MACHINE LEARNING LAB - TUTORIAL 3

Juan Fernando Espinosa

303158

1. DATA PROCESSSING

IMPORT DATASETS

```
import pandas as pd
import numpy as np
%matplotlib inline
import math
import matplotlib.pyplot as plt
from google.colab import files
from google.colab import drive
```

```
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/airq402.data"
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/parkinsons_updrs.data"
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/winequality-red.csv"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/airq402.data'
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/parkinsons_updrs.data'
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/winequality-red.csv'
```

> To ease the cleaning step of the dataset I created an array with all possible incongruent fields to transform into NaN strings for future dropping. Dataframes are created.

```
column_names_airq402 = ['City_1','City_2','Average Fare','Distance','Average weekly passengers','market leading airline','market sha
missing_values = ['-','na','Nan','nan','n/a','?']
airq402 = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/airq402.data',delim_whitespace=True, names=column_name
parkinsons_updrs = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/parkinsons_updrs.data', sep=',', na_values =
winequality_red = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 3/winequality-red.csv', sep=';', na_values = mis
```

AIRQ402 Dataset

> Search for missing/incongruent values.

```
airq402.dtypes
print(airq402.count())
airq402.head()
```

```
City_1                        1000
City_2                        1000
Average Fare                  1000
Distance                      1000
Average weekly passengers     1000
market leading airline        1000
market share                  1000
Average fare2                 1000
Low price airline             1000
market share2                 1000
price                         1000
dtype: int64
```

	City_1	City_2	Average Fare	Distance	Average weekly passengers	market leading airline	market share	Average fare2	Low price airline	market 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

```
check = airq402.empty
airq402["City_1"].value_counts()
print('checking missing values:',check)
print('Sum of errors:',airq402.isnull().sum())
```

```
checking missing values: False
Sum of errors: City_1                       0
City_2                        0
Average Fare                  0
Distance                      0
Average weekly passengers     0
market leading airline        0
market share                  0
Average fare2                 0
Low price airline             0
market share2                 0
price                         0
dtype: int64
```

> Encode it to transform all the non-values columns into numerical values.

```
dummy_airq402 = pd.get_dummies(airq402)
dummy_airq402.head()
```

	Average Fare	Distance	Average weekly passengers	market share	Average fare2	market share2	price	City_1_ABQ	City_1_ACY	City_1_ALB	City_1_AMA	City_1_ATL	City
0	114.47	528	424.56	70.19	111.03	70.19	111.03	0	0	0	0	0	
1	122.47	860	276.84	75.10	123.09	17.23	118.94	0	0	0	0	0	
2	214.42	852	215.76	78.89	223.98	2.77	167.12	0	0	1	0	0	
3	69.40	288	606.84	96.97	68.86	96.97	68.86	0	0	1	0	0	
4	158.13	723	313.04	39.79	161.36	15.34	145.42	0	0	1	0	0	

5 rows × 217 columns

> Check if there are columns that don't represent any advantage in the dataset

```
airq402_pearsonCorr = airq402.corr(method='pearson')
airq402_pearsonCorr['price']
```

```
Average Fare                  0.866410
Distance                      0.583239
Average weekly passengers    -0.142314
market share                 -0.307672
Average fare2                 0.826511
market share2                -0.240186
price                         1.000000
Name: price, dtype: float64
```

Observations:

- The market share2 column does not impact the price neither because the popularity of a flight in a general statement does not increase or decrease in comparison to the brand and it has the market share column which address in a better way the relationship.
- Average Fare columns has the highest correlation. Therefore, the one which best address the relationship is going to be selected: **Average Fare**. The output measuring this Average Fare returns a good overview of the second Average Fare column.
- Cities columns does not add value in measuring the final price even though they are crucial with a final output to find the most effective, expensive, used, cheaper, and so more examples of flights and routes. In other words, the best output of them would be "grouping by" cities category".
- Finally, same effect with cities are experienced by airlines and market share dominant companies. Encoding the categories are helpful to know the impact of a low-price, mid-price and high-price airline but for the purpose of this exercise they turn out to be irrelevant.

```
airq402_new=airq402.drop(columns=['Average fare2', 'market share2', 'City_1','City_2', 'Low price airline', 'market leading airline'
airq402_new.head()
```

	Average Fare	Distance	Average weekly passengers	market share	price
0	114.47	528	424.56	70.19	111.03
1	122.47	860	276.84	75.10	118.94
2	214.42	852	215.76	78.89	167.12
3	69.40	288	606.84	96.97	68.86
4	158.13	723	313.04	39.79	145.42

> It is important to scale all the results of the dataset to make more accurate studies.

```
def scale(value):
    new = (value-value.min())/(value.max()-value.min())
    return new
```

```
airq402_new_scale = airq402_new.copy()

airq402_new_scale['Average weekly passengers'] = scale(airq402_new_scale['Average weekly passengers'])
airq402_new_scale['market share'] = scale(airq402_new_scale['market share'])
airq402_new_scale['Average Fare'] = scale(airq402_new_scale['Average Fare'])
airq402_new_scale['price'] = scale(airq402_new_scale['price'])
airq402_new_scale['Distance'] = scale(airq402_new_scale['Distance'])
airq402_new_scale.head()
```

	Average Fare	Distance	Average weekly passengers	market share	price
0	0.182344	0.160550	0.027727	0.637877	0.181539
1	0.205155	0.287462	0.010882	0.697522	0.204918
2	0.467338	0.284404	0.003917	0.743562	0.347324
3	0.053834	0.068807	0.048513	0.963192	0.056897
4	0.306835	0.235092	0.015010	0.268586	0.283185

> Split the dataset in **Train** and **Test** set.

```
airq402_new_train = airq402_new_scale.sample(frac=0.8)
airq402_new_test = airq402_new_scale.drop(airq402_new_train.index)
```

Parkinsons updrs

> This dataset does not contain any categorical column, therefore, no encodification needed. In addition, no missing values found.

> Proceeding to identify the important columns in the dataset.

```
parkinsons_updrs.head()
check = parkinsons_updrs.empty
print('checking missing values:',check)
print('Sum of errors:',parkinsons_updrs.isnull().sum())
parkinsons_updrs.head()
```

```
checking missing values: False
Sum of errors: subject#        0
age              0
sex              0
test_time        0
motor_UPDRS      0
total_UPDRS      0
Jitter(%)        0
Jitter(Abs)      0
Jitter:RAP       0
Jitter:PPQ5      0
Jitter:DDP       0
Shimmer          0
Shimmer(dB)      0
Shimmer:APQ3     0
Shimmer:APQ5     0
Shimmer:APQ11    0
Shimmer:DDA      0
NHR              0
HNR              0
RPDE             0
DFA              0
PPE              0
dtype: int64
```

	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PPQ5	Jitter:DDP	Shimmer
0	1	72	0	5.6431	28.199	34.398	0.00662	0.000034	0.00401	0.00317	0.01204	0.02565
1	1	72	0	12.6660	28.447	34.894	0.00300	0.000017	0.00132	0.00150	0.00395	0.02024
2	1	72	0	19.6810	28.695	35.389	0.00481	0.000025	0.00205	0.00208	0.00616	0.01675
3	1	72	0	25.6470	28.905	35.810	0.00528	0.000027	0.00191	0.00264	0.00573	0.02309
4	1	72	0	33.6420	29.187	36.375	0.00335	0.000020	0.00093	0.00130	0.00278	0.01703

```
Parkinson_pearsonCorr = parkinsons_updrs.corr(method='pearson')
Parkinson_pearsonCorr['total_UPDRS']
```

```
subject#         0.253643
age              0.310290
sex             -0.096559
test_time        0.075263
motor_UPDRS      0.947231
total_UPDRS      1.000000
Jitter(%)        0.074247
Jitter(Abs)      0.066927
Jitter:RAP       0.064015
Jitter:PPQ5      0.063352
Jitter:DDP       0.064027
Shimmer          0.092141
Shimmer(dB)      0.098790
Shimmer:APQ3     0.079363
Shimmer:APQ5     0.083467
Shimmer:APQ11    0.120838
Shimmer:DDA      0.079363
NHR              0.060952
HNR             -0.162117
RPDE             0.156897
DFA             -0.113475
PPE              0.156195
Name: total_UPDRS, dtype: float64
```

Observations: Considering a Parkinson study, the total UPDRS bear into account mental state, behavior, mood, motor examination of a patient, daily activities and thepary complications. Therefore, only the column *test_time* must be dropped.

- All the Jitter columns has pretty similar correlations. Therefore, taking into account one of them will give a good perspective of the impact of all of them. In consequence, the column **Jitter(Abs)** will remain.
- Similar effect is experienced with Shimmer columns. From the following columns: Shimmer, Shimmer(dB), Shimmer:APQ3, Shimmer:APQ5, Shimmer:APQ11, and Shimmer:DDA, the remaining columns are going to be: **Shimmer:APQ11** and **Shimmer:APQ5.**

> Perhaps, someone could say that Sex and Age are not relevant. Especially in this case, both give insights of the illness.

```
parkinsons_updrs_clean = parkinsons_updrs.drop(columns=['test_time','Shimmer', 'Shimmer(dB)', 'Shimmer:APQ3', 'Shimmer:DDA', 'Jitter
parkinsons_updrs_clean.head()
```

	subject#	age	sex	motor_UPDRS	total_UPDRS	Jitter(Abs)	Shimmer:APQ5	Shimmer:APQ11	NHR	HNR	RPDE	DFA	PPE
0	1	72	0	28.199	34.398	0.000034	0.01309	0.01662	0.014290	21.640	0.41888	0.54842	0.16006
1	1	72	0	28.447	34.894	0.000017	0.01072	0.01689	0.011112	27.183	0.43493	0.56477	0.10810
2	1	72	0	28.695	35.389	0.000025	0.00844	0.01458	0.020220	23.047	0.46222	0.54405	0.21014
3	1	72	0	28.905	35.810	0.000027	0.01265	0.01963	0.027837	24.445	0.48730	0.57794	0.33277
4	1	72	0	29.187	36.375	0.000020	0.00929	0.01819	0.011625	26.126	0.47188	0.56122	0.19361

> In order to avoid large values I am going to scale all the values.

```
parkinsons_updrs_clean_scale = parkinsons_updrs_clean.copy()

parkinsons_updrs_clean_scale['age'] = scale(parkinsons_updrs_clean_scale['age'])
parkinsons_updrs_clean_scale['sex'] = scale(parkinsons_updrs_clean_scale['sex'])
parkinsons_updrs_clean_scale['motor_UPDRS'] = scale(parkinsons_updrs_clean_scale['motor_UPDRS'])
parkinsons_updrs_clean_scale['Jitter(Abs)'] = scale(parkinsons_updrs_clean_scale['Jitter(Abs)'])
parkinsons_updrs_clean_scale['Shimmer:APQ5'] = scale(parkinsons_updrs_clean_scale['Shimmer:APQ5'])
parkinsons_updrs_clean_scale['Shimmer:APQ11'] = scale(parkinsons_updrs_clean_scale['Shimmer:APQ11'])
parkinsons_updrs_clean_scale['NHR'] = scale(parkinsons_updrs_clean_scale['NHR'])
parkinsons_updrs_clean_scale['HNR'] = scale(parkinsons_updrs_clean_scale['HNR'])
parkinsons_updrs_clean_scale['RPDE'] = scale(parkinsons_updrs_clean_scale['RPDE'])
parkinsons_updrs_clean_scale['DFA'] = scale(parkinsons_updrs_clean_scale['DFA'])
parkinsons_updrs_clean_scale['subject#'] = scale(parkinsons_updrs_clean_scale['subject#'])

parkinsons_updrs_clean_scale.head()
```

	subject#	age	sex	motor_UPDRS	total_UPDRS	Jitter(Abs)	Shimmer:APQ5	Shimmer:APQ11	NHR	HNR	RPDE	DFA
0	0.0	0.734694	0.0	0.671862	34.398	0.071164	0.067543	0.051764	0.018723	0.551717	0.328638	0.097793
1	0.0	0.734694	0.0	0.679056	34.894	0.032819	0.053186	0.052753	0.014474	0.704771	0.348330	0.144300
2	0.0	0.734694	0.0	0.686250	35.389	0.050458	0.039375	0.044291	0.026651	0.590568	0.381812	0.085362
3	0.0	0.734694	0.0	0.692342	35.810	0.054856	0.064878	0.062791	0.036834	0.629169	0.412583	0.181761
4	0.0	0.734694	0.0	0.700522	36.375	0.040353	0.044524	0.057515	0.015160	0.675585	0.393664	0.134202

> Split the dataset in **Train** and **Test** set.

```
parkinsons_updrs_clean_train = parkinsons_updrs_clean_scale.sample(frac=0.8)
parkinsons_updrs_clean_test = parkinsons_updrs_clean_scale.drop(parkinsons_updrs_clean_train.index)
```

Red Wine quality

```
winequality_red.head()
```

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

This dataset does not contain any categorical column, therefore, no encodification needed.

> Proceeding to check missing values.

```
check = winequality_red.empty
print('checking missing values:',check)
print('Sum of errors:',winequality_red.isnull().sum())
winequality_red.head()
```

```
checking missing values: False
Sum of errors: fixed acidity          0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

	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

No errors found in the dataset.

> Proceeding to identify the relation between categories. Considering **Quality** as the final result of all the information in the columns the relation will be hold onto quality.

```
winequality_red_pearsonCorr = winequality_red.corr(method='pearson')
winequality_red_pearsonCorr['quality']
```

```
fixed acidity           0.124052
volatile acidity       -0.390558
citric acid             0.226373
residual sugar          0.013732
chlorides              -0.128907
free sulfur dioxide    -0.050656
total sulfur dioxide   -0.185100
density                -0.174919
pH                     -0.057731
sulphates               0.251397
alcohol                 0.476166
quality                 1.000000
Name: quality, dtype: float64
```

Observations: Residual sugar, free sulfur dioxide and pH values has a minimum impact on the final quality of a red wine. Therefore excluding from our calculus will affect in no ways the final result.

```
winequality_red.drop(columns=['residual sugar', 'pH', 'free sulfur dioxide'])
winequality_red.head()
```

	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

Scale the info in order to make it easier to run the algorithm

```
winequality_red_normalized = scale(winequality_red)
winequality_red_normalized.head()
winequality_red_normalized1=(winequality_red-winequality_red.mean())/winequality_red.std()
```

> Split the dataset in **Train** and **Test** set

```
winequality_red_normalized_train = winequality_red_normalized.sample(frac=0.8)
winequality_red_normalized_test = winequality_red_normalized.drop(winequality_red_normalized_train.index)
winequality_red_normalized_train.values
```

```
winequality_red_normalized_test.values
```

```
array([[0.28318584, 0.52054795, 0.        , ..., 0.20958084, 0.21538462,
         0.4       ],
       [0.18584071, 0.31506849, 0.08      , ..., 0.1257485 , 0.12307692,
         0.4       ],
       [0.2920354 , 0.1369863 , 0.51      , ..., 0.4491018 , 0.12307692,
         0.6       ],
       ...,
       [0.14159292, 0.30136986, 0.09      , ..., 0.16167665, 0.44615385,
         0.4       ],
       [0.2300885 , 0.36986301, 0.33      , ..., 0.26946108, 0.67692308,
         0.6       ],
       [0.11504425, 0.29452055, 0.1       , ..., 0.25748503, 0.43076923,
         0.6       ]])
```

1.2 LINEAR REGRESSION WITH GRADIENT DESCENT

In this section to keep an order I will organize by dataset.

> As a starting point, the main code of the algorithm is presented. In sections below the code will be called by a print function to retrieve results for the different datasets.

```
# Minimization of the gradient descent
def minimize_GD(X, y, u, num_iters, e, beta,n):
  beta_old = beta
  loss_decrease = []
  sum_RMSE = 0
  RMSE_plot = []
  for i in range(num_iters):
    y_hat = np.dot(X.T,beta_old)
    loss = sum((y_hat - y)**2)
    # Measure the values of the new betas.
    Beta_new = beta_old - u*((-2)*X@(y - X.T@beta_old))
    # Call of the function for loss calculation
    loss_calculation = function(y, X, Beta_new, beta_old)
    loss_decrease.append(loss_calculation)
    RMSE = RMSE_function(X, y, y_hat)
    sum_RMSE+=RMSE
    RMSE_plot.append(RMSE)
    beta_old = Beta_new
  return Beta_new, loss_decrease, sum_RMSE, RMSE_plot


def learn_linreg_GD(X, y, u, num_iters,e):
    X = X
    y = Y
    n = X.shape[0]
    beta = np.zeros(n)
    beta = np.reshape(beta, (len(beta),1))
    beta_hat, loss, RMSE, RMSE_plot = minimize_GD(X, y, u, num_iters, e, beta, n)
    return  beta_hat,loss, RMSE, RMSE_plot

# Function for the Loss.
def function(y, X, Beta_new, beta_old):
  a = abs(np.sum((y - X.T@beta_old)**2) - np.sum((y - X.T@Beta_new)**2))
  return a

#Function for the RMSE.
def RMSE_function(X, y, y_hat):
  a = np.sqrt(sum((y - y_hat)**2)/y.shape[0])
  return a
```

AIRQ402 DataSet

> First, we train the algorithm by using the 80% of the data.

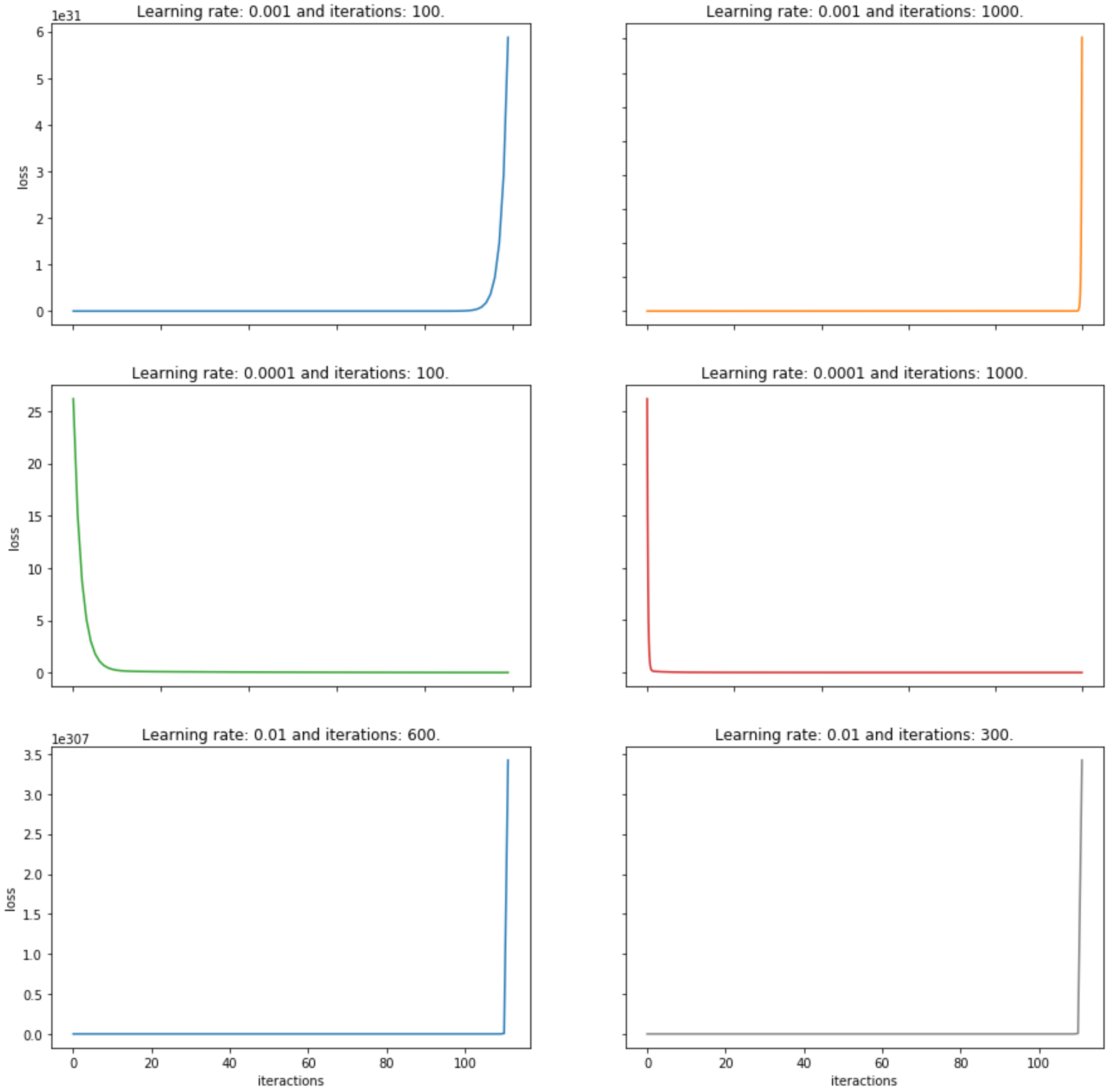
```
Y = airq402_new_train['price'].values
Y = np.reshape(Y, (len(Y),1))
X = airq402_new_train.drop(['price'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[ 0.14193581]
 [ 0.2346373 ]
 [ 0.18218075]
 [-0.0124996 ]
 [-0.01324309]]
 Loss [26.171357886126486, 15.11839249155981, 8.761055642659109, 5.103760816228451, 2.999044102742932, 1.787112140067201, 1.0885
 RMSE [10.91474768]
```

> Next, a graph showing us the behavior of the information are required to establish an optimum number of iterations and learning rate.

```
a, b, c, c1 = learn_linreg_GD(X.T, Y, 0.001, 100, 0.5)
d, e, f, f1 = learn_linreg_GD(X.T, Y, 0.001, 1000, 0.5)
h, i, j, j1 = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
k, l, m, m1 = learn_linreg_GD(X.T, Y, 0.0001, 1000, 0.5)
n, o, p, p1 = learn_linreg_GD(X.T, Y, 0.01, 600, 0.5)
q, r, u, u1 = learn_linreg_GD(X.T, Y, 0.01, 300, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.001 and iterations: 100.')
axs[0, 1].plot(range(1000), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.001 and iterations: 1000.')
axs[1, 0].plot(range(100), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.0001 and iterations: 100.')
axs[1, 1].plot(range(1000), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.0001 and iterations: 1000.')
axs[2, 0].plot(range(600), o, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.01 and iterations: 600.')
axs[2, 1].plot(range(300), r, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.01 and iterations: 300.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='loss')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```

```
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in add

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: invalid value encountered in double_scalars
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in add
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in square

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: overflow encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered in subtract
  # Remove the CWD from sys.path while we load stuff.
```



Observations:

- The optimum learning rate should be minor than 0.001 otherwise the gradient will be bigger than the previous value and at the moment of making it absolute it will increase.
- Even though the values has been scaled they are too big that at some point it causes overflow of information.
- The more iteractions given to the algorithm, the less iterations it needs to reach convergence. It tends to take more iteractions to reach the convergence.

> Second, we test the algorithm by using the 20% remaining of data.

```
Y = airq402_new_test['price'].values
Y = np.reshape(Y, (len(Y),1))
X = airq402_new_test.drop(['price'], axis=1).values
column_one = np.ones((X.shape[0],1))
```

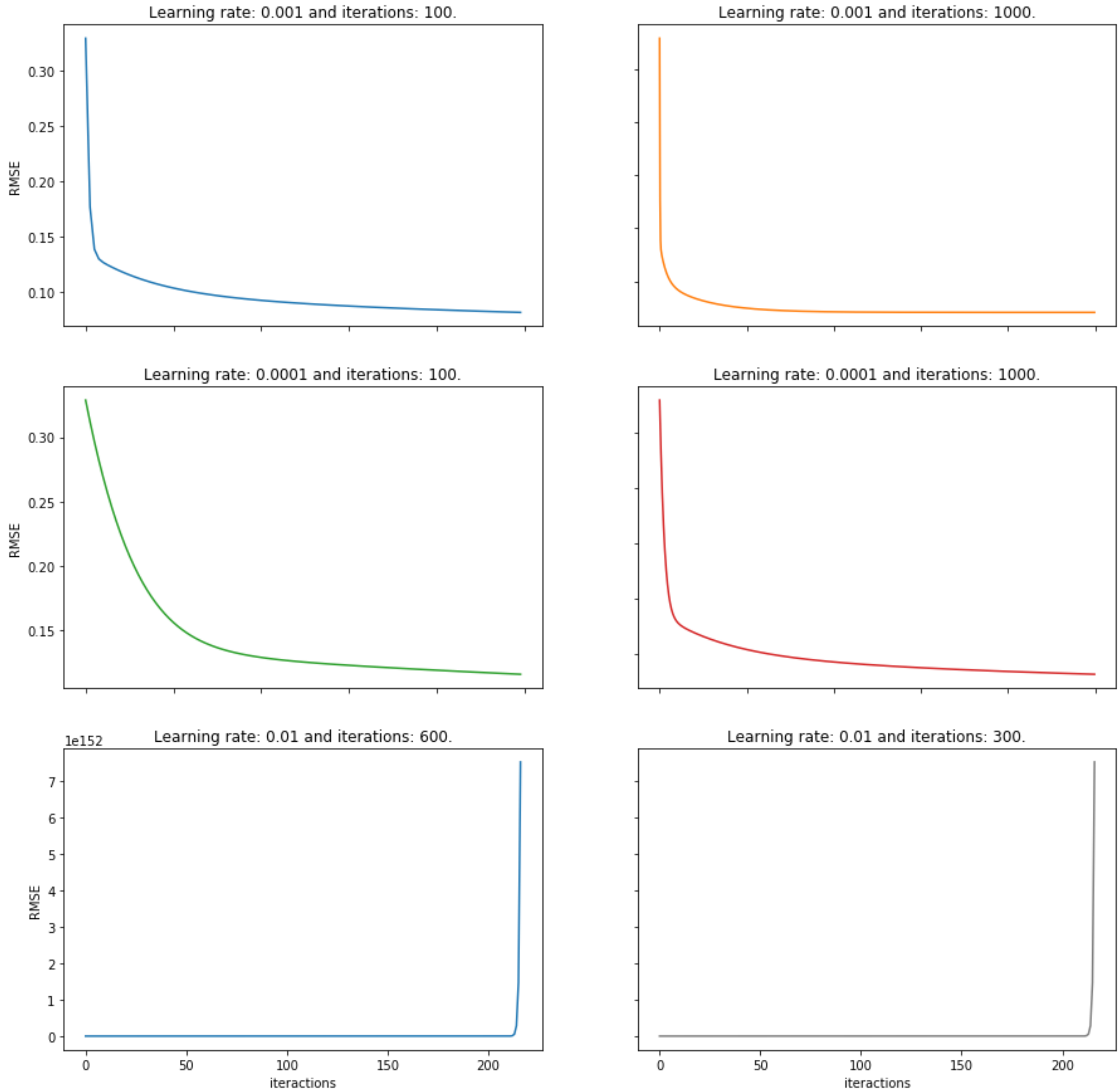
```
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[0.17810831]
 [0.12907062]
 [0.13370021]
 [0.00724456]
 [0.04716737]]
 Loss [2.143885309263883, 1.8906949186721889, 1.6675861761385615, 1.4709837454478265, 1.2977372451729323, 1.14507073961693, 1.0
 RMSE [14.53297341]
```

> Plotting the graph to see the behavior of the RMSE in regards to the number or iteractions.

```
a, b, c, f1 = learn_linreg_GD(X.T, Y, 0.001, 100, 0.5)
d, e, f, f2 = learn_linreg_GD(X.T, Y, 0.001, 1000, 0.5)
h, i, j, f3 = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
k, l, m, f4 = learn_linreg_GD(X.T, Y, 0.0001, 1000, 0.5)
n, o, p, f5 = learn_linreg_GD(X.T, Y, 0.01, 600, 0.5)
q, r, u, f6 = learn_linreg_GD(X.T, Y, 0.01, 300, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), f1)
axs[0, 0].set_title('Learning rate: 0.001 and iterations: 100.')
axs[0, 1].plot(range(1000), f2, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.001 and iterations: 1000.')
axs[1, 0].plot(range(100), f3, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.0001 and iterations: 100.')
axs[1, 1].plot(range(1000), f4, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.0001 and iterations: 1000.')
axs[2, 0].plot(range(600), f5, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.01 and iterations: 600.')
axs[2, 1].plot(range(300), f6, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.01 and iterations: 300.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='RMSE')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```

```
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in add

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: invalid value encountered in double_scalars
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in add
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in square

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: overflow encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered in subtract
  # Remove the CWD from sys.path while we load stuff.
```



Observations:

- Same effect happens while dealing with the learning rate. In addition, an interesting trend is possible to appreciate: a learning rate of 0.0001 reaches convergence faster than a learning rate of 0.001.
- Addressing the RMSE, the more iterations the algorithm runs it show us the good performance and minimization of the error. As it is possible to see in figure 2 with 1000 iteractions, from 300 iterations and on the algorithm has reached convergence. Therefore, it is an optimal number of iterations.

Parkinsons Updrs

> First, we train the algorithm by using the 80% of the data.

```
Y = parkinsons_updrs_clean_train['total_UPDRS'].values
Y = np.reshape(Y, (len(Y),1))
```

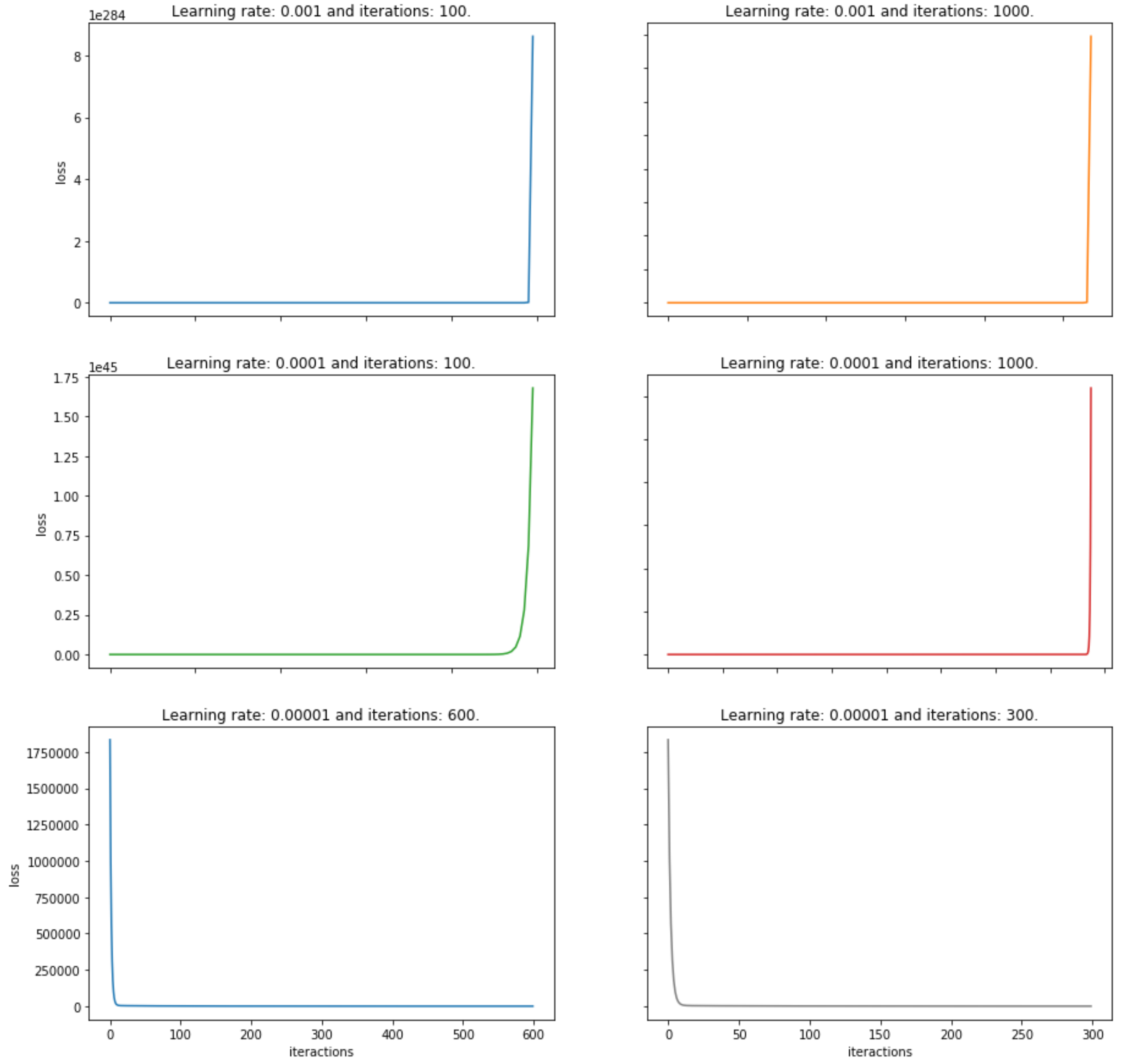
```
X = parkinsons_updrs_clean_train.drop(['total_UPDRS'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[-2.82925173e+20]
 [-1.48445929e+20]
 [-1.68451652e+20]
 [-9.96373764e+19]
 [-1.38272158e+20]
 [-2.71725744e+19]
 [-3.21530399e+19]
 [-2.66468571e+19]
 [-1.29317328e+19]
 [-1.55050557e+20]
 [-1.36533199e+20]
 [-1.12637855e+20]
 [-6.29505839e+19]]
 Loss [5868057.427833743, 14430859.144509997, 35357186.63336449, 86526301.84152436, 211664889.17823932, 517716624.56338894, 126
 RMSE [1.37836831e+21]
```

> Next, a graph showing us the behavior of the information are required to establish an optimum number of iterations and learning rate.

```
a, b, c, c1 = learn_linreg_GD(X.T, Y, 0.001, 100, 0.5)
d, e, f, f1 = learn_linreg_GD(X.T, Y, 0.001, 1000, 0.5)
h, i, j, j1 = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
k, l, m, m1 = learn_linreg_GD(X.T, Y, 0.0001, 1000, 0.5)
n, o, p, p1 = learn_linreg_GD(X.T, Y, 0.00001, 600, 0.5)
q, r, u, u1 = learn_linreg_GD(X.T, Y, 0.00001, 300, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.001 and iterations: 100.')
axs[0, 1].plot(range(1000), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.001 and iterations: 1000.')
axs[1, 0].plot(range(100), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.0001 and iterations: 100.')
axs[1, 1].plot(range(1000), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.0001 and iterations: 1000.')
axs[2, 0].plot(range(600), o, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.00001 and iterations: 600.')
axs[2, 1].plot(range(300), r, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.00001 and iterations: 300.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='loss')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```

```
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in add

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: invalid value encountered in double_scalars
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in add
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in square

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: overflow encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: invalid value encountered in matmul
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
```



Observations:

- The optimum learning rate should be minor than 0.0001 otherwise the gradient will be bigger than the previous value and at the moment of making it absolute it will increase as it is possible to appreciate in the first 4 graphs.
- Even though the values has been scaled they are too big that at some point it causes overflow of information.

> Second, we test the algorithm by using the 20% remaining of data.

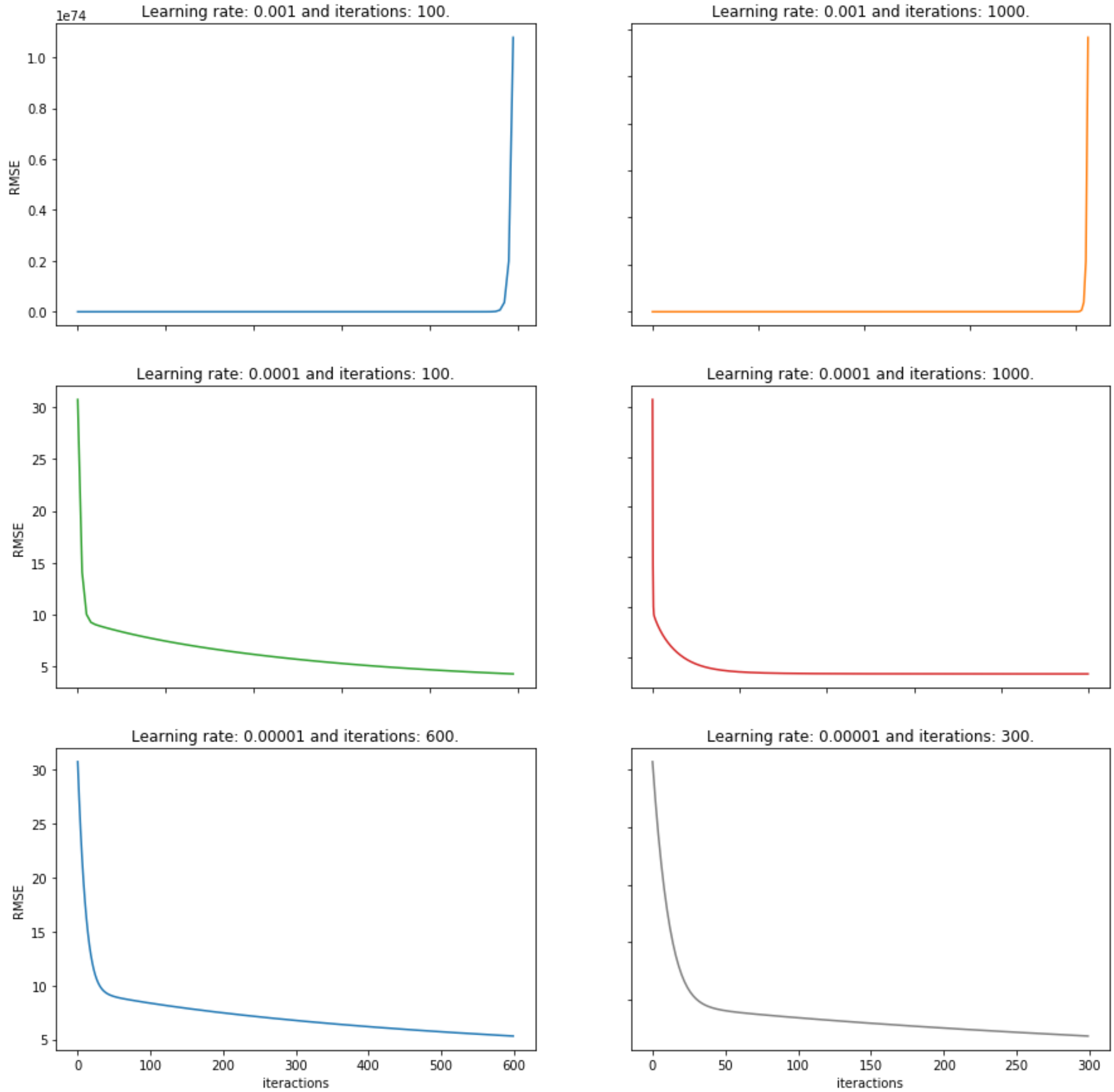
```
Y = parkinsons_updrs_clean_test['total_UPDRS'].values
Y = np.reshape(Y, (len(Y),1))
X = parkinsons_updrs_clean_test.drop(['total_UPDRS'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[ 6.45445286]
 [ 4.1022893 ]
 [ 7.23006865]
 [-2.64528906]
 [29.46862808]
 [ 0.76087603]
 [ 0.78896519]
 [ 0.65088705]
 [ 0.33271323]
 [ 2.61493005]
 [ 4.09134588]
 [-2.14816413]
 [ 1.90705176]]
 Loss [875438.8260034982, 115610.34347000116, 17416.76598893109, 4620.093007389907, 2853.393455444777, 2518.476218624972, 2375.9
 RMSE [636.82732705]
```

> Plotting the graph to see the behavior of the RMSE in regards to the number or iteractions.

```
a, b, c, f1 = learn_linreg_GD(X.T, Y, 0.001, 100, 0.5)
d, e, f, f2 = learn_linreg_GD(X.T, Y, 0.001, 1000, 0.5)
h, i, j, f3 = learn_linreg_GD(X.T, Y, 0.0001, 100, 0.5)
k, l, m, f4 = learn_linreg_GD(X.T, Y, 0.0001, 1000, 0.5)
n, o, p, f5 = learn_linreg_GD(X.T, Y, 0.00001, 600, 0.5)
q, r, u, f6 = learn_linreg_GD(X.T, Y, 0.00001, 300, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), f1)
axs[0, 0].set_title('Learning rate: 0.001 and iterations: 100.')
axs[0, 1].plot(range(1000), f2, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.001 and iterations: 1000.')
axs[1, 0].plot(range(100), f3, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.0001 and iterations: 100.')
axs[1, 1].plot(range(1000), f4, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.0001 and iterations: 1000.')
axs[2, 0].plot(range(600), f5, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.00001 and iterations: 600.')
axs[2, 1].plot(range(300), f6, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.00001 and iterations: 300.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='RMSE')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```

```
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:90: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in add

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: invalid value encountered in double_scalars
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in add
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: overflow encountered in square

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:37: RuntimeWarning: overflow encountered in square
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: overflow encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered in matmul
  # Remove the CWD from sys.path while we load stuff.
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered in subtract
  # Remove the CWD from sys.path while we load stuff.
```



Observations:

- Graph 3 and 5 shows a different behavior as the previous seen. The increasing number of iteraction has a minimum effect in reaching convergence and also the variation in the learning rate is not having an impact in the velocity of the gradient.
- Learning rates less than < 0.0001 returns a gradient bigger than the Betas values making the curve to increase rather than to decrease.

Red Wine Quality

↳ *10 celdas ocultas*

1.2 **PART B** STEP LENGTH FOR GRADIENT DESCENT

In order to keep organization the information is going to be presented in organization according to the algorithm.

Steplength Backtracking

> As a starting point the main code of the algorithm is presented from which future calculations in each dataset are going to be executed.

```
# DEF general function F(x) for multivariate Linear Regression
def main_function(X, y, beta_old):
  main_function = np.dot(((y - X.T@beta_old).T),(y - X.T@beta_old))
  return main_function

# DEF of the derivative of the function
def derivative(X, y, beta_old):
  derivative = ((-2)*X@(y - X.T@beta_old))
  return derivative

def main_function_new(X, y, u, beta_old):
  new_function_X = main_function(X, y,beta_old - u*derivative(X, y, beta_old))
  return new_function_X


def stepsize_backtracking(X, y, beta_old, a, b):
  u = 1.
  left = main_function_new(X, y, u, beta_old)
  right = main_function(X, y, beta_old) - a*u*((derivative(X, y, beta_old)).T@(derivative(X, y, beta_old)))
  while (left) > (right):
    u = b*u
    left = main_function_new(X, y, u, beta_old)
    right = main_function(X, y, beta_old) - a*u*((derivative(X, y, beta_old)).T@(derivative(X, y, beta_old)))
  return u

# Minimization of the gradient descent
def minimize_GD(X, y, u, num_iters, beta, n, a, b):
  beta_old = beta
  loss_decrease = []
  sum_RMSE = 0
  RMSE_plot = []
  u_array = []
  for i in range(num_iters):
    y_hat = np.dot(X.T,beta_old)
    loss = sum((y_hat - y)**2)
    # Stepsize Backtracking
    u = stepsize_backtracking(X, y, beta_old, a, b)

    # Measure the values of the new betas.
    Beta_new = beta_old - u*derivative(X, y, beta_old)
    # Call of the function for loss calculation
    loss_calculation = function(y, X, Beta_new, beta_old)
    loss_decrease.append(loss_calculation)
    RMSE = RMSE_function(X, y, y_hat)
    sum_RMSE+=RMSE
    RMSE_plot.append(RMSE)
    u_array.append(u)
    beta_old = Beta_new
  return Beta_new, loss_decrease, sum_RMSE, RMSE_plot


def learn_linreg_GD(X, y, num_iters, a, b):
    X = X
    y = Y
    n = X.shape[0]
    beta = np.zeros(n)
    beta = np.reshape(beta, (len(beta),1))
    beta_hat, loss, RMSE, RMSE_plot = minimize_GD(X, y, u, num_iters, beta, n, a, b)
    return  beta_hat,loss, RMSE, RMSE_plot

# Function for the Loss.
def function(y, X, Beta_new, beta_old):
  a = abs(np.sum((y - X.T@beta_old)**2) - np.sum((y - X.T@Beta_new)**2))
  return a

#Function for the RMSE.
def RMSE_function(X, y, y_hat):
  a = np.sqrt(sum((y - y_hat)**2)/y.shape[0])
  return a
```

AIRQ402 Dataset

↳ *9 celdas ocultas*

Parkinsons UPDRS dataset

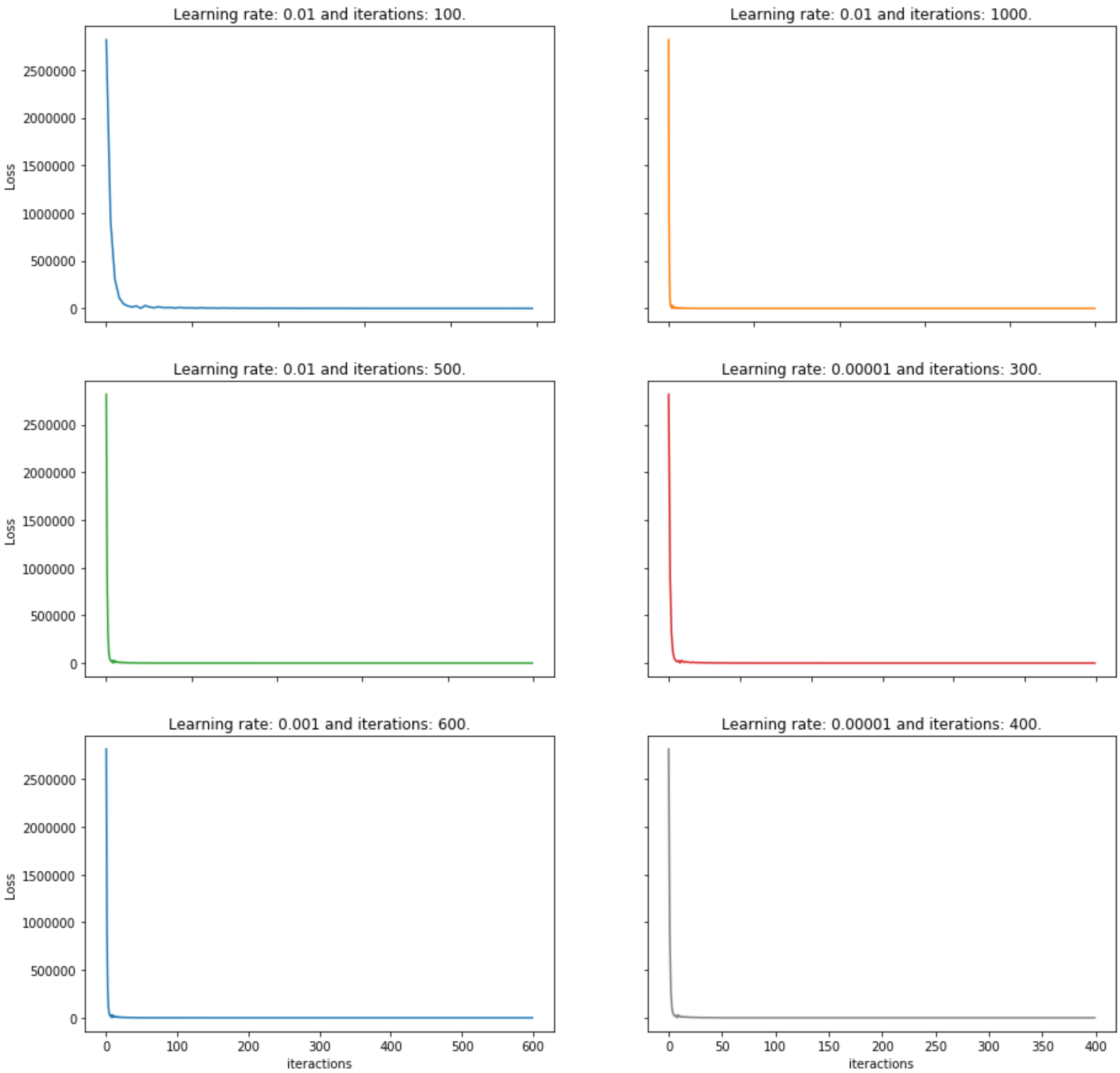
> First, we train the algorithm in the training dataset.

```
Y = parkinsons_updrs_clean_train['total_UPDRS'].values
Y = np.reshape(Y, (len(Y),1))
X = parkinsons_updrs_clean_train.drop(['total_UPDRS'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 1000, 0.01, 0.5)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[ 8.02912125]
 [ 1.8987217 ]
 [ 3.69779818]
 [-1.73808362]
 [41.39397807]
 [ 3.07711776]
 [ 0.39848363]
 [-4.64275532]
 [-1.22554859]
 [-1.74049651]
 [ 2.47850431]
 [-1.03861753]
 [-3.63218597]]
 Loss [2816477.1448810003, 915431.8573672174, 306969.17145349, 111259.37122457376, 47498.49190113158, 26036.44855910621, 14595.
 RMSE [3381.51620791]
```

> We plot a visualization of the behavior of the data in the set

```
a, b, c, f1 = learn_linreg_GD(X.T, Y, 100, 0.01, 0.5)
d, e, f, f2 = learn_linreg_GD(X.T, Y, 1000, 0.01, 0.5)
h, i, j, f3 = learn_linreg_GD(X.T, Y, 500, 0.01, 0.5)
k, l, m, f4 = learn_linreg_GD(X.T, Y, 300, 0.00001, 0.5)
n, o, p, f5 = learn_linreg_GD(X.T, Y, 600, 0.001, 0.5)
q, r, u, f6 = learn_linreg_GD(X.T, Y, 400, 0.00001, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.01 and iterations: 100.')
axs[0, 1].plot(range(1000), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.01 and iterations: 1000.')
axs[1, 0].plot(range(500), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.01 and iterations: 500.')
axs[1, 1].plot(range(300), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.00001 and iterations: 300.')
axs[2, 0].plot(range(600), o, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.001 and iterations: 600.')
axs[2, 1].plot(range(400), r, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.00001 and iterations: 400.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='Loss')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```



Observations:

- It takes more or less 15 iterations to reach convergence with a 0.1 learning rate. Independently of the learning rate, the loss decrease with high speed no matter the number of iterations.

> Check the accuracy of the algorithm in the test set

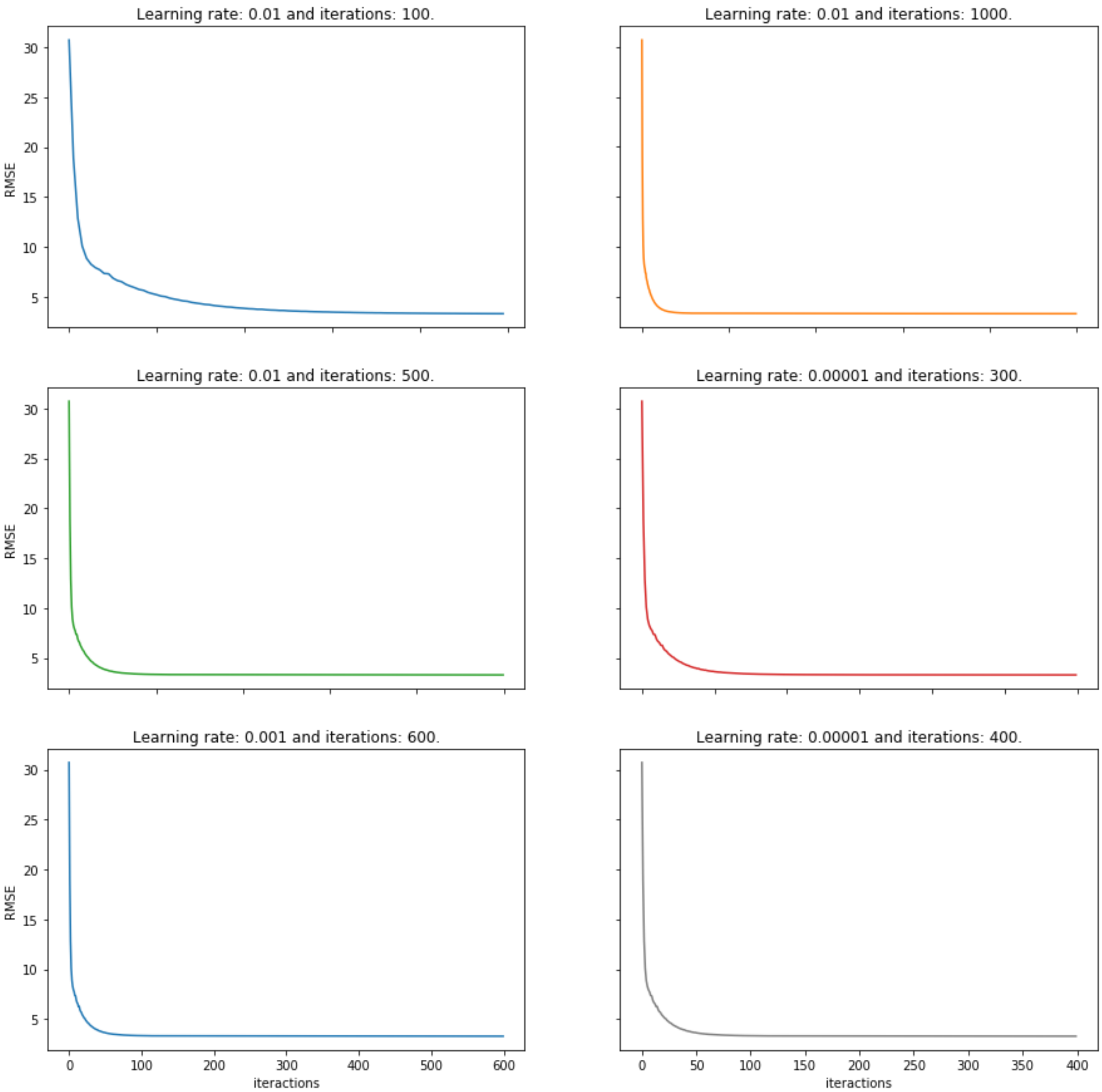
```
Y = parkinsons_updrs_clean_test['total_UPDRS'].values
Y = np.reshape(Y, (len(Y),1))
X = parkinsons_updrs_clean_test.drop(['total_UPDRS'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 1000, 0.01, 0.5)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

Betas [[8.53484743]

> Plot the RMSE values in regards to the number of iterations

[42.31908026]

```
a, b, c, f1 = learn_linreg_GD(X.T, Y, 100, 0.01, 0.5)
d, e, f, f2 = learn_linreg_GD(X.T, Y, 1000, 0.01, 0.5)
h, i, j, f3 = learn_linreg_GD(X.T, Y, 500, 0.01, 0.5)
k, l, m, f4 = learn_linreg_GD(X.T, Y, 300, 0.00001, 0.5)
n, o, p, f5 = learn_linreg_GD(X.T, Y, 600, 0.001, 0.5)
q, r, u, f6 = learn_linreg_GD(X.T, Y, 400, 0.00001, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), f1)
axs[0, 0].set_title('Learning rate: 0.01 and iterations: 100.')
axs[0, 1].plot(range(1000), f2, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.01 and iterations: 1000.')
axs[1, 0].plot(range(500), f3, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.01 and iterations: 500.')
axs[1, 1].plot(range(300), f4, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.00001 and iterations: 300.')
axs[2, 0].plot(range(600), f5, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.001 and iterations: 600.')
axs[2, 1].plot(range(400), f6, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.00001 and iterations: 400.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='RMSE')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```



In comparison with the graphs above, we can see how much the iterations increase in a graph with high learning rate (1). The less number of

Red Wine Quality

↳ 9 celdas ocultas

Bold Driver Step Size

The algorithm is presented

↳ 1 celda oculta

AIRQ402 dataset

↳ 11 celdas ocultas

Parkinsons UPDRS dataset

↳ 10 celdas ocultas

Red Wine Quality

First, we measure the algorithm in the training set.

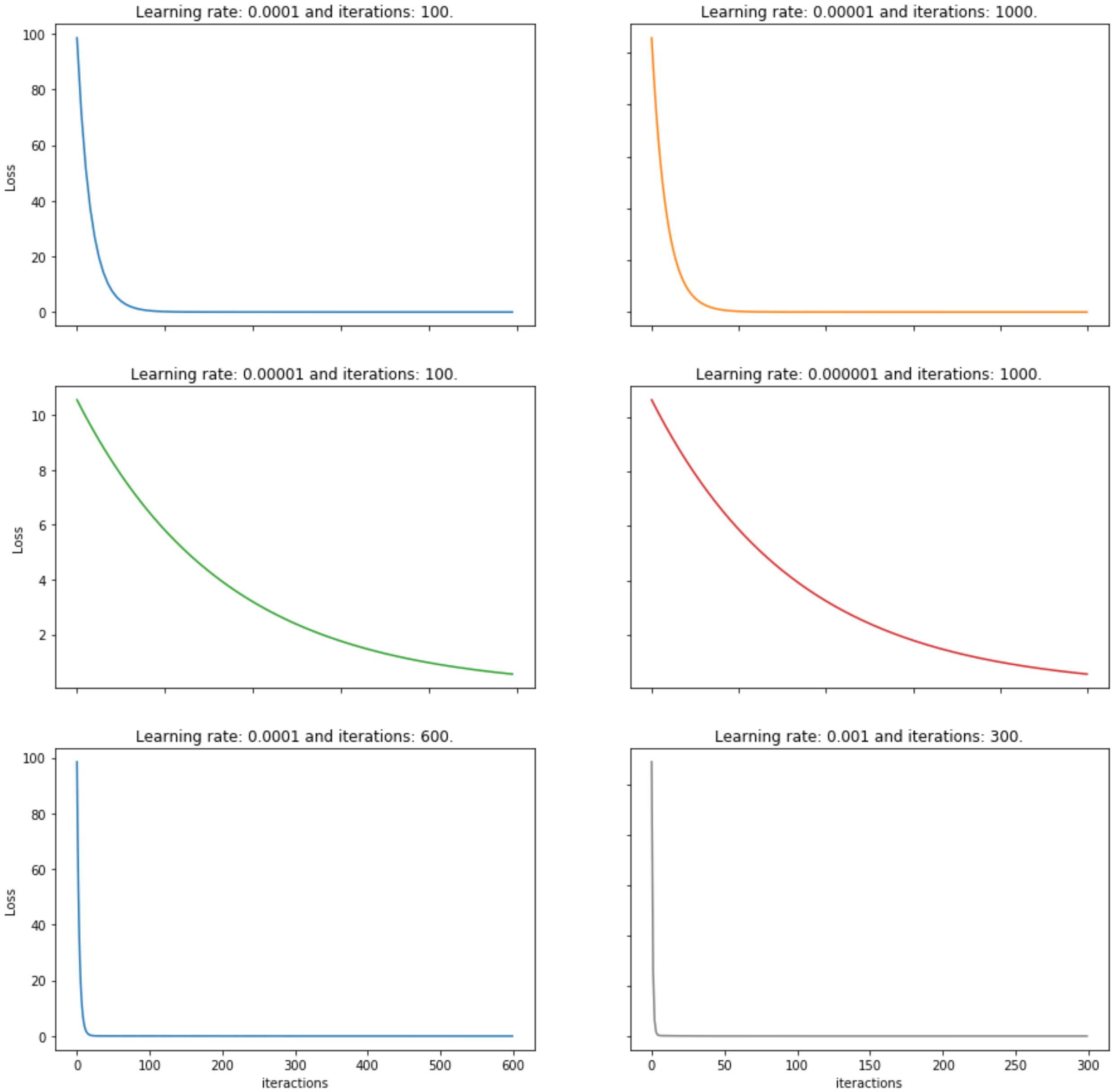
```
Y = winequality_red_normalized_train['quality'].values
Y = np.reshape(Y, (len(Y),1))
Ytest = winequality_red_normalized_test['quality'].values
Ytest = np.reshape(Ytest, (len(Ytest),1))
X = winequality_red_normalized_train.drop(['quality'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 100, 0.001, 1.5, 0.2)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[ 0.3547984 ]
 [ 0.08747707]
 [-0.16525196]
 [ 0.07552859]
 [ 0.00994564]
 [-0.04716793]
 [ 0.01555724]
 [-0.08616493]
 [-0.03115222]
 [ 0.09638909]
 [ 0.18279574]
 [ 0.37740067]]
 Loss [273.18824208799015, 65.31349916096086, 15.906621444965467, 4.14694292968727, 1.3321176559962247, 0.6435102950755542, 0.4
 RMSE [14.43271242]
```

Plotting the relation between the loss function and the iterations to see the impact and the curve until convergence.

```
a, b, c, f1 = learn_linreg_GD(X.T, Y, 100, 0.0001, 1.5, 0.2)
d, e, f, f2 = learn_linreg_GD(X.T, Y, 1000, 0.00001, 1.5, 0.2)
h, i, j, f3 = learn_linreg_GD(X.T, Y, 100, 0.00001, 1.5, 0.2)
k, l, m, f4 = learn_linreg_GD(X.T, Y, 1000, 0.000001, 1.5, 0.2)
n, o, p, f5 = learn_linreg_GD(X.T, Y, 600, 0.0001, 1.5, 0.2)
q, r, u, f6 = learn_linreg_GD(X.T, Y, 300, 0.001, 1.5, 0.2)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.0001 and iterations: 100.')
axs[0, 1].plot(range(1000), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.00001 and iterations: 1000.')
axs[1, 0].plot(range(100), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.00001 and iterations: 100.')
axs[1, 1].plot(range(1000), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.000001 and iterations: 1000.')
axs[2, 0].plot(range(600), o, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.0001 and iterations: 600.')
axs[2, 1].plot(range(300), r, 'tab:gray')
```

```
axs[2, 1].set_title('Learning rate: 0.001 and iterations: 300.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='Loss')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```



Observations:

- A bigger learning rate allows us to see how the curve of the loss goes donw faster than the others. Figure 6. Nonetheless, lower learning rate predict with more accuracy.
- The smaller the learning rate the more pronounced curvature it will have.

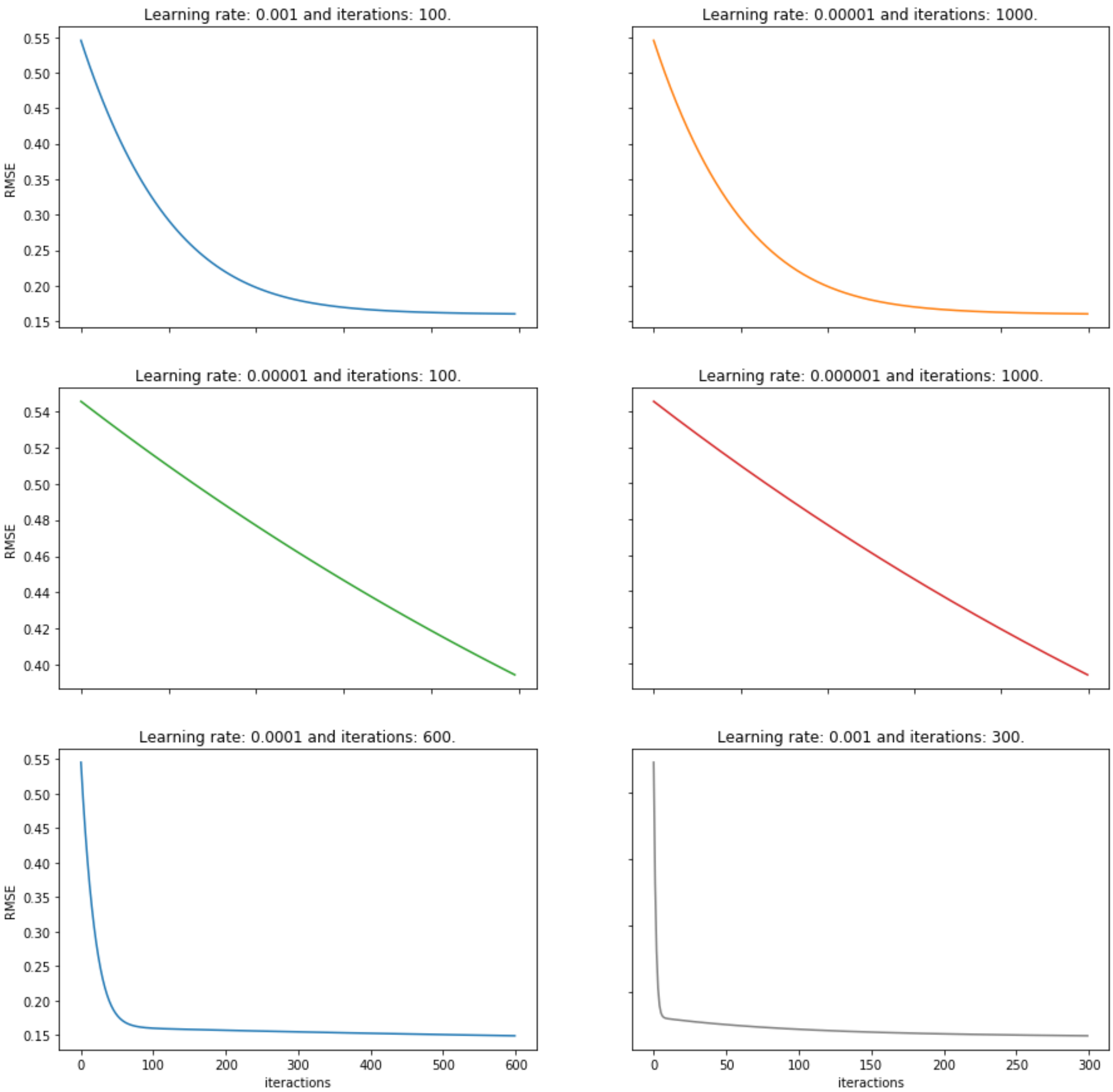
> Check the efectiveness of the algorithm in the test set.

```
Y = winequality_red_normalized_test['quality'].values
Y = np.reshape(Y, (len(Y),1))
X = winequality_red_normalized_test.drop(['quality'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)

Betas, Loss, RMSE, RMSE_plot = learn_linreg_GD(X.T, Y, 100, 0.001, 1.5, 0.2)
print('Betas', Betas,'\n', 'Loss', Loss,'\n', 'RMSE', RMSE)
```

```
Betas [[ 0.28915805]
 [ 0.11276127]
 [-0.03016974]
 [ 0.12861739]
 [ 0.00482627]
 [ 0.0026185 ]
 [ 0.05585596]
 [-0.02092428]
 [ 0.04934809]
 [ 0.10464348]
 [ 0.10047022]
 [ 0.22802516]]
Loss [52.76776848417289, 20.718844717596134, 8.146040014126578, 3.21359264686388, 1.2784104975763189, 0.5190387901013587, 0.22
RMSE [15.93843788]
```

Plotting the relation between the RMSE and the iterations to see the impact and the curve until convergence.

```
a, b, c, f1 = learn_linreg_GD(X.T, Y, 100, 0.0001, 1.5, 0.2)
d, e, f, f2 = learn_linreg_GD(X.T, Y, 1000, 0.00001, 1.5, 0.2)
h, i, j, f3 = learn_linreg_GD(X.T, Y, 100, 0.00001, 1.5, 0.2)
k, l, m, f4 = learn_linreg_GD(X.T, Y, 1000, 0.000001, 1.5, 0.2)
n, o, p, f5 = learn_linreg_GD(X.T, Y, 600, 0.0001, 1.5, 0.2)
q, r, u, f6 = learn_linreg_GD(X.T, Y, 300, 0.001, 1.5, 0.2)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), f1)
axs[0, 0].set_title('Learning rate: 0.001 and iterations: 100.')
axs[0, 1].plot(range(1000), f2, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.00001 and iterations: 1000.')
axs[1, 0].plot(range(100), f3, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.00001 and iterations: 100.')
axs[1, 1].plot(range(1000), f4, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.000001 and iterations: 1000.')
axs[2, 0].plot(range(600), f5, 'tab:blue')
axs[2, 0].set_title('Learning rate: 0.0001 and iterations: 600.')
axs[2, 1].plot(range(300), f6, 'tab:gray')
axs[2, 1].set_title('Learning rate: 0.001 and iterations: 300.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='RMSE')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```



Observations:

- Lower learning rate, the more calculations are required and the RMSE diminishes efectively. A bigger learning rate experiments otherwise.

COMPARISON BETWEEN MODELS

- Bold Driver algorithm reaches convergence at a faster pace than the rest because since the condition determines if the step size must grow or decrease. If the $F_0(B)$ decreases, the learning rate could do bigger steps. Same happens otherwise.
- Backtracking algo reduces the number of iterations until convergence because each time it is fitting a perfect learning rate which is optimum to reach convergence. Nonetheless, in order to find that ideal learning rate it takes a number of iteractions that the previous model don't experience.
- The basic multivariate linear regression model works well but it is not effective when talking about big datasets. It is time-consuming and expensive. Therefore, it is a good call to choose adaptative learning rate algorithms.