In [1]:

```
# Import required packages
import numpy as np                # for mathematical operations, for arrays
import pandas as pd               # Used for different data manipulation tasks
import matplotlib.pyplot as plt   # Used for Plotting
import random                     # For random number generator
import warnings
warnings.filterwarnings('ignore')# To avoid warnings note while running the notebook
```

WEEK 3: EE4708, ASSIGNMENT - 3a

Raj Jain | CH17B66

Section 1 : Implementing Gradient Descent

Gradient Descent is an first order optimization algorithm to find the local minimum differentiable function (usually a loss function or an objective function). We start with a random point on the function and move in the negative direction of the gradient of the function to reach the find minimum.

Let $F(x)$ be a differentiable function parameterised by x and x^* be the parameter value which minimises $F(x)$. The initial value is $x_0 = a_0$. To determine x^* using gradient descent, at each of the i^{th} the parameter updation rule is:

$$x_i = x_{i-1} - \eta \nabla F(a_{i-1}), \qquad i \geq 1$$

where x_i is updated value of the parameter in the i^{th} iteration.

$$\eta \nabla F(a_{i-1}) = \eta \frac{dF}{dx}|_{x=a_{i-1}}$$

Stopping Criteria: Following are three ways which are used as stopping criteria in gradient descent algorithm:

1. Limiting number of iterations
2. Threshold for change in the parameter (i.e difference between the updated parameter and parameter in the previous iteration) --> Change in parameter > Threshold
3. Threshold for change in gradient step

Question : Find x^* which minimises $F(x) = (x+5)^2$ using gradient descent

Note : Few variables are set to a given value, read the instructions in the pseudo code

Pseudo Code:

1. Initialise the parameter ('x') with some random number,set learning rate = 0.01(section 4 has details on "how to select appropriate learning rate for a given problem"), set threshold = 0.000001,
2. Define a function for calculating differential of F (Hint : Use lambda method for single line of code)
3. Compute the updated value of the parameter using the formula: $x_i = x_i - 1 - \eta \nabla F(a_{i-1})$
4. Iterate over step 3 until the stopping criteria is satisfied (You have to choose stopping criteria appropriately)

1.1 Implementing Code

Implement the above pseudo code and print the optimum value of x and $F(x)$. Also print the number of iterations required to reach that optimum.

Note : Carefully choose your stopping criteria for reaching optimum

In [106]:

```
# Write your code here
#Initializing
x = 1
eta = 0.01 #Learning Rate
threshold = 0.000001
#Gradient Descent - dF/dx = 2(x+5)
while True:
    x_new = x - eta*(2*(x+5))
    if abs(x_new-x)<threshold:
        x = x_new
        break
    x = x_new

print('Optimum value of x which minimises F(x) = (x+5)^2 is = %.2f' %x)
```

```
Optimum value of x which minimises F(x) = (x+5)^2 is = -5.00
```

Section 2 : Linear Regression - Finding the Regression parameters using Normal Equation (Analytical Approach)

Dataset : dataset1.csv (given)

Given a dataset (here dataset1.csv), find the coefficients (parameters) using the Normal Equation and predict the outputs for the same dataset (dataset1.csv). Given input features in matrix X and observed output or target vector y, the regression parameters are given by the equation:

$$\theta = \left(X^T X\right)^{-1} X^T \vec{y}.$$

Pseudo Code:

1. Append ones to X, let say that as "X_appended" (This is to be done when there is intercept, if you know before hand that intercept is zero, then this step can be ignored)
2. Find the pseudo Inverse of the X_appended
3. Find the dot product of transpose of X_appended and target vector (y),
4. Find "theta" (parameters) using "Normal Equation" given above (i.e Matrix multiplication of outputs of Step 2 and Step 3)
5. Predict ouputs for X*new by appending it with ones and then using the formula $y=X*{new}\theta$

Note : When you perform matrix multiplications or dot products on vectors or matrices, always keep track of the matrix or vectors shapes. This helps you in avoiding implementation errors.

Implementation Note: Step 1 is crucial. We store each example as a row in the the X matrix in Python `numpy`. To take into account the intercept term (θ_0), we add an additional first column to X and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

In [2]:

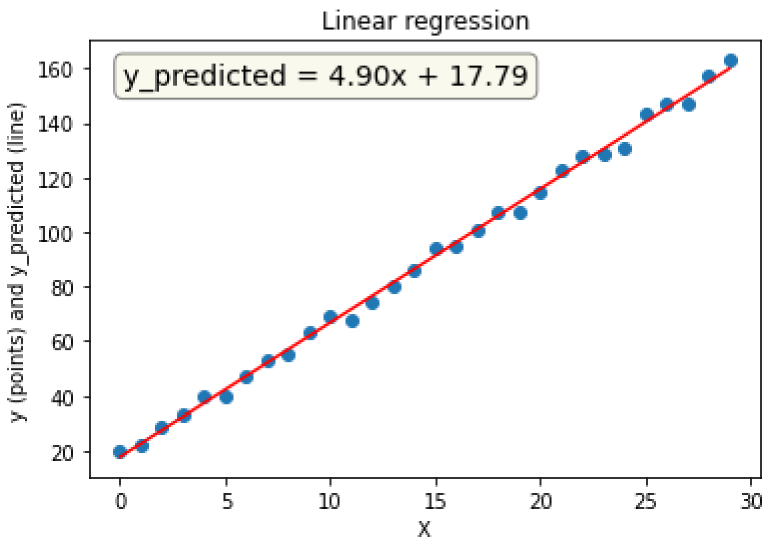
```
# Import X, y from "dataset1.csv"
# X and y are numpy array, Hint : store the data in the pandas dataframe and then conve
rt it to numpy array
dataset1 = pd.read_csv("dataset1.csv")
X = dataset1["X"].to_numpy()
y = dataset1["y"].to_numpy()
```

2.1 Function to predict the outputs

Implement the predict function below. Predict the outputs for the given data (dataset1.csv), plot the predicted outputs Vs input features and scatter plot for dataset1.csv (X,y)

In [54]:

```
def predict(X, theta):
    """
    Predictions of X for a given theta.

    Parameters
    ----------
    X : array with size of (m x n)

    theta : array with size of (n+1, 1).

    Returns
    -------
    predictions : array of size (m x 1)

    """
    X_reshaped = np.reshape(X,(X.size,1))
    m=X_reshaped.shape[0]
    X_appended = np.append(X_reshaped,np.ones((m,1)),axis=1)

    prediction=np.dot(X_appended,theta)
    return prediction

# Your predictions for the dataset1.csv

X_reshaped = np.reshape(X,(X.size,1))
y_reshaped = np.reshape(y,(y.size,1))
m=X_reshaped.shape[0]
X_appended = np.append(X_reshaped,np.ones((m,1)),axis=1)

theta = np.dot(np.linalg.pinv(X_appended),y_reshaped)

predictions = predict(X,theta)


# plot a line Predictions Vs features and scatter plot for the training data X,y
fig,ax = plt.subplots()

textstr = 'y_predicted = %.2fx + %.2f' %(theta[0],theta[1])
ax.plot(X,predictions,'r')
ax.scatter(X,y)
ax.set_xlabel('X')
ax.set_ylabel('y (points) and y_predicted (line)')
ax.set_title('Linear regression')

props = dict(boxstyle='round', facecolor='beige', alpha=0.5)
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)
plt.show();
print(theta.shape)
```



```
(2, 1)
```

Section 3 : Linear Regression - Find the Regression parameters using Gradient Descent

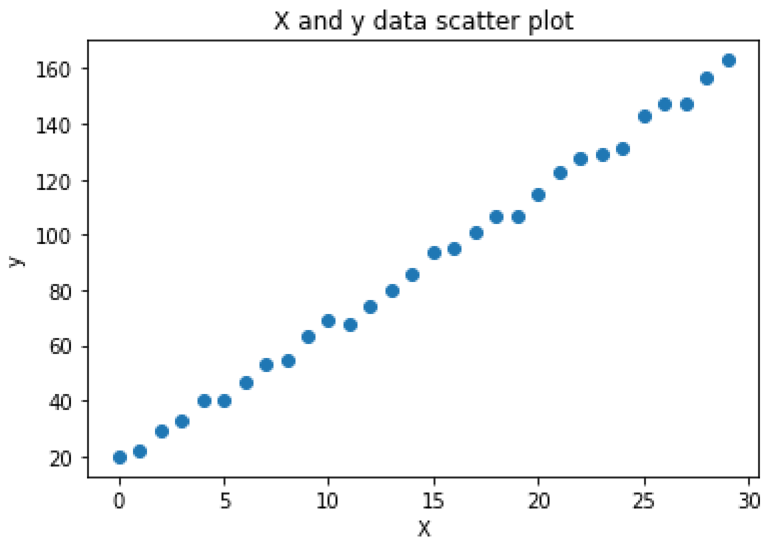
This section is about applying gradient descent algorithm to find theta at which a cost function (Mean Squared Error) is minimum. This can be done using the gradient descent algorithm discussed in Section 1, where x is replaced by the regression parameters.

Refer to pseudo code in Section 1 for gradient descent algorithm. In simple terms, we initialise the regression parameters with some random number and we compute the gradient step to update the regression parameters.

3.1. Import and visualise the dataset

In [4]:

```
# Import the data from dataset1.csv - Same as Section 2
# Import X and y as numpy array, Hint : store the data in the pandas dataframe and then
convert it to numpy array
dataset1 = pd.read_csv("dataset1.csv")
X = dataset1["X"].to_numpy()
y = dataset1["y"].to_numpy()

# Scatter Plot the data X and y for visualisation
plt.scatter(X,y)
plt.xlabel('X')
plt.ylabel('y')
plt.title('X and y data scatter plot')
plt.show()
```



3.2. Compute Cost Function - Mean Squared Error (MSE)

The cost function used is Mean Squared Error (MSE) represented as $J(\theta)$ and is given by:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where the hypothesis $h_\theta(x)$ is a linear model given by:

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

As you perform gradient descent to learn minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. This way you can know whether you choose high learning rate or low learning rate. Suppose if your learning rate is too high, you can see your cost will go up and down, but will not converge. If you choose good learning rate, then you can see your cost going down smoothly and converges.

In this you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation.

In [11]:

```
def computeCost(X, y, theta):
    """
    Compute cost for linear regression. Computes the cost of using theta as the
    parameter for linear regression to fit the data points in X and y.

    Parameters
    ----------
    X : array with the shape of (m x n+1)
        where m is the number of examples,n is the number of features
        n+1 == n features + 1 for bias term(intercept as a feature)

    y : array with the shape of (m,1)

    theta : array with the shape of (n+1,1)

    Returns
    -------
    J : float -> The value of the regression cost function.

    """
    # Write your code here
    m = X.shape[0]
    n = X.shape[1] - 1

    J = (1/(2*m))*np.sum((np.dot(X,theta)-y)**2)

    return J
```

3.3. Implementation of Gradient Descent

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \qquad \text{simultaneously update } \theta_j \text{ for all } j$$

With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost J(θ).

Important implementation notes

The function `gradientDescent` calls `computeCost` on every iteration and saves the cost to a python list. If you have implemented gradient descent and `computeCost` correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm.

In [97]:

```
def gradientDescent(X_appended, y, theta, alpha, precision = 0.001, max_iter=25000):
    """
    Performs gradient descent to learn `theta`. Updates theta by taking `num_iters`
    gradient steps with learning rate `alpha`.

    Parameters
    ----------
    X_appended : array with shape of (m x n+1). Note : n+1 = n features + 1 intercept

    y : array with shape of (m, 1)

    theta : array with shape of (n+1,1)

    alpha : float value, called as "learning rate"

    precision : float, (One of the Stopping criteria). This is compared with the change
in the cost from previous iter.

    Returns
    -------
    theta : array wih shape of (n+1,1),The learned linear regression parameters

    J_history : A python list for the values of the cost function after every iteratio
n. This is to check for convergence

    Count : Integer, Number of iterations taken to converge

    Cost : Float, Mean squared error at the end of the iteration


    Hint:
    ------------
    1. Peform a single gradient step on the parameter vector theta.
    2. Loop over the number of iterations to update step by step.
    """
    # Write your code here
    J_history = []
    J = computeCost(X_appended,y,theta)
    J_history.append(J)

    m = X_appended.shape[0]
    n = X_appended.shape[0] - 1
    count = 1
    while True:
        count+=1
        theta = theta - alpha*(1/m)*(np.dot(np.transpose(X_appended),(np.dot(X_appended
,theta) - y)))
        J = computeCost(X_appended,y,theta)
        J_history.append(J)
        cost = J
        if abs(J - J_history[-2]) < precision:
            break

        if count>max_iter:
            break

    return theta, J_history, cost, count
```

Run gradientDescent function to train the model here

Note : Use the learning rate(alpha) and precision given below. Initialise theta with zeros is suggested here (but you can initialise with different numbers)

Print the final theta (learnt parameters), number of iterations to converge, value of cost function at the convergence.

In [102]:

```
# initialize fitting parameters with zeros
theta = np.zeros((2,1))

# Play with these setting to see how these parameters play a huge, for a decent converg
e use below parameters
precision = 0.000001
alpha = 0.001

##################################### Your Code here ##################################
######
theta, J_history, cost, count = gradientDescent(X_appended, y_reshaped, theta, alpha =
0.001, precision = 0.000001)
print('Parameters theta are',theta)
print('Final training loss is: %.2f' %cost)
print('No. of iterations is %d' % count)
```

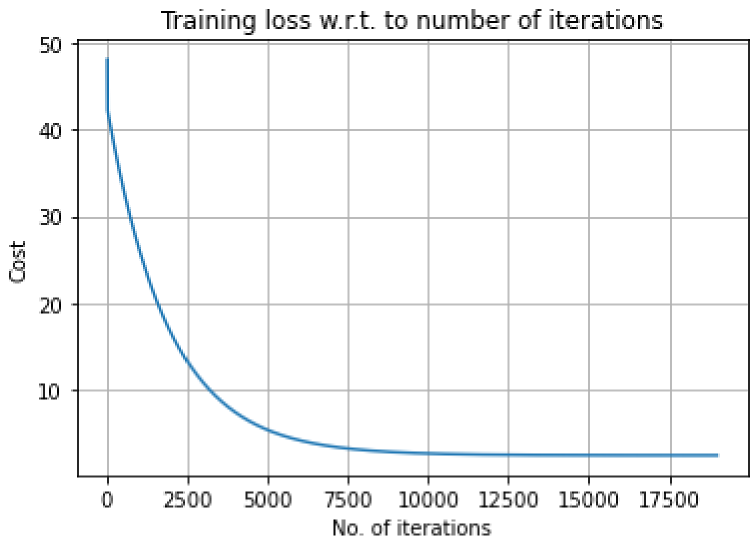
```
Parameters theta are [[ 4.90802378]
 [17.6687466 ]]
Final training loss is: 2.42
No. of iterations is 18996
```

3.4. Plot the Training Curve

Tip: Initially the change in the loss(J) are drastic and dominating. So while plotting you may remove the first 10% of the iterations and then plot to see a smooth curve in the training process

In [99]:

```
plt.plot(np.arange(10,len(J_history),1),J_history[10:])
plt.ylabel('Cost')
plt.xlabel('No. of iterations')
plt.title('Training loss w.r.t. to number of iterations')
plt.grid()
plt.show();
```

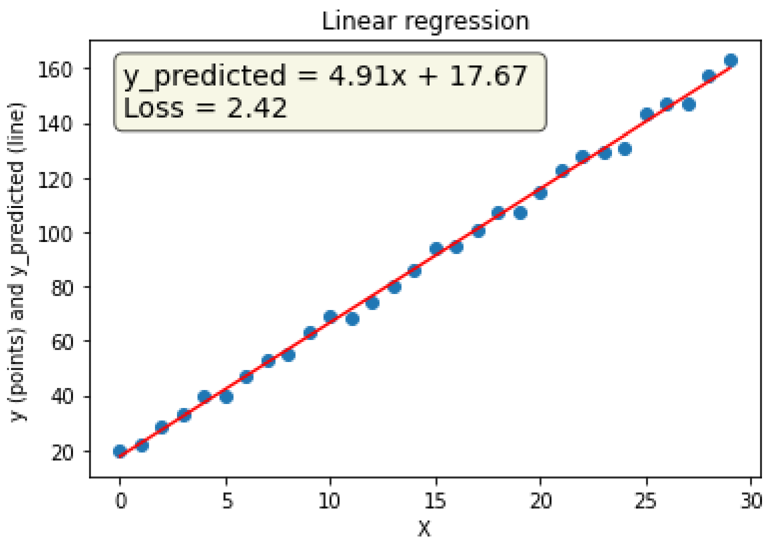


3.5. Predict Outputs and Plot the results

Same as Step 5 in the Section 2. You can use the same function predict(X, theta)

In [100]:

```
# plot a line for predictions and scatter plot for the training data X,y
predictions = predict(X,theta)

fig,ax = plt.subplots()

textstr = 'y_predicted = %.2fx + %.2f \nLoss = %.2f' %(theta[0],theta[1],cost)
ax.plot(X,predictions,'r')
ax.scatter(X,y)
ax.set_xlabel('X')
ax.set_ylabel('y (points) and y_predicted (line)')
ax.set_title('Linear regression')

props = dict(boxstyle='round', facecolor='beige', alpha=0.7)
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)
plt.show();
```



Section 4 : Selecting appropriate learning rate

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and 'blow up', resulting in values which are too large for computer calculations. In these situations, `numpy` will tend to return NaNs. NaN stands for 'not a number' and is often caused by undefined operations that involve $-\infty$ and $+\infty$. So dont worry even if you cost as inf or NaN

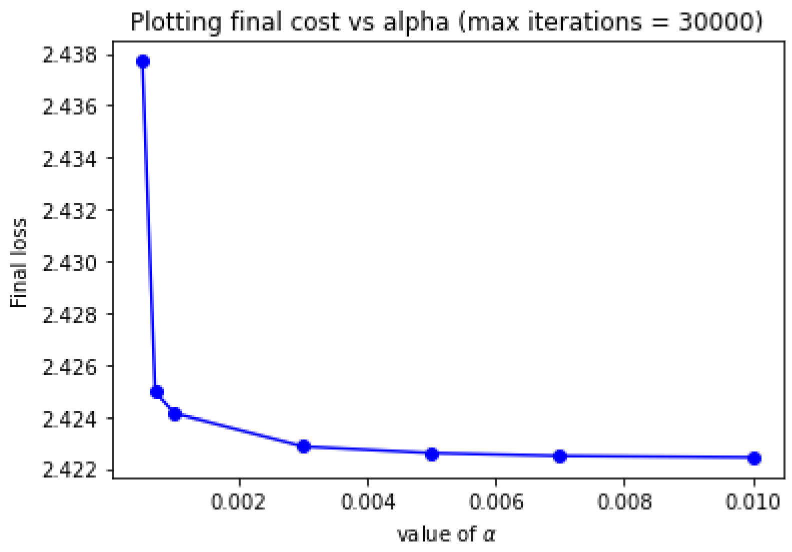
Repeat the training as in Section 3 with different values of alpha as listed below. Print the alpha, cost and number of iterations it took for every alpha

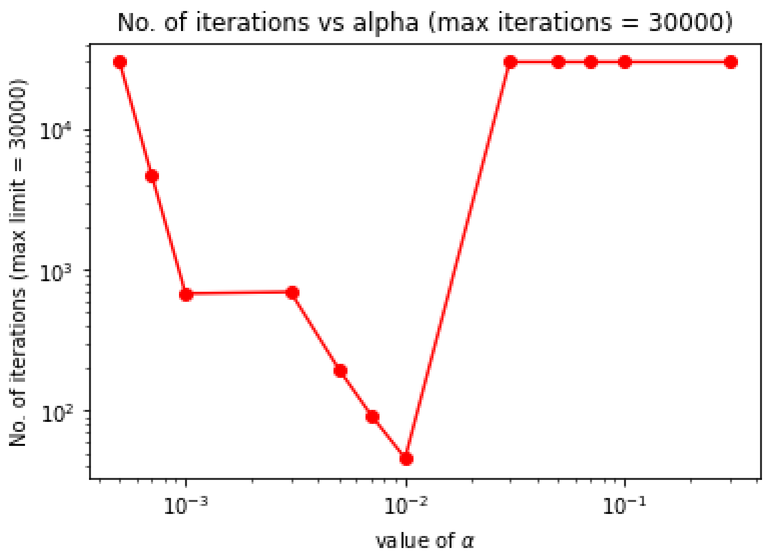
In [82]:

```
theta = np.zeros((2,1))
precision = 0.000001

# You can change this set of alphas and try out different, but keep your range less tha
n 1
alphas = [0.0005, 0.0007, 0.001, 0.003, 0.005, 0.007, 0.01,0.03, 0.05, 0.07, 0.1, 0.3]
cost_alpha = []
count_alpha = []
# Write your code here
for alpha in alphas:
    theta, J_history, cost, count = gradientDescent(X_appended, y_reshaped, theta, alph
a, precision = 0.000001, max_iter=30000)
    cost_alpha.append(cost)
    count_alpha.append(count)
```

In [114]:

```
#Plotting final cost for alpha (max iterations = 30000)
plt.plot(alphas,cost_alpha,color="blue",marker="o")
plt.xlabel('value of '+ r'$\alpha$')
plt.ylabel('Final loss')
plt.title('Plotting final cost vs alpha (max iterations = 30000)')
plt.show();


plt.loglog(alphas,count_alpha,color="r",marker="o")
plt.xlabel('value of '+ r'$\alpha$')
plt.ylabel('No. of iterations (max limit = 30000)')
plt.title('No. of iterations vs alpha (max iterations = 30000)')
plt.show();
```





Section 5: Questions on Analytical and Gradient Descent approaches

1. Are the results from both the analytical and gradient descent approach are same? If not, why?
2. When do u prefer analytical approach and gradient descent approach? Hint: Explain interms of size of the dataset
3. Can we use different cost function in place of MSE? If we do, where will be changes in the gradient descent step?(Just mention using words, equations are not required)
4. How does the initialisation of the parameters θ effects the convergence? In this case we initialised θ to zeros. What happens if we choose closer to solution and farther to solution?
5. Any drawbacks of Gradient descent that you could think of? Hint : Interms of Convergence rate.
6. What if number of features increases and how does it effect the gradient descent approach and analytical approach? Note:This might require some time to look for appropriate concepts. But do note that this difference is very important to understand.
7. Comment your observations on selecting appropriate learning rate.

Write your answers in the following Markdown

1. Are the results from both the analytical and gradient descent approach are same? If not, why?

The two results are not the same as gradient descent is an iterative numerical approach to find the optimum. It takes us close to the optimum in the specified precision range but still does not reach the theoritical optimum. On the other hand, the analytical solution is the exact optimum here.

2. When do u prefer analytical approach and gradient descent approach?

The main reason why gradient descent is used for linear regression is the computational complexity (Matrix inversion cost is order of O(N^3): it's computationally cheaper (faster) to find the solution using the gradient descent in some cases with large datsets. For SMALLER DATASETS analytical approach is preferred.

3. Can we use different cost function in place of MSE? If we do, where will be changes in the gradient descent step?

We can use mean absolute error cost instead of MSE. Cost is defined then as the mean absolute difference between the predicted and true 'y'. The update rule for theta needs to be changed accordingly. The gradient of each datapoint would be 1 if absolute difference is positive, and -1 if negative.

4. How does the initialisation of the parameters θ effects the convergence? In this case we initialised θ to zeros. What happens if we choose closer to solution and farther to solution?

Initialization of θ affects the convergence. For a sufficiently small learning rate, initialising closer to the optimum would result in faster convergence as compared to farther initialization. For more complicated functions initialization becomes important to reach convergence as initialisation from some points may get stuck in local minimas.

5. Any drawbacks of Gradient descent that you could think of? Hint : Interms of Convergence rate.

As we reach closer to the optimum, gradient descent converges the cost slowly. Also, if the learning rate is not suitable then we may never reach an optimum.

6. What if number of features increases and how does it effect the gradient descent approach and analytical approach?

As the number of features increases, calculating analytical solutions becomes more and more computationaly complex. Gradient descent is a better alternative in that case.

7. Comment your observations on selecting appropriate learning rate.

For larger learning rates (>0.03), the cost blows up and we do not achieve convergence. For smaller learning rates, the number of iterations is very high. For a learning rate of 0.01, convergence is achieved in less than 100 iterations, and hence is the best option among the ones we tried.

```
In [ ]:
```