linearRegression

October 29, 2023

[1]:
```
import pandas as pd
```

[2]:
```
#Reading data from the csv file
studentPerformance = pd.read_csv("Student_Performance.csv")
```

[3]:
```
#Displaying the first five rows
studentPerformance.head()
```

[3]:

	Hours Studied	Previous Scores	Extracurricular Activities	Sleep Hours	Sample Question Papers Practiced	Performance Index
0	7	99	Yes	9	1	91.0
1	4	82	No	4	2	65.0
2	8	51	Yes	7	2	45.0
3	5	52	Yes	5	2	36.0
4	7	75	No	8	5	66.0

This output clearly shows the first five rows of the dataset.

[4]:
```
studentPerformance.tail()
```

[4]:

	Hours Studied	Previous Scores	Extracurricular Activities	Sleep Hours	Sample Question Papers Practiced	Performance Index
9995	1	49	Yes	4	2	23.0
9996	7	64	Yes	8	5	58.0
9997	6	83	Yes	8	5	74.0
9998	9	97	Yes	7	0	95.0
9999	7	74	No	8		

```
9999                                1               64.0
```

This output clearly shows the last five rows of the dataset.

[5]:
```
studentPerformance.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 6 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   Hours Studied                     10000 non-null  int64
 1   Previous Scores                   10000 non-null  int64
 2   Extracurricular Activities        10000 non-null  object
 3   Sleep Hours                       10000 non-null  int64
 4   Sample Question Papers Practiced  10000 non-null  int64
 5   Performance Index                 10000 non-null  float64
dtypes: float64(1), int64(4), object(1)
memory usage: 468.9+ KB
```

This displays more information about the dataset in detail. That is, the data types of each column, the number of non-null values and the memory usage.

[6]:
```
studentPerformance.describe()
```

[6]:
```
       Hours Studied  Previous Scores   Sleep Hours  \
count   10000.000000     10000.000000  10000.000000
mean        4.992900        69.445700      6.530600
std         2.589309        17.343152      1.695863
min         1.000000        40.000000      4.000000
25%         3.000000        54.000000      5.000000
50%         5.000000        69.000000      7.000000
75%         7.000000        85.000000      8.000000
max         9.000000        99.000000      9.000000

       Sample Question Papers Practiced  Performance Index
count                      10000.000000       10000.000000
mean                           4.583300          55.224800
std                            2.867348          19.212558
min                            0.000000          10.000000
25%                            2.000000          40.000000
50%                            5.000000          55.000000
75%                            7.000000          71.000000
max                            9.000000         100.000000
```

The output shows the results of calculations based on the statistics of numerical columns.

[7]:
```
#Checking for missing values
missingValues = studentPerformance.isnull().sum()
```

```
print("Missing Values:\n", missingValues)
```

```
Missing Values:
 Hours Studied                      0
Previous Scores                     0
Extracurricular Activities          0
Sleep Hours                         0
Sample Question Papers Practiced    0
Performance Index                   0
dtype: int64
```

The **isnull()** function is used to determine whether each element in a DataFrame or Series is null(missing) or not. When you chain the isnull() function with **sum()**, it calculates the number of missing values in each column of a DataFrame. For example, if you have a DataFrame named footballPerformance and you want to know how many missing values are there in each column, you can use footballPerformance.isnull().sum(). This will return a Series where the index represents column names, and the values represent the count of missing values in each respective column. Judging from the output, it appears that there are no null values.

[8]:
```
#Checking for duplicate rows on all columns and displaying them
duplicateRows = studentPerformance.duplicated(keep=False)
print(duplicateRows)
```

```
0       False
1       False
2       False
3       False
4       False
        …
9995    False
9996    False
9997    False
9998    False
9999    False
Length: 10000, dtype: bool
```

Thus, there are no duplicate rows. How? The **duplicated()** method is a Pandas DataFrame method that identifies duplicate rows in a DataFrame or a dataset. By defualt, it considers the first occurrence of a duplicate as not a duplicate(i.e: it keeps the first instance and marks subsequent occurences as duplicates). Moreover, the **'keep'** parameter is set to **'False'**, which means all duplicates will be marked as **'True'**. If this parameter were set to True, only the first occurrence of a duplicate would be marked as True, and subsequent duplicates would be marked as False. 'duplicateRows': This line assigns the resulting Boolean Series(indicating duplicates) to the variable duplicateRows. In this Series, True indicates a duplicate row, and False indicates a non-duplicate row.

[9]:
```
#Identifying categorical features
studentPerformance.dtypes
```

[9]:
```
Hours Studied                          int64
Previous Scores                        int64
Extracurricular Activities            object
Sleep Hours                            int64
Sample Question Papers Practiced       int64
Performance Index                    float64
dtype: object
```

The output above indicates that there are only categorical features in the Extracurricular Activities. Why? Because the dtype function displays the data type of each column.

[10]:
```
#Specifying columns to be converted to numeric data types
columnConvert = ['Extracurricular Activities']
studentPerformance[columnConvert] = studentPerformance[columnConvert].apply(pd.
 ↪to_numeric, errors='coerce')

#Checking for outliers
Q1 = studentPerformance.quantile(0.25)
Q3 = studentPerformance.quantile(0.75)
interquartileRange = Q3 - Q1

lowerBound = Q1 - 1.5 * interquartileRange
upperBound = Q3 + 1.5 * interquartileRange
outliers = (studentPerformance < lowerBound) | (studentPerformance > upperBound)

#Displaying columns with outliers
column_outliers = outliers.any()
print("Columns with outliers:\n")
print(column_outliers[column_outliers].index)
```

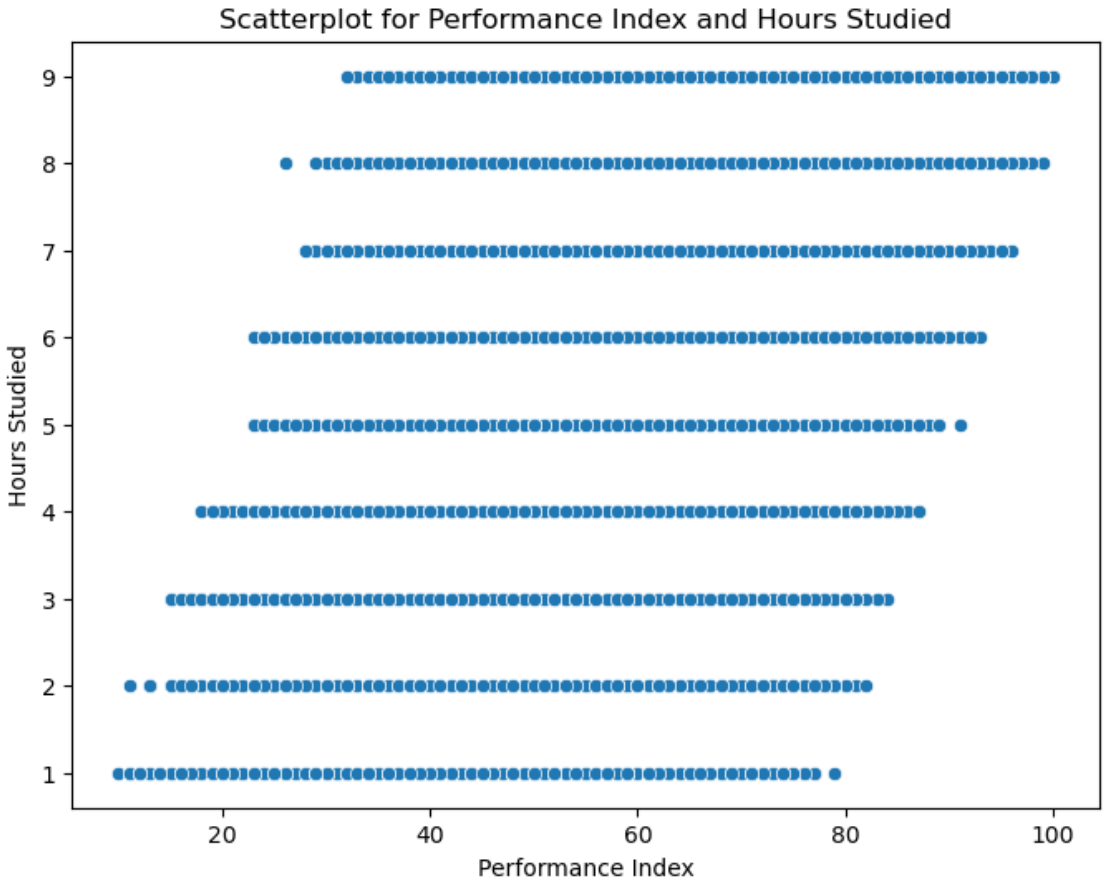
```
Columns with outliers:

Index([], dtype='object')
```

This output indicates that there are no outliers in the dataset. Also, the **pd.to_numeric()** functions is used to convert the columns of 'studentPerformance' DataFrame to numeric data types. And the **outliers.any()** is used as part of the condition to identify the presence of outliers in a dataset. In the context of the pd.to_numeric() function in pandas, the errors='coerce' parameter specifies how to handle errors that might occur during the conversion of data to numeric format. When you set **errors='coerce'** in pd.to_numeric(), it means that if any parsing errors occur during the conversion(for example, if there are non-numeric strings in the data), those errors will be replaced with NaN(Not a Number) values.

[11]:
```
#Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import mean_squared_error, r2_score
```

[13]:
```
#Using seaborn to visualize the a scatterplot of the data
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Performance Index', y='Hours Studied',␣
 ↪data=studentPerformance)
plt.title('Scatterplot for Performance Index and Hours Studied')
plt.show()
```



plt.figure() is a function from the Matplotlib library that creates a new figure, and figsize=(8, 6) sets the width and height of the figure to 8 and 6 inches, respectively. This line ensures that the upcoming plot will be displayed in an 8x6-inch figure. **sns.scatterplot()** creates a scatterplot using Seaborn's **scatterplot()** function. **data=studentPerformance** specifies the dataset the dataset (studentPerformance in this case) from which the data will be extracted for the plot.

[14]:
```
#Pinpointing the independent variable (X) and the dependent variable (y)
X = studentPerformance['Performance Index'].values.reshape(-1, 1)
y = studentPerformance['Hours Studied'].values
#Dividing the dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

.values: This converts the column data into a NumPy array, which is necessary for many machine learning operations. **.reshape(-1, 1)**: This reshapes the 1-dimensional array into a 2-dimensional array with a single column. Machine learning models in libraries like Scikit-Learn typically expect input features(X) to be in a 2-dimensional format, where each row represents a sample and each column represents a feature. Reshaping the array in this way ensures compatibility with various machine learning algorithms. **train_test_split(X, y, test_size=0.2, random_state=42)**: This function is from the Scikit-Learn library and is used to split the dataset into training and testing sets. **X** and **y** are the input features and target variable, respectively. **test_size=0.2** specifies that 20% of the data will be used for testing, and the remaining 80% will be used for training. **random_state=42** sets a seed for the random number generator, ensuring that the split is reproducible. **X_train, X_test, y_train, y_test**: These variables store the training and testing sets for both the input features(X_train, X_test) and the target variable(y_train, y_test), allowing them to be used for training and evaluating machine learning models, respectively.

[15]:
```
#Creating a linear regression model
lrModel = LinearRegression()
#Using training data to train the model
lrModel.fit(X_train, y_train)
```

[15]:
```
LinearRegression()
```

fit() function fits a machine learning model to the training data. Thus when the last line of code is executed, the linear regression model(**lrModel**) is trained using the provided training data(**X_train** for features and **y_train** for target values). During training, the model learns the relationships between the input features and the target variable, adjusting its internal parameters to minimize the difference between the predicted values and the actual target values from the training dataset. After this line of code is executed, the **lrModel** object is no longer just an empty model; it's now a trained model capable of making predictions based on new, unseen data. The model has learned from the patterns present in the training data and can be used to make predictions on similar data points.

[16]:
```
#Making predictions on the test data
y_pred = lrModel.predict(X_test)
```

The **predict()** method is used to make predictions based on input features provided in **X_test** which represents the testing data for the input features. It contains the features from the testing dataset that the model did not see during the training phase. The model uses these unseen features to make predictions. In summary, this line of code takes the testing data(**X_test**), feeds it into the trained machine learning model (**lrModel**), and generates predictions for the target variable. The predicted values are stored in the **y_pred** variable, which can be used for various evaluation metrics and further analysis to assess the model's performance on unseen data.

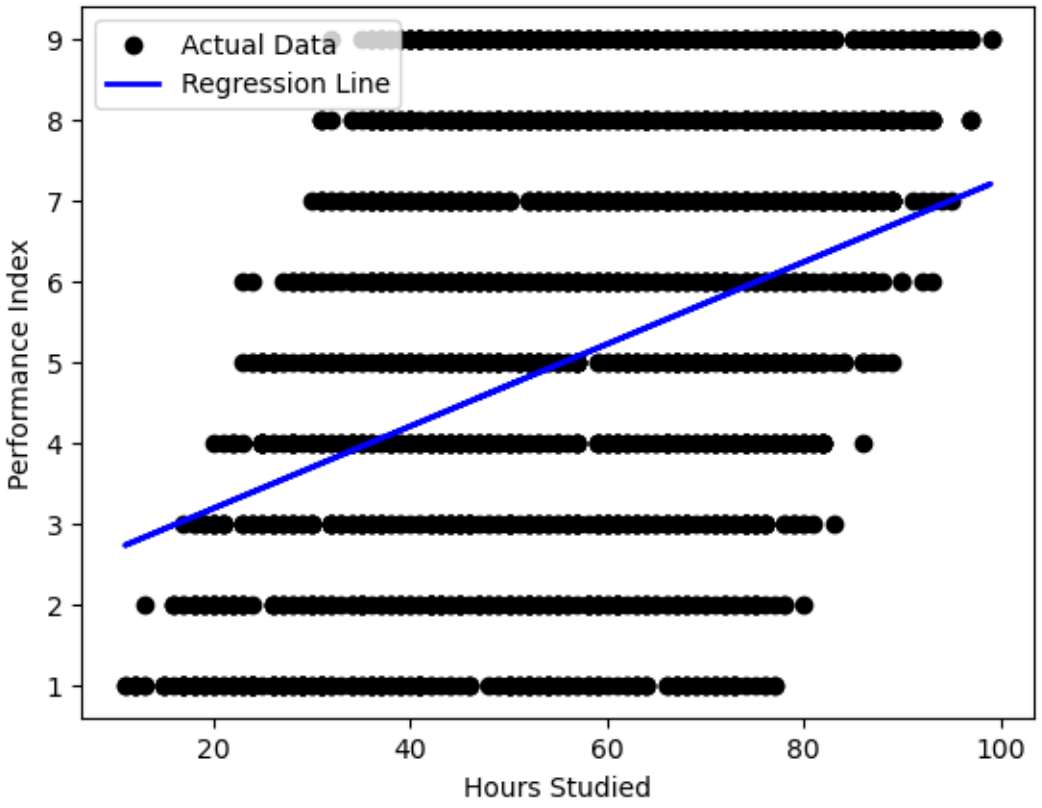
[17]:
```
#Calculate and print model performance metrics
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print(f'Mean Squared Error: {mse}')
print(f'R-squared Value: {r2}')
```

```
Mean Squared Error: 5.721408587633923
R-squared Value: 0.13147599900221407
```

mean_squared_error(y_test, y_pred): This function, *mean_squared_error()*, is from the Scikit-Learn library. It calculates the mean squared error between the true target values(**y_test**) and the predicted values(**y_pred**). Mean Squared Error measures the average of the squares of the errors, which are the differences between actual and predicted values. **r2_score(y_test, y_pred)**: This function, **r2_score()**, also from Scikit-Learn, computes the R-squared value, which is a statistical measure of how well the model explains the variance in the test data. R-squared values range from 0 to 1; a higher R-squared value indicates a better fit of the model to the data

[18]:
```
#Visualizing the Regression Line
plt.scatter(X_test, y_test, color='black', label='Actual Data')
plt.plot(X_test, y_pred, color='blue', linewidth=2, label='Regression Line')
plt.xlabel('Hours Studied')
plt.ylabel('Performance Index')
plt.legend()
plt.show()
```



label='Actual Data': Provides a label for the data points on the plot. **color='blue'**: Specifies that the regression line will be displayed in blue. **linewidth=2**: Sets the width of the line to 2 pixels. **label='Regression Line'**: Provides a label for the regression line on the plot.

Implementing Multiple Linear Regression

[21]:
```
#Interpreting model coefficients for each independent variable
modelCoefficients = pd.DataFrame({'Feature Name':['Performance Index'],␣
 ↪'Coefficient': lrModel.coef_})
print(modelCoefficients)
```

```
        Feature Name  Coefficient
0  Performance Index     0.050845
```

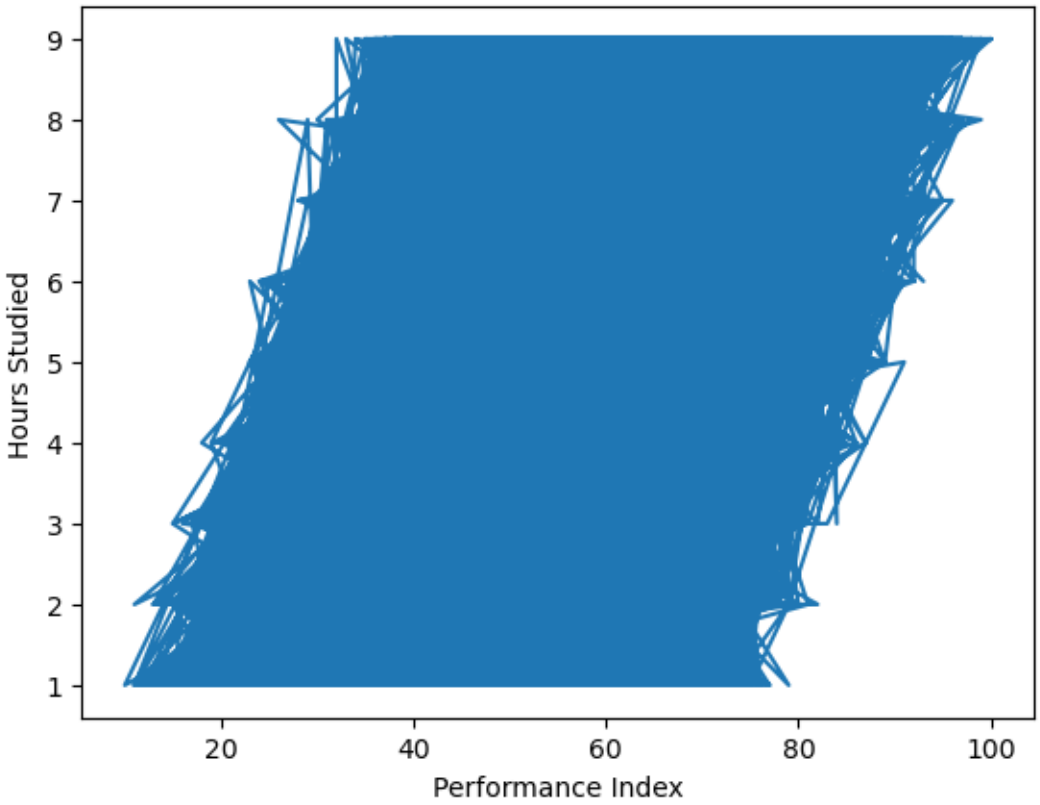
pd.DataFrame(): This function from the Pandas library creates a new DataFrame. **{'Feature Name':['Performance Index'], 'Coefficient': lrModel.coef_}**: This dictionary specifies the data for the DataFrame. Here, 'Feature Name' is set to 'Performance Index', and 'Coefficient' is set to the coefficients obtained from the trained linear regression model(**lrModel.coef_**). In a linear regression context, coefficients represent the weights applied to the features during prediction.

[24]:
```
#Predicting new values
newData = np.array([[91.0]])
predictedValue = lrModel.predict(newData)
print(f'Predicted Value: {predictedValue[0]}')
```

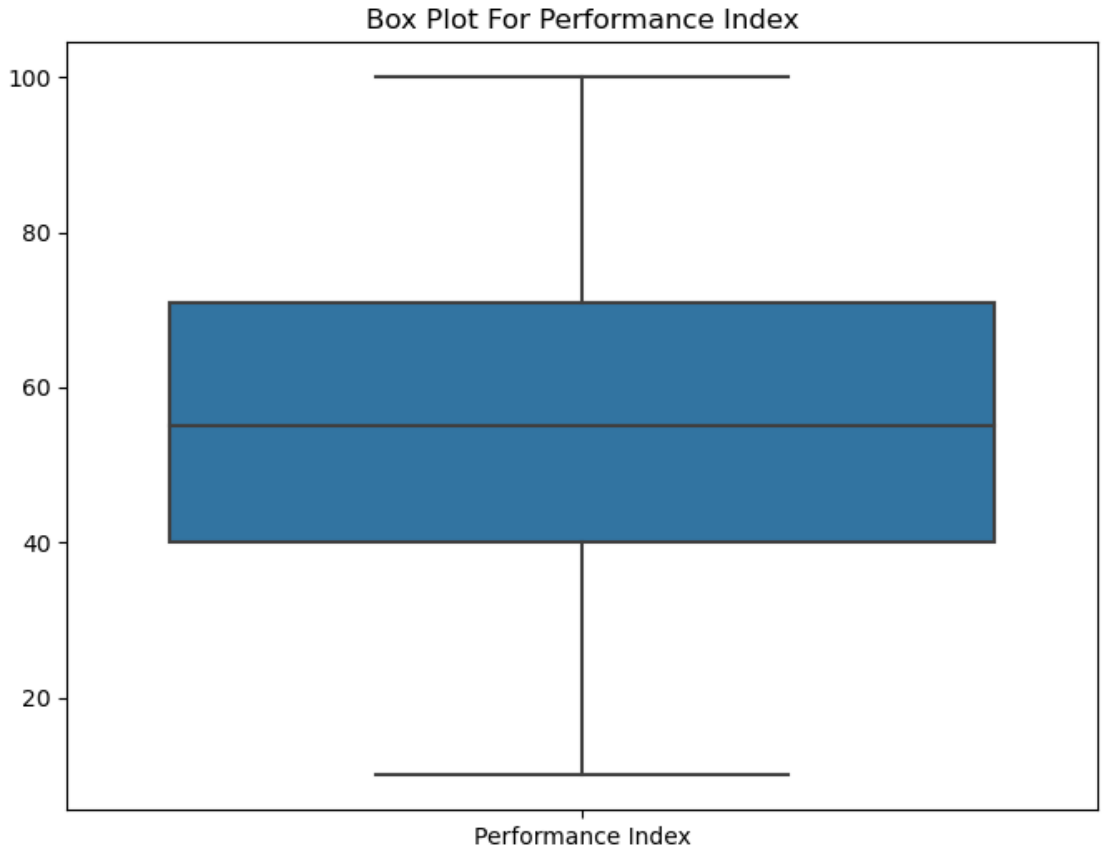
```
Predicted Value: 6.797835710725227
```

This creates a NumPy array containing a single data point, **91.0**, representing the feature('Hours Studied' in this case) for which the prediction is to be made. This uses the trained linear regression model(**lrModel**) to predict the target value for the input feature provided in newData.

[26]:
```
#Implementing a line chart
plt.plot(X, y)
plt.xlabel('Performance Index')
plt.ylabel('Hours Studied')
plt.show()
```



[27]:
```
#Using boxplot to visualize the data
plt.figure(figsize=(8, 6))
sns.boxplot(data=studentPerformance[['Performance Index']])
plt.title("Box Plot For Performance Index")
plt.show()
```



Judging from the output, the median is situated at no. 58, whilst the Q1(first quartile) is situated between 60 and 80 on the y-axis as well as 40 for the Q3(third quartile).