

▼ BITS F464 - Semester 1 - MACHINE LEARNING

ASSIGNMENT 1 - LINEAR MODELS FOR REGRESSION AND CLASSIFICATION

Team number:

(In Title case, separated with commas) **Full names of all students in the team:**

(Separated by commas) **Id number of all students in the team:**

This assignment aims to identify the differences between three sets of Machine Learning models.

▼ 1. Dataset Generation

You are given a sample Diabetes dataset. Using this, please develop your own dataset consisting of 500 records. You can use the given code to generate your own dataset. Submit the generated dataset as a .csv file along with your python notebook.

```
!pip install --upgrade ipython #run this line if you are getting a version issue
%pip install sdv
%pip install urllib3 == 1.26.7
```

```
Requirement already satisfied: traitlets>=5 in /usr/local/lib/python3.10/dist-packages (from ipython) (5.7.1)
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from ipython) (1.1.3)
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dist-packages (from ipython) (4.8.0)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ipython) (0.8.3)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.10/dist-packages (from pexpect>4.3->ipython) (0.7.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30->ipython) (0.2.2)
Requirement already satisfied: executing>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from stack-data->ipython) (2.0.0)
Requirement already satisfied: asttokens>=2.1.0 in /usr/local/lib/python3.10/dist-packages (from stack-data->ipython) (2.4.0)
Requirement already satisfied: pure-eval in /usr/local/lib/python3.10/dist-packages (from stack-data->ipython) (0.2.2)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from asttokens>=2.1.0->stack-data->ipython) (1.16.0)
Requirement already satisfied: sdv in /usr/local/lib/python3.10/dist-packages (1.4.0)
Requirement already satisfied: boto3<2,>=1.15.0 in /usr/local/lib/python3.10/dist-packages (from sdv) (1.28.57)
Requirement already satisfied: botocore<2,>=1.18 in /usr/local/lib/python3.10/dist-packages (from sdv) (1.31.57)
Requirement already satisfied: cloudpickle<3.0,>=2.1.0 in /usr/local/lib/python3.10/dist-packages (from sdv) (2.2.1)
Requirement already satisfied: Faker<15,>=10 in /usr/local/lib/python3.10/dist-packages (from sdv) (14.2.1)
Requirement already satisfied: graphviz<1,>=0.13.2 in /usr/local/lib/python3.10/dist-packages (from sdv) (0.20.1)
Requirement already satisfied: tqdm<5,>=4.15 in /usr/local/lib/python3.10/dist-packages (from sdv) (4.66.1)
Requirement already satisfied: copulas<0.10,>=0.9.0 in /usr/local/lib/python3.10/dist-packages (from sdv) (0.9.1)
Requirement already satisfied: ctgan<0.8,>=0.7.4 in /usr/local/lib/python3.10/dist-packages (from sdv) (0.7.4)
Requirement already satisfied: deepecho<0.5,>=0.4.2 in /usr/local/lib/python3.10/dist-packages (from sdv) (0.4.2)
Requirement already satisfied: rdt<2,>=1.7.0 in /usr/local/lib/python3.10/dist-packages (from sdv) (1.7.0)
Requirement already satisfied: sdmetrics<0.12,>=0.11.0 in /usr/local/lib/python3.10/dist-packages (from sdv) (0.11.1)
Requirement already satisfied: numpy<1.25.0,>=1.23.3 in /usr/local/lib/python3.10/dist-packages (from sdv) (1.23.5)
Requirement already satisfied: pandas>=1.3.4 in /usr/local/lib/python3.10/dist-packages (from sdv) (1.5.3)
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from boto3<2,>=1.15.0->sdv) (1.0.1)
Requirement already satisfied: s3transfer<0.8.0,>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from boto3<2,>=1.15.0->sdv) (0.7.0)
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/local/lib/python3.10/dist-packages (from botocore<2,>=1.18->sdv) (2.8.2)
Requirement already satisfied: urllib3<1.27,>=1.25.4 in /usr/local/lib/python3.10/dist-packages (from botocore<2,>=1.18->sdv) (1.2.6)
Requirement already satisfied: matplotlib<4,>=3.6.0 in /usr/local/lib/python3.10/dist-packages (from copulas<0.10,>=0.9.0->sdv) (3.6.0)
Requirement already satisfied: scipy<2,>=1.9.2 in /usr/local/lib/python3.10/dist-packages (from copulas<0.10,>=0.9.0->sdv) (1.11.0)
Requirement already satisfied: scikit-learn<2,>=1.1.3 in /usr/local/lib/python3.10/dist-packages (from ctgan<0.8,>=0.7.4->sdv) (1.3.2)
Requirement already satisfied: torch>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from ctgan<0.8,>=0.7.4->sdv) (2.0.1+cu118)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.3.4->sdv) (2023.3.post1)
Requirement already satisfied: psutil<6,>=5.7 in /usr/local/lib/python3.10/dist-packages (from rdt<2,>=1.7.0->sdv) (5.9.5)
Requirement already satisfied: plotly<6,>=5.10.0 in /usr/local/lib/python3.10/dist-packages (from sdmetrics<0.12,>=0.11.0->sdv) (5.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (1.0.7)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (4.22.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (23.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (9.5.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib<4,>=3.6.0->copulas<0.10,>=0.9.0->sdv) (3.1.1)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from plotly<6,>=5.10.0->sdmetrics<0.12,>=0.11.0->sdv) (8.1.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil<3.0.0,>=2.1->botocore<2,>=1.18->sdv) (1.16.0)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn<2,>=1.1.3->ctgan<0.8,>=0.7.4->sdv) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn<2,>=1.1.3->ctgan<0.8,>=0.7.4->sdv) (3.1.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch>=1.11.0->ctgan<0.8,>=0.7.4->sdv) (3.12.2)
```

```

from sdv.datasets.local import load_csvs
datasets = load_csvs(folder_name="/content/") #change folder location

diabetes_table = datasets['diabetes2_csv']

from sdv.metadata import SingleTableMetadata

metadata = SingleTableMetadata()

metadata.detect_from_csv(filepath="/content/diabetes2_csv.csv")    #change file path

print(type(metadata))

<class 'sdv.metadata.single_table.SingleTableMetadata'>

diabetes_table.head()

```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```

from sdv.lite import SingleTablePreset

synthesizer = SingleTablePreset(
    metadata,
    name='FAST_ML'
)

```

```
# !pip install --upgrade ipython
```

```

synthesizer.fit(
    data=diabetes_table
)

```

```

synthetic_data = synthesizer.sample(
    num_rows=500
)

```

```
synthetic_data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	7	149	96	19	10	38.387409	0.561331	40	1
1	0	151	44	6	105	26.125923	0.463959	27	1
2	3	169	57	24	240	33.224573	0.541364	36	0
3	4	86	61	35	0	32.918264	0.526311	39	0
4	6	75	62	31	77	37.453830	0.178734	21	0

```
synthetic_data.shape
```

```
(500, 9)
```

```
sensitive_column_names = ['Pregnancies', 'Glucose', 'BloodPressure',
                          'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction']
```

```
diabetes_table[sensitive_column_names].head(3)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	
0	6	148	72	35	0	33.6	0.627	
1	1	85	66	29	0	26.6	0.351	
2	8	183	64	0	0	23.3	0.672	

```
synthesizer.save('my_synthesizer.pkl')
```

```
synthesizer = SingleTablePreset.load('my_synthesizer.pkl')
```

```
synthetic_data.to_csv("/content/generated_00.csv", index=False)
```

2. Preprocess and perform exploratory data analysis of the dataset obtained

We have considered the following in Preprocessing:

1. We see that the dataset doesn't have None values, they were actually written as zero.
2. Of the following features only pregnancies could take 0 as feasible value and even target could have 0 or 1 as outcome, hence we have replaced all 0s of other columns with None then replaced them with average values of the columns.
3. We also removed any duplicate data present

```
import numpy as np
```

```
dt1 = synthetic_data.drop(columns=['Outcome', 'Pregnancies'], axis=1)
```

```
# diabetes_table.iloc[:,1:7].replace(0, np.nan, inplace=True)
dt1.replace(0, np.nan, inplace=True)
```

```
dt1.head()
```

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	
0	149	96	19.0	10.0	38.387409	0.561331	40	
1	151	44	6.0	105.0	26.125923	0.463959	27	
2	169	57	24.0	240.0	33.224573	0.541364	36	
3	86	61	35.0	NaN	32.918264	0.526311	39	
4	75	62	31.0	77.0	37.453830	0.178734	21	

```
dt1["Outcome"] = synthetic_data["Outcome"]
dt1["Pregnancies"] = synthetic_data["Pregnancies"]
```

```
last_column_name = dt1.columns[-1]
```

```
# Extract the last column
last_column = dt1.pop(last_column_name)
```

```
# Insert the last column as the first column
dt1.insert(0, last_column_name, last_column)
```

```
dt1.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	7	149	96	19.0	10.0	38.387409	0.561331	40	1
1	0	151	44	6.0	105.0	26.125923	0.463959	27	1
2	3	169	57	24.0	240.0	33.224573	0.541364	36	0
3	4	86	61	35.0	NaN	32.918264	0.526311	39	0
4	6	75	62	31.0	77.0	37.453830	0.178734	21	0

```
synthetic_data = dt1
```

```
synthetic_data['Insulin'].value_counts(dropna=False)
```

```

NaN      124
106.0     5
35.0      5
68.0      4
63.0      4
...
378.0     1
206.0     1
125.0     1
22.0      1
297.0     1
Name: Insulin, Length: 216, dtype: int64
```

```
synthetic_data['Pregnancies'].fillna(
    diabetes_table['Pregnancies'].mean(), inplace=True)
```

```
synthetic_data.drop_duplicates(inplace=True)
```

```
synthetic_data['Glucose'].fillna(
    synthetic_data['Glucose'].mean(), inplace=True)
```

```
synthetic_data['BloodPressure'].fillna(
    synthetic_data['BloodPressure'].mean(), inplace=True)
```

```
synthetic_data['SkinThickness'].fillna(
    synthetic_data['SkinThickness'].mean(), inplace=True)
```

```
synthetic_data['Insulin'].fillna(
    synthetic_data['Insulin'].mean(), inplace=True)
```

```
synthetic_data['BMI'].fillna(synthetic_data['BMI'].mean(), inplace=True)
```

```
synthetic_data['DiabetesPedigreeFunction'].fillna(
    synthetic_data['DiabetesPedigreeFunction'].mean(), inplace=True)
```

```
synthetic_data['Age'].fillna(synthetic_data['Age'].mean(), inplace=True)
```

```
synthetic_data['Outcome'].fillna(
    synthetic_data['Outcome'].mean(), inplace=True)
```

```
synthetic_data
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	7	149	96	19.0	10.000	38.387409	0.561331	40	1
1	0	151	44	6.0	105.000	26.125923	0.463959	27	1
2	3	169	57	24.0	240.000	33.224573	0.541364	36	0
3	4	86	61	35.0	129.375	32.918264	0.526311	39	0
4	6	75	62	31.0	77.000	37.453830	0.178734	21	0
...
495	0	101	87	46.0	297.000	38.732291	0.421418	21	0
496	4	124	72	28.0	27.000	33.576185	0.630074	50	0
497	4	181	35	38.0	249.000	35.137482	0.434926	35	1
498	4	152	81	49.0	165.000	36.694959	0.820895	45	1

Here we are further plotting graphs and checking how our data is doing, in the process we also checked for any outliers still present as synthetic data generation automatically removes them.

```
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import rcParams

from sklearn import model_selection
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_auc_score
from sklearn.metrics import f1_score, confusion_matrix, precision_recall_curve, roc_curve
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.preprocessing import StandardScaler

import plotly.express as px
from plotly.subplots import make_subplots
import plotly.graph_objects as go

import warnings
warnings.filterwarnings(action='ignore')

def get_clf_eval(y_test, pred=None, pred_proba=None):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)

    roc_auc = roc_auc_score(y_test, pred_proba)

    # ROC-AUC print
    print('accuracy: {0:.4f}, precision: {1:.4f}, recall: {2:.4f},\
    F1: {3:.4f}, AUC:{4:.4f}'.format(accuracy, precision, recall, f1, roc_auc))
    return confusion

synthetic_data.to_csv("/content/generated_05.csv", index=False) #changed file name

diabetes_df = pd.read_csv("/content/generated_05.csv")
diabetes_df.head().T.style.set_properties(**{'background-color': 'grey',
                                             'color': 'white',
                                             'border-color': 'white'})
```

	0	1	2	3	4
Pregnancies	7.000000	0.000000	2.000000	4.000000	6.000000

Here we are checking the target imbalance

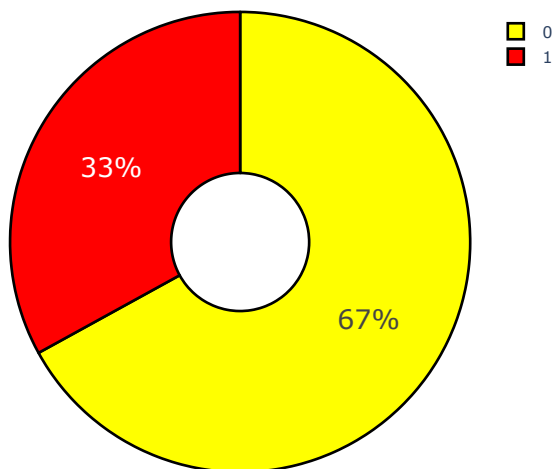
```

colors = ['yellow', 'red']
labels = ['0', '1']
values = diabetes_df['Outcome'].value_counts()/diabetes_df['Outcome'].shape[0]

# Use `hole` to create a donut-like pie chart
fig = go.Figure(data=[go.Pie(labels=labels, values=values, hole=.3)])
fig.update_traces(hoverinfo='label+percent', textinfo='percent', textfont_size=20,
                  marker=dict(colors=colors, line=dict(color='#000000', width=2)))
fig.update_layout(
    title_text="Outcome")
fig.show()

```

Outcome



Details of various columns

```

def highlight_min(s, props=''):
    return np.where(s == np.nanmin(s.values), props, '')

diabetes_df.describe().style.apply(highlight_min, props='color:Black;background-color:Grey', axis=0)

```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000	500.000000
mean	4.006000	120.454000	71.080000	23.931868	129.375000	32.518053	0.511352	34.450000	0.330000
std	3.052967	33.702754	18.964517	12.110620	73.304675	7.887027	0.299594	10.082600	0.470684
min	0.000000	14.000000	22.000000	1.000000	1.000000	10.903340	0.078000	21.000000	0.000000
25%	2.000000	98.000000	58.000000	15.000000	77.000000	27.161187	0.255483	27.000000	0.000000
50%	4.000000	120.500000	71.000000	23.931868	129.375000	32.593184	0.508918	33.000000	0.000000
75%	6.000000	145.000000	85.000000	32.000000	165.000000	37.500736	0.735819	41.000000	1.000000
max	17.000000	199.000000	122.000000	55.000000	378.000000	59.937412	1.413874	73.000000	1.000000

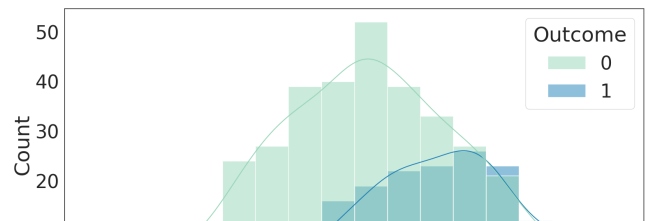
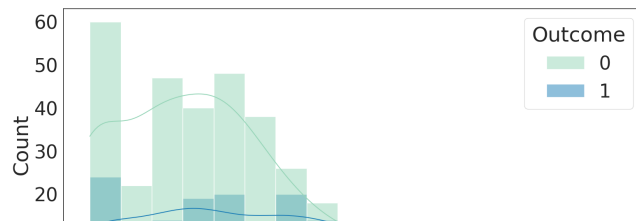
Here, we are checking if we still are left with any outliers even after synthetic data removes them

```

feature_names = [cname for cname in diabetes_df.loc[:, 'Age'].columns]
rcParams['figure.figsize'] = 40,60
sns.set(font_scale = 3)
sns.set_style("white")
sns.set_palette("bright")
plt.subplots_adjust(hspace=0.5)

```

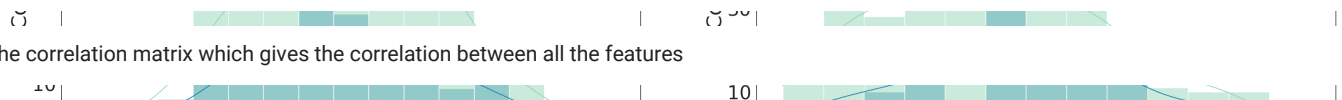
```
i = 1;
for name in feature_names:
    plt.subplot(5,2,i)
    sns.histplot(data=diabetes_df, x=name, hue="Outcome",kde=True,palette="YlGnBu")
    i = i + 1
```



```
zero_features = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
total_count = diabetes_df['Glucose'].count()
```

```
for feature in zero_features:
    zero_count = diabetes_df[diabetes_df[feature]==0][feature].count()
    print('{0} 0 number of cases {1}, percent is {2:.2f} %'.format(feature, zero_count, 100*zero_count/total_count))
```

```
Pregnancies 0 number of cases 84, percent is 16.80 %
Glucose 0 number of cases 0, percent is 0.00 %
BloodPressure 0 number of cases 0, percent is 0.00 %
SkinThickness 0 number of cases 0, percent is 0.00 %
Insulin 0 number of cases 0, percent is 0.00 %
BMI 0 number of cases 0, percent is 0.00 %
```



This the correlation matrix which gives the correlation between all the features



```
corr=diabetes_df.corr().round(2)
```

```
sns.set(font_scale=1.15)
plt.figure(figsize=(14, 10))
sns.set_palette("bright")
sns.set_style("white")
mask = np.zeros_like(corr)
mask[np.triu_indices_from(mask)] = True
sns.heatmap(corr, annot=True, cmap='gist_yarg_r', mask=mask, cbar=True)
plt.title('Correlation Plot')
```



```
Text(0.5, 1.0, 'Correlation Plot')
```

Correlation Plot

Pregnancies

Glucose

BloodPressure

0.06

0.08

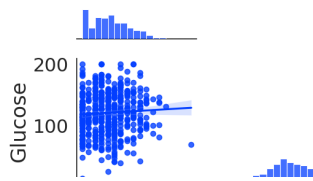
0.13

0.5

0.4

```
sns.set(font_scale=2)
plt.figure(figsize=(10, 8))
sns.set_style("white")
sns.set_palette("bright")
sns.pairplot(diabetes_df, kind = 'reg', corner = True, palette = 'YlGnBu' )
```

```
<seaborn.axisgrid.PairGrid at 0x78d498ccac0>
<Figure size 1000x800 with 0 Axes>
```



```
synthetic_data.to_csv("/content/generated_04.csv", index=False) #changed file name
```



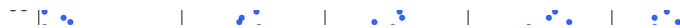
```
synthetic_data['Insulin'].value_counts(dropna=False)
```

```
129.375    124
106.000     5
35.000     5
68.000     4
63.000     4
```

```
...
```

```
378.000     1
206.000     1
125.000     1
22.000      1
297.000     1
```

```
Name: Insulin, Length: 216, dtype: int64
```



```
synthetic_data.shape
```

```
(500, 9)
```



3. Comparison of Stochastic Gradient Descent and Batch Gradient Descent using Linear Regression

Stochastic Gradient Descent



```
import csv
import random
import numpy as np
import matplotlib.pyplot as plt
```

```
# Step 1: Load the data from the CSV file
```

```
def load_data(filename):
```

```
    X, Y = [], []
```

```
    with open(filename, 'r') as csvfile:
```

```
        reader = csv.reader(csvfile)
```

```
        next(reader) # Skip the header row if it exists
```

```
        for row in reader:
```

```
            data_point = [float(val) for val in row]
```

```
            X.append(data_point[:-1]) # All columns except the last one are features
```

```
            Y.append(data_point[-1]) # The last column is the target
```

```
    return np.array(X), np.array(Y)
```

```
def standardize_data(X):
```

```
    mean = np.mean(X, axis=0)
```

```
    std_dev = np.std(X, axis=0)
```

```
    standardized_X = (X - mean) / std_dev
```

```
    return standardized_X
```

```
# Step 2: Define the multivariate linear regression model
```

```
def multivariate_linear_regression(X, Y, learning_rate=0.001, epochs=200):
```

```
    n_samples = len(X)
```

```
    n_features = len(X[0])
```

```
    weights = np.zeros(n_features) # Initialize weights with zeros
```

```
    bias = 0 # Initialize bias
```

```
    cost_history = [] # List to store the cost at each iteration
```

```
    for epoch in range(epochs):
```

```
        for i in range(n_samples):
```

```
            x = X[i]
```

```

    y = Y[i]

    # Compute the predicted value
    y_pred = np.dot(x, weights) + bias

    # Compute the gradient of the loss with respect to weights and bias
    error = y_pred - y
    d_weights = (1/n_samples) * np.dot(x, error)
    d_bias = (1/n_samples) * np.sum(error)

    # Update the model parameters using the learning rate
    weights -= learning_rate * d_weights
    bias -= learning_rate * d_bias

    # Calculate and store the cost (MSE) at each epoch
    mse = calculate_mse(X, Y, weights, bias)
    cost_history.append(mse)

return weights, bias, cost_history

# Step 3: Calculate Mean Squared Error (MSE)
def calculate_mse(X, Y, weights, bias):
    n_samples = len(X)
    mse = np.sum((np.dot(X, weights) + bias - Y) ** 2) / n_samples
    return mse

# Step 4: Calculate R-squared (R2) score
def calculate_r2_score(Y_true, Y_pred):
    Y_true = np.array(Y_true)
    Y_pred = np.array(Y_pred)
    y_mean = np.mean(Y_true)
    ss_res = np.sum((Y_true - Y_pred) ** 2)
    ss_tot = np.sum((Y_true - y_mean) ** 2)
    r2 = 1 - (ss_res / ss_tot)
    return r2

def train_test_split(X, Y, test_size=0.2):
    n = len(X)
    test_indices = random.sample(range(n), int(test_size * n))
    train_indices = [i for i in range(n) if i not in test_indices]
    X_train = [X[i] for i in train_indices]
    Y_train = [Y[i] for i in train_indices]
    X_test = [X[i] for i in test_indices]
    Y_test = [Y[i] for i in test_indices]
    return X_train, Y_train, X_test, Y_test

# Step 5: Load the data and perform multivariate linear regression
if __name__ == "__main__":
    X, Y = load_data("generated_04.csv")
    X = standardize_data(X)

    learning_rate = 0.01
    epochs = 100

    X_train, Y_train, X_test, Y_test = train_test_split(X, Y, test_size=0.2)

    weights, bias, cost_history = multivariate_linear_regression(X_train, Y_train, learning_rate, epochs)

    # Calculate R2 score on test data
    Y_pred_test = np.dot(X_test, weights) + bias
    r2_test = calculate_r2_score(Y_test, Y_pred_test)

    print(f"Multivariate Linear Regression: Y = {bias} + {weights} * X")
    print(f"Mean Squared Error: {calculate_mse(X_test, Y_test, weights, bias)}")
    print(f"R-squared (R2) Score on Test Data: {r2_test}")

    Multivariate Linear Regression: Y = 0.2074090408501061 + [ 0.04900296  0.096607 -0.02208844  0.03011795  0.02068157  0.06125355
 0.01837764  0.00949503] * X
    Mean Squared Error: 0.19258223799327692
    R-squared (R2) Score on Test Data: 0.14179038327416715

weights

array([ 0.04900296,  0.096607, -0.02208844,  0.03011795,  0.02068157,
        0.06125355,  0.01837764,  0.00949503])

X_train

```

```

array([-1.96726979e-03, -3.10493014e-01, -1.85162283e+00, 2.34919224e-15,
        0.00000000e+00, -2.25102018e+00, -5.99847323e-01, -1.43956144e-01]),
array([0.98166763, 0.19442197, 0.62917173, 1.4934188, 0.37722923,
        1.65314838, 0.20014208, 0.65028465]),
array([2.62105912, 0.75873872, -0.58483412, -1.31684623, -1.02927253,
        1.1683338, -1.44791225, 1.54380554]),
array([-0.65772387, -0.90451064, 0.15412596, 0.4189057, -1.53452073,
        0.7790112, 0.58607259, -0.54107654]),
array([0.32591103, -1.35002386, -0.05700549, 1.24545424, -0.81078682,
        0.14149907, -1.40240404, -0.24323624]),
array([-0.00196727, -0.25109125, 1.63204612, -0.49029769, 1.08730779,
        -0.94693806, -1.44791225, -0.04467604]),
array([-0.00196727, 0.40232814, 0.84030318, 1.74138336, 1.64717743,
        -0.22954815, -1.2014918, -1.33531733]),
array([-0.65772387, 0.66963607, -0.4792684, 0.99748968, 2.88981491,
        1.65821615, 0.19587675, -1.33531733]),
array([-1.31348046, -0.4886983, 1.5264804, 0.58421541, -0.42843682,
        0.18491954, -0.90637174, -1.33531733]),
array([0.98166763, -0.36989478, -1.37657707, -0.40764284, 1.08730779,
        -0.69587164, 0.77741227, -0.24323624]),
array([-0.32984557, 0.58053343, -0.26813694, -0.49029769, -0.8790636,
        0.55033908, 0.72543963, -0.24323624]),
array([1.30954592, -1.94404149, -0.42648553, -0.90357196, -1.42527788,
        -0.68931269, -0.79095036, 0.35244435]),
array([-0.32984557, 0.87754224, -0.05700549, -0.07702342, -1.49355466,
        0.36537592, -0.68060035, 1.24596524]),
array([-1.96726979e-03, -1.02331417e+00, -6.90399847e-01, 2.34919224e-15,
        -3.19193968e-01, -1.04355171e+00, -5.23070284e-01, 1.14668515e+00]),
array([0.98166763, -0.60750183, -1.53492565, -1.23419138, -1.68472966,
        -0.67673564, -1.44791225, 0.74956475]),
array([0.65378933, -0.60750183, -1.53492565, -1.39950108, -0.90637431,
        -0.99358956, 0.09796137, -0.04467604]),
array([0.32591103, -0.99361329, -0.32091981, 0.17094114, 0.,
        0.07234313, -0.79430785, -1.33531733]),
array([-1.31348046, -0.07288596, -0.26813694, -0.40764284, -0.89271896,
        0.90195324, 0.76024349, -1.33531733]),
array([-0.00196727, -0.63720271, 0.31247455, -0.40764284, -1.56183145,
        -0.02118, -1.44791225, -1.33531733]),
array([-1.31348046, -1.08271593, -0.00422263, 0.91483482, -1.56183145,
        -0.05351102, -0.48073542, -1.03747704]),
array([-0.65772387, 1.23395282, -0.32091981, 0.08828628, 0.9917203,
        0.49572263, 1.88533533, -0.24323624]),
array([-0.00196727, 0.58053343, -0.69039985, 0.50156055, 1.07365244,
        -0.26963285, 0.90954039, -0.34251634]),
array([0.65378933, -0.07288596, -0.26813694, -0.24233313, 1.14192922,
        -0.20967411, -1.06181648, -0.54107654]),
array([1.96530252, -0.75600623, 0.9458689, -0.32498799, 0.04950067,
        0.9332536, -1.44791225, 1.04740505]),
array([-1.31348046, 0.52113166, -0.32091981, -1.56481079, -0.3738154,
        0.07112908, -1.12213978, -0.34251634]),
array([-1.31348046, -0.57780095, 0.84030318, 1.82403821, 2.2889792,
        0.78869534, -0.3004868, -1.33531733]),
array([-0.00196727, 0.10531933, 0.04856023, 0.33625084, -1.39796716,
        0.13429541, 0.39667388, 1.54380554]),
array([-1.96726979e-03, 9.36944004e-01, 5.23606000e-01, 2.07200277e+00,
        4.86472090e-01, 5.30122261e-01, 1.03424099e+00, 1.04740505e+00]),
array([-0.98560217, -0.3401939, -0.84874844, -0.15967828, -1.24775824,
        0.9384583, 0.5392205, -1.33531733])

```

▼ Batch Gradient Descent

We have first done feature scaling, by subtracting mean then dividing it by (max-min) of each column/feature, later proceeded with batch gradient descent

```
import pandas as pd
import numpy as np
```

```
df = pd.read_csv('/content/generated_04.csv')
df.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	7	149	96	19.0	10.000	38.387409	0.561331	40	1
1	0	151	44	6.0	105.000	26.125923	0.463959	27	1
2	3	169	57	24.0	240.000	33.224573	0.541364	36	0
3	4	86	61	35.0	129.375	32.918264	0.526311	39	0
4	6	75	62	31.0	77.000	37.453830	0.178734	21	0

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 9 columns):
 #   Column                Non-Null Count  Dtype
---  -
 0   Pregnancies           500 non-null    int64
 1   Glucose               500 non-null    int64
 2   BloodPressure         500 non-null    int64
 3   SkinThickness         500 non-null    float64
 4   Insulin               500 non-null    float64
 5   BMI                   500 non-null    float64
 6   DiabetesPedigreeFunction 500 non-null    float64
 7   Age                   500 non-null    int64
 8   Outcome               500 non-null    int64
dtypes: float64(4), int64(5)
memory usage: 35.3 KB
```

```
# Select ratio
ratio = 0.75
```

```
total_rows = df.shape[0]
train_size = int(total_rows*ratio)
```

```
# Split data into test and train
df_train = df[0:train_size]
df_test = df[train_size:]
```

```
# ndf_x=df.iloc[:,8].to_numpy()
ndf_x=df_train.iloc[:,8].to_numpy()
```

```
ndf_y=df_train.iloc[:,8]].to_numpy()
```

```
ndf_x
```

```
array([[ 7.         , 149.         , 96.         , ..., 38.38740931,
        0.5613309 , 40.         ],
       [ 0.         , 151.         , 44.         , ..., 26.1259232 ,
        0.4639594 , 27.         ],
       [ 3.         , 169.         , 57.         , ..., 33.22457298,
        0.54136377, 36.         ],
       ...,
       [ 3.         , 116.         , 92.         , ..., 43.25967386,
        0.47676083, 33.         ],
       [ 2.         , 108.         , 91.         , ..., 34.74993884,
        0.41291055, 21.         ],
       [ 2.         , 125.         , 54.         , ..., 33.15205425,
        0.22688252, 25.         ]])
```

```
#do scaling of features
ndf_avg = np.mean(ndf_x, axis = 0)
ndf_avg
```

```
array([ 4.072         , 121.22933333, 71.43733333, 23.74654945,
       129.63166667, 32.67226174,  0.53174403, 34.43466667])
```

```
ndf_max = np.max(ndf_x, axis = 0)
ndf_max
```

```
array([ 17.         , 199.         , 122.         , 55.         ,
       378.         , 59.93741155,  1.41387391, 73.         ])
```

```
ndf_min = np.min(ndf_x, axis = 0)
ndf_min
```

```
array([ 0.         , 14.         , 22.         , 1.         , 1.         ,
       10.9033396,  0.078         , 21.         ])
```

```
ndf_mx_mn= ndf_max-ndf_min
ndf_mx_mn
```

```
array([ 17.         , 185.         , 100.         , 54.         ,
       377.         , 49.03407195,  1.33587391, 52.         ])
```

```
mxmn_szd=np.tile(ndf_mx_mn, (df_train.shape[0], 1))
mxmn_szd
```

```
array([[ 17.         , 185.         , 100.         , ..., 49.03407195,
        1.33587391, 52.         ],
       ...,
       [ 17.         , 185.         , 100.         , ..., 49.03407195,
        1.33587391, 52.         ]])
```

```
[ 17.      , 185.      , 100.      , ..., 49.03407195,
  1.33587391, 52.      ],
[ 17.      , 185.      , 100.      , ..., 49.03407195,
  1.33587391, 52.      ],
...,
[ 17.      , 185.      , 100.      , ..., 49.03407195,
  1.33587391, 52.      ],
[ 17.      , 185.      , 100.      , ..., 49.03407195,
  1.33587391, 52.      ],
[ 17.      , 185.      , 100.      , ..., 49.03407195,
  1.33587391, 52.      ]])
```

```
avg_szd=np.tile(ndf_avg, (df_train.shape[0], 1))
avg_szd
```

```
array([[ 4.072      , 121.22933333, 71.43733333, ..., 32.67226174,
        0.53174403, 34.43466667],
 [ 4.072      , 121.22933333, 71.43733333, ..., 32.67226174,
        0.53174403, 34.43466667],
 [ 4.072      , 121.22933333, 71.43733333, ..., 32.67226174,
        0.53174403, 34.43466667],
 ...,
 [ 4.072      , 121.22933333, 71.43733333, ..., 32.67226174,
        0.53174403, 34.43466667],
 [ 4.072      , 121.22933333, 71.43733333, ..., 32.67226174,
        0.53174403, 34.43466667],
 [ 4.072      , 121.22933333, 71.43733333, ..., 32.67226174,
        0.53174403, 34.43466667]])
```

```
ndfx_scld=(ndf_x-avg_szd)/(mxmn_szd)
ndfx_scld
```

```
array([[ 0.17223529,  0.15011171,  0.24562667, ...,  0.11655462,
        0.02214795,  0.10702564],
 [-0.23952941,  0.16092252, -0.27437333, ..., -0.13350591,
        -0.05074179, -0.14297436],
 [-0.06305882,  0.25821982, -0.14437333, ...,  0.01126383,
        0.00720108,  0.03010256],
 ...,
 [-0.06305882, -0.02826667,  0.20562667, ...,  0.2159195 ,
        -0.04115897, -0.02758974],
 [-0.12188235, -0.07150991,  0.19562667, ...,  0.04237211,
        -0.08895561, -0.25835897],
 [-0.12188235,  0.02038198, -0.17437333, ...,  0.00978488,
        -0.22821129, -0.1814359 ]])
```

```
ndf_y
```

```
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[1],
[0],
[1],
[0],
[1],
[0],
[1]]])
```

```
df_train.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	7	149	96	19.0	10.000	38.387409	0.561331	40	1
1	0	151	44	6.0	105.000	26.125923	0.463959	27	1
2	3	169	57	24.0	240.000	33.224573	0.541364	36	0
3	4	86	61	35.0	129.375	32.918264	0.526311	39	0
4	6	75	62	31.0	77.000	37.453830	0.178734	21	0

```
n_samples = df_train.shape[0] # No. of samples taken
n_features = df.shape[1]-1 # No. of features / No. of values in x
n_targets = 1 # we have only 1 target outcome
```

```
import numpy as np
```

```
# Define your loss function
def loss_fn(y_hat, y):
    return np.mean((y_hat - y) ** 2)
```

```
# Define your hyperparameters
learning_rate = 0.001
num_epochs = 1000
```

```
# Initialize W and b based on your actual dimensions
W = np.random.normal(size=(n_targets, n_features))
b = np.random.normal(size=(n_targets, 1))
```

```
loss_history = []
```

```
for epoch in range(num_epochs):
    loss_a = np.zeros((n_targets, n_features), dtype=np.float64)
    loss_b = np.zeros((n_targets, 1), dtype=np.float64)
    loss = 0
```

```
    for i in range(n_samples):
        X_i, y_i = ndfx_scld[i], ndf_y[i]
        y_hat = np.dot(W, X_i) + b

        dL_dy = 2 * (y_hat - y_i)
        dy_dW = X_i
        dy_db = 1
```

```
        dL_dW = dL_dy * dy_dW
        loss_a += dL_dW
```

```
        dL_db = dL_dy * dy_db
        loss_b += dL_db
        loss += loss_fn(y_hat, y_i)
```

```
W -= learning_rate * loss_a
b -= learning_rate * loss_b
```

```
loss_history.append(loss)
print("Epoch: {}, Loss: {}".format(epoch + 1, loss))
```

```
Epoch: 946, Loss: 66.69997342510634
Epoch: 947, Loss: 66.69997342508371
Epoch: 948, Loss: 66.69997342506163
Epoch: 949, Loss: 66.69997342503996
Epoch: 950, Loss: 66.69997342501884
Epoch: 951, Loss: 66.69997342499815
Epoch: 952, Loss: 66.69997342497794
Epoch: 953, Loss: 66.69997342495815
Epoch: 954, Loss: 66.69997342493883
Epoch: 955, Loss: 66.69997342491985
Epoch: 956, Loss: 66.69997342490137
Epoch: 957, Loss: 66.69997342488331
Epoch: 958, Loss: 66.69997342486558
Epoch: 959, Loss: 66.69997342484837
Epoch: 960, Loss: 66.69997342483141
Epoch: 961, Loss: 66.69997342481493
Epoch: 962, Loss: 66.69997342479877
Epoch: 963, Loss: 66.69997342478293
Epoch: 964, Loss: 66.69997342476746
Epoch: 965, Loss: 66.69997342475233
Epoch: 966, Loss: 66.69997342473755
Epoch: 967, Loss: 66.69997342472313
Epoch: 968, Loss: 66.69997342470901
Epoch: 969, Loss: 66.69997342469517
Epoch: 970, Loss: 66.6999734246817
Epoch: 971, Loss: 66.69997342466841
Epoch: 972, Loss: 66.69997342465548
Epoch: 973, Loss: 66.69997342464289
Epoch: 974, Loss: 66.69997342463051
Epoch: 975, Loss: 66.6999734246184
Epoch: 976, Loss: 66.69997342460663
Epoch: 977, Loss: 66.69997342459504
Epoch: 978, Loss: 66.69997342458376
Epoch: 979, Loss: 66.69997342457265
Epoch: 980, Loss: 66.69997342456193
Epoch: 981, Loss: 66.69997342455129
Epoch: 982, Loss: 66.6999734245409
Epoch: 983, Loss: 66.69997342453094
Epoch: 984, Loss: 66.69997342452103
Epoch: 985, Loss: 66.6999734245114
Epoch: 986, Loss: 66.69997342450193
Epoch: 987, Loss: 66.6999734244927
Epoch: 988, Loss: 66.69997342448362
Epoch: 989, Loss: 66.69997342447482
Epoch: 990, Loss: 66.69997342446614
Epoch: 991, Loss: 66.69997342445774
Epoch: 992, Loss: 66.69997342444942
Epoch: 993, Loss: 66.69997342444135
Epoch: 994, Loss: 66.69997342443352
Epoch: 995, Loss: 66.69997342442575
Epoch: 996, Loss: 66.69997342441816
Epoch: 997, Loss: 66.69997342441081
Epoch: 998, Loss: 66.69997342440361
Epoch: 999, Loss: 66.69997342439653
Epoch: 1000, Loss: 66.69997342438961
```

W

```
array([[ 0.38183121,  1.03240145, -0.2172194 ,  0.31115492, -0.12662233,
         0.48786513,  0.01151499, -0.03261999]])
```

b

```
array([[0.36]])
```

▼ Insights drawn (plots, markdown explanations)

From the graphs below, we understand that linear regression is not a good model for classification task as although the model is doing okish with trained data the model fails to do well with the test data, this is observed in both batch and stochastic gradient decent. And from the cost functions we understand that gradient descent takes lesser number of iterations than stochastic but then it consumes more time for computation of total gradient of all samples at every iterations.

```
from matplotlib import pyplot as plt
```

```
#scaling test data to jsut in case but observed no change after scaling which was expected
tndf_x=df_test.iloc[:,8].to_numpy()
```

```
tndf_y=df_test.iloc[:,8]].to_numpy()
```

```
#do scaling of features
tndf_avg = np.mean(ndf_x, axis = 0)
```



```

tndf_max = np.max(ndf_x, axis = 0)

tndf_min = np.min(ndf_x, axis = 0)

tndf_mx_mn= tndf_max-tndf_min

tmxmn_szd=np.tile(tndf_mx_mn, (df_test.shape[0], 1))

tavg_szd=np.tile(tndf_avg, (df_test.shape[0], 1))

tndfx_scl=(tndf_x-tavg_szd)/(tmxmn_szd)

# Customize the font size globally for various elements
plt.rc('font', size=8)          # Set the font size for labels, titles, and tick labels
plt.rc('axes', titlesize=7)     # Set the font size for plot titles

```

We see that when a scatter plot of predicted results($y_{\text{pred}}/Y_{\text{pred_test}}$) and actual results(ndf_y/Y_{test}) is plotted then batch gradient seems to perform better for this data set as it has less scattered away points all across $x=0$ and $x=1$ whereas stochastic has more scattering away from 0 and 1 meaning it has less accurate predictions.

```

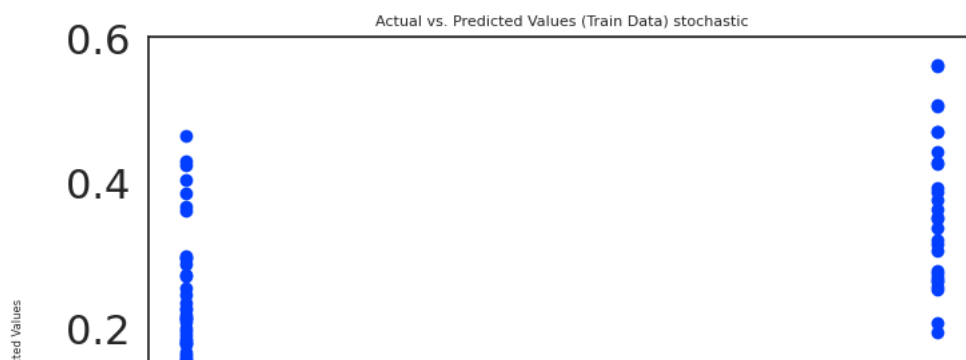
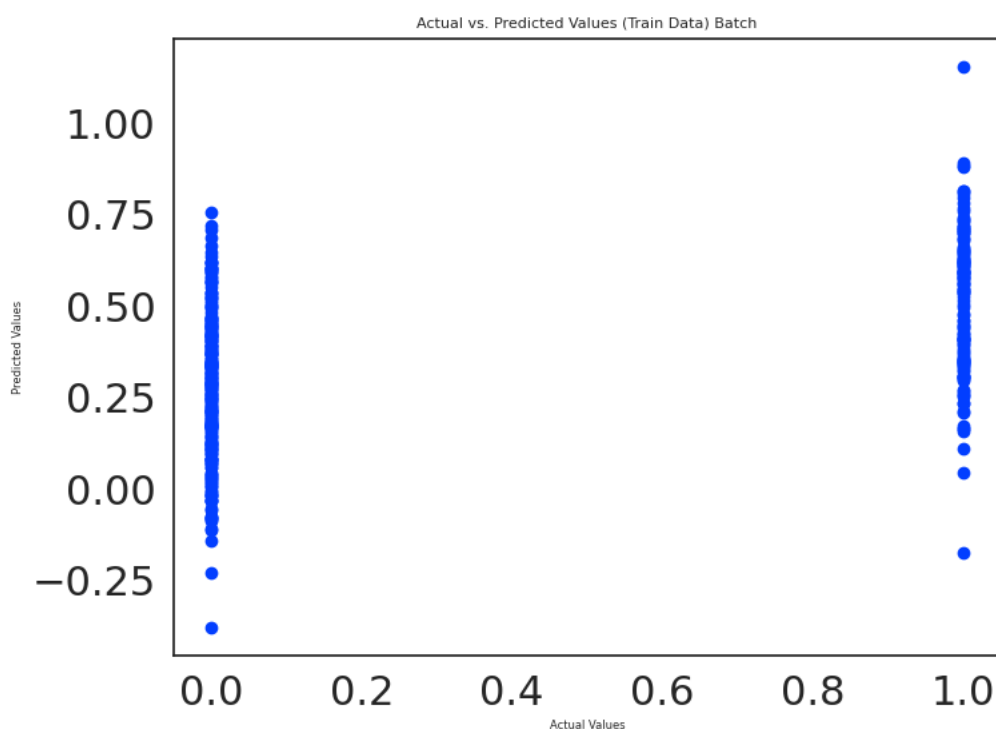
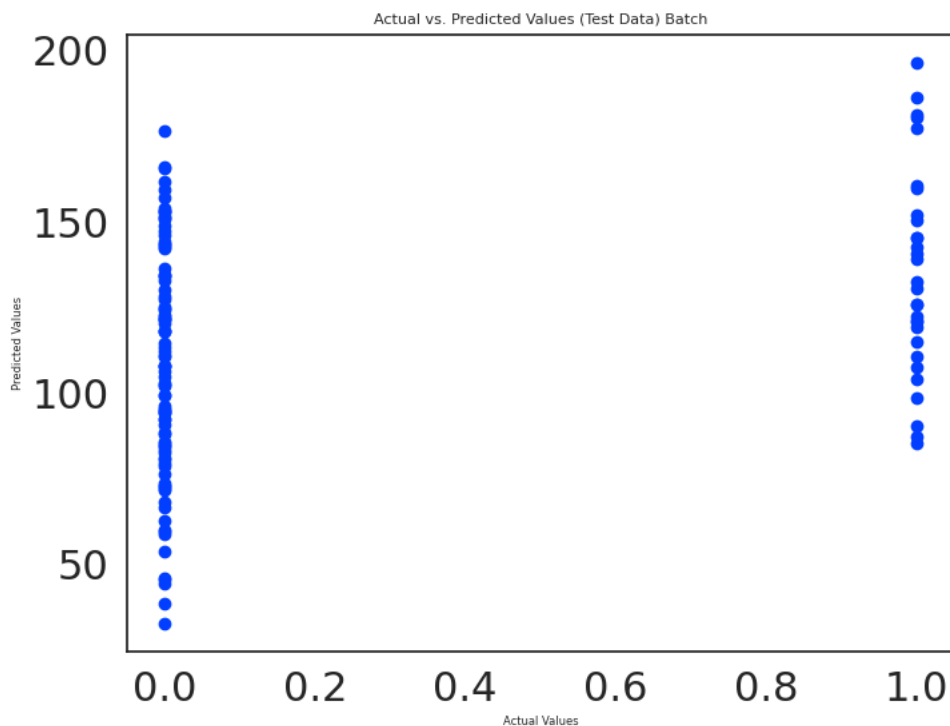
# # # Plot the regression line
# plt.scatter(ndfx_scl, ndf_y.T, label='Data')
# plt.plot(X, y_hat_arr, color='red', label='Regression Line')
# plt.xlabel('X')
# plt.ylabel('Y')
# plt.legend()
# plt.show()

# Assuming you have the training data 'X_train' and the true target values 'y_train'
plt.figure(figsize=(6, 4))
y_pred = np.dot(tndf_x, W[0].T) + b[0][0]
plt.scatter(tndf_y, y_pred)
plt.xlabel('Actual Values', fontsize=4)
plt.ylabel('Predicted Values', fontsize=4)
plt.title('Actual vs. Predicted Values (Test Data) Batch', fontsize=8)
plt.show()

# Assuming you have the training data 'X_train' and the true target values 'y_train'
plt.figure(figsize=(6, 4))
y_pred = np.dot(ndfx_scl, W[0].T) + b[0][0]
plt.scatter(ndf_y, y_pred)
plt.xlabel('Actual Values', fontsize=4)
plt.ylabel('Predicted Values', fontsize=4)
plt.title('Actual vs. Predicted Values (Train Data) Batch', fontsize=8)
plt.show()

#for stochastic
plt.figure(figsize=(8, 6))
plt.scatter(Y_test, Y_pred_test)
plt.xlabel('Actual Values', fontsize=4)
plt.ylabel('Predicted Values', fontsize=4)
plt.title('Actual vs. Predicted Values (Train Data) stochastic', fontsize=8)
plt.show()

```

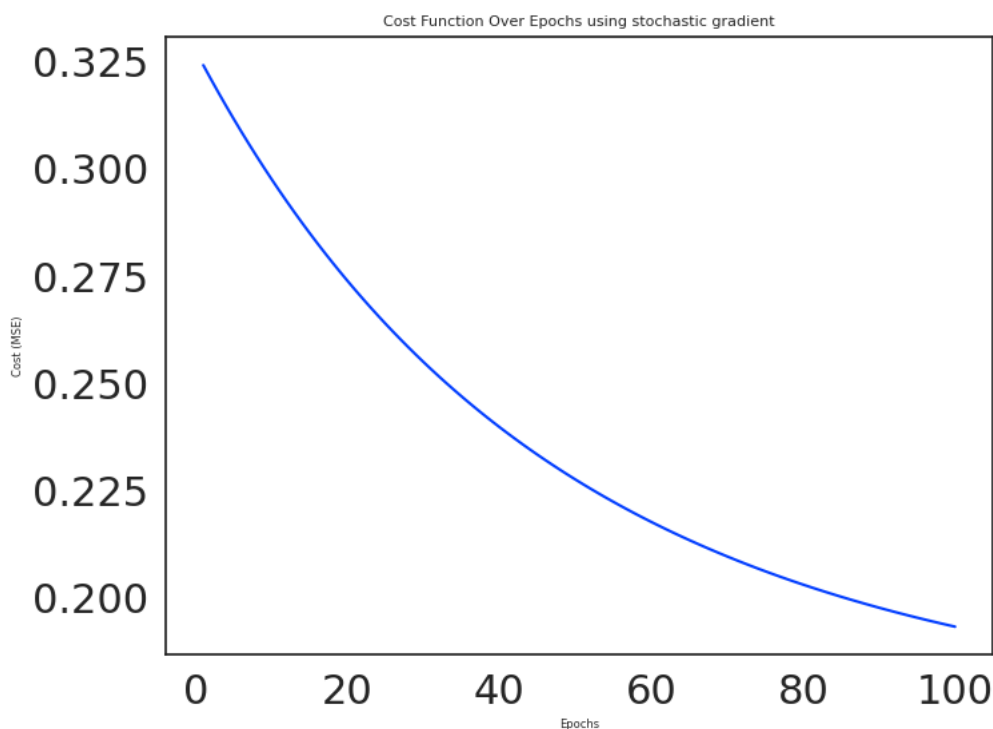
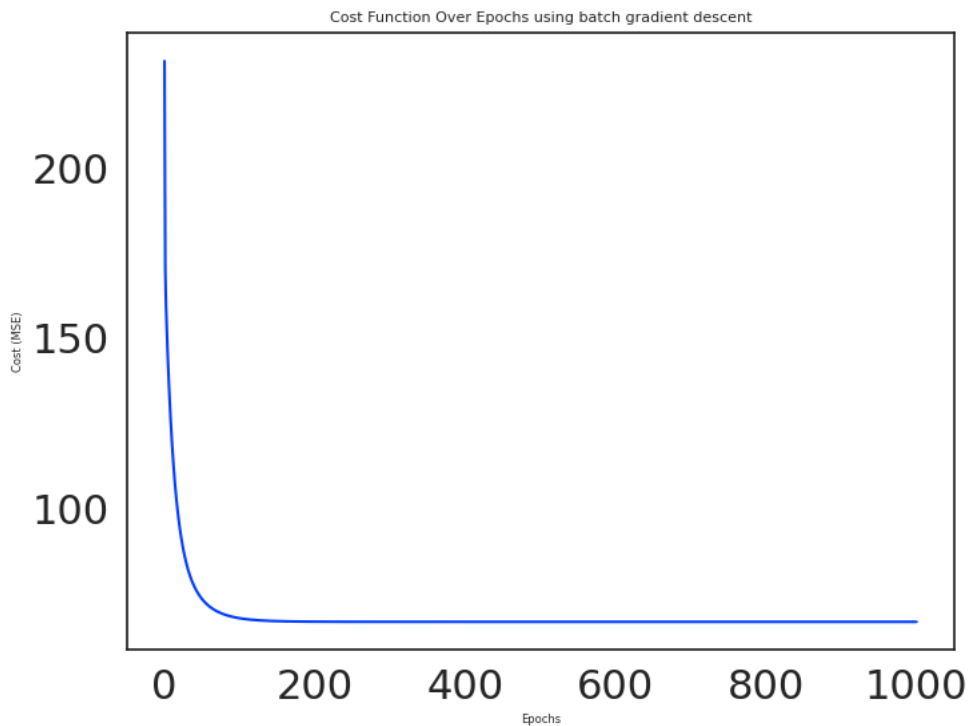


We see in the cost vs epochs plot that bgd has sudden drop in cost in the starting iterations and then stabilizes for rest of the iterations meaning that cost decreases drastically in the first iterations and then slowly start to decrease less whereas in sgd cost decreases in a smooth constant curve and no sudden decrease in seen no matter what iteration at what point is happening.

```
# Plot the cost function over epochs
plt.figure(figsize=(6, 4))
plt.plot(range(1, num_epochs + 1), loss_history)
```

```
plt.xlabel('Epochs', fontsize=4)
plt.ylabel('Cost (MSE)', fontsize=4)
plt.title('Cost Function Over Epochs using batch gradient descent', fontsize=6)
plt.show()
```

```
# Plot the cost function over epochs
plt.figure(figsize=(6, 4))
plt.plot(range(1, epochs + 1), cost_history)
plt.xlabel('Epochs', fontsize=4)
plt.ylabel('Cost (MSE)', fontsize=4)
plt.title('Cost Function Over Epochs using stochastic gradient', fontsize=6)
plt.show()
```



▼ 4. Comparison of Lasso and Ridge Regression using Polynomial Regression

Lasso Regression

```

class LassoRegression:
    """
    This class implements the Lasso Regression model from scratch.
    Lasso regression is a type of linear regression that uses shrinkage.
    Shrinkage is where data values are shrunk towards a central point, like the mean.
    A tuning parameter, lambda controls the strength of the L1 penalty.
    """

    def __init__(self, learning_rate= 0.01, n_iters= 10, lambda_l= 0.5):
        self.lr = learning_rate
        self.iterations = n_iters
        self.lambda_l = lambda_l
        self.theta = None
        self.bias = None
        self.loss = []

    def _linear_model(self, X):
        """
        Compute the linear model for the given feature matrix.

        Args:
            X (numpy.ndarray): the feature matrix to compute.

        Returns:
            numpy.ndarray: the computed linear model.
        """
        return np.dot(X, self.theta) + self.bias

    def _initialize_parameters(self, n_features):
        """
        Initialize the weights and bias with random values.

        Args:
            n_features (int): the number of features in the dataset.
        """
        self.theta = np.random.random_sample(n_features)
        self.bias = np.random.random_sample()

    def _compute_theta_derivative(self, X, y, linear_model):
        """
        Compute the derivative of theta for the given feature matrix, target vector, and linear model.

        Args:
            X (numpy.ndarray): the feature matrix.
            y (numpy.ndarray): the target vector.
            linear_model (numpy.ndarray): the computed linear model.

        Returns:
            numpy.ndarray: the computed derivative of theta.
        """
        n_samples, n_features = X.shape
        d_theta = np.zeros(n_features)

        for j in range(n_features):
            if self.theta[j] > 0:
                #This condition checks whether the current value of the feature coefficient (theta) is positive.
                d_theta[j] = -(2/n_samples) * (np.dot(X[:, j], (y - linear_model)) + self.lambda_l)
            else:
                d_theta[j] = -(2/n_samples) * (np.dot(X[:, j], (y - linear_model)) - self.lambda_l)

        return d_theta

    def _compute_bias_derivative(self, X, y, linear_model):
        """
        Compute the derivative of the bias for the given feature matrix, target vector, and linear model.

        Args:
            X (numpy.ndarray): The feature matrix.
            y (numpy.ndarray): The target vector.
            linear_model (numpy.ndarray): The predicted target values (linear model output).

        Returns:
            float: The derivative of the bias.
        """
        d_bias = -(2/X.shape[0]) * np.sum(y - linear_model)
        return d_bias

    def _validate_inputs(self, X, y):
        """
        Validate the dimensions of the feature matrix and the target vector.

```

```

    Args:
        X (numpy.ndarray): The feature matrix.
        y (numpy.ndarray): The target vector.

    Raises:
        AssertionError: If the number of samples in the feature matrix and the target vector is not equal.
    """
    assert X.shape[0] == y.shape[0], 'The number of samples in the feature matrix and the target vector should be equal.'

def _calculate_cost(self, y, z):
    """
    Calculate the Lasso cost (error) for the given target, prediction, and lambda.

    Args:
        y (numpy.ndarray): The true target values.
        z (numpy.ndarray): The predicted target values.

    Returns:
        float: The calculated Lasso cost (root mean squared error + L1 penalty).
    """
    n_samples = y.shape[0]
    lasso_loss = (1/n_samples) * np.sum(np.square(y - z)) + (self.lambda_1 * np.sum(np.abs(self.theta)))
    return np.sqrt(lasso_loss)

def fit(self, X, y):
    self._validate_inputs(X, y)
    self._initialize_parameters(X.shape[1])

    for _ in range(self.iterations):
        linear_model = self._linear_model(X)
        d_theta = self._compute_theta_derivative(X, y, linear_model)
        d_bias = self._compute_bias_derivative(X, y, linear_model)
        self.theta -= self.lr * d_theta
        self.bias -= self.lr * d_bias
        self.loss.append(self._calculate_cost(y, linear_model))

def predict(self, X):
    """
    Predict the target value for the given feature matrix.

    Args:
        X (numpy.ndarray): the feature matrix to predict.

    Returns:
        numpy.ndarray: the predicted target values.
    """
    return self._linear_model(X)

def cost(self):
    """
    Return the computed loss for the model.

    Returns:
        list: the list of loss values for each iteration.
    """
    return self.loss

import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def my_plot(cost):
    iterations = [i for i in range(1, 11)]
    fig, ax = plt.subplots()
    ax.plot(iterations, cost, color='red')
    ax.set_xlabel('Iterations')
    ax.set_ylabel('Cost')
    ax.set_title('Cost vs Iterations')

def surface_plot(X, y, prediction, degree):
    fig = plt.figure(figsize=[30, 50])
    ax = fig.add_subplot(5, 2, degree, projection='3d')

    ax.scatter(X[:,0], X[:,1], y, zdir='z', s=20, c='#9467bd', depthshade=True)

    Y_plot = np.asarray(prediction) #converting the prediction data to a numpy array.
    X = np.asarray(X)

```

```
ax.plot_trisurf(X[:, 0], X[:, 1], Y_plot, cmap = 'viridis')
ax.set_xlabel('$Age$', fontsize=10, rotation=150)
ax.set_ylabel('$BMI$', fontsize=10, rotation=150)
ax.set_zlabel('$Insurance$', fontsize=10, rotation=60)
ax.set_title('Degree: ' + str(degree), fontsize=20)
plt.show()
```

```
from sklearn.preprocessing import PolynomialFeatures
```

```
# This class is used to generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or
```

```
poly = PolynomialFeatures(9, include_bias= False)
```

```
X = synthetic_data.iloc[:, :8]
```

```
X.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	
0	7	149	96	19.0	10.000	38.387409	0.561331	40	
1	0	151	44	6.0	105.000	26.125923	0.463959	27	
2	3	169	57	24.0	240.000	33.224573	0.541364	36	
3	4	86	61	35.0	129.375	32.918264	0.526311	39	
4	6	75	62	31.0	77.000	37.453830	0.178734	21	

```
X = poly.fit_transform(X)
```

```
""" standardizing features.
```

```
first subtracts the mean of each feature (column-wise, as specified by axis=0), and then divides the result by the standard deviation of
```

```
X = (X - X.mean(axis= 0)) / X.std(axis= 0)
```

```
y = synthetic_data.iloc[:, 8]
```

```
from sklearn.model_selection import train_test_split
```

```
X_train_k, X_test_k, y_train_k, y_test_k = train_test_split(X, y)
```

```
y_train_k = y_train_k.to_numpy()
```

```
y_test_k = y_test_k.to_numpy()
```

```
lasso = LassoRegression(learning_rate= 0.01, n_iters= 10, lambda_l= 0.5)
```

```
lasso.fit(X_train_k, y_train_k)
```

```
y_predicted_l_k = lasso.predict(X_train_k)
```

```
cost_lasso = lasso.cost()
```

▼ Ridge Regression

```
import pandas as pd
```

```
import numpy as np
```

```
from sklearn.model_selection import train_test_split
```

```
teams = pd.read_csv("/content/generated_04.csv")
```

```
def ridge_fit(X,y,alpha):
```

```
    x_mean=X.mean()
```

```
    x_std=X.std()
```

```
    X=(X-x_mean)/x_std
```

```
    penalty = alpha * np.identity(X.shape[1])
```

```
    penalty[0][0]=0
```

```
    A=np.dot(X.T, y)
```

```
    B=np.linalg.inv((X.T @ X) + penalty)@A
```

```
    return B
```

```
def ridge_predict(test_X,B):
```

```
    x_mean=test_X.mean()
```

```
    x_std=test_X.std()
```

```
    test_X=(test_X- x_mean)/x_std
```

```

predictions=np.dot(test_X,B)
return predictions

def generate_polynomial_features(X, degree):
    if degree < 1:
        raise ValueError("Degree must be at least 1.")

    n_samples, n_features = X.shape
    X_poly = np.ones((n_samples, 1)) # Initialize with a column of ones (bias term)

    for d in range(1, degree + 1):
        for feature in range(n_features):
            new_feature = X[:, feature].reshape(-1, 1) ** d
            X_poly = np.hstack((X_poly, new_feature))

    return X_poly


predictors=["Pregnancies","Glucose","BloodPressure","SkinThickness","Insulin","BMI","DiabetesPedigreeFunction","Age"]
target="Outcome"

train, test = train_test_split(teams, test_size=0.2, random_state=1)
X_train = train[predictors].copy()
y_train = train[[target]].copy()
X_test = test[predictors].copy()
y_test = test[[target]].copy()

X_tra=generate_polynomial_features(X_train.to_numpy(),2)
X_tes=generate_polynomial_features(X_test.to_numpy(),2)

```

y_train

	Outcome	
238	1	
438	0	
475	0	
58	0	
380	0	
...	...	
255	0	
72	0	
396	0	
235	1	
37	0	

400 rows × 1 columns

```

B=ridge_fit(X_tra,y_train,0.005)
y_pred=ridge_predict(X_tes,B)

```

```

print(B)
print(B.shape)

```

```

[[-1.45758212e+01]
 [ 1.00505431e+00]
 [ 1.02359485e+00]
 [-5.82841134e-01]
 [ 6.06971446e-01]
 [ 1.57242083e+00]
 [ 1.39358790e-02]
 [ 3.86241849e-02]
 [ 1.96524188e-01]
 [ 9.28930809e+00]
 [ 1.90402581e-01]
 [-1.77181220e-01]
 [ 5.73240227e-01]
 [-6.62263258e-03]
 [ 1.41493569e+00]
 [ 3.17093465e-02]
 [-1.22486676e-01]]
(17, 1)

```

y_pred

```

array([[ 0.02845395],
       [ 0.90118155],
       [ 0.46818708],
       [ 0.15126193],
       [ 0.51958735],
       [ 0.20622684],
       [ 0.58843275],
       [ 0.0900462 ],
       [ 0.27028647],
       [ 0.70903371],
       [ 0.32061336],
       [ 0.04624382],
       [ 0.59392052],
       [ 0.2066136 ],
       [ 0.39918854],
       [ 0.45552006],
       [ 0.10862233],
       [ 0.46060109],
       [ 0.70477254],
       [ 0.42642466],
       [ 0.11314603],
       [ 0.65080885],
       [ 0.32629914],
       [ 0.28992583],
       [ 0.03188381],
       [ 0.41223067],
       [ 0.26265076],
       [ 0.15408431],
       [ 0.39101906],
       [ 0.52934473],
       [ 0.0589499 ],
       [ 0.36287771],
       [ 0.26744656],
       [ 0.5303745 ],
       [ 0.4633887 ],
       [ 0.42209596],
       [ 0.34126243],
       [ 0.25410488],
       [ 0.21462291],
       [ 0.02875982],
       [ 0.04488191],
       [ 0.02789094],
       [ 0.61736097],
       [ 0.08741565],
       [ 1.03361187],
       [ 0.09397077],
       [ 0.41888118],
       [ 0.72083946],
       [ 0.13976138],
       [ 0.25863318],
       [-0.02760337],
       [ 0.28409995],
       [ 0.33779857],
       [ 0.54123648],
       [ 0.69419847],
       [-0.03928136],
       [ 0.39105625],
       [ 0.56563829],

```

```

from sklearn.metrics import mean_squared_error
best_alpha = None
best_mse = float('inf')
alphas=10*np.linspace(10,-2,100)*0.5
# Iterate over alpha values and train models

train, test = train_test_split(teams, test_size=0.2, random_state=1)
X_train = train[predictors].copy()
y_train = train[[target]].copy()
X_test = test[predictors].copy()
y_test = test[[target]].copy()
degree=2
X_tra=generate_polynomial_features(X_train.to_numpy(),degree)
X_tes=generate_polynomial_features(X_test.to_numpy(),degree)

mse_values=[]
for alpha in alphas:

    B=ridge_fit(X_tra,y_train,alpha)
    y_pred=ridge_predict(X_tes,B)
    mse = mean_squared_error(y_test, y_pred)
    mse_values.append(mse)
    if mse < best_mse:
        best_mse = mse

```

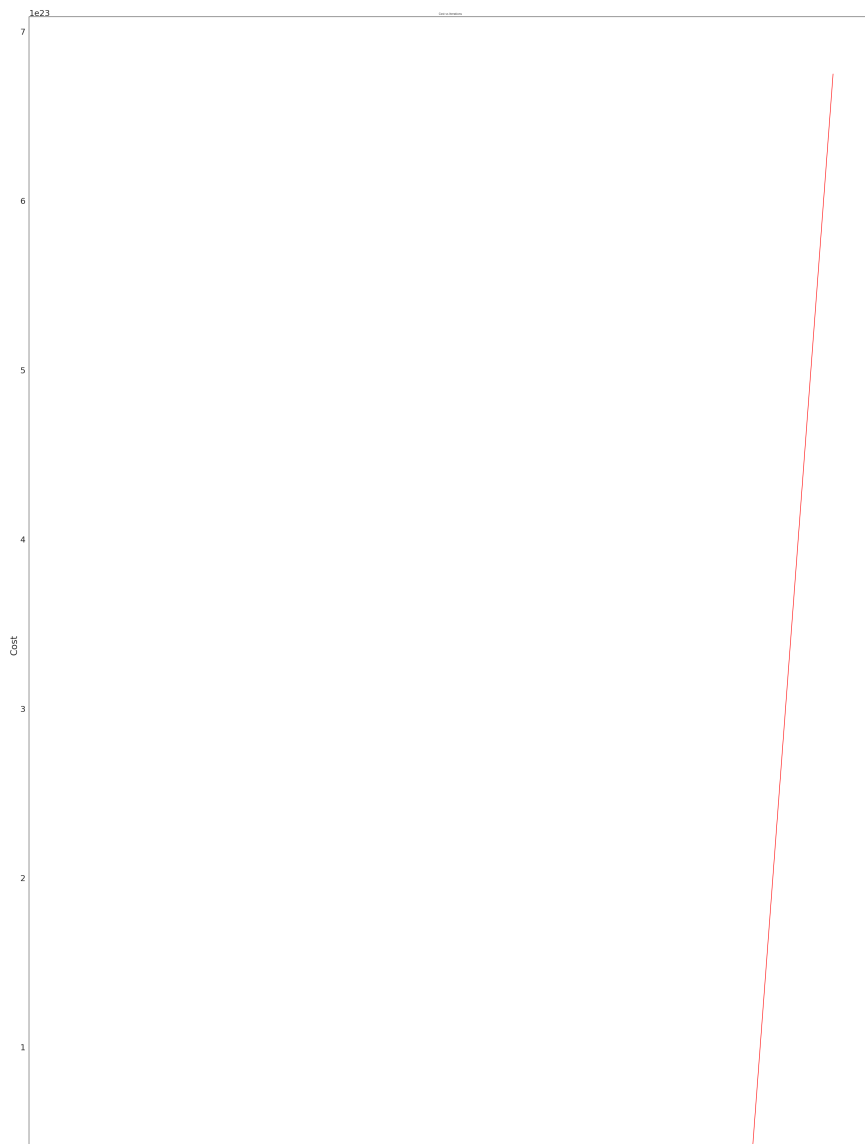


```
best_alpha = alpha  
  
print(best_alpha)  
  
0.005
```

▼ ***Insights drawn (plots, markdown explanations)***

Plots showing the relationship between all 8 features in Lasso.

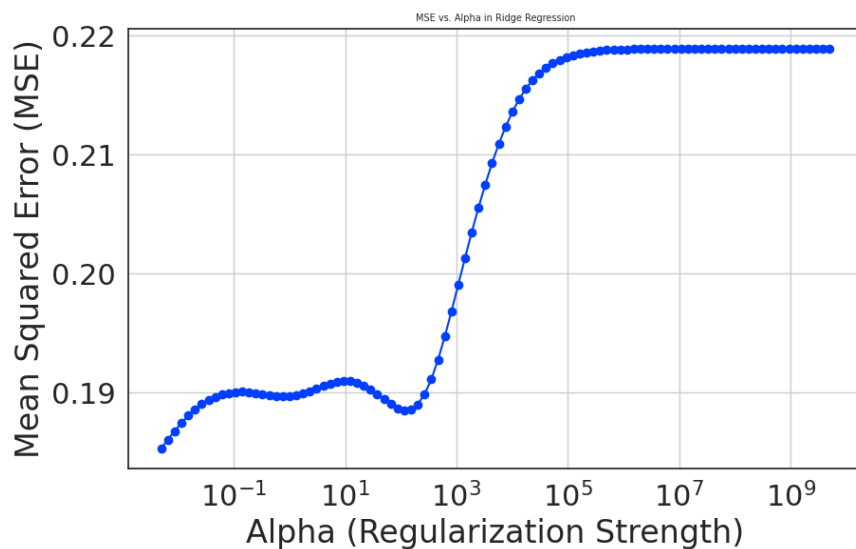
```
my_plot(cost_lasso)
```



```
surface_plot(X_train_k, y_train_k, y_predicted_l_k, 8)
```

Degree: 8

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.plot(alphas, mse_values, marker='o', linestyle='--')
plt.title('MSE vs. Alpha in Ridge Regression')
plt.xlabel('Alpha (Regularization Strength)')
plt.ylabel('Mean Squared Error (MSE)')
plt.xscale('log') # Use a logarithmic scale for alpha values
plt.grid(True)
plt.show()
```



5. Comparison of Logistic Regression and Least Squares Classification

Logistic Regression

```
import numpy as np

def sigmoid(x):
    return 1/(1+np.exp(-x))

class LogisticRegression():

    def __init__(self, lr=0.01, n_iters=10000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_features, n_samples = X.shape
        self.weights = np.zeros((n_features, 1))
        self.bias = 0

        for _ in range(self.n_iters):
            linear_pred = np.dot(self.weights.T, X) + self.bias
            predictions = sigmoid(linear_pred)
            dw = (1/n_samples) * np.dot(predictions - y, X.T)
            db = (1/n_samples) * np.sum(predictions - y)

            self.weights = self.weights - self.lr * dw.T
            self.bias = self.bias - self.lr * db
```

```

def predict(self, X):
    linear_pred = np.dot(self.weights.T,X) + self.bias
    y_pred = sigmoid(linear_pred)
#    class_pred = [0 if y<=0.5 else 1 for y in y_pred]
    y_scores = y_pred
    y_pred=y_pred>0.65
    y_pred = np.array(y_pred, dtype = 'int64')
    return y_pred

def predict_proba(self, X):
    linear_pred = np.dot(self.weights.T, X) + self.bias
    y_proba = sigmoid(linear_pred)
    return np.hstack((1 - y_proba, y_proba))

def feature_scaling(X):
    n_features = X.shape[0]
    for i in range(n_features):
        mean = np.mean(X[i, :])
        std_dev = np.std(X[i, :])
        X[i, :] = (X[i, :] - mean) / std_dev
    return X

import numpy as np
def split_train_test(data,test_ratio):
    np.random.seed(42)
    shuffled=np.random.permutation(len(data))
    test_set_size=int(len(data)*test_ratio)
    test_indices=shuffled[:test_set_size]
    train_indices=shuffled[test_set_size:]
    return data.iloc[train_indices],data.iloc[test_indices]

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

data=pd.read_csv("/content/generated_04.csv")
# train_set,test_set=split_train_test(data,0.2)
# X_train=train_set.drop("Outcome",axis=1)
# y_train=train_set[["Outcome"]]
# X_test=test_set.drop("Outcome",axis=1)
# y_test=test_set[["Outcome"]]

X = data.iloc[:, :-1].values # Features
y = data.iloc[:, -1].values # Target
split_index = int(0.8 * X.shape[0])
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]

# X_train = X_train.values
# y_train = y_train.values
# X_test = X_test.values
# y_test = y_test.values
X_train = X_train.T
y_train = y_train.reshape(1, X_train.shape[1])
X_test = X_test.T
y_test = y_test.reshape(1, X_test.shape[1])

X_train = feature_scaling(X_train)
X_test = feature_scaling(X_test)

clf = LogisticRegression(lr=0.01)
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)

def accuracy(y_pred, y_test):
    return (1-np.sum(np.absolute(y_pred-y_test))/y_test.shape[1])*100

acc = accuracy(y_pred, y_test)
print(acc)
# Calculate the confusion matrix manually
def confusion_matrix(y_true, y_pred):
    TP = np.sum(np.logical_and(y_true == 1, y_pred == 1))
    TN = np.sum(np.logical_and(y_true == 0, y_pred == 0))
    FP = np.sum(np.logical_and(y_true == 0, y_pred == 1))
    FN = np.sum(np.logical_and(y_true == 1, y_pred == 0))

```

```

    return np.array([[TN, FP], [FN, TP]])

# Calculate the confusion matrix
conf_matrix = confusion_matrix(y_test[0], y_pred[0])

print("Confusion Matrix:")
print(conf_matrix)

# Extract TP, TN, FP, FN from the confusion matrix
TP = conf_matrix[1, 1]
TN = conf_matrix[0, 0]
FP = conf_matrix[0, 1]
FN = conf_matrix[1, 0]

# Calculate additional metrics like precision, recall, and F1-score
precision = TP / (TP + FP)
recall = TP / (TP + FN)
f1_score = 2 * (precision * recall) / (precision + recall)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1_score}")

80.0
Confusion Matrix:
[[69  5]
 [15 11]]
Precision: 0.6875
Recall: 0.4230769230769231
F1 Score: 0.5238095238095238

```

X_train

```

array([[ 0.96309135, -1.29352436, -0.32640334, ...,  0.318344 ,
         0.318344 , -0.97115069],
       [ 0.84082247,  0.89951863,  1.42778405, ..., -2.15268161,
        -0.94941036, -0.1863603 ],
       [ 1.30098825, -1.44321169, -0.75716171, ...,  0.24552674,
        -1.12657324, -0.70438863],
       ...,
       [ 0.73614458, -0.80493056,  0.0872576 , ...,  0.38100485,
        -1.24534747, -2.31543419],
       [ 0.13129803, -0.19382106,  0.06462866, ..., -0.51545823,
        -1.34649891, -0.33812062],
       [ 0.56817415, -0.71583522,  0.17309434, ..., -0.51829532,
        -0.32075542, -0.51829532]])

```

▼ Least Squares Classification

```

import numpy as np
import pandas as pd

df = pd.read_csv('/content/generated_04.csv')
X = df.iloc[:, :-1].values # Features
y = df.iloc[:, -1].values # Target
Y = np.array([[1, 0] if val == 0 else [0, 1] for val in y])
# Y_train = np.array([[1, 0] if val == 0 else [0, 1] for val in y_train])
# Y_test = np.array([[1, 0] if val == 0 else [0, 1] for val in y_test])

split_index = int(0.8 * X.shape[0])
X_train1, X_test1 = X[:split_index], X[split_index:]
Y_train1, Y_test1 = Y[:split_index], Y[split_index:]
y_train1, y_test1 = y[:split_index], y[split_index:]

def optimize_W(X, Y):
    # Calculate the weight vector using the least squares formula
    W = np.linalg.inv(X.T @ X) @ X.T @ Y
    return W

W = optimize_W(X_train1, Y_train1)

Y_pred1 = np.dot(X_test1, W)

Y_pred_new1 = np.where(Y_pred1[:, 0:1] > Y_pred1[:, 1:2], 1, 0)
Y_pred_new1 = np.array([[1, 0] if val == 0 else [0, 1] for val in Y_pred_new1])

from sklearn.metrics import accuracy_score

Y_pred_binary1 = np.argmax(Y_pred1, axis=1)

# Convert true labels to binary labels (1 or 0)

```

```

Y_true_binary1 = np.argmax(Y_test1, axis=1)

accuracy = accuracy_score(Y_true_binary1, Y_pred_binary1)
print(f"Accuracy: {accuracy * 100:.2f}%")

def confusion_matrix(y_true, y_pred):
    TP = np.sum(np.logical_and(y_true == 1, y_pred == 1))
    TN = np.sum(np.logical_and(y_true == 0, y_pred == 0))
    FP = np.sum(np.logical_and(y_true == 0, y_pred == 1))
    FN = np.sum(np.logical_and(y_true == 1, y_pred == 0))

    return np.array([[TN, FP], [FN, TP]])

conf_matrix = confusion_matrix(Y_true_binary1, Y_pred_binary1)

print("Confusion Matrix:")
print(conf_matrix)

# Extract TP, TN, FP, FN from the confusion matrix
TP = conf_matrix[1, 1]
TN = conf_matrix[0, 0]
FP = conf_matrix[0, 1]
FN = conf_matrix[1, 0]

# Calculate precision, recall, and F1-score
precision = TP / (TP + FP)
recall = TP / (TP + FN)
f1_score = 2 * (precision * recall) / (precision + recall)

print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1_score:.2f}")

```

```

Accuracy: 77.00%
Confusion Matrix:
[[64 10]
 [13 13]]
Precision: 0.57
Recall: 0.50
F1 Score: 0.53

```

▼ **Insights drawn (plots, markdown explanations)**

Let's analyze and draw insights from the results of both the Logistic Regression and Least Squares Classification models:

Logistic Regression Results:

- **Accuracy:** 80.0%
 - The Logistic Regression model achieves an accuracy of 80.0%, which means it correctly classifies 80.0% of the examples in the test dataset. This is a relatively good accuracy.
- **Confusion Matrix:**
 - True Negatives (TN): 69
 - False Positives (FP): 5
 - False Negatives (FN): 15
 - True Positives (TP): 11
 - The confusion matrix provides a breakdown of the model's predictions.
 - The model correctly predicted 69 negative cases (TN) and 11 positive cases (TP).
 - It incorrectly predicted 5 negative cases as positive (FP) and 15 positive cases as negative (FN).
- **Precision:** 0.6875
 - Precision measures the accuracy of positive predictions. In this case, it means that when the model predicts a positive outcome, it is correct approximately 68.75% of the time.
- **Recall (Sensitivity):** 0.4231
 - Recall, also known as sensitivity or True Positive Rate, is the fraction of actual positive cases that the model correctly identifies. The model captures approximately 42.31% of the actual positive cases.
- **F1 Score:** 0.5238
 - The F1 Score is the harmonic mean of precision and recall. It provides a balance between precision and recall. The F1 score of 0.5238 suggests that the model achieves a reasonable trade-off between precision and recall.

Least Squares Classification Results:

- **Accuracy: 77.0%**
 - The Least Squares Classification model achieves an accuracy of 77.0%, which is slightly lower than the Logistic Regression model but still reasonably good.
- **Confusion Matrix:**
 - True Negatives (TN): 64
 - False Positives (FP): 10
 - False Negatives (FN): 13
 - True Positives (TP): 13
 - Similar to the Logistic Regression model, the confusion matrix provides information about the model's predictions.
 - The model correctly predicted 64 negative cases (TN) and 13 positive cases (TP).
 - It incorrectly predicted 10 negative cases as positive (FP) and 13 positive cases as negative (FN).
- **Precision: 0.57**
 - The precision of 0.57 indicates that when the model predicts a positive outcome, it is correct approximately 57% of the time.
- **Recall (Sensitivity): 0.50**
 - Recall, or sensitivity, at 0.50 means that the model captures 50% of the actual positive cases.
- **F1 Score: 0.53**
 - The F1 Score of 0.53 suggests that the model also achieves a reasonable balance between precision and recall.

Insights:

- Both models perform reasonably well but have different trade-offs.
- Logistic Regression has a higher accuracy, precision, and F1 Score compared to Least Squares Classification.
- Least Squares Classification has a slightly higher recall compared to Logistic Regression, indicating that it captures more positive cases.
- Both models exhibit similar F1 Scores, indicating a similar balance between precision and recall. Logistic Regression has a slightly higher accuracy and precision, making it more suitable if minimizing false positives is crucial. Least Squares Classification has a slightly higher recall, suggesting that it may capture more true positives.

PLOTS

1. Precision-Recall Curve:

- This curve illustrates the trade-off between precision and recall for different classification thresholds.
- The x-axis represents recall, which is the fraction of true positives correctly identified by the model.
- The y-axis represents precision, which is the fraction of true positives among all positive predictions made by the model.
- The curve shows how changing the threshold affects both precision and recall. You can choose a threshold that balances precision and recall based on your specific application's requirements.
- A higher curve indicates better model performance, with higher precision and recall at the chosen threshold.

2. Confusion Matrix Heatmap:

- The confusion matrix heatmap provides a visual representation of the model's classification performance.
- It shows the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) in a grid.
- The diagonal elements (top-left to bottom-right) represent correct predictions, while off-diagonal elements represent errors.
- This heatmap helps you understand where the model is making mistakes, such as misclassifying negatives as positives (FP) or failing to identify positives (FN).

3. Histogram of Predicted Probabilities:

- The histogram displays the distribution of predicted probabilities for the positive class (Class 1).
- It shows how confident the model is in assigning the positive class label to different examples.
- Peaks on the right side of the histogram indicate high-confidence positive predictions, while peaks on the left side indicate high-confidence negative predictions.
- The spread and shape of the histogram can give insights into the model's certainty when making predictions.

These plots collectively provide a comprehensive view of your model's performance, its ability to balance precision and recall, the areas where it may be making errors, and the distribution of predicted probabilities. They help you make informed decisions about model threshold selection and assess the model's suitability for your specific application.

Least Square Classification

```
import numpy as np
import matplotlib.pyplot as plt
```

```

# Calculate precision-recall curve
def precision_recall_curve(y_true, y_scores):
    thresholds = np.linspace(0, 1, num=100)
    precision_values = []
    recall_values = []

    for threshold in thresholds:
        y_pred = (y_scores > threshold).astype(int)
        TP = np.sum(np.logical_and(y_true == 1, y_pred == 1))
        FP = np.sum(np.logical_and(y_true == 0, y_pred == 1))
        FN = np.sum(np.logical_and(y_true == 1, y_pred == 0))

        if TP + FP == 0:
            precision = 1 # Handle the case of zero precision gracefully
        else:
            precision = TP / (TP + FP)

        if TP + FN == 0:
            recall = 1 # Handle the case of zero recall gracefully
        else:
            recall = TP / (TP + FN)

        precision_values.append(precision)
        recall_values.append(recall)

    return precision_values, recall_values

y_scores = Y_pred1[:, 1] # Use the scores for class 1
precision_values, recall_values = precision_recall_curve(Y_true_binary1, y_scores)

# Plot the precision-recall curve
plt.figure(figsize=(6,4))
plt.plot(recall_values, precision_values, marker='o')
plt.title('Precision-Recall Curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.grid(True)
plt.show()

# Calculate the confusion matrix manually (as in your existing code)

# Plot confusion matrix heatmap
plt.figure(figsize=(6, 4))
plt.imshow(conf_matrix, cmap='Blues', interpolation='nearest')
plt.colorbar()
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.xticks([0, 1], ['Negative', 'Positive'])
plt.yticks([0, 1], ['Negative', 'Positive'])
plt.show()

# Plot histogram of predicted probabilities
plt.figure(figsize=(6,4))
plt.hist(y_scores, bins=20, alpha=0.5, color='blue', edgecolor='black')
plt.title('Histogram of Predicted Probabilities')
plt.xlabel('Predicted Probability for Class 1')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

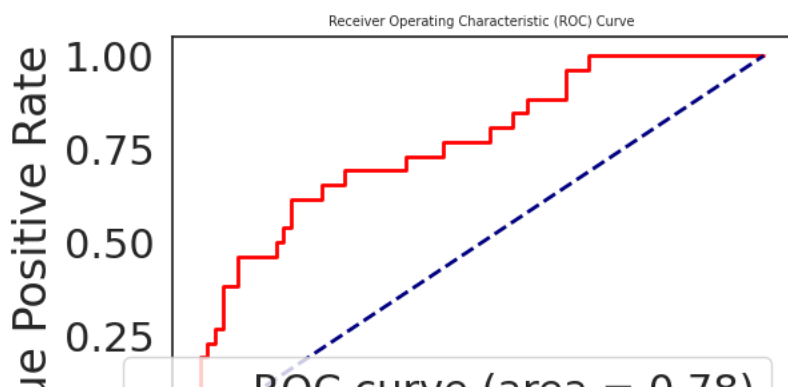
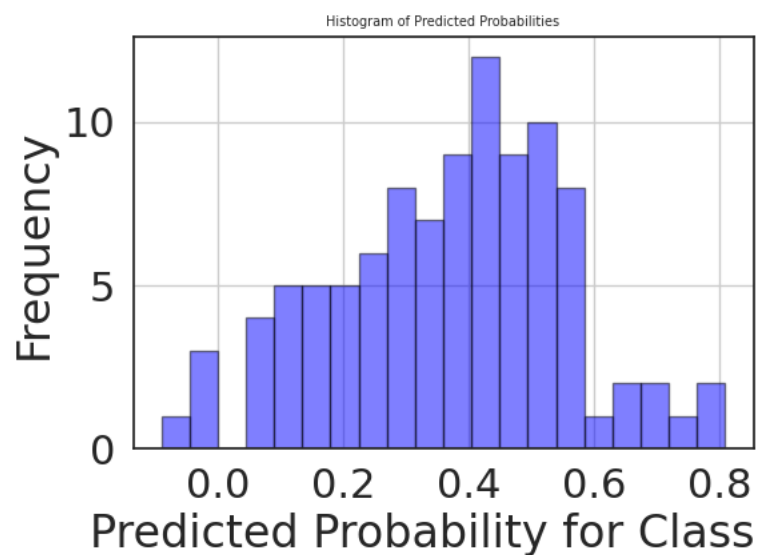
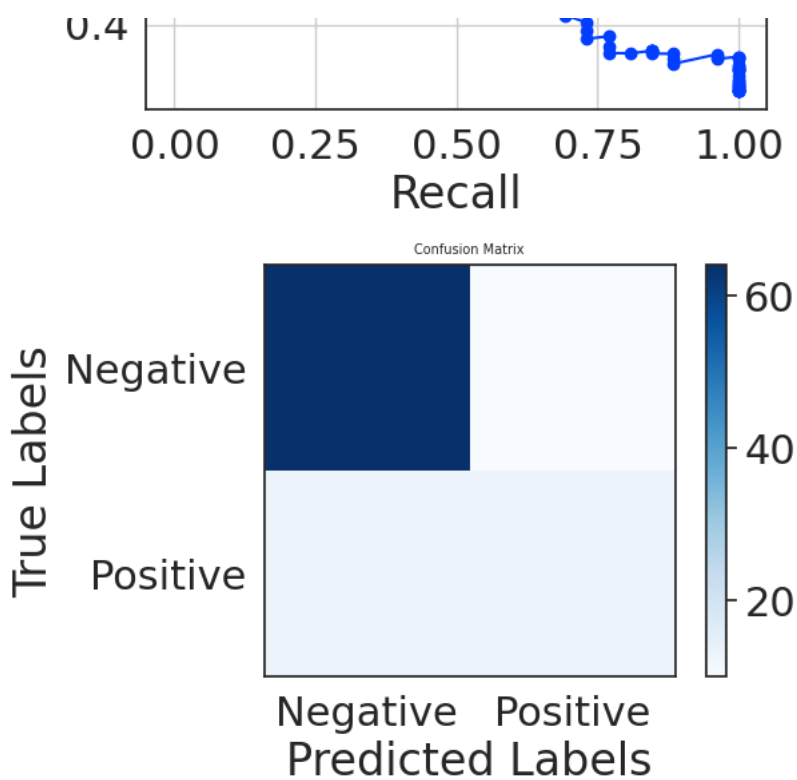
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, roc_curve, auc, precision_recall_curve
from sklearn.model_selection import train_test_split

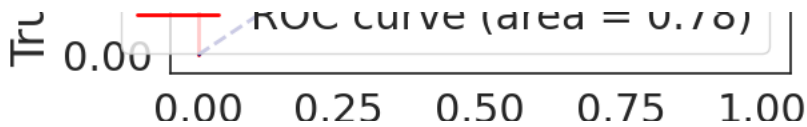
fpr, tpr, _ = roc_curve(np.argmax(Y_test1, axis=1), Y_pred1[:, 1])
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6,4))
plt.plot(fpr, tpr, color='red', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')

plt.show()

```



Logistic Regression

```
import numpy as np
import matplotlib.pyplot as plt

# Calculate precision-recall curve
def precision_recall_curve(y_true, y_scores):
    thresholds = np.linspace(0, 1, num=100)
    precision_values = []
    recall_values = []

    for threshold in thresholds:
        y_pred = (y_scores > threshold).astype(int)
        TP = np.sum(np.logical_and(y_true == 1, y_pred == 1))
        FP = np.sum(np.logical_and(y_true == 0, y_pred == 1))
        FN = np.sum(np.logical_and(y_true == 1, y_pred == 0))

        if TP + FP == 0:
            precision = 1
        else:
            precision = TP / (TP + FP)

        if TP + FN == 0:
            recall = 1
        else:
            recall = TP / (TP + FN)

        precision_values.append(precision)
        recall_values.append(recall)

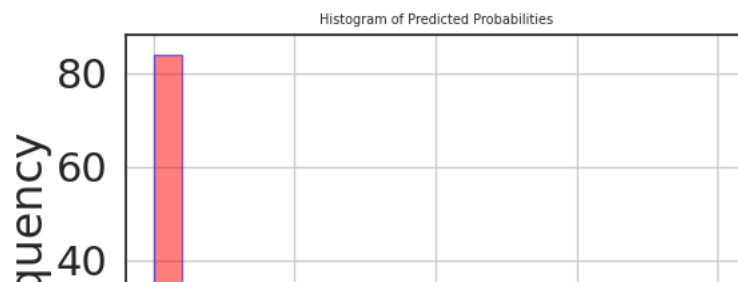
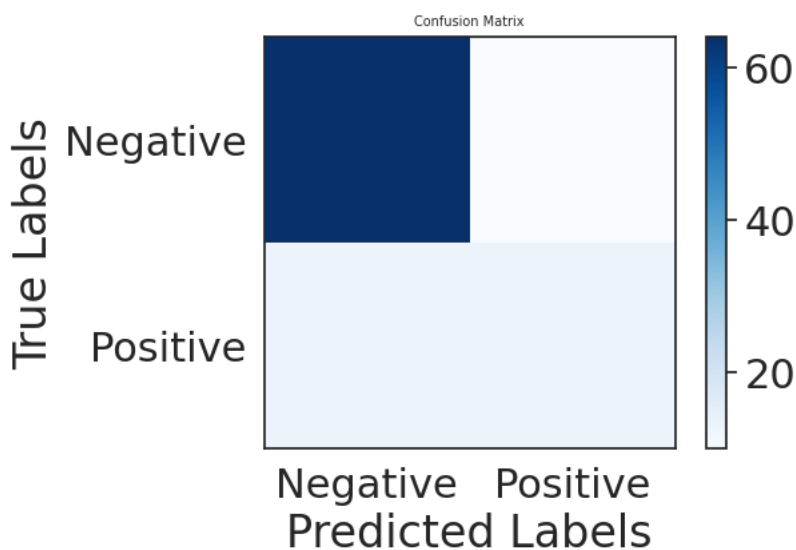
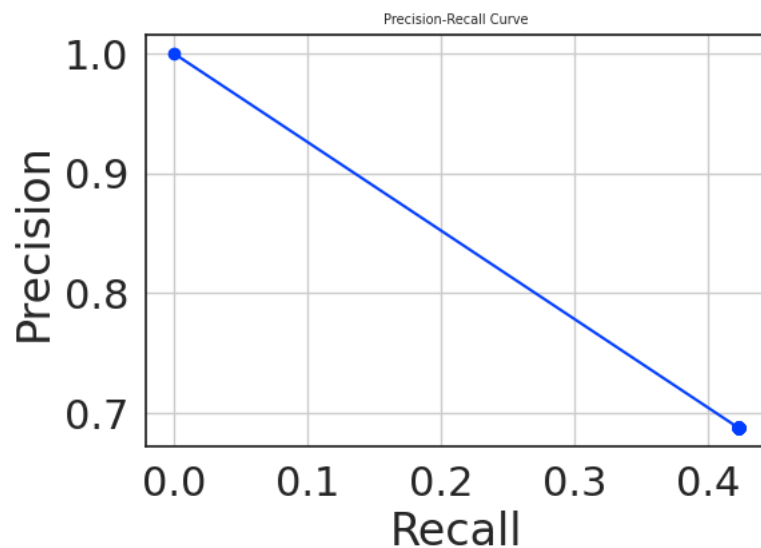
    return precision_values, recall_values

y_scores = clf.predict(X_test).flatten()
precision_values, recall_values = precision_recall_curve(y_test[0], y_scores)

# Plot the precision-recall curve
plt.figure(figsize=(6, 4))
plt.plot(recall_values, precision_values, marker='o')
plt.title('Precision-Recall Curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.grid(True)
plt.show()

# Plot confusion matrix heatmap
plt.figure(figsize=(6,4))
plt.imshow(conf_matrix, cmap='Blues', interpolation='nearest')
plt.colorbar()
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.xticks([0, 1], ['Negative', 'Positive'])
plt.yticks([0, 1], ['Negative', 'Positive'])
plt.show()

# Plot histogram of predicted probabilities
plt.figure(figsize=(6,4))
plt.hist(y_scores, bins=20, alpha=0.5, color='red', edgecolor='blue')
plt.title('Histogram of Predicted Probabilities')
plt.xlabel('Predicted Probability')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



```
from sklearn.metrics import roc_curve, auc

# Calculate ROC curve
def roc_curve_values(y_true, y_scores):
    fpr, tpr, thresholds = roc_curve(y_true, y_scores)
    return fpr, tpr, thresholds

fpr, tpr, thresholds = roc_curve_values(y_test[0], y_scores)

# Calculate AUC (Area Under the ROC Curve)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure(figsize=(6,4))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```