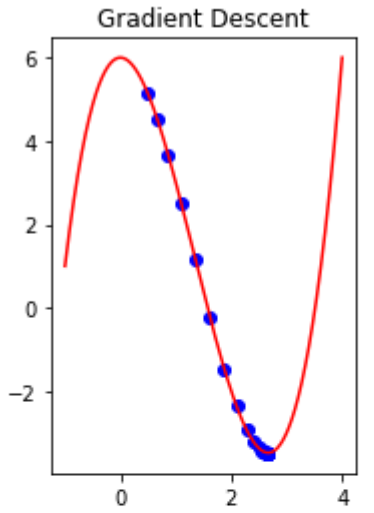
For this example let's write a new function which takes precision instead of iteration number.

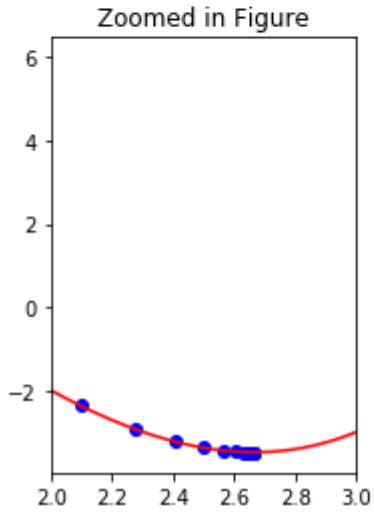
```
def gradient_precision(x_start, precision, learning_rate):

  # These x and y value lists will be used later for visualisation.
  x_grad = [x_start]
  y_grad = [f_x(x_start)]while True:

    # Get the Slope value from the derivative function for x_start
    # Since we need negative descent (towards minimum), we use '-' of derivative
    x_start_derivative = - f_x_derivative(x_start)

    # calculate x_start by adding the previous value to
    # the product of the derivative and the learning rate calculated above.
    x_start += (learning_rate * x_start_derivative)



    x_grad.append(x_start)
    y_grad.append(f_x(x_start))
    # Break out of the loop as soon as we meet precision.
    if abs(x_grad[len(x_grad)-1] - x_grad[len(x_grad)-2]) <= precision:
      breakprint ("Local minimum occurs at: {:.2f}".format(x_start))
  print ("Number of steps taken: ",len(x_grad)-1)
  plot_gradient(x, f_x(x) ,x_grad, y_grad)
```

Now let's call this function with parameters **x_start = 0.5, precision = 0.001, learning rate = 0.05**

gradient_precision(0.5, 0.001, 0.05)

```
Local minimum occurs at: 2.67
Number of steps taken:  20
```





gradient_precision(0.5, 0.001, 0.05)

Local Minimum = 2.67 Number of Steps = 20

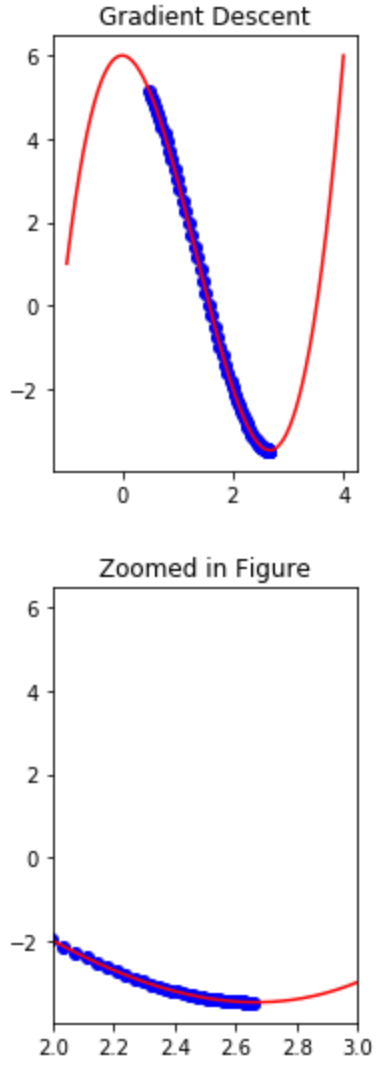
Our gradient Descent algorithm was able to find the local minimum in just 20 steps! So, in the previous method we were unnecessarily running 980 iterations!

Now that we are able to successfully minimize **f(x)** i.e. find the minimum value of x for which **f(x)** is minimum, Let's play around with learning rate values and see how it affects the algorithm output.

Learning rate = 0.01

gradient_precision(0.5, 0.001, 0.01)

```
Local minimum occurs at: 2.66
Number of steps taken:  85
```



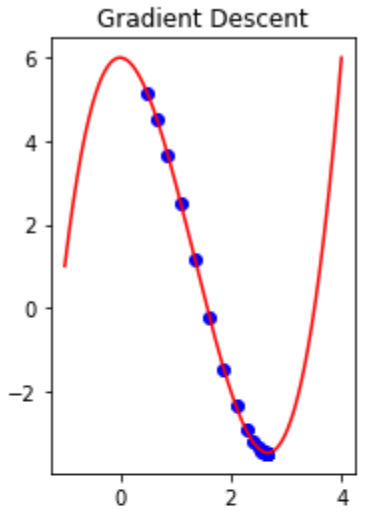
x_start = 0.5 ,precision = 0.001 , learning rate = 0.01

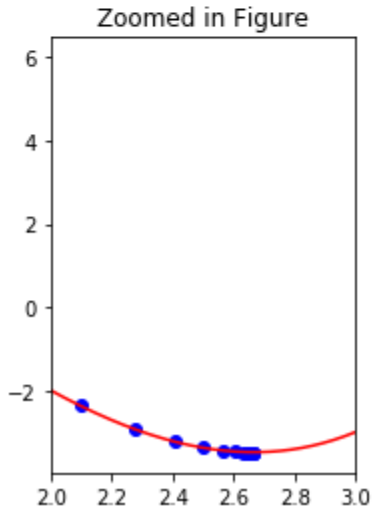
Since learning rate was lesser, which means the number of steps taken to reach local minimum was higher (85). As we can see in the graph, 85 x values plotted in blue, meaning our Algorithm was slower in finding local minimum.

Learning rate = 0.05

gradient_precision(0.5, 0.001, 0.05)

```
Local minimum occurs at: 2.67
Number of steps taken:  20
```





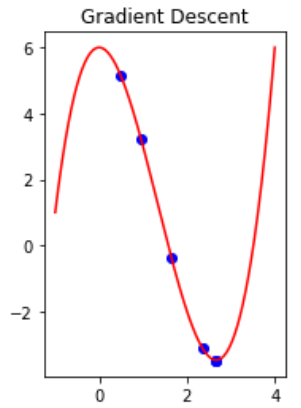
x_start = 0.5 ,precision = 0.001 , learning rate = 0.05

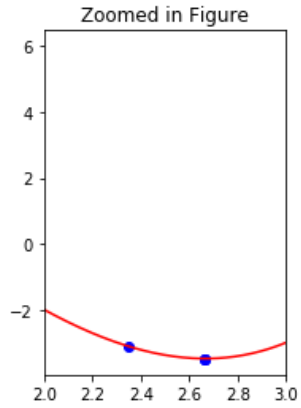
For the same precision value and x_start value, but learning rate = 0.05, we see that our Algorithm was able to find local minimum in just 20 steps. This shows that by increasing learning rate , the algorithm reaches local minimum faster.

Learning rate = 0.14

gradient_precision(0.5, 0.001, 0.14)

```
Local minimum occurs at: 2.67
Number of steps taken:  6
```

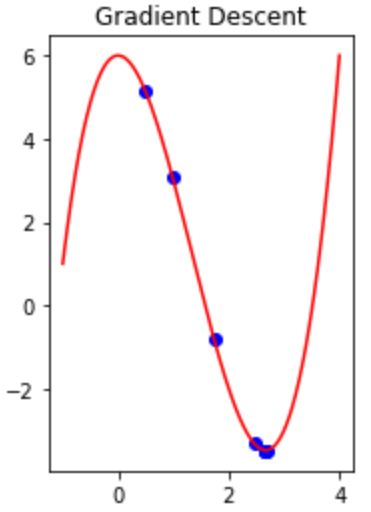


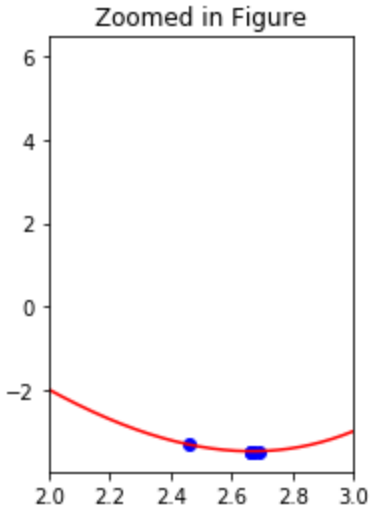


x_start = 0.5 ,precision = 0.001 , learning rate = 0.14

By increasing the learning rate to 0.14, the Algorithm was able to find local minimum in just 6 steps. Don't fall into the trap that increasing learning rate will always reduce the number of iterations the algorithm takes to find the local minimum. Let's just increase the learning rate by 0.01 and see the results.

```
Local minimum occurs at: 2.67
Number of steps taken:  8
```





x_start = 0.5 ,precision = 0.001 , learning rate = 0.15

Tips for Gradient Descent

This section lists some tips and tricks for getting the most out of the gradient descent algorithm for machine learning.

- **Plot Cost versus Time**: Collect and plot the cost values calculated by the algorithm each iteration. The expectation for a well performing gradient descent run is a decrease in cost each iteration. If it does not decrease, try reducing your learning rate.
- **Learning Rate**: The learning rate value is a small real value such as 0.1, 0.001 or 0.0001. Try different values for your problem and see which works best.

- **Rescale Inputs**: The algorithm will reach the minimum cost faster if the shape of the cost function is not skewed and distorted. You can achieved this by rescaling all of the input variables (X) to the same range, such as [0, 1] or [-1, 1].
- **Few Passes**: Stochastic gradient descent often does not need more than 1-to-10 passes through the training dataset to converge on good or good enough coefficients.
- **Plot Mean Cost**: The updates for each training dataset instance can result in a noisy plot of cost over time when using stochastic gradient descent. Taking the average over 10, 100, or 1000 updates can give you a better idea of the learning trend for the algorithm.

Why it is popular?

1. Optimization is a big part of machine learning
2. Gradient descent is by far the most popular optimization strategy used in Machine Learning and Deep Learning at the moment.
3. It is used when training Data models, can be combined with every algorithm and is easy to understand and implement
4. Many statistical techniques and methods use GD to minimize and optimize their processes.