code-copy

August 17, 2023

0.1 Assignment 1

0.1.1 Name: Bannuru Rohit Kumar Reddy

0.1.2 Roll Number: 21CS30011

Libraries required :

[57]:
```
# import all the necessary libraries here
import pandas as pd

# Library used for splitting the data
from sklearn.model_selection import train_test_split

# Library used for Matrices Calculations
import numpy as np

# Library used for Normalizing the data :
from sklearn.preprocessing import StandardScaler

# Library used for plotting graphs :
import matplotlib.pyplot as plt
```

0.2 Analytic Solution :

Loading the Dataset

[58]:
```
df = pd.read_csv('../../dataset/linear-regression.csv')
print(df.shape)
```

```
(1599, 12)
```

Splitting the dataset into training, test and validation :

[59]:
```
# Splitting into training, validation, and testing sets
train_ratio = 0.5
val_ratio = 0.3
test_ratio = 0.2



# Assuming df contains your DataFrame
```

```
train_data, temp_data = train_test_split(df, test_size=0.5, random_state=42)
test_data, valid_data  = train_test_split(temp_data, test_size=0.6,␣
 ↪random_state=42)

print("Train data shape:", train_data.shape)
print("Validation data shape:", valid_data.shape)
print("Test data shape:", test_data.shape)

# Observing the kind of data we are dealing with :
print(train_data.head())
```

```
Train data shape: (799, 12)
Validation data shape: (480, 12)
Test data shape: (320, 12)
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
1097            8.6              0.52         0.38             1.5      0.096
110             7.8              0.56         0.19             1.8      0.104
943             9.8              0.50         0.34             2.3      0.094
42              7.5              0.49         0.20             2.6      0.332
746             8.2              0.34         0.38             2.5      0.080

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
1097                  5.0                  18.0  0.99666  3.20       0.52
110                  12.0                  47.0  0.99640  3.19       0.93
943                  10.0                  45.0  0.99864  3.24       0.60
42                    8.0                  14.0  0.99680  3.21       0.90
746                  12.0                  57.0  0.99780  3.30       0.47

      alcohol  quality
1097      9.4        5
110       9.5        5
943       9.7        7
42       10.5        6
746       9.0        6
```

Converting the data into NumPy arrays and adding a column of 1's Note: The 1's are being added to implement the bias term (The constatn which appears in gradient descent)

[60]:
```
# Convert DataFrame to NumPy arrays

X_train = train_data.iloc[:, :-1].values
Y_train = train_data.iloc[:, -1].values

X_test = test_data.iloc[:, :-1].values
Y_test = test_data.iloc[:, -1].values

print("Train input features shape:", X_train.shape)
print("Train output feature shape:", Y_train.shape)

# Assuming you have X_train and Y_train defined
X = X_train
y = Y_train

# Add a column of ones to the input features matrix to represent the bias term
X_train = np.c_[np.ones((X.shape[0], 1)), X]
X_test = np.c_[np.ones((X_test.shape[0], 1)), X_test]
```

```
Train input features shape: (799, 11)
Train output feature shape: (799,)
```

Theta = (X^T * X)^-1 * X^T * y

where X is a matrix of size (m, n+1) where m is the number of training examples and n is the number of features. y is a vector of size (m, 1) and Theta is a vector of size (n+1, 1). f = Theta0 + Theta1 * x + … + ThetaN * x^N

[61]:
```
# Analytic solution :
# theta : (Xt*X)^-1*Xt*y

# Calculate X^T*X
X_transpose_X = np.dot(X_train.T, X_train)

# Calculate the inverse of X^T*X
X_transpose_X_inv = np.linalg.inv(X_transpose_X)

# Calculate theta
theta = np.dot(np.dot(X_transpose_X_inv, X_train.T), y)

print("Theta:", theta)
```

```
Theta: [ 1.03434328e+01  4.21803861e-03 -1.28736172e+00 -2.81156682e-01
  1.59731779e-02 -1.78832680e+00  3.08338676e-03 -3.04438457e-03
 -6.74416245e+00 -1.82876873e-01  6.77925417e-01  2.98166487e-01]
```

Calculating the RMSE, and R2 Errors for the test set to see hwo well the model is performing :

[62]:
```
def calculate_rmse(y_true, y_pred):
    """
    Calculate Root Mean Squared Error (RMSE).

    Parameters:
    y_true (numpy array): True values.
    y_pred (numpy array): Predicted values.

    Returns:
```

```python
    float: RMSE value.
    """
    squared_diff = np.square(y_true - y_pred)
    mean_squared_error = np.mean(squared_diff)

    print("Mean Squared Error of Test dataset:", mean_squared_error)
    rmse = np.sqrt(mean_squared_error)
    return rmse

def calculate_r2(y_true, y_pred):
    """
    Calculate the R-squared error.

    Parameters:
    y_true (array-like): The true values.
    y_pred (array-like): The predicted values.

    Returns:
    float: R-squared error.
    """
    mean_y_true = sum(y_true) / len(y_true)
    ss_total = sum((y - mean_y_true)**2 for y in y_true)
    ss_residual = sum((yt - yp)**2 for yt, yp in zip(y_true, y_pred))

    r2 = 1 - (ss_residual / ss_total)
    return r2


# Assuming you have theta and Xtest_with_bias defined
Y_pred = np.dot(X_test, theta)

# Calculate RMSE
rmse = calculate_rmse(Y_test, Y_pred)
print("Root Mean Squared Error of Test dataset:", rmse)

# Calculate R2 score
r2_score_analytical = calculate_r2(Y_test, Y_pred)
print("R2 Error of Test dataset: ", r2_score_analytical)
```

```
Mean Squared Error of Test dataset: 0.4496722602705587
Root Mean Squared Error of Test dataset: 0.6705760659839857
R2 Error of Test dataset:  0.2890118051432079
```

0.3 Gradient ascent :

Preprocessing the data

Includes Normalization

[63]:
```
data_df = pd.read_csv('../../dataset/linear-regression.csv')

# Split the dataset into training, validation, and test sets
train_data_df,temp_data_df = train_test_split(data_df, test_size=0.5,␣
 ↪random_state=42)
validation_data_df, test_data_df = train_test_split(temp_data_df, test_size=0.
 ↪4, random_state=42)

# Convert DataFrame to numpy arrays
train_data = train_data_df.to_numpy()
validation_data = validation_data_df.to_numpy()
test_data = test_data_df.to_numpy()

# Split data into X and y
X_train = train_data[:, :-1]
y_train = train_data[:, -1]
X_validation = validation_data[:, :-1]
y_validation = validation_data[:, -1]
X_test = test_data[:, :-1]
y_test = test_data[:, -1]

# normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_validation = scaler.transform(X_validation)
X_test = scaler.transform(X_test)

# Add a column of ones to X_train, X_validation, and X_test
X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
X_validation = np.hstack((np.ones((X_validation.shape[0], 1)), X_validation))
X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
```

Defining functions useful for performing gradient descent and analysis :

[64]:
```
# predictor function after finding theta
def y_predictor(X_test, theta):
    return np.matmul(X_test, theta)

# mean squared error function
def mean_squared_error(y_test, y_pred):
    return np.sum((y_pred - y_test)**2)/y_test.size

# root mean squared error function
def root_mean_squared_error(y_test, y_pred):
    return np.sqrt(np.mean((y_pred - y_test)**2))

# r2 score function
```

```
def r2_score(y_test, y_pred):
    return 1 - (np.sum((y_test - y_pred) ** 2) / np.sum((y_test - np.
 ↪mean(y_test)) ** 2))
```

Main Function :

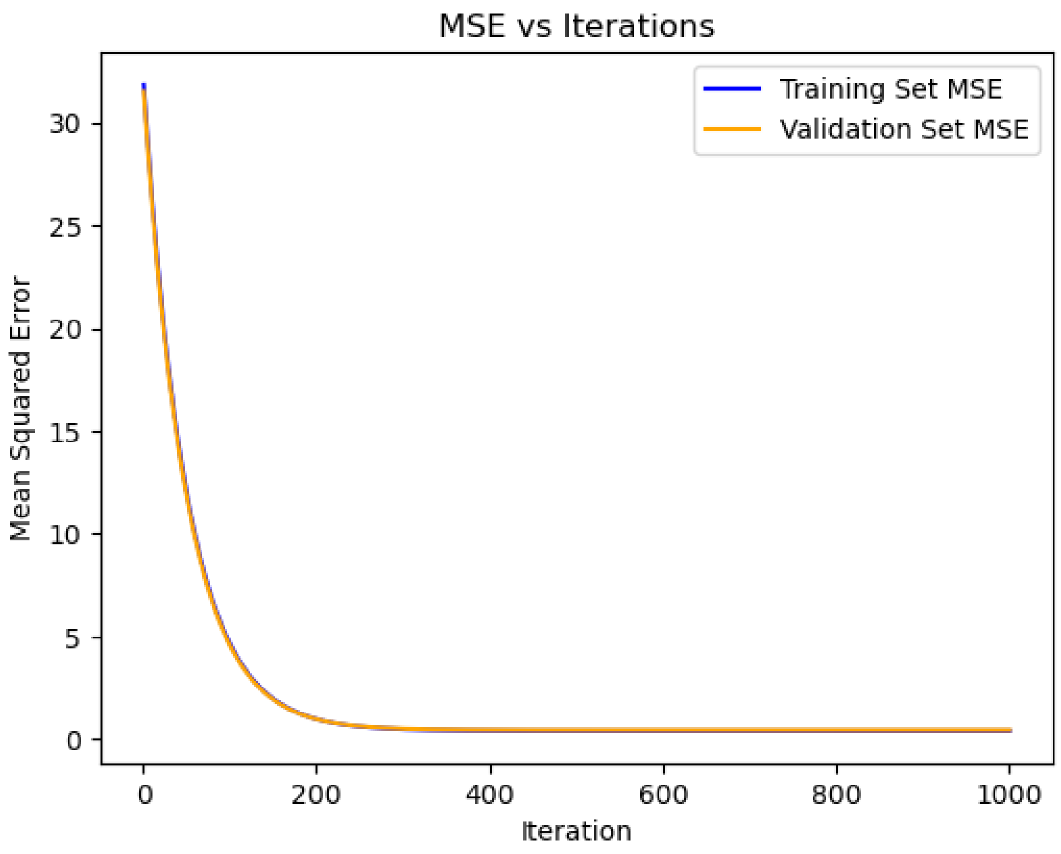
[65]:
```
# gradient descent function
def gradient_descent(X_train, y_train, X_validation, y_validation, alpha,␣
 ↪iterations):

    # Initializing lists to store the mse and iteration values for the plot
    validation_mse = []
    training_mse=[]
    iteration = []

    theta = np.zeros(X_train.shape[1])

    for i in range(iterations):

        # calculating the gradient term as an (n+1 X 1) matrix
        # gradient matrix= (X^T)*(X*theta-Y)

        # loss matrix=(X*theta-Y)
        loss=np.dot( X_train, theta) - y_train

        # X^T*loss
        gradient=np.dot(X_train.T,loss)/y_train.size

        # Directly repeating theta=theta-(alpha)gradient until convergence
        theta = theta - alpha * gradient

        # storing values of mse vs iteration for plotting the graph
        iteration.append(i+1)
        training_mse.append(mean_squared_error(y_train, np.dot(X_train, theta)))
        validation_mse.append(mean_squared_error(y_validation, np.
 ↪dot(X_validation,theta)))

    # Predicting the output using the calculated theta matrix and evaluating␣
 ↪errors for validation set data
    y_pred = y_predictor(X_validation, theta)

    # Plotting the predicted values against the actual values
    plt.plot(iteration, training_mse, label='Training Set MSE', color='blue')
    plt.plot(iteration, validation_mse, label='Validation Set MSE',␣
 ↪color='orange')
```

```
    plt.xlabel('Iteration')
    plt.ylabel('Mean Squared Error')
    plt.title('MSE vs Iterations')
    plt.legend()
    plt.show()

    return theta
```

0.3.1 Training on various values

Learning Rate = 0.01, epochs = 1000

[66]:
```
# Set hyperparameters
alpha = 0.01
iterations = 1000

# Perform gradient descent to learn theta
theta = gradient_descent(X_train, y_train, X_validation, y_validation, alpha,␣
 ↪iterations)

# Predicting the output using the calculated theta matrix and evaluating errors␣
 ↪for validation set data
y_pred = y_predictor(X_validation, theta)

mse = mean_squared_error(y_validation, y_pred)

# Calculating the root mean squared error
rmse = root_mean_squared_error(y_validation, y_pred)

# Calculating the R2 score
r2_score_val = r2_score(y_validation, y_pred)

print('Mean Squared Error1: ', mse)
print('Root Mean Squared Error1: ', rmse)
print('R2 Score1: ', r2_score_val)
```

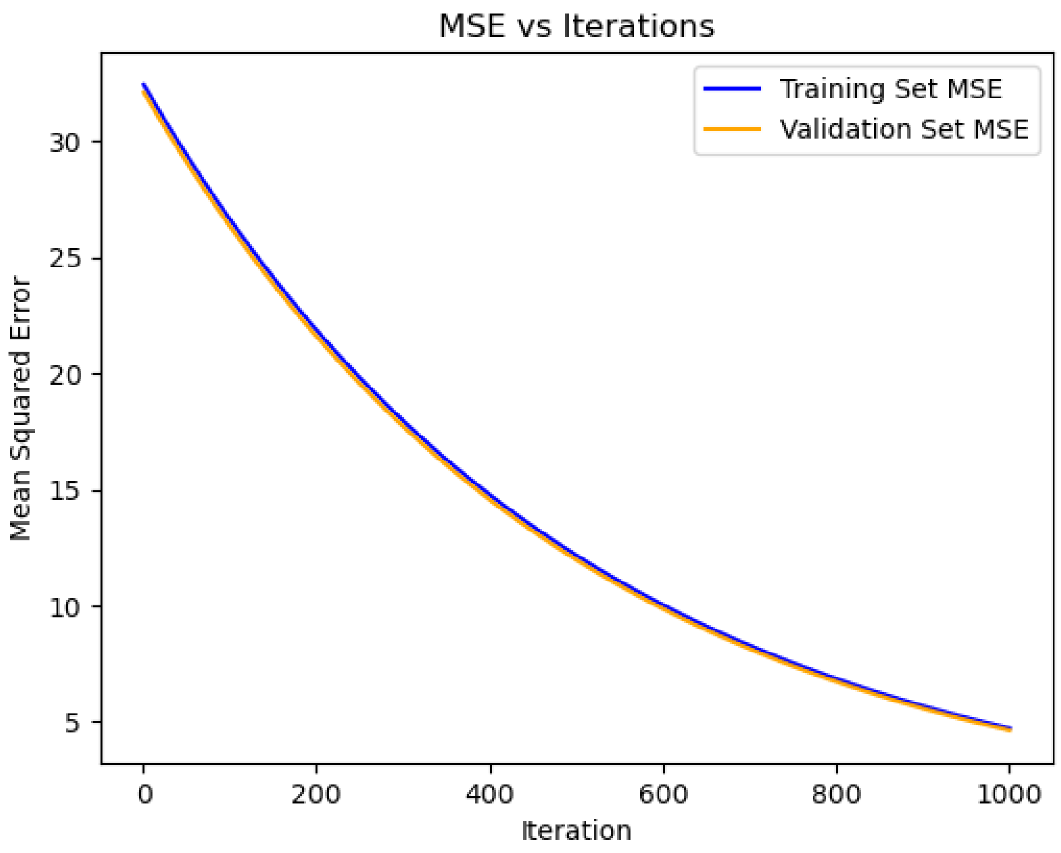


```
Mean Squared Error1:  0.4414484382303795
Root Mean Squared Error1:  0.6644158624162878
R2 Score1:  0.27795802924608537
```

Learning Rate = 0.001, epochs = 1000

[67]:
```
# Set hyperparameters
alpha = 0.001
iterations = 1000

# Perform gradient descent to learn theta
theta = gradient_descent(X_train, y_train, X_validation, y_validation, alpha,␣
  ↪iterations)

# Predicting the output using the calculated theta matrix and evaluating errors␣
  ↪for validation set data
y_pred = y_predictor(X_validation, theta)

mse = mean_squared_error(y_validation, y_pred)
```

```
# Calculating the root mean squared error
rmse = root_mean_squared_error(y_validation, y_pred)

# Calculating the R2 score
r2_score_val = r2_score(y_validation, y_pred)

print('Mean Squared Error1: ', mse)
print('Root Mean Squared Error1: ', rmse)
print('R2 Score1: ', r2_score_val)
```

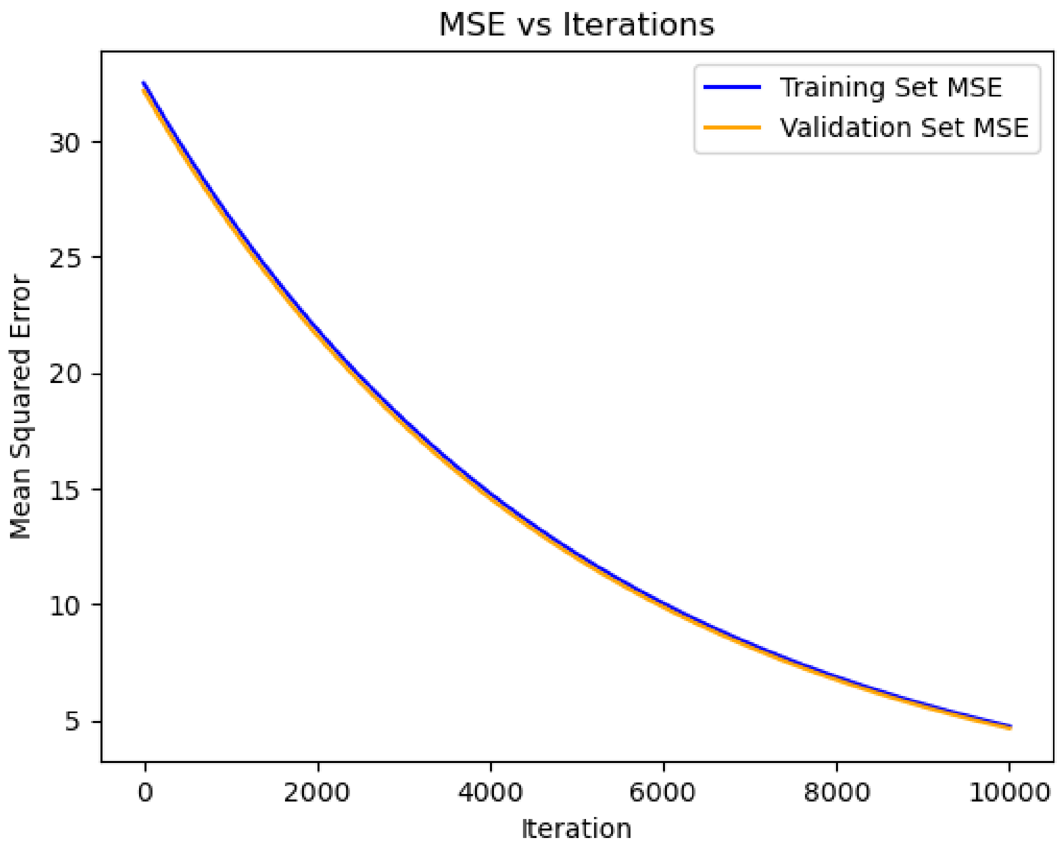


```
Mean Squared Error1:  4.632374444268015
Root Mean Squared Error1:  2.1522951573304288
R2 Score1:  -6.576805088307521
```

Learning Rate = 0.0001, epochs = 1000

[68]:
```
# Set hyperparameters
alpha = 0.0001
iterations = 10000

# Perform gradient descent to learn theta
```

```
theta = gradient_descent(X_train, y_train, X_validation, y_validation, alpha,␣
  ↪iterations)

# Predicting the output using the calculated theta matrix and evaluating errors␣
  ↪for validation set data
y_pred = y_predictor(X_validation, theta)

mse = mean_squared_error(y_validation, y_pred)

# Calculating the root mean squared error
rmse = root_mean_squared_error(y_validation, y_pred)

# Calculating the R2 score
r2_score_val = r2_score(y_validation, y_pred)

print('Mean Squared Error1: ', mse)
print('Root Mean Squared Error1: ', rmse)
print('R2 Score1: ', r2_score_val)
```



```
Mean Squared Error1:  4.636207974499204
```

```
Root Mean Squared Error1:  2.153185541122549
R2 Score1:  -6.583075287686112
```

0.4 Therefore we can see that the first Model performs the best