MACHINE LEARNING LAB - TUTORIAL 5

Juan Fernando Espinosa

303158

1. PRE-PROCESS GIVEN DATASETS

```
import pandas as pd
import numpy as np
from numpy import random
%matplotlib inline
import math
import matplotlib.pyplot as plt
from google.colab import files
from google.colab import drive
from mpl_toolkits.mplot3d import Axes3D
```

```
# Importing Bank CSV
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 5/bank.csv"

# Importing Wine Quality - red
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 5/winequality-red.csv"

# Importing Wine Quality - white
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 5/winequality-white.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 5/bank.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 5/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 5/winequality-white.csv'
```

1.1 Pre-processing Bank Dataset

```
missing_values = ['-','na','Nan','nan','n/a','?']
bank = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 5/bank.csv', sep=';', na_values = missing_values)
```

```
bank.head()
```

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	oct	79	1	-1	0
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	may	220	1	339	4
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	apr	185	1	330	1
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	jun	199	4	-1	0
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	may	226	1	-1	0

```
# Check for missing or incongruent values
check = bank.empty
print('checking missing values:',check)
print('Sum of errors:',bank.isnull().sum())
```

```
checking missing values: False
Sum of errors: age          0
job          0
marital      0
education    0
default      0
balance      0
housing      0
loan         0
contact      0
day          0
month        0
duration     0
campaign     0
pdays        0
previous     0
poutcome     0
y            0
dtype: int64
```

> convert non-numeric data into numeric one using dummies. Bear into account that it is not feasible to get dummies to the target column. Therefore, replacing the classification with {0, 1} it is an adequate action.

```
#transform the y column into numeric 0 and 1
print(bank["y"].value_counts())
bank['y'] = bank['y'].replace(['no', 'yes'], [0, 1])
```

```
no     4000
yes     521
Name: y, dtype: int64
```

```
# Encoding the information with get_dummies.
bank = pd.get_dummies(bank)
bank.head()
```

	age	balance	day	duration	campaign	pdays	previous	y	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_manageme
0	30	1787	19	79	1	-1	0	0	0	0	0	0	
1	33	4789	11	220	1	339	4	0	0	0	0	0	
2	35	1350	16	185	1	330	1	0	0	0	0	0	
3	30	1476	3	199	4	-1	0	0	0	0	0	0	
4	59	0	5	226	1	-1	0	0	0	1	0	0	

> Normalizing the dataset before splitting

```
# Bearing in mind the feedback given in the previous lab, the "y" column is not normalized.
def normalize(dataset):
    dataNorm=((dataset-dataset.min())/(dataset.max()-dataset.min()))
    dataNorm["y"]=dataset["y"]
    return dataNorm
```

```
data = normalize(bank)
data.head()
```

	age	balance	day	duration	campaign	pdays	previous	y	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	jo
0	0.161765	0.068455	0.600000	0.024826	0.000000	0.000000	0.00	0	0.0	0.0	0.0	0.0	
1	0.205882	0.108750	0.333333	0.071500	0.000000	0.389908	0.16	0	0.0	0.0	0.0	0.0	
2	0.235294	0.062590	0.500000	0.059914	0.000000	0.379587	0.04	0	0.0	0.0	0.0	0.0	
3	0.161765	0.064281	0.066667	0.064548	0.061224	0.000000	0.00	0	0.0	0.0	0.0	0.0	
4	0.588235	0.044469	0.133333	0.073486	0.000000	0.000000	0.00	0	0.0	1.0	0.0	0.0	

> Split into train (80%) and test (20%) sets

```
bank_train = data.sample(frac=0.8)
bank_test = data.drop(bank_train.index)
```

1.2 Pre-processing Wine quality-red and Wine Quality-white Datasets

```
missing_values = ['-','na','Nan','nan','n/a','?']
red_wine = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 5/winequality-red.csv', sep=';', na_values = missing_va
red_wine.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	

```
missing_values = ['-','na','Nan','nan','n/a','?']
white_wine = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 5/winequality-white.csv', sep=';', na_values = missin
white_wine.head()
```

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

> Concatenating both white and red wine datasets

Given two datasets with similar number of columns and considering that the objective is to measure Wine quality no matter the type of wine, the decision of concatenating both datasets helps me to accomplish the goal

```
frames = [red_wine, white_wine]
wine_quality = pd.concat(frames)
wine_quality.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	

```
# Check for missing or incongruent values
check = wine_quality.empty
print('checking missing values:',check)
print('Sum of errors:',wine_quality.isnull().sum())
```

```
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
```

> Convert non-numeric data into numeric ones using "Get Dummies"

```
wine_quality.dtypes
```

```
fixed acidity           float64
volatile acidity        float64
citric acid             float64
residual sugar          float64
chlorides               float64
free sulfur dioxide     float64
total sulfur dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
dtype: object
```

All the columns in the dataframe are numeric, therefore, no need of encoding.

> Normalizing the dataset

```
def normalize(dataset):
    dataNorm=((dataset-dataset.min())/(dataset.max()-dataset.min()))
    dataNorm["quality"]=dataset["quality"]
    return dataNorm
wine = normalize(wine_quality)
wine.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	0.297521	0.413333	0.000000	0.019939	0.111296	0.034722	0.064516	0.206092	0.612403	0.191011	0.202899	5
1	0.330579	0.533333	0.000000	0.030675	0.147841	0.083333	0.140553	0.186813	0.372093	0.258427	0.260870	5
2	0.330579	0.453333	0.024096	0.026074	0.137874	0.048611	0.110599	0.190669	0.418605	0.241573	0.260870	5
3	0.611570	0.133333	0.337349	0.019939	0.109635	0.055556	0.124424	0.209948	0.341085	0.202247	0.260870	6
4	0.297521	0.413333	0.000000	0.019939	0.111296	0.034722	0.064516	0.206092	0.612403	0.191011	0.202899	5

> Split into train (80%) and test (20%) sets

```
wine_train = wine.sample(frac=0.8)
wine_test = wine.drop(wine_train.index)
```

To maintain order and information flow exercise 2 and 3 are going to be presented for each dataset separately.

BANK DATASET

2. LINEAR CLASSIFICATION WITH GRADIENT DESCENT

First, presentation of the main functions and model for Linear Classification with Gradient Descent.

```
# Function to get the minibatches.
def mini_batch(X, y, batch_size):
  #mini_batches = []
  random_index = random.choice(len(y), len(y), replace=False)
  X_shuffle = X[random_index,:]
  y_shuffle = y[random_index,:]
  mini_batches = [(X_shuffle[i:i+batch_size,:], y_shuffle[i:i+batch_size]) for i in range(0, len(y), batch_size)]
  return mini_batches

# Function that measures the derivative of f(beta).
def gradient(X, y, beta, parameter):
```

```
  gradient = ((2/X.shape[0])*(X.T@(X@beta - y))+ 2*parameter*beta)
  return gradient

# Function that measures the loss function for the betas.
def loss_function(X, y, beta, parameter):
  loss = ((1/X.shape[0])*np.dot((y - X@beta).T,(y - X@beta))) + parameter*np.dot(beta.T, beta)
  return loss

# Function that measures the error of the model.
def RMSE_function(X, y, beta):
  error = np.sqrt(np.sum((y - X@beta)**2)/X.shape[0])
  return error


#Body of the algorithm: Stochastic Gradient Descent with fixed learning rate.

def SGD(X, y, x_test, y_test, u, parameter, batch_size, num_iters, beta):
  RMSE_epoch = []
  RMSE_test  = {}
  loss = loss_function(X, y, beta, parameter)

  for i in range(num_iters):
      mini_batches = mini_batch(X, y, batch_size)
      for j in mini_batches:
        X_mini = j[0]
        y_mini = j[1]
        beta_hat = beta - u*gradient(X_mini, y_mini, beta, parameter)
        loss_old = loss
        loss = loss_function(X_mini, y_mini, beta_hat, parameter)
        beta = beta_hat
      error = RMSE_function(X, y, beta_hat)
      RMSE_epoch.append(error)
      RMSE_test[i] = RMSE_function(x_test, y_test, beta_hat)

  return beta_hat, RMSE_epoch, RMSE_test
```

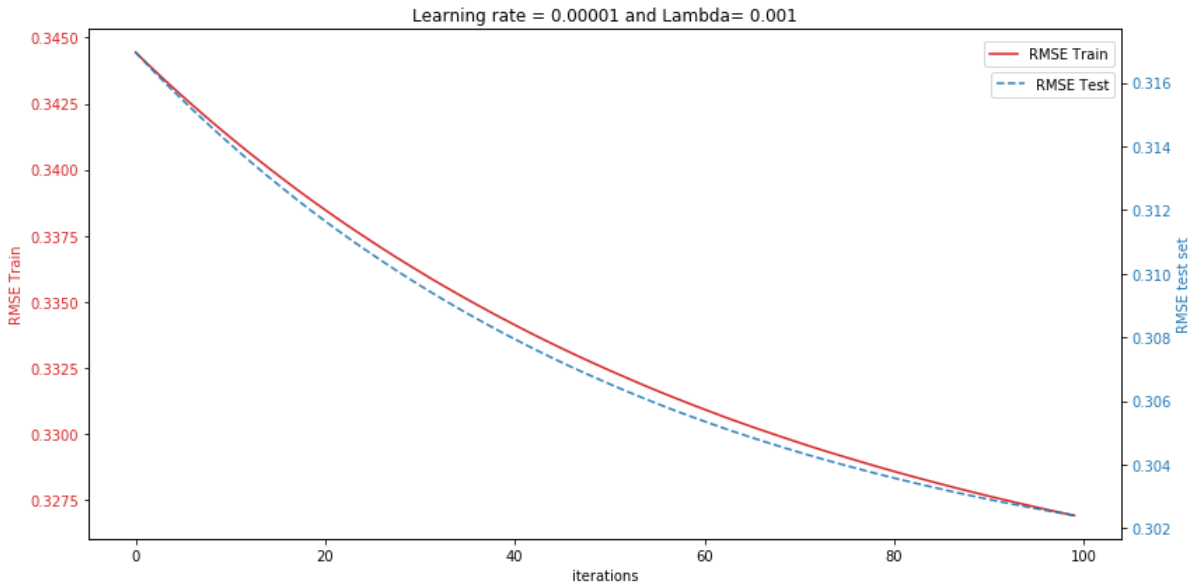
Learning rate = 0.00001 and **Lambda=** 0.01 | 100 iterations

```
y = bank_train['y'].values
y = np.reshape(y, (len(y),1))
X = bank_train.drop(['y'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))
y1 = bank_test['y'].values
y1 = np.reshape(y1, (len(y1),1))
X1 = bank_test.drop(['y'], axis=1).values
column_one = np.ones((X1.shape[0],1))
X1 = np.concatenate((column_one, X1), axis = 1)

betas, RMSE, RMSE_test = SGD(X, y, X1, y1, 0.00001, 0.01, 50, 100, beta)
print('betas', betas.T,'\n', 'RMSE',RMSE, '\n', 'RMSE',RMSE_test)
```

```
betas [[0.01173854 0.00414278 0.00079232 0.00549908 0.00262136 0.0002372
  0.00109159 0.00064478 0.00139596 0.00115234 0.00028006 0.00026195
  0.00319766 0.00137754 0.00053347 0.00066516 0.00050951 0.00185632
  0.00027503 0.00023355 0.00194159 0.00573473 0.00406222 0.00114306
  0.00555347 0.00468832 0.00035369 0.01156732 0.00017122 0.00771207
  0.00402647 0.01111483 0.00062371 0.01026061 0.00107247 0.00040546
  0.00140502 0.00161529 0.00033844 0.00101169 0.00030583 0.00126869
  0.00117356 0.00050532 0.00136433 0.00102975 0.00117933 0.00054127
  0.00166719 0.0009275  0.00262087 0.00652298]]
 RMSE [0.3444446317204611, 0.3440957631705907, 0.34375345817348507, 0.34341988336950763, 0.3430889552437503, 0.3427658472847421
 RMSE {0: 0.3169586975427122, 1: 0.3166430192990708, 2: 0.31633397063418894, 3: 0.3160333457322565, 4: 0.31573559914178106, 5: (
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test.items()
x,y = zip(*lista)

color = 'tab:blue'
```

```
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(x, y, '--', color=color, alpha=0.9)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
plt.title('Learning rate = 0.00001 and Lambda= 0.001', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```



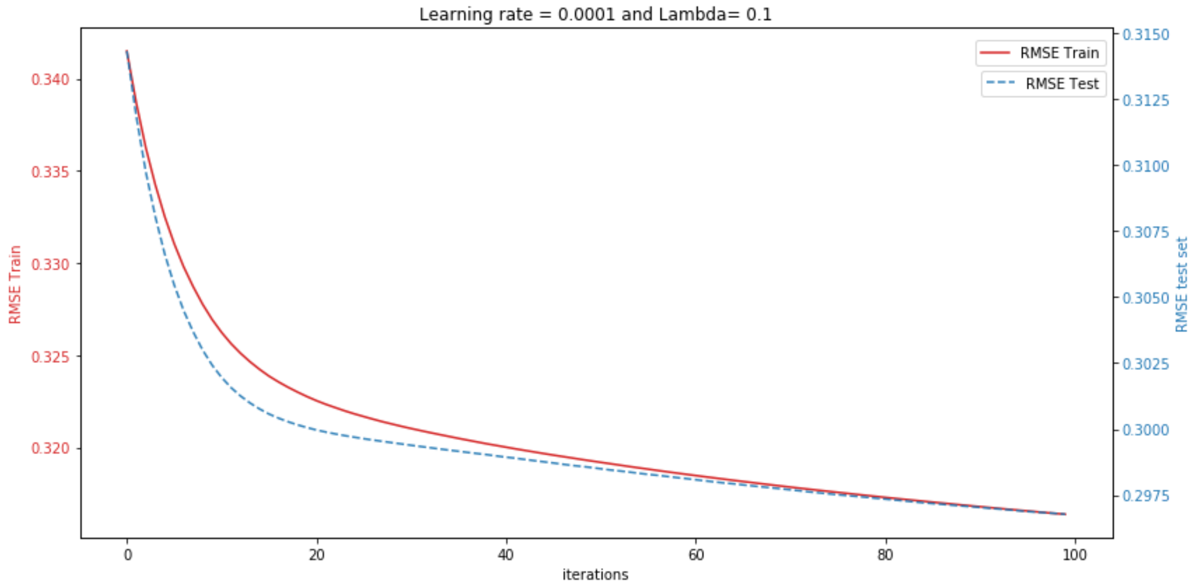
Learning rate = 0.0001 and **Lambda=** 0.1 | 100 iterations

```
y = bank_train['y'].values
y = np.reshape(y, (len(y),1))
X = bank_train.drop(['y'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))
y1 = bank_test['y'].values
y1 = np.reshape(y1, (len(y1),1))
X1 = bank_test.drop(['y'], axis=1).values
column_one = np.ones((X1.shape[0],1))
X1 = np.concatenate((column_one, X1), axis = 1)

betas1, RMSE1, RMSE_test1 = SGD(X, y, X1, y1, 0.0001, 0.1, 50, 100, beta)
print('betas', betas1.T,'\n', 'RMSE',RMSE1)
```

```
betas [[ 0.02304676  0.01006418  0.00179174  0.0080457   0.01689213 -0.00088637
   0.00541214  0.00367971  0.00377018 -0.00395214 -0.00042638  0.00021298
   0.00872044  0.00791897  0.00138396 -0.00089954  0.00298985  0.00219963
  -0.00020602  0.00133483  0.00762566  0.00233878  0.01308231 -0.00033386
   0.00926729  0.0139795   0.00013383  0.02245544  0.00059131  0.0265626
  -0.00351584  0.0272401  -0.00419335  0.0310324   0.00470488 -0.01269053
   0.00654502  0.00020583  0.00262542  0.00427613 -0.00032232 -0.00222192
   0.00217802  0.00357619 -0.00724349  0.00099336  0.0086568   0.00377773
   0.00476769  0.00414376  0.02092918 -0.00679388]]
 RMSE [0.34149043018749053, 0.3386743181444486, 0.33625435368887713, 0.33424990712180624, 0.3325132835340075, 0.331045522184778
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE1, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test1.items()
xx,yy = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(xx, yy, '--', color=color, alpha=0.9)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
```

```
plt.title('Learning rate = 0.0001 and Lambda= 0.1', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```

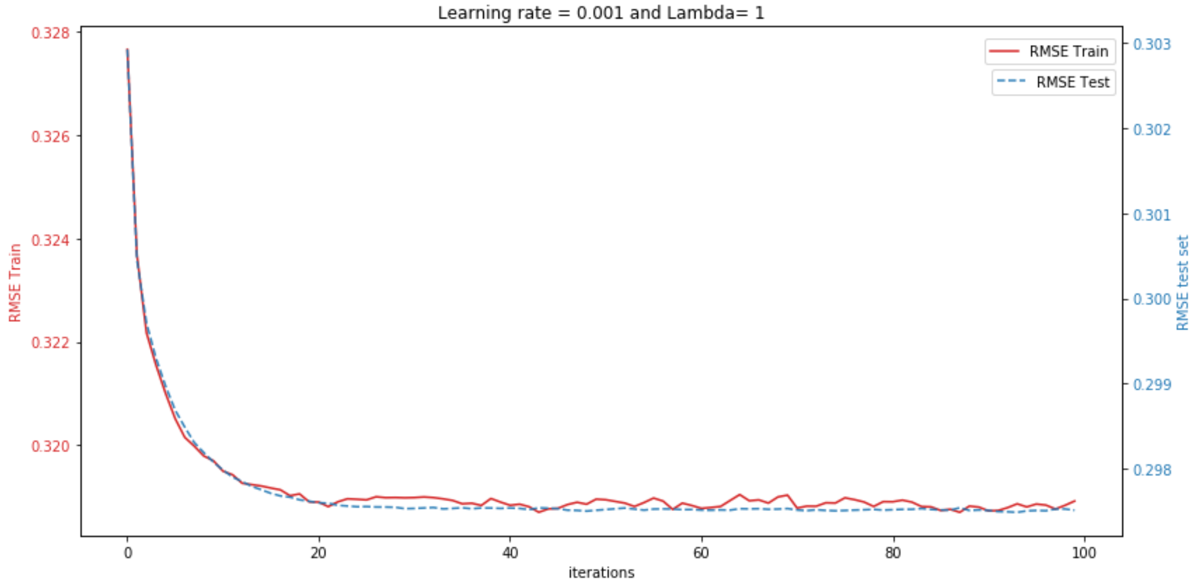


Learning rate = 0.001 and **Lambda=** 1 | 100 iterations

```
y = bank_train['y'].values
y = np.reshape(y, (len(y),1))
X = bank_train.drop(['y'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))
y1 = bank_test['y'].values
y1 = np.reshape(y1, (len(y1),1))
X1 = bank_test.drop(['y'], axis=1).values
column_one = np.ones((X1.shape[0],1))
X1 = np.concatenate((column_one, X1), axis = 1)

betas2, RMSE2, RMSE_test2 = SGD(X, y, X1, y1, 0.001, 1, 50, 100, beta)
print('betas', betas1.T,'\n', 'RMSE',RMSE1)
```

```
betas [[ 0.02304676  0.01006418  0.00179174  0.0080457   0.01689213 -0.00088637
   0.00541214  0.00367971  0.00377018 -0.00395214 -0.00042638  0.00021298
   0.00872044  0.00791897  0.00138396 -0.00089954  0.00298985  0.00219963
  -0.00020602  0.00133483  0.00762566  0.00233878  0.01308231 -0.00033386
   0.00926729  0.0139795   0.00013383  0.02245544  0.00059131  0.0265626
  -0.00351584  0.0272401  -0.00419335  0.0310324   0.00470488 -0.01269053
   0.00654502  0.00020583  0.00262542  0.00427613 -0.00032232 -0.00222192
   0.00217802  0.00357619 -0.00724349  0.00099336  0.0086568   0.00377773
   0.00476769  0.00414376  0.02092918 -0.00679388]]
 RMSE [0.34149043018749053, 0.3386743181444486, 0.33625435368887713, 0.33424990712180624, 0.3325132835340075, 0.331045522184778
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE2, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test2.items()
xxx,yyy = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(xxx, yyy, '--', color=color, alpha=0.9)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
plt.title('Learning rate = 0.001 and Lambda= 1', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```



3. HYPER PARAMETER TUNNING

Hyper-parameter tuning and Cross validation

```
def cross_validation(data):
  a = [0.0000001, 0.004, 0.003, 0.02]
  parameter = [0.001, 0.03, 0.4, 10]
  pairs = [[r, b] for r in a for b in parameter] # Pairing all learning rate and parameters possible variations
  RMSE_folds = []
  RMSE_avg = []
  RMSE_epoch = []
  hyperparameters = []
  beta = np.zeros(n)
  beta = np.reshape(beta, (len(beta),1))

  for i in pairs:
    loss = loss_function(X, y, beta, i[1])
    for j in range(0, len(data), (len(data)//5)):  # Loop to define test and training set considering the folds.
      test = X[j:j+(len(X)//5)]
      for k in range(0, len(X)):
        if k != j:
          train = X[:k+(len(X))]
      y_train = data['y'].values    # In the next 5 lines, X and Y are being defined.
      y_train = np.reshape(y_train, (len(y_train),1))
      X_train = data.drop(['y'], axis=1).values
      column_one = np.ones((data.shape[0],1))
      X_train = np.concatenate((column_one, X_train), axis = 1)
      mini_batches = mini_batch(X_train, y_train, 50)
      for j in mini_batches:
        X_mini = j[0]
        y_mini = j[1]
        beta_hat = beta - i[0]*gradient(X_mini, y_mini, beta, i[1])
        loss_old = loss
        loss = loss_function(X_mini, y_mini, beta_hat, i[1])
        beta = beta_hat
    error = RMSE_function(X, y, beta_hat) #measuring error for each epoch

    RMSE_epoch.append(error)
    RMSE_folds = sum(RMSE_epoch)/len(RMSE_epoch)  # Average RMSE per fold.
    RMSE_avg.append(RMSE_folds)
    hyperparameters.append(i)
  values = list(zip(RMSE_avg, hyperparameters))
  optimum = min(values)
  return optimum[1], hyperparameters, RMSE_avg
```

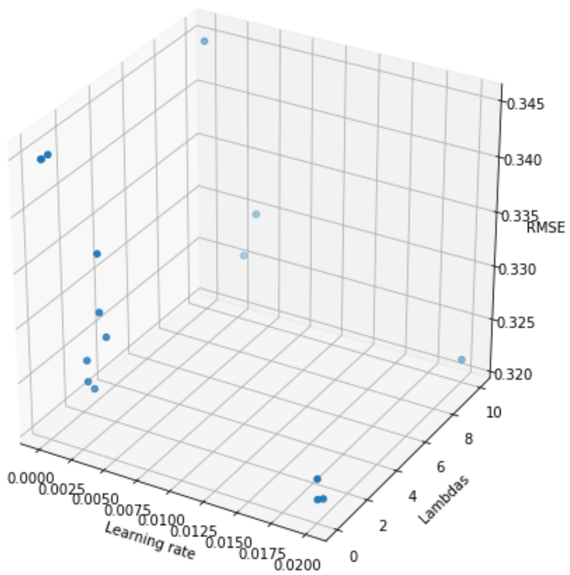
```
optimum, hyperparameters, RMSE_avg = cross_validation(data)
```

```
print('Optimum learning rate and lambda', optimum)
```

```
Optimum learning rate and lambda [0.02, 0.4]
```

> 3D representation of all the combinations of learning rate, lambdas and the Average RMSE.

```
param_final=[]
for i in RMSE_avg:
  param = i
  param_final.append(param)

lambdas1 = []
learning = []
for i in hyperparameters:
  learning_rate = i[0]
  lambdas = i[1]
  lambdas1.append(lambdas)
  learning.append(learning_rate)

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(learning, lambdas1, param_final, marker='o')
ax.set_xlabel('Learning rate')
ax.set_ylabel('Lambdas')
ax.set_zlabel('RMSE')
plt.show()
```

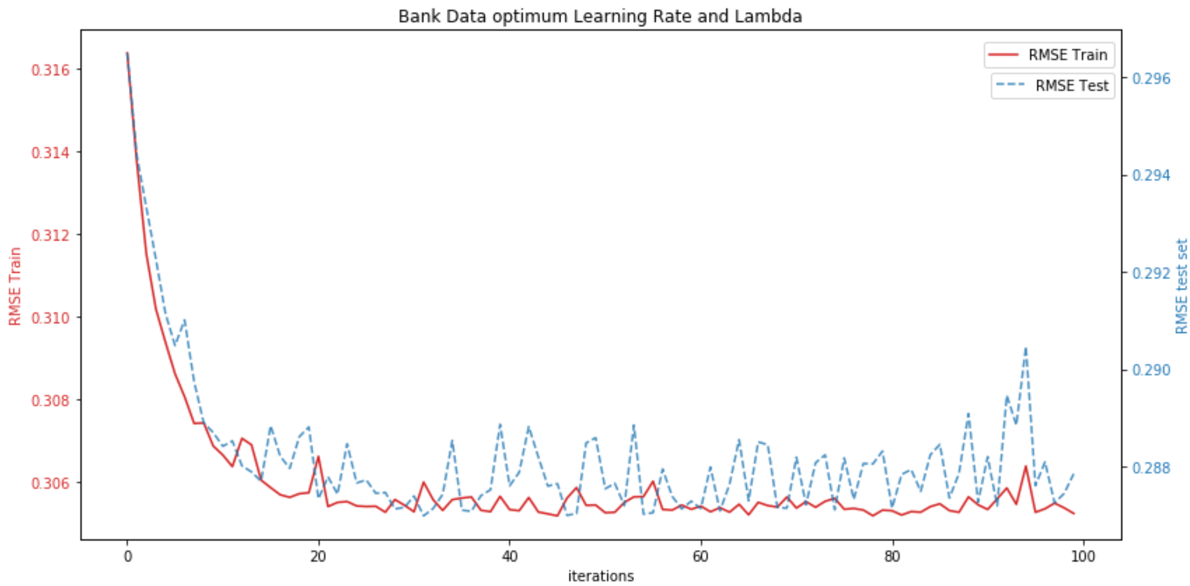


Training the model on complete training data and test data using the optimum learning rate and lambda.

```
y = bank_train['y'].values
y = np.reshape(y, (len(y),1))
X = bank_train.drop(['y'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))
y1 = bank_test['y'].values
y1 = np.reshape(y1, (len(y1),1))
X1 = bank_test.drop(['y'], axis=1).values
column_one = np.ones((X1.shape[0],1))
X1 = np.concatenate((column_one, X1), axis = 1)

betas, RMSE, RMSE_test = SGD(X, y, X1, y1, 0.01, 0.1, 50, 100, beta)
print('betas', betas.T,'\n', 'RMSE',RMSE)
```

```
betas [[ 0.03424468  0.02036978  0.00295673  0.00415563  0.10410692 -0.00765958
   0.00884841  0.00993823  0.00388714 -0.00891179 -0.00415966 -0.00212799
   0.01138593  0.02979504  0.00331131 -0.00763515  0.01157302 -0.00199394
  -0.00752948  0.00665026  0.02049231 -0.00493298  0.01868536 -0.00021076
   0.01674568  0.01938328 -0.00167352  0.02925564  0.00498905  0.03996557
  -0.00572088  0.04139701 -0.00715233  0.04086319  0.01566406 -0.02228256
   0.01412862 -0.02170763  0.01354602  0.00546565 -0.01419169 -0.01983751
   0.00753003  0.01793549 -0.01709969 -0.01192736  0.04299461  0.01740816
  -0.01718508  0.00060298  0.10280584 -0.05197905]]
 RMSE [0.3163850827376349, 0.31375002928383994, 0.3115224883248845, 0.3101891334921306, 0.30938489281357556, 0.3086211437086397
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test.items()
x1,y1 = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(x1, y1, '--', color=color, alpha=0.8)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
plt.title('Bank Data optimum Learning Rate and Lambda', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```



4. CONCLUSIONS

Notes: There is a tiny difference in the graphs above between training data and test data. It is not possible to visualize clearly in the graphs.

1. The Bank dataset after a simple test has a **bad performance** on unseen data (test set).
2. There is an obvious relationship between the **Learning rate** and **Lambda**: The bigger the amount of bias you add to the model in order to avoid overfitting the smaller the steps should be kept to guarantee a good generalization (applicable in this exercise).
3. If a high learning rate is chosen, the model will not be able to work adequately because the bias is going to be high enough and never reaches convergence and the data will be overfitted.
4. As it is possible to appreciate the model needs bias to avoid incurring in overfitting. The more bias is added to the data the less number of iterations it needs to reach convergence.
5. The last combination is meaningful: it is necessary to find an **optimal value** of learning rate and lambda, otherwise, one extra unit could make the model to incur in overfitting.
6. While mixing **Learning rate =** 0.0001 and **Lambda=** 0.1 we can appreciate that the data fits the test set better than the training set. Meaning that the generalization of the model is good.

7. In exercise 3, after finding the optimal combination the model converges in both training and test sets although there is bias as it is possible to appreciate in curve blue of the graph.

WINE QUALITY

2. LINEAR CLASSIFICATION WITH GRADIENT DESCENT

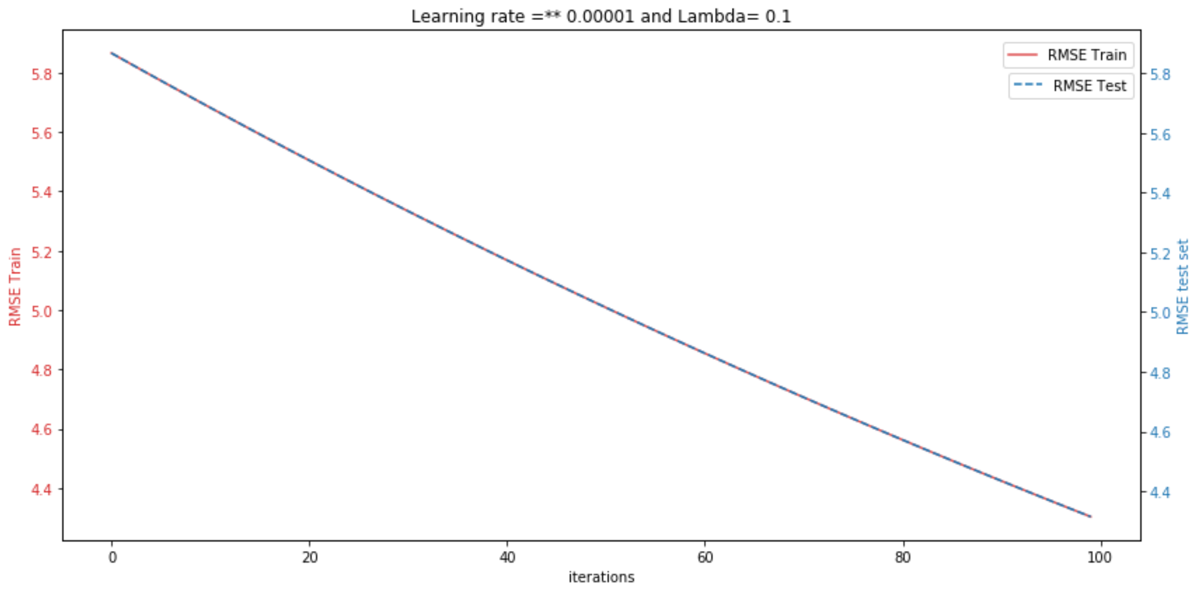
Learning rate = 0.00001 and **Lambda=** 0.1 | 100 iterations

```
y = wine_train['quality'].values
y = np.reshape(y, (len(y),1))
X = wine_train.drop(['quality'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))

y2 = wine_test['quality'].values
y2 = np.reshape(y2, (len(y2),1))
X2 = wine_test.drop(['quality'], axis=1).values
column_one = np.ones((X2.shape[0],1))
X2 = np.concatenate((column_one, X2), axis = 1)
```

```
betas, RMSE, RMSE_test3 = SGD(X, y, X2, y2, 0.00001, 0.1, 50, 100, beta)
print('betas', betas.T,'\n', 'RMSE',RMSE)
```

```
betas [[1.02288961 0.2869254  0.17124687 0.19822463 0.07573786 0.07786254
  0.10543773 0.25835414 0.14636624 0.39488471 0.178914   0.38209089]]
 RMSE [5.866135206694968, 5.847442898563746, 5.828816404290657, 5.810255804444567, 5.79176093408541, 5.773331183758878, 5.75496
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE, color=color, alpha=0.8)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test3.items()
x,y = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(x, y, '--', color=color, alpha=1)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
plt.title('Learning rate =** 0.00001 and Lambda= 0.1', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```

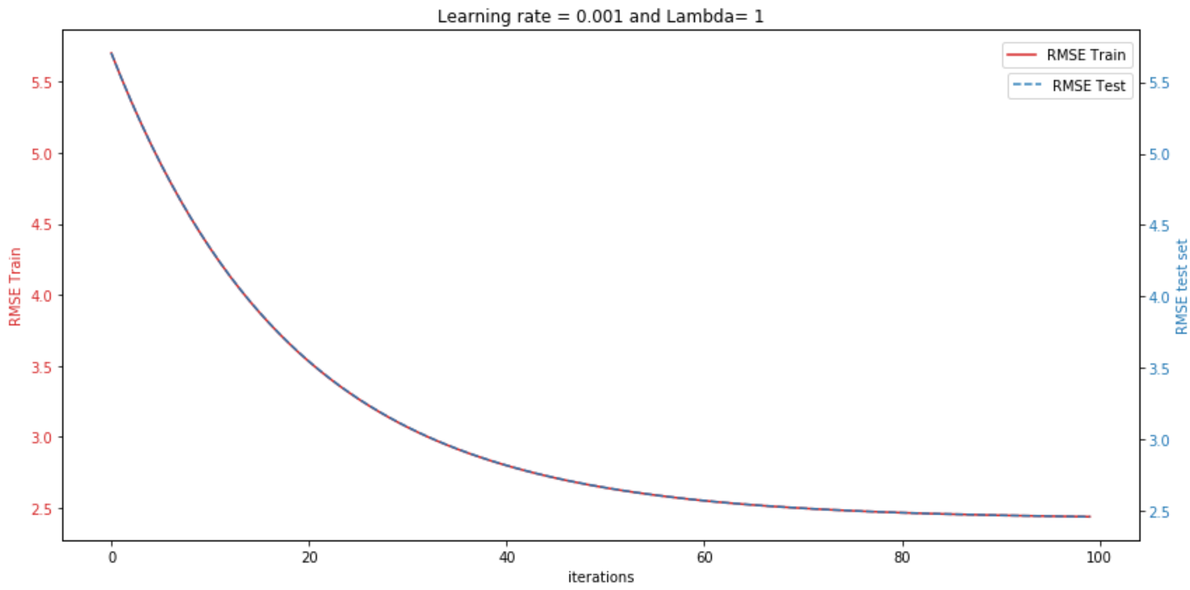


Learning rate = 0.001 and **Lambda=** 1 | 100 iterations

```
y = wine_train['quality'].values
y = np.reshape(y, (len(y),1))
X = wine_train.drop(['quality'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))

y2 = wine_test['quality'].values
y2 = np.reshape(y2, (len(y2),1))
X2 = wine_test.drop(['quality'], axis=1).values
column_one = np.ones((X2.shape[0],1))
X2 = np.concatenate((column_one, X2), axis = 1)

betas, RMSE, RMSE_test4 = SGD(X, y, X2, y2, 0.0001, 1, 50, 100, beta)
print('betas', betas.T,'\n', 'RMSE',RMSE)
```

```
betas [[2.25721711 0.62692277 0.36407548 0.43781133 0.167922   0.16713976
  0.23406029 0.56991746 0.31708071 0.86471821 0.39183006 0.85806246]]
 RMSE [5.701919843753534, 5.528588293140477, 5.364397728486257, 5.208885200356202, 5.061570900632905, 4.922043085214565, 4.78988
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test4.items()
xx,yy = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(xx, yy, '--', color=color, alpha=0.9)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
plt.title('Learning rate = 0.001 and Lambda= 1', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```



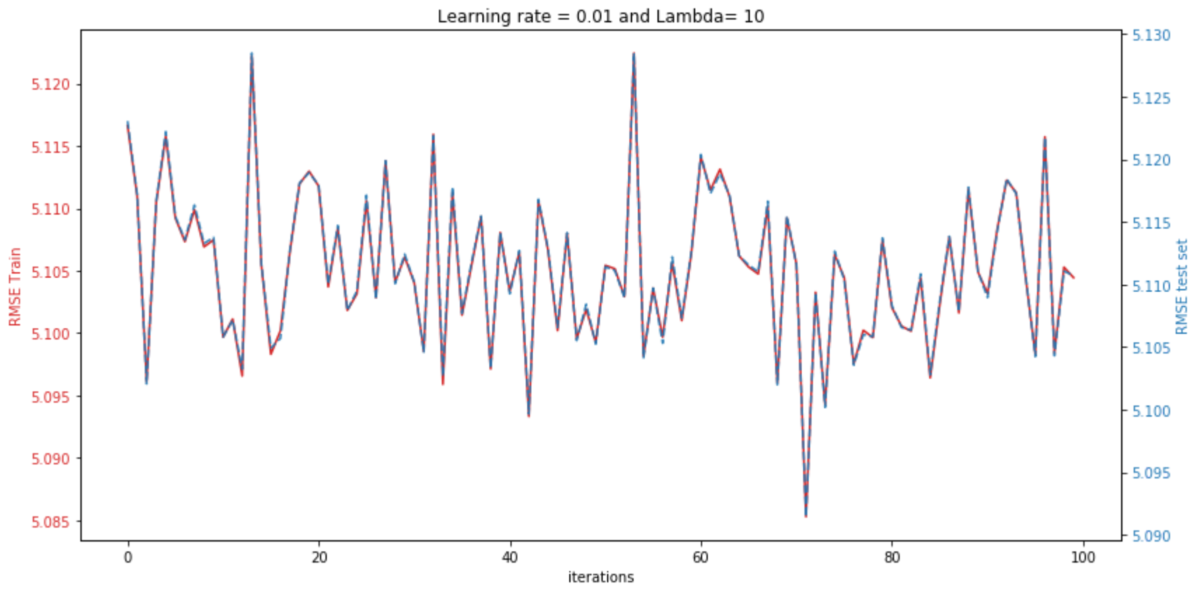
Learning rate = 0.01 and Lambda= 10 | 100 iterations

I chose a high number of lambda in order to appreciate the effect the data will experience if the regularization is added more than the optimal.

```
y = wine_train['quality'].values
y = np.reshape(y, (len(y),1))
X = wine_train.drop(['quality'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))

y2 = wine_test['quality'].values
y2 = np.reshape(y2, (len(y2),1))
X2 = wine_test.drop(['quality'], axis=1).values
column_one = np.ones((X2.shape[0],1))
X2 = np.concatenate((column_one, X2), axis = 1)

betas, RMSE, RMSE_test5 = SGD(X, y, X2, y2, 0.01, 10, 50, 100, beta)
print('betas', betas.T,'\n', 'RMSE',RMSE)
```

```
betas [[0.502376   0.14544759 0.08550248 0.09864669 0.03760521 0.03792005
  0.05354624 0.12828428 0.07358164 0.19329989 0.09045055 0.1854454 ]]
 RMSE [5.116639259322914, 5.111029798219598, 5.096107265123259, 5.1104640708230455, 5.115801653412504, 5.109234301573035, 5.107
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()
lista = RMSE_test5.items()
xxx,yyy = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(xxx, yyy, '--', color=color, alpha=0.9)
ax2.tick_params(axis='y', labelcolor=color)
plt.title('Learning rate = 0.01 and Lambda= 10', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```



3. HYPER PARAMETER TUNNING

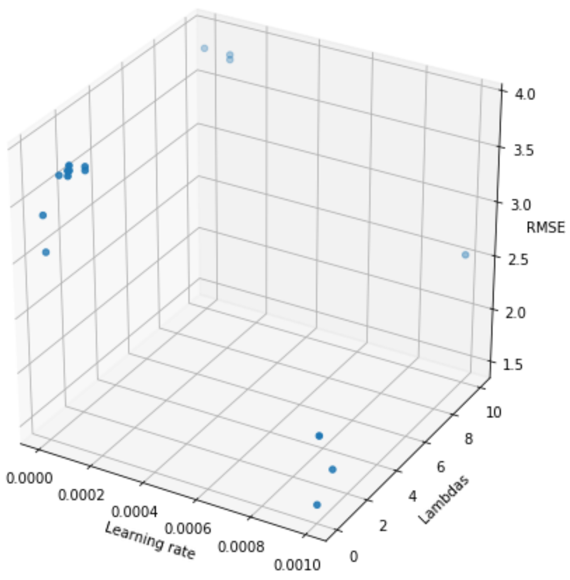
Hyper-parameter tuning and Cross validation

```
def cross_validation(data):
  a = [0.001, 0.000001, 0.0001, 0.0001]
  parameter = [0.1, 0.01, 1, 10]
  pairs = [[r, b] for r in a for b in parameter] # Pairing all learning rate and parameters possible variations
  RMSE_folds = []
  RMSE_avg = []
  RMSE_epoch = []
  hyperparameters = []
  beta = np.zeros(n)
  beta = np.reshape(beta, (len(beta),1))

  for i in pairs:
    loss = loss_function(X, y, beta, i[1])
    for j in range(0, len(data), (len(data)//5)): # Loop to define test and training set considering the folds.
      test = X[j:j+(len(X)//5)]
      for k in range(0, len(X)):
        if k != j:
          train = X[:k+(len(X))]
      y_train = data['quality'].values        # In the next 5 lines, X and Y are being defined.
      y_train = np.reshape(y_train, (len(y_train),1))

      X_train = data.drop(['quality'], axis=1).values
      column_one = np.ones((data.shape[0],1))
      X_train = np.concatenate((column_one, X_train), axis = 1)
      mini_batches = mini_batch(X_train, y_train, 50)
      for j in mini_batches:
        X_mini = j[0]
        y_mini = j[1]
        beta_hat = beta - i[0]*gradient(X_mini, y_mini, beta, i[1])
        loss_old = loss
        loss = loss_function(X_mini, y_mini, beta_hat, i[1])
        beta = beta_hat
      error = RMSE_function(X, y, beta_hat)
      RMSE_epoch.append(error)
    RMSE_folds = sum(RMSE_epoch)/len(RMSE_epoch)      # Average RMSE per fold.
    RMSE_avg.append(RMSE_folds)

    hyperparameters.append(i)
  values = list(zip(RMSE_avg, hyperparameters))
  optimum = min(values)
  return optimum[1], hyperparameters, RMSE_avg

optimum, hyperparameters, RMSE_avg = cross_validation(wine)

for i in hyperparameters:
  learning_rate = i[0]
  lambdas = i[1]

print('Optimum learning rate and lambda', optimum)
```

```
Optimum learning rate and lambda [0.001, 0.01]
```

> 3D representation of all the combinations of learning rate, lambdas and the Average RMSE.

```
param_final=[]
for i in RMSE_avg:
  param = i
  param_final.append(param)

lambdas1 = []
learning = []
for i in hyperparameters:
  learning_rate = i[0]
  lambdas = i[1]
  lambdas1.append(lambdas)
  learning.append(learning_rate)

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(learning, lambdas1, param_final, marker='o')
ax.set_xlabel('Learning rate')
ax.set_ylabel('Lambdas')
ax.set_zlabel('RMSE')
plt.show()
```

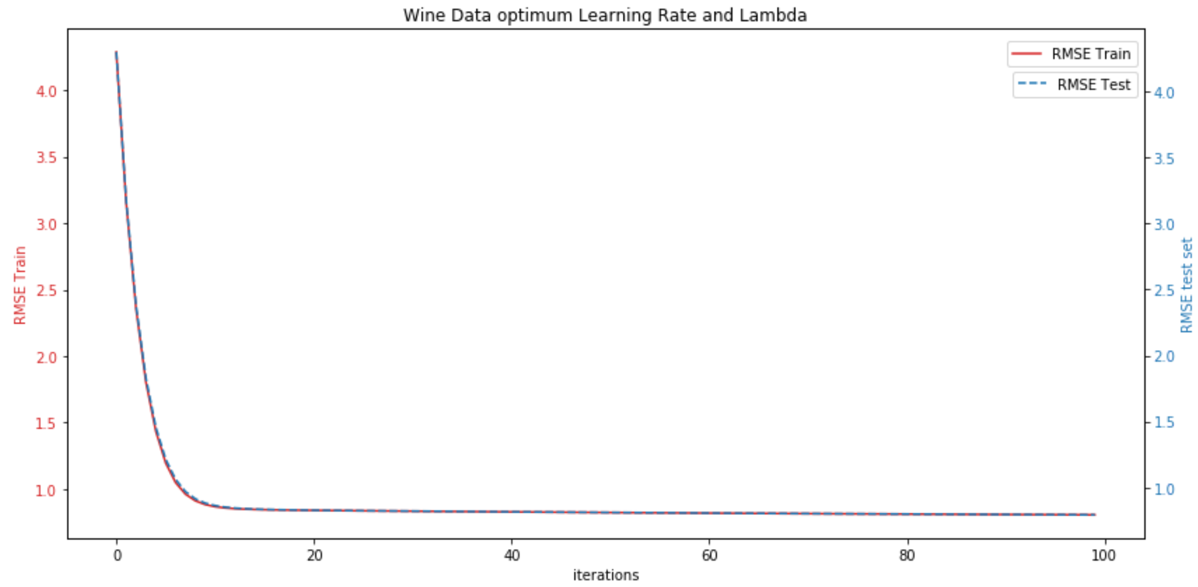


Training the model on complete training data and test data using the optimum learning rate and lambda.

```
y = wine_train['quality'].values
y = np.reshape(y, (len(y),1))
X = wine_train.drop(['quality'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
n = X.shape[1]
beta = np.zeros(n)
beta = np.reshape(beta, (len(beta),1))

y2 = wine_test['quality'].values
y2 = np.reshape(y2, (len(y2),1))
X2 = wine_test.drop(['quality'], axis=1).values
column_one = np.ones((X2.shape[0],1))
X2 = np.concatenate((column_one, X2), axis = 1)


betas, RMSE_wine, RMSE_test_wine = SGD(X, y, X2, y2, 0.001, 0.01, 50, 100, beta)
print('betas', betas.T,'\n', 'RMSE',RMSE)
```

```
betas [[3.77382786 0.83823722 0.12356294 0.73018653 0.32072088 0.12490287
  0.43642652 0.9201853  0.34279164 1.20401555 0.56318971 1.88307065]]
 RMSE [5.116639259322914, 5.111029798219598, 5.096107265123259, 5.1104640708230455, 5.115801653412504, 5.109234301573035, 5.1073
```

```
fig, ax1 = plt.subplots(1,1,figsize=(12,6))
color = 'tab:red'
ax1.set_xlabel('iterations')
ax1.set_ylabel('RMSE Train', color=color)
ax1.plot(range(100), RMSE_wine, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(['RMSE Train'], bbox_to_anchor=(1,0.99))

ax2 = ax1.twinx()
lista = RMSE_test_wine.items()
xs,ys = zip(*lista)

color = 'tab:blue'
ax2.set_ylabel('RMSE test set', color=color)
ax2.plot(xs, ys, '--',color=color, alpha=1)
ax2.tick_params(axis='y', labelcolor=color)
ax2.legend(['RMSE Test'], bbox_to_anchor=(1,0.93))
plt.title('Wine Data optimum Learning Rate and Lambda', fontdict=None, loc='center', pad=None)
fig.tight_layout()
plt.show()
```



4. CONCLUSIONS

1. The Wine quality dataset has a **good performance** on unseen data (test set). In other words, it is not incurring in overfitting.
2. The opposite effect happens with this dataset: since the data generalizes well it requires small bias to improve the model and reach convergence accurately.
3. The last combination of **learning rate** and **lambda** shows the consequences of adding more bias to a model when it is not required: Never reaches convergence and overfitting takes place.
4. By checking the previous conclusion, on exercise 3 an **ideal combination** must be selected. The output is a relative low learning rate and lambda. The number of iterations is reduced sharply while picking an optimal lambda and learning rate.

BIBLIOGRAPHY

> Stochastic Gradient Descent - Mini-batch and more - Adventures in Machine Learning. (2019). Retrieved 27 November 2019, from https://adventuresinmachinelearning.com/stochastic-gradient-descent/