Code Design

Code can be divided into following subsections for clear understanding:

A. Importing Required Libraries:

You need to import some essential basic manipulation libraries to do some math operations and hence we are importing these libraries here. The comments in the code specify exactly what purpose the library serves.

B. Data Preprocessing

1. First we use pandas library to read the dataset from the location where it has been uploaded.

   The dataset used in this project consists of three independent features i.e. age(X1), bmi(X2) and number of children of an individual(X3) and a dependent feature insurance amount for that person(Y).Our goal is to use these independent features to predict the values of dependent feature for which we need to devise algorithms to learn the coefficients w.r.t. each independent variable.

2. Since the given dataset contains features with different scales hence we need to perform feature scaling i.e. we need to normalize/ standardize the entire dataset using

   X -> (X - mean)/(standard deviation)

3. Train Validation and Test Splitting the original dataset as per 70:10:20 ration

C. Data Learning Without Using SKLearn Library

1. Defining Our Evaluation Metric Class:

This class is used to calculate how efficient our model is. Now we know that there are various evaluation metric measures but here we have implemented three of them namely RMSE(root mean squared error) , MSE(mean squared error) and Total Error. Formula for each of these are given below.

$$\text{RMSE} = \sqrt{(\Sigma_1^n \quad (y - \underline{y})^2)/n}$$

$$\text{MSE} = (\Sigma_1^n \quad (y - \underline{y})^2)/n$$

$$\text{Total Error(SSRES)} = 0.5 * \Sigma_1^n \quad (y - \underline{y})^2$$

For obvious reasons the lower the error i.e. closer to 0 the better is our model.All of these evaluation metrics are proportional to each other and hence can be used based on one's will.

2. Using Normal Equations (Vectorized Formula) as Optimization Method:

Here first I have created a class which performs this optimization for us. The name of the class is 'NormalEquation'.This class contains several functions whose purpose is clearly mentioned in the code.

To understand the math and derivation of the vectorized formula used in this class the reader must go through the notes linked above. Once the reader has understood the mathematical derivation for the vectorized formula for solving multiple linear regression using the normal equation provided in the notes the reader will take no time to understand the code with the additional help provided via comments in the code wherever necessary.

The object of class ‘NormalEquation’ and ‘Evaluation Metric’ is created to use both these classes.

Since there is no hyperparameter in normal equations to be tuned there is no use of validation dataset here

Train and Test Results using best hyperparameters

```
Cofficients : [-0.02981352  0.2999276   0.15798607  0.03405244]

RMSE--Train: 0.9263889843274103
MSE--Train: 0.858196550283171
Total Error--Train: 401.63598553252405

RMSE--Test: 0.9814144893426208
MSE--Test: 0.9631743998916371
Total Error--Test: 129.06536958547937
```

3. Using Gradient Descent (Vectorized Formula) as Optimization Algorithm:

Here first I have created a class which performs this optimization for us. The name of the class is ‘GradientDescent’.This class contains several functions whose purpose is clearly mentioned in the code.
To understand the math and derivation of the vectorized formula used in this class the reader must go through the notes linked above. Once the reader has understood the mathematical derivation for the vectorized formula for solving multiple linear regression using the gradient descent provided in the notes the reader will take no time to understand the code with the additional help provided via comments in the code wherever necessary.

In this method we don't directly go to the minima but instead we reach close to the global minima of the cost/error function gradually by jumping from point to another point.This new point in every iteration is decided by the gradient at that old point.

As we know gradient descent can be implemented with different types of stopping criteria but here the stopping criteria for this implementation that I have taken is when change in error difference is less than 0.00000001 but if no of iterations specified by the user is less than the no of iterations taken to reach error difference less than 0.00000001 then the algorithm stops when the no of iterations is equal to the one defined by the user.

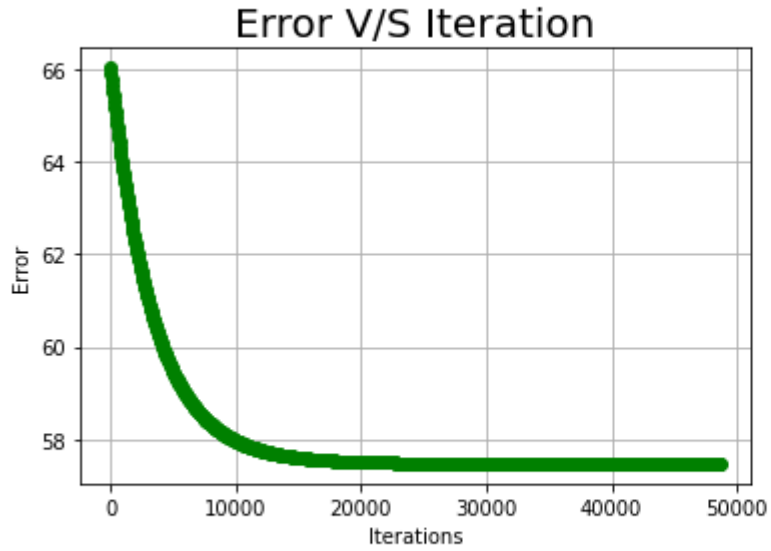
The object of class 'GradientDescent' and 'Evaluation Metric' is created to use both these classes.

Performing Hyperparameter Tuning Using Validation Dataset

Now as we know the learning rate is a hyperparameter I am doing hyperparameter tuning by trying out 3 different learning rates using the validation dataset.

For Learning Rate = 10^{-06}

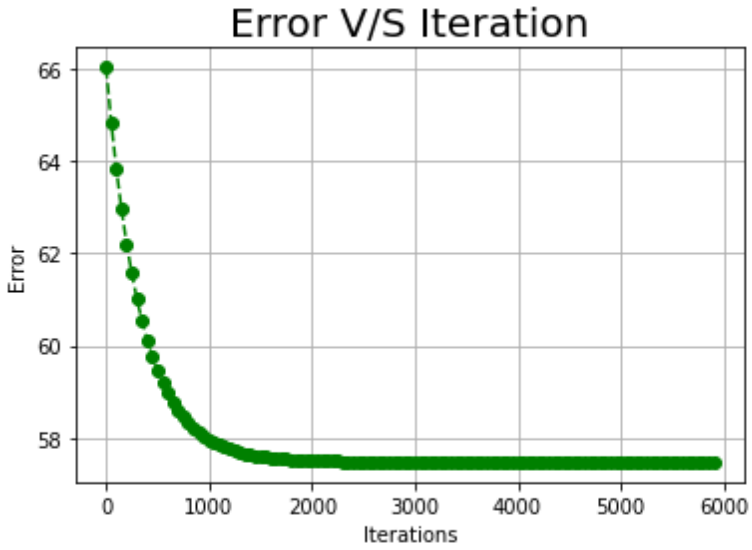
```
Cofficients : [0.08527702 0.25221958 0.17837615 0.12299002]
Text(0.5, 0, 'Iterations')Text(0, 0.5, 'Error')Text(0.5, 1.0, 'Error V/S Iteration')
[<matplotlib.lines.Line2D at 0x7fea63187190>]
```




```
RMSE--Validation: 0.9261870107561726
MSE--Validation: 0.8578223788934546
Total Error--Validation: 57.47409938586146
```

For Learning Rate = 10^{-05}

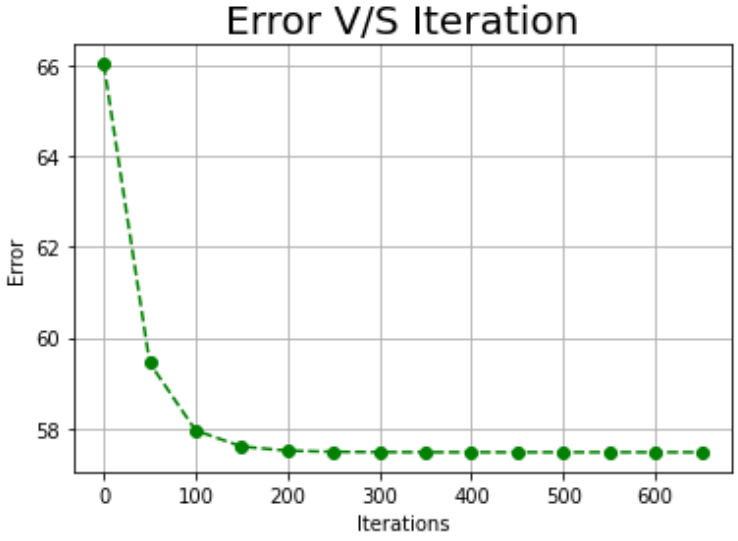
```
Cofficients : [0.0855208  0.25277892 0.17825576 0.12302133]
Text(0.5, 0, 'Iterations')Text(0, 0.5, 'Error')Text(0.5, 1.0, 'Error V/S Iteration')
[<matplotlib.lines.Line2D at 0x7fea62c80150>]
```




```
RMSE--Validation: 0.9261866812133527
MSE--Validation: 0.8578217684570046
Total Error--Validation: 57.47405848661931
```

For Learning Rate = 10^{-03}

```
Cofficients : [0.08558922 0.25296096 0.17820213 0.12301935]
Text(0.5, 0, 'Iterations')Text(0, 0.5, 'Error')Text(0.5, 1.0, 'Error V/S Iteration')
[<matplotlib.lines.Line2D at 0x7fea62bf0e90>]
```




```
RMSE--Validation: 0.9261866475547985
MSE--Validation: 0.8578217061087965
Total Error--Validation: 57.474054309289365
```

As we can conclude from the following graphs that as the learning rate decreases the no of iterations taken to reach the minima increases thus training time increases.

As we can observe that validation error achieved for different learning rates are close to what was achieved using normal equations, this ensures that our gradient descent algorithm is converging close to the global minima of the cost/error function because in the normal equation what we calculated is error exactly at the global minima.

The minimum validation error differs slightly from each other in each case because as we know we are not reaching exact global minima we reach very close to the global minima and hence at different times we might reach different close to the global minima.

Hence the best learning rate is 10^(-3) and hence using this for next steps

Train and Test Results using best hyperparameters

```
Cofficients : [-0.02980791  0.29991968  0.15798763  0.03405608]

RMSE--Train: 0.9263889843831978
MSE--Train: 0.8581965503865328
Total Error--Train: 401.63598558089734

RMSE--Test: 0.9814131414593207
MSE--Test: 0.9631717542290525
Total Error--Test: 129.06501506669304
```

4. Using Stochastic Gradient Descent (Vectorized Formula) as Optimization Algorithm:

Here first I have created a class which performs this optimization for us. The name of the class is 'StochasticGradientDescent'.This class contains several functions whose purpose is clearly mentioned in the code.

To understand the math and derivation of the vectorized formula used in this class the reader must go through the notes linked above. Once the reader has understood the mathematical derivation for the vectorized formula for solving multiple linear regression using the stochastic gradient descent provided in the notes the reader will take no time to understand the code with the additional help provided via comments in the code wherever necessary.

This algorithm is almost exactly the same as gradient descent but the only difference here is that while considering the error function in gradient descent we used error due to all points but in stochastic gradient descent we consider error only due to a single random point in the dataset.

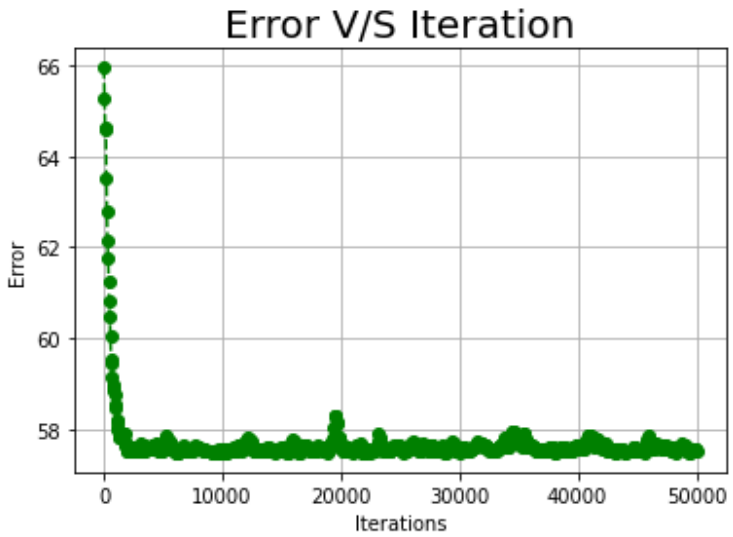
Here I have taken no of iterations specified by the user as the stopping criteria.

Performing Hyperparameter Tuning Using Validation Dataset

Now as we know the learning rate is a hyperparameter I am doing hyperparameter tuning by trying out 3 different learning rates using the validation dataset.

For Learning Rate = 0.001

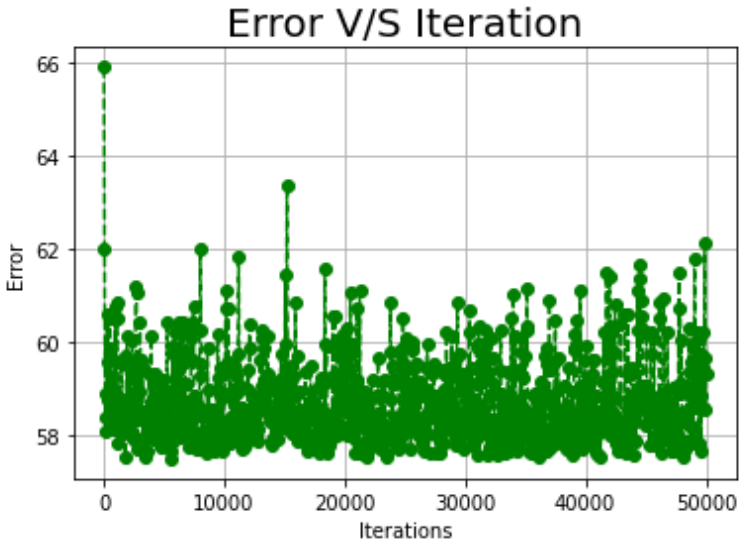
```
Cofficients : [0.10789888 0.25351574 0.18632207 0.13510856]
Text(0.5, 0, 'Iterations')Text(0, 0.5, 'Error')Text(0.5, 1.0, 'Error V/S Iteration')
[<matplotlib.lines.Line2D at 0x7fea62b330d0>]
```




```
RMSE--Validation: 0.9265685525023666
MSE--Validation: 0.8585292824863308
Total Error--Validation: 57.52146192658416
```

For Learning Rate = 0.01

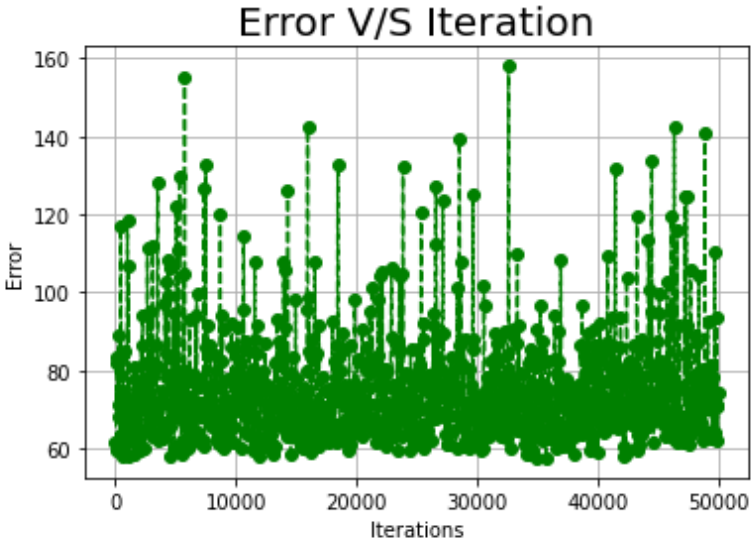
```
Cofficients : [0.00924915 0.30992488 0.03409306 0.10033304]
Text(0.5, 0, 'Iterations')Text(0, 0.5, 'Error')Text(0.5, 1.0, 'Error V/S Iteration')
[<matplotlib.lines.Line2D at 0x7fea62b03e50>]
```




```
RMSE--Validation: 0.9409664176439494
MSE--Validation: 0.8854177991336875
Total Error--Validation: 59.322992541957056
```

For Learning Rate = 0.1

```
Cofficients : [0.37546785 0.66815513 0.27407633 0.22060282]
Text(0.5, 0, 'Iterations')Text(0, 0.5, 'Error')Text(0.5, 1.0, 'Error V/S Iteration')
[<matplotlib.lines.Line2D at 0x7fea62a72b10>]
```




```
RMSE--Validation: 1.0538124204842174
MSE--Validation: 1.110520617566805
Total Error--Validation: 74.40488137697594
```

Hence the best learning rate is 0.001 and hence using this for next steps

Train and Test Results using best hyperparameters

```
Cofficients : [-0.04985783  0.29977371  0.10749117  0.0719731 ]

RMSE--Train: 0.9287108537581553
MSE--Train: 0.8625038498882017
Total Error--Train: 403.6518017476784

RMSE--Test: 0.9854823830218444
MSE--Test: 0.9711755272464132
Total Error--Test: 130.13752065101937
```

D. Data Learning Using SKLearn Library

Firstly an object is created of the linear regression class already defined in SKLearn Library.Then the validation dataset is fitted using this object.

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
RMSE --validation : 0.9261866438534724
R2_Score --validation : 0.12703877483387993
```

Then the train and test dataset were fitted and following results were obtained

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
linear model coeff (w): [0.2999276  0.15798607 0.03405244]
linear model intercept (b): -0.029813521392349177

rmse --train = 0.9263889843274103
r2_score --train = 0.13083309574623003
rmse --test = 0.9814144893426208
r2_score --test = 0.08303693882592467
```

Hence we can see that these values are very close to the one that I got via my implementation and this confirms that what I have implemented is absolutely correct.