Lab Course Machine Learning
Exercise 3

1.1 Data preprocessing:

Function name : read_data
Parameter : Filename and column names

This function reads the data and returns the data in the format of data frame along with the headings.

```
def read_data(filename,columns):
    if columns == None:
        data = pd.read_csv(filename, sep='\s+', delimiter = ";")
        return data
    data = pd.read_csv(filename, sep='\s+',header = None)
    data.columns = columns
    return data
```

Output:

Airline data

	City1	City2	Avg_Fare1	Distance	Avg_Psngr/week	Ld_Airline	Mrkt_Share1	Avg_Fare2	Low_Prc_Air	Mrkt_Share2	Price
0	CAK	ATL	114.47	528	424.56	FL	70.19	111.03	FL	70.19	111.03
1	CAK	MCO	122.47	860	276.84	FL	75.10	123.09	DL	17.23	118.94
2	ALB	ATL	214.42	852	215.76	DL	78.89	223.98	CO	2.77	167.12
3	ALB	BWI	69.40	288	606.84	WN	96.97	68.86	WN	96.97	68.86
4	ALB	ORD	158.13	723	313.04	UA	39.79	161.36	WN	15.34	145.42

Red Wine data

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

White Wine data

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6

Function name : check_na_null
Parameter : Dataframe and column names

This function checks the presence of null values or NA or nan values within the dataset and removes the same. This function also generates dummies for non-numeric values and returns the same.

```
def check_na_null(data,column_dummies):
    data.isnull().values.any()
    data.dropna(inplace = True)
    data = pd.get_dummies(data=data, columns=column_dummies)
    return data
```

Output:

Dummies and nan or NA value check for Airline data

	Avg_Fare1	Distance	Avg_Psngr/week	Mrkt_Share1	Avg_Fare2	Mrkt_Share2	Price	City1_ABQ	City1_ACY	City1_ALB	...	Low_Prc_Air_G4	Low_Prc_Air_
0	114.47	528	424.56	70.19	111.03	70.19	111.03	0	0	0	...	0	
1	122.47	860	276.84	75.10	123.09	17.23	118.94	0	0	0	...	0	
2	214.42	852	215.76	78.89	223.98	2.77	167.12	0	0	1	...	0	
3	69.40	288	606.84	96.97	68.86	96.97	68.86	0	0	1	...	0	
4	158.13	723	313.04	39.79	161.36	15.34	145.42	0	0	1	...	0	

5 rows × 217 columns

Function name : text_to_number
Parameter : Dataframe and column names

This function converts the text within a column to numbers. For example, a city with name ABQ will be converted to a specific number for all the available instances.

```
def text_to_number(data,column):
    for i in column:
        Airdata[i] = pd.Categorical(Airdata[i])
        Airdata[i] = Airdata[i].cat.codes
    return data
```

Output:

Alphabetical data within City1, City2, Leading and Low Price Airlines are converted to numbers

	City1	City2	Avg_Fare1	Distance	Avg_Psngr/week	Ld_Airline	Mrkt_Share1	Avg_Fare2	Low_Prc_Air	Mrkt_Share2	Price
0	16	0	114.47	528	424.56	6	70.19	111.03	8	70.19	111.03
1	16	40	122.47	860	276.84	6	75.10	123.09	6	17.23	118.94
2	2	0	214.42	852	215.76	4	78.89	223.98	5	2.77	167.12
3	2	7	69.40	288	606.84	14	96.97	68.86	17	96.97	68.86
4	2	52	158.13	723	313.04	12	39.79	161.36	17	15.34	145.42

Function name : create_Test_Train_data
Parameter : Data of input variables and target

This function splits the input variables to Train and Test data with 80% and 20% of the entire data respectively.

```
def create_Test_Train_data(X_data,Y_data):
    Y_train = Y_data[:math.ceil(0.8*len(Y_data))]
    Y_test = Y_data[math.ceil(0.8*len(Y_data)):]
    X_train = X_data[:math.ceil(0.8*len(X_data))]
    X_test = X_data[math.ceil(0.8*len(X_data)):]
    return Y_train,Y_test,X_train,X_test
```

Output:

Airline data

```
X_train:  (800, 6)
Y_train:  (800,)
X_test:   (200, 6)
Y_test:   (200,)
```

White wine data

```
X_train:  (3919, 5)
Y_train:  (3919,)
X_test:   (979, 5)
Y_test:   (979,)
```

Red wine data

```
X_train:  (1280, 5)
Y_train:  (1280,)
X_test:   (319, 5)
Y_test:   (319,)
```

1.1 Linear Regression with Gradient Descent:

1.1.1 Airline dataset:

To begin with the problem, first we've to find the correlation between the input variables and the price for better prediction of the target value. For this, we can use pearson co-efficient which gives the correlation between two variables.

Function name : pearson_coefficient
Parameter : Data for correlation

The Pearson correlation coefficient (PCC) is a measure of the linear correlation between two variables X and Y. PCC or 'r' has a value between +1 and −1

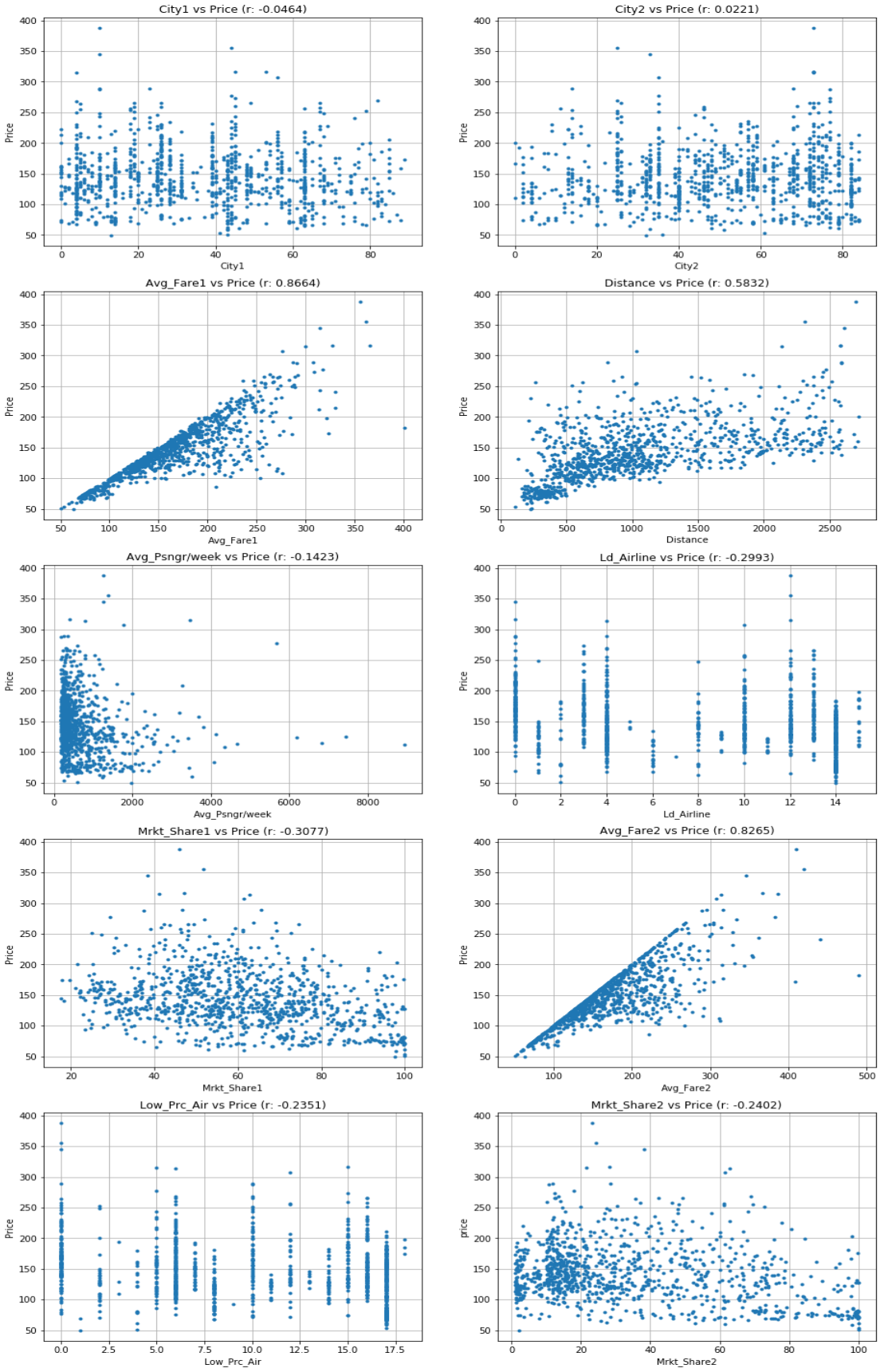
$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

```
def pearson_coefficient(x,y):
    x = x - np.mean(x)
    y = y - np.mean(y)
    return (np.sum(x*y)/np.sqrt(np.sum(x*x)*np.sum(y*y)))
```

The plot between the variables and price along with the correlation coefficient are as follows.

Output:

Correlation between Price and other variables for the airline data



From the plots between the input variables and price, it was concluded that there is no relation between City1, City2, Market Leading Airline, Low Price Airline and price. Hence, these columns were dropped down from the dataset to be used for creating the model.

```
Required_airdata = Airdata.drop(columns = ['City1','City2','
                                           Ld_Airline','Low_Prc_Air'])
```

Output:

Data after dropping the poorly correlated columns

	Avg_Fare1	Distance	Avg_Psngr/week	Mrkt_Share1	Avg_Fare2	Mrkt_Share2	Price
0	114.47	528	424.56	70.19	111.03	70.19	111.03
1	122.47	860	276.84	75.10	123.09	17.23	118.94
2	214.42	852	215.76	78.89	223.98	2.77	167.12
3	69.40	288	606.84	96.97	68.86	96.97	68.86
4	158.13	723	313.04	39.79	161.36	15.34	145.42

To begin with the model generation, we have to initialise the randomly distributed values for beta depending on the number of input variables

```
initialbeta = np.random.random_sample(X_data.shape[1]+1)
```

Output:

Randomly distributed values for beta

```
array([0.79961881, 0.16464345, 0.25359076, 0.49451503, 0.40709532,
       0.86338983, 0.05701312])
```

Function name : RMSE
Parameter : Actual data and predicted data

This function returns the Root Mean Squared Error (RMSE) between the actual data and predicted data.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

```
def RMSE(y_data,y_pred):
    return np.sqrt(np.sum(pow((y_data - y_pred),2)))/len(y_pred)
```

Function name : linalg_prediction
Parameter : beta values and data

This function returns the dot product between the data and beta values which is the predicted data.

```
def linalg_prediction(beta,data):
    X = np.ones((data.shape[0],data.shape[1]+1))
    X[:,1:] = data
    return np.dot(beta,X.T)
```

Function name : armijo_steplength
Parameter : beta values, training data of X and Y, function and function gradient

This function is for providing the dynamic step length for generation a model with gradient descent. Initially, the step length is set to 1 and is halved for every iteration until the required condition is satisfied.

```
def armijo_steplength(beta,X_train,Y_train,func_gradient,func_x):
    alpha = 1
    delta = 0.15
    sq_gradient = np.dot(func_gradient,func_gradient)
    for i in range (0,1000):
        alpha = alpha/2
        Y_prediction = linalg_prediction(beta-alpha*func_gradient,X_train)
        residual_alpha_grad = Y_prediction - Y_train
        func_alpha_grad =np.dot(residual_alpha_grad.T,residual_alpha_grad)

        if func_x - func_alpha_grad >  alpha*delta*sq_gradient:
            break
    return alpha
```

Function name : bold_steplength
Parameter : alpha,beta values, training data of X and Y, function and function gradient

This function is also for generating dynamic step length for generating a model with gradient descent. Initially, the step length is set to 1 and the alpha plus and alpha minus are set randomly. The value of alpha is updated for every iteration until the difference between function and its gradient is greater than zero.

```
def bold_steplength(alpha,beta,X_train,Y_train,func_gradient,func_x):
    alphaplus = 1.9
    alphaminus = 0.9
    alpha = alpha * alphaplus
    sq_gradient = np.dot(func_gradient,func_gradient)
    for i in range (0,1000):
        alpha = alpha * alphaminus
        Y_prediction = linalg_prediction(beta-alpha*func_gradient,X_train)
        residual_alpha_grad = Y_prediction - Y_train
        func_alpha_grad =np.dot(residual_alpha_grad.T,residual_alpha_grad)

        if func_x - func_alpha_grad >  0:
            break
    return alpha
```

Function name : linalg_GD
Parameter : values of alpha, beta, training and testing data of X and Y, steplength_type

This function generates the model using linear regression with gradient descent with three different type of step lengths i.e., constant step length, Armijo step length and Bold Driver step length. The type of the step length to be used is given as a parameter to this function. The function predicts the price from the training data and also calculates the RMSE, difference between the functions for every iterations and appends the same into respective lists for plotting of graphs. It also checks if the function is converged or not for every iteration and updates the values of beta continuously until the maximum iteration is reached or the function is converged.

```
def linalg_GD(alpha,beta,Y_train,Y_test,X_train,X_test,steplength_type):
    rmse_graph = []
    iteration_graph = []
    diff_graph  = []
    Y_prediction = linalg_prediction(beta,X_train)
    residual = Y_prediction - Y_train
    func_x = np.dot(residual.T,residual)
    rmse = RMSE(Y_test,linalg_prediction(beta,X_test))
    rmse_graph.append(rmse)
    iteration_graph.append(0)
    diff_graph.append(func_x)

    X = np.ones((X_train.shape[0],X_train.shape[1]+1))
    X[:,1:] = X_train
    iterations = 1000
    for i in range (1,iterations):
        func_gradient = 2 * np.dot(X.T,residual)
        if steplength_type == "Armijo":
            alpha = armijo_steplength(beta,X_train,Y_train,
                                      func_gradient,func_x)
        if steplength_type == "Bold":
            alpha = bold_steplength(alpha,beta,X_train,Y_train,
                                    func_gradient,func_x)
        beta = beta - (alpha * func_gradient)
        Y_prediction = np.dot(beta,X.T)
        residual = Y_prediction - Y_train
        func_x_1 = np.dot(residual.T,residual)

        if math.isnan(abs(func_x_1 - func_x)):
            break

        y_pred_test = linalg_prediction(beta,X_test)
        rmse = RMSE(Y_test,y_pred_test)
        rmse_graph.append(rmse)
        iteration_graph.append(i)
        diff_graph.append(abs(func_x_1 - func_x))

        if abs(func_x_1 - func_x) < 1.1e-25:
            print("Function converged in "+ str(i) +" iterations")
            return beta,rmse_graph,iteration_graph,diff_graph,y_pred_test
        func_x = func_x_1
    return beta,rmse_graph,iteration_graph,diff_graph,y_pred_test
```

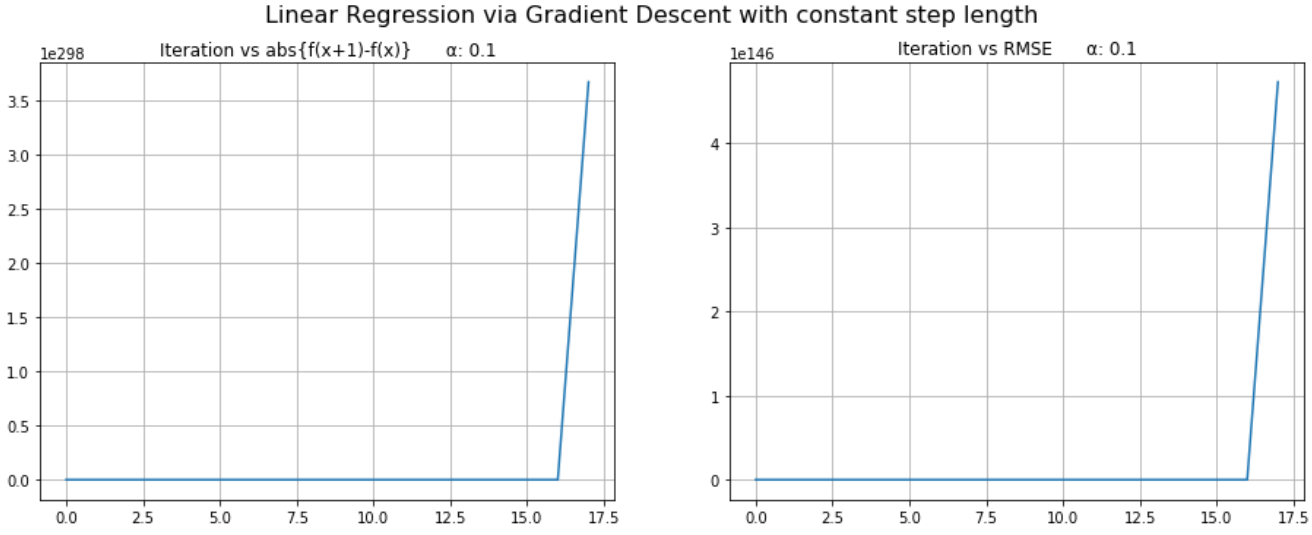
Gradient Descent with constant step length:

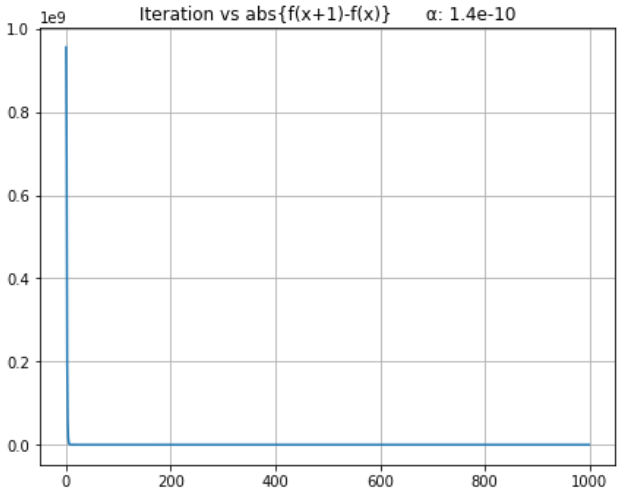
Now, we try to generate a model with three different constal values of step length and record the RMSE on test set and difference between the functions. In each iteration of the minimize-GD algorithm, these are recorded and plotted against the iterations

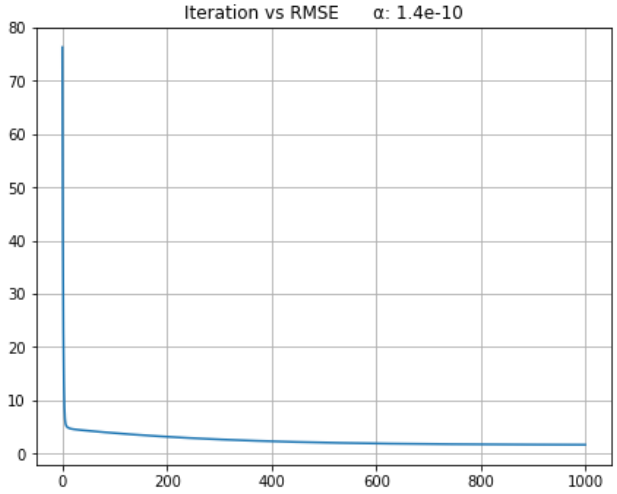
```
alpha = [0.1,1.4e-10,1.54e-11]
steplength_type = None
plot_rmse = []
plot_iter = []
plot_diff = []
beta1,rmse_graph,iteration_graph,diff_graph,y_pred_constant =
        linalg_GD(alpha[0],initialbeta,Y_train,Y_test,X_train,X_test,steplength_type)
figure, ((ax1, ax2))  = plt.subplots(1,2,sharey='none')
figure.set_size_inches(15.5, 5.5)
rmse1 = rmse_graph[-1]
ax1.plot(iteration_graph,diff_graph)
ax1.grid()
ax1.set_title("Iteration vs abs{f(x+1)-f(x)}      α: "+str(alpha[0]))
ax2.plot(iteration_graph,rmse_graph)
ax2.set_title("Iteration vs RMSE      α: "+str(alpha[0]))
ax2.grid()
plt.show
```

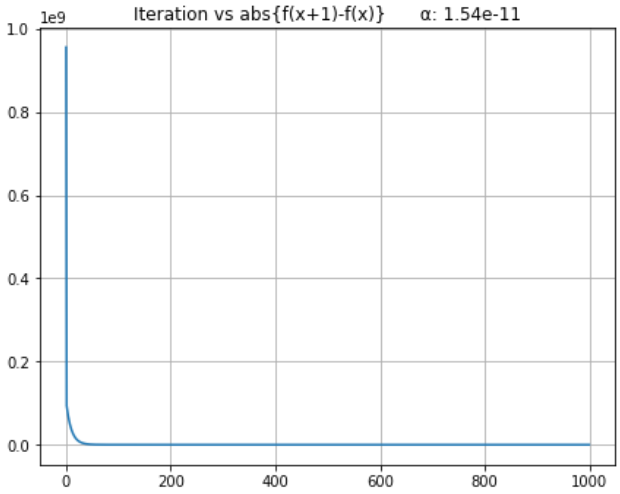
Output:

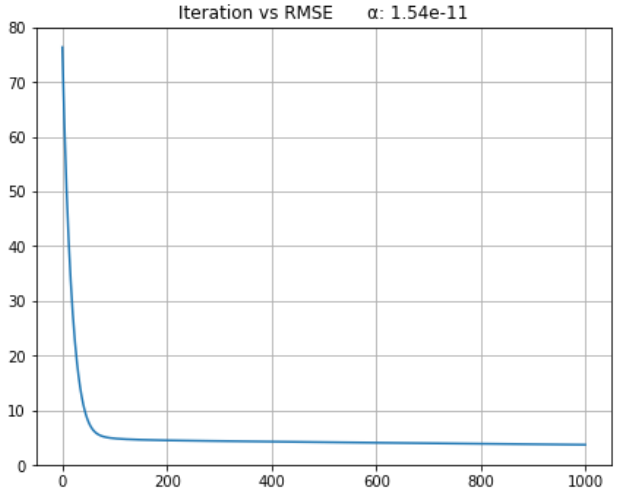
```
RMSE (α=0.1)  =  inf
RMSE (α=1.4e-10)  =  1.6473705064766335
RMSE (α=1.54e-11) =  3.7571897172243327


Best Model:
α = 1.4e-10
β =  [ 0.34003565  0.12649917  0.01098474 -0.00736119  0.17524091  0.5117166
  0.6774013 ]
RMSE 1.6473705064766335
```









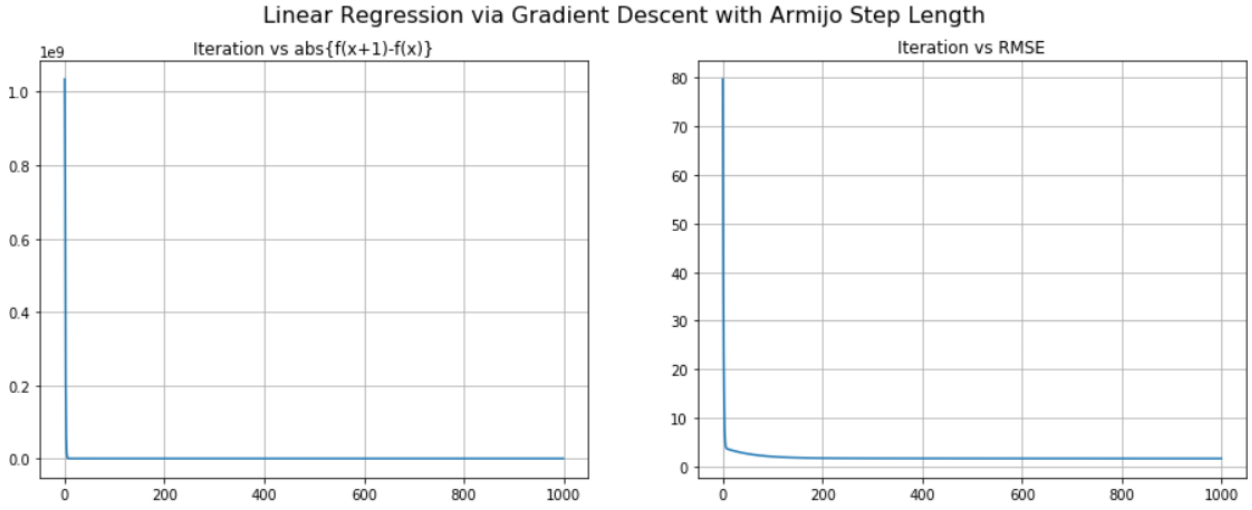


From the above results, it can be seen that for a constant step length of 0.1, the function diverges and hits the infinity. For a constant step length of 1.4e-10, a model is generated with the minimum error compared to other models. The model generated with constant step length of 1.54e-11 also shows considerable results but is poorly fit compared to the second model.

Gradient Descent with Armijo step length:

```
alpha = [1]
steplength_type = "Armijo"
plot_rmse_armijo = []
plot_iter_armijo = []
plot_diff_armijo = []
beta,rmse_graph_armijo,iteration_graph_armijo,diff_graph_armijo,Y_prediction =
        linalg_GD(alpha[0],initialbeta,Y_train,Y_test,X_train,X_test,steplength_type)
figure, ((ax1, ax2))  = plt.subplots(1,2,sharey='none')
figure.set_size_inches(15.5, 5.5)
ax1.plot(iteration_graph_armijo,diff_graph_armijo)
ax1.grid()
ax1.set_title("Iteration vs abs{f(x+1)-f(x)}")
ax2.plot(iteration_graph_armijo,rmse_graph_armijo)
ax2.set_title("Iteration vs RMSE")
ax2.grid()
plt.suptitle("Linear Regression via Gradient Descent with Armijo Step Length\n",fontsize = "16")
plt.show
```

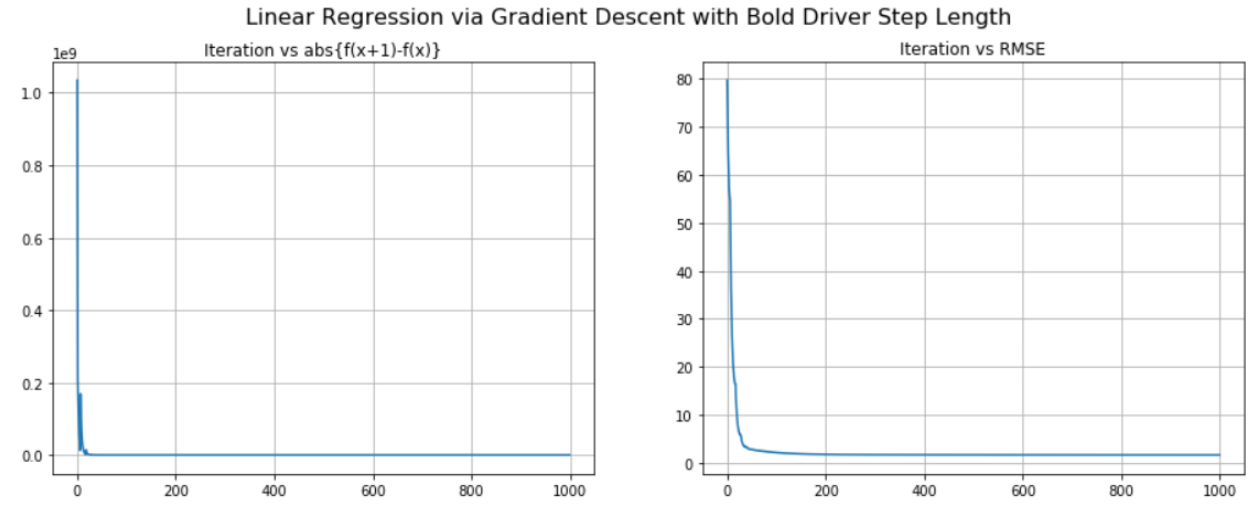
Output:



Gradient Descent with Bold Driver step length:

```
alpha = [1]
steplength_type = "Bold"
plot_rmse_bold = []
plot_iter_bold = []
plot_diff_bold = []
beta,rmse_graph_bold,iteration_graph_bold,diff_graph_bold,Y_prediction_bold =
        linalg_GD(alpha[0],initialbeta,Y_train,Y_test,X_train,X_test,steplength_type)
figure, ((ax1, ax2))  = plt.subplots(1,2,sharey='none')
plt.suptitle("Linear Regression via Gradient Descent with Bold Driver Step Length",fontsize = "16")
figure.set_size_inches(15.5, 5.5)
ax1.plot(iteration_graph_bold,diff_graph_bold)
ax1.grid()
ax1.set_title("Iteration vs abs{f(x+1)-f(x)}")
ax2.plot(iteration_graph_bold,rmse_graph_bold)
ax2.set_title("Iteration vs RMSE")
ax2.grid()
```

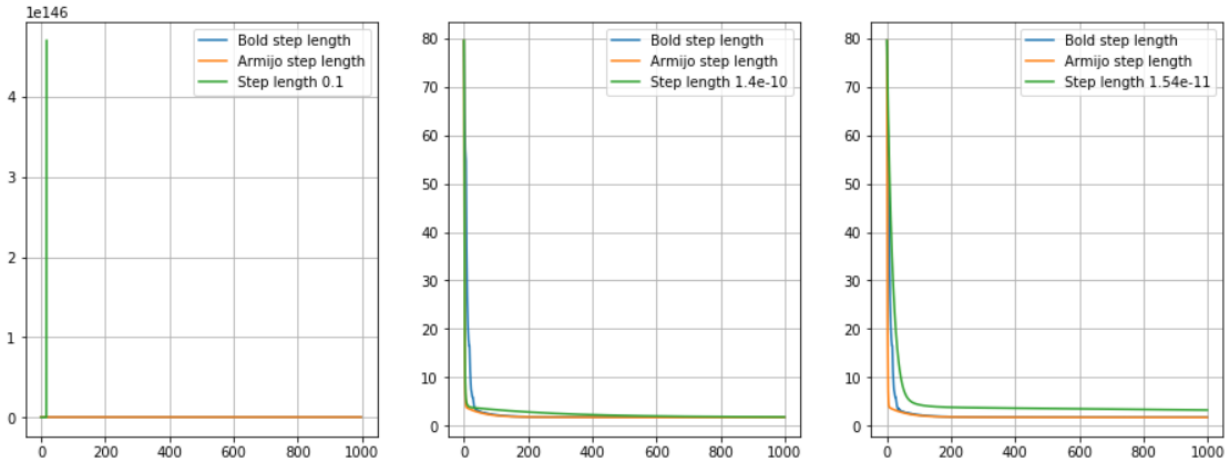
Output:



RMSE Comparision:

Comparing RMSE among Armijo, Bold driver and constant step length

```
RMSE with Armijo step length:  1.704129535373877
RMSE with Bold driver step length:  1.708548501392007
RMSE with constant step length:  1.7886323901940244
```



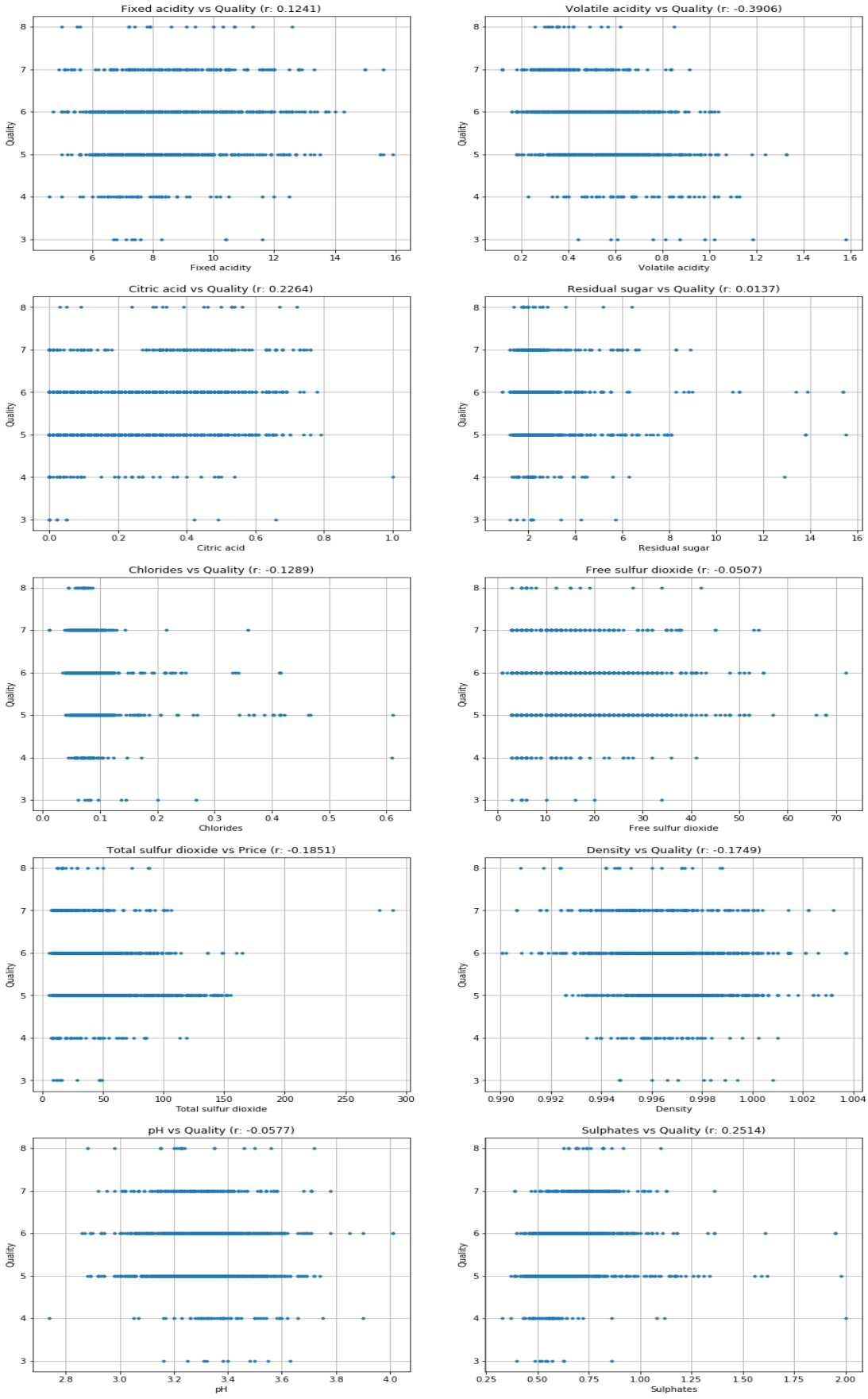
The above graphs compares the RMSE graphs of armijo step length, bold driver step length and the three fixed step lengths. It can be seen that the graph 2 with fixed step length provides the best fit and especially armijo step length method has the least root mean squared error.

Prediction Comparision:

	Actual Price	Armijo steplength	Bold steplength	Constant steplength
171	209.70	184.834419	184.260714	188.367050
863	138.98	126.232934	126.305823	131.922461
793	80.44	94.507795	93.206394	84.236574
905	211.30	181.054265	180.788436	181.289362
298	157.19	142.636958	142.137293	146.476085
188	86.79	97.926870	97.865172	100.324043
428	186.07	158.620723	158.850348	169.248914
7	174.00	177.066661	176.744429	171.929607
66	198.04	154.841883	155.018165	164.678515
843	78.85	93.725044	92.603583	87.733363
932	91.97	81.860153	81.484686	80.001361
597	231.52	207.515829	207.529209	215.919838
637	156.99	132.758242	132.566284	132.573010
467	153.88	159.251888	159.260556	165.058787
632	177.02	151.953695	151.741398	156.305932

As we compare the RMSE and predicted values among the minimize_GD algorithm with constant, Armijo and Bolt driver step length, we can see that the model generated by minimising the function with dynamic step length , Armijo step length, seems to be the best among the 3 types.

1.1.2 Red wine quality dataset:



From the plots between the parameters and price, we can conclude that there is no relation between residual sugar, free sulphur dioxide, pH and quality. Hence, I am dropping out these columns from the dataset to be used for creating the model. After creating the model, as it could be seen that there's poor correlation between fixed acidity, chlorides, total sulphur dioxide and quality. At last, these columns are also dropped down from the dataset.

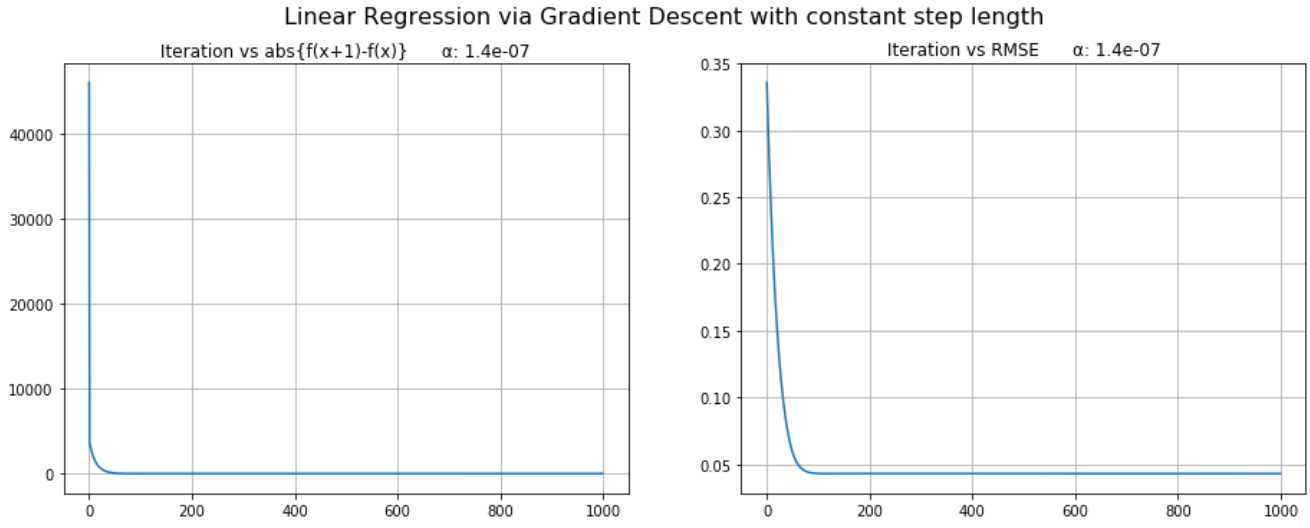
Dataset after dropping unwanted columns

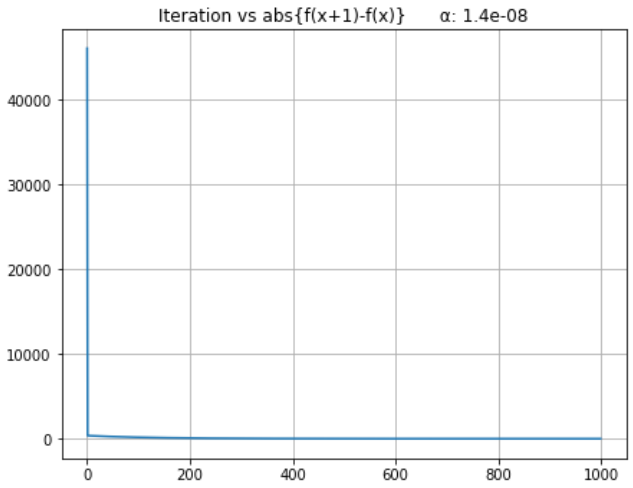
	volatile acidity	citric acid	density	sulphates	alcohol	quality
0	0.70	0.00	0.9978	0.56	9.4	5
1	0.88	0.00	0.9968	0.68	9.8	5
2	0.76	0.04	0.9970	0.65	9.8	5
3	0.28	0.56	0.9980	0.58	9.8	6
4	0.70	0.00	0.9978	0.56	9.4	5

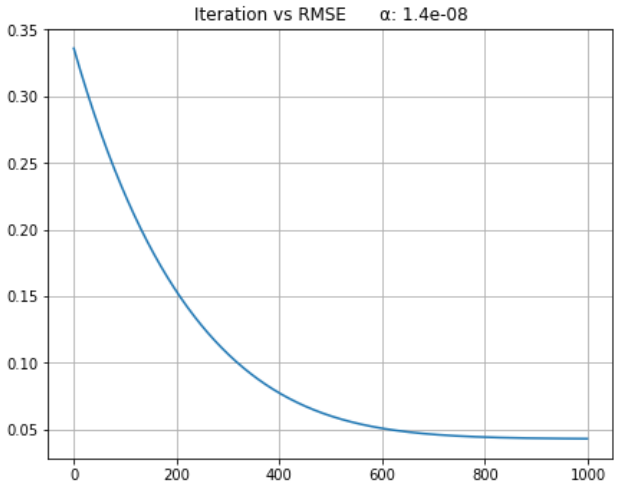
GD with constant step length:

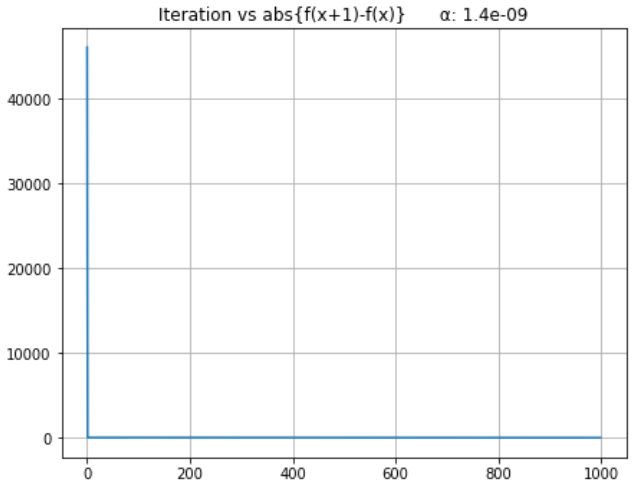
Now, we try to generate a model with three different constal values of step length and record the RMSE on test set and difference between the functions. In each iteration of the minimize-GD algorithm, these are recorded and plotted against the iterations

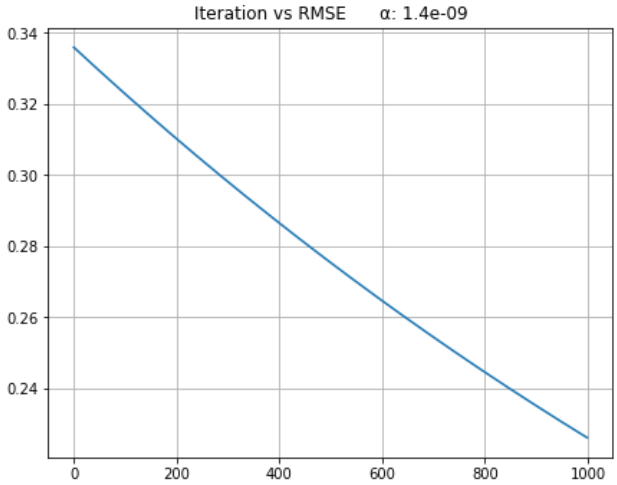
```
RMSE (α=1.4e-07)   =  0.04300215170748823
RMSE (α=1.4e-08)   =  0.043157746971550635
RMSE (α=1.54e-09) =  0.22612504203206632


Best Model:
α = 1.4e-07
β =  [0.63792344 0.771257   0.03420301 0.26599861 0.36539915 0.39004254]
RMSE 0.04300215170748823
```







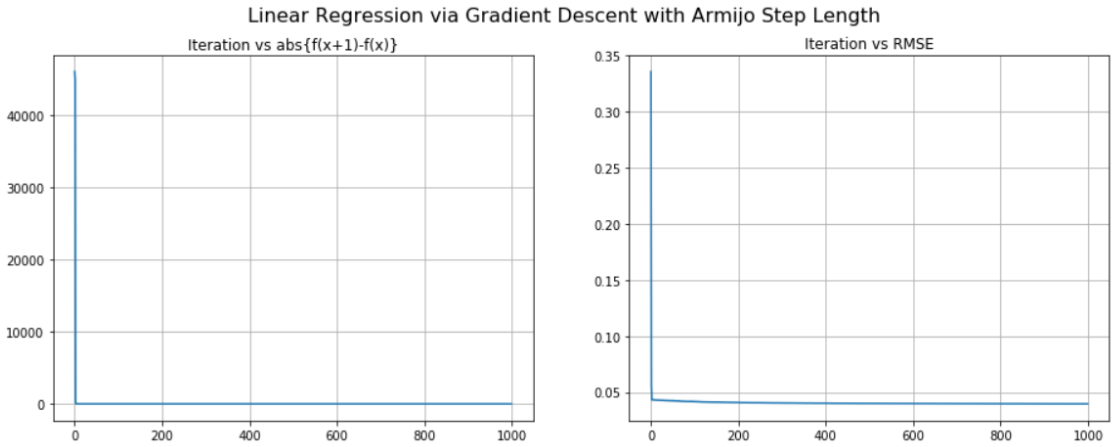




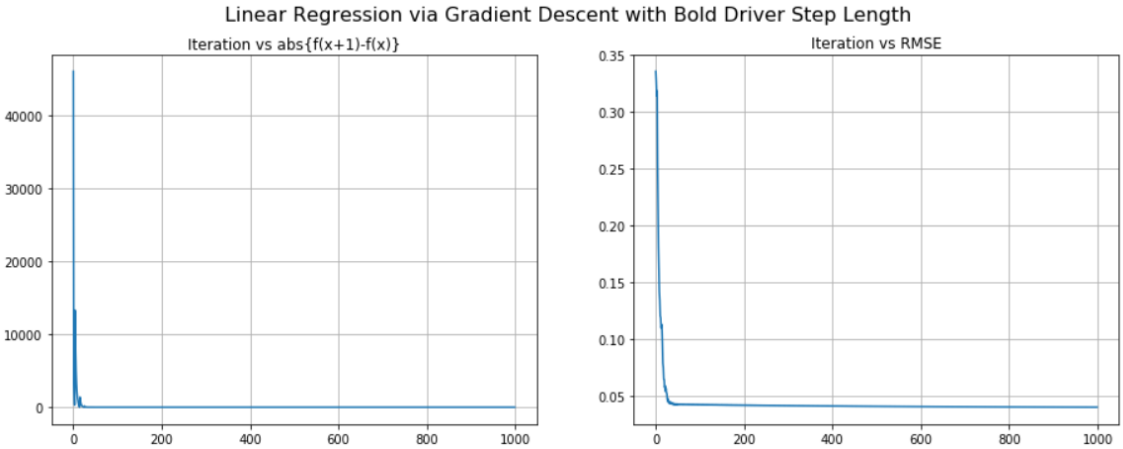
From the above results, it can be seen that for all the constant step lengths the function converges and provides similar RMSE for 1.4e-07 and 1.4e-08. But when we look at the RMSE value of 1.4e-09, it increases and as a result we can say that the minimum might have been over shooted. This makes a model to fit poorly. But, the model with step length 1.4e-07 is well fit and has an RMSE value of as low as 0.0430.

	Actual Price	Prediction (α=1.4e-07)	Prediction (α=1.4e-08)	Prediction (α=1.54e-09)
1232	5	5.249582	5.353750	9.155306
217	5	5.277620	5.384923	9.034300
476	5	5.253137	5.354623	9.233143
359	6	5.426789	5.529526	9.451296
1215	6	5.778755	5.897595	10.396387
67	5	5.771997	5.896635	10.168116
860	5	5.297853	5.408679	9.209924
251	6	5.435294	5.549121	9.508934
927	4	5.274574	5.381748	9.069574
979	5	5.619371	5.731655	9.885777
696	6	5.333070	5.443807	9.248377
615	5	5.476988	5.585414	9.468916
1467	4	6.102236	6.236773	10.632232
27	5	5.281175	5.383871	9.191009
1322	5	5.846836	5.968562	10.507187

GD with Armijo step length:

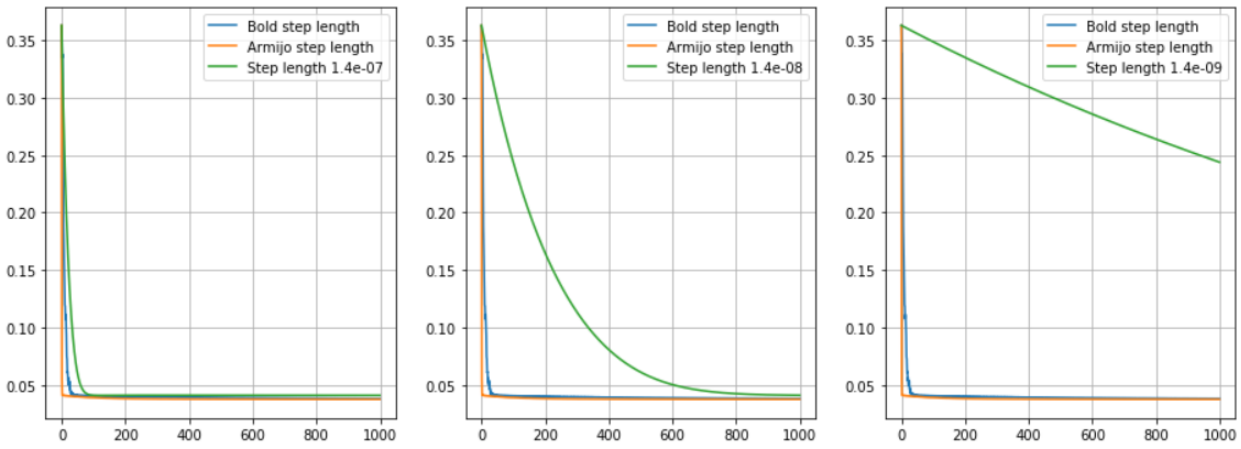


GD with Bold driven step length:



RMSE Comparision:

```
RMSE with Armijo step length:  0.037420796478959795
RMSE with Bold driver step length:  0.03809487833910137
RMSE with constant step length:  0.04085390666572421
```



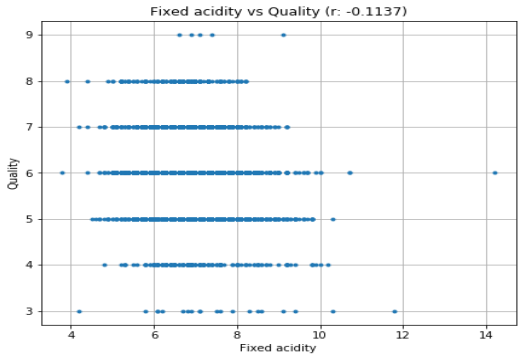
The above graphs compares the RMSE graphs of armijo step length, bold driver step length and the three fixed step lengths. It can be seen that the graph 1 with fixed step length of 1.4e-07 provides the best fit and especially armijo step length method has the least root mean squared error.

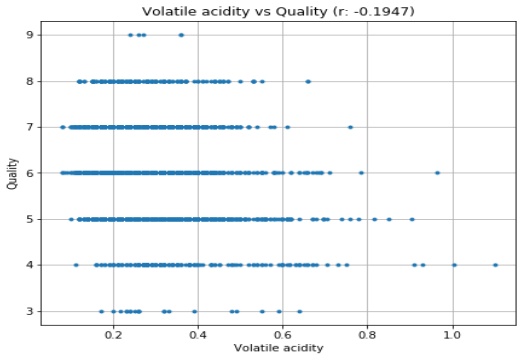
Prediction comparision:

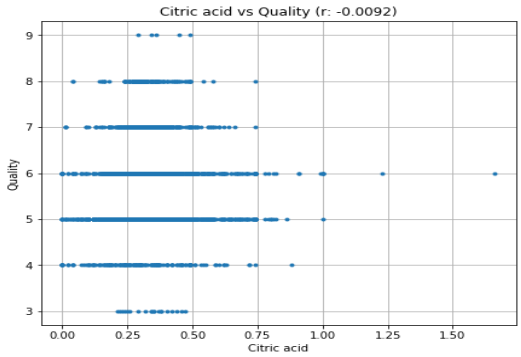
	Actual Price	Armijo steplength	Bold steplength	Constant steplength
1232	5	5.392650	5.300437	5.353750
217	5	4.922527	5.065975	5.384923
476	5	5.669386	5.468402	5.354623
359	6	5.649946	5.613470	5.529526
1215	6	6.089621	5.975225	5.897595
67	5	5.467097	5.566800	5.896635
860	5	5.101344	5.126610	5.408679
251	6	5.376972	5.321633	5.549121
927	4	5.030307	5.110045	5.381748
979	5	5.709453	5.706658	5.731655
696	6	5.180412	5.173936	5.443807
615	5	5.304922	5.431429	5.585414
1467	4	5.284289	5.657059	6.236773
27	5	5.608928	5.402875	5.383871
1322	5	6.047205	5.981427	5.968562

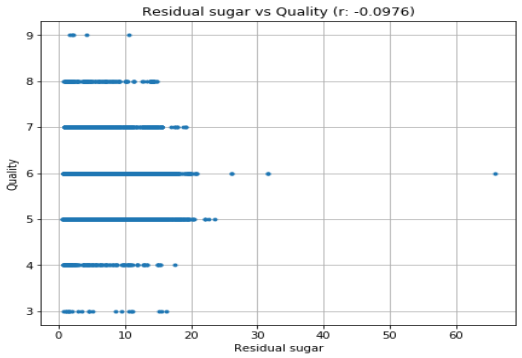
As we compare the RMSE and predicted values among the minimize_GD algorithm with constant, Armijo and Bolt driver step length, we can see that the model generated by minimising the function with dynamic step length , Armijo step length, seems to be the best among the 3 types with error as low as 0.0374.

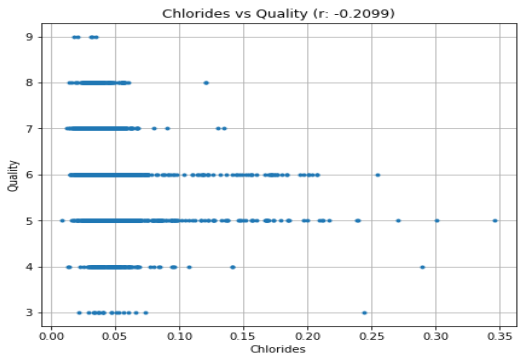
1.1.2 White wine quality dataset:

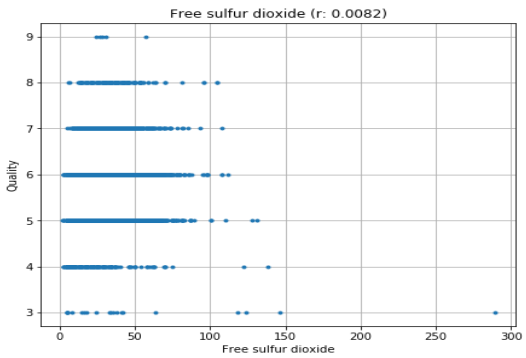


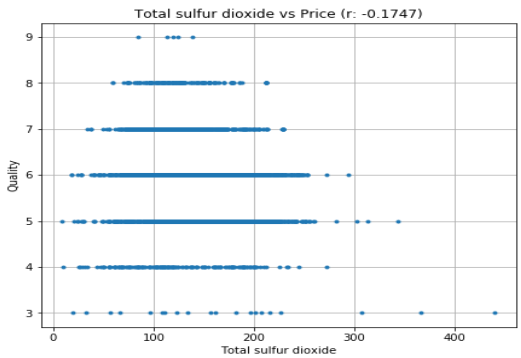


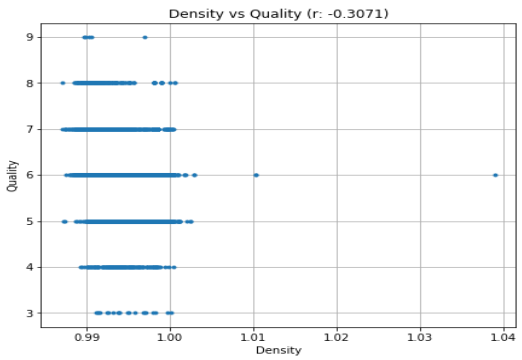


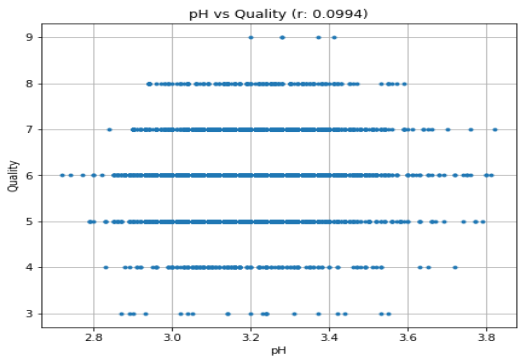


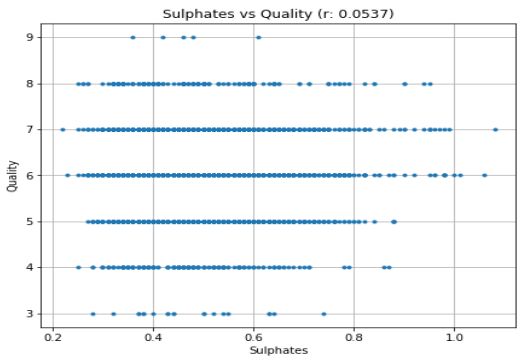












From the plots between the parameters and price, we can conclude that there is no relation between citric acid, residual sugar, free sulphur dioxide, pH, sulphates and quality. Hence, I am dropping out these columns from the dataset to be used for creating the model. After creating the model, as it could be seen that there's poor correlation between total sulphur dioxide and quality. At last, these columns are also dropped down from the dataset.

Dataset after dropping unwanted columns

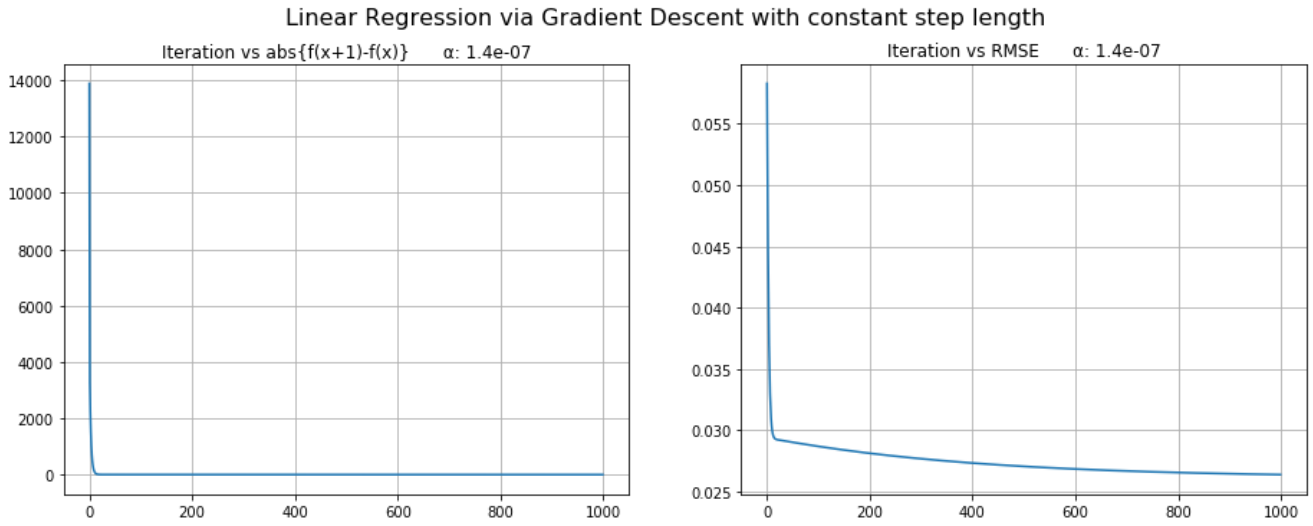
	fixed acidity	volatile acidity	chlorides	density	alcohol	quality
0	7.0	0.27	0.045	1.0010	8.8	6
1	6.3	0.30	0.049	0.9940	9.5	6
2	8.1	0.28	0.050	0.9951	10.1	6
3	7.2	0.23	0.058	0.9956	9.9	6
4	7.2	0.23	0.058	0.9956	9.9	6

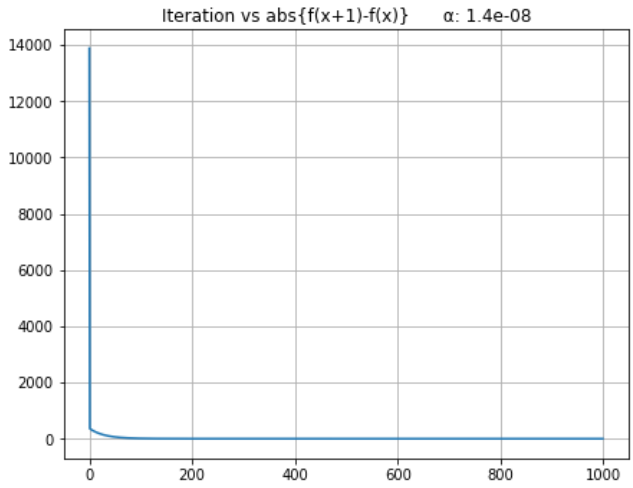
GD with constant step length:

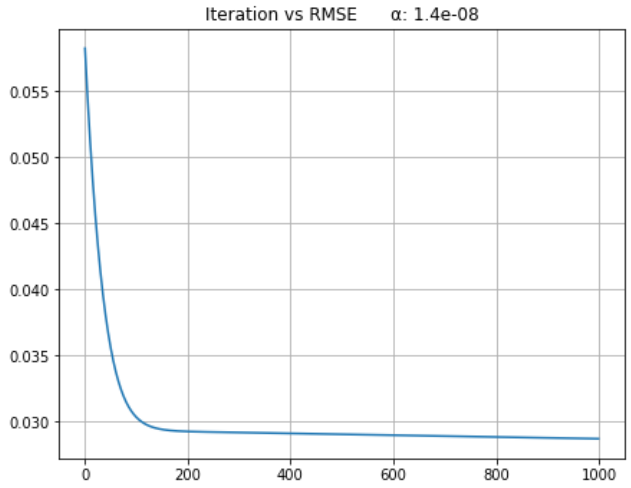
Now, we try to generate a model with three different constal values of step length and record the RMSE on test set and difference between the functions. In each iteration of the minimize-GD algorithm, these are recorded and plotted against the iterations

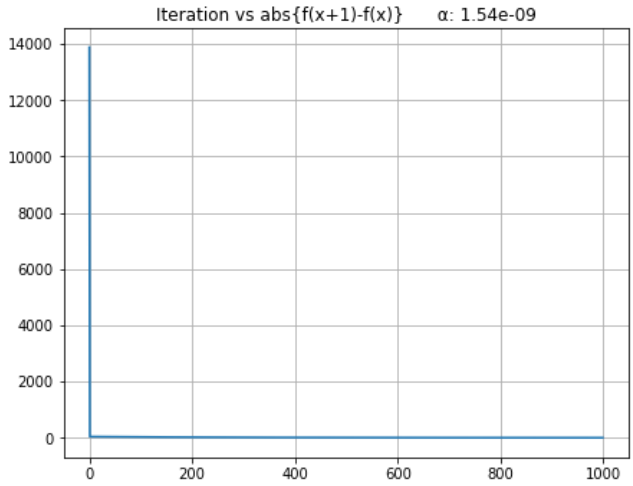
```
RMSE (α=1.4e-07)  =  0.026392359248040252
RMSE (α=1.4e-08)  =  0.028695348168449375
RMSE (α=1.54e-09) =  0.030012510014148678
```

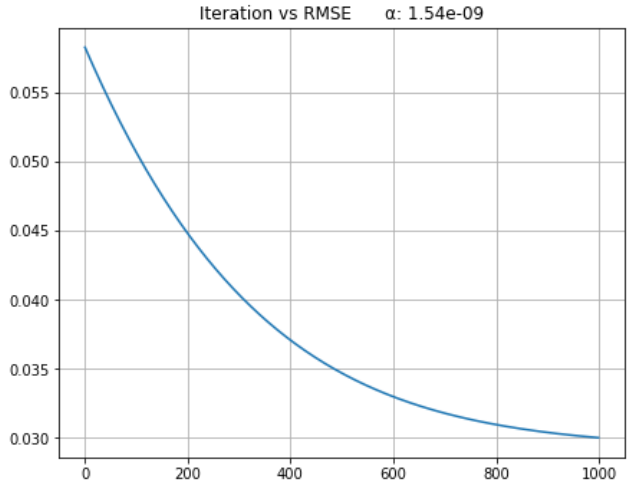
```
Best Model:
α = 1.4e-07
β =  [0.30537145 0.22321859 0.76801112 0.24087882 0.01701348 0.35989537]
RMSE 0.026392359248040252
```







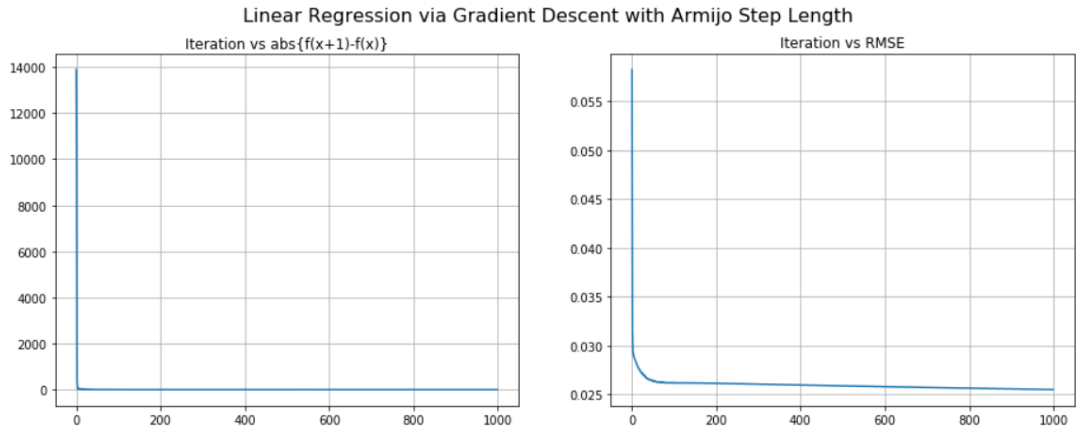




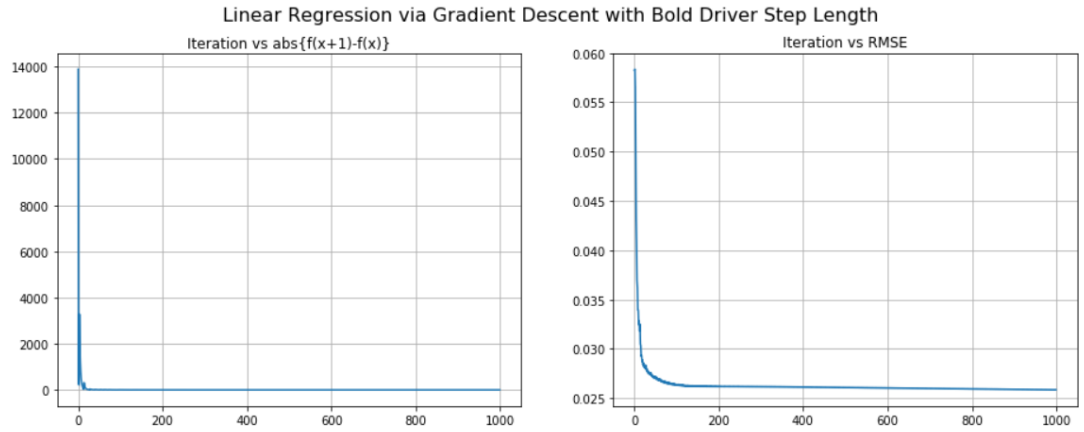
From the above results, it can be seen that for all the constant step lengths the function converges and provides similar RMSE for all the step lengths.The model with step length 1.4e-07 is well fit and has an RMSE value of as low as 0.0430.

	Actual Price	Prediction (α=1.4e-07)	Prediction (α=1.4e-08)	Prediction (α=1.54e-09)
2421	5	5.469747	5.611488	5.848779
4476	6	6.393967	5.895867	6.065581
4120	5	5.440061	5.788060	6.058426
711	6	6.795141	6.847203	7.124954
2929	6	5.250690	5.404343	5.635749
1390	6	5.428345	5.281337	5.470287
1804	6	5.608022	5.958014	6.235133
3832	5	5.553947	5.682554	5.919066
2046	5	5.786620	6.293540	6.603881
2539	5	5.519270	5.455468	5.659756
3589	6	5.415280	5.555388	5.792488
2058	5	5.424203	5.750113	6.015768
2105	5	5.373047	5.573689	5.814833
1843	6	6.435135	6.667834	6.960510
3065	6	5.537888	5.678600	5.922292

GD with Armijo step length:

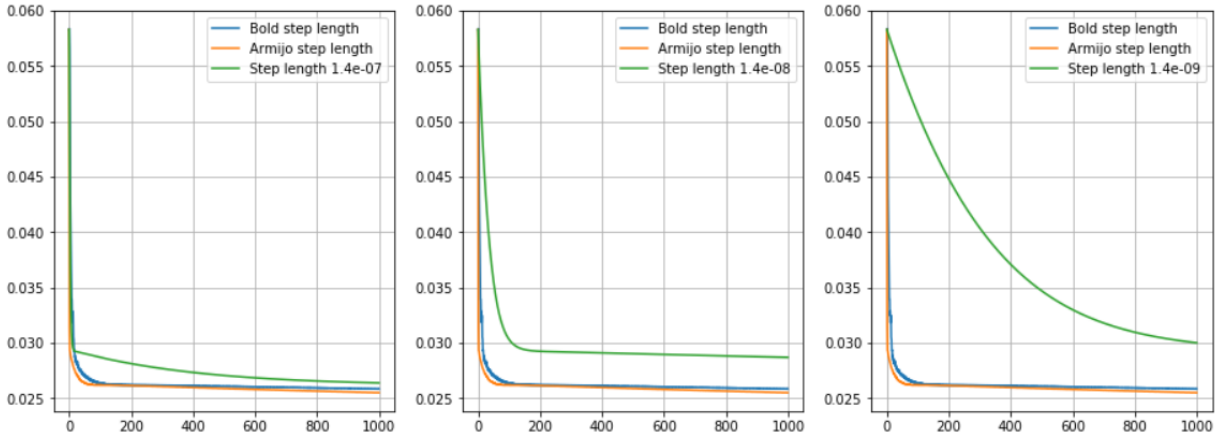


GD with Bold driven step length:



RMSE comparision:

```
RMSE with Armijo step length:  0.025510080964902298
RMSE with Bold driver step length:  0.025862259542041136
RMSE with constant step length:  0.026392359248040252
```



The above graphs compares the RMSE graphs of armijo step length, bold driver step length and the three fixed step lengths. Althogh the RMSE value is similar for all the three constant step lengths, it can be seen that the graph 1 with fixed step length of 1.4e-07 provides the best fit and especially armijo step length method has the least root mean squared error.

Prediction comparision:

	Actual Price	Armijo steplength	Bold steplength	Constant steplength
2421	5	5.431822	5.406717	5.611488
4476	6	6.676916	6.659116	5.895867
4120	5	5.272535	5.256746	5.788060
711	6	6.681368	6.711307	6.847203
2929	6	5.241428	5.200315	5.404343
1390	6	5.594609	5.545883	5.281337
1804	6	5.415867	5.410794	5.958014
3832	5	5.497629	5.484384	5.682554
2046	5	5.471692	5.484619	6.293540
2539	5	5.595782	5.568286	5.455468
3589	6	5.409594	5.369759	5.555388
2058	5	5.267496	5.252205	5.750113
2105	5	5.285877	5.269475	5.573689
1843	6	6.244363	6.266103	6.667834
3065	6	5.535111	5.493413	5.678600

As we compare the RMSE and predicted values among the minimize_GD algorithm with constant, Armijo and Bolt driver step length, we can see that the model generated by minimising the function with dynamic step length , Armijo step length, seems to be the best among the 3 types with error as low as 0.0255.