```python
# In this code I have implemented logistic regression using batch gradient without feature scaling and found out the accuracy.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Book1.xlsx')
data=np.array(data)
data=data[1:,:]
df=pd.DataFrame(data)

x=df.iloc[:,0:2]
y=df.iloc[:,2]

# filter out the applicants that got admitted
admitted=df.loc[y==1]
# print(admitted)
 # filter out the applicants that din't get admission
not_admitted=df.loc[y==0]
plt.scatter(admitted.iloc[:, 0], admitted.iloc[:, 1], s=10, label='Admitted')
plt.scatter(not_admitted.iloc[:, 0], not_admitted.iloc[:, 1], s=10, label='Not Admitted')
plt.legend()
# plt.show()

one=np.ones((len(x),1))
x=np.append(one,x,axis=1)
y=np.array(y).reshape((len(y),1))

#splitting the data, 70% for training and 30% for testing
split_pct=int(0.7*len(x))
#print(split_pct)
train_x, test_x = x[:split_pct], x[split_pct:]
train_y, test_y= y[:split_pct], y[split_pct:]

# defining functions to use
def sigmoid(z):
    return 1. / (1 + np.exp(-z))
def z(w, x):
    return np.dot(x,w.T)
def hypothesis(w, x):
    return sigmoid(z(w, x))
def costfunct(w,x,y):
    one_case=-y*np.log(hypothesis(w,x))
    zero_case=-(1-y)*np.log(1-hypothesis(w,x))
    cost= one_case+zero_case
    return (1/len(x))*sum(cost)
def gradient(theta, x, y):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
    return (1 / m) * np.dot((hypothesis(theta,x) - y).T,x)
#Batch GDA
def batch_gradient_descent(X,Y,learning_rate,iterations):
    cost_function = 0  # initalize our costfunct
    m=X.shape[0]
    theta=np.zeros(X.shape[1]).reshape(1,X.shape[1])
    for i in range(0,iterations):
       # prediction = Hypothesis
        prediction =hypothesis(theta,X)
```

```python
        # print(prediction)
        loss=prediction-Y

        theta=theta-learning_rate* sum(gradient(theta,X,Y))
    return theta

# Function to find ACCURACY
def accuracy(test_x,test_y,w,probab_threshold=0.5):
    predicted_classes = (hypothesis(w,test_x) >= probab_threshold).astype(int)
    predicted_classes = predicted_classes.flatten().astype(int)
    test_y=test_y.flatten().astype(int)
    # The following formula also gives the same result
    # h=predicted_classes==test_y
    # the sum() functn of an array with boolean values adds up only trues, i.e. 1
    # In this case 27 are correct predictions out of 30
    # accuracy = print((sum(h)/len(test_y))*100)
    accuracy = np.mean(predicted_classes == test_y)
    return accuracy * 100


# Finding parameters by Batch GDA
# Learning rate=0.01 , epochs=50000
w=batch_gradient_descent(train_x,train_y,0.01,50000)
# Finding accuracy
acc=accuracy(test_x,test_y,w)
print('The accuracy is: ', acc)
w=w.flatten()
# w=[-25.16131856, 0.20623159, 0.20147149]
print("The parameters are:", w)
# Taking two random values of X,
# to put in the equation of straight line to get the values of Y,
# Now having values of X and Y, plotting the graph.
x_values=np.linspace(30,100,2)
# to view X values
# print(x_values)
y_values = - (w[0] + np.dot(w[1], x_values)) / w[2]
# to view Y values
# print(y_values)
''' I could have use this too...to randomly pick X values for plotting the Decision Boundary
x_values = [np.min(x[:, 1] - 5), np.max(x[:, 2] + 5)]'''
plt.plot(x_values, y_values, label='Decision Boundary')
plt.show()


# Finding parameters by Stochatic GDA
# Learning rate=0.01 , epochs=50000
# w1=stochastic_gradient_descent(train_x,train_y,0.01,500)
# print(w1)
```

File Edit Selection View Go Run Terminal Help LogisticRegressionBGDA.py - ML - Visual Studio Code

EXPLORER

ML
- __pycache__
- .vscode
- BatchGDA2.py M
- BatchGDARegularization.py
- BatchGDAwithFS.py
- Book1.xlsx U
- Cleveland.xlsx U
- Hp.xlsx
- Linear Regression using Normal Equ...
- LogisticRegressionBGDA.py U
- LogisticRegressionMIniBatchGD... U
- LogisticRegressionStochasticGD... U
- logisticregressionwithFS.py U
- LRQ1.b.py U
- LRQ2.py U
- LWR.py M
- MinibatchGDA.py
- rough.py
- rough2.py M
- rough3.py U
- rough4.py U
- StochasticGDA.py M
- tempCodeRunnerFile.py M

OUTLINE

TIMELINE

LogisticRegressionBGDA.py U

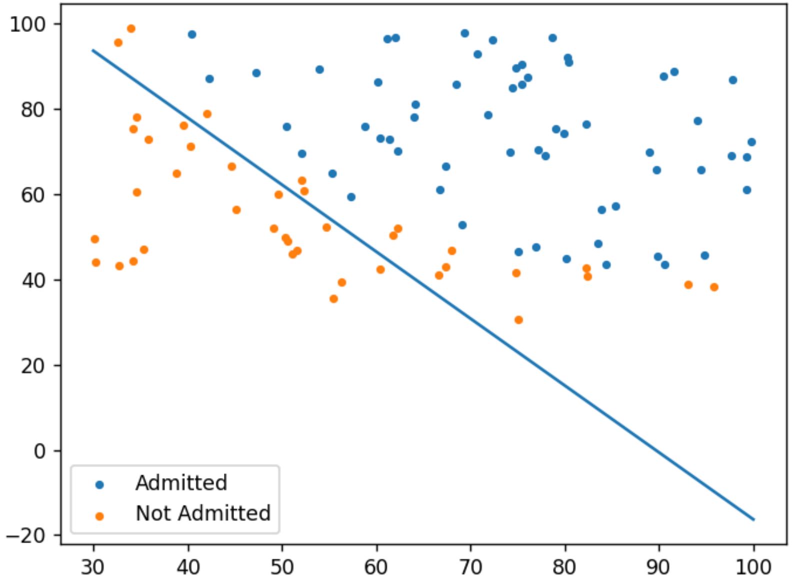
LogisticRegressionBGDA.py > ...

```python
# In this code I have implemented logistic regression using batch gradient without feature scaling and found out the accuracy.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Boo
data=np.array(data)
data=data[1:,:]
df=pd.DataFrame(data)

x=df.iloc[:,0:2]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
PS D:\Prakriti\2 VSCode Python\ML> python -u "d:\Prakriti\2 VSCod
The accuracy is:  90.0
The parameters are: [-31.09844286   0.34704469   0.22116241]
```

Figure 1



x=51.8 y=58.0

Code Code Code Code Code Code Code Code Code Code Code

master* ⊗ 0 ⚠ 0 Ln 63, Col 1 Spaces: 4 UTF-8 CRLF Python 3.10.5 64-bit

27°C Cloudy ENG IN 23:27 09-07-2022

```python
# In this code I have implemented logistic regression using batch gradient with feature scaling and found out the accuracy.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Book1.xlsx')
data=np.array(data)
data=data[1:,:]
df=pd.DataFrame(data)

x=df.iloc[:,0:2]
y=df.iloc[:,2]

# filter out the applicants that got admitted
admitted=df.loc[y==1]
# print(admitted)
 # filter out the applicants that din't get admission
not_admitted=df.loc[y==0]
plt.scatter(admitted.iloc[:, 0], admitted.iloc[:, 1], s=10, label='Admitted')
plt.scatter(not_admitted.iloc[:, 0], not_admitted.iloc[:, 1], s=10, label='Not Admitted')
plt.legend()
# plt.show()

one=np.ones((len(x),1))
x=np.append(one,x,axis=1)
y=np.array(y).reshape((len(y),1))

# performing Feature scaling
# A=Marks in Exam1
# B=Marks in Exam2
A=x[:,1]
A_norm= (A-min(A))/(max(A)-min(A))
B=x[:,2]
B_norm=(B-min(A))/(max(B)-min(B))
X_norm=np.c_[x[:,0],A_norm,B_norm]

#splitting the data, 70% for training and 30% for testing
split_pct=int(0.7*len(X_norm))
#print(split_pct)
train_x, test_x = X_norm[:split_pct], X_norm[split_pct:]
train_y, test_y= y[:split_pct], y[split_pct:]

# defining functions to use
def sigmoid(z):
    return 1. / (1 + np.exp(-z))
def z(w, x):
    return np.dot(x,w.T)
def hypothesis(w, x):
    return sigmoid(z(w, x))
def costfunct(w,x,y):
    one_case=-y*np.log(hypothesis(w,x))
    zero_case=-(1-y)*np.log(1-hypothesis(w,x))
    cost= one_case+zero_case
    return (1/len(x))*sum(cost)
def gradient(theta, x, y):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
```

```python
    return (1 / m) * np.dot((hypothesis(theta,x) - y).T,x)

def batch_gradient_descent(X,Y,learning_rate,iterations):
    cost_function = 0  # initalize our costfunct
    m=X.shape[0]
    theta=np.zeros(X.shape[1]).reshape(1,X.shape[1])
    for i in range(0,iterations):
       # prediction = Hypothesis
        prediction =hypothesis(theta,X)
        # print(prediction)
        loss=prediction-Y

        theta=theta-learning_rate* sum(gradient(theta,X,Y))
    return theta


# Function to find ACCURACY
def accuracy(test_x,test_y,w,probab_threshold=0.5):
    predicted_classes = (hypothesis(w,test_x) >= probab_threshold).astype(int)
    predicted_classes = predicted_classes.flatten().astype(int)
    test_y=test_y.flatten().astype(int)
    # The following formula also gives the same result
    # h=predicted_classes==test_y
    # the sum() functn of an array with boolean values adds up only trues, i.e. 1
    # In this case 27 are correct predictions out of 30
    # accuracy = print((sum(h)/len(test_y))*100)
    accuracy = np.mean(predicted_classes == test_y)
    return accuracy * 100


# Finding parameters by Batch GDA
# Learning rate=0.01 , epochs=50000
w=batch_gradient_descent(train_x,train_y,1,500000)
# Finding accuracy
acc=accuracy(test_x,test_y,w)
print('The accuracy is: ', acc)
w=w.flatten()
# w=[-25.16131856, 0.20623159, 0.20147149]
print("The parameters are:", w)
# Taking two random values of X,
# to put in the equation of straight line to get the values of Y,
# Now having values of X and Y, plotting the graph.
x_values=np.linspace(30,100,2)
# to view X values
# print(x_values)
y_values = - (w[0] + np.dot(w[1], x_values)) / w[2]
# to view Y values
# print(y_values)
''' I could have use this too...to randomly pick X values for plotting the Decision Boundary
x_values = [np.min(x[:, 1] - 5), np.max(x[:, 2] + 5)]'''
plt.plot(x_values, y_values, label='Decision Boundary')
plt.show()
```

```python
# In this code I have implemented Logistic Regression using Stochastic without Feature scaling.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Book1.xlsx')
data=np.array(data)
data=data[1:,:]
df=pd.DataFrame(data)

x=df.iloc[:,0:2]
y=df.iloc[:,2]

# filter out the applicants that got admitted
admitted=df.loc[y==1]
# print(admitted)
 # filter out the applicants that din't get admission
not_admitted=df.loc[y==0]
plt.scatter(admitted.iloc[:, 0], admitted.iloc[:, 1], s=10, label='Admitted')
plt.scatter(not_admitted.iloc[:, 0], not_admitted.iloc[:, 1], s=10, label='Not Admitted')
plt.legend()
# plt.show()

one=np.ones((len(x),1))
x=np.append(one,x,axis=1)
y=np.array(y).reshape((len(y),1))


#splitting the data, 70% for training and 30% for testing
split_pct=int(0.7*len(x))
#print(split_pct)
train_x, test_x = x[:split_pct], x[split_pct:]
train_y, test_y= y[:split_pct], y[split_pct:]


# defining functions to use
def sigmoid(z):
    return 1. / (1 + np.exp(-z))
def z(w, x):
    return np.dot(x,w.T)
def hypothesis(w, x):
    return sigmoid(z(w, x))
def costfunct(w,x,y):
    one_case=-y*np.log(hypothesis(w,x))
    zero_case=-(1-y)*np.log(1-hypothesis(w,x))
    cost= one_case+zero_case
    return (1/len(x))*sum(cost)
def gradient(theta, x, y):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
    return (1 / m) * np.dot((hypothesis(theta,x) - y).T,x)


#Stochastic GDA
def stochastic_gradient_descent(x,y,learning_rate,iterations):
    theta = np.zeros(x.shape[1])
    for i in range(iterations):
```

```python
            random_index=np.random.randint(len(x),dtype='int')
            xi = x[random_index,0:3].reshape(1,x.shape[1])
            yi = y[random_index,0].reshape(1,1)

            prediction = hypothesis(theta,xi)
            loss=prediction-yi
            gradient=np.dot(loss,xi)
            # Updating the parameters i.e.theta here
            theta = theta - learning_rate*gradient
    return theta

# Function to find ACCURACY
def accuracy(test_x,test_y,w,probab_threshold=0.5):
    predicted_classes = (hypothesis(w,test_x) >= probab_threshold).astype(int)
    predicted_classes = predicted_classes.flatten().astype(int)
    test_y=test_y.flatten().astype(int)
    # The following formula also gives the same result
    # h=predicted_classes==test_y
    # the sum() functn of an array with boolean values adds up only trues, i.e. 1
    # In this case 27 are correct predictions out of 30
    # accuracy = print((sum(h)/len(test_y))*100)
    accuracy = np.mean(predicted_classes == test_y)
    return accuracy * 100


# Finding parameters by Stochatic GDA
# Learning rate=0.01 , epochs=50000
w=stochastic_gradient_descent(train_x,train_y,0.01,50000)
# Finding accuracy
acc=accuracy(test_x,test_y,w)
print('The accuracy is: ', acc)
w=w.flatten()
# w=[-25.16131856, 0.20623159, 0.20147149]
print("The parameters are:", w)
# Taking two random values of X,
# to put in the equation of straight line to get the values of Y,
# Now having values of X and Y, plotting the graph.
x_values=np.linspace(30,100,2)
# to view X values
# print(x_values)
y_values = - (w[0] + np.dot(w[1], x_values)) / w[2]
# to view Y values
# print(y_values)
''' I could have use this too...to randomly pick X values for plotting the Decision Boundary
x_values = [np.min(x[:, 1] - 5), np.max(x[:, 2] + 5)]'''
plt.plot(x_values, y_values, label='Decision Boundary')
plt.show()
```

File Edit Selection View Go Run Terminal Help ● LogisticRegressionStochasticGDA.py - ML - Visual Studio Code

EXPLORER

ML
- __pycache__
- .vscode
- BatchGDA2.py M
- BatchGDARegularization.py
- BatchGDAwithFS.py
- Book1.xlsx U
- Cleveland.xlsx U
- Hp.xlsx
- Linear Regression using Normal Equ...
- LogisticRegressionBGDA.py U
- LogisticRegressionStochasticGD... U
- logisticregressionwithFS.py U
- LRQ1.b.py U
- LRQ2.py U
- LWR.py M
- MinibatchGDA.py
- rough.py
- rough2.py M
- rough3.py U
- rough4.py U
- StochasticGDA.py M
- tempCodeRunnerFile.py M

OUTLINE

TIMELINE

LogisticRegressionStochasticGDA.py U ● Book1.xlsx U

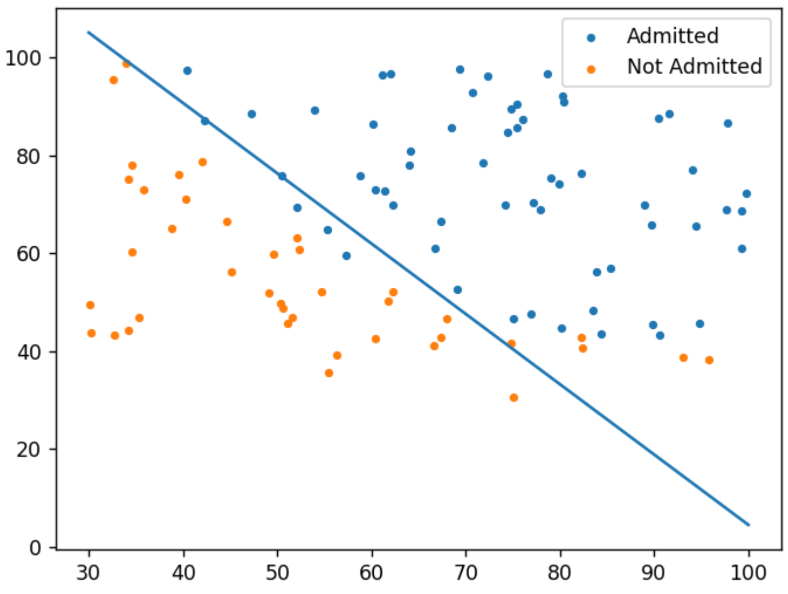
LogisticRegressionStochasticGDA.py > ...

```python
# In this code I have implemented Logistic Regression using Stochastic without Feature scaling.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Boo
data=np.array(data)
data=data[1:,:]
df=pd.DataFrame(data)

x=df.iloc[:,0:2]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
PS D:\Prakriti\2 VSCode Python\ML>

                                > python -u "d:\Prakriti\2 VSCod
The accuracy is:  86.66666666666667
The parameters are: [-35.9484987    0.3485414    0.24252846]
```

Figure 1



master* ⊗ 0 ⚠ 0 Ln 1, Col 96 Spaces: 4 UTF-8 CRLF Python 3.10.5 64-bit

28°C Cloudy ENG IN 22:49 09-07-2022

```python
# In this code I have implemented Logistic Regression using Mini Batch without Feature scaling.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Book1.xlsx')
data=np.array(data)
data=data[1:,:]
df=pd.DataFrame(data)

x=df.iloc[:,0:2]
y=df.iloc[:,2]

# filter out the applicants that got admitted
admitted=df.loc[y==1]
# print(admitted)
 # filter out the applicants that din't get admission
not_admitted=df.loc[y==0]
plt.scatter(admitted.iloc[:, 0], admitted.iloc[:, 1], s=10, label='Admitted')
plt.scatter(not_admitted.iloc[:, 0], not_admitted.iloc[:, 1], s=10, label='Not Admitted')
plt.legend()
# plt.show()

one=np.ones((len(x),1))
x=np.append(one,x,axis=1)
y=np.array(y).reshape((len(y),1))

#splitting the data, 70% for training and 30% for testing
split_pct=int(0.7*len(x))
#print(split_pct)
train_x, test_x = x[:split_pct], x[split_pct:]
train_y, test_y= y[:split_pct], y[split_pct:]


# defining functions to use
def sigmoid(z):
    return 1. / (1 + np.exp(-z))
def z(w, x):
    return np.dot(x,w.T)
def hypothesis(w, x):
    return sigmoid(z(w, x))
def costfunct(w,x,y):
    one_case=-y*np.log(hypothesis(w,x))
    zero_case=-(1-y)*np.log(1-hypothesis(w,x))
    cost= one_case+zero_case
    return (1/len(x))*sum(cost)
def gradient(theta, x, y):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
    return (1 / m) * np.dot((hypothesis(theta,x) - y).T,x)


#Mini Batch GDA
def mini_batch_gradient_descent(x,y,iterations,learning_rate,batch_size=20):
    theta = np.zeros(x.shape[1]).reshape(1,x.shape[1])
    shuffled_indices = np.random.permutation(len(y))
    x_shuffled = x[shuffled_indices]
```

```python
    y_shuffled = y[shuffled_indices]
    for i in range(iterations):
        xi = x_shuffled[i:i+batch_size]
        yi = y_shuffled[i:i+batch_size]
        prediction = hypothesis(theta,xi)
        loss=prediction-yi
        gradient=sum(np.dot(loss.T,xi))
        # Updating the parameters i.e.theta here
        theta =theta - (learning_rate * gradient)/batch_size

    return theta

# Function to find ACCURACY
def accuracy(test_x,test_y,w,probab_threshold=0.5):
    predicted_classes = (hypothesis(w,test_x) >= probab_threshold).astype(int)
    predicted_classes = predicted_classes.flatten().astype(int)
    test_y=test_y.flatten().astype(int)
    # The following formula also gives the same result
    # h=predicted_classes==test_y
    # the sum() functn of an array with boolean values adds up only trues, i.e. 1
    # In this case 27 are correct predictions out of 30
    # accuracy = print((sum(h)/len(test_y))*100)
    accuracy = np.mean(predicted_classes == test_y)
    return accuracy * 100


# Finding parameters by Mini Batch GDA
# Learning rate=0.01 , epochs=50000
w=mini_batch_gradient_descent(train_x,train_y,25,0.0001)
# Finding accuracy
acc=accuracy(test_x,test_y,w)
print('The accuracy is: ', acc)
w=w.flatten()
# w=[-25.16131856, 0.20623159, 0.20147149]
print("The parameters are:", w)
# Taking two random values of X,
# to put in the equation of straight line to get the values of Y,
# Now having values of X and Y, plotting the graph.
x_values=np.linspace(30,100,2)
# to view X values
# print(x_values)
y_values = - (w[0] + np.dot(w[1], x_values)) / w[2]
# to view Y values
# print(y_values)
'''I could have use this too...to randomly pick X values for plotting the Decision Boundary
x_values = [np.min(x[:, 1] - 5), np.max(x[:, 2] + 5)] '''
plt.plot(x_values, y_values, label='Decision Boundary')
plt.show()
```

```python
# In this code I have implemented Logistic Regression for multiclass Regression using sklearn libraryon Cleveland data
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression

from warnings import simplefilter
# ignore all future warnings
simplefilter(action='ignore', category = FutureWarning)


df = pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Cleveland.xlsx')


df.columns = ['age', 'sex', 'cp', 'trestbps', 'chol',
              'fbs', 'restecg', 'thalach', 'exang',
              'oldpeak', 'slope', 'ca', 'thal', 'target']
# print(df)
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

from sklearn.metrics import confusion_matrix
cm_test = confusion_matrix(y_pred, y_test)

y_pred_train = classifier.predict(X_train)
cm_train = confusion_matrix(y_pred_train, y_train)

print()
print('Accuracy for training set for Logistic Regression = {}'.format((cm_train[0][0] + cm_train[1][1])/len(y_train)))
print('Accuracy for test set for Logistic Regression = {}'.format((cm_test[0][0] + cm_test[1][1])/len(y_test)))

```

LRQ2.py - ML - Visual Studio Code

```python
# In this code I have implemented Logistic Regression for multiclass Regression using sklearn libraryon Cleveland data
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression

df = pd.read_excel(r'D:\Prakriti\2 VSCode Python\ML\Cleveland.xlsx')


df.columns = ['age', 'sex', 'cp', 'trestbps', 'chol',
              'fbs', 'restecg', 'thalach', 'exang',
              'oldpeak', 'slope', 'ca', 'thal', 'target']
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
Open file in editor (ctrl + click) thon\ML> python -u "d:\Prakriti\2 VSCode Python\ML\LRQ2.py"
C:\Users\hp\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\linear_model\_logistic.py:444: ConvergenceWarning: lbfgs failed to converge
(status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
[[31  9]
 [ 4 17]]

Accuracy for training set for Logistic Regression = 0.8512396694214877
Accuracy for test set for Logistic Regression = 0.7868852459016393
PS D:\Prakriti\2 VSCode Python\ML>
```