Data Analysis and Machine-Learning

Chapter 3.2:

Generalized Linear Model (2) Logit Function and Logistic Regression

1. MLE (Maximum Likelihood Estimation)

When the probability function for a given dataset is defined as:

$$L(\theta; x, y)$$

Given that the distribution of x and y are dependent on the parameter θ, our goal is to estimate the parameter θ that maximizes the possibility function. When assuming normal distribution as regards to the dataset, the given probability function would be a probability density function of a normal distribution. Thus, the parameter for estimation can be defined as average and standard deviation:

$$L(\theta; x, y) = \prod_{i=1}^{n} f(x_i, y_i; \theta)$$

As such, possibility function refers to a type of function for the parameters defined as the multiplication of probability density function.

1.1. Maximum Log Likelihood

A problem of calculating by "multiplication" for probability density function is that the multiplication of probability values results in continuously diminishing value; in other words, the more you multiply by probability (for p<=1), the value only becomes smaller. This is why maximum log likelihood method converts multiplication to aggregation (sum):

$$L(\theta; x, y) = \sum_{i=1}^{n} \log\left(f(x_i, y_i; \theta)\right)$$

1.2. Maximum Negative Log Likelihood

In order to synchronize with general error functions which are formed in a direction of minimizing the value of errors maximum negative log likelihood adopts negative sign:

$$L(\theta; x, y) = \sum_{i=1}^{n} -\log\left(f(x_i, y_i; \theta)\right)$$

2. From Linear Regression Model to Classification Model

In order to process classification data using generalized linear model, let us start by adopting linear regression model and converting it by applying appropriate function.

As covered in chapter 2, the fundamental structure of linear regression model can be defined as:

$$f(x) = ax + b$$

For binary classifications, it is possible to assume a model following Bernoulli distribution as follows:

$$y_i \sim Ber(f(x))$$

This being the case, it is possible to model stochastic estimation:

$$f(x) = \mathbb{P}(Y = 1|x)$$

Thus, it is necessary to find a way to convert the following model into values from 0 to 1 (as in the case for probabilities):

$$\mathbb{P}(Y = 1|x) = f(ax + b)$$

The function 'f(x)' should be (1) continuous, (2) increasing/decreasing, and (3) consisting of values between 0 and 1. This is why, representatively, the cdf (cumulative distribution function) of normal distribution is adopted as the activation function for the conversion of (ax+b), e.g., logit and sigmoid.

Here, I illustrate the mathematical application process for conversion:

For binary classifications (y = 0 or 1):

$$P(y = 0) + P(y = 1) = 1$$

Solving Formula for Logit conversion:

$$\log\left(\frac{p(y = 1|x)}{p(y = 0|x)}\right) = ax + b$$

$$\left(\frac{p(y = 1|x)}{p(y = 0|x)}\right) = e^{ax+b}$$

$$\left(\frac{p(y = 1|x)}{1 - p(y = 1|x)}\right) = e^{ax+b}$$

$$\therefore p(y = 1|x) = e^{ax+b}(1 - p(y = 1|x)) = e^{ax+b} - e^{ax+b}p(y = 1|x)$$

$$p(y = 1|x)(1 + e^{ax+b}) = e^{ax+b}$$

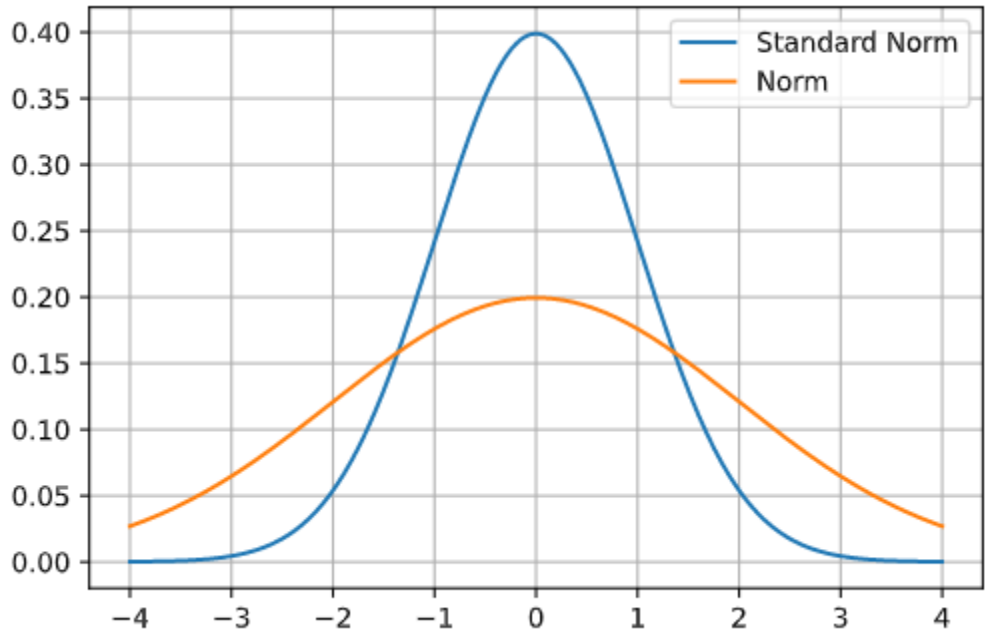
$$\therefore p(y = 1|x) = \frac{e^{ax+b}}{1 + e^{ax+b}} = \frac{1}{e^{-(ax+b)} + 1}$$

※pdf (probability density function) and cdf (cumulative distribution function)

Continuous random variables refer to the case where random variables have continuous values, and the probability distribution of such random variable x is referred to as probability density function. When visualizing the representative case of normal distribution:

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
stdRv = norm(loc=0, scale=1)
rv = norm(loc=0, scale=2)
x = np.linspace(-4, 4, 1500)
plt.plot(x, stdRv.pdf(x), label='Standard Norm')
plt.plot(x, rv.pdf(x), label='Norm')
plt.legend()
plt.grid()
plt.show()
```

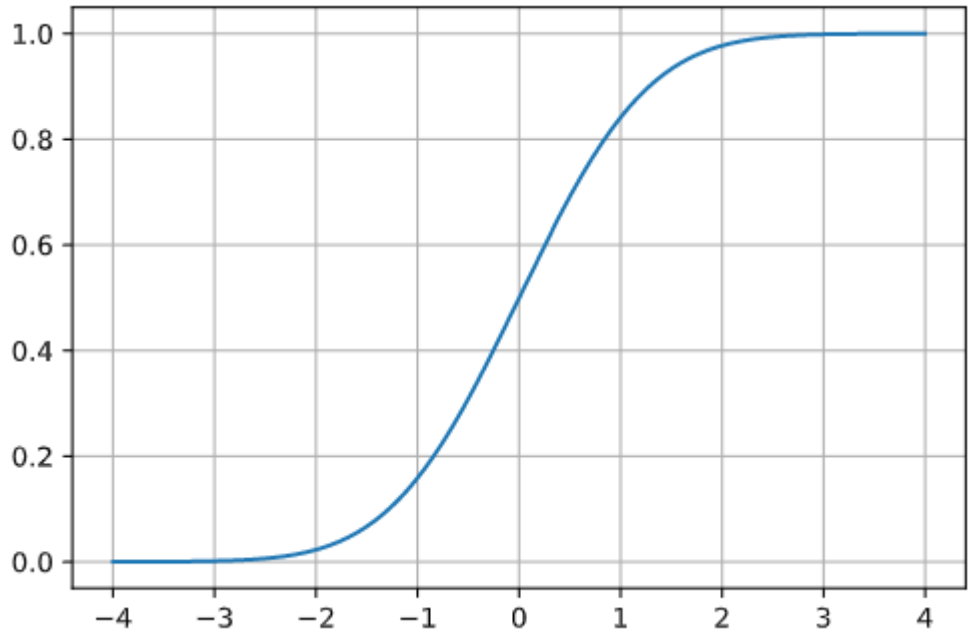
Output:



Cumulative distribution function “cumulates” each value of probability mass function (via summation), which is why the cdf of normal distribution is shaped as:

```
x = np.linspace(-4, 4, 1500)
plt.plot(x, stdRv.cdf(x))
plt.grid()
plt.show()
```

Output:



3. Application

Putting all pieces together, now we get the basic structure of generalized linear model to sort classification datasets (logit), and maximum likelihood estimation model for optimization (cross-entropy), each defined as:

$$f(x) = logit(ax + b)$$

$$E = \sum_i -y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)$$

Now, it is possible to conduct actual applications on sample datasets.

#Import necessary libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn import datasets
from sklearn.linear_model import LogisticRegression
import scipy.stats as sp
from scipy.optimize import minimize
from scipy.special import logit
```

#Generate sample datasets

```
x, y = datasets.make_classification(
    n_samples=8, n_features=2, n_classes=2, random_state=123,
```

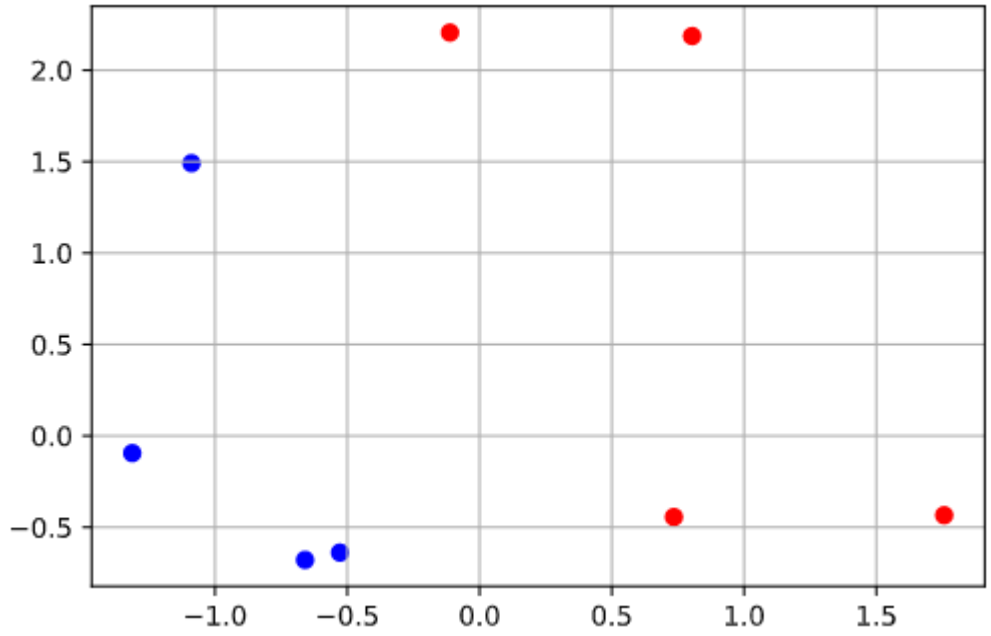
```
    n_informative=1, n_redundant=0, n_clusters_per_class=1
)
np.c_[y, x]
```

output:

```
array([[ 1.        , -0.11166034,  2.20593008],
       [ 1.        ,  1.75652211, -0.43435128],
       [ 1.        ,  0.80351538,  2.18678609],
       [ 0.        , -1.0887545 ,  1.49138963],
       [ 0.        , -0.52755975, -0.638902  ],
       [ 0.        , -0.65949851, -0.67888615],
       [ 1.        ,  0.73494356, -0.44398196],
       [ 0.        , -1.31281138, -0.09470897]])
```

#Visualize scatterplot

```
plt.scatter(x[:,0], x[:,1], c=y, cmap='bwr')
plt.grid()
plt.show()
```



#Define logit function

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

#Define cross-entropy

```
def cross_entropy(beta):
  yhat = logit( np.dot(x, beta[1:]) + beta[0] )
```

```
  return np.mean( - y * np.log(yhat) - (1 - y) * np.log(1 - yhat)  )
```

#Find beta coefficients

```
beta = np.array([0, 0, 0])
result = minimize(cross_entropy, beta)
result.x
```

Output:

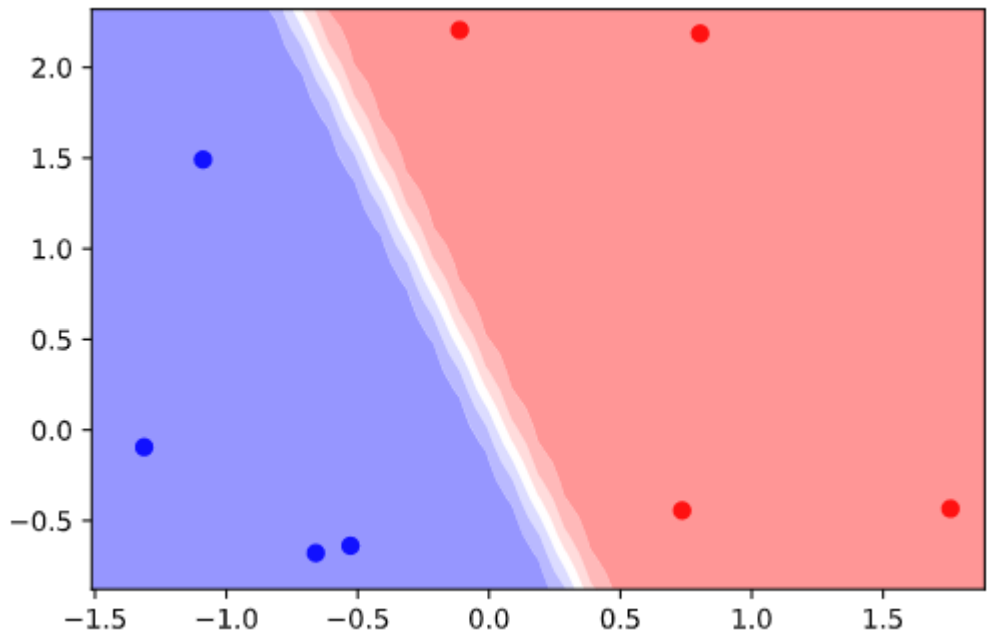
```
array([-0.80219006, 18.23175153,  6.11756816])
```

#Visualize the model estimation results

```
x1, x2 = x[:,0], x[:,1]
xy1, xy2 = np.meshgrid(
    np.arange(x1.min() - 0.2, x1.max() + 0.2, 0.1),
    np.arange(x2.min() - 0.2 , x2.max() + 0.2 , 0.1)
)

beta = result.x
xx = np.c_[xy1.flatten(), xy2.flatten()]
yhat = logit(np.dot(xx, beta[1:]) + beta[0] )

plt.scatter(x1, x2, c = y, cmap='bwr')
plt.contourf(xy1, xy2, yhat.reshape(xy1.shape), cmap='bwr', alpha=0.48)
plt.show()
```



4. Evaluation Methods for Classification Models

		Actual Y	
		True (e.g., 1)	False (e.g., 0)
Estimation Result (Yhat)	True (e.g., 1)	True Positive (TP) (Actual: 1, Estimation: 1)	False Positive (FP) (Actual: 0, Estimation: 1)
	False (e.g., 0)	False Negative (FN) (Actual: 1, Estimation: 0)	True Negative (TN) (Actual: 0, Estimation: 0)

#Accuracy

The ratio of TP and TN out of all data:

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

If data imbalances are severe, accuracy scores may not be an accurate indicator. For instance, if the number of TP is overwhelmingly large, TN misclassification rates are not reflected concisely.

#Precision

The ratio of True Positive out of True Positive and False Positive, i.e., the ratio of actual True out of Trues predicted by the model:

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

Recall

The ratio of Trues predicted by the model, out of actual Trues:

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

#F-1 Score

The harmonic mean of Precision and Recall:

$$2 \cdot \frac{1}{\frac{1}{Precision} + \frac{1}{Recall}} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

As illustrated, F1-score reflects both precision and recall scores. Since the harmonic mean method diminishes the values of bias (of heavy weights), F-1 score is relatively immune to the imbalanced structure of data labels.

4.1. Applications on Sample Datasets

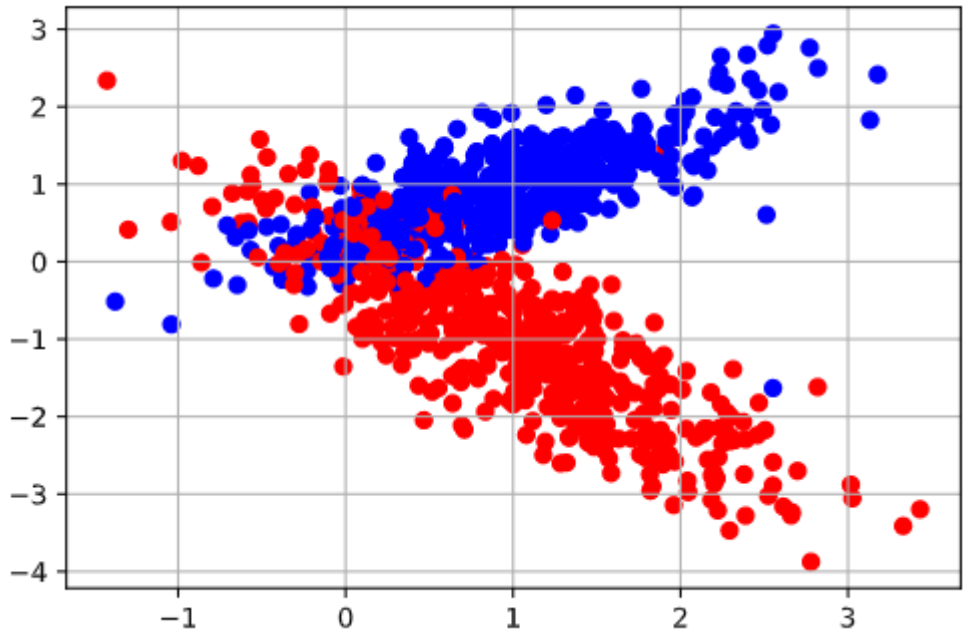
To illustrate effectively, here I generate samples with larger dataset:

#Generate Samples:

```
x, y = datasets.make_classification(
    n_samples=1000, n_features=2, n_informative=2, n_classes=2, n_redundant=0,
    n_clusters_per_class=1, random_state=1
)
np.c_[y, x]
```

#Visualize Scatterplot

```
plt.scatter(x[:,0], x[:,1], c=y, cmap='bwr')
plt.grid()
plt.show()
```




```
beta = np.array([0, 0, 0])
result = minimize(cross_entropy, beta)
result.x
```

Output:

```
array([ 1.0604636 , -1.03504223, -3.15872411])
```

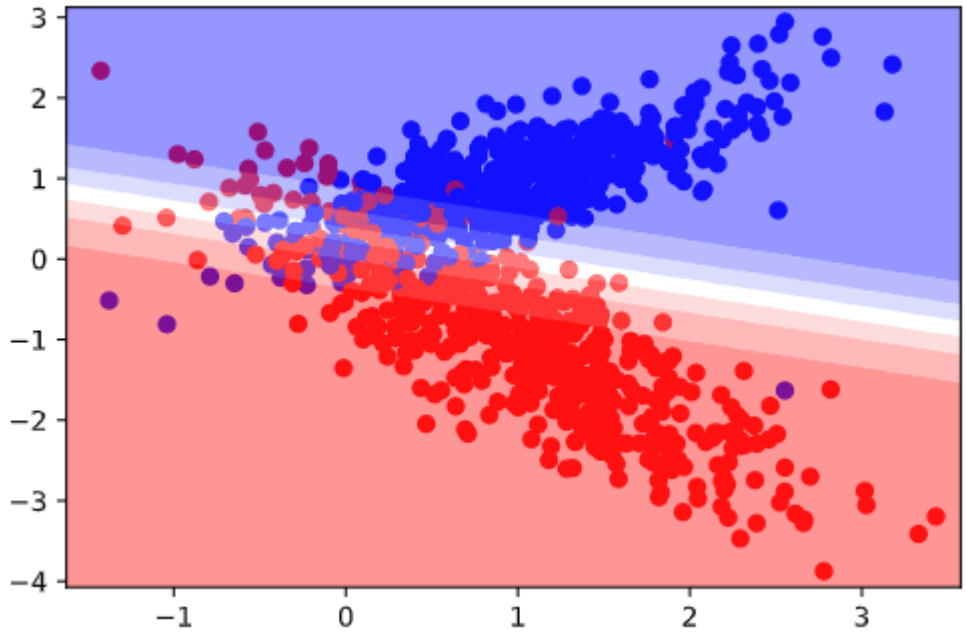
#Visualization of the Model

```
x1, x2 = x[:,0], x[:,1]
xy1, xy2 = np.meshgrid(
    np.arange(x1.min() - 0.2, x1.max() + 0.2, 0.1),
```

```
    np.arange(x2.min() - 0.2 , x2.max() + 0.2 , 0.1)
)

beta = result.x
xx = np.c_[xy1.flatten(), xy2.flatten()]
yhat = logit(np.dot(xx, beta[1:]) + beta[0] )

plt.scatter(x1, x2, c = y, cmap='bwr')
plt.contourf(xy1, xy2, yhat.reshape(xy1.shape), cmap='bwr', alpha=0.48)
plt.show()
```



#Define threshold for evaluation

For evaluation, it is necessary to first designate a certain threshold point to decide a certain level of value for Yhat as a threshold point to be assumed as the classified Y value (e.g., either 1 or 0) in a generated classification model;

```
def cut_evaluate(yhat, threshold):
  yhat = yhat.copy()
  yhat[yhat > threshold] = 1
  yhat[yhat < threshold] = 0

  return yhat.astype(int)
```

#Create Confusion Matrix

```
from sklearn.metrics import confusion_matrix
beta = result.x
yhat = logit( np.dot(x, beta[1:]) + beta[0] )
yhat = cut_evaluate(yhat, 0.5)
```

```
cmat = confusion_matrix(y, yhat)
cmat
```

output:

```
array([[450,  48],
       [ 50, 452]], dtype=int64)
```

#Accuracy

```
Accuracy = (cmat[0,0] + cmat[1,1]) / np.sum(cmat)
Accuracy
```

Output:

```
0.902
```

#Precision

```
precision = cmat[0,0] / (cmat[0,0] + cmat[0,1])
precision
```

Output:

```
0.9
```

#Recall

```
recall = cmat[0,0] / (cmat[0,0] + cmat[1,0])
recall
```

Output:

```
0.9036144578313253
```

#F-1 Score

```
F1score = 2 * ( (precision * recall)/(precision + recall)  )
F1score
```

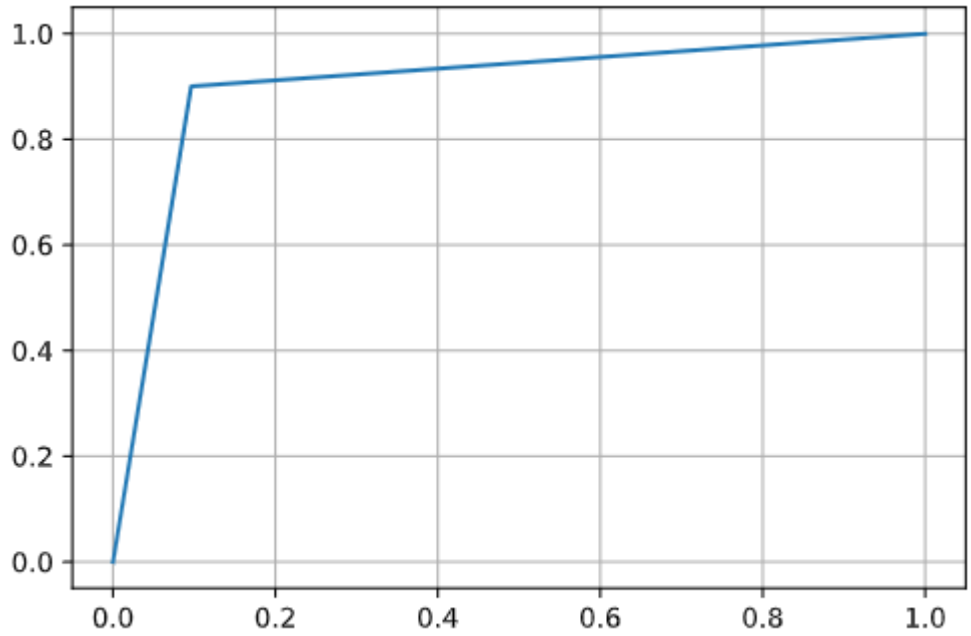
Output:

```
0.9018036072144289
```

#ROC Curve & ROC_AUC Score

```
from sklearn.metrics import roc_curve, roc_auc_score
fpr, tpr, _ = roc_curve(yhat, y)
plt.plot(fpr, tpr)
```

```
plt.grid()
plt.show()
```




```
roc_auc_score(y, yhat)
```

Output:

```
0.902
```

So far, all the processes are done manually, step-by-step, to illustrate the wholesome procedures together with mathematical backgrounds & model-structuring logic as regards to classification data analysis using generalized linear models. For applications, it is possible to simplify the whole process using python libraries, as depicted in the following paragraph. However, it is worth mentioning that understanding the mathematical background, modelling mechanism, and the whole logic/process of using logit function, as demonstrated so far, is essential for data scientists to create a powerful and customizable optimization model. Plus, after all, what would be the point of using modules and libraries without understanding the mechanism of them?

#Using logistic regression modules

```
from sklearn.linear_model import LogisticRegression
logitModel = LogisticRegression(penalty='none').fit(x, y)
print(logitModel.coef_)
print(logitModel.intercept_)
print(logitModel.score(x, y))
```

output:

```
[[-1.03499955 -3.15864365]]
```

```
[1.0604751]
```

```
0.902
```

#Estimation and confusion matrix generation

```
yhat = logitModel.predict(x)
confusion_matrix(y, yhat)
```

Output:

```
array([[450,  50],
       [ 48, 452]], dtype=int64)
```

#Evaluation

```
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score, precision
_score, recall_score
yhat = logitModel.predict(x)
acc = accuracy_score(y, yhat)
pre = precision_score(y, yhat)
rec = recall_score(y, yhat)
f1 = f1_score(y, yhat)
roc = roc_auc_score(y, yhat)
print(f'accuracy={acc}, precision={pre}, recall={rec}, f-
1 Score={f1}, ROC_AUC_Score={roc}')
```

Output:

```
accuracy=0.902,       precision=0.900398406374502,       recall=0.904,       f-1
Score=0.9021956087824352, ROC_AUC_Score=0.902
```