Multivariate Regression - Roshan Parajuli

Multivariate Regression is a method used to measure the degree at which more than one independent variable (predictors) and more than one dependent variable (responses), are linearly related. The method is broadly used to predict the behavior of the response variables associated to changes in the predictor variables, once a desired degree of relation has been established.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Importing the necessary libraries for computation.

Numpy contains a multi-dimensional array and data structures.

Pandas helps to manipulate and analyse data.

Pyplot module in matplotlib is imported to plot and analyse the data as well as the outcome.

In [3]:
```
df = pd.read_csv('Restaurant_Profit.csv')

X = df.iloc[:,:-1]

# All rows, except the last column.
# There are multiple ways to do this. The last column could have been dropped as well.
#X = df.drop(['Profit'],axis="columns") is the syntax for dropping the last column.

y = df.iloc[:,-1]
# All rows from only last or the 4th column in this case.

df.head()
```

Out[3]:

	Miscellaneous_Expenses	Food_Innovation_Spend	Advertising	City	Profit
0	138671.80	167497.20	475918.10	Chicago	202443.83
1	153151.59	164745.70	448032.53	Mumbai	201974.06
2	102919.55	155589.51	412068.54	Tokyo	201232.39
3	120445.85	146520.41	387333.62	Chicago	193083.99
4	93165.77	144255.34	370302.42	Tokyo	176369.94

read_csv method in pandas library is mostly used to read comma separated values files. However, It can also read text files (with .txt extension) or other files with .xlsx extension or so on. If the data inside the files are not separated by a comma, we need a delimiter to specify what separates one value from the other. The csv file is loaded into the dataframe namely df in this case.

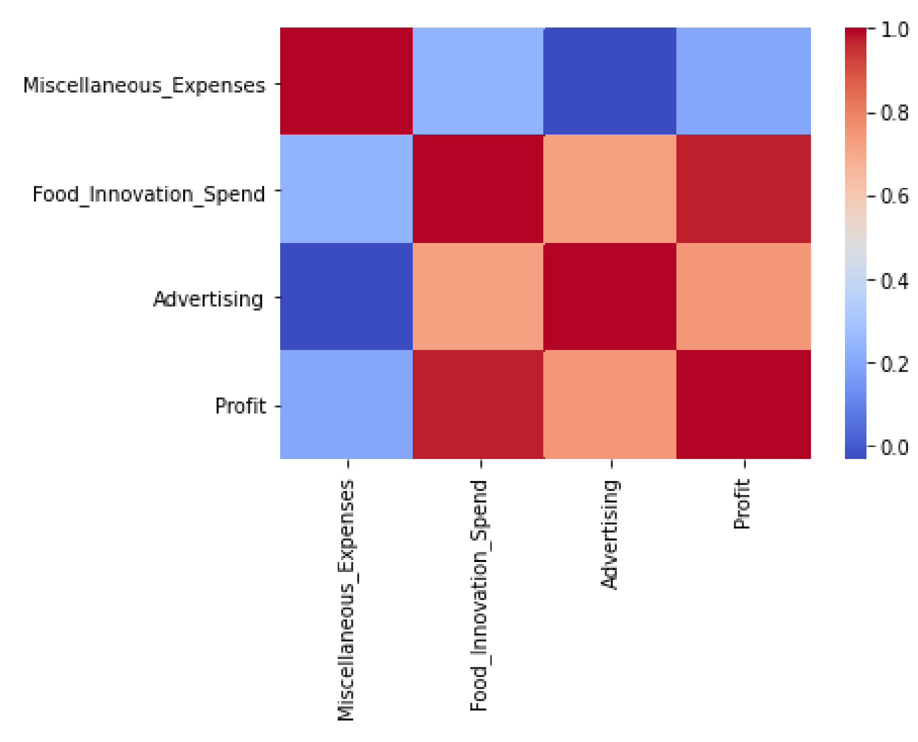
We can filter the rows and columns from the dataframe by using the iloc method. iloc stands for Integer based Locating. We are omitting the last column in the dataframe in X because X is going to be our input and we need to predict the last column and thus, it is stored in y.

head method in dataframe is used to show the top 5 data from our dataframe if no any specific number is passed as an argument to the method.

Plotting the correlation in a seaborn heatmap to find out which features are more related to each other and which are not.

In [4]:
```
sns.heatmap(df.corr(), cmap='coolwarm')
```

Out[4]: <AxesSubplot:>



In [5]:
```
X = pd.concat([pd.Series(1,index=df.index,name='Bias0'),df],axis=1)
X.head()
```

Out[5]:

	Bias0	Miscellaneous_Expenses	Food_Innovation_Spend	Advertising	City	Profit
0	1	138671.80	167497.20	475918.10	Chicago	202443.83
1	1	153151.59	164745.70	448032.53	Mumbai	201974.06
2	1	102919.55	155589.51	412068.54	Tokyo	201232.39
3	1	120445.85	146520.41	387333.62	Chicago	193083.99
4	1	93165.77	144255.34	370302.42	Tokyo	176369.94

On our input, we are adding a bias column and initializing everyone of them with 1 because it will not affect the other data points upon being multiplied to the other data points. This is an optional step.

In [6]:
```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()

X.City = le.fit_transform(X.City)
```

The City column in our dataframe contains categorical string values. To convert those data to model-understandable numerical value, we use label encoder from preprocessing module which is present in scikit learn library. The fit_transform method in label encoder takes a column with categorical data and assigns a number to each and every category which makes the data ready for the model.

In [7]:
```
X.City.unique()
```

Out[7]: array([0, 1, 2])

There are 3 unique cities in our dataframe which is being represented by the unique method.

In [8]:
```
for i in range(0,len(X.columns)):
    X.iloc[:,i] = X.iloc[:,i]/np.max(X.iloc[:,i])
```

Data Normalization

Data is normalized to bring data to the same scale. The model might get more affected by some columns with large number of data and less affected by some which have numerically low values which ultimately makes the model biased. Here, every element from every column is being divided by the maximum value of data from the same column to bring uniformity to he data. Only inputs are normalized and thus, the data in y remains as is.

In [9]:
```
theta = np.array([0]*len(X.columns))
theta
```

Out[9]: array([0, 0, 0, 0, 0, 0])

Here an array of thetas are initialized which contains zero as of now. They can be anything. These are initial values which are going to be optimized by the gradient descent function coded a few blocks below.

In [11]:
```
# Length of the entire dataframe.
m =len(df)
m
```

Out[11]: 50

The total length of the dataset is found out to be 50 with the help of len function and is soon going to be used to compute cost and optimze the data on gradient descent.

In [12]:
```
def hypothesis(theta,X):
    return theta*X
```

This is a hypothesis funtion which returns the product of the theta and the input value. It is used for prediction or to be simply put, y-hat.

In [13]:
```
def computeCost(X, y, theta):
    y1 = hypothesis(theta,X)
    y1 = np.sum(y1, axis='columns')
    return np.sqrt(np.sum((y1-y)**2)/(2*m))
```

Here we are finding cost function to estimate how badly model is performing. Put simply, a cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between X and y. This is typically expressed as a difference or distance between the predicted value and the actual value. It can be estimated by iteratively running the model to compare estimated predictions against “ground truth” — the known values of y. The objective of a ML model, therefore, is to find parameters, weights or a structure that minimises the cost function.

This is the calculation of RMSE (Root Mean Square Error). Other metrics used to determine the accuracy of the model could have been MSE (Mean Square Error) and MAE (Mean Absolute Error). MAE is not used in this case because it fails to penalize or punish large errors in prediction. RMSE has the benefit of penalizing large errors more so can be more appropriate in some cases, for example, if being off by 10 is more than twice as bad as being off by 5. But if being off by 10 is just twice as bad as being off by 5, then MAE is more appropriate.

In short, MAE measures the absolute average distance between the real data and the predicted data, but it fails to punish large errors in prediction.

MSE measures the squared average distance between the real data and the predicted data. Here, larger errors are well noted (better than MAE). But the disadvantage is that it also squares up the units of data as well. So, evaluation with different units is not at all justified.

RMSE is the square root of MSE. Also, this metrics solves the problem of squaring the units.

In [15]:
```
# Optimizing data with gradient descent function
def gradientDescent(X, y, theta, alpha, it):
    J = []
    for i in range(0,it):
        y1 = hypothesis(theta,X)
        y1 = np.sum(y1, axis='columns')
        for c in range(0, len(X.columns)):
            theta[c] = theta[c] - alpha*(sum((y1-y)*X.iloc[:,c])/m)
        j = computeCost(X, y, theta)
        J.append(j)
    return J, j, theta
```

Gradient Descent is used here to optimize the random theta which are initialized ealier. It is simply an optimization algorithm that's used when training a machine learning model. It is able to tweak its parameters iteratively to minimize a given function to its local minimum.

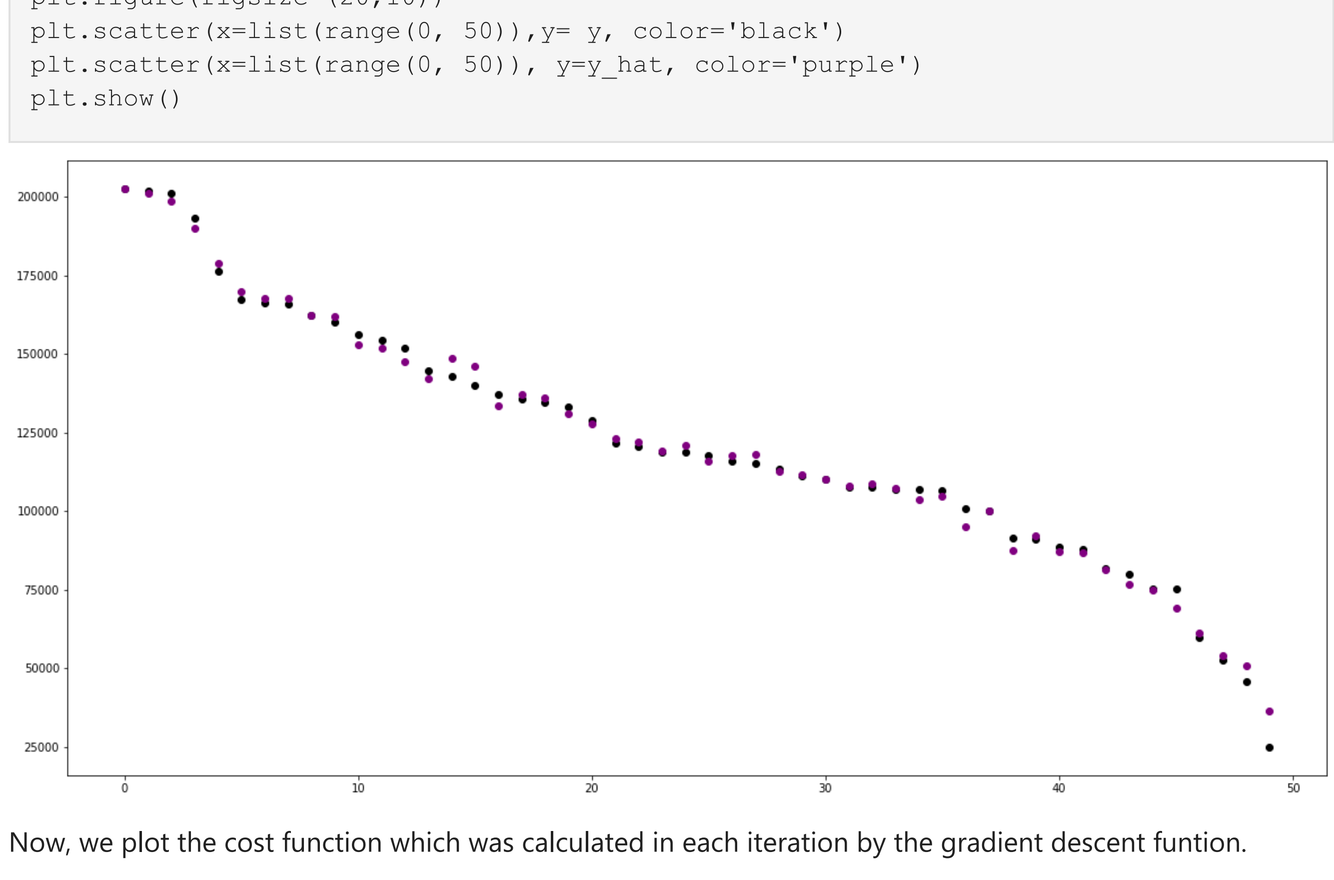
In [16]:
```
J, j, theta = gradientDescent(X, y, theta, 0.05, 10000)
```

In [17]:
```
y_hat = hypothesis(theta, X)
y_hat = np.sum(y_hat, axis=1)
```

y_hat here is the prediction of the model of the given y. It is the summation of y_hat in the column after being passed through the hypothesis function.

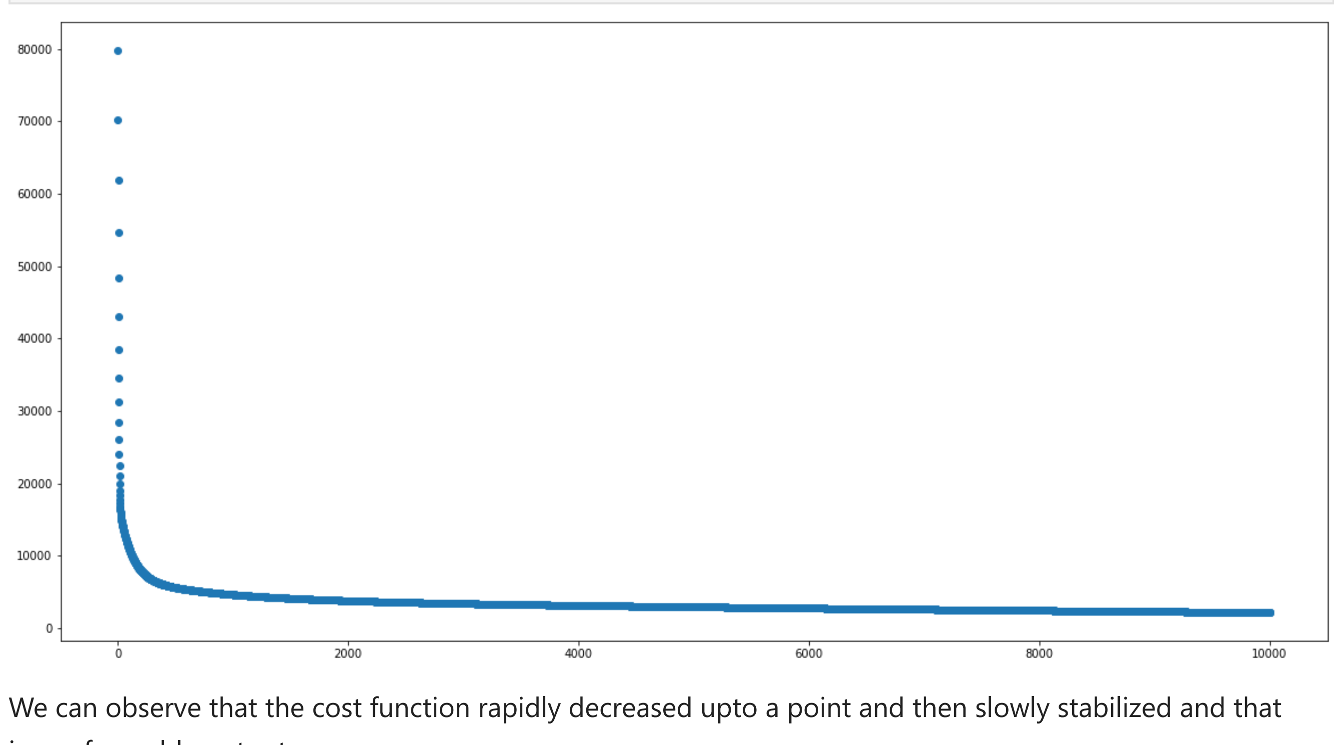
Now, we plot the original y values and the predicted y values to determine how accurate the model has predicted.

In [20]:
```
# Plotting the values of y and y_hat
%matplotlib inline
import matplotlib.pyplot as plt
plt.figure(figsize=(20,10))
plt.scatter(x=list(range(0, 50)),y= y, color='black')
plt.scatter(x=list(range(0, 50)), y=y_hat, color='purple')
plt.show()
```



Now, we plot the cost function which was calculated in each iteration by the gradient descent funtion.

In [24]:
```
plt.figure(figsize=(20,10))
plt.scatter(x=list(range(0, 10000)), y=J)
plt.show()
```



We can observe that the cost function rapidly decreased upto a point and then slowly stabilized and that is our favorable output.

In [28]:
```
# Mean squared error
from sklearn.metrics import mean_squared_error
mean_squared_error(y, y_hat)
```

Out[28]: 9453079.37135061

Finally, we calculate the accuracy of the model also commonly known as r_squared. It represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model.

In [32]:
```
# Checking the accuracy of the model a.k.a r squared.
from sklearn.metrics import r2_score

r2_score(y,y_hat)
```

Out[32]: 0.9940624956426545

The model is 99.40% accurate.