Lab Session-III(a)
(Based on Logistic Regression)

Dr. JASMEET SINGH
ASSISTANT PROFESSOR, CSED
TIET, PATIALA

Introduction- Logistic Regression

- In Logistic Regression, we use logistic (sigmoid) hypothesis function to predict the values corresponding to input variables.

$$y^{\wedge} = f(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots \ldots\ldots\ldots \beta_k x_k)}}$$

- The predicted value of hypothesis lies between 0 and 1 (i.e., probability of label 1 for the given values of input variables and constant β's).

- Depending upon the values of f(x) labels are assigned as 0 or 1 as follows:

$$y_{label} = \begin{cases} 1\, if & y^{\wedge} \geq 0.5 \\ 0 if & y^{\wedge} < 0.5 \end{cases}$$

How to find optimal values of β's ?

- In order to find optimal value of β's, we apply gradient descent optimization using following steps:

1. Initialize $\beta_0 = 0$, $\beta_1 = 0$, $\beta_2 = 0$,……………………………… $\beta_k = 0$
2. Update parameters until convergence or for fixed number of iterations using following equation:

$$\beta_j = \beta_j - \alpha \times \frac{\partial J}{\partial \beta_j}$$

for j=0,1,2,3……………….k

where k are the total number of features; α *is the learning rate*

$$\frac{\partial J}{\partial \beta_j} = \frac{1}{n} \times (\sum_{i=1}^{n} (f(x_i) - y_i)$$

where $f(x_i) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_{i1}+\beta_2 x_{i2}+\beta_3 x_{i3}+\cdots\ldots\ldots\ldots\beta_k x_{ik})}}$ and y_i is the actual label of the ith training example

Implementation Logistic Regression (Step-by-Step)

- Following steps are followed for implementation of logistic regression:

1. Load the dataset.
2. Handle Null Values, remove noise, outliers, check for class balancing (if required).
3. Separate the dataset into X (input/independent variables) and Y (dependent feature). Scale the feature values of X in a fixed range and add a new column (in the beginning) with all values 1 (for vector implementation of .
4. Split the dataset into train and test set.
5. Using the train set to find the optimal value of $\boldsymbol{\beta}\hat{}$ matrix (coefficients) for which cost function is minimum.
6. Predict the values of the output variable on the test set and label examples on the basis of prediction.
7. Perform the performance evaluation of the trained model.

Step 1 (Logistic Regression): Load the Dataset

- For implementation of Logistic Regression we will use heart attack labeling dataset downloadable from the following link:

https://drive.google.com/file/d/1hj_5K51WKKcOscHrjteXuWFu87Coe8XG/view?usp=sharing

- The dataset checks the presence of heart disease in the patient on the basis of age, sex, chest pain type, resting blood pressure, serum cholesterol, fasting blood sugar, electrocardiographic results, maximum heart rate, exercise induced angina, oldpeak, he slope of the peak, number of major vessels colored by flourosopy, thal: 3 = normal; 6 = fixed defect; 7 = reversable defect

Code:

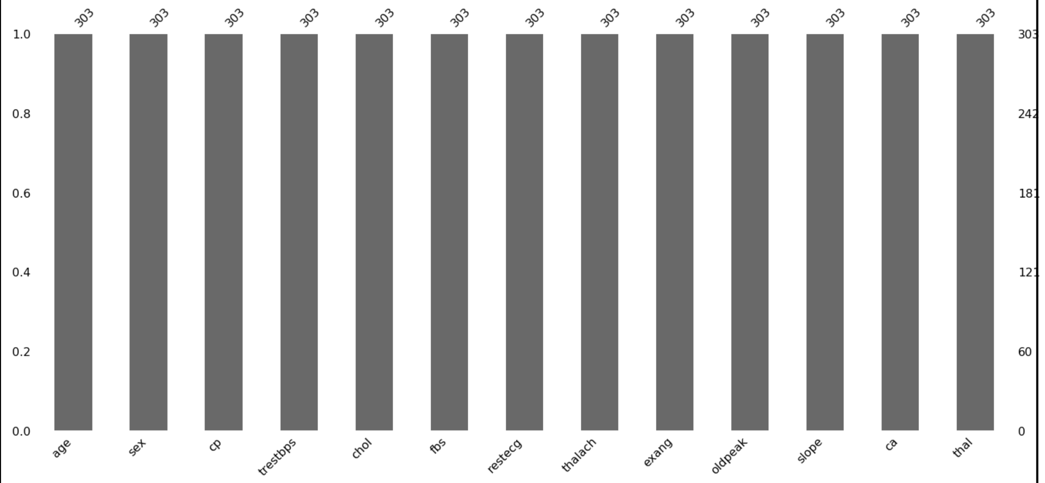
```
import pandas as pd

df=pd.read_csv('C:/Users/jasme/Downloads/heart.csv')

df.info()
```

Step 2: Pre-processing

Checking for NULL values:

import missingno as msn

msn.bar(df.iloc[:,0:13])

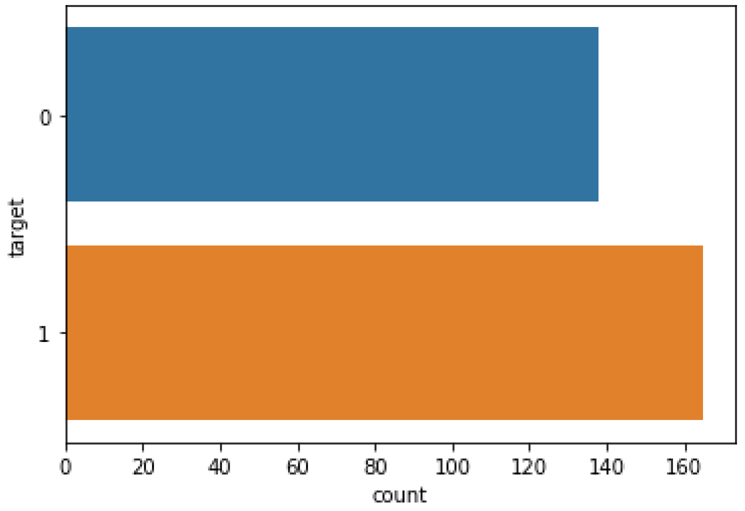


There are no missing values in any input feature.

Checking for class balancing:

import seaborn as sns

sns.countplot(y=df.iloc[:,13],data=df)



Approximate number of samples are balanced; so no class balancing.

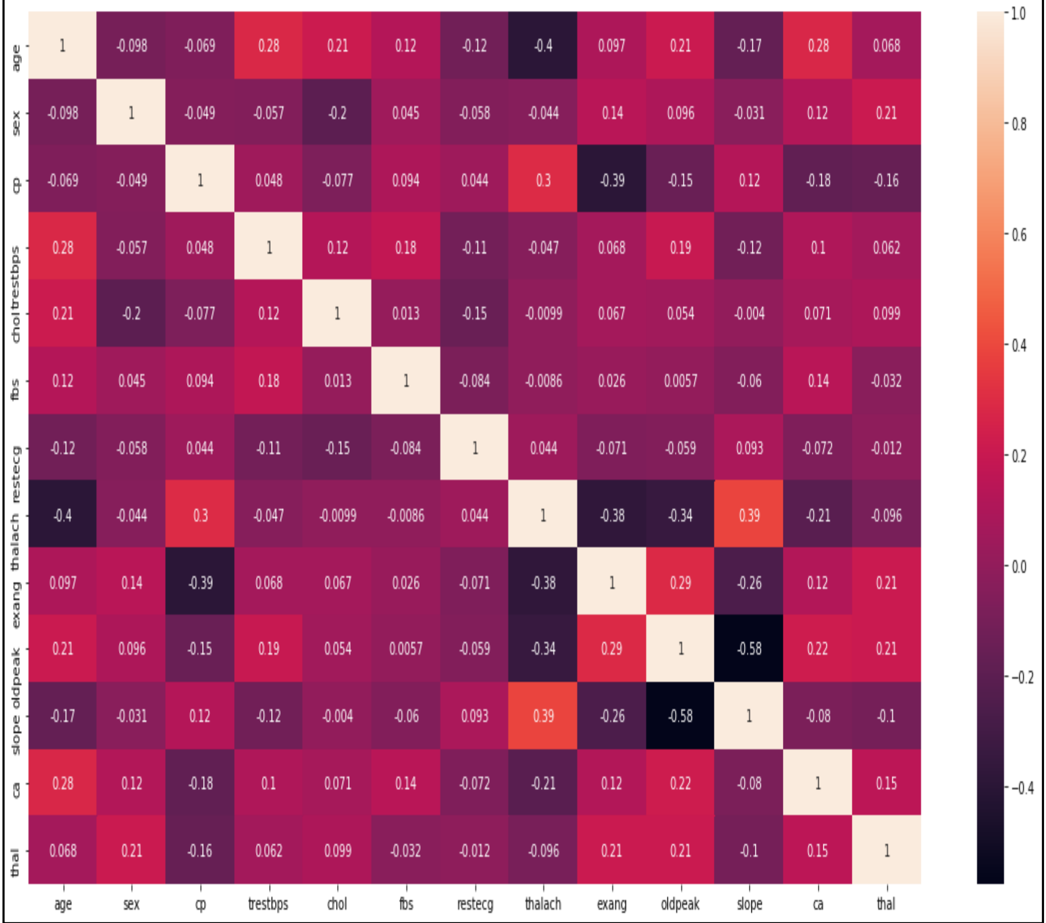
Step 2: Pre-processing (Contd…)

Checking for correlation between input features:

Code:

```
sns.heatmap(df.iloc[:,:13].corr(),annot=True)
```

- As clear from the figure, the correlation values are quite less (less than 0.5)



	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
age	1	-0.098	-0.069	0.28	0.21	0.12	-0.12	-0.4	0.097	0.21	-0.17	0.28	0.068
sex	-0.098	1	-0.049	-0.057	-0.2	0.045	-0.058	-0.044	0.14	0.096	-0.031	0.12	0.21
cp	-0.069	-0.049	1	0.048	-0.077	0.094	0.044	0.3	-0.39	-0.15	0.12	-0.18	-0.16
trestbps	0.28	-0.057	0.048	1	0.12	0.18	-0.11	-0.047	0.068	0.19	-0.12	0.1	0.062
chol	0.21	-0.2	-0.077	0.12	1	0.013	-0.15	-0.0099	0.067	0.054	-0.004	0.071	0.099
fbs	0.12	0.045	0.094	0.18	0.013	1	-0.084	-0.0086	0.026	0.0057	-0.06	0.14	-0.032
restecg	-0.12	-0.058	0.044	-0.11	-0.15	-0.084	1	0.044	-0.071	-0.059	0.093	-0.072	-0.012
thalach	-0.4	-0.044	0.3	-0.047	-0.0099	-0.0086	0.044	1	-0.38	-0.34	0.39	-0.21	-0.096
exang	0.097	0.14	-0.39	0.068	0.067	0.026	-0.071	-0.38	1	0.29	-0.26	0.12	0.21
oldpeak	0.21	0.096	-0.15	0.19	0.054	0.0057	-0.059	-0.34	0.29	1	-0.58	0.22	0.21
slope	-0.17	-0.031	0.12	-0.12	-0.004	-0.06	0.093	0.39	-0.26	-0.58	1	-0.08	-0.1
ca	0.28	0.12	-0.18	0.1	0.071	0.14	-0.072	-0.21	0.12	0.22	-0.08	1	0.15
thal	0.068	0.21	-0.16	0.062	0.099	-0.032	-0.012	-0.096	0.21	0.21	-0.1	0.15	1

Step 3: Splitting Input and Output features

- Separate the dataset into X (input/independent variables) and Y (dependent feature).
- Scale the feature values of X in a fixed range.
- Add a new column (in the beginning) with all values 1.

Code:

```
from sklearn.preprocessing import StandardScaler

import numpy as np

X=df.iloc[:,0:13]

Y=df.iloc[:,13]

scaler=StandardScaler()

X_scaled=scaler.fit_transform(X)

X_scaled=np.insert(X_scaled,0,values=1,axis=1)
```

Step 4: Train/Test Split

- We can split the train and test sets using train/test split of sklearn.model_selection as follows:

Code:

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.3, random_state=42)
```

Parameters:

test_size is the percentage of test set from the total dataset; random_state s used for initializing the internal **random** number generator, which will decide the **splitting** of data into **train** and **test** indices

If random_state is None or np.random, then a randomly-initialized RandomState object is returned.

If random_state is an integer, then it is used to seed a new RandomState object.

Step 5: Optimal β's using Gradient Descent

Code:

```
n=1000 #number of iterations
alpha=0.01
m,k=X_train.shape
beta=np.zeros(k)
for i in range(n):
    cost_gradient=np.zeros(k)
    z=X_train.dot(beta)
    predicted=1/(1+np.exp(-z))
    difference=predicted-Y_train
    for j in range(k):
        cost_gradient[j]=np.sum(difference.dot(X_train[:,j]))
    for j in range(k):
        beta[j]=beta[j]-(alpha/m)*cost_gradient[j]
print(beta)
```

Step 6: Predicting and Labeling

- In this step, we predict the values of output variable on the test set.
- The predicted value for each test example is computed as:

$$y^{\wedge} = \frac{1}{1 + e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\beta_3 x_3+\cdots\ldots\ldots\ldots\beta_k x_k)}}$$

- Depending upon the values of $y^{\wedge}$, labels are assigned as 0 or 1 as follows:

$$y_{label} = \begin{cases} 1\ if \quad y^{\wedge} \geq 0.5 \\ 0 if \quad y^{\wedge} < 0.5 \end{cases}$$

Code:

```
Y_predict=1/(1+np.exp(-(X_test.dot(beta))))
Y_label=np.zeros(len(Y_predict))
for i in range(len(Y_predict)):
    if(Y_predict[i]>=0.5):
        Y_label[i]=1
```

Step 7: Performance Evaluation

The performance of a classification model is computed is measured in terms of Precision, Recall, F1-Score and accuracy.

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\ Score = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Where, TP is True Positive, TN is True Negative, FP is False Positive, and FN is False Negative

The above formulas gives metrics for positive class, similarly it can be computed for negative class and the average is then computed using macro (simple mean) average and weighted average.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Step 7: Performance Evaluation Contd…

Code:

```
TP=0
TN=0
FP=0
FN=0
Y_test=np.array(Y_test).reshape(-1,1)
for i in range(len(Y_label)):
    if(Y_test[i]==1 and Y_label[i]==1):
        TP=TP+1
    if(Y_test[i]==1 and Y_label[i]==0):
        FN=FN+1
    if(Y_test[i]==0 and Y_label[i]==1):
        FP=FP+1
    if(Y_test[i]==0 and Y_label[i]==0):
        TN=TN+1
print(TP,TN,FP,FN)
```

```
accuracy=(TP+TN)/(TP+TN+FP+FN)
#For positive class:
precision_pos=TP/(TP+FP)
recall_pos=TP/(TP+FN)
f1_score_pos=2*precision_pos*recall_pos/(precision_pos+recall_pos)
print(precision_pos,recall_pos,f1_score_pos)
#For negative class
precision_neg=TN/(TN+FN)
recall_neg=TN/(TN+FP)
f1_score_neg=2*precision_neg*recall_neg/(precision_neg+recall_neg)
print(precision_neg,recall_neg,f1_score_neg)
```

Step 7: Performance Evaluation Contd…

Macro Average:

```
macro_precision=(precision_pos+precision_neg)/2

macro_recall=(recall_pos+recall_neg)/2

macro_f1_score=(f1_score_pos+f1_score_neg)/2

print(macro_precision,macro_recall,macro_f1_score)
```

Weighted Average:

```
l1=len(Y_test[Y_test==0])

l2=len(Y_test[Y_test==1])

weighted_precision=(l1*precision_neg+l2*precision_pos)/(l1+l2)

weighted_recall=(l1*recall_neg+l2*recall_pos)/(l1+l2)

weighted_f1_score=(l1*f1_score_neg+l2*f1_score_pos)/(l1+l2)

print(weighted_precision,weighted_recall,weighted_f1_score)
```

Logistic Regression using In-built Function

- We can also use inbuilt function for Logistic Regression in sklearn using following code:

```
from sklearn.linear_model import LogisticRegression

from sklearn import metrics

lr=LogisticRegression(solver='sag’)

#sag denote Stochastic Average Gradient

model=lr.fit(X_train,Y_train)

Y_label1=model.predict(X_test)

print(metrics.classification_report(Y_test,Y_label1))
```