GROUP 21

- Sai Phani Ram Popuri : **2205577**
- Sandeep Potla : **2151524**
- Sai Suma Podila : **2149229**
- Manivardhan Reddy Pidugu : **2146807**
- Praveen : **ppraveen@uh.edu**

Homework 2:

Part 1

(a) Notice that we used 100 epochs which was waste of time and we could have stopped earlier since after about epoch 55 or so, the loss is not getting lower significantly. Modify the above code so that if the change in loss is less than 10%, you exit the iterations.

(b) The above class uses batch gradient descent to find the minimum of the loss function. Modify the original code and use the stochastic gradient descent instead. Iterate over many iterations and see how the RMSE changes. The graph of RMSE for the batch gradient descent is smooth and decreasing as the number of iterations increases. What can you say about the graph of RMSE when the stochastic gradient descent is used?

In [27]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

import warnings
warnings.filterwarnings('ignore')

df = pd.read_csv("/Users/beingrampopuri/Downloads/iris_dataset.csv")
df.head()
```

Out[27]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [28]:
```
np.random.seed(42)
```

In [29]:
```
df = df.iloc[:50][["sepal_length", "sepal_width"]]
df.head()
```

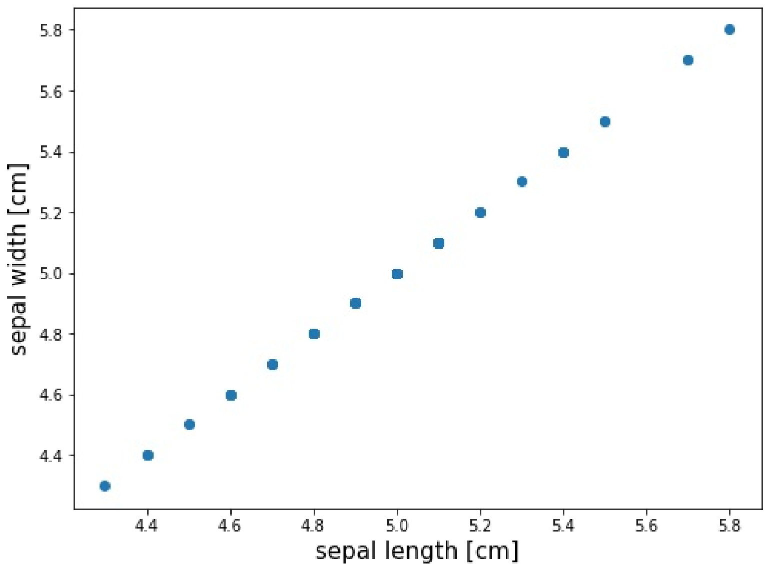
Out[29]:

	sepal_length	sepal_width
0	5.1	3.5
1	4.9	3.0
2	4.7	3.2
3	4.6	3.1
4	5.0	3.6

In [30]:
```
df.shape
```

Out[30]:
```
(50, 2)
```

In [31]:
```
# We create the scatter plot

plt.figure(figsize = (8, 6))

plt.scatter(df['sepal_length'], df['sepal_length'])
plt.xlabel("sepal length [cm]", fontsize = 15)
plt.ylabel("sepal width [cm]", fontsize = 15);
```



In [32]:
```python
# We compute covariance between the two variables
df.cov()
```

Out[32]:

	sepal_length	sepal_width
sepal_length	0.124249	0.100298
sepal_width	0.100298	0.145180

We compute the correlation between the Sepal_length and sepal_width

OBSERVATIONS

- Direction: Positively Correlated
- Strength: Medium

In [33]:
```python
df.corr()
```

Out[33]:

	sepal_length	sepal_width
sepal_length	1.00000	0.74678
sepal_width	0.74678	1.00000

Updated MyLinReg class

In [34]:
```python
class MyLinReg(object):
    """
    A class used to represent a single artificial neuron for linear regression.

    ...

    Attributes
    ----------
    activation_function : function
        The activation function applied to the preactivation linear function.

    theta : numpy.ndarray
        The weights and bias of the single neuron. The last entry being the bias.
        This attribute is created when the fit method is called.

    errors : list
        A list containing the mean squared error computed after each iteration
        of batch gradient descent.

    Methods
    -------
    fit(self, X, y, alpha = 0.001, epochs = 10)
        Iterates the batch gradient descent algorithm through each sample
        a total of epochs number of times with learning rate alpha. The data
        consists of the feature vector X and the associated target y.

    predict(self, X)
        Uses the weights and bias, the feature vector X, and the
        activation_function to make a prediction on each data instance.
    """
    def __init__(self, activation_function):
        self.activation_function = activation_function
```

```
        # Initialized a variable that will hold the prev_error (or) error obtained in the prev.iteration
        self.prev_errors = 0

    def fit(self, X, y, alpha = 0.001, epochs=10):
        self.theta = np.random.rand(X.shape[1] + 1)
        self.errors = []
        n = X.shape[0]

        for idx in range(epochs):
            errors = 0
            sum_1 = 0
            sum_2 = 0
            for xi, yi in zip(X, y):
                sum_1 += (self.predict(xi) - yi)*xi
                sum_2 += (self.predict(xi) - yi)
                errors += ((self.predict(xi) - yi)**2)

            self.theta[:-1] -= 2*alpha*sum_1/n
            self.theta[-1] -= 2*alpha*sum_2/n
            self.errors.append(errors/n)

            print('{} -> {}'.format(self.prev_errors, self.errors[idx]))

            '''
                The below 'if' condition verifies if the difference in error value is < 1 % of the previous.

                LOGIC:

                if the condition is met, then the epochs loop will break and program gets terminated.
                Else,
                prev_error = error in current iteration.

                where error_curr_iteration = self.error[-1] = Last appended element of the errors list.

            '''
            if (abs((self.prev_errors - self.errors[-1]) / self.prev_errors) < 0.01):
                print('Condition met, Breaking the loop.')
                print('\n')

                # Printing the final values of weights and the bias
                print('Weights: \n', self.theta[:-1])
                print('Intercept: ', self.theta[-1])
                break
            else:
                self.prev_errors = self.errors[-1]

        return self

    def predict(self, X):
        weighted_sum = np.dot(X, self.theta[:-1]) + self.theta[-1]
        return self.activation_function(weighted_sum)
```

In [35]:
```
X = df[['sepal_length']].to_numpy()
```

In [36]:
```
X.shape
```

Out[36]:
```
(50, 1)
```

In [37]:
```
y = df['sepal_width'].to_numpy()
```

In [38]:
```
# We instantiate an instance of MyLinReg class with identity activation function

def identity_function(z):
    return z

model = MyLinReg(identity_function)
model.fit(X, y, alpha = 0.001, epochs = 10)
```

```
0 -> 0.4365926345804007
0.4365926345804007 -> 0.3997479844259732
0.3997479844259732 -> 0.3666602722284338
0.3666602722284338 -> 0.33694641196084896
0.33694641196084896 -> 0.3102623799672599
0.3102623799672599 -> 0.28629923186562545
0.28629923186562545 -> 0.2647795255976996
0.2647795255976996 -> 0.24545410921200236
0.24545410921200236 -> 0.22809923618892428
0.22809923618892428 -> 0.2125139749092665
```

Out[38]:
```
<__main__.MyLinReg at 0x7fcaf0080070>
```

From the below code snippet, we are performing mainly three actions:

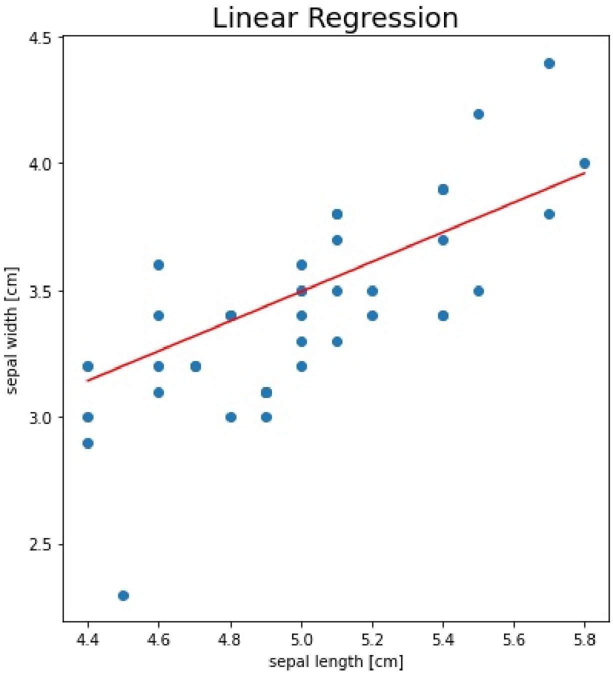
1. Fitting the values to the model.

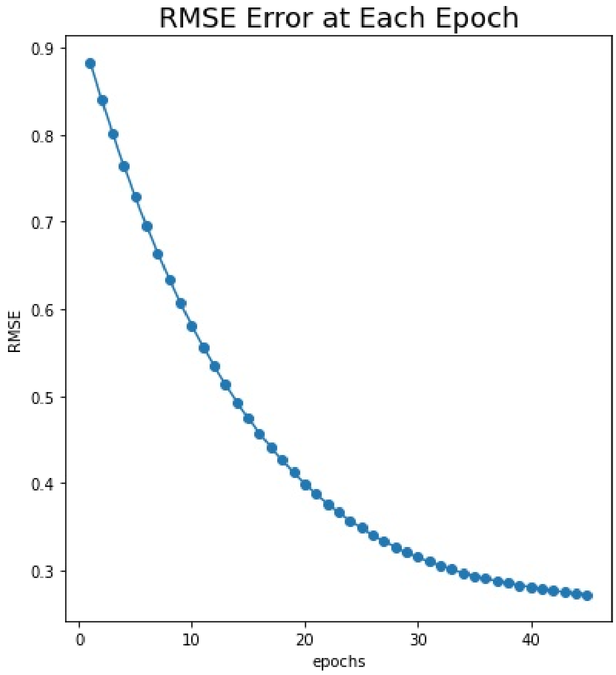
2. Printing the Prev_error and current error.
3. Plotting the graphs.

In [39]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [40]:
```
model = MyLinReg(identity_function)
print('PREV_ERROR -> CURRENT_ERROR')

'''
    Model Fitting
'''
model.fit(X_train, y_train, alpha = 0.001, epochs = 100)


domain_x = np.linspace(np.min(X_train), np.max(X_train), 2)
domain_y = model.predict(domain_x.reshape(-1, 1))


'''
    Plotting the Linear regression curve
'''

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14,7))

ax1.scatter(X_train, y_train)
ax1.plot(domain_x, domain_y, color = "red")
ax1.set_xlabel("sepal length [cm]")
ax1.set_ylabel("sepal width [cm]")
ax1.set_title("Linear Regression", fontsize = 18)

'''
    Plotting the graph of # of iterations Vs error
'''


ax2.plot(range(1, len(model.errors) + 1),
         np.sqrt(model.errors),
         marker = "o")
ax2.set_xlabel("epochs")
ax2.set_ylabel("RMSE")
ax2.set_xticks(range(0, len(model.errors) + 1, 10))
ax2.set_title("RMSE Error at Each Epoch", fontsize = 18);
```

```
PREV_ERROR -> CURRENT_ERROR
0 -> 0.7800579304377213
0.7800579304377213 -> 0.7072090477573986
0.7072090477573986 -> 0.641816476578103
0.641816476578103 -> 0.583117038712654
0.583117038712654 -> 0.5304256697899958
0.5304256697899958 -> 0.48312742404730724
0.48312742404730724 -> 0.44067029745811537
0.44067029745811537 -> 0.4025587854370364
0.4025587854370364 -> 0.3683480999347778
0.3683480999347778 -> 0.3376389784326265
0.3376389784326265 -> 0.3100730242535547
0.3100730242535547 -> 0.2853285238079173
0.2853285238079173 -> 0.2631166919579189
0.2631166919579189 -> 0.24317830168146135
0.24317830168146135 -> 0.22528065870106592
0.22528065870106592 -> 0.20921488576954922
0.20921488576954922 -> 0.1947934849180601
0.1947934849180601 -> 0.18184814921610365
0.18184814921610365 -> 0.17022779850517966
0.17022779850517966 -> 0.1597968161815918
0.1597968161815918 -> 0.1504334664503752
0.1504334664503752 -> 0.1420284735785327
0.1420284735785327 -> 0.13448374656640746
0.13448374656640746 -> 0.12771123435316636
0.12771123435316636 -> 0.12163189819579567
0.12163189819579567 -> 0.11617478922851239
0.11617478922851239 -> 0.11127622043702337
0.11127622043702337 -> 0.10687902338396604
0.10687902338396604 -> 0.10293188101096004
0.10293188101096004 -> 0.09938872873058177
0.09938872873058177 -> 0.09620821681855751
0.09620821681855751 -> 0.09335322783189952
0.09335322783189952 -> 0.09079044342089197
0.09079044342089197 -> 0.08848995547930816
0.08848995547930816 -> 0.08642491709468916
0.08642491709468916 -> 0.08457122922501911
0.08457122922501911 -> 0.08290725944508127
0.08290725944508127 -> 0.08141358948005523
0.08141358948005523 -> 0.08007278857988782
0.08007278857988782 -> 0.07886921008954803
0.07886921008954803 -> 0.0777888088409898
0.0777888088409898 -> 0.07681897723565315
0.07681897723565315 -> 0.0759483981044641
0.0759483981044641 -> 0.07516691262810073
0.07516691262810073 -> 0.07446540177605879
Condition met, Breaking the loop.


Weights:
 [0.58488473]
Intercept:  0.5689567118846798
```





Predictions using X_test

In [41]:
```
predictions = model.predict(X_test)
```

In [42]:
```
predictions
```

Out[42]:
```
array([3.08396105, 3.55186884, 3.37640342, 3.37640342, 3.55186884,
       3.66884578, 3.49338036, 3.49338036, 3.61035731, 3.55186884])
```

Actual Values of the test dataset.

In [43]:
```
y_test
```

Out[43]:
```
array([3. , 3.4, 3.1, 3. , 3.5, 3.7, 3.4, 3. , 4.1, 3.8])
```

In [44]:
```
result = pd.DataFrame()
result['Actual Value'] = y_test
result['Predicted Value'] = predictions
result['Residue'] = result['Actual Value'] - result['Predicted Value']
result
```

Out[44]:

	Actual Value	Predicted Value	Residue
0	3.0	3.083961	-0.083961
1	3.4	3.551869	-0.151869
2	3.1	3.376403	-0.276403
3	3.0	3.376403	-0.376403
4	3.5	3.551869	-0.051869
5	3.7	3.668846	0.031154
6	3.4	3.493380	-0.093380
7	3.0	3.493380	-0.493380
8	4.1	3.610357	0.489643
9	3.8	3.551869	0.248131

Part B of the 1st Question

NOTES

- Unlike in Gradient Descent, Stochastic Gradient Descent updates the parameters after passing through each record.
- The loss function will not be as smooth in case of the Gradient Descent rather in a Zig-Zag manner.
- Relatively faster in convergence when compared to Gradient descent.

In [45]:
```
class Stochastic_Grad_Descent(object):

    def __init__(self, activation_function):
        self.activation_function = activation_function

    def fit(self, X, y, alpha = 0.001, epochs=10):
        self.theta = np.random.rand(X.shape[1] + 1)
        self.errors = []
        n = X.shape[0]

        for idx in range(epochs):
            errors = 0
            sum_1 = 0
            sum_2 = 0
            for xi, yi in zip(X, y):
                sum_1 = (self.predict(xi) - yi)*xi
                sum_2 = (self.predict(xi) - yi)
                errors = ((self.predict(xi) - yi)**2)

                self.theta[:-1] -= 2*alpha*sum_1
                self.theta[-1] -= 2*alpha*sum_2
                self.errors.append(errors)

        return self

    def predict(self, X):
        weighted_sum = np.dot(X, self.theta[:-1]) + self.theta[-1]
        return self.activation_function(weighted_sum)
```

In [46]:
```
X = df[['sepal_length']].to_numpy()
```

In [47]:
```
y = df['sepal_width'].to_numpy()
```

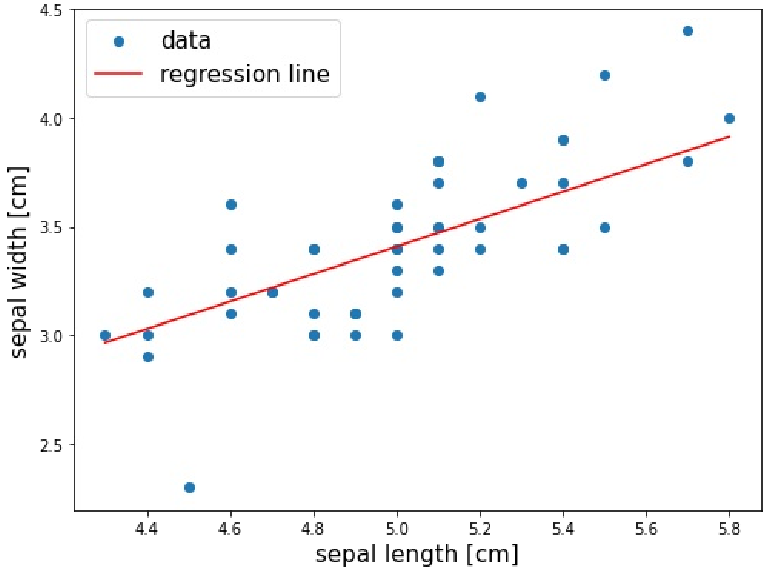
In [48]:
```
def identity_function(z):
    return z

model = Stochastic_Grad_Descent(identity_function)
model.fit(X, y, alpha = 0.001, epochs = 10)
```

Out[48]: <__main__.Stochastic_Grad_Descent at 0x7fcaf006dc40>

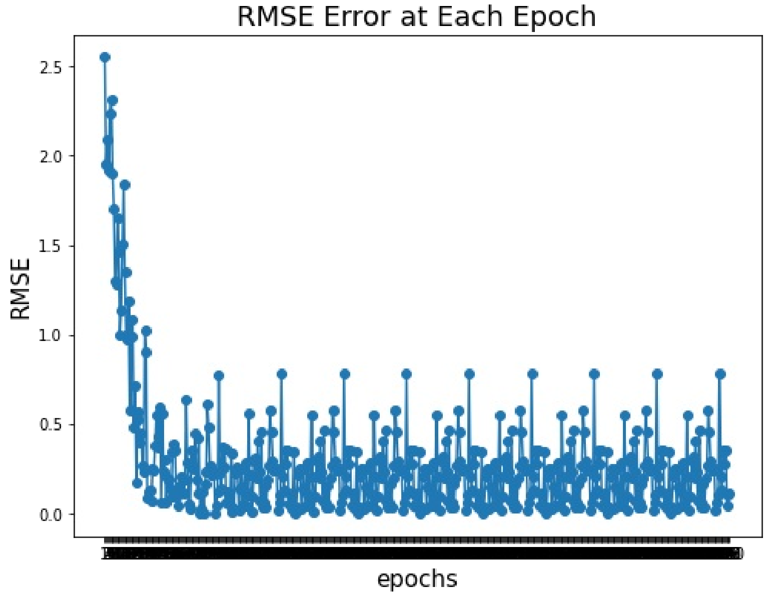
In [49]:
```
domain_x.reshape(-1, 1)
```

Out[49]:
```
array([[4.4],
       [5.8]])
```

In [50]:
```
domain_x = np.linspace(np.min(X), np.max(X), 5)
domain_y = model.predict(domain_x.reshape(-1, 1))

plt.figure(figsize = (8, 6))

plt.scatter(X, y, label = "data")
plt.plot(domain_x, domain_y, color="red", label ="regression line")
plt.xlabel("sepal length [cm]", fontsize = 15)
plt.ylabel("sepal width [cm]", fontsize = 15)
plt.legend(fontsize=15);
```

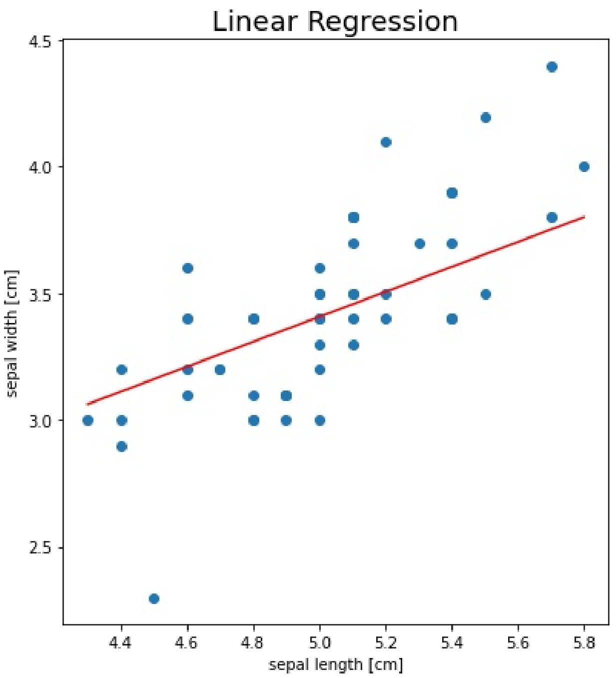


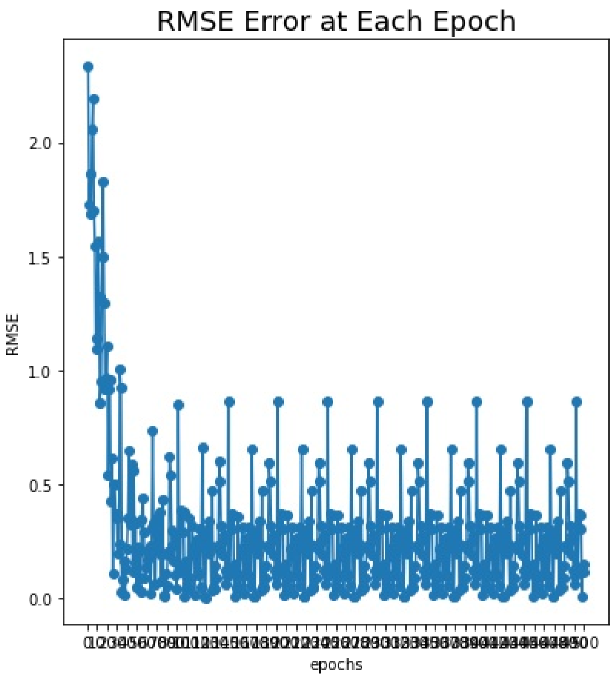
In [51]:
```
plt.figure(figsize = (8, 6))

plt.plot(range(1, len(model.errors) + 1),
         np.sqrt(model.errors),
         marker = "o")
plt.xlabel("epochs", fontsize = 15)
plt.ylabel("RMSE", fontsize = 15)
plt.xticks(range(1, len(model.errors) + 1))
plt.title("RMSE Error at Each Epoch", fontsize = 18);
```



In [52]:
```
model = Stochastic_Grad_Descent(identity_function)
model.fit(X, y, alpha = 0.001, epochs = 10)

domain_x = np.linspace(np.min(X), np.max(X), 2)
domain_y = model.predict(domain_x.reshape(-1, 1))


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14,7))

ax1.scatter(X, y)
```

```
ax1.plot(domain_x, domain_y, color = "red")
ax1.set_xlabel("sepal length [cm]")
ax1.set_ylabel("sepal width [cm]")
ax1.set_title("Linear Regression", fontsize = 18)

ax2.plot(range(1, len(model.errors)+1),
         np.sqrt(model.errors),
         marker = "o")
ax2.set_xlabel("epochs")
ax2.set_ylabel("RMSE")
ax2.set_xticks(range(0, len(model.errors) + 1, 10))
ax2.set_title("RMSE Error at Each Epoch", fontsize = 18);
```





Observations

- The graph resembles a zig-zag pattern rather a smoother decay.
- It is because, the error values keeps on fluctuating up and down.
- On an average, it does converges to the minimal error value after a certain number of iterations.

Processing math: 100%

GROUP 21

- Sai Phani Ram Popuri : **2205577**
- Sandeep Potla : **2151524**
- Sai Suma Podila : **2149229**
- Manivardhan Reddy Pidugu : **2146807**
- Praveen : **ppraveen@uh.edu**

Importing Essential Libraries

In [65]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.linear_model import SGDRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import validation_curve

from sklearn.model_selection import train_test_split

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, StandardScaler

ads_data = pd.read_csv("/Users/beingrampopuri/Downloads/advertising.csv")

import warnings
warnings.filterwarnings('ignore')
```

In [66]:
```
np.random.seed(42)
```

Function to detect Outliers

In [67]:
```
def outliers(df, feature):

    Q1 = df[feature].quantile(0.25)
    Q3 = df[feature].quantile(0.75)
    IQR = Q3 - Q1

    lowerBound = Q1 - 1.5*IQR
    upperBound = Q3 + 1.5*IQR

    outlier_indices = df.index[(df[feature] < lowerBound) | (df[feature] > upperBound)]

    return outlier_indices
```

Function to replace Outliers with desired Central tendency

In [68]:
```
def replace_outliers(df, feature, outlier_list, measure):
    replacement = 0.0
    if measure == 'Mean':
        replacement = df[feature].mean()
    elif(measure == 'Median'):
        replacement = df[feature].median()
    else:
        replacement = df[feature].mode()
    for idx in outlier_list:
        df[feature][idx] = replacement
    return
```

- No Outliers present in the 'TV' column

In [69]:
```
TV_outliers = outliers(ads_data, 'TV')
TV_outliers
```

Out[69]:
```
Int64Index([], dtype='int64')
```

- No outliers present in the 'Radio' Column

In [70]:
```
Radio_outliers = outliers(ads_data, 'Radio')
Radio_outliers
```

Out[70]: Int64Index([], dtype='int64')

There are two outliers present in the 'Newspaper' column

- We have to replace this with one of (Mean, Median, Mode)
- Mean is not a good measure when there are outliers present.
- Mode is to be used in case of Categorical variables.

- When there is a skewness observed in the distribution of a feature data, Median best fits the purpose.

In [71]:
```
Newspaper_outliers = outliers(ads_data, 'Newspaper')
Newspaper_outliers
```

Out[71]: Int64Index([16, 101], dtype='int64')

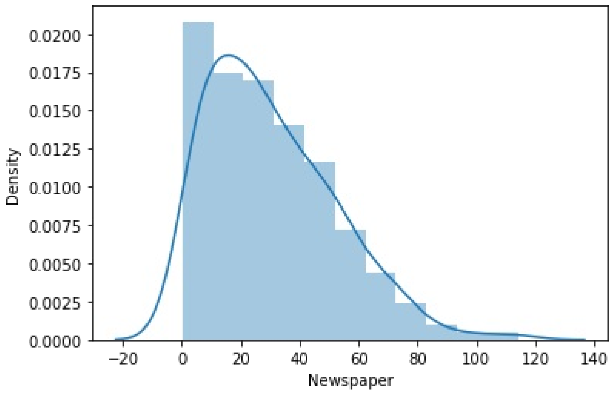
Plotting the 'Newspaper' column data distribution.

Observations

- Data is skewed to the left.
- Since there is a presence of outliers, Median is chosen to replace those values.

In [72]:
```
sns.distplot(ads_data['Newspaper'])
```

Out[72]: <AxesSubplot:xlabel='Newspaper', ylabel='Density'>



Replacing the outliers

In [73]:
```
replace_outliers(ads_data, 'Newspaper', Newspaper_outliers, 'Median')
```

In [74]:
```
Newspaper_outliers = outliers(ads_data, 'Newspaper')
Newspaper_outliers
```

Out[74]: Int64Index([], dtype='int64')

In [79]:
```
X = ads_data[["TV", "Radio"]]
y = ads_data[["Sales"]]
```

Fitting the data without Scaling

In [80]:
```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.15, random_state=42)

scaler = MinMaxScaler()
scaler.fit(x_train, y_train)
```

Out[80]: MinMaxScaler()

In [81]:
```
sgd = SGDRegressor(random_state = 42)
sgd.fit(x_train, y_train)
```

Out[81]: SGDRegressor(random_state=42)

Notice that the weights and the bias are out of context. Very unusual.

In [82]:
```
w = sgd.coef_
b = sgd.intercept_
```

```
w, b
```

Out[82]:

```
(array([ 1.43645723e+11, -1.66150291e+11]), array([4.321975e+10]))
```

Finding the Mean Absolute Error for training and test set.

In [83]:

```
mean_abs_train = mean_absolute_error(y_train, sgd.predict(x_train))
mean_abs_test = mean_absolute_error(y_test, sgd.predict(x_test))

print('Mean Abs Error Train: ', mean_abs_train)
print('Mean Abs Error Test: ', mean_abs_test)
```

```
Mean Abs Error Train:  18279520296188.008
Mean Abs Error Test:  15914415475536.998
```

In [84]:

```
mean_squared_train = mean_squared_error(y_train, sgd.predict(x_train))
mean_squared_test = mean_squared_error(y_test, sgd.predict(x_test))

print(f"RMSE on the training data: {np.sqrt(mean_squared_train)}\n")
print(f"RMSE on the test data: {np.sqrt(mean_squared_test)}")
```

```
RMSE on the training data: 21617873498944.027

RMSE on the test data: 19343184248413.812
```

Note that the mean absolute error is too high in case of without scaling.

- SGD is sensitive to feature scaling.
- In the below code snippet we are trying to add a feature scaling step that reduces the error.

Scaling

In [85]:

```
ads_data[['TV', 'Radio']] = Min_Max_Scaler.fit_transform(ads_data[['TV', 'Radio']])
```

In [86]:

```
ads_data.head()
```

Out[86]:

	TV	Radio	Newspaper	Sales
0	0.775786	0.762097	69.2	22.1
1	0.148123	0.792339	45.1	10.4
2	0.055800	0.925403	69.3	12.0
3	0.509976	0.832661	58.5	16.5
4	0.609063	0.217742	58.4	17.9

In [87]:

```
ads_data.head()
```

Out[87]:

	TV	Radio	Newspaper	Sales
0	0.775786	0.762097	69.2	22.1
1	0.148123	0.792339	45.1	10.4
2	0.055800	0.925403	69.3	12.0
3	0.509976	0.832661	58.5	16.5
4	0.609063	0.217742	58.4	17.9

In [88]:

```
X = ads_data.iloc[:, 0:2]
y = ads_data.iloc[:, 3]
```

In [89]:

```
sgd_Regressor = SGDRegressor()
```

In [90]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

In [91]:

```
sgd_Regressor.fit(X_train, y_train)
```

Out[91]:

```
SGDRegressor()
```

In [92]:

```
w = sgd_Regressor.coef_
b = sgd_Regressor.intercept_
w, b
```

Out[92]:

```
(array([14.55611555,  4.94403729]), array([5.82277741]))
```

In [93]:

```
print(f"The linear regression model based on the training data is \n")
print(f"predicted_sales = {np.round(w[0],3)} * TV + {np.round(w[1],3)} * Radio + {np.round(b,3)}")
```

```
The linear regression model based on the training data is

predicted_sales = 14.556 * TV + 4.944 * Radio + [5.823]
```

Notice that there is a significant decrease in the error values.

In [94]:
```python
mse_train = mean_squared_error(y_train, sgd_Regressor.predict(X_train))
mse_test = mean_squared_error(y_test, sgd_Regressor.predict(X_test))

print(f"RMSE on the training data: {np.sqrt(mse_train)}\n")
print(f"RMSE on the test data: {np.sqrt(mse_test)}")
```

```
RMSE on the training data: 1.736212179500144

RMSE on the test data: 1.6931557303863511
```

In [45]:
```python
predictions = sgd_Regressor.predict(X_train)
predictions
```

Out[45]:
```
array([20.83092201, 16.98209946, 13.08562483, 10.46983514, 20.49816964,
        8.35172971, 22.99065694,  7.92898758, 12.89250263, 10.94663009,
       11.69469853,  9.74616642, 16.86531641, 17.82863955, 15.65632746,
       18.1560006 , 15.74474228, 19.26238095, 17.16189635, 21.46597565,
       11.22093485, 14.56436992, 11.19878474, 18.05349298, 10.59063009,
       17.13105593, 13.58123738, 23.28181505, 11.83876989, 22.89710391,
        8.03711083, 18.84841078, 23.98459618, 20.8480349 , 18.74516618,
       16.7062133 , 14.10901218, 11.89679293, 19.26639286, 15.41718098,
       15.76495864, 10.73089153, 19.94626756, 12.88860933, 21.03388961,
       12.00610823,  9.63084616, 19.10759099, 15.97239795, 18.33038211,
       10.42885453, 20.87604903, 23.92319895, 18.17587404, 18.63579802,
       15.29954235, 16.50917374,  9.597705  , 17.50382117, 20.72948784,
       17.97639279,  6.48877496,  7.1617715 , 14.75090875, 19.07236223,
       18.70646869, 20.39764391, 22.63501576, 17.99699848, 20.76289973,
       18.73973319, 21.72604023, 12.47680368, 17.13455989, 16.80850305,
       10.49720683, 17.36224435, 17.20179062, 19.37838534, 19.941743  ,
       13.7592767 , 19.63098862, 20.36956403, 10.2576534 , 12.21810734,
       10.64722558, 13.66340531, 17.71731357,  7.35996143, 13.59744183,
       21.01851667, 17.07978772, 15.96039275,  6.43692835, 11.35304443,
       19.09627636, 15.96970787, 19.12563638, 23.57611186, 10.40814313,
       24.28103989, 11.46206967, 16.48494721, 24.21933025, 13.79923027,
       21.5817974 , 10.27725612, 15.93186596, 10.92203184, 21.10266003,
       18.54288271, 12.68336978, 17.97864068, 14.40979783, 16.08400754,
        7.05867431,  9.92881195, 14.39493754, 13.29568373, 18.18377279,
       11.90522841, 20.59119067, 10.57078544,  9.78281628, 12.5874696 ,
       16.7511289 , 16.60111015, 14.0692941 , 15.28767338, 18.74103095,
        8.88982734, 18.06671903, 19.29540996, 21.24305769, 12.62460977,
        8.12902962, 19.11051654, 19.83855126, 14.93838653, 20.57966462])
```

Base Variant of SGDRegressor Pipeline

In [95]:
```python
sgd_pipeline = Pipeline([("feature scaling", MinMaxScaler()), ("sgd", SGDRegressor())])
sgd_pipeline.fit(X_train, y_train)
```

Out[95]:
```
Pipeline(steps=[('feature scaling', MinMaxScaler()), ('sgd', SGDRegressor())])
```

In [96]:
```python
mse_train = mean_squared_error(y_train, sgd_pipeline.predict(X_train))
mse_test = mean_squared_error(y_test, sgd_pipeline.predict(X_test))

print(f"RMSE on the training data: {np.sqrt(mse_train)}\n")
print(f"RMSE on the test data: {np.sqrt(mse_test)}")
```

```
RMSE on the training data: 1.7342190466316927

RMSE on the test data: 1.688588328933666
```

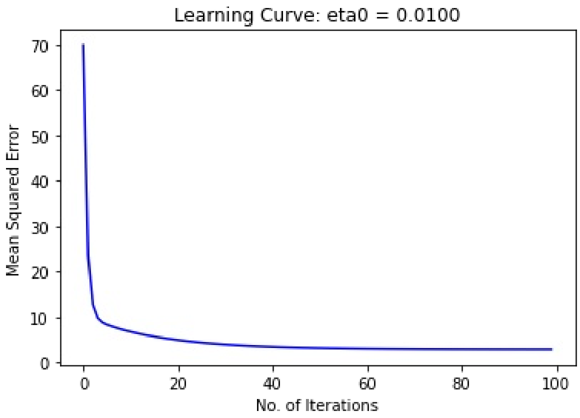
Even after scaling the data, the error is still high as per the industry standards.

- Follow the below steps to minimize the error.
- **STEP 1**: Instantiate the SGDRegressor( ) with warm_start = true and tolerance = - infinity.
- **STEP 2**: Train SGD Step by Step and record the loss in each step.
- **STEP 3**: Plot the learning curves and see if there are any issues with the data.

In [97]:
```python
eta0 = 1e-2

sgd_pipeline = Pipeline([("feature scaling", MinMaxScaler()),
                          ("sgd", SGDRegressor(max_iter = 1, early_stopping = True,
                                                 tol = -np.infty, warm_start = True, random_state=42))])
loss = []

for epoch in range(100):
    sgd_pipeline.fit(X_train, y_train)
    loss.append(mean_squared_error(y_train, sgd_pipeline.predict(X_train)))
```

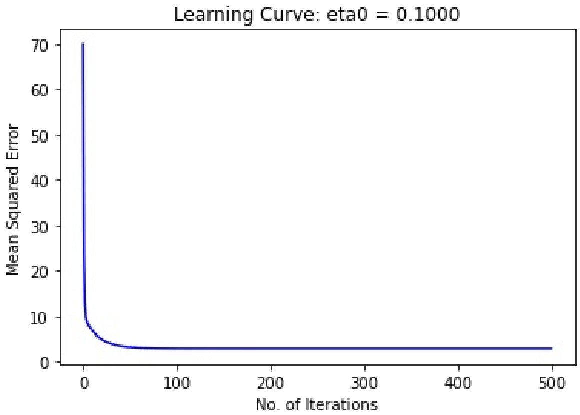
```
plt.plot(np.arange(len(loss)), loss, 'b-')
plt.xlabel('No. of Iterations')
plt.ylabel('Mean Squared Error')
plt.title(f'Learning Curve: eta0 = {eta0:.4f}')
```

Out[97]: Text(0.5, 1.0, 'Learning Curve: eta0 = 0.0100')



In [98]:
```
eta0 = 1e-1

sgd_pipeline = Pipeline([("feature scaling", MinMaxScaler()),
                          ("sgd", SGDRegressor(max_iter = 1, early_stopping = True,
                                               tol = -np.infty, warm_start = True, random_state=42))])
loss = []

for epoch in range(500):
    sgd_pipeline.fit(X_train, y_train)
    loss.append(mean_squared_error(y_train, sgd_pipeline.predict(X_train)))

plt.plot(np.arange(len(loss)), loss, 'b-')
plt.xlabel('No. of Iterations')
plt.ylabel('Mean Squared Error')
plt.title(f'Learning Curve: eta0 = {eta0:.4f}')
```

Out[98]: Text(0.5, 1.0, 'Learning Curve: eta0 = 0.1000')



- This is an ideal learning curve, where the training loss reduces monotonically as the training progresses.

In [99]:
```
print('No. of. iterations before reaching the convergence criteria: ', sgd_pipeline[-1].n_iter_)
print('No. of Weight updates: ', sgd_pipeline[-1].t_)
```

```
No. of. iterations before reaching the convergence criteria:  1
No. of Weight updates:  141.0
```

In [100…
```
mse_train = mean_squared_error(y_train, sgd_pipeline.predict(X_train))
mse_test = mean_squared_error(y_test, sgd_pipeline.predict(X_test))

print(f"RMSE on the training data: {np.sqrt(mse_train)}\n")
print(f"RMSE on the test data: {np.sqrt(mse_test)}")
```

```
RMSE on the training data: 1.6937727049258322

RMSE on the test data: 1.539532701664419
```

Key points

Fixing the learning rates through validation curves

- Step 1: Provide the list of values to be tried for Hyper-parameters

- Step 2: Instantiate an object of validation_curve with estimator, training features and label. Set 'scoring' parameter for relevant scores.
- Step 3: Convert scores to error
- Step 4: Plot Validation curve with Hyper-parameter on X-axis and error on Y-axis
- Step 5: Fix the Hyper-parameter value where the test error is the least.

Predicting the outcomes and residue calculation.

In [101…

```
predictions = sgd_pipeline.predict(X_test)
predictions
```

Out[101]:

```
array([16.95471138, 20.3451164 , 23.63569173,  9.28733082, 21.82715094,
       12.52634403, 21.11434896,  8.76614239, 17.26259251, 16.660228  ,
        9.09785065,  8.4966473 , 17.91796638,  8.22794286, 12.63909645,
       14.88669215,  8.14618303, 17.96097387, 11.02904015, 20.54756258,
       20.61212659, 12.2951041 , 11.06064182, 22.20211371,  9.55927491,
        7.97080787, 20.84327574, 13.90043339, 10.80152967,  8.11342759,
       15.95643033, 10.71916393, 20.7090977 , 10.26662047, 21.49754334,
       21.28853446, 12.305904  , 22.65731623, 12.72394709,  6.52152877,
       11.9407911 , 15.40889292,  9.94159103,  9.5349432 , 17.2466592 ,
        7.31652233, 10.33331787, 15.29812504, 11.11814278, 11.82188239,
       13.90486271, 14.7173825 , 10.35597328,  9.28128326,  9.08803035,
       12.46138839, 10.58756895, 24.8986902 ,  7.99163497, 15.92960633])
```

In [107…

```
actual = y_test
```

In [108…

```
result = pd.DataFrame()
result['Actual Value'] = actual
result['Predicted Value'] = predictions
result['Residue'] = actual - predictions
```

In [109…

```
result.head()
```

Out[109]:

	Actual Value	Predicted Value	Residue
95	16.9	16.954711	-0.054711
15	22.4	20.345116	2.054884
30	21.4	23.635692	-2.235692
158	7.3	9.287331	-1.987331
128	24.7	21.827151	2.872849

GROUP 21

- Sai Phani Ram Popuri : **2205577**
- Sandeep Potla : **2151524**
- Sai Suma Podila : **2149229**
- Manivardhan Reddy Pidugu : **2146807**
- Praveen : **ppraveen@uh.edu**

Importing Essential libraries

In [258…

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import MinMaxScaler
from sklearn.feature_selection import SelectKBest, f_regression

import warnings
warnings.filterwarnings('ignore')

df = pd.read_csv('/Users/beingrampopuri/Downloads/mtcars.csv')
```

Querying the first 5 records of the data frame.

In [259…

```
df.head()
```

Out[259]:

	Unnamed: 0	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

- It has been observed that one of the columns is not labelled. The below snippet handles such cases.

In [260…

```
df.rename( columns={'Unnamed: 0':'Vehicle Model'}, inplace=True )
```

In [261…

```
df.head(10)
```

Out[261]:

	Vehicle Model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
5	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
6	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
7	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
8	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
9	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

- Fetching the insights of the data frame as a whole.

In [262…

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 12 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Vehicle Model  32 non-null     object
 1   mpg            32 non-null     float64
 2   cyl            32 non-null     int64
 3   disp           32 non-null     float64
 4   hp             32 non-null     int64
 5   drat           32 non-null     float64
 6   wt             32 non-null     float64
 7   qsec           32 non-null     float64
 8   vs             32 non-null     int64
 9   am             32 non-null     int64
 10  gear           32 non-null     int64
 11  carb           32 non-null     int64
dtypes: float64(5), int64(6), object(1)
memory usage: 3.1+ KB
```

Obtaining the statistical parameters of each individual feature.

In [263…

```
df.describe()
```

Out[263]:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
count	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.000000	32.0000
mean	20.090625	6.187500	230.721875	146.687500	3.596563	3.217250	17.848750	0.437500	0.406250	3.687500	2.8125
std	6.026948	1.785922	123.938694	68.562868	0.534679	0.978457	1.786943	0.504016	0.498991	0.737804	1.6152
min	10.400000	4.000000	71.100000	52.000000	2.760000	1.513000	14.500000	0.000000	0.000000	3.000000	1.0000
25%	15.425000	4.000000	120.825000	96.500000	3.080000	2.581250	16.892500	0.000000	0.000000	3.000000	2.0000
50%	19.200000	6.000000	196.300000	123.000000	3.695000	3.325000	17.710000	0.000000	0.000000	4.000000	2.0000
75%	22.800000	8.000000	326.000000	180.000000	3.920000	3.610000	18.900000	1.000000	1.000000	4.000000	4.0000
max	33.900000	8.000000	472.000000	335.000000	4.930000	5.424000	22.900000	1.000000	1.000000	5.000000	8.0000

Splitting the data frame into Features and Output Values.

In [264…

```
X = df.iloc[:, 2:]
y = df['mpg']
```

- Creating Test and Training sets with 15% of the data labelled as test set.

In [265…

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.15, random_state=42)
```

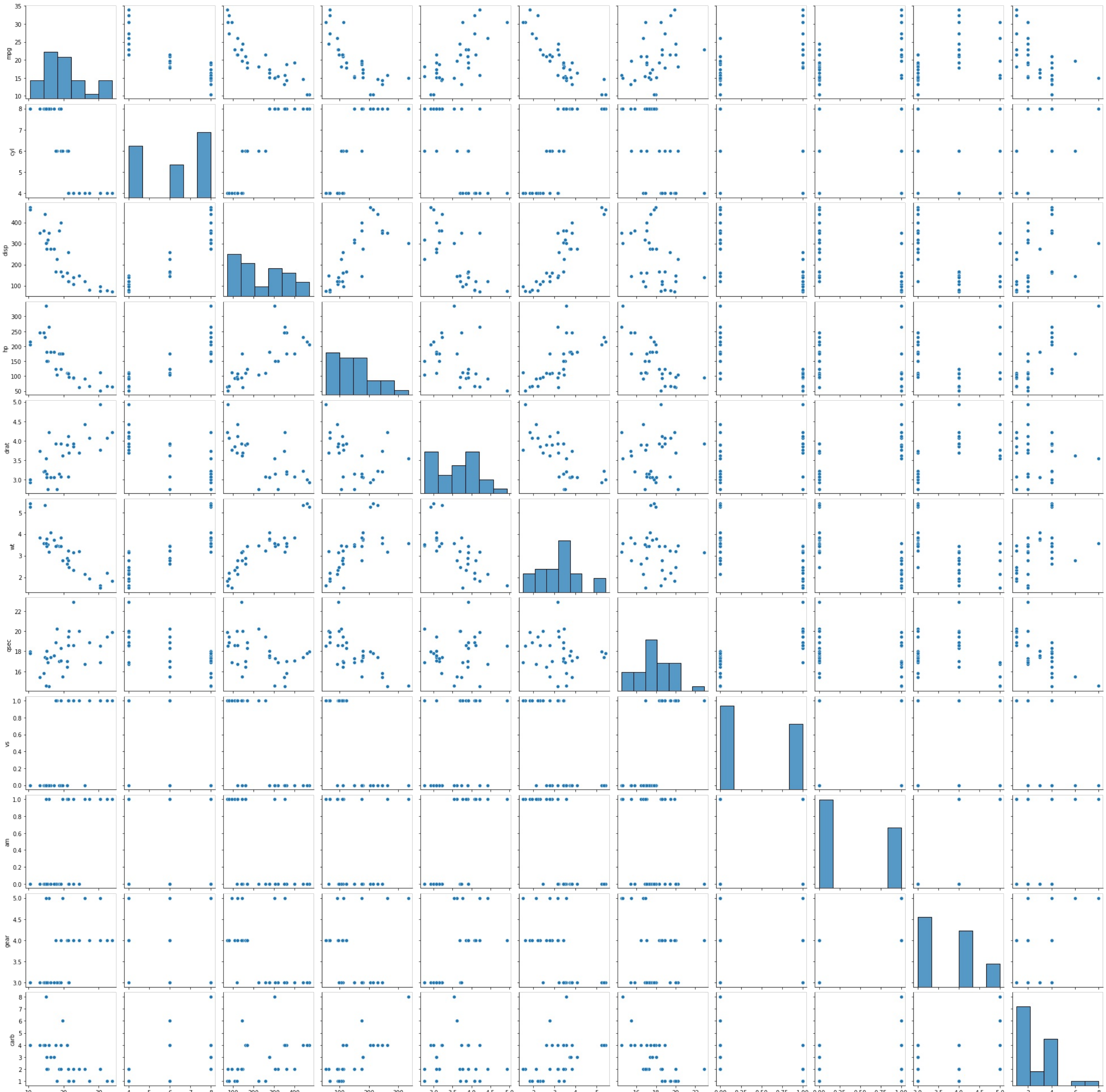
- Plotting the relationship between each individual features.

In [266…

```
sns.pairplot(df)
```

Out[266]:

```
<seaborn.axisgrid.PairGrid at 0x7ff0582b4850>
```



- Fetching the Correlation matrix that Quantifies the relationship between any two features.
- The below corr( ) function makes use of Pearson's correlation coefficient.
- Greater the magnitude, stronger is the relation between the variables.

- Positive Correlation: As X1 increases, X2 increases.
- Negative Correlation: As X1 increases, X2 decreases.

In [267…
```
'''
To choose 3 most dominant features to carry out multiple linear regression, we need to make use of
CORRELATION COEFFICIENT.

From this below matrix, we need to pick the features that has the strongest relationship with the output variab
which is mpg (miles per gallon).

The direction of the relationship could be either POSITIVE or NEGATIVE.

'''
```

Out[267]:
```
'\nTo choose 3 most dominant features to carry out multiple linear regression, we need to make use of \nCORREL
ATION COEFFICIENT. \n\nFrom this below matrix, we need to pick the features that has the strongest relationshi
p with the output variable, \nwhich is mpg (miles per gallon).\n\nThe direction of the relationship could be e
ither POSITIVE or NEGATIVE.\n\n'
```

In [268…
```
df.corr()
```

Out[268]:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
mpg	1.000000	-0.852162	-0.847551	-0.776168	0.681172	-0.867659	0.418684	0.664039	0.599832	0.480285	-0.550925
cyl	-0.852162	1.000000	0.902033	0.832447	-0.699938	0.782496	-0.591242	-0.810812	-0.522607	-0.492687	0.526988
disp	-0.847551	0.902033	1.000000	0.790949	-0.710214	0.887980	-0.433698	-0.710416	-0.591227	-0.555569	0.394977
hp	-0.776168	0.832447	0.790949	1.000000	-0.448759	0.658748	-0.708223	-0.723097	-0.243204	-0.125704	0.749812
drat	0.681172	-0.699938	-0.710214	-0.448759	1.000000	-0.712441	0.091205	0.440278	0.712711	0.699610	-0.090790
wt	-0.867659	0.782496	0.887980	0.658748	-0.712441	1.000000	-0.174716	-0.554916	-0.692495	-0.583287	0.427606
qsec	0.418684	-0.591242	-0.433698	-0.708223	0.091205	-0.174716	1.000000	0.744535	-0.229861	-0.212682	-0.656249
vs	0.664039	-0.810812	-0.710416	-0.723097	0.440278	-0.554916	0.744535	1.000000	0.168345	0.206023	-0.569607
am	0.599832	-0.522607	-0.591227	-0.243204	0.712711	-0.692495	-0.229861	0.168345	1.000000	0.794059	0.057534
gear	0.480285	-0.492687	-0.555569	-0.125704	0.699610	-0.583287	-0.212682	0.206023	0.794059	1.000000	0.274073
carb	-0.550925	0.526988	0.394977	0.749812	-0.090790	0.427606	-0.656249	-0.569607	0.057534	0.274073	1.000000

We can make use of the sklearn's - SelectKBest class that automates the feature selection process.

- There are multiple scores that can be used to pick the best 'k' features namely: p-value, F_score, Chi_score etc.
- Here, I am making use of the chi score as a performance metric.

In [269…
```
from sklearn.feature_selection import SelectKBest, chi2, f_regression
```

- Initializing the Object and defining the parameters as required.
- fit_transform( ): This method will help us with two things:
  1. Fitting the training set to the regression curve,
  2. Scaling the data and transforming the data.

In [270…
```
x_train_new = SelectKBest(score_func = f_regression,k=3).fit_transform(x_train, y_train)
```

In [271…
```
x_train_new.shape
```

Out[271]:
```
(27, 3)
```

Since we have chosen 3 features in our argument, in our resultant array we got the values pertaining to the 3 features from the x_train dataframe.

In [272…
```
x_train_new
```

```
Out[272]: array([[  6.   , 167.6  ,   3.44 ],
       [  8.   , 301.   ,   3.57 ],
       [  4.   ,  79.   ,   1.935],
       [  8.   , 275.8  ,   3.73 ],
       [  6.   , 160.   ,   2.62 ],
       [  8.   , 360.   ,   3.44 ],
       [  8.   , 440.   ,   5.345],
       [  6.   , 225.   ,   3.46 ],
       [  8.   , 275.8  ,   3.78 ],
       [  8.   , 275.8  ,   4.07 ],
       [  8.   , 350.   ,   3.84 ],
       [  6.   , 160.   ,   2.875],
       [  4.   , 108.   ,   2.32 ],
       [  4.   , 120.3  ,   2.14 ],
       [  6.   , 258.   ,   3.215],
       [  8.   , 318.   ,   3.52 ],
       [  4.   ,  95.1  ,   1.513],
       [  8.   , 304.   ,   3.435],
       [  4.   ,  75.7  ,   1.615],
       [  4.   , 121.   ,   2.78 ],
       [  4.   , 120.1  ,   2.465],
       [  4.   , 146.7  ,   3.19 ],
       [  6.   , 167.6  ,   3.44 ],
       [  8.   , 472.   ,   5.25 ],
       [  8.   , 351.   ,   3.17 ],
       [  4.   ,  71.1  ,   1.835],
       [  8.   , 360.   ,   3.57 ]])
```

```
In [273…: x_train_new.shape
```

```
Out[273]: (27, 3)
```

Observation: Though we have secured the feature values, it is a bit ambiguous what those 3 features are.

- We need to compare the values with those in the dataset, to arrive at the desired and the most important features.

```
In [274…: x_train.sort_index()
```

Out[274]:

	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	8	360.0	175	3.15	3.440	17.02	0	0	3	2
5	6	225.0	105	2.76	3.460	20.22	1	0	3	1
6	8	360.0	245	3.21	3.570	15.84	0	0	3	4
7	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	6	167.6	123	3.92	3.440	18.30	1	0	4	4
10	6	167.6	123	3.92	3.440	18.90	1	0	4	4
11	8	275.8	180	3.07	4.070	17.40	0	0	3	3
12	8	275.8	180	3.07	3.730	17.60	0	0	3	3
13	8	275.8	180	3.07	3.780	18.00	0	0	3	3
14	8	472.0	205	2.93	5.250	17.98	0	0	3	4
16	8	440.0	230	3.23	5.345	17.42	0	0	3	4
18	4	75.7	52	4.93	1.615	18.52	1	1	4	2
19	4	71.1	65	4.22	1.835	19.90	1	1	4	1
20	4	120.1	97	3.70	2.465	20.01	1	0	3	1
21	8	318.0	150	2.76	3.520	16.87	0	0	3	2
22	8	304.0	150	3.15	3.435	17.30	0	0	3	2
23	8	350.0	245	3.73	3.840	15.41	0	0	3	4
25	4	79.0	66	4.08	1.935	18.90	1	1	4	1
26	4	120.3	91	4.43	2.140	16.70	0	1	5	2
27	4	95.1	113	3.77	1.513	16.90	1	1	5	2
28	8	351.0	264	4.22	3.170	14.50	0	1	5	4
30	8	301.0	335	3.54	3.570	14.60	0	1	5	8
31	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Based on the Chi Square Test, we found that disp, hp, carb are the best 3 features.

In [275…
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error

from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
```

In [276…
```
X1 = df[['disp', 'hp', 'carb']]
y1 = df['mpg']
```

In [277…
```
x_train, x_test, y_train, y_test = train_test_split(X1, y1, test_size=0.3, random_state=42)
```

Fitting the model based on x_train and y_train

In [278…
```
regression = LinearRegression()
regression.fit(x_train, y_train)
```

Out[278]:
```
LinearRegression()
```

Weights and y-intercept values

In [279…
```
w = regression.coef_
b = regression.intercept_
w, b
```

Out[279]:
```
(array([-0.03608767, -0.00732251, -0.60961123]), 30.9993068086036)
```

Equation to predict the output : mpg

In [280…
```
print(f"The multiple linear regression model based on the training data is \n")
print(f"predicted_mpg = {np.round(w[0],3)} * disp  {np.round(w[1], 3)} * hp  {np.round(w[2], 3)} * carb + {np.r
```

```
The multiple linear regression model based on the training data is

predicted_mpg = -0.036 * disp  -0.007 * hp  -0.61 * carb + 30.999
```

In [281…
```
x_test
```

Out[281]:

	disp	hp	carb
29	145.0	175	6
15	460.0	215	4
24	400.0	175	2
17	78.7	66	1
8	140.8	95	2
9	167.6	123	4
30	301.0	335	8
25	79.0	66	1
12	275.8	180	3
0	160.0	110	4

In [282…
```
y_test
```

Out[282]:
```
29    19.7
15    10.4
24    19.2
17    32.4
8     22.8
9     19.2
30    15.0
25    27.3
12    17.3
0     21.0
Name: mpg, dtype: float64
```

In [283…
```
arr = np.array(x_test[['disp', 'hp', 'carb']])

x_test['Predictions'] = regression.predict(arr)
```

```
x_test
```

Out[283]:

	disp	hp	carb	Predictions
29	145.0	175	6	20.827487
15	460.0	215	4	10.386192
24	400.0	175	2	14.063575
17	78.7	66	1	27.066310
8	140.8	95	2	24.003301
9	167.6	123	4	21.611899
30	301.0	335	8	12.806986
25	79.0	66	1	27.055484
12	275.8	180	3	17.899441
0	160.0	110	4	21.981358

In [284…

```
x_test
```

Out[284]:

	disp	hp	carb	Predictions
29	145.0	175	6	20.827487
15	460.0	215	4	10.386192
24	400.0	175	2	14.063575
17	78.7	66	1	27.066310
8	140.8	95	2	24.003301
9	167.6	123	4	21.611899
30	301.0	335	8	12.806986
25	79.0	66	1	27.055484
12	275.8	180	3	17.899441
0	160.0	110	4	21.981358

Calculating the difference in predicted values and actual values of mpg

In [285…

```
x_test['Actual'] = y_test
x_test['Residue'] = x_test['Predictions'] - x_test['Actual']

x_test
```

Out[285]:

	disp	hp	carb	Predictions	Actual	Residue
29	145.0	175	6	20.827487	19.7	1.127487
15	460.0	215	4	10.386192	10.4	-0.013808
24	400.0	175	2	14.063575	19.2	-5.136425
17	78.7	66	1	27.066310	32.4	-5.333690
8	140.8	95	2	24.003301	22.8	1.203301
9	167.6	123	4	21.611899	19.2	2.411899
30	301.0	335	8	12.806986	15.0	-2.193014
25	79.0	66	1	27.055484	27.3	-0.244516
12	275.8	180	3	17.899441	17.3	0.599441
0	160.0	110	4	21.981358	21.0	0.981358

Part 4

- Why Mean Squared Error is the best natural algorithm for regression problems?
- Probabilistic Interpretation of Linear Regression

Assumptions of Probabilistic Interpretation of Regression

- There is an error term involved, that handles any neglected values that might have a pertinent role in defining the outcome.
- The error terms are NORMALLY DISTRIBUTED.

Remarks

- The Mean Squared Error derives its inspiration from the Principle of Maximum Likelihood in Probability.
- It states: "From the Normal Equation, we should should chooseθsoas to make the data as high probability as possible."
- Finding Maximum of f(x) = Minimum of -f(x) which is our loss function in linear regression.

Therefore, the Mean Squared Error best fits the regression problems.