

DS Lab 5

Aim:- Perform Regression Analysis using Scipy and Sci-kit learn.

Problem Statement:

- a) Perform Logistic regression to find out relation between variables
- b) Apply regression model technique to predict the data on the above dataset.

Dataset Description:

Rows: 100,000

Columns: 14

Fields:

- **Region, Country:** Geographic data for sales analysis.
- **Item Type:** Product category (e.g., Snacks, Cosmetics, Personal Care).
- **Sales Channel:** Online or Offline sales mode.
- **Order Priority:** Order urgency (Low, Medium, High, Critical).
- **Order & Ship Date:** Sales and shipment timeline.
- **Units Sold:** Quantity of items sold.
- **Unit Price & Unit Cost:** Selling price and production cost per unit.
- **Total Revenue, Total Cost, Total Profit:** Financial performance metrics.

1-

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LogisticRegression
```

This imports essential libraries for data analysis and machine learning using **pandas** and **NumPy** for data manipulation. It includes **scikit-learn** modules for preprocessing (LabelEncoder, StandardScaler), model training (LinearRegression,

LogisticRegression), and evaluation (mean_squared_error). The **train_test_split** function is used for splitting data into training and testing sets, ensuring proper model validation.

2-

```
[ ] from google.colab import files
    uploaded = files.upload()

[ ] Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
    Saving Dataset_Ds.csv to Dataset_Ds (1).csv

[ ] df=pd.read_csv('Dataset_Ds.csv')
```

This is for uploading a CSV file in Google Colab using **files.upload()** from the **google.colab** module. Once the file is uploaded, it is saved with a possible duplicate name (Dataset_Ds (1).csv). The **pd.read_csv('Dataset_Ds.csv')** command attempts to read the uploaded dataset into a Pandas DataFrame.

3-

```
[ ] # Convert date columns to datetime format
    df["Order Date"] = pd.to_datetime(df["Order Date"], errors="coerce")
    df["Ship Date"] = pd.to_datetime(df["Ship Date"], errors="coerce")

[ ] # Drop rows with missing date values
    df.dropna(subset=["Order Date", "Ship Date"], inplace=True)
```

This method is essential for ensuring accurate date-based analysis and maintaining data integrity. Converting **"Order Date"** and **"Ship Date"** to datetime format allows for operations like calculating delivery time, filtering by date range, and time-series analysis. The **errors="coerce"** parameter ensures invalid entries are converted to NaT instead of causing errors. Dropping rows with missing dates prevents incorrect calculations and maintains dataset reliability, especially when analyzing sales trends, shipment delays, or performance metrics.

4-

```
[ ] # Create a new feature: Shipping Time (days between order and shipment)
    df["Shipping Time"] = (df["Ship Date"] - df["Order Date"]).dt.days

[ ] # Drop unnecessary columns
    df.drop(columns=["Order ID", "Order Date", "Ship Date"], inplace=True)

[ ] # Fill missing numerical values with median
    df.fillna(df.median(numeric_only=True), inplace=True)
```

This method performs three key data preprocessing steps. First, it calculates a new feature, **"Shipping Time"**, which represents the number of days between the **Order Date** and **Ship Date**. This is crucial for analyzing shipping efficiency, identifying potential delays, and improving logistics performance. Next, it drops unnecessary columns like **Order ID, Order Date, and Ship Date**, which are no longer needed after extracting the shipping time, helping to reduce data complexity and memory usage. Finally, it handles missing values by filling them with the median of numerical columns, ensuring data consistency while preventing bias from extreme values. These steps improve the dataset's quality for further analysis and modeling.

5-

```
▶ # Encode categorical columns
categorical_cols = ["Region", "Country", "Item Type", "Sales Channel", "Order Priority"]
label_encoders = {}
for col in categorical_cols:
    df[col].fillna(df[col].mode()[0], inplace=True)
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
    label_encoders[col] = le
```

This method is encoding categorical columns to prepare the data for machine learning models, which generally require numerical inputs. It first identifies categorical columns such as **Region, Country, Item Type, Sales Channel, and Order Priority**. Missing values in these columns are filled with the most frequently occurring value (mode) to ensure consistency. Then, **Label Encoding** is

applied to convert categorical values into numerical representations. A **LabelEncoder** object is created for each column, which is then used to transform categorical data into integer values. The fitted encoders are stored in the **label_encoders** dictionary for possible inverse transformation later. This step ensures that categorical data is properly formatted for model training.

6-

```
▶ scaler = StandardScaler()
numerical_cols = ["Units Sold", "Unit Price", "Unit Cost", "Total Revenue", "Total Cost", "Shipping Time"]
df[numerical_cols] = scaler.fit_transform(df[numerical_cols])

[ ] profit_median = df["Total Profit"].median()
df["Profit Category"] = np.where(df["Total Profit"] >= profit_median, 1, 0) # 1 = High Profit, 0 = Low Profit
```

This first, it standardizes numerical columns using **StandardScaler()**, ensuring all features have a mean of 0 and a standard deviation of 1, which helps machine learning models converge faster. Second, it categorizes **Total Profit** into "High Profit" (1) and "Low Profit" (0) based on whether it is above or below the median. This simplifies profit analysis and aids classification models.

7-

```
[ ] # Define features (X) and target (y)
X = df.drop(columns=["Total Profit", "Profit Category"]) # Features
y = df["Profit Category"] # Target variable (0 or 1)

[ ] # Split data into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=50)
```

This method is done to prepare the dataset for machine learning. Defining **features (X) and target (y)** is crucial because the model needs independent variables (**X**) to learn patterns and predict the dependent variable (**y**). Removing **Total Profit** and **Profit Category** from **X** ensures that no direct target-related information leaks into the model, preventing biased learning. The **train-test split** is performed to evaluate the model's generalization ability. Training the model on **75%** of the data and testing it on **25%** ensures it can make accurate predictions on unseen data. The **random_state=50** ensures that the data split remains consistent across different runs, leading to reproducible results.

8-

```
# Train Logistic Regression model
model = LogisticRegression(max_iter=5000)
model.fit(X_train, y_train)

LogisticRegression
LogisticRegression(max_iter=5000)

[ ] # Predict categories
y_pred = model.predict(X_test)
```

This code trains and uses a **Logistic Regression model** for classification. The `model.fit(X_train, y_train)` function trains the model using the training data, learning the relationship between features and target labels. The **max_iter=5000** parameter ensures sufficient iterations for convergence, avoiding issues where the model fails to optimize properly.

After training, `model.predict(X_test)` is used to generate predictions on the test data. This step helps assess the model's performance by comparing predicted categories (`y_pred`) with actual values (`y_test`). Logistic Regression is well-suited for binary classification, making it ideal for predicting **Profit Category (0 = Low, 1 = High)** in this case.

9-

```
print(y_pred[:10])

[1 1 1 1 0 0 1 0 0 1]
```

The output `[1 1 1 0 0 1 0 0 1]` represents the **predicted profit categories** for the first 10 test samples. We know that:

- 1 means **High Profit**
- 0 means **Low Profit**

So, the model is predicting which businesses or transactions have high or low profit based on the input data. The sequence suggests that some cases are expected to be highly profitable (1), while others are predicted to have lower profits (0).

10-

```
[ ] from sklearn.metrics import accuracy_score, classification_report

accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Model Accuracy: 0.9880

Classification Report:

	precision	recall	f1-score	support
0	0.99	0.99	0.99	12560
1	0.99	0.99	0.99	12440
accuracy			0.99	25000
macro avg	0.99	0.99	0.99	25000
weighted avg	0.99	0.99	0.99	25000

The displayed results evaluate the performance of a classification model using the accuracy score and a classification report. The model achieves an impressive accuracy of **98.80%**, meaning it correctly predicts profit categories in nearly all test cases. The classification report provides additional metrics, including precision, recall, and F1-score, all of which are **0.99** for both profit categories (0 and 1). This indicates that the model is highly effective at distinguishing between different profit levels, making very few classification errors. The support values show that the dataset is balanced, with approximately equal instances of both categories.

11-

```
✓ 0s from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)
```

Confusion Matrix:

```
[[12434  126]
 [  174 12266]]
```

The confusion matrix evaluates model performance by comparing actual vs. predicted values. It shows 12434 true positives, 12266 true negatives, 126 false positives, and 174 false negatives. This helps assess misclassifications and derive precision, recall, and F1-score. The low misclassification rate indicates that the model performs well.

12-

```
✓ [22] from sklearn.linear_model import LinearRegression  
0s      from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

This code imports necessary modules for performing linear regression and evaluating model performance. `LinearRegression` from `sklearn.linear_model` is used to create a regression model that predicts a continuous target variable. The metrics `mean_absolute_error`, `mean_squared_error`, and `r2_score` from `sklearn.metrics` are used to assess model accuracy. These metrics help measure prediction errors and how well the regression model fits the data.

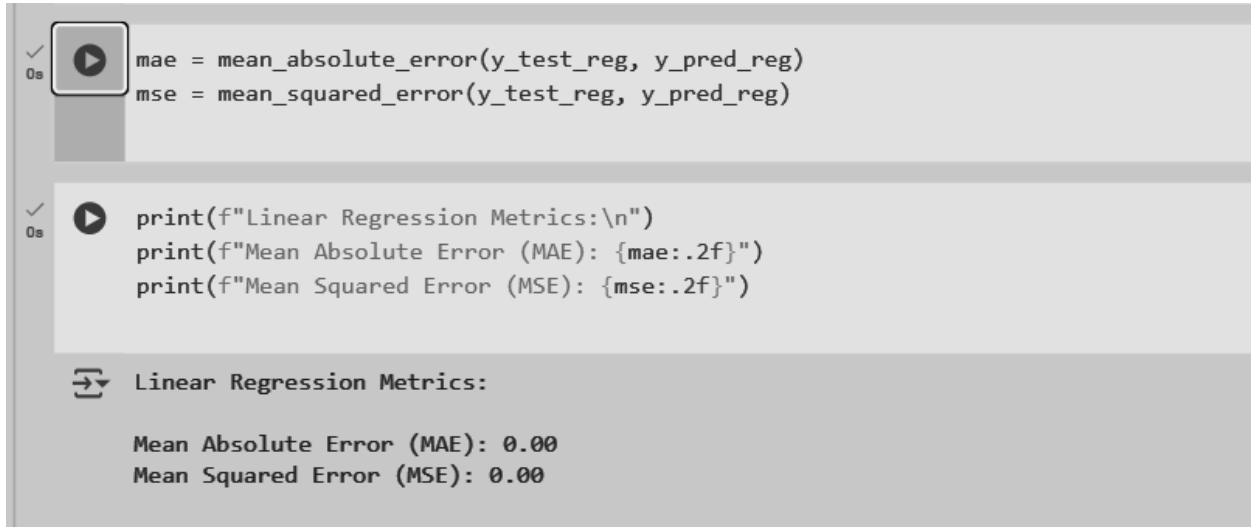
13-

```
✓ [24] y_reg = df["Total Profit"]  
0s  
✓ [25] X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X, y_reg, test_size=0.25, random_state=50)  
0s  
✓ [26] linear_model = LinearRegression()  
0s      linear_model.fit(X_train_reg, y_train_reg)  
      LinearRegression  
      LinearRegression()  
✓ [27] y_pred_reg = linear_model.predict(X_test_reg)  
0s
```

This method performs linear regression to predict "Total Profit" based on input features from the dataset. First, `y_reg = df["Total Profit"]` extracts the target variable. Then, the dataset is split into training and testing sets using `train_test_split()`, with 25% of the data reserved for testing and `random_state=50` ensuring reproducibility. A `LinearRegression` model is created and trained using `linear_model.fit(X_train_reg, y_train_reg)`, learning the relationship between input

features and profit. Finally, predictions for the test set are generated using `linear_model.predict(X_test_reg)`.

14-



The image shows a Jupyter Notebook interface with two code cells. The first cell contains two lines of Python code to calculate the Mean Absolute Error (MAE) and Mean Squared Error (MSE) for a linear regression model. The second cell contains three lines of Python code to print the calculated metrics. Below the code cells, the output of the second cell is displayed, showing the calculated MAE and MSE values, both of which are 0.00.

```
mae = mean_absolute_error(y_test_reg, y_pred_reg)
mse = mean_squared_error(y_test_reg, y_pred_reg)
```

```
print(f"Linear Regression Metrics:\n")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
```

Linear Regression Metrics:

Mean Absolute Error (MAE): 0.00
Mean Squared Error (MSE): 0.00

Conclusion:-The experiment involves training a Logistic Regression model to predict whether a given instance falls into a profit category (0 or 1) based on sales data. The model learns from past data, where each record has various features except for the total profit and profit category. The target variable (Profit Category) is binary, meaning the model predicts either 0 (Low Profit) or 1 (High Profit). After training, the model was tested on new, unseen data, where it made predictions such as [1, 1, 1, 0, 0, 1, 0, 0, 1], indicating which instances belong to the high or low-profit group. The classification report shows that the model is highly accurate (98.80%), meaning it correctly identifies profit categories in most cases. This prediction helps businesses understand sales trends and optimize strategies for better profitability. Also by using Linear regression we predicted the total profit and also we calculated the Mean absolute error and Mean Squared Error