EXPT NO: 1 **A python program to implement univariate regression**

DATE: **bivariate regression and multivariate regression.**

AIM:

To write a python program to implement univariate regression, bivariate regression and multivariate regression.

PROCEDURE:

Implementing univariate, bivariate, and multivariate regression using the Iris dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score
```

Step 2: Load the Iris Dataset

The Iris dataset can be loaded and display the first few rows of the dataset .

```
# Load the Iris dataset

iris = sns.load_dataset('iris')
```

```python
# Display the first few rows of the dataset

print(iris.head())
```

OUTPUT :

```
   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
```

Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```python
# Check for missing values
print(iris.isnull().sum())

# Display the basic statistical details
print(iris.describe())
```

OUTPUT :

```
sepal_length    0
sepal_width     0
petal_length    0
petal_width     0
species         0
dtype: int64
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

Step 4: Univariate Regression

Univariate regression involves predicting one variable based on a single predictor.

4.1 : Select the Features

Choose one feature (e.g., sepal_length) and one target variable (e.g., sepal_width).

```
X_uni = iris[['sepal_length']]
y_uni = iris['sepal_width']
```

4.2 : Split the Data

Split the data into training and testing sets.
Fit the linear regression model on the training data.

```
X_uni_train, X_uni_test, y_uni_train, y_uni_test = train_test_split(X_uni,
y_uni,
test_size=0.2, random_state=42)
```

4.3: Train the model

```
uni_model = LinearRegression()
uni_model.fit(X_uni_train, y_uni_train)
```

```
LinearRegression
LinearRegression()
```

4.4 : Make Predictions

Use the model to make predictions on the test data.

```
y_uni_pred = uni_model.predict(X_uni_test)
```

4.5 : Evaluate the Model

Evaluate the model performance using metrics like Mean Squared Error (MSE) and R-squared.

```
print(f'Univariate MSE: {mean_squared_error(y_uni_test, y_uni_pred)}')
print(f'Univariate R-squared: {r2_score(y_uni_test, y_uni_pred)}')
```

OUTPUT :

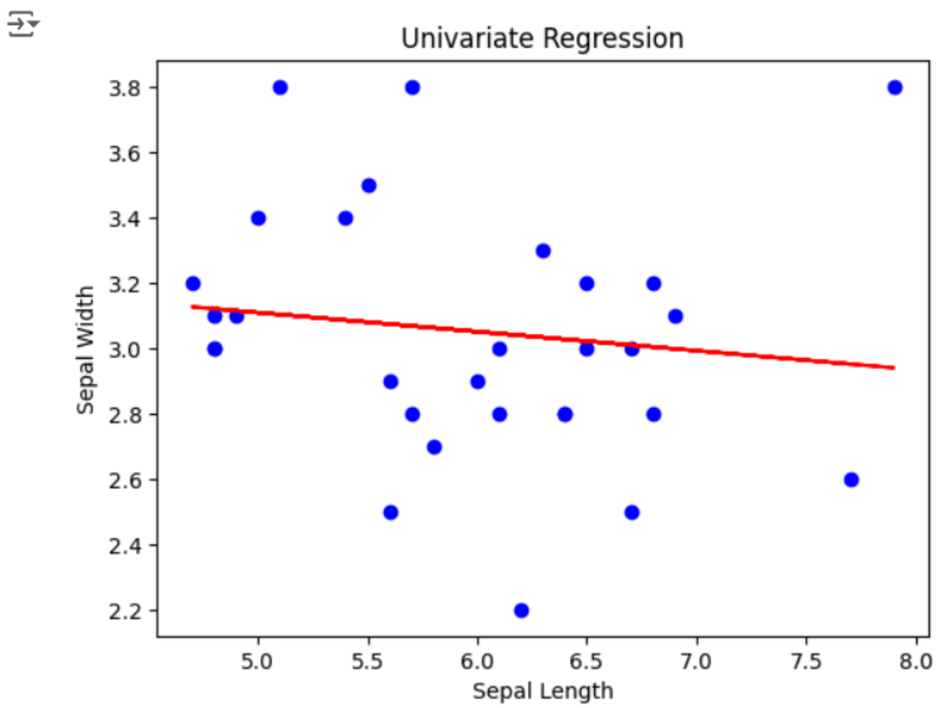
```
Univariate MSE: 0.13961895650579023
Univariate R-squared: 0.024098626473972984
```

4.6 : Visualize the Results

Visualize the relationship between the predictor and the target variable.

```
plt.scatter(X_uni_test, y_uni_test, color='blue')
plt.plot(X_uni_test, y_uni_pred, color='red')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Univariate Regression')
plt.show()
```

OUTPUT :



Step 5 : Bivariate Regression

Bivariate regression involves predicting one variable based on two predictors.

5.1 : Select the Features

Choose two features (e.g., sepal_length, petal_length) and one target variable (e.g., sepal_width).

```
X_bi = iris[['sepal_length', 'petal_length']]

y_bi = iris['sepal_width']
```

5.2 : Split the Data

Split the data into training and testing sets.

```
X_bi_train, X_bi_test, y_bi_train, y_bi_test = train_test_split(X_bi, y_bi,

test_size=0.2, random_state=42)
```

5.3 : Train the Model

Fit the linear regression model on the training data.

```
bi_model = LinearRegression()

bi_model.fit(X_bi_train, y_bi_train)
```

OUTPUT :

```
LinearRegression
LinearRegression()
```

5.4 : Make Predictions

Use the model to make predictions on the test data.

```
y_bi_pred = bi_model.predict(X_bi_test)
```

5.5 : Evaluate the Model

Evaluate the model performance using metrics like MSE and R-squared.

```
print(f'Bivariate MSE: {mean_squared_error(y_bi_test, y_bi_pred)}')
print(f'Bivariate R-squared: {r2_score(y_bi_test, y_bi_pred)}')
```

OUTPUT :

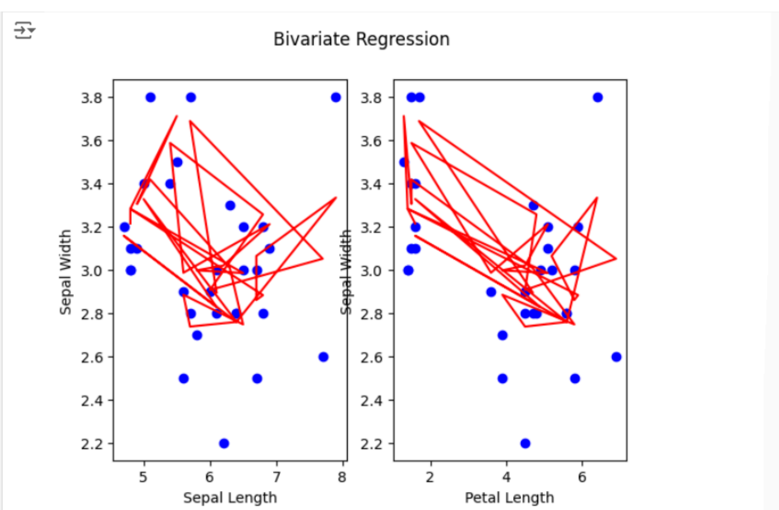
```
Bivariate MSE: 0.08308605032913309
Bivariate R-squared: 0.4192494152204116
```

5.6 : Visualize the Results

Since visualizing in 3D is challenging, we can plot the relationships between the target and each predictor separately.

```
# Sepal Length vs Sepal Width

plt.subplot(1, 2, 1)

plt.scatter(X_bi_test['sepal_length'], y_bi_test, color='blue')

plt.plot(X_bi_test['sepal_length'], y_bi_pred, color='red')

plt.xlabel('Sepal Length')

plt.ylabel('Sepal Width')

# Petal Length vs Sepal Width

plt.subplot(1, 2, 2)

plt.scatter(X_bi_test['petal_length'], y_bi_test, color='blue')

plt.plot(X_bi_test['petal_length'], y_bi_pred, color='red')

plt.xlabel('Petal Length')

plt.ylabel('Sepal Width')

plt.suptitle('Bivariate Regression')

plt.show()
```

OUTPUT :



Step 6: Multivariate Regression

Multivariate regression involves predicting one variable based on multiple predictors.

6.1 : Select the Features

Choose multiple features (e.g., sepal_length, petal_length, petal_width) and one target variable (e.g., sepal_width).

```
X_multi = iris[['sepal_length', 'petal_length', 'petal_width']]

y_multi = iris['sepal_width']
```

6.2 : Split the Data

Split the data into training and testing sets.

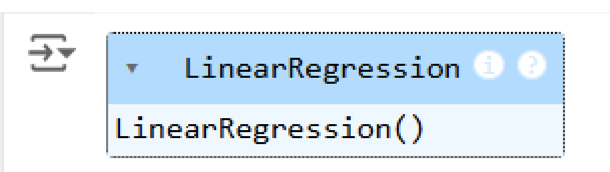
```
X_multi_train, X_multi_test, y_multi_train, y_multi_test =
train_test_split(X_multi,

y_multi, test_size=0.2, random_state=42)
```

6.3 : Train the Model

Fit the linear regression model on the training data.

```
multi_model = LinearRegression()
multi_model.fit(X_multi_train, y_multi_train)
```

OUTPUT :



6.4 : Make Predictions

Use the model to make predictions on the test data.

```
y_multi_pred = multi_model.predict(X_multi_test)
```

6.5 : Evaluate the Model

Evaluate the model performance using metrics like MSE and R-squared.

```
print(f'Multivariate MSE: {mean_squared_error(y_multi_test, y_multi_pred)}')
print(f'Multivariate R-squared: {r2_score(y_multi_test, y_multi_pred)}')
```

OUTPUT :

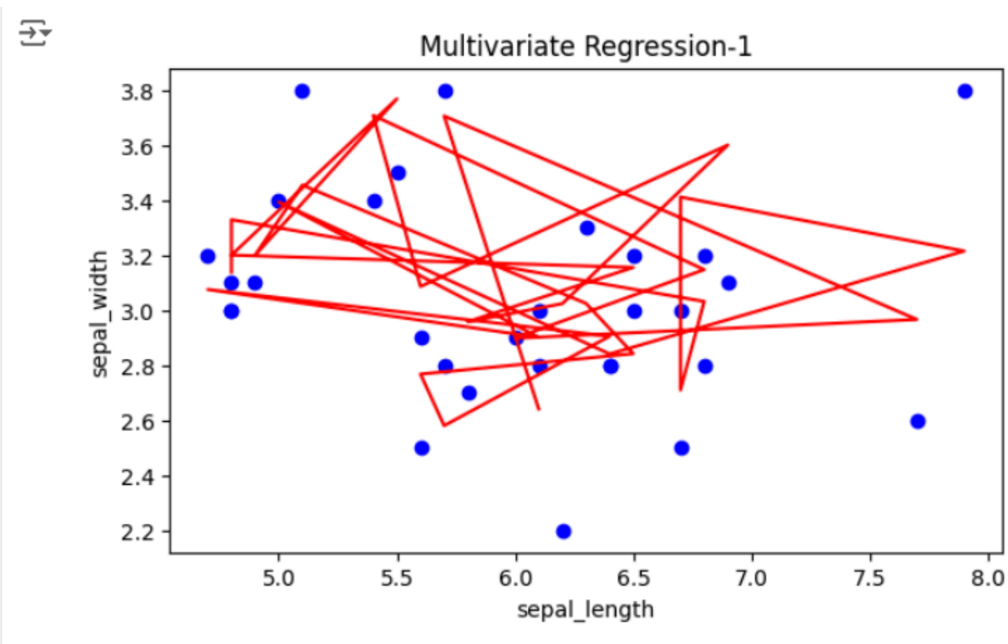
```
Multivariate MSE: 0.0868353771078583
Multivariate R-squared: 0.39304256448374897
```

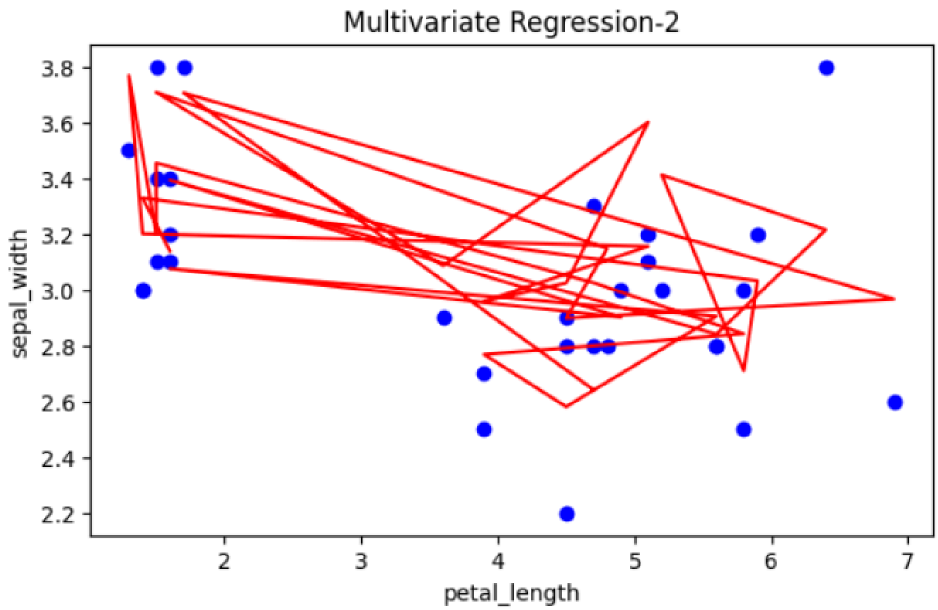
Step 7: Visualize the multivariate regression

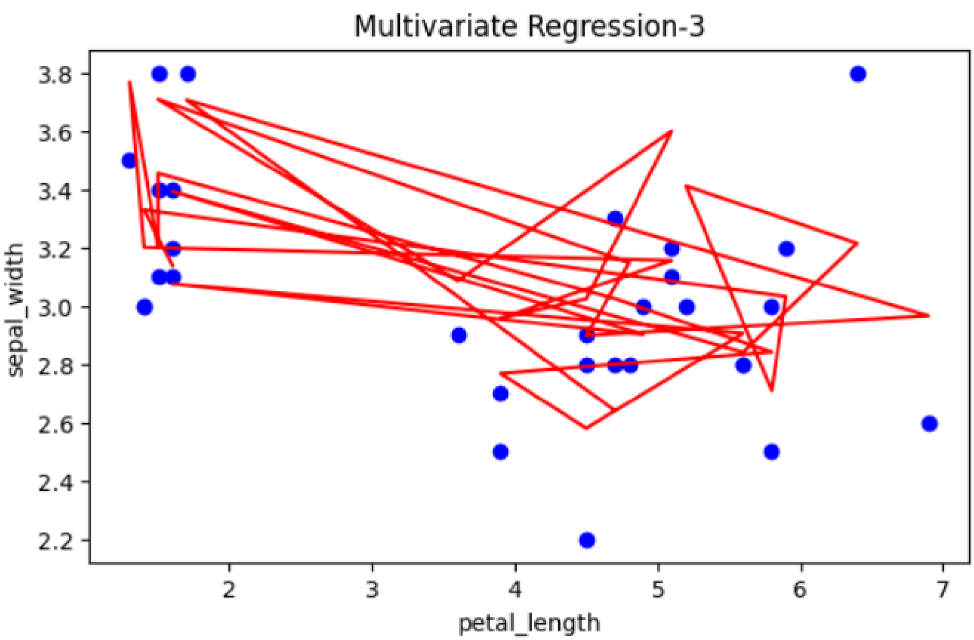
```
plt.figure(figsize=(15,4))

plt.subplot(1, 2, 1)

plt.scatter(X_multi_test['sepal_length'], y_multi_test, color='blue')

plt.plot(X_multi_test['sepal_length'], y_multi_pred, color='red')

plt.xlabel('sepal_length')

plt.ylabel('sepal_width')

plt.title('Multivariate Regression-1')

plt.show()

plt.figure(figsize=(15,4))

plt.subplot(1, 2, 1)

plt.scatter(X_multi_test['petal_length'], y_multi_test, color='blue')

plt.plot(X_multi_test['petal_length'], y_multi_pred, color='red')

plt.xlabel('petal_length')

plt.ylabel('sepal_width')

plt.title('Multivariate Regression-2')

plt.show()
```

```
plt.figure(figsize=(15,4))

plt.subplot(1, 2, 2 )

plt.scatter(X_multi_test['petal_length'], y_multi_test, color='blue')

plt.plot(X_multi_test['petal_length'], y_multi_pred, color='red')

plt.xlabel('petal_length')

plt.ylabel('sepal_width')

plt.title('Multivariate Regression-3')

plt.show()
```

OUTPUT :




Multivariate Regression-2
sepal_width
petal_length
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2
3
4
5
6
7


Multivariate Regression-3
sepal_width
petal_length
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2
3
4
5
6
7

Step 8: Interpret the Results

After implementing and evaluating the models, interpret the coefficients to understand the influence of each predictor on the target variable.

```
print('Univariate Coefficients:', uni_model.coef_)
print('Bivariate Coefficients:', bi_model.coef_)
print('Multivariate Coefficients:', multi_model.coef_)
```

OUTPUT :

```
Univariate Coefficients: [-0.05829418]
Bivariate Coefficients: [ 0.56420418 -0.33942806]
Multivariate Coefficients: [ 0.62934965 -0.63196673  0.6440201 ]
```

RESULT:

This step-by-step process will help us to implement univariate, bivariate, and multivariate regression models using the Iris dataset and analyze their performance.

EXPT NO : 2 **A python program to implement Simple linear**

DATE: **Regression using Least Square Method**

AIM:

To write a python program to implement Simple linear regression using Least Square Method.

PROCEDURE:

Implementing Simple linear regression using Least Square method using the headbrain dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import pandas as pd

import matplotlib.pyplot as plt

import numpy as np
```

Step 2: Load the Iris Dataset

The HeadBrain dataset can be loaded.

```
data = pd.read_csv('/content/headbrain.csv')
```

Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
x,y=np.array(list(data['Head Size(cm^3)'])),np.array(list(data['Brain Weight(grams)']))

print(x[:5],y[:5])
```

OUTPUT :

```
[4512 3738 4261 3777 4177] [1530 1297 1335 1282 1590]
```

Step 4 :Compute the Least Squares Solution

Apply the least squares formula to find the regression coefficients.

```
def get_line(x,y):

 x_m,y_m = np.mean(x), np.mean(y)

 print(x_m,y_m)

 x_d,y_d=x-x_m,y-y_m

 m = np.sum(x_d*y_d)/np.sum(x_d**2)

 c = y_m - (m*x_m)

 print(m, c)

 return lambda x : m*x+c

lin=get_line(x,y)
```

OUTPUT :

```
3633.9915611814345 1282.873417721519
0.2634293394893993 325.5734210494428
```

Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
def get_error(line_fuc, x, y):

 y_m = np.mean(y)

 y_pred = np.array([line_fuc(_) for _ in x])

 ss_t = np.sum((y-y_m)**2)
```

```
 ss_r = np.sum((y-y_pred)**2)

 return 1-(ss_r/ss_t)

get_error(lin, x, y)
```

```
from sklearn.linear_model import LinearRegression

x = x.reshape((len(x),1))

reg=LinearRegression()

reg=reg.fit(x, y)

print(reg.score(x, y))
```

OUTPUT :

```
1.0
```

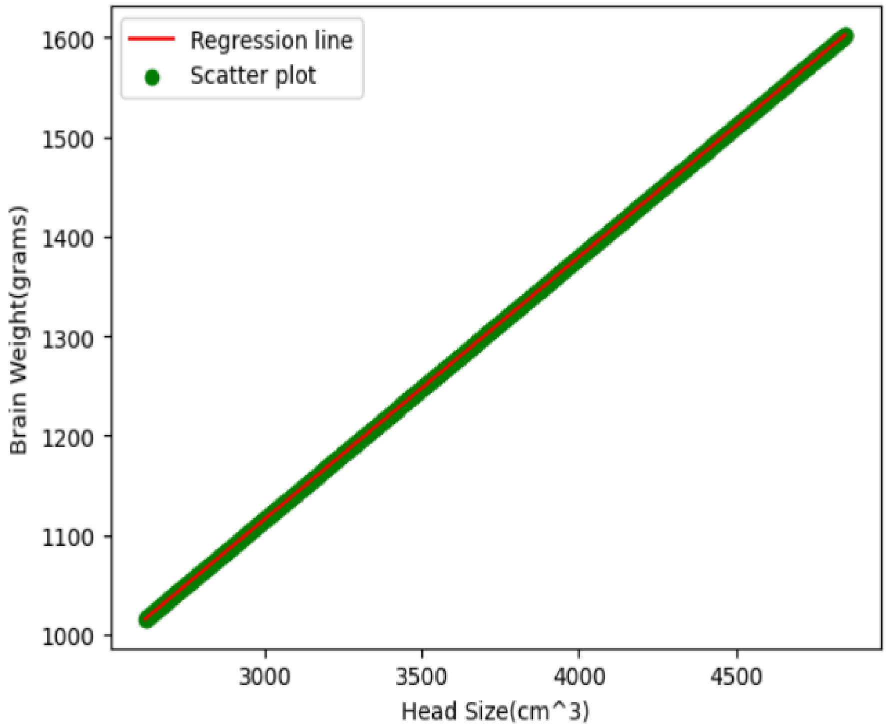
```
1.0
```

Step 6 :Visualize the Results

Plot the original data points and the fitted regression line.

```
x=np.linspace(np.min(x)-100,np.max(x)+100,1000)

y=np.array([lin(x)for x in x])

plt.plot(x, y, color='red', label='Regression line')

plt.scatter(x, y, color='green', label='Scatter plot')

plt.xlabel('Head Size(cm^3)')

plt.ylabel('Brain Weight(grams)')

plt.legend()

plt.show()
```

OUTPUT :



RESULT:

This step-by-step process will help us to implement least square regression models using the HeadBrain dataset and analyze their performance.

EXPT NO : 3 A python program to implement Logistic Model

DATE:

AIM:

To write a python program to implement a Logistic Model.

PROCEDURE:

Implementing Logistic method using the iris dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
# Step 1: Import Necessary Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
```

Step 2: Load the Iris Dataset

The iris dataset can be loaded.

```
# Step 2: Load the Dataset

# For this example, we'll use a built-in dataset from sklearn. You can
replace it with your dataset.

from sklearn.datasets import load_iris
```

```python
# Load the iris dataset

data = load_iris()

X = data.data

y = (data.target == 0).astype(int)  # For binary classification (classifying Iris-setosa)
```

Step 3: Data Preprocessing

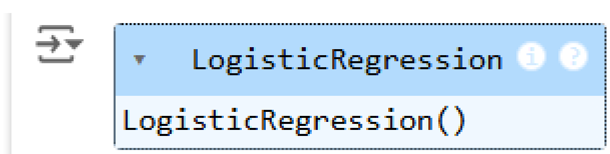
Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```python
# Step 3: Prepare the Data

# Split 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)
```

Step 4 : Train a Model

```python
# Step 4: Create and Train the Model

model = LogisticRegression()

model.fit(X_train, y_train)
```

OUTPUT :



Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```python
# Step 5: Make Predictions

y_pred = model.predict(X_test)
```

Step 6 : Evaluate the Model

Evaluate the model performance.

```
# Step 6: Evaluate the Model

accuracy = accuracy_score(y_test, y_pred)

conf_matrix = confusion_matrix(y_test, y_pred)

class_report = classification_report(y_test, y_pred)

# Print evaluation metrics

print(f"Accuracy: {accuracy}")

print("Confusion Matrix:")

print(conf_matrix)

print("Classification Report:")

print(class_report)
```

OUTPUT :

```
Accuracy: 1.0
Confusion Matrix:
[[20  0]
 [ 0 10]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        20
           1       1.00      1.00      1.00        10

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

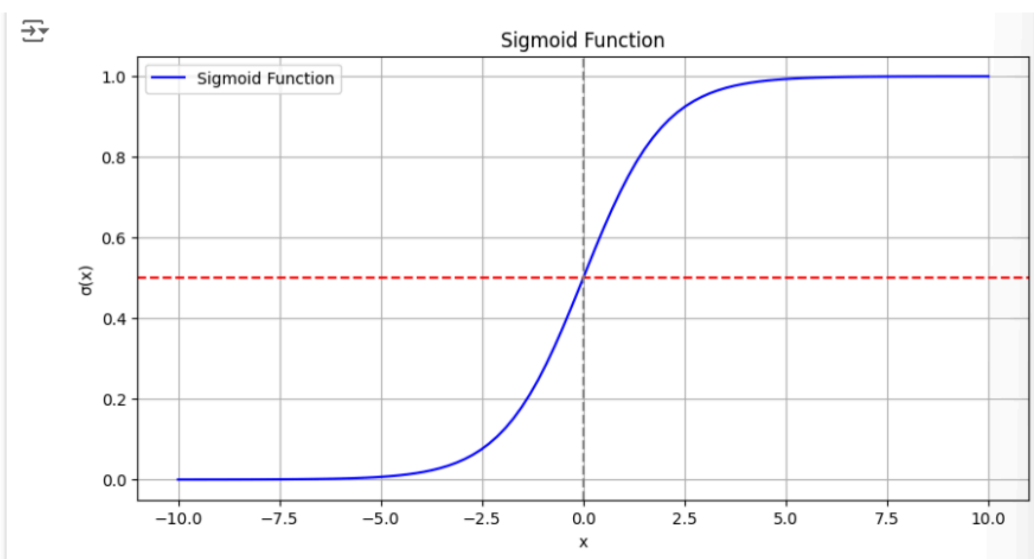
Step 7 :Visualize the Results

Plot the original data points and the fitted regression line.

```
# Step 7: Visualize Results (Optional)

x_values = np.linspace(-10, 10, 100)
```

```
sigmoid_values = 1 / (1 + np.exp(-x_values))


# Plot the sigmoid function

plt.figure(figsize=(10, 5))

plt.plot(x_values, sigmoid_values, label='Sigmoid Function', color='blue')

plt.title('Sigmoid Function')

plt.xlabel('x')

plt.ylabel('σ(x)')

plt.grid()

plt.axhline(0.5, color='red', linestyle='--')  # Line at y=0.5

plt.axvline(0, color='gray', linestyle='--')  # Line at x=0

plt.legend()

plt.show()
```

OUTPUT :



RESULT:

This step-by-step process will help us to implement Logistic models using the Iris dataset and analyze their performance.

EXPT NO : 4 **A python program to implement Single Layer**

DATE: **Perceptron**

AIM:

To write a python program to implement Single layer perceptron.

PROCEDURE:

Implementing Single layer perceptron method using the Keras dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import pandas as pd

from tensorflow import keras

import matplotlib.pyplot as plt
```

Step 2: Load the Keras Dataset

The Keras dataset can be loaded.

```
(X_train,y_train),(X_test,y_test)=keras.datasets.mnist.load_data()
```

Step 3: Data Preprocessing

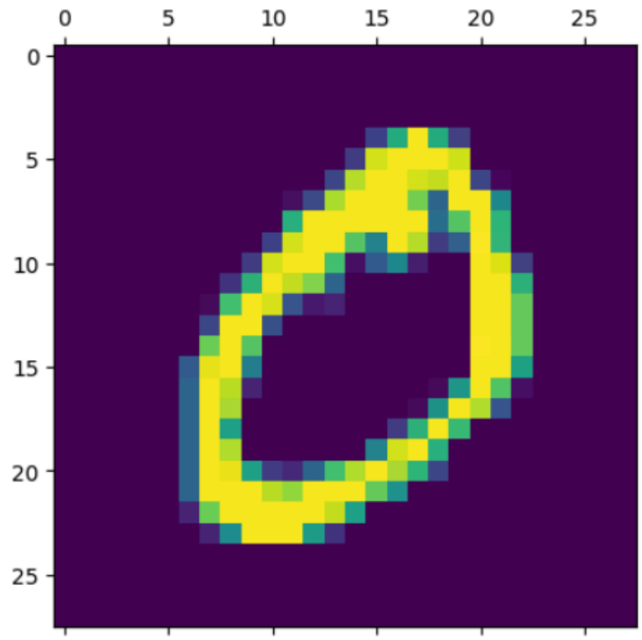
Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
print(f"Training set:{X_train.shape}")

print(f"Testing set:{X_test.shape}")



print(X_train[1].shape)
```

```
plt.matshow(X_train[1])
```

OUTPUT :

```
Training set:(60000, 28, 28)
Testing set:(10000, 28, 28)
(28, 28)
<matplotlib.image.AxesImage at 0x7b83d00435e0>
```



Step 4 : Train a Model

```
#Normalizing the dataset

x_train=X_train/255

x_test=X_test/255



#Flatting the dataset in order to compute for model building

x_train_flatten=x_train.reshape(len(x_train),28*28)

x_test_flatten=x_test.reshape(len(x_test),28*28)

x_train_flatten.shape
```

Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
model=keras.Sequential([

    keras.layers.Dense(10,input_shape=(784,),

                      activation='sigmoid')

])



model.compile(

     optimizer='adam',

     loss='sparse_categorical_crossentropy',

     metrics=['accuracy'])



model.fit(x_train_flatten,y_train,epochs=5



          )
```

OUTPUT :

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8180 - loss: 0.7118
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9148 - loss: 0.3101
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 956us/step - accuracy: 0.9238 - loss: 0.2769
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 940us/step - accuracy: 0.9250 - loss: 0.2744
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 990us/step - accuracy: 0.9239 - loss: 0.2706
<keras.src.callbacks.history.History at 0x7b83d00c6a70>
```

Step 6 : Evaluate the Model

Evaluate the model performance.

```
model.evaluate(x_test_flatten,y_test)
```

OUTPUT :

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9138 - loss: 0.3021
[0.26686596870422363, 0.9257000088691711]
```

RESULT:

This step-by-step process will help us to implement Single Layer Perceptron models using the Keras dataset and analyze their performance.

EXPT NO : 5 **A python program to implement Multi Layer**

DATE: **Perceptron With Backpropagation**

AIM:

To write a python program to implement Multilayer perceptron with backpropagation .

PROCEDURE:

Implementing Multilayer perceptron with backpropagation using the Keras dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
# importing modules

import tensorflow as tf

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Activation

import matplotlib.pyplot as plt
```

Step 2: Load the Keras Dataset

The Keras dataset can be loaded.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

OUTPUT :

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```python
# Cast the records into float values

x_train = x_train.astype('float32')

x_test = x_test.astype('float32')



# normalize image pixel values by dividing

# by 255

gray_scale = 255

x_train /= gray_scale

x_test /= gray_scale



print("Feature matrix:", x_train.shape)

print("Target matrix:", x_test.shape)

print("Feature matrix:", y_train.shape)

print("Target matrix:", y_test.shape)
```

OUTPUT :

```
Feature matrix: (60000, 28, 28)
Target matrix: (10000, 28, 28)
Feature matrix: (60000,)
Target matrix: (10000,)
```

Step 4 : Train a Model

```
model = Sequential([

  # reshape 28 row * 28 column data to 28*28 rows

  Flatten(input_shape=(28, 28)),

  # dense layer 1

  Dense(256, activation='sigmoid'),

  # dense layer 2

  Dense(128, activation='sigmoid'),

  # output layer

  Dense(10, activation='sigmoid'),

])
```

OUTPUT:

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning:
  super().__init__(**kwargs)
```

Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
model.compile(optimizer='adam',

      loss='sparse_categorical_crossentropy',
```

```
      metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10,

    batch_size=2000,

    validation_split=0.2)
```

OUTPUT:

```
Epoch 1/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 5s 115ms/step - accuracy: 0.3546 - loss: 2.1596 - val_accuracy: 0.68
Epoch 2/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 4s 53ms/step - accuracy: 0.7116 - loss: 1.3743 - val_accuracy: 0.82
Epoch 3/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - accuracy: 0.8221 - loss: 0.8221 - val_accuracy: 0.872
Epoch 4/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 65ms/step - accuracy: 0.8720 - loss: 0.5676 - val_accuracy: 0.892
Epoch 5/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 99ms/step - accuracy: 0.8907 - loss: 0.4444 - val_accuracy: 0.902
Epoch 6/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 102ms/step - accuracy: 0.8993 - loss: 0.3852 - val_accuracy: 0.91
Epoch 7/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 104ms/step - accuracy: 0.9088 - loss: 0.3416 - val_accuracy: 0.91
Epoch 8/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - accuracy: 0.9119 - loss: 0.3188 - val_accuracy: 0.922
Epoch 9/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - accuracy: 0.9191 - loss: 0.2911 - val_accuracy: 0.92
Epoch 10/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 99ms/step - accuracy: 0.9245 - loss: 0.2704 - val_accuracy: 0.92
<keras.src.callbacks.history.History at 0x7d9ca1406a40>
```

Step 6 : Evaluate the Model

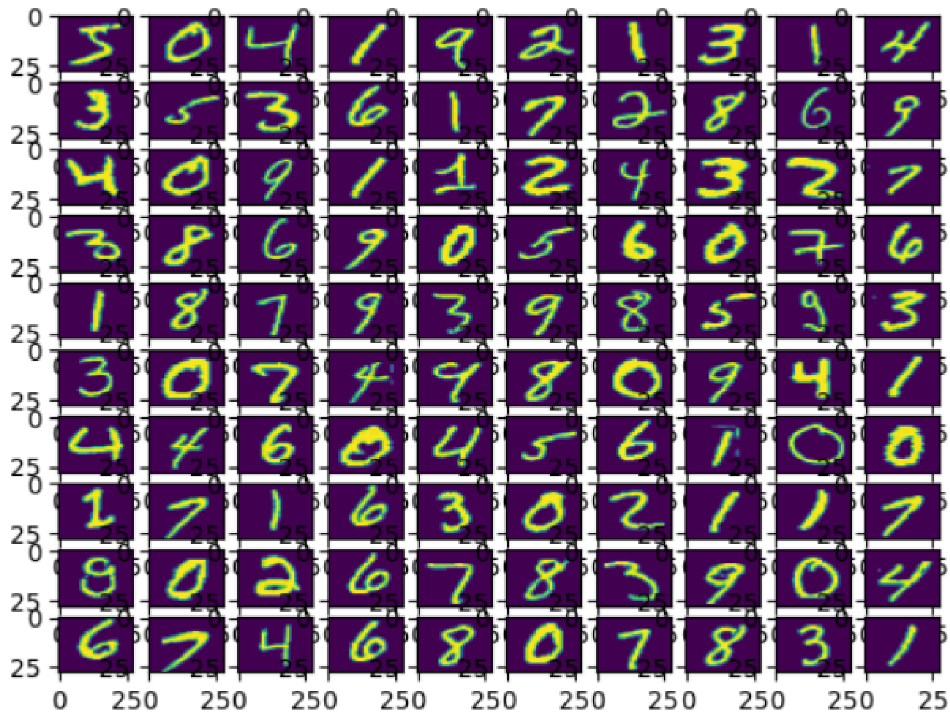
Evaluate the model performance.

```
results = model.evaluate(x_test, y_test, verbose = 0)

print('test loss, test acc:', results)

fig, ax = plt.subplots(10, 10)

k = 0

for i in range(10):

  for j in range(10):

    ax[i][j].imshow(x_train[k].reshape(28, 28),
```

```
            aspect='auto')

    k += 1

plt.show()
```

OUTPUT :

```
test loss, test acc: [0.2589016258716583, 0.9277999997138977]
```



RESULT:

This step-by-step process will help us to implement MultiLayer Perceptron with Backpropagation models using the Keras dataset and analyze their performance.