Machine Learning

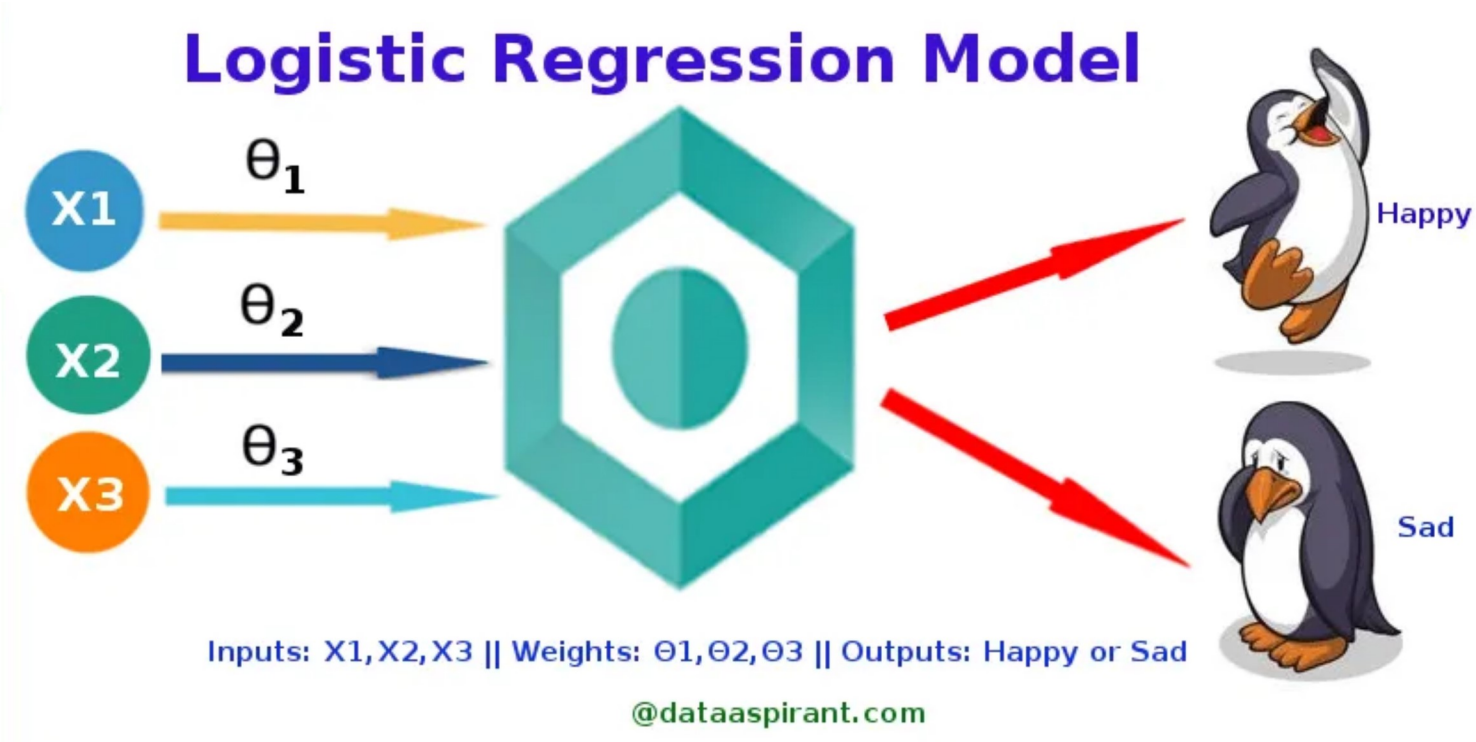
Lab08 - Logistic Regression

Objectives:

- Learn Logistic Regression

Version: 2025-02-05

This lab is by YP Wong yp@ypwong.net.



Source: https://dataaspirant.com/how-logistic-regression-model-works/

References:

- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- https://www.keboola.com/blog/logistic-regression-machine-learning
- https://medium.com/analytics-vidhya/a-comprehensive-guide-to-logistic-regression-e0cf04fe738c
- https://www.baeldung.com/cs/cross-validation-k-fold-loo
- https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff
- https://medium.com/@abhishekjainindore24/all-about-logistic-regression-bd135b6e3993

Refresh

Import Libraries

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
%matplotlib inline
```

Logistic Regression

Logistic regression is a supervised machine learning algorithm that is widely used for classification tasks. It predicts the probability that a given input belongs to one of two classes, often denoted as 0 and 1. The model outputs **probabilities**, which are values between 0 and 1, and uses the **logistic sigmoid function** to map these probabilities from a **linear combination of the input features**.

One could use the logistic regression model in the following scenarios:

- Build an email classifier to tell us whether an incoming email should be marked as “spam” or “not spam”.
- Check radiological images to predict whether a tumour is benign or malignant.
- Pour through historic bank records to predict whether a customer will default on their loan repayments or repay the loan.

The Logistic Sigmoid Function

The sigmoid function is a function that produces an s-shaped curve. It is a key mathematical component of the logistic regression model. It takes any real value as an argument and maps it to a range between 0 and 1.

The sigmoid function is defined as the following:

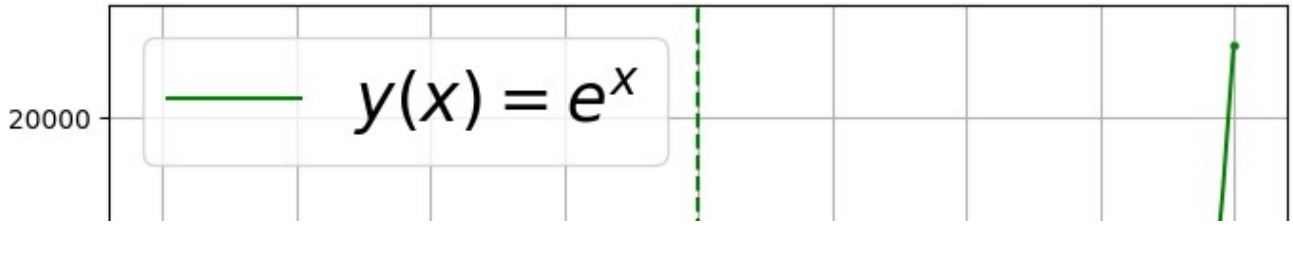
$$\sigma(z) = \frac{1}{1 + \frac{1}{e^z}} = \frac{1}{1 + e^{-z}}$$

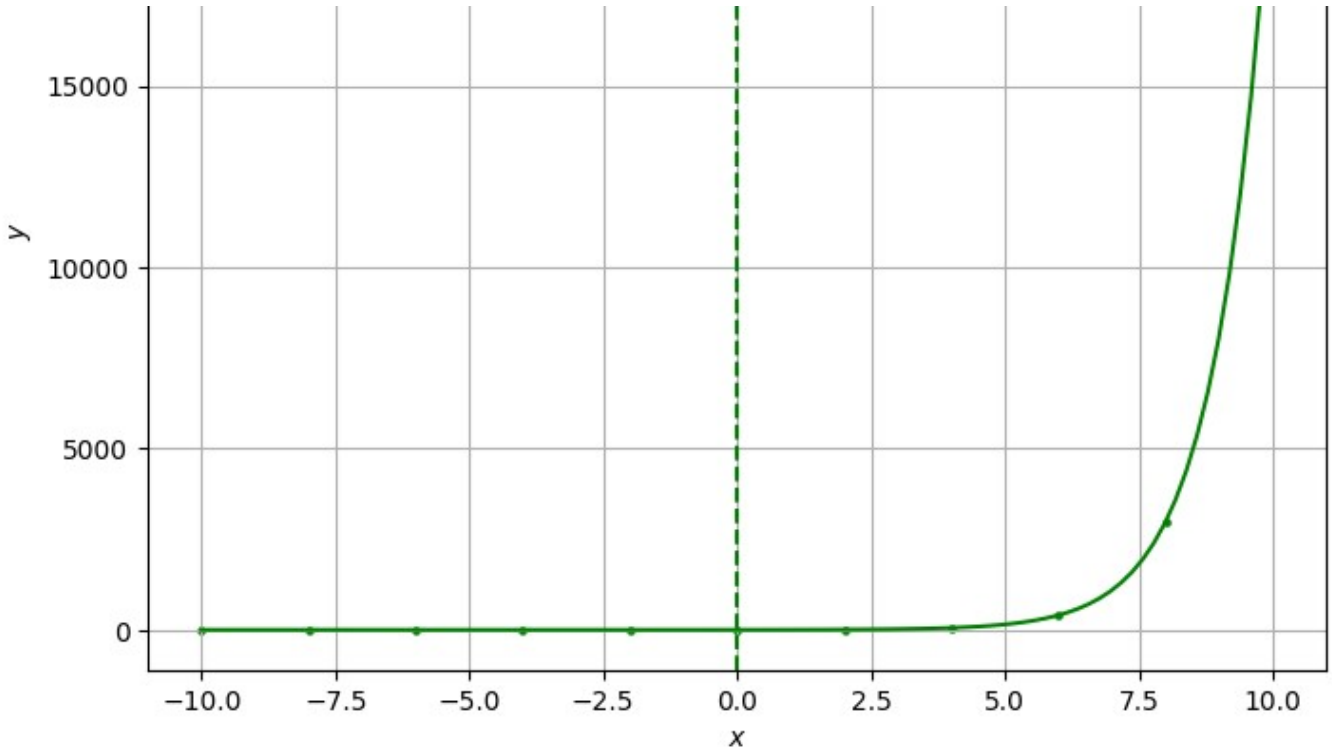
Where:

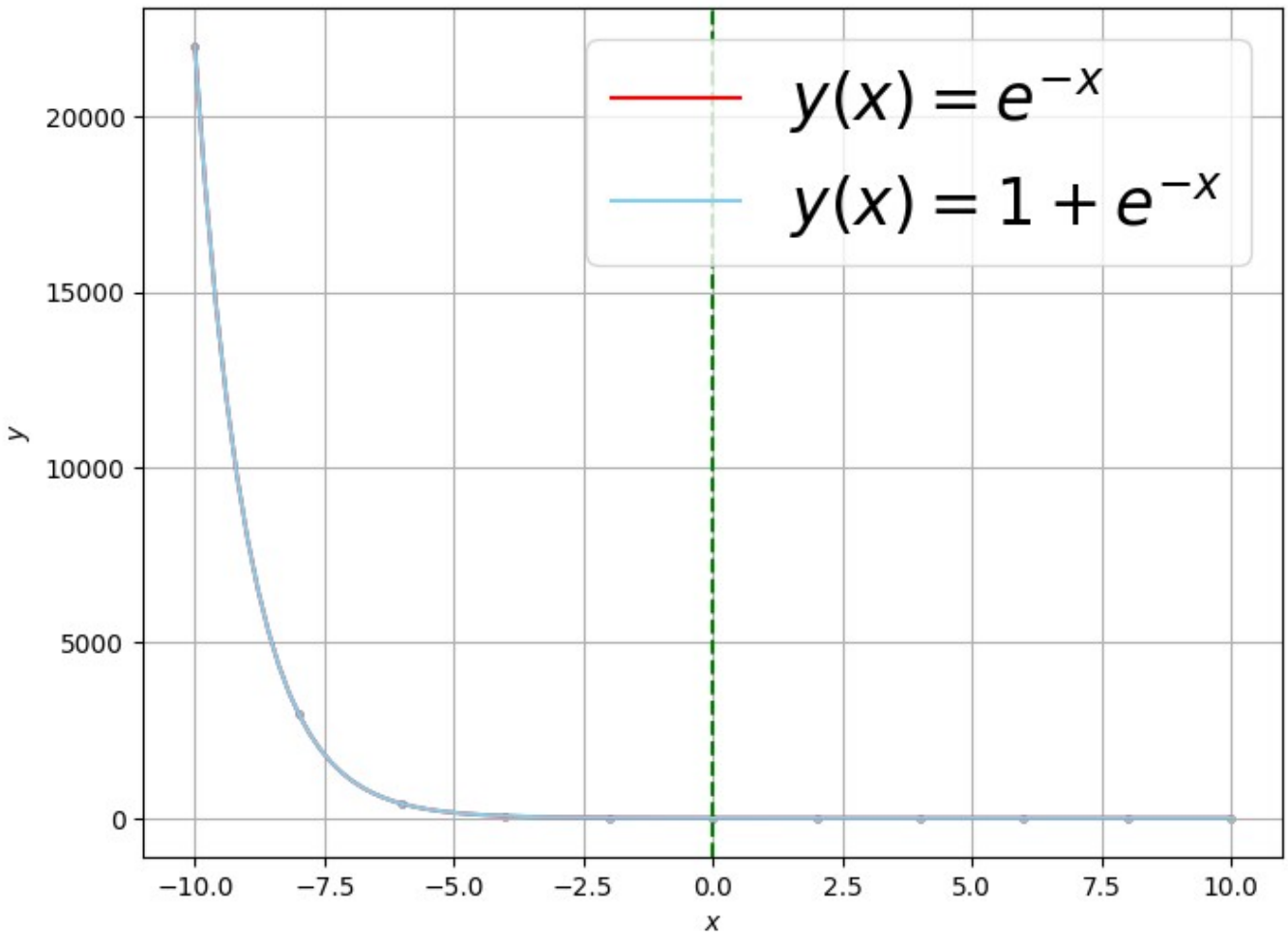
- $\sigma(z)$ is the output of the sigmoid function, which represents the probability of the positive class.
- z is the input to the sigmoid function, which is typically the result of a linear combination of the input features in logistic regression.

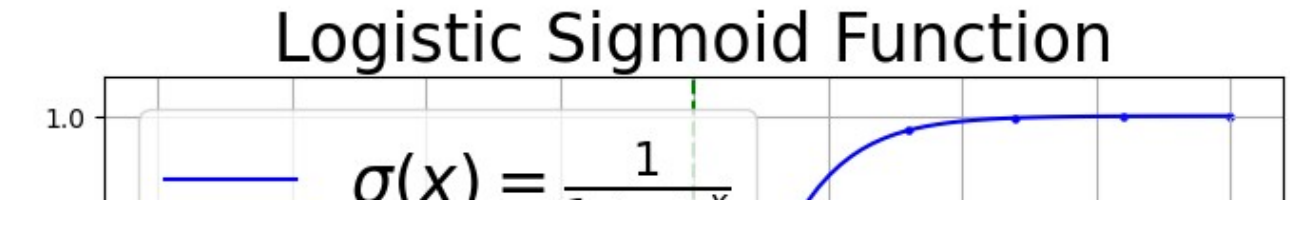
Visualization of Exponentials and the Sigmoid Function

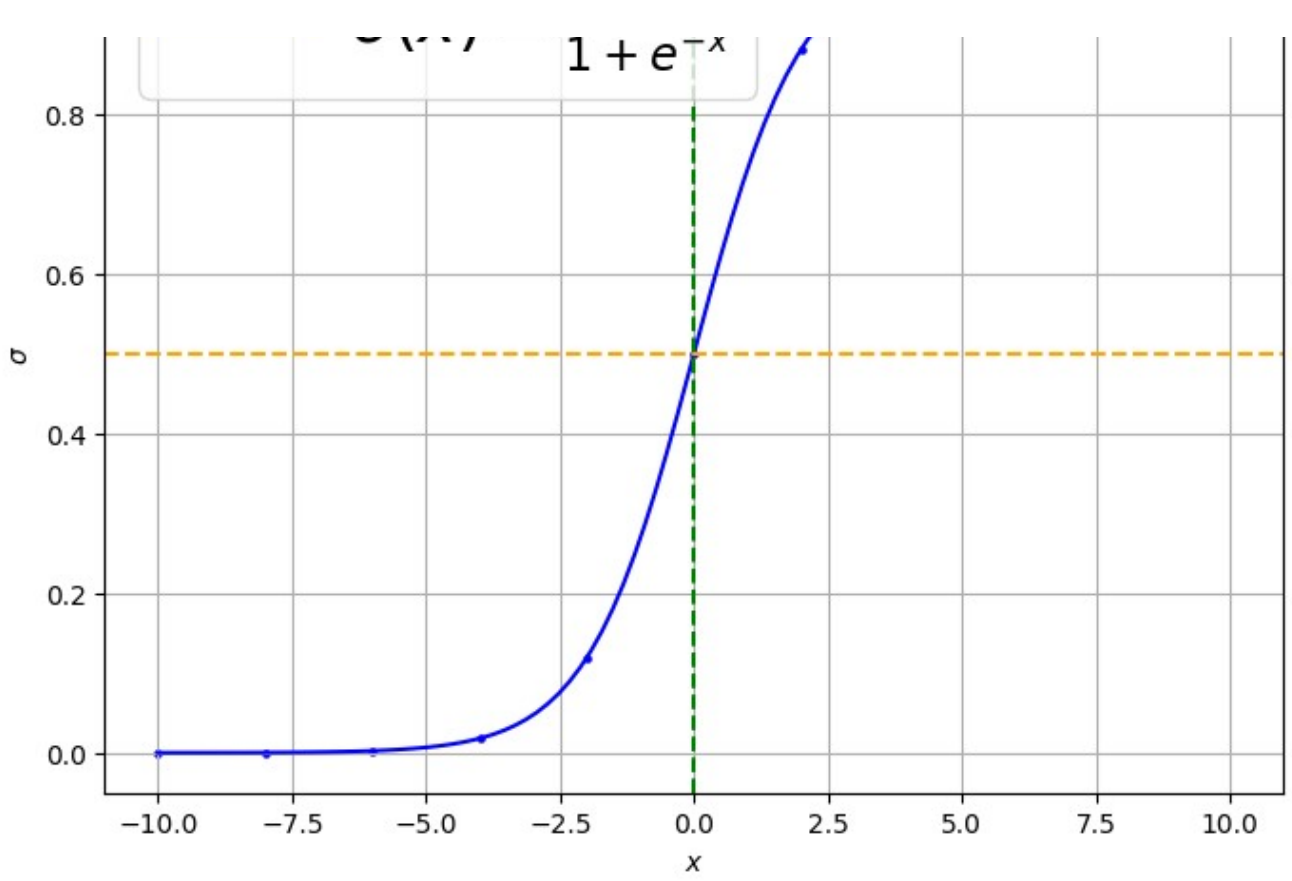
```
import matplotlib.pyplot as plt
%matplotlib inline

xlim_min, xlim_max = -10, 10

X     = np.linspace(xlim_min, xlim_max, 100)
X_pts = np.linspace(xlim_min, xlim_max, 11)

fig = plt.figure(figsize=(8, 6))
plt.plot   (X,     np.exp(X    ), color ="green", label = "$y(x) = e^{x}$")
plt.scatter(X_pts, np.exp(X_pts), color ="green", s = 5)
plt.axvline(x = 0, linestyle="dashed", color = "green")
plt.legend(loc = "best", fontsize = 25)
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.grid(True)
plt.show()
print()

fig = plt.figure(figsize=(8, 6))
plt.plot   (X,     np.exp(-X    ), color ="red", label = "$y(x) = e^{-x}$")
plt.scatter(X_pts, np.exp(-X_pts), color ="red", s = 5)
plt.plot    (X,    1 + np.exp(-X    ), color ="skyblue", label = "$y(x) = 1 + e^{-x}$")
plt.scatter(X_pts, 1 + np.exp(-X_pts), color ="skyblue", s = 5)
plt.axvline(x = 0, linestyle="dashed", color = "green")
plt.legend(loc = "best", fontsize = 25)
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.grid(True)
plt.show()
print()

fig = plt.figure(figsize=(8, 6))
plt.plot    (X,    1 / (1 + np.exp(-X    )), color ="blue", label = "$\sigma(x) = \\frac{1}{1 + e^{-x}}$
plt.scatter(X_pts, 1 / (1 + np.exp(-X_pts)), color ="blue", s = 5)
plt.axvline(x = 0, linestyle="dashed", color = "green")
plt.axhline(y = 0.5, linestyle="dashed", color = "orange")
plt.legend(loc = "best", fontsize = 25)
plt.xlabel("$x$")
plt.ylabel("$\sigma$")
plt.title("Logistic Sigmoid Function", fontsize = 25)
plt.grid(True)
plt.show()
print()

for x in X_pts:
  print("sigmoid(%7.3f) = %.7f" % (x, 1 / (1 + np.exp(-x))))
```




15000
10000
5000
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x


$y(x) = e^{-x}$
$y(x) = 1 + e^{-x}$
20000
15000
10000
5000
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x


Logistic Sigmoid Function
$\sigma(x) = \frac{1}{}$
1.0




```
sigmoid(-10.000) = 0.0000454
sigmoid( -8.000) = 0.0003354
sigmoid( -6.000) = 0.0024726
sigmoid( -4.000) = 0.0179862
sigmoid( -2.000) = 0.1192029
sigmoid(  0.000) = 0.5000000
sigmoid(  2.000) = 0.8807971
sigmoid(  4.000) = 0.9820138
sigmoid(  6.000) = 0.9975274
sigmoid(  8.000) = 0.9996646
sigmoid( 10.000) = 0.9999546
```

The Logistic Regression Model

The logistic regression model uses the logistic sigmoid function to predict probabilities. It combines a linear model (similar to linear regression) with the sigmoid function to make predictions about the probability that an observation belongs to the positive class.

The **Logistic Regression Model**, which is the linear combination of the input feature variables can be represented as such:

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Where $x_1, x_2, \ldots, x_n$ are the input feature variables, and $\theta_0, \theta_1, \theta_2, \ldots, \theta_n$ are the model coefficients (parameters).

Then, with the output of the model, z, we may calculate the probability of belonging to the positive class by using the logistic sigmoid function. The logistic sigmoid function maps the linear combination z to a value

between 0 and 1, representing a probability:

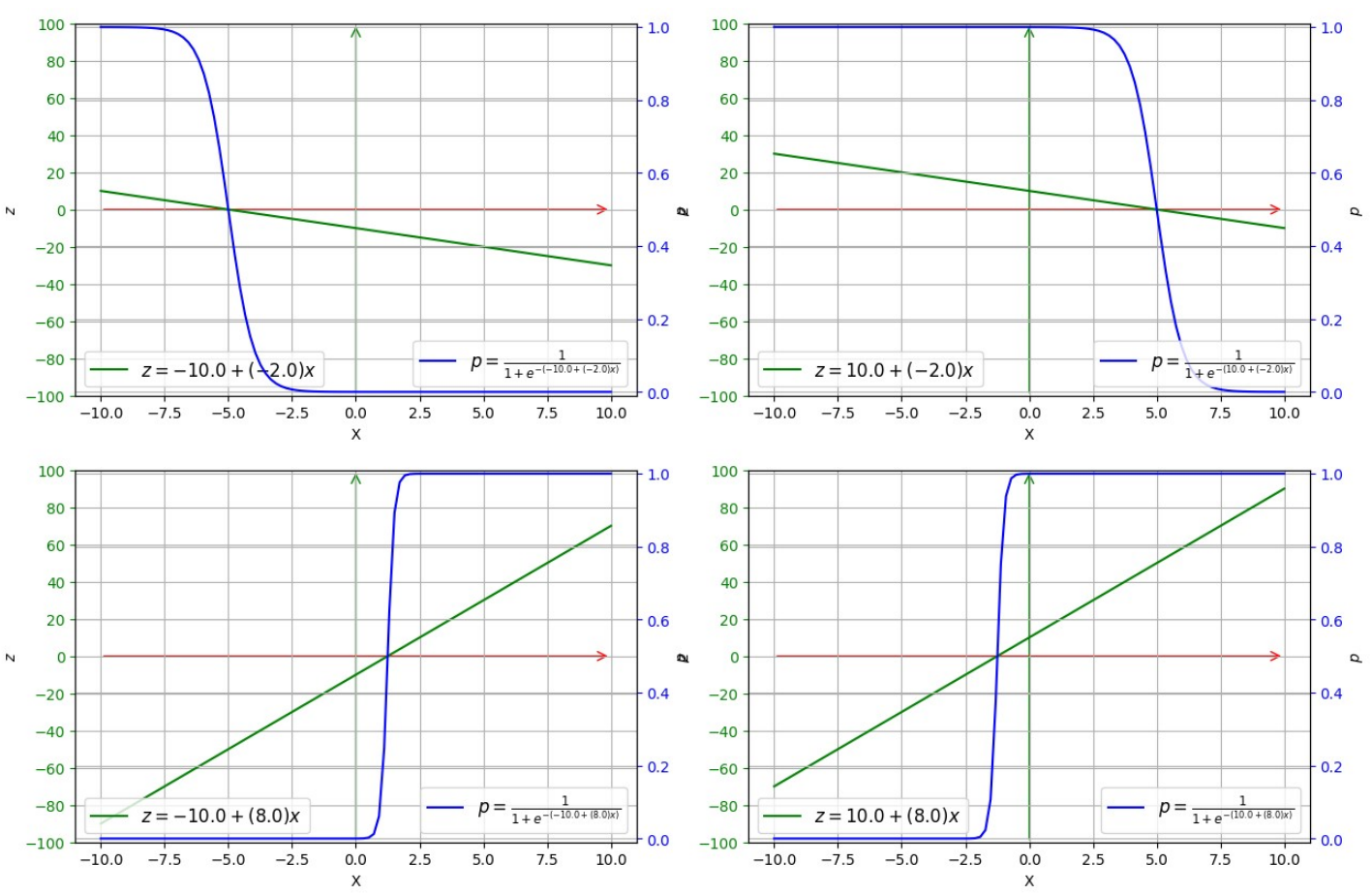
$$P(\text{Positive Class}) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n)}}$$

This output is interpreted as the probability that the instance belongs to the positive class in **binary classification problems**.

Thus, the **Logistic Regression Model** predicts the probability $p = \frac{1}{1+e^{-(\theta_0+\theta_1 x_1+\theta_2 x_2+\ldots+\theta_n x_n)}}$ of an instance belonging to the positive class.

```
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.patches import FancyArrowPatch

xlim_min, xlim_max = -10, 10
X = np.linspace(xlim_min, xlim_max, 100)

zlim_min, zlim_max = -100, 100

# coef0 = np.array( [ [-10, 0, 10],
#                     [-10, 0, 10],
#                     [-10, 0, 10] ] )
# coef1 = np.array( [ [-2, -2, -2],
#                     [ 2,  2,  2],
#                     [ 8,  8,  8] ] )

coef0 = np.array( [ [-10, 10],
                    [-10, 10] ] )
coef1 = np.array( [ [-2, -2],
                    [ 8,  8] ] )

fig, axs = plt.subplots(2, 2, figsize=(15, 10))

for i, axs_row in enumerate(axs):
  for j, ax1 in enumerate(axs_row):

    Z = coef0[i, j] + coef1[i, j] * X

    ax1.plot(X, Z, color = "green", label = "$z = %.1f + (%.1f) x$" % (coef0[i, j], coef1[i, j]) )

    ax2 = ax1.twinx()
    ax2.plot(X, 1 / (1 + np.exp(-Z)), color = "blue", label = "$p = \\frac{1}{1 + e^{-(%.1f + (%.1f) x

    arrow_x = FancyArrowPatch((xlim_min, 0), (xlim_max, 0), color='red',
                              arrowstyle='->', mutation_scale=15)
    arrow_z = FancyArrowPatch((0, zlim_min), (0, zlim_max), color='green',
                              arrowstyle='->', mutation_scale=15)
    ax1.add_patch(arrow_x)
    ax1.add_patch(arrow_z)

    ax1.set_xlabel('X')
    ax1.set_ylabel('$z$')
    ax2.set_ylabel('$p$')

    ax1.set_ylim(zlim_min, zlim_max)
```

```
    ax2.set_ylim(-0.01, 1.01)

    ax1.tick_params('y', colors = "green")
    ax2.tick_params('y', colors = "blue")
    ax1.locator_params(axis = 'x', nbins = 10)
    ax1.locator_params(axis = 'y', nbins = 10)
    ax2.locator_params(axis = 'y', nbins = 10)
    ax1.grid(True)
    ax2.grid(True)

    ax1.legend(fontsize = 12, loc = "lower left")
    ax2.legend(fontsize = 12, loc = "lower right")

plt.show()
```



The Log-Odds (Logit) Function, Logistic Regression Function & Decision Boundary

To recap, in the context of a simple model with a single input feature x_1, the **Logistic Regression Model** is represented as:

$$z = \theta_0 + \theta_1 x_1$$

The probability of belonging to the positive class is given by the logistic sigmoid function:

$$P(\text{Positive Class}) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1)}}$$

Here, the logistic sigmoid function maps the linear combination $z = \theta_0 + \theta_1 x_1$ to a value between 0 and 1, interpreted as the probability that an input instance belongs to the positive class with label $y = 1$.

Thus, the **Logistic Regression Model** predicts the probability that the label y is 1 given input x_1:

$$p(y = 1|x_1) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1)}}$$

The Log-Odds (Logit) Function

In probability theory, the **odds** of an event happening are defined as the ratio of the probability of the event occurring to the probability of it not occurring, thus if p is the probability of success, then:

$$odds = \frac{p}{1 - p}$$

In logistic regression, we first calculate the log-odds (also called the logit), which is a measure of how likely an observation belongs to the positive class. This log-odds is the natural logarithm of the odds of the event occurring.

$$\text{log-odds}(p) = \text{logit}(p) = log\left(\frac{p}{1 - p}\right)$$

Where:

- p is the probability that the observation belongs to the positive class (i.e., $p = P(\text{Positive Class})$).
- The odds is the ratio of the probability of the event happening to the probability of it not happening, i.e., $\frac{p}{1-p}$.

Further observation reveals:

$$P(\text{Positive Class}) = p(y = 1|x_1) = p = \frac{1}{1 + e^{-z}}$$

$$\frac{1}{p} - 1 = e^{-z}$$

$$\frac{1 - p}{p} = \frac{1}{e^z}$$

$$\frac{p}{} = e^z$$

$$\frac{}{1-p} = e$$

$$log\left(\frac{p}{1-p}\right) = log(e^z)$$

$$log\left(\frac{p}{1-p}\right) = z \cdot log(e) = z \cdot 1 = z$$

$$log\left(\frac{p}{1-p}\right) = \theta_0 + \theta_1 x_1$$

since $z = \theta_0 + \theta_1 x_1$.

Thus, the log-odds is also modeled as a linear combination of the input features $x_1, x_2, \ldots, x_n$, and θ_0, θ_1, and their corresponding coefficients $\theta_2, \ldots, \theta_n$:

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

This expression represents the log-odds of the positive class before we transform it into a probability using the logistic (sigmoid) function. The value of zz can range from −∞−∞ to +∞+∞, and it's this value that determines the probability of classifying an observation as belonging to the positive class.

In summary, the **log-odds**, or **logit**, is the **natural logarithm of the odds** as defined by:

$$\text{log-odds}(p) = \text{logit}(p) = log\left(\frac{p}{1-p}\right) = z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \ldots + \theta_n x_n$$

The Decision Boundary

In binary classification problems, an input instance is classified as the **positive** class (usually labeled as $y = 1$) if the predicted probability of the positive class is greater than or equal to 0.5 ($p(y = 1|x_1) \geq 0.5$). This can be expressed as:

$$y = \begin{cases} 0 \text{ (negative)} & \text{if } \frac{1}{1+e^{-(\theta_0+\theta_1 x_1)}} < 0.5, \\ 1 \text{ (positive)} & \text{if } \frac{1}{1+e^{-(\theta_0+\theta_1 x_1)}} \geq 0.5 \end{cases}$$

This can also be written using an **indicator function**:

$$y = 1\left(\frac{1}{1+e^{-(\theta_0+\theta_1 x_1)}} \geq 0.5\right)$$

The **decision boundary** is the point where the model is equally likely to predict either class — that is, when the probability of the positive class (label $y = 1$) is exactly 0.5 ($P(\text{Positive Class}) = 0.5$). This occurs when the logistic function satisfies the following:

$$\frac{1}{1+e^{-z}} = 0.5$$

Thus, to find the decision boundary, we solve the equation for z:

$$\frac{1}{1+e^{-z}} = \frac{1}{2}$$

$$1 + e^{-z} = 2$$

$$e^{-z} = 1$$

$$-z = 0$$

$$z = 0$$

$$\theta_0 + \theta_1 x_1 = 0$$

The **decision boundary**, where the probability is equal to 0.5 is thus the line:

$$z = \theta_0 + \theta_1 x_1 = 0 \quad \text{when} \quad p(y = 1|x_1) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1)}} = 0.5$$

Summary

The **decision boundary** is identified at the points where:

$$p = 1 - p = \frac{1}{2} \rightarrow \frac{p}{1-p} = 1 \rightarrow \log\left(\frac{p}{1-p}\right) = 0$$

$$\rightarrow \text{logit}(p) = \log\left(\frac{p}{1-p}\right) = z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = 0$$

$$\rightarrow P(\text{Positive Class}) = p(y = 1|x_1) = p = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n)}} = \frac{1}{2}$$

```
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.ticker import AutoLocator

def plot_one_feature_logistic_regression_results(
                  X, y, model,
                  plot_data_points = True,
                  plot_predicted_points = True,
                  plot_predicted_probability_points = False,
                  plot_predicted_curve = False,
                  plot_logistic_regression_curve = False,
                  plot_logit_line = False,
                  X_new = None,
                  xlim = "auto",
                  ylim = (-0.1, 1.1),
                  figure_size = (8, 6),
                  title_font_size = 20
                 ):

  plt.figure(figsize=figure_size)

  if xlim == "auto":
    xlim_min, xlim_max = np.min(X[:, 0]), np.max(X[:, 0])
    dx = (xlim_max - xlim_min) * 0.1
    xlim_min, xlim_max = xlim_min - dx, xlim_max + dx
  else:
    xlim_min, xlim_max = xlim
  plt.xlim(xlim_min, xlim_max)
  plt.locator_params(axis = 'x', nbins = 20)
```

```
plt.locator_params(axis = 'y', nbins = 20)

x_values = np.linspace(xlim_min, xlim_max, 100).reshape(-1, 1)

ylim_min, ylim_max = ylim
plt.ylim(ylim_min, ylim_max)
plt.xlabel('X')
plt.ylabel('y')

if plot_data_points:
  plt.scatter(X, y, color = "blue", s = 50, marker = "o",
              label = "Data Points")

if model != None:

  if plot_predicted_curve:
    y_values = model.predict(x_values)
    plt.plot(x_values, y_values, color = "green",
             label = "Predicted Curve")

  if plot_logit_line or plot_logistic_regression_curve:
    logit_y_values = model.intercept_ + model.coef_ * x_values

    if plot_logit_line:
      plt.plot(x_values, logit_y_values, color = "orange",
               linestyle='dashdot', label = "Logit Line")

    if plot_logistic_regression_curve:
      # p_values = 1 / (1 + np.exp(-logit_y_values))   # This works too
      p_values = model.predict_proba(x_values)[:, 1]
      plt.plot(x_values, p_values, color = "orange",
               linestyle='dotted', label = "Logistic Sigmoid Curve")

    plt.axhline(y = 0.5, color = "orange",
                linestyle='dashed')

  if plot_predicted_points:
    y_pred = model.predict(X)
    plt.scatter(X, y_pred, color = "green", alpha=0.5, s = 70, marker = "*",
                # edgecolors = 'lightblue', linewidths = 5,
                label = "Predicted Points")

  if plot_predicted_probability_points:
    p_pred = model.predict_proba(X)[:, 1]
    plt.scatter(X, p_pred, color = "orange", alpha=0.5, s = 30, marker = "o",
                label = "Predicted Probability")

  if X_new is not None:
    y_pred_new = model.predict(X_new)
    plt.scatter(X_new, y_pred_new, color = "cyan", alpha=0.9, s = 150, marker = "*",
                label = "New Predicted Points")

  # plt.title("$p(y = 1 | X) = \\frac{1}{1 + e^{-(%.3f + (%.3f) X )}}$" % (model.intercept_, model.c
  #           fontsize = title_font_size)
  title_text = "Logit: $z = %.3f + (%.3f) X$\nSigmoid: $p(y = 1 | X) = \\frac{1}{1 + e^{-z}}$" % (mo
  plt.title(title_text, fontsize = title_font_size)
```

```
  plt.legend()
  plt.grid(True)

  # plt.show()

def plot_predicted_results(X, y, model, X_new = None):
  plot_one_feature_logistic_regression_results(
                X, y, model,
                plot_data_points = True,
                plot_predicted_points = True,
                plot_predicted_probability_points = True,
                plot_predicted_curve = True,
                plot_logistic_regression_curve = True,
                plot_logit_line = True,
                X_new = X_new
              )

def plot_points(X, y):
  plot_one_feature_logistic_regression_results( X, y, model = None, plot_data_points = True)
```

```
import seaborn as sns

def plot_confusion_matrix(confusion_matrix,
                          figure_size = (3, 3),
                          title_font_size = 15,
                          font_size = 12):

  num_classes = confusion_matrix.shape[0]
  fig, ax = plt.subplots(figsize = figure_size)

  # class_names=[0,1] # name of classes
  # tick_marks = np.arange(len(class_names))
  class_names = np.arange(num_classes) # name of classes
  tick_marks  = np.arange(num_classes)

  plt.xticks(tick_marks, class_names)
  plt.yticks(tick_marks, class_names)

  # create heatmap
  sns.heatmap(pd.DataFrame(confusion_matrix),
              annot = True, cmap = "YlGnBu", fmt = 'g')

  ax.xaxis.set_label_position("top")
  plt.tight_layout()

  plt.title('Confusion matrix', y = 1.1, fontsize = title_font_size)
  plt.ylabel('Actual Labels', fontsize = font_size)
  plt.xlabel('Predicted Labels', fontsize = font_size)
```

Types of Logistic Regression

1. Binary Logistic Regression

- **Description:** Predicts the probability of an observation belonging to one of two classes (0 or 1).
- **Example:** Predicting whether an email is spam (1) or not (0).

2. Multiclass/Multinomial Logistic Regression

- **Description:** Extends logistic regression to handle more than two classes.
- **Also Known As:** Multinomial Logistic Regression.
- **Example:** Predicting the genre of a movie (Action, Drama, Comedy).

3. Single-Predictor/Univariate Logistic Regression

- **Description:** Involves one independent variable to predict binary outcomes.
- **Example:** Predicting student pass/fail based on hours of study.

4. Multiple/Multivariate Logistic Regression

- **Description:** Uses multiple independent variables for binary classification.
- **Example:** Predicting loan approval based on income, credit score, and age.

Key Takeaways

- **Binary vs. Multiclass:** Consider the number of classes in the dependent variable.
- **Single vs. Multiple Predictors:** Examine the number of independent variables used for prediction.

Single-Predictor Binary Logistic Regression

Single-Predictor Logistic Regression refers to logistic regression with a single **independent variable**, sometimes also known as:

- One-Predictor Binary Logistic Regression
- Single-Variable Binary Logistic Regression
- One-Variable Binary Logistic Regression
- Univariate Binary Logistic Regression

Binary Logistic Regression refers to logistic regression with an outcome variable (**dependent variable**) being binary, meaning it can take one of two values (usually coded as 0 and 1).

Creating and Fitting Single-Predictor Binary Logistic Regression

Consider a $m \times 1$ input data matrix $\mathbf{X}$ for our independent variables (also known as features or predictors) and $\mathbf{y}$ for our dependent variable containing the label values:

$$\mathbf{X} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(m)} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Here, m represents the data size (number of points = number of observations), and the data dimension

Here, m represents the data size (number of points = number of observations), and the data dimension (number of features) is 1.

Each element $y^{(i)}$ in $\mathbf{y}$ typically contains the integer value of 1 to represent the **positive class**, and 0 otherwise.

The **Single-Predictor Binary Logistic Regression** aims to fit the input data ($\mathbf{X}$) to obtain the **best-fit** model parameters θ_0 and θ_1, which are are learned through the training process (usually by minimizing the *cost function*), leading to:

$$z^{(i)} = \theta_0 + \theta_1 x^{(i)} \quad \text{for } i = 1, 2, \ldots, m.$$

$$p(y^{(i)} = 1 | x^{(i)}) = \frac{1}{1 + e^{-(z^{(i)})}} \quad \text{for } i = 1, 2, \ldots, m.$$

$$p(y^{(i)} = 1 | x^{(i)}) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x^{(i)})}} \quad \text{for } i = 1, 2, \ldots, m.$$

It is customary to classify an input instance as the positive class (usually labeled as $y = 1$) if $p(y^{(i)} = 1 | x^{(i)}) \geq 0.5$. Consequently, the predicted label $\hat{y}^{(i)}$ is determined as follows:

$$\hat{y}^{(i)} = \begin{cases} 0 & \text{if } \frac{1}{1+e^{-(\theta_0 + \theta_1 x^{(i)})}} < 0.5, \\ 1 & \text{if } \frac{1}{1+e^{-(\theta_0 + \theta_1 x^{(i)})}} \geq 0.5 \end{cases} \quad \text{for } i = 1, 2, \ldots, m.$$

The above expression is commonly denoted using an **indicator function**

$$\hat{y}^{(i)} = 1\left(\frac{1}{1 + e^{-(\theta_0 + \theta_1 x^{(i)})}} \geq 0.5\right) \quad \text{for } i = 1, 2, \ldots, m.$$

The predicted label vector is represented as:

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix}$$

The logistic regression model aims to find the optimal values for the pair of parameters θ_0 and θ_1 that minimize the **cost function** and provide the **best fit** for the data. This function measures the difference between the model's predictions and the actual labels in the training data. The cost function for logistic regression is based on the log-likelihood of the observed labels given the model's predictions.

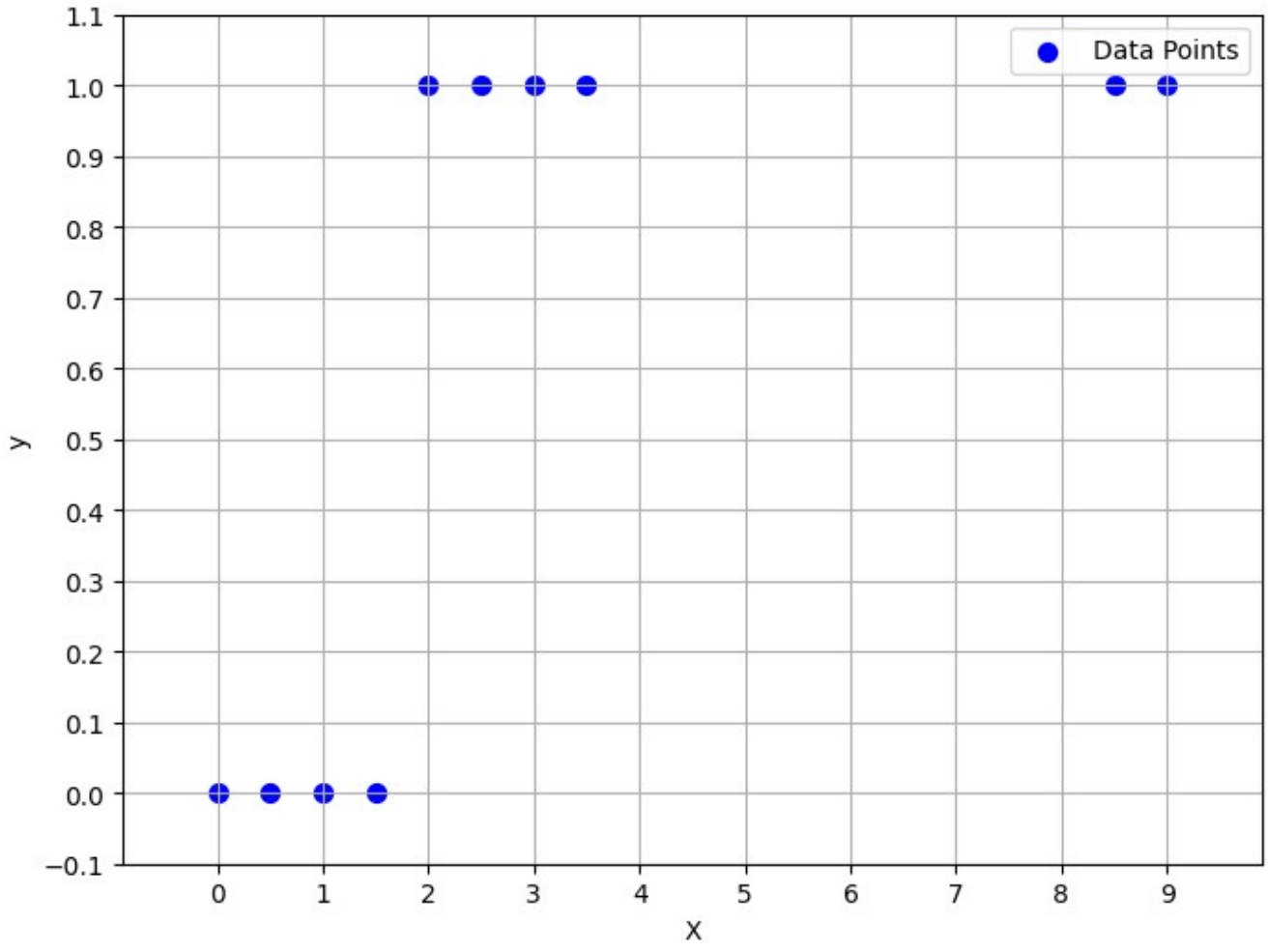
Single-Predictor Binary Logistic Regression With `scikit-learn`: Example 1

Creating the dataset:

```
X = np.array([0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 8.5, 9]).reshape((-1, 1))
y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
```

```
print(" X", X.shape, ":\n", X)
print("\n y", y.shape, ":\n", y)

plot_points(X, y)
```

```
 X (10, 1) :
[[0. ]
[0.5]
[1. ]
[1.5]
[2. ]
[2.5]
[3. ]
[3.5]
[8.5]
[9. ]]

y (10,) :
[0 0 0 0 1 1 1 1 1 1]
```



Linear Regression Not Suitable Tool

Why?

- The target variable y takes binary values (0 or 1), which is more common in classification tasks. Using linear regression which is designed for continuous outcomes would lead to outcomes that are not constrained between 0 and 1

constrained between 0 and 1.

- The data shown below has a non-linear relationship, which isn't ideal for logistic regression that has an s-shaped curve.

Performing simple linear regression with the dataset:

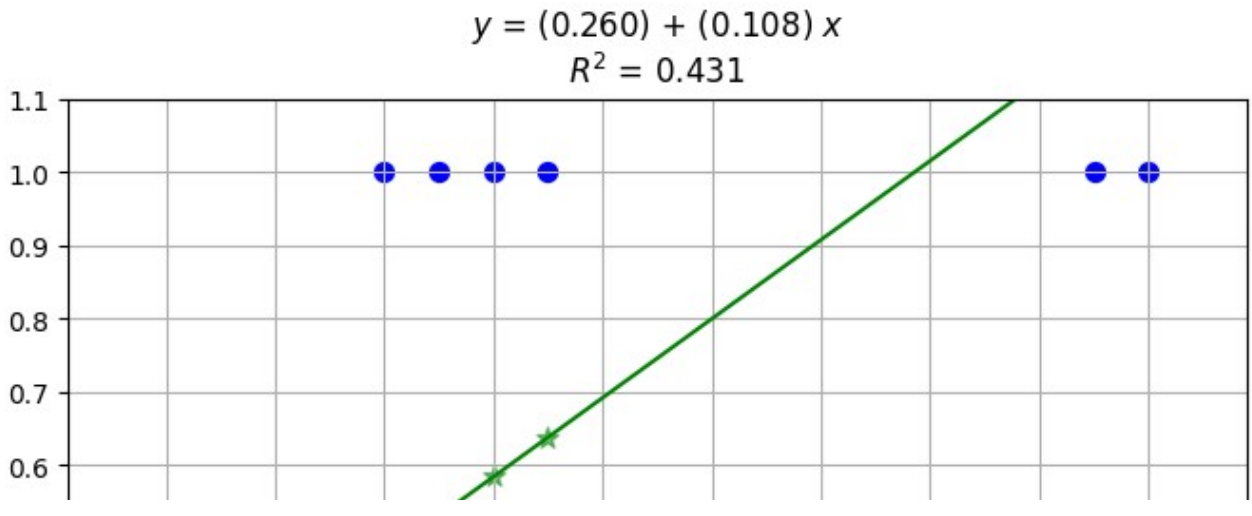
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()

model.fit(X, y)

r_sq = model.score(X, y)

print(f"{'coefficient of determination':30}= {r_sq}")
print(f"{'intercept':30}= {model.intercept_}")
print(f"{'slope':30}= {model.coef_}\n")

y_pred = model.predict(X)
print(f"predicted response:\n{y_pred}")
```

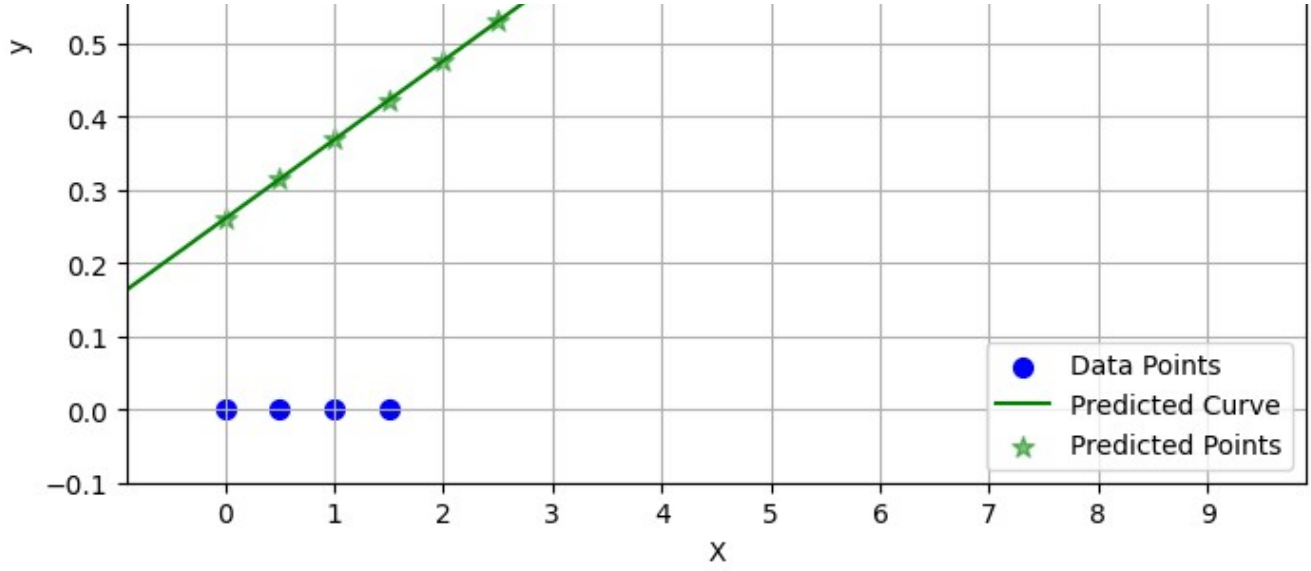
```
coefficient of determination  = 0.4313395113732098
intercept                     = 0.2603201347935972
slope                         = [0.10783488]

predicted response:
[0.26032013 0.31423757 0.36815501 0.42207245 0.47598989 0.52990733
 0.58382477 0.63774221 1.1769166  1.23083404]
```

Plotting the linear regression model, you can see that it is not a suitable model for the target variables:

```
plot_one_feature_logistic_regression_results(X, y, model = model, plot_predicted_points = True,
                                             plot_predicted_curve = True)
coef_value = model.coef_[0]
plt.title("$y$ = (%0.3f) + (%0.3f) $x$\n$R^2$ = %0.3f" % (model.intercept_, coef_value, r_sq))
```

```
<ipython-input-4-e08b0e75c97a>:84: DeprecationWarning: Conversion of an array with ndim > 0 to a s
  title_text = "Logit: $z = %.3f + (%.3f) X$\nSigmoid: $p(y = 1 | X) = \\frac{1}{1 + e^{-z}}$" % (
Text(0.5, 1.0, '$y$ = (0.260) + (0.108) $x$\n$R^2$ = 0.431')
```





Create and Fitting the Logistic Regression Model Using `scikit-learn`

Initializing and fitting the binary logistic regression model:

```
# Logistic Regression model
# (liblinear is a common solver used for binary classification for small to medium datasets)
model = LogisticRegression(solver = 'liblinear', random_state = 0)

# Fitting the model
model.fit(X, y)
```

```
LogisticRegression
LogisticRegression(random_state=0, solver='liblinear')
```

Evaluating the model accuracy with accuracy metric:

```
mean_accuracy = model.score(X, y)
print(f"{'mean_accuracy':23} = {mean_accuracy}")

y_pred = model.predict(X)
mean_accuracy_eq = np.sum(y == y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")
```

```
mean_accuracy           = 0.9
mean_accuracy (manual)  = 0.9
y                       = [0 0 0 0 1 1 1 1 1 1]
y_pred                  = [0 0 0 1 1 1 1 1 1 1]
```

Display parameters of the model and verifying the results of the dataset:

Display parameters of the model and verifying the results of the dataset:

```
# Output model parameters
print(f"{'intercept_':16} = {model.intercept_}")
print(f"{'coefficient_':16} = {model.coef_}")

# Output model classes
# print(f"\n{'model classes':16} = ", model.classes_)
print(f"\nModel Classes: ", model.classes_)

# Predicted probabilities
p_pred = model.predict_proba(X)
print(f"\nPredicted probabilities from the model:\n{p_pred}")

# Manually compute probabilities using equation
logit_y_values = model.intercept_ + model.coef_ * X
p_pred_eq = 1 / (1 + np.exp(-logit_y_values))
p_pred_eq = np.append(1 - p_pred_eq, p_pred_eq, axis = 1)
print(f"\nManually computed probabilities:\n{p_pred_eq}")

# Checking that results from both methods are very close, up to 0.0000001 margin of error
print("\nChecking that results from both methods are very close:\n", np.abs(p_pred - p_pred_eq) < 0.00
```

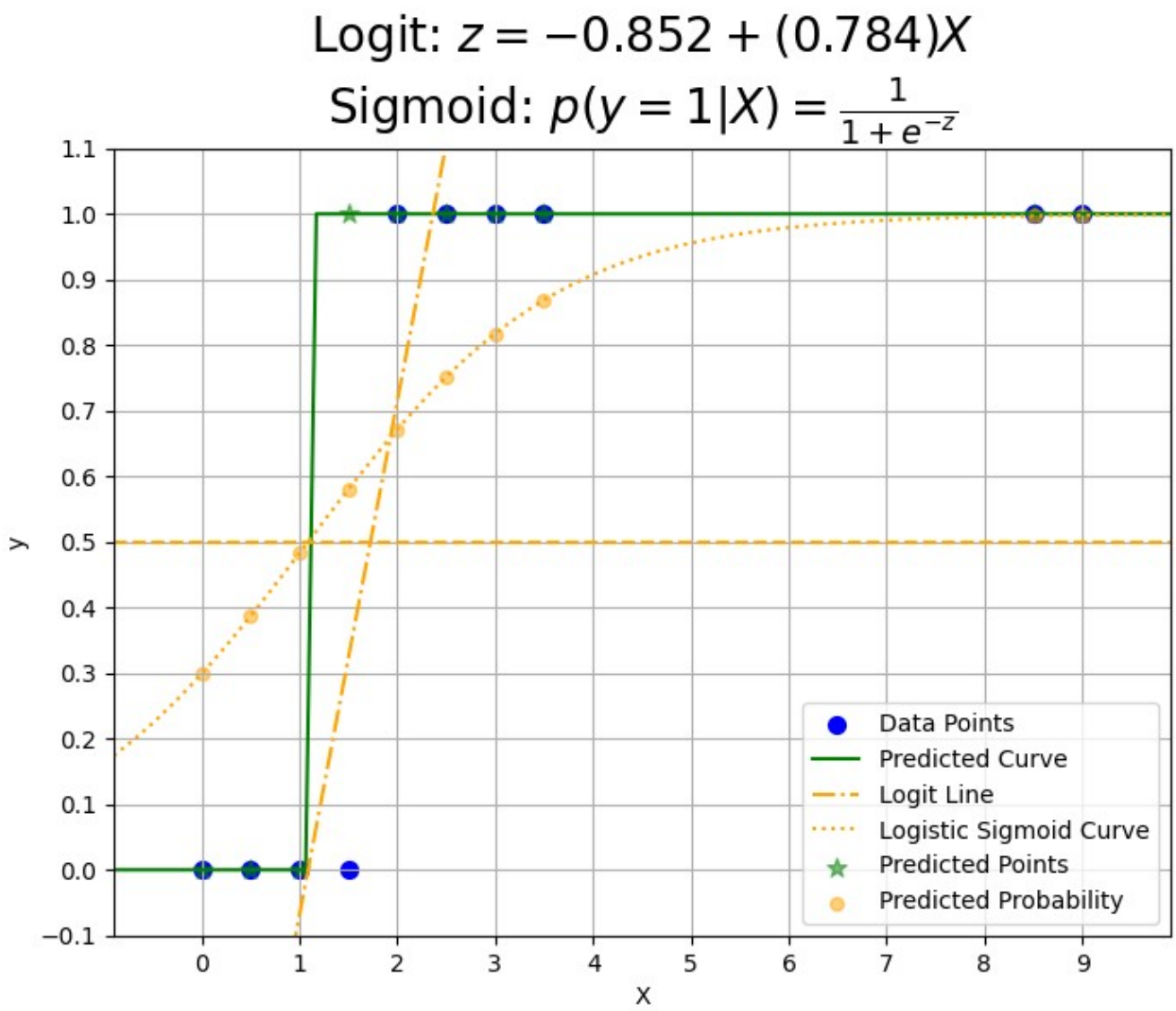
```
intercept_       = [-0.85243488]
coefficient_     = [[0.78352294]]

Model Classes:  [0 1]

Predicted probabilities from the model:
[[0.70107767 0.29892233]
 [0.61317392 0.38682608]
 [0.51722117 0.48277883]
 [0.41998145 0.58001855]
 [0.32858078 0.67141922]
 [0.24854779 0.75145221]
 [0.18270401 0.81729599]
 [0.1312564  0.8687436 ]
 [0.0029959  0.9970041 ]
 [0.00202679 0.99797321]]

Manually computed probabilities:
[[0.70107767 0.29892233]
 [0.61317392 0.38682608]
 [0.51722117 0.48277883]
 [0.41998145 0.58001855]
 [0.32858078 0.67141922]
 [0.24854779 0.75145221]
 [0.18270401 0.81729599]
 [0.1312564  0.8687436 ]
 [0.0029959  0.9970041 ]
 [0.00202679 0.99797321]]

Checking that results from both methods are very close:
 [[ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
```

```
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]]
```

Plotting the result of the model:

```
plot_predicted_results(X, y, model = model)
```

```
<ipython-input-4-e08b0e75c97a>:84: DeprecationWarning: Conversion of an array with ndim > 0 to a s
  title_text = "Logit: $z = %.3f + (%.3f) X$\nSigmoid: $p(y = 1 | X) = \\frac{1}{1 + e^{-z}}$" % (
```



Confusion Matrix, Precision, Recall & F_1 - Score

Here we evaluate the performance of the model based on true labels (y) and predicted labels (y_pred):

Confusion Matrix:

$tp = \text{True Positive} = \text{Predicted Positive, Actual Positive}$

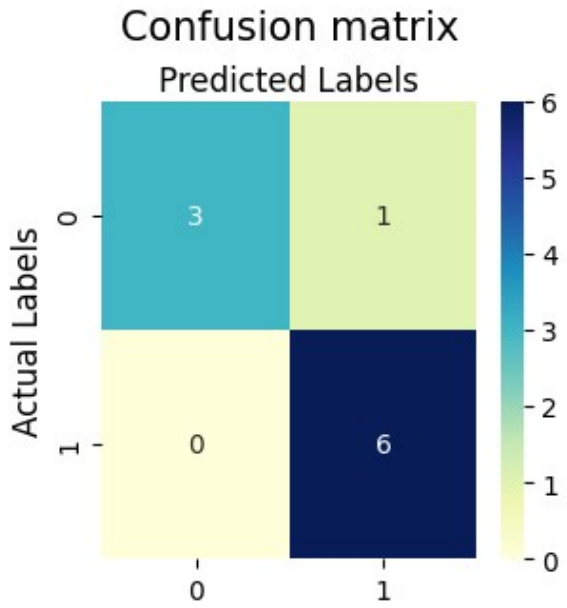
$tn = \text{True Negative} = \text{Predicted Negative, Actual Negative}$

$fp = \text{False Positive} = \text{Predicted Positive, Actual Negative}$

$fn = \text{False Negative} = \text{Predicted Negative, Actual Positive}$

```
c_matrix = confusion_matrix(y, y_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
[[3 1]
 [0 6]]
```



Precision, Recall & F_1 - Score:

```
def my_precision_recall_fscore_support(confusion_matrix):
  num_class = confusion_matrix.shape[0]
  support = np.sum(confusion_matrix, axis = 1)
  pred = np.sum(confusion_matrix, axis = 0)
  tp = np.diag(c_matrix)
  fp = pred - tp
  fn = support - tp
  precision = tp / (tp + fp)
  recall = tp / (tp + fn)
  f1_score = 2 * precision * recall / (precision + recall)

  return precision, recall, f1_score, support

def my_classification_report(y, y_pred, target_names = None):
  c_matrix = confusion_matrix(y, y_pred)
  num_class = c_matrix.shape[0]
  precision, recall, f1_score, support = my_precision_recall_fscore_support(c_matrix)

  if target_names == None:
    spaces_string = " " * 14
  else:
    max_length = len(max(target_names, key=len))
    spaces_string = " " * (max_length + 2)
```

```
    spaces_string = " " * (max_length - 2)

  str = "%sprecision    recall  f1-score   support\n\n" % spaces_string

  for i in range(num_class):
    if target_names == None:
      str += "%12d  " % i
    else:
      str += f"%{max_length}s  " % target_names[i]

    str += "%9.2f%10.2f%10.2f%10d\n" % (precision[i], recall[i], f1_score[i], support[i])
  return str
```

```
report = classification_report(y, y_pred)
print("sklearn classification report:\n\n", report)
```

```
sklearn classification report:

                  precision    recall  f1-score   support

               0       1.00      0.75      0.86         4
               1       0.86      1.00      0.92         6

        accuracy                           0.90        10
       macro avg       0.93      0.88      0.89        10
    weighted avg       0.91      0.90      0.90        10
```

```
report = my_classification_report(y, y_pred)
print("custom classification report:\n\n", report)
```

```
custom classification report:

                  precision    recall  f1-score   support

               0       1.00      0.75      0.86         4
               1       0.86      1.00      0.92         6
```

Precision

- Precision is the ratio of correctly predicted positive observations to the total predicted positives.
- It measures the accuracy of the positive predictions made by the model.
- A high precision indicates that when the model predicts a positive class, it is likely to be correct.
- **Precision is more critical where false positives are costly, example in spam email detection.**
- Formula: $precision = \frac{tp}{tp+fp}$

Recall (Sometimes it is called **Sensitivity**)

- Recall is the ratio of correctly predicted positive observations to the total actual positives.
- It measures the ability of the model to capture all the positive instances in the dataset.
- A high recall indicates that the model is good at finding all the positive instances in the dataset.
- **Recall is more critical where false negatives are costly, example in fraud detection**
- Formula: $recall = \frac{tp}{tp+fn}$

$tp + fn$

F1-score

- The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall, considering both false positives and false negatives.
- It ranges from 0 to 1, where:
  - 1 indicates perfect precision and recall,
  - 0 indicates poor performance.
- It is particularly useful when there is an imbalance between the classes in the dataset. It penalizes models that favor one metric over the other.
- Formula: $F_1 = \frac{2 \text{ x } precision \text{ x } recall}{precision + recall}$

Support

- Support represents the number of actual occurrences of each class in the specified dataset.
- It is the count of true instances for each class.
- It helps in understanding the distribution of classes in the dataset.
- Example: If your dataset has 1000 instances, and 200 are of class 1, the support for class 1 is 200.

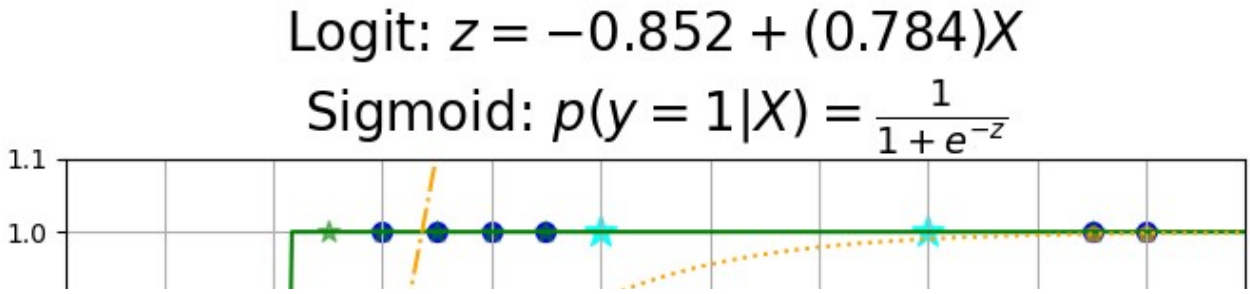
Reference: https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics

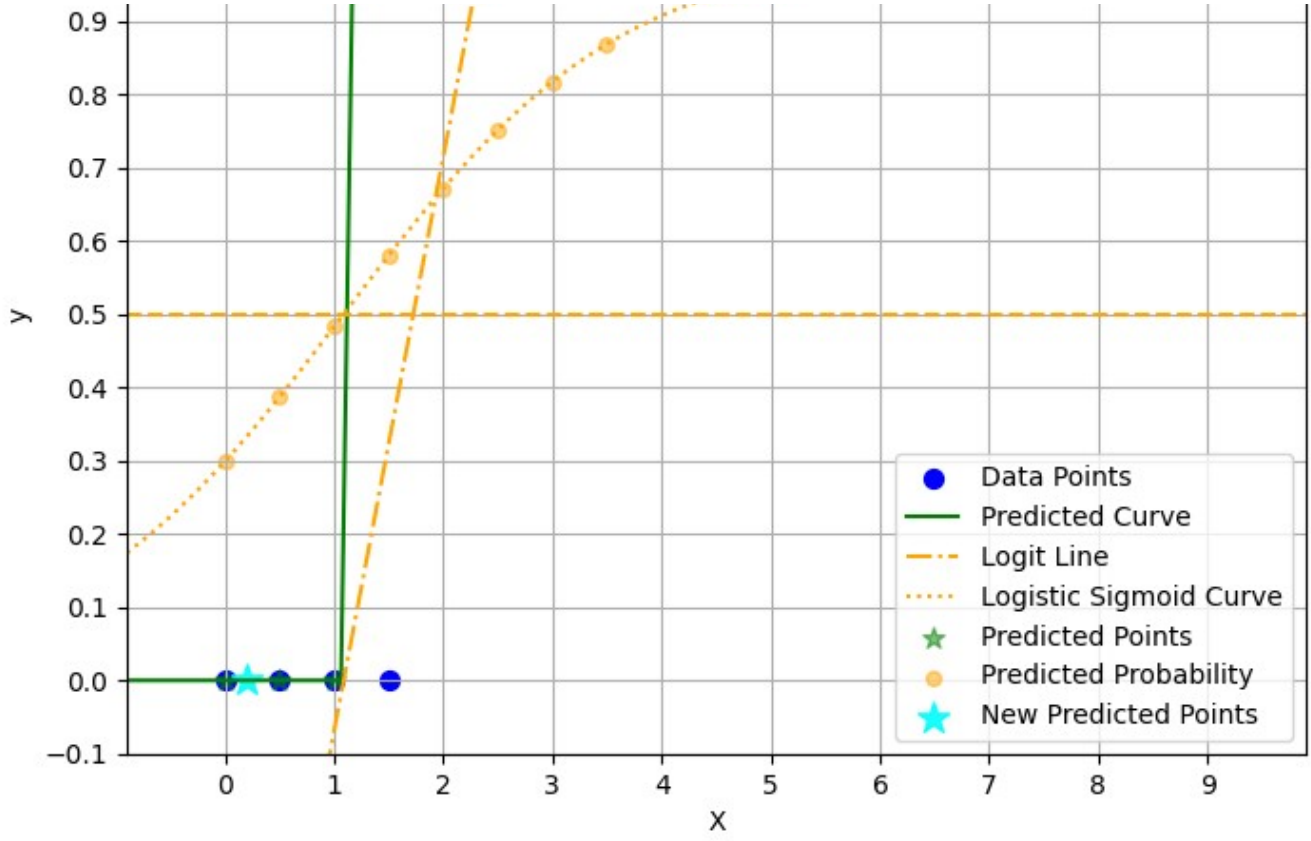
Predict New Data Points

Create new data points, predict the results and display:

```
X_new = np.array([0.2, 4, 7]).reshape((-1, 1))
print("X_new =\n", X_new)

y_pred_new = model.predict(X_new)
print("\ny_pred_new =\n", y_pred_new)

plot_predicted_results(X, y, model = model, X_new = X_new)
```

```
X_new =
 [[0.2]
 [4. ]
 [7. ]]

y_pred_new =
 [0 1 1]
<ipython-input-4-e08b0e75c97a>:84: DeprecationWarning: Conversion of an array with ndim > 0 to a s
  title_text = "Logit: $z = %.3f + (%.3f) X$\nSigmoid: $p(y = 1 | X) = \\frac{1}{1 + e^{-z}}$" % (
```





Improve the Model

One of the ways to improve the model is by **tuning the hyperparameters**. A hyperparameter is parameter that is set before the learning process begins. One of the hyperparameters in logistic regression is the **regularization strength**, known as the C value. Regularization is a technique used to prevent **overfitting** by adding a penalty term to the model's loss function. This helps the model avoid becoming too sensitive to small fluctuations or noise in the training data.

Overfitting and Underfitting:

- **Overfitting** is when the model learns too much on the data and "fits" it too closely, performing well on the training set, but less so on unseen test data.
- **Underfitting** means the failure of the model to learn from the training data properly—it ignores random noise but is too simple to capture the structure in the data.

To choose the best C value, we may use a method such as the **K-Fold Cross-Validation** method. By splitting the data into several subsets (i.e. folds), the model can be trained on one subset of the data and validated on the remaining data. This way, the model's performance for different values of C can be evaluated and we can choose the one that performs best.

```
from sklearn.model_selection import LeaveOneOut, GridSearchCV

# Define the parameter grid for C
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}

# Create the logistic regression model
```

```
model_gs = LogisticRegression(solver='liblinear')

# Set up grid search cross-validation
grid_search = GridSearchCV(model_gs, param_grid, cv=4)

# Fit the model and find the best C
grid_search.fit(X, y)

# Print the best C and the corresponding score
print(f"Best C: {grid_search.best_params_['C']}")
print(f"Best cross-validation score: {grid_search.best_score_}")
```

```
Best C: 10
Best cross-validation score: 0.7916666666666666
```

Our dataset only consists of 10 data points, with only 2 data points for each fold, it might not provide a reliable evaluation of the model's performance because:

- With such small subsets, some folds might not adequately represent the full data distribution.
- There is a higher risk of
- If the classes are imbalanced, some folds could lack samples of a given class, making it impossible to train the model properly on that fold.

Conversely, **Leave-One-Out Cross-Validation (LOO CV)** is more appropriate for a smaller dataset, as it uses more training samples in each iteration. Since we have 10 points, this creates 10 folds (1 data point in 1 fold) and each data point will be used as a validation point exactly once.

```
# Define the parameter grid for C
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}

# Create the logistic regression model
model_gs = LogisticRegression(solver='liblinear')

# Set up Leave-One-Out Cross-Validation
loo = LeaveOneOut()

# Set up grid search cross-validation with LOO CV
grid_search = GridSearchCV(model_gs, param_grid, cv=loo)

# Fit the model and find the best C
grid_search.fit(X, y)

# Print the best C and the corresponding score
print(f"Best C = {grid_search.best_params_['C']}")
print(f"Best cross-validation score = {grid_search.best_score_}")
```

```
Best C = 10
Best cross-validation score = 0.9
```

From the results of the cross-validation methods above, we:

set the regularization strength (or C value) equal to 10.0, instead of the default value of 1.0.

This larger C value has allowed the model to fit the data more closely, providing a higher accuracy result.

- Higher values of C correspond to less regularization, which leads to a model that fits the training data more closely.
- In this case, we increased the value of C to weaken regularization, making it fit the data more closely.

Initializing and fitting the binary logistic regression model:

```
# Logistic Regression model
model_reg = LogisticRegression(solver = 'liblinear', C = 10.0, random_state = 0)

# Fitting the model
model_reg.fit(X, y)
```

```
LogisticRegression(C=10.0, random_state=0, solver='liblinear')
```

Evaluating the model accuracy with accuracy metric:

```
# Evaluating model accuracy
mean_accuracy = model_reg.score(X, y)
print(f"{'mean_accuracy':23} = {mean_accuracy}")

y_pred = model_reg.predict(X)
mean_accuracy_eq = np.sum(y == y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")
```

```
mean_accuracy           = 1.0
mean_accuracy (manual)  = 1.0
y                       = [0 0 0 0 1 1 1 1 1 1]
y_pred                  = [0 0 0 0 1 1 1 1 1 1]
```

Display parameters of the model and verifying the results of the dataset:

```
# Output model parameters
print(f"{'intercept_':16} = {model_reg.intercept_}")
print(f"{'coefficient_':16} = {model_reg.coef_}")

# Output model classes
print(f"\nModel Classes: ", model_reg.classes_)

# Predicted probabilities
p_pred = model_reg.predict_proba(X)
print(f"\nPredicted probabilities from the model:\n{p_pred}")

# Manually compute probabilities using equation
```

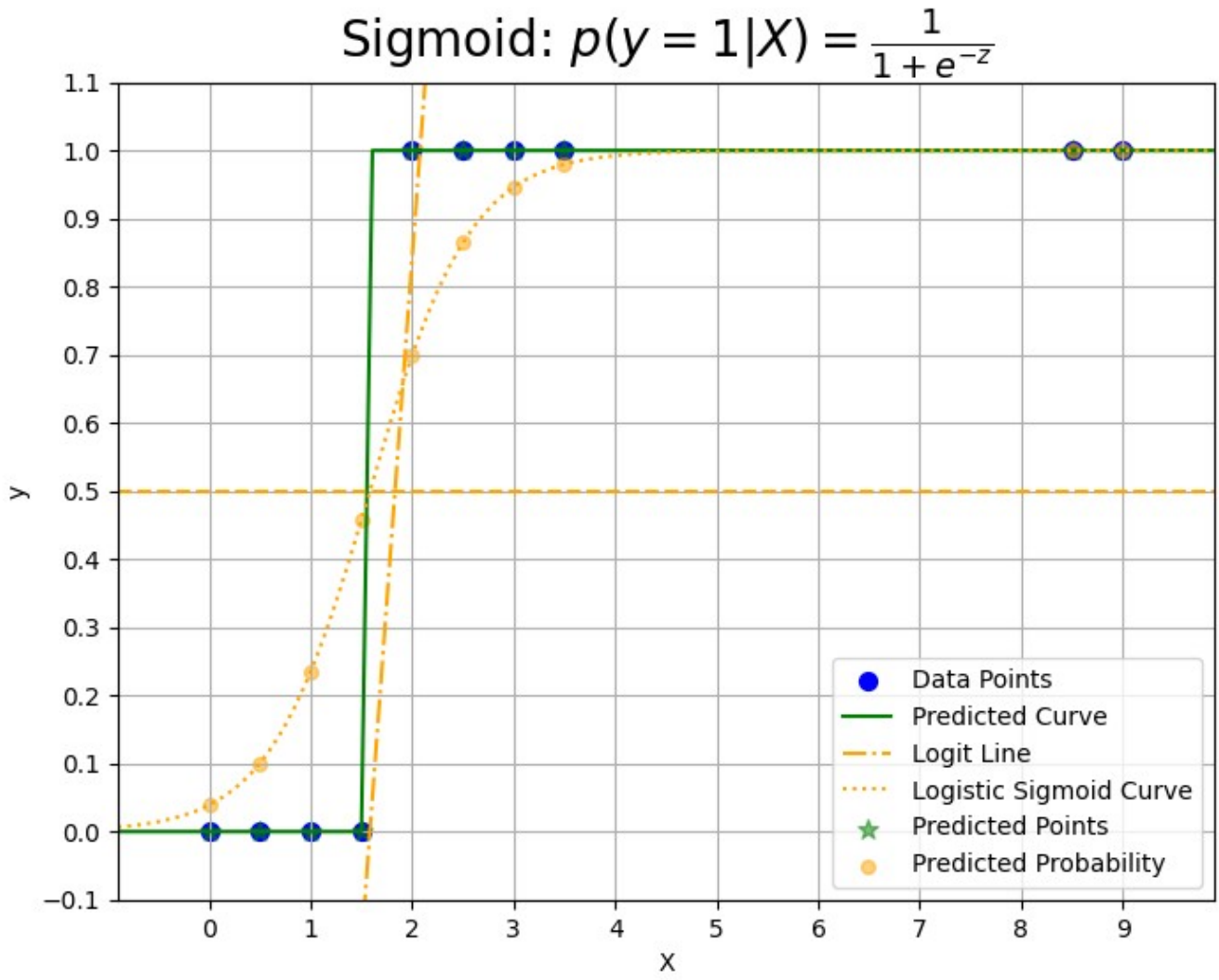
```
logit_y_values = model_reg.intercept_ + model_reg.coef_ * X
p_pred_eq = 1 / (1 + np.exp(-logit_y_values))
p_pred_eq = np.append(1 - p_pred_eq, p_pred_eq, axis = 1)
print(f"\nManually computed probabilities:\n{p_pred_eq}")

# Checking that results from both methods are very close, up to 0.0000001 margin of error
print("\nChecking that results from both methods are very close:\n", np.abs(p_pred - p_pred_eq) < 0.00
```

```
    intercept_       = [-3.20608655]
    coefficient_     = [[2.02382846]]

    Model Classes:  [0 1]

    Predicted probabilities from the model:
    [[9.61062683e-01 3.89373166e-02]
     [8.99724961e-01 1.00275039e-01]
     [7.65353572e-01 2.34646428e-01]
     [5.42483287e-01 4.57516713e-01]
     [3.01204150e-01 6.98795850e-01]
     [1.35464285e-01 8.64535715e-01]
     [5.38907693e-02 9.46109231e-01]
     [2.02863309e-02 9.79713669e-01]
     [8.34480749e-07 9.99999166e-01]
     [3.03352646e-07 9.99999697e-01]]

    Manually computed probabilities:
    [[9.61062683e-01 3.89373166e-02]
     [8.99724961e-01 1.00275039e-01]
     [7.65353572e-01 2.34646428e-01]
     [5.42483287e-01 4.57516713e-01]
     [3.01204150e-01 6.98795850e-01]
     [1.35464285e-01 8.64535715e-01]
     [5.38907693e-02 9.46109231e-01]
     [2.02863309e-02 9.79713669e-01]
     [8.34480749e-07 9.99999166e-01]
     [3.03352646e-07 9.99999697e-01]]

    Checking that results from both methods are very close:
     [[ True  True]
     [ True  True]
     [ True  True]
     [ True  True]
     [ True  True]
     [ True  True]
     [ True  True]
     [ True  True]
     [ True  True]
     [ True  True]]
```

Plotting the result of the model:

```
plot_predicted_results(X, y, model = model_reg)
```

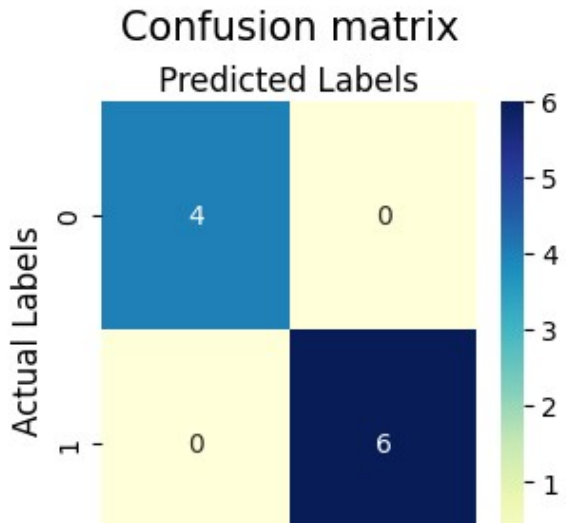
```
    <ipython-input-4-e08b0e75c97a>:84: DeprecationWarning: Conversion of an array with ndim > 0 to a s
      title_text = "Logit: $z = %.3f + (%.3f) X$\nSigmoid: $p(y = 1 | X) = \\frac{1}{1 + e^{-z}}$" % (
```



Confusion Matrix:

```
c_matrix = confusion_matrix(y, y_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
[[4 0]
 [0 6]]
```



Precision, Recall & F_1 - Score:

```
report = classification_report(y, y_pred)
print(report)
```

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00         4
               1       1.00      1.00      1.00         6

        accuracy                           1.00        10
       macro avg       1.00      1.00      1.00        10
    weighted avg       1.00      1.00      1.00        10
```

```
report = my_classification_report(y, y_pred)
print(report)
```

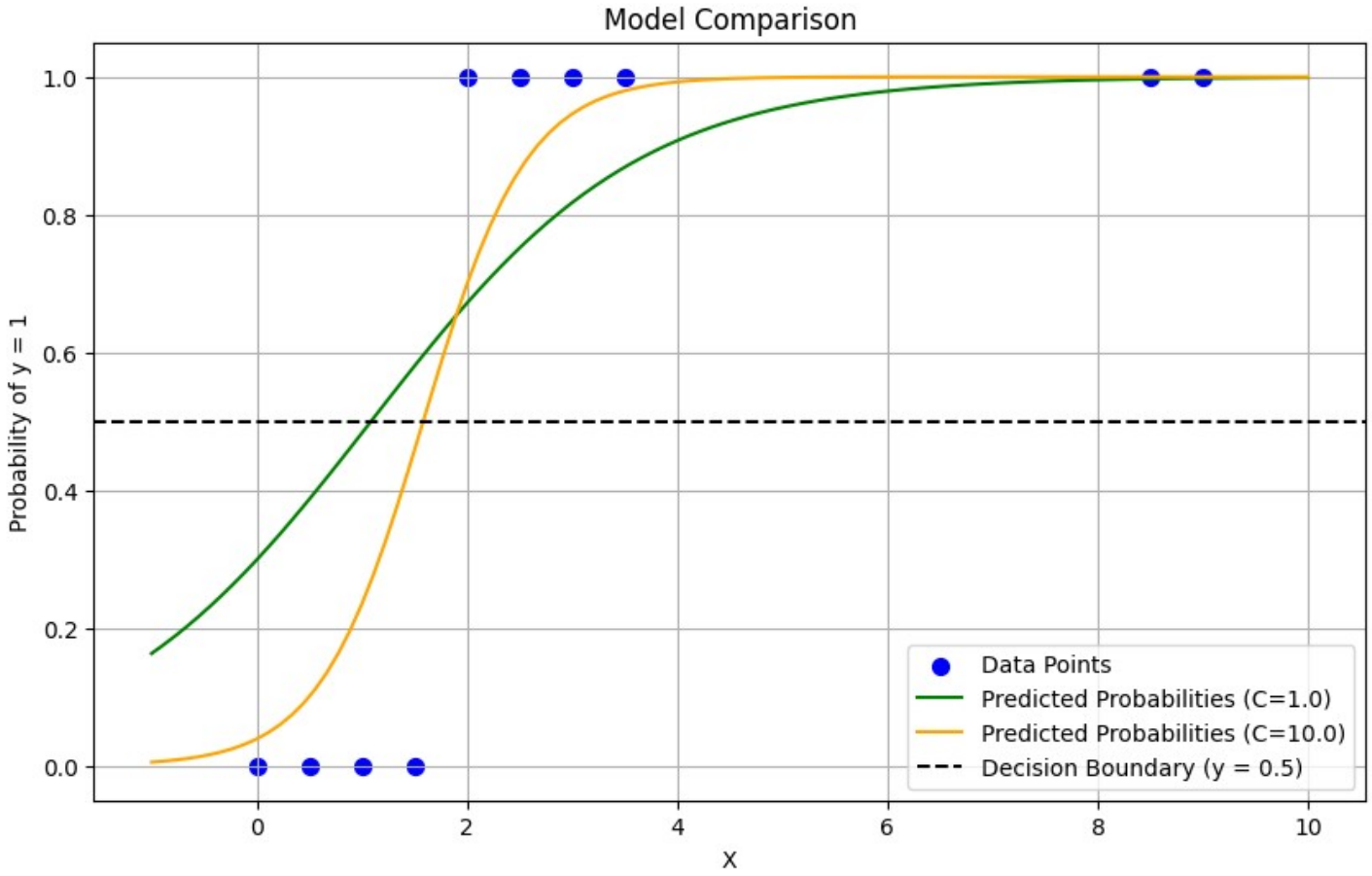
```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00         4
               1       1.00      1.00      1.00         6
```

Comparison of the original model with the improved model (tuned hyperparameter):

```
# Function to plot model results (decision boundary, data points, and predictions)
def plot_comparison(X, y, model_1, model_2, label_1, label_2, title="Model Comparison"):
    plt.figure(figsize=(10, 6))

    # Define a range of values for X (to plot the decision boundaries)
    x_values = np.linspace(np.min(X) - 1, np.max(X) + 1, 100).reshape(-1, 1)

    # Predict probabilities for each model using the same x values
    p_pred_1 = model_1.predict_proba(x_values)[:, 1]
    p_pred_2 = model_2.predict_proba(x_values)[:, 1]

    # Plot the data points
    plt.scatter(X, y, color='blue', s=50, marker='o', label="Data Points")

    # Plot the predicted curves for both models
    plt.plot(x_values, p_pred_1, color='green', label=f"Predicted Probabilities ({label_1})")
    plt.plot(x_values, p_pred_2, color='orange', label=f"Predicted Probabilities ({label_2})")

    # Plot decision boundary for both models
    plt.axhline(y=0.5, color='black', linestyle='--', label="Decision Boundary (y = 0.5)")

    # Labels and Title
    plt.xlabel("X")
    plt.ylabel("Probability of y = 1")
```

```
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()

print(model)
# Compare the two models (model and model_reg)
plot_comparison(X, y, model, model_reg, label_1="C=1.0", label_2="C=10.0")
```

```
LogisticRegression(random_state=0, solver='liblinear')
```



Binary Logistic Regression With `scikit-learn`: Example 2

Model Training, evaluation, and visualization

```
X = np.array([0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 8.5, 9]).reshape((-1, 1))
y = np.array([0, 1, 0, 0, 1, 1, 1, 1, 1, 1])

print(" X", X.shape, ":\n", X)
print("\n y", y.shape, ":\n", y)

# Initializing and fitting the binary logistic regression model
model = LogisticRegression(solver = 'liblinear', C = 10.0, random_state = 0)

model.fit(X, y)
```

```
# Evaluating the model accuracy with accuracy metric
mean_accuracy = model.score(X, y)
print(f"\n{'mean_accuracy':23} = {mean_accuracy}")

y_pred = model.predict(X)
mean_accuracy_eq = np.sum(y == y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")

# Display parameters of the model and verifying the probability of the dataset
print(f"\n{'intercept_':16} = {model.intercept_}")
print(f"{'coefficient_':16} = {model.coef_}")

print(f"\nModel Classes: ", model.classes_)

p_pred = model.predict_proba(X)
print(f"\nPredicted probabilities from the model:\n{p_pred}")

# Compute probability using equation
logit_y_values = model.intercept_ + model.coef_ * X
p_pred_eq = 1 / (1 + np.exp(-logit_y_values))
p_pred_eq = np.append(1 - p_pred_eq, p_pred_eq, axis = 1)
print(f"\nManually computed probabilities:\n{p_pred_eq}")

print("\nChecking that results from both methods are very close:\n", np.abs(p_pred - p_pred_eq) < 0.00
```

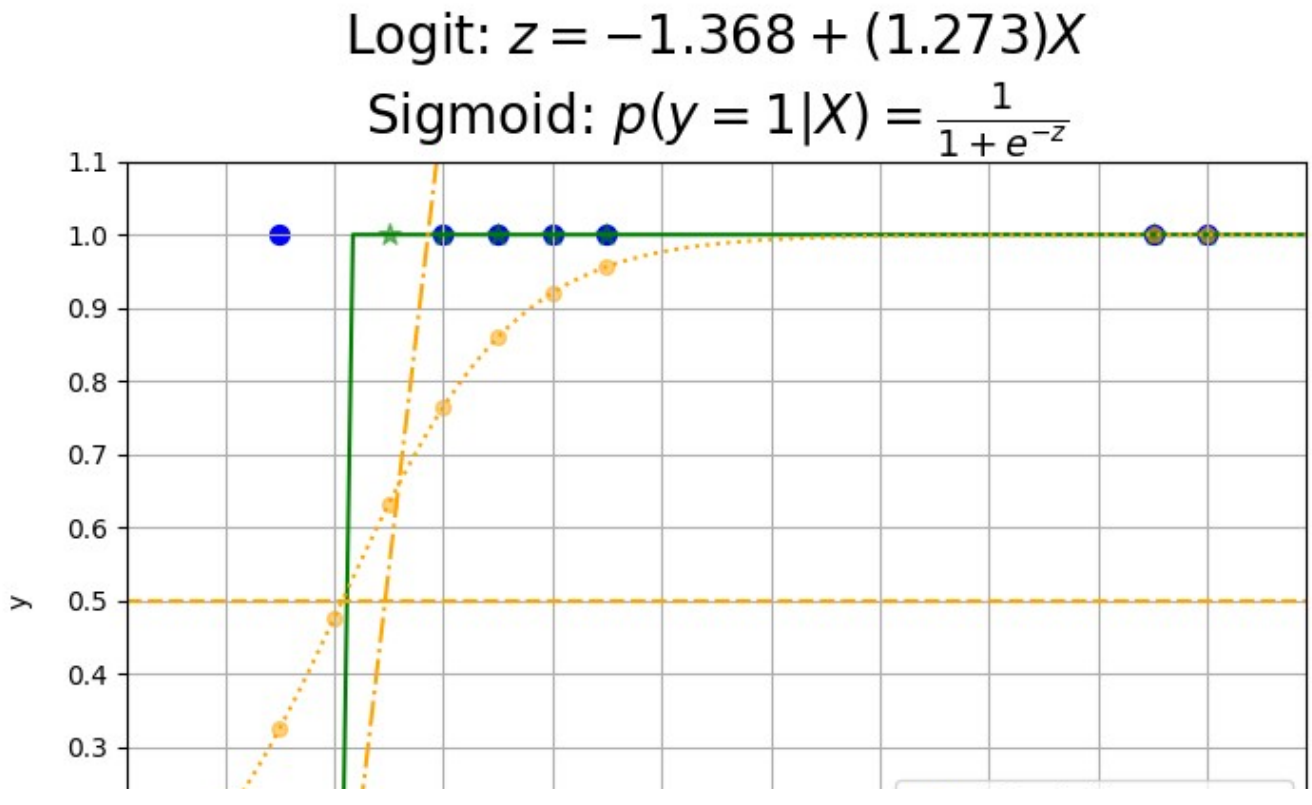
```
 X (10, 1) :
 [[0. ]
 [0.5]
 [1. ]
 [1.5]
 [2. ]
 [2.5]
 [3. ]
 [3.5]
 [8.5]
 [9. ]]

 y (10,) :
 [0 1 0 0 1 1 1 1 1 1]

mean_accuracy           = 0.8
mean_accuracy (manual)  = 0.8
y                       = [0 1 0 0 1 1 1 1 1 1]
y_pred                  = [0 0 0 1 1 1 1 1 1 1]

intercept_       = [-1.3683015]
coefficient_     = [[1.27276582]]

Model Classes:  [0 1]

Predicted probabilities from the model:
[[7.97105597e-01 2.02894403e-01]
 [6.75226153e-01 3.24773847e-01]
 [5.23865772e-01 4.76134228e-01]
```

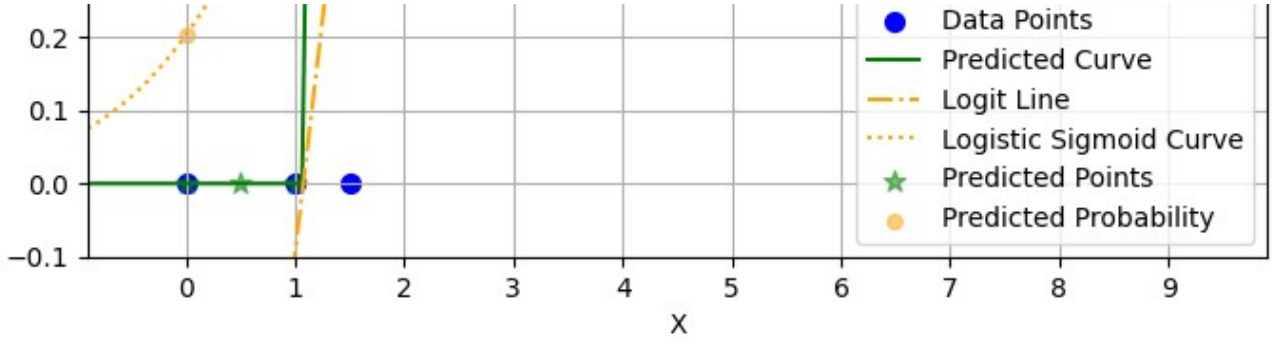
```
 [3.67990518e-01 6.32009482e-01]
 [2.35550591e-01 7.64449409e-01]
 [1.40202023e-01 8.59797977e-01]
 [7.94388450e-02 9.20561155e-01]
 [4.36726232e-02 9.56327377e-01]
 [7.86670137e-05 9.99921333e-01]
 [4.16323727e-05 9.99958368e-01]]

Manually computed probabilities:
[[7.97105597e-01 2.02894403e-01]
 [6.75226153e-01 3.24773847e-01]
 [5.23865772e-01 4.76134228e-01]
 [3.67990518e-01 6.32009482e-01]
 [2.35550591e-01 7.64449409e-01]
 [1.40202023e-01 8.59797977e-01]
 [7.94388450e-02 9.20561155e-01]
 [4.36726232e-02 9.56327377e-01]
 [7.86670137e-05 9.99921333e-01]
 [4.16323727e-05 9.99958368e-01]]

Checking that results from both methods are very close:
 [[ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
```

Plotting the result of the model:

```
plot_predicted_results(X, y, model = model)
```

```
<ipython-input-4-e08b0e75c97a>:84: DeprecationWarning: Conversion of an array with ndim > 0 to a s
  title_text = "Logit: $z = %.3f + (%.3f) X$\nSigmoid: $p(y = 1 | X) = \\frac{1}{1 + e^{-z}}$" % (
```

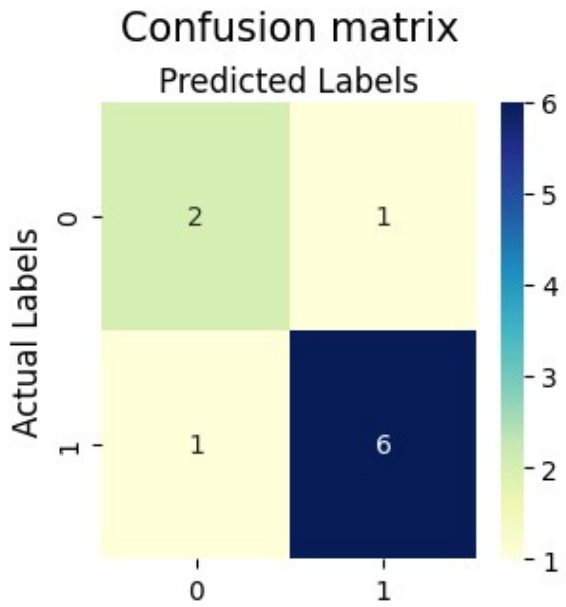




Confusion Matrix:

```
c_matrix = confusion_matrix(y, y_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
[[2 1]
 [1 6]]
```



Precision, Recall & F_1 - Score:

```
report = classification_report(y, y_pred)
print(report)
```

```
                precision    recall  f1-score   support

             0       0.67      0.67      0.67         3
             1       0.86      0.86      0.86         7

      accuracy                           0.80        10
     macro avg       0.76      0.76      0.76        10
  weighted avg       0.80      0.80      0.80        10
```

```
report = my_classification_report(y, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.67      0.67      0.67         3
           1       0.86      0.86      0.86         7
```

Multiple-Predictor Binary Logistic Regression

Multiple-Predictor Logistic Regression is also known as:

- Multiple Binary Logistic Regression
- Multiple Logistic Regression for Binary Classification
- Multiple Binary Logistic Regression
- Multiple Binary Logistic Regression with Multiple Predictors

It is a type of logistic regression used when there are multiple **independent variables** (predictors), to predict a **binary outcome** (dependent variable). Multiple-predictor logistic regression allows for the inclusion of several independent variables, which can be continuous or categorical.

Recap: Binary Logistic Regression refers to logistic regression with an outcome variable (dependent variable) being binary, meaning it can take one of two values (usually coded as 0 and 1).

Create and Fitting Multiple-Predictor Binary Logistic Regression

Consider a $m \times n$ input data matrix:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Where:

- $\mathbf{X}$ is our independent variables (also known as features or predictors)
- $\mathbf{y}$ is our dependent variable containing the label values

and

- m represents the data size (number of points = number of observations)
- n represents the data dimension (number of features)

Each element $y^{(i)}$ in $\mathbf{y}$ typically either contains the integer value of 1 to represent the **positive class**, and 0 otherwise.

The **Multiple-Predictor Binary Logistic Regression** aims to fit the input data ($\mathbf{X}$) to obtain the **best-fit** model

parameters $\theta_0, \theta_1, \ldots, \theta_n$ leading to:

$$
\begin{aligned}
z^{(1)} &= \theta_0 + \theta_1 x_1^{(1)} + \theta_2 x_2^{(1)} + \ldots + \theta_n x_n^{(1)} \\
z^{(2)} &= \theta_0 + \theta_1 x_1^{(2)} + \theta_2 x_2^{(2)} + \ldots + \theta_n x_n^{(2)} \\
&\vdots \\
z^{(m)} &= \theta_0 + \theta_1 x_1^{(m)} + \theta_2 x_2^{(m)} + \ldots + \theta_n x_n^{(m)}
\end{aligned}
$$

If we define vector $\mathbf{z}$ and θ, and rewrite $\mathbf{X}$ by augmenting an extra column of ones in the first column (the new θ is often called **Augmented Data Matrix** , we have

$$
\mathbf{z} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}
$$

we get

$$
\mathbf{z} = \begin{bmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{bmatrix} = \begin{bmatrix} \theta_0 + \theta_1 x_1^{(1)} + \theta_2 x_2^{(1)} + \ldots + \theta_n x_n^{(1)} \\ \theta_0 + \theta_1 x_1^{(2)} + \theta_2 x_2^{(2)} + \ldots + \theta_n x_n^{(2)} \\ \vdots \\ \theta_0 + \theta_1 x_1^{(m)} + \theta_2 x_2^{(m)} + \ldots + \theta_n x_n^{(m)} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}
$$

$$
\mathbf{z} = \mathbf{X}\theta
$$

To predict the label

$$
p(y^{(i)} = 1 | x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}) = \frac{1}{1 + e^{-(z^{(i)})}} \quad \text{for } i = 1, 2, \ldots, m.
$$

$$
p(y^{(i)} = 1 | x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \ldots + \theta_n x_n^{(i)})}} \quad \text{for } i = 1, 2, \ldots, m.
$$

This can be written more compactly as

$$
\mathbf{p}(\mathbf{y} = 1 | \mathbf{X}) = \frac{1}{1 + e^{-\mathbf{X}\theta}}
$$

It is customary to classify an input instance as the positive class (usually labeled as $y = 1$) if $p(y^{(i)} = 1 | x_1^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)}) \geq 0.5$. Consequently, the predicted label $\hat{y}^{(i)}$ is determined as follows:

$$
\hat{y}^{(i)} = \begin{cases} 0 & \text{if } \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \ldots + \theta_n x_n^{(i)})}} < 0.5, \\ 1 & \text{if } \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \ldots + \theta_n x_n^{(i)})}} \geq 0.5. \end{cases} \quad \text{for } i = 1, 2, \ldots, m.
$$

The above expression is commonly denoted using an **indicator function**

$$\hat{y}^{(i)} = 1\left(\frac{1}{1+e^{-(\theta_0+\theta_1 x_1^{(i)}+\theta_2 x_2^{(i)}+\ldots+\theta_n x_n^{(i)})}} \geq 0.5\right) \quad \text{for } i = 1, 2, \ldots, m.$$

The predicted label vector is thus as:

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix}$$

$$\hat{\mathbf{y}} = 1\left(\frac{1}{1+e^{-\mathbf{X}\theta}} \geq 0.5\right)$$

Create and Fitting the Logistic Regression Model Using `scikit-learn`

Creating the dataset:

```
X = np.array( [ [  5, 35],
                [ 10, 50],
                [ 25, 50],
                [ 40, 15],
                [ 25, 90],
                [ 40, 40],
                [ 75, 75],
                [ 80, 15],
                [ 90, 60],
                [100, 40]
              ] )

y = np.array( [0, 0, 0, 0, 0, 1, 1, 1, 1, 1] )

print(" X", X.shape, ":\n", X)
print("\n y", y.shape, ":\n", y)
```

```
 X (10, 2) :
 [[  5  35]
 [ 10  50]
 [ 25  50]
 [ 40  15]
 [ 25  90]
 [ 40  40]
 [ 75  75]
 [ 80  15]
 [ 90  60]
 [100  40]]

 y (10,) :
 [0 0 0 0 0 1 1 1 1 1]
```

Initializing and fitting the multinomial logistic regression model:

Initializing and fitting the multinomial logistic regression model:

```
model = LogisticRegression(solver = 'liblinear', C = 20, random_state = 42,
                           tol = 1e-10, max_iter = 1000) # Note: we use 20 here instead of the default

model.fit(X, y)
```

```
LogisticRegression(C=20, max_iter=1000, random_state=42, solver='liblinear',
                   tol=1e-10)
```

Evaluating the model accuracy with accuracy metric:

```
mean_accuracy = model.score(X, y)
print(f"\n{'mean_accuracy':23} = {mean_accuracy}")

y_pred = model.predict(X)
mean_accuracy_eq = np.sum(y == y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")
```

```
mean_accuracy           = 0.8
mean_accuracy (manual)  = 0.8
y                       = [0 0 0 0 0 1 1 1 1 1]
y_pred                  = [0 0 0 1 0 0 1 1 1 1]
```

Display parameters of the model and verifying the probability of the dataset

```
print(f"{'n_iter_':16} = {model.n_iter_}")
print(f"{'intercept_':16} = {model.intercept_}")
print(f"{'coefficient_':16} = {model.coef_}")

print(f"\nModel Classes: ", model.classes_)

p_pred = model.predict_proba(X)
print(f"\nPredicted probabilities from the model:\n{p_pred}")

# Compute probability using equation
# NOTE: we need to sum up over the 2 features using np.sum()
logit_y_values = model.intercept_ + np.sum(model.coef_ * X, axis = 1).reshape((-1, 1))
p_pred_eq = 1 / (1 + np.exp(-logit_y_values))
p_pred_eq = np.append(1 - p_pred_eq, p_pred_eq, axis = 1)
print(f"\nManually computed probabilities:\n{p_pred_eq}")

print("\nChecking that results from both methods are very close:\n", np.abs(p_pred - p_pred_eq) < 0.00
```

```
n_iter_           = [13]
intercept_        = [-3.93329741]
coefficient       = [[ 0.10399613 -0.01006136]]
```

```
coefficient_     = [[ 0.10599015 -0.01666156]]

Model Classes:  [0 1]

Predicted probabilities from the model:
[[0.97736725 0.02263275]
 [0.96759197 0.03240803]
 [0.86252976 0.13747024]
 [0.48110212 0.51889788]
 [0.90369021 0.09630979]
 [0.54386358 0.45613642]
 [0.0426227  0.9573773 ]
 [0.01426649 0.98573351]
 [0.00798104 0.99201896]
 [0.00232001 0.99767999]]

Manually computed probabilities:
[[0.97736725 0.02263275]
 [0.96759197 0.03240803]
 [0.86252976 0.13747024]
 [0.48110212 0.51889788]
 [0.90369021 0.09630979]
 [0.54386358 0.45613642]
 [0.0426227  0.9573773 ]
 [0.01426649 0.98573351]
 [0.00798104 0.99201896]
 [0.00232001 0.99767999]]

Checking that results from both methods are very close:
 [[ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]
 [ True  True]]
```

Plotting the result of the model in 3D:

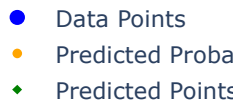
```
import plotly.graph_objects as go

points = go.Scatter3d(
                      x = X[:, 0],
                      y = X[:, 1],
                      z = y.reshape(-1),
                      mode = 'markers',
                      marker = {"size": 8, "color": "blue"},
                      name = 'Data Points'
                     )

points_pred = go.Scatter3d(
                      x = X[:, 0],
                      y = X[:, 1],
                      z = y_pred.reshape(-1),
                      mode = 'markers',
                      marker = {"size": 4, "color": "green", "symbol": "diamond"},
```

```
                     name = 'Predicted Points'
                    )

points_p_pred = go.Scatter3d(
                     x = X[:, 0],
                     y = X[:, 1],
                     z = p_pred[:, 1],
                     mode = 'markers',
                     marker = {"size": 5, "color": "orange"},
                     name = 'Predicted Probability'
                    )

coef0 = model.intercept_
coef1 = model.coef_.squeeze()[0]
coef2 = model.coef_.squeeze()[1]

logit_X1, logit_X2 = np.meshgrid(np.linspace(0, 100, 100),
                                 np.linspace(0, 100, 100))
logit_y = model.intercept_ + (coef1 * logit_X1 + coef2 * logit_X2)

surface_logit_plane = go.Surface(x = logit_X1, y = logit_X2, z = logit_y,
                           opacity = 0.4, colorscale = 'Blues',
                           showscale = False,
                           name = 'Logit Plane'
                          )

p_y = 1 / (1 + np.exp(-logit_y))
surface_p_plane = go.Surface(x = logit_X1, y = logit_X2, z = p_y,
                           opacity = 0.5, colorscale = 'YlOrRd',
                           showscale = False,
                           name = 'Predicted Probability'
                          )

horizontal_plane = go.Surface(x = logit_X1, y = logit_X2, z = 0.5 * np.ones_like(logit_X1),
                           opacity = 0.3, colorscale='YlOrBr',
                           showscale = False,
                           name = 'Plane y = 0.5'
                          )

fig = go.Figure(data = [horizontal_plane, surface_logit_plane, surface_p_plane,
                        points, points_p_pred, points_pred])

title_text = "$P(y = 1 | X1, X2) = \\frac{1}{1 + e^{-(%.3f + (%.3f) X1 + (%.3f) X2 )}}$" % (coef0, coe
# title_text = "Logistic Regression"

fig.update_layout(scene = {"xaxis_title": "X1",
                           "yaxis_title": "X2",
                           "zaxis_title": "y",
                           "zaxis_range": [-0.1, 1.1],
                           "zaxis_dtick": 0.1
                           },
                  width = 800,
                  height = 600,
                  scene_camera = { "eye": {"x": -1, "y": -1.5, "z": 1.5} },
                  title = {"text": title_text,
                           "font": {"size": 40},
```

```
                            "x": 0.5,
                            "xanchor": "center"
                           }
                )

fig.show()
```

```
<ipython-input-37-e8124a2f2c47>:60: DeprecationWarning:

Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure y
```

$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(-3.933+(0.104)X1+(-0.010)X2)}}$$



Plotting the result of the model in 2D:

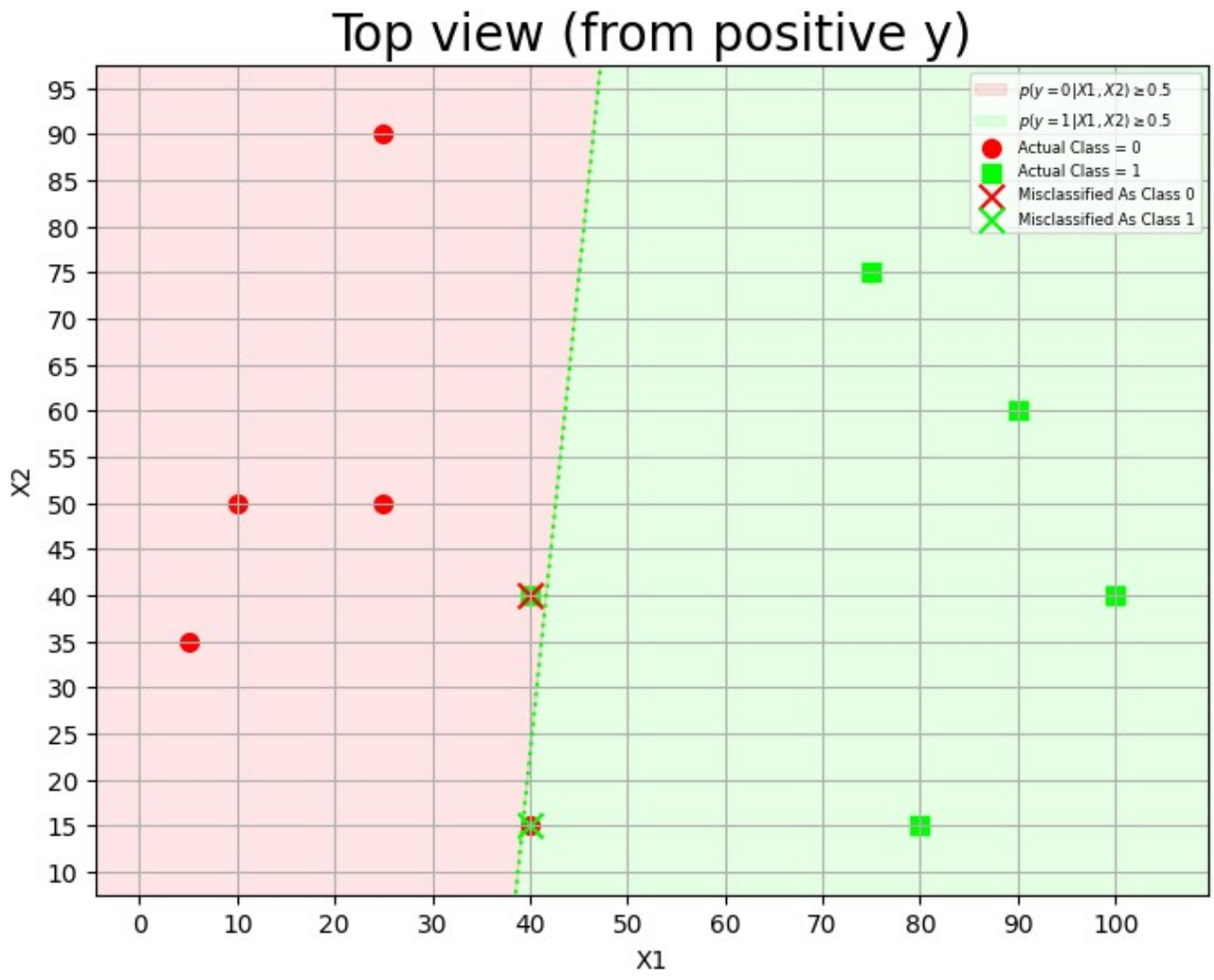
```
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.colors import LinearSegmentedColormap

# Given y value:
#    y = coef0 + coef1 * X1 + coef2 * X2
#    X2 = (y - coef0 - coef1 * X1) / coef2
def get_logit_X2(coef0, coef1, coef2, X1, y):
  return (y - coef0 - coef1 * X1) / coef2
```

```
def getY(X1, X2):
  array = np.array([X1, X2]).reshape(1, -1)
  return model.predict(array)[0]

def plot_logistic_regression_top_view(X, y, model,
                                      xlim = "auto", ylim = "auto",
                                      figure_size = (8, 6),
                                      plot_logit_line = True,
                                      plot_logit_region = True,
                                      plot_classes_all_points = True,
                                      X_new = None
                                      ):
  plt.figure(figsize=figure_size)

  # colors = ["red", "green", "blue", "cyan", "magenta"]
  markers = ["o", "s", "d", "p", "h"]
  num_classes = model.classes_.size
  colors = [(1, 0, 0),  # Red
            (0, 1, 0),  # Green
            (0, 0, 1),  # Blue
            (0, 1, 1),  # Cyan
            (1, 0, 1)   # Magenta
            ]
  my_cmap = LinearSegmentedColormap.from_list('custom_cmap', colors, N=num_classes)

  if xlim == "auto":
    xlim_min, xlim_max = np.min(X[:, 0]), np.max(X[:, 0])
    dx = (xlim_max - xlim_min) * 0.1
  else:
    xlim_min, xlim_max = xlim
    dx = 0

  if ylim == "auto":
    ylim_min, ylim_max = np.min(X[:, 1]), np.max(X[:, 1])
    dy = (ylim_max - ylim_min) * 0.1
  else:
    ylim_min, ylim_max = ylim
    dy = 0

  xlim_min -= dx
  xlim_max += dx
  ylim_min -= dy
  ylim_max += dy

  plt.xlim(xlim_min, xlim_max)
  plt.ylim(ylim_min, ylim_max)
  plt.locator_params(axis = 'x', nbins = 20)
  plt.locator_params(axis = 'y', nbins = 20)
  plt.xlabel('X1')
  plt.ylabel('X2')

  coef0 = model.intercept_
  coef1 = model.coef_[:, 0]
  coef2 = model.coef_[:, 1]
```

```
logit_X1 = np.linspace(xlim_min, xlim_max, 100).reshape(-1, 1)

logit_X2_y0            = get_logit_X2(coef0, coef1, coef2, logit_X1, y = 0)
logit_X2_y1000         = get_logit_X2(coef0, coef1, coef2, logit_X1, y = 1000)
logit_X2_yNegative1000 = get_logit_X2(coef0, coef1, coef2, logit_X1, y = -1000)

logit_X1 = logit_X1.reshape(-1)

if num_classes == 2:
  plt.fill_between(logit_X1, logit_X2_y0[:, 0], logit_X2_yNegative1000[:, 0],
                   color=colors[0], alpha=0.1,
                   label = "$p(y = 0 | X1, X2) \geq 0.5$")
  plt.fill_between(logit_X1, logit_X2_y0[:, 0], logit_X2_y1000[:, 0],
                   color=colors[1], alpha=0.1,
                   label = "$p(y = 1 | X1, X2) \geq 0.5$")
  plt.plot(logit_X1, logit_X2_y0[:, 0],
           linestyle='dotted', color = colors[1])
else:
  if plot_logit_region:
    for i in range(num_classes):
      plt.fill_between(logit_X1, logit_X2_y0[:, i], logit_X2_y1000[:, i], color=colors[i], alpha=0.1
                       label = "$p(y = %d | X1, X2) \geq 0.5$" % i)
  if plot_logit_line:
    for i in range(num_classes):
      plt.plot(logit_X1, logit_X2_y0[:, i],
               linestyle='dotted', color = colors[i])
  if plot_classes_all_points:
      X1_values = np.linspace(xlim_min, xlim_max, 100)
      X2_values = np.linspace(ylim_min, ylim_max, 100)
      X1, X2 = np.meshgrid(X1_values, X2_values)
      Y = np.vectorize(getY)(X1, X2)
      for i in range(num_classes):
        plt.scatter(X1[Y==i], X2[Y==i], color = colors[i], s = 2, alpha = 0.3,
                    label = f"Class = {i}")


for i in range(num_classes):
  plt.scatter(X[y == i][:, 0], X[y == i][:, 1], color = colors[i], s = 50,
              label = f"Actual Class = {i}", marker = markers[i])

y_pred = model.predict(X)
X_misclassified = X[y != y_pred]
y_misclassified = y_pred[y != y_pred]
for i in range(num_classes):
  Xi = X_misclassified[y_misclassified == i]
  if len(Xi) == 0:
    continue
  plt.scatter(Xi[:, 0],
              Xi[:, 1],
              color = colors[i],
              s = 100, marker = "x",
              label = f"Misclassified As Class {i}")

if X_new is not None:
  y_pred_new = model.predict(X_new)
  for i in range(num_classes):
    Xi = X_new[y_pred_new == i]
```

```
      if len(Xi) == 0:
        continue
      plt.scatter(Xi[:, 0],
                  Xi[:, 1],
                  color = colors[i],
                  alpha=0.9, s = 150, marker = "*",
                  label = f"New Predicted As Class {i}")

  plt.title("Top view (from positive y)", fontsize = 20)

  plt.legend(fontsize = 6)
  plt.grid(True)
  plt.show()
```

```
plot_logistic_regression_top_view(X, y, model)
```

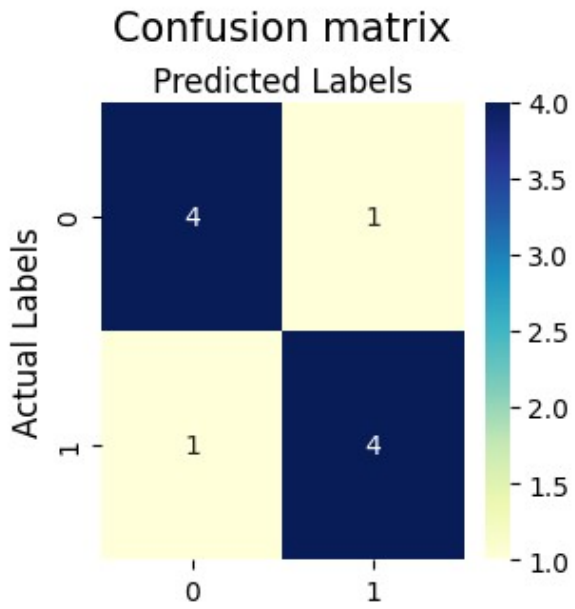


Confusion Matrix:

```
c_matrix = confusion_matrix(y, y_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
    [[4 1]
```

```
[1 4]]
```



Precision, Recall & F_1 - Score:

```
report = classification_report(y, y_pred)
print(report)
```

```
                precision    recall  f1-score   support

             0       0.80      0.80      0.80         5
             1       0.80      0.80      0.80         5

      accuracy                           0.80        10
     macro avg       0.80      0.80      0.80        10
  weighted avg       0.80      0.80      0.80        10
```

```
report = my_classification_report(y, y_pred)
print(report)
```

```
                precision    recall  f1-score   support

             0       0.80      0.80      0.80         5
             1       0.80      0.80      0.80         5
```

Understanding Bias-Variance Tradeoff in Cross-Validation for Hyperparameter Tuning (A continuation of the discussion in "Improve the Model" subsection)

In the previous subsection `Single-Predictor Binary Logistic Regression` > `Single-Predictor Binary Logistic Regression With scikit-learn: Example 1` > `Improve the Model`, we have tested the use of cross-validation to find the optimal C value using k-fold cross validation, which was 10.0. The model seemed to have improved its results in accuracy, precision, recall, and f1-score compared to the one with the default value of 1.0.

With the multiple-predictor model that we have trained just now, we used a value of 20.0 for the regularization strength, which was an arbitrary number selected. The results returned was an accuracy, precision, recall, and f1-score of 0.8.

We want to try and tune our hyperparameters so that the model would perform well. To do so, we use cross-validation method to select the best C. However, the results returns was 0.01, which is a smaller value than 20.0, and an even smaller value than the default 1.0. The accuracy and other metrics will then fall. Using a value of 30 increases the accuracy to 1.0. Why is that?

To recap:

- Large C = Less regularization → Risk of overfitting
- Small C = More regularization → Risk of underfitting

Cross-validation for hyperparameter tuning:

```
# Define the parameter grid for C
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 20, 30, 100]}

# Create the logistic regression model
model = LogisticRegression(solver='liblinear')

# Set up Leave-One-Out Cross-Validation
loo = LeaveOneOut()

# Set up grid search cross-validation with LOO CV
grid_search = GridSearchCV(model, param_grid, cv=loo)

# Fit the model and find the best C
grid_search.fit(X, y)

# Print the best C and the corresponding score
print(f"Best C = {grid_search.best_params_['C']}")
print(f"Best cross-validation score = {grid_search.best_score_}")

# Returns best C as 0.001
```

```
Best C = 0.001
Best cross-validation score = 0.9
```

Training models with the regularization strength of 30 (model_1) vs 0.001 (model_2):

```
model_1 = LogisticRegression(C = 30, random_state = 42,
                             tol = 1e-10, max_iter = 1000) # Use of C=30 here

model_1.fit(X, y)
```

```
LogisticRegression
LogisticRegression(C=30, max_iter=1000, random_state=42, tol=1e-10)
```

```
model_2 = LogisticRegression(C = 0.001, random_state = 42,
                             tol = 1e-10, max_iter = 1000) # Use of C=30 here

model_2.fit(X, y)
```

```
LogisticRegression(C=0.001, max_iter=1000, random_state=42, tol=1e-10)
```

Display the results for comparison:

```
# Model 1 Results and Accuracy
print("Model 1 Results and Accuracy:")

m1_mean_accuracy = model_1.score(X, y)
print(f"{'mean_accuracy':23} = {m1_mean_accuracy}")

m1_y_pred = model_1.predict(X)
m1_mean_accuracy_eq = np.sum(y == m1_y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {m1_mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {m1_y_pred}")

# Model 2 Results and Accuracy
print("\nModel 2 Results and Accuracy:")

m2_mean_accuracy = model_2.score(X, y)
print(f"{'mean_accuracy':23} = {m2_mean_accuracy}")

m2_y_pred = model_2.predict(X)
m2_mean_accuracy_eq = np.sum(y == m2_y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {m2_mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")
```

```
Model 1 Results and Accuracy:
mean_accuracy           = 1.0
mean_accuracy (manual)  = 1.0
y                       = [0 0 0 0 0 1 1 1 1 1]
y_pred                  = [0 0 0 0 0 1 1 1 1 1]

Model 2 Results and Accuracy:
mean_accuracy           = 0.9
mean_accuracy (manual)  = 0.9
y                       = [0 0 0 0 0 1 1 1 1 1]
y_pred                  = [0 0 0 1 0 0 1 1 1 1]
```

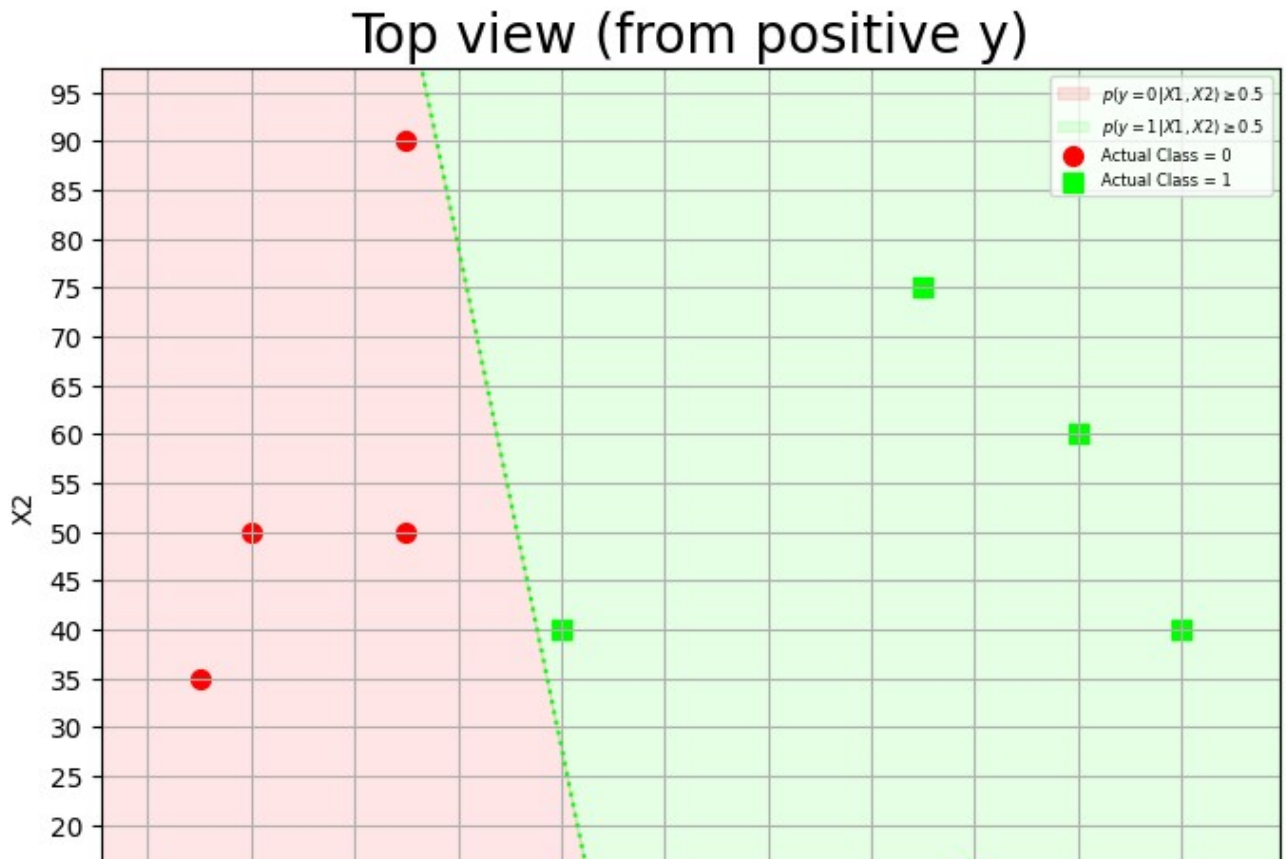
```
# Model 1 classification report
m1_report = classification_report(y, m1_y_pred)
print("Model 1 Classification Report:\n", m1_report)
```

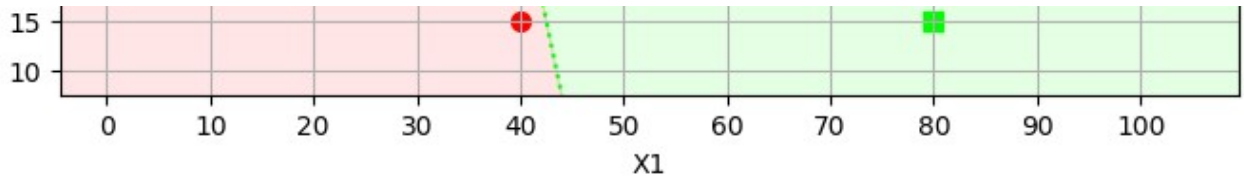
```
# Model 2 classification report
m2_report = classification_report(y, m2_y_pred)
print("\nModel 2 Classification Report:\n", m2_report)
```

```
Model 1 Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      1.00      1.00         5

    accuracy                           1.00        10
   macro avg       1.00      1.00      1.00        10
weighted avg       1.00      1.00      1.00        10


Model 2 Classification Report:
               precision    recall  f1-score   support

           0       0.83      1.00      0.91         5
           1       1.00      0.80      0.89         5

    accuracy                           0.90        10
   macro avg       0.92      0.90      0.90        10
weighted avg       0.92      0.90      0.90        10
```

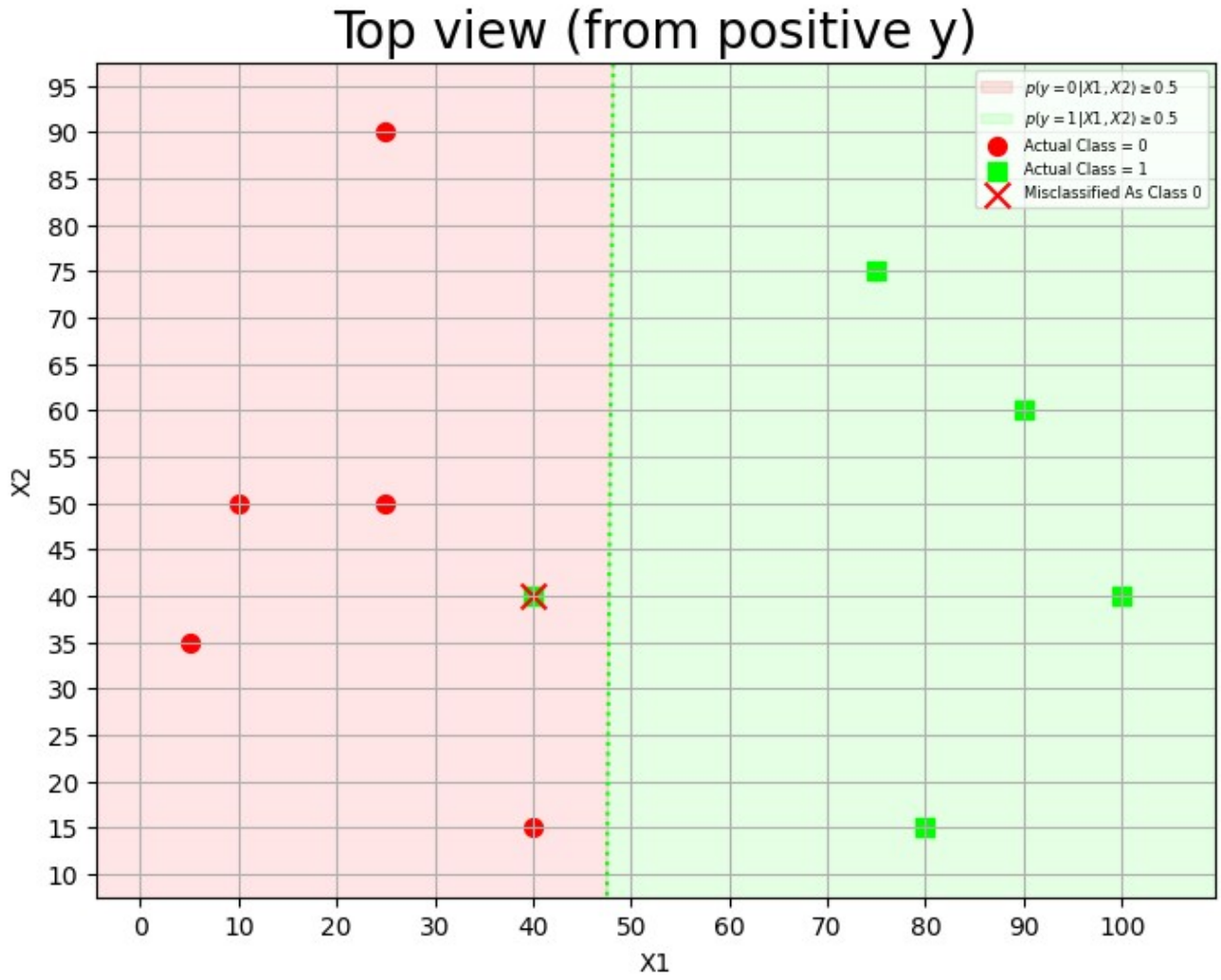
```
# Display top view for both
print("Model 1 Plot:")
plot_logistic_regression_top_view(X, y, model_1)
print("\n\nModel 2 Plot:")
plot_logistic_regression_top_view(X, y, model_2)
```

```
Model 1 Plot:
```



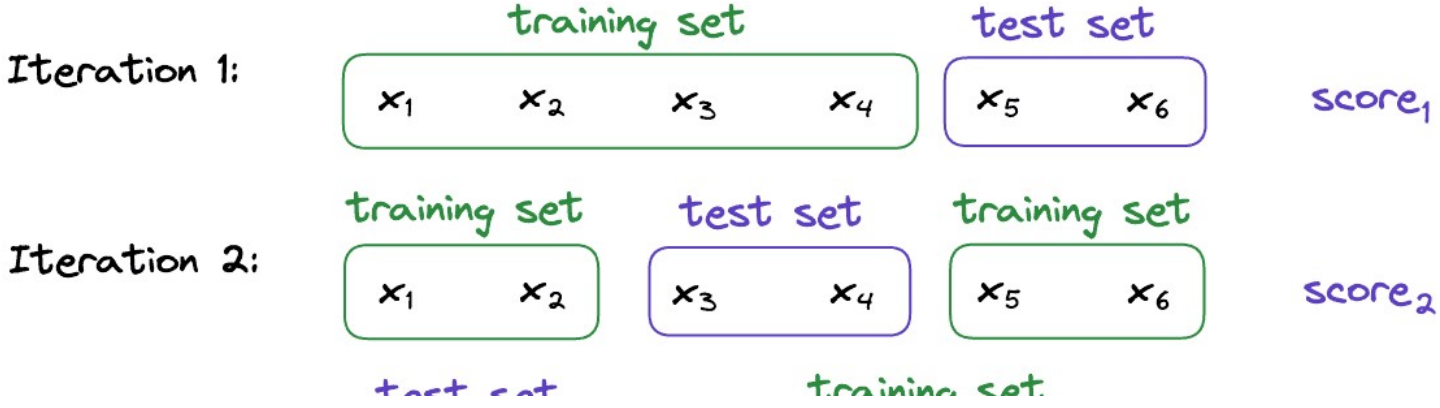


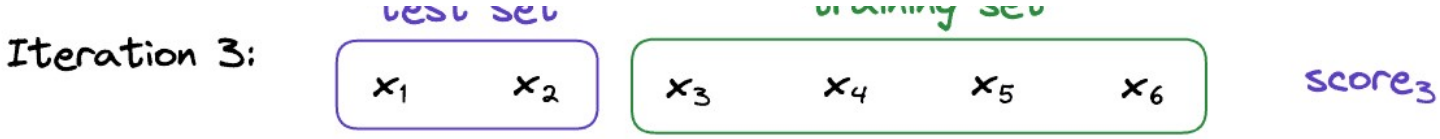
Model 2 Plot:



Why cross-validation chooses C=0.001 as the best hyperparameter, even though the C=30 model returns higher accuracy

1) The Workings behind K-Fold Cross-Validation:





Remember how the k-fold cross validation method works? We choose an amount for how large we want the subsets to be, and split all the data into equally sized subsets. The amount of folds are the amount of total subsets. Each subset will be used as test data only once (1), and the rest will be used as training data. This happens for however many folds there are. The results of each fold are averaged to assess the performance.

In this case, $C = 0.001$ has provided a better result on the test data on average compared to $C = 30$. This means that the model is overfitting for $C = 30$, and fails to generalize well for the test data.

2) The Goal of Cross-Validation: Cross-validation aims to balance bias and variance

As mentioned above, the k-fold cross-validation uses folds to assesses accuracy. They do this using the train-test-split method, where the average of accuracy is obtained for all folds. This way, the cross-validation method can control the tradeoff between bias and variance (bias–variance tradeoff)—it finds a good balance between underfitting and overfitting.

From Wikipedia:

- The bias error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).
- The variance is an error from sensitivity to small fluctuations in the training set. High variance may result from an algorithm modeling the random noise in the training data (overfitting).

That is why even though we measure the accuracy of the model using its training data to be 1.0, it does not mean it will perform better when predicting unseen data. It simply means that the model has been overfitted to the training data (high variance).

To recap:

K-fold cross-validation helps select the best hyperparameter (C) for a model by balancing the bias-variance tradeoff.

A large C (low regularization) can lead to overfitting, where the model fits the training data too closely and fails to generalize well to unseen data. This is why testing a model with the C=30 parameter using the training data, we obtained a 1.0 accuracy.

On the other hand, a small C (high regularization) might cause underfitting, but helps avoid overfitting, leading to better generalization across test data.

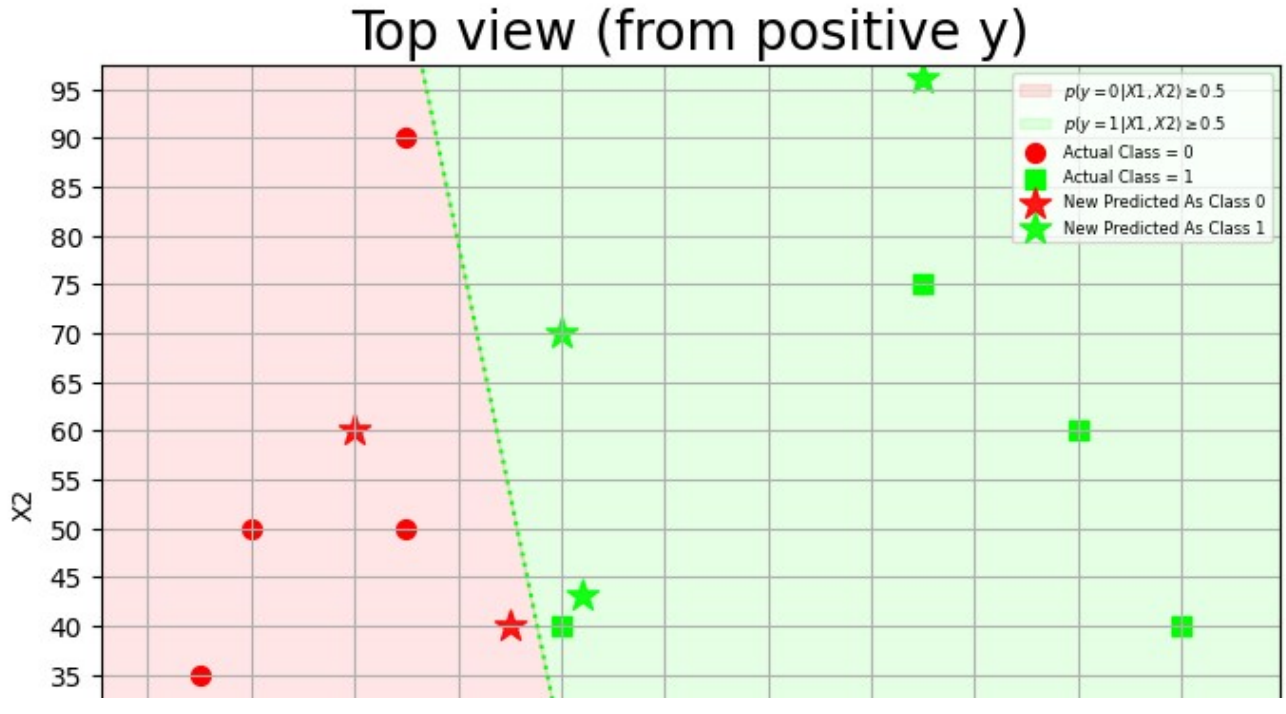
Thus,

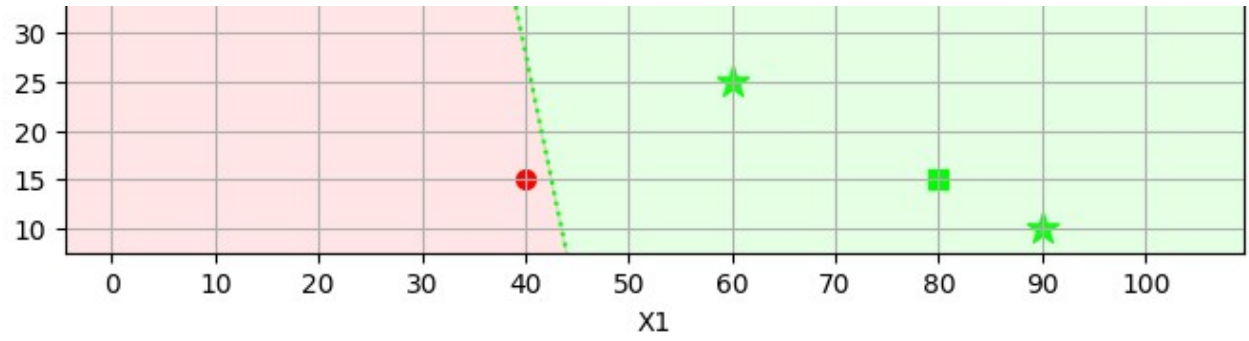
- Large C = Less regularization → Lower bias but higher variance (potential overfitting).
- Small C = More regularization → Higher bias but lower variance (potential underfitting).

Predict New Data Points

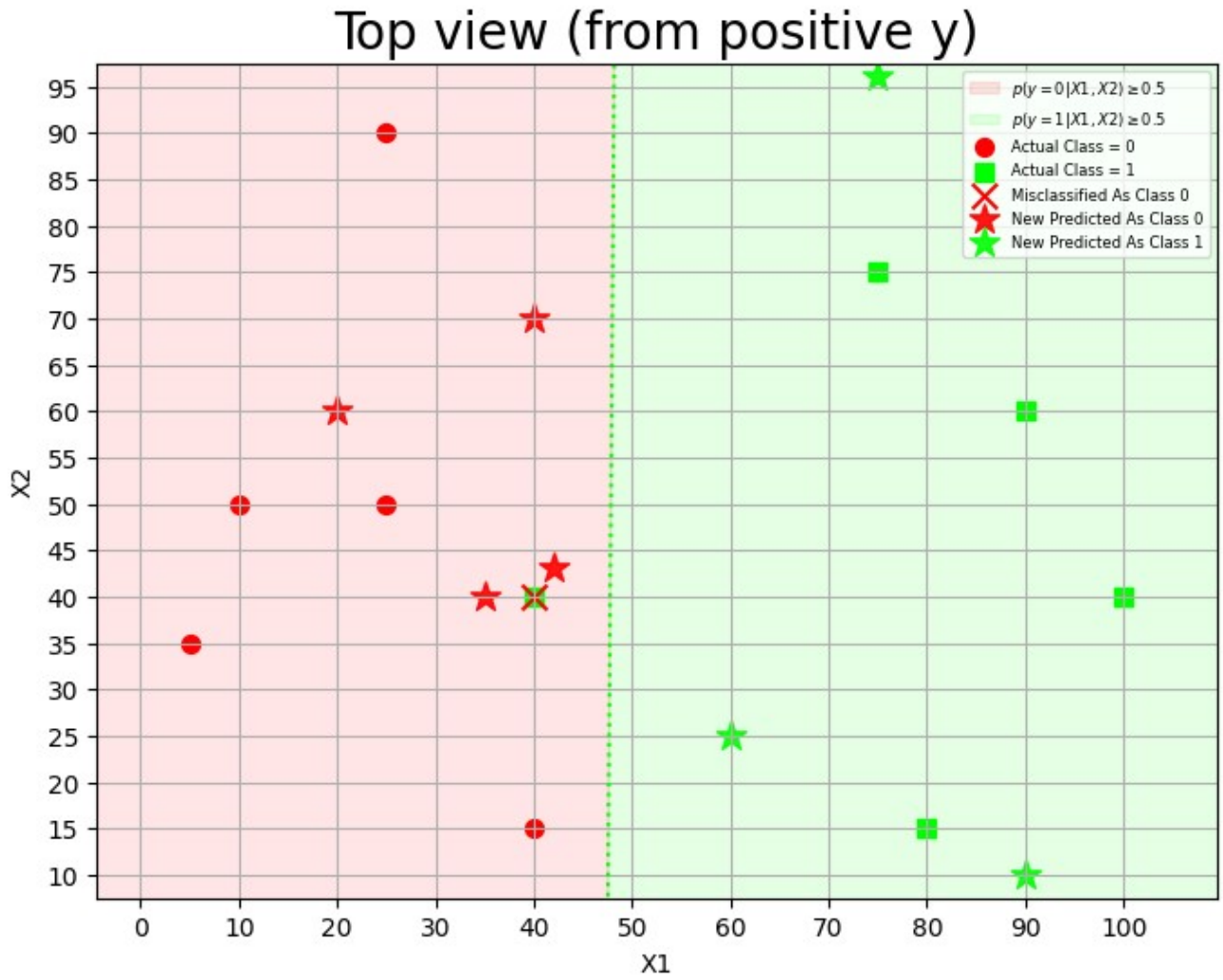
```
X_new = np.array( [ [ 35, 40],
                    [ 40, 70],
                    [ 20, 60],
                    [ 60, 25],
                    [ 42, 43],
                    [ 75, 96],
                    [ 90, 10]
                  ] )

print(" X_new", X_new.shape, ":\n", X)

y_pred_m1 = model_1.predict(X_new)
y_pred_m2 = model_2.predict(X_new)
print("\ny_pred_m1 = ", y_pred_m1)
print("y_pred_m2 = ", y_pred_m2)

print("\nModel 1 Plot:")
plot_logistic_regression_top_view(X, y, model_1, X_new=X_new)
print("\n\nModel 2 Plot:")
plot_logistic_regression_top_view(X, y, model_2, X_new=X_new)
```

```
 X_new (7, 2) :
 [[  5  35]
 [ 10  50]
 [ 25  50]
 [ 40  15]
 [ 25  90]
 [ 40  40]
 [ 75  75]
 [ 80  15]
 [ 90  60]
 [100  40]]

y_pred_m1 =  [0 1 0 1 1 1 1]
y_pred_m2 =  [0 0 0 1 0 1 1]

Model 1 Plot:
```





Model 2 Plot:



TODO: Trying Different Datasets and How It Affects the Selection of C

From the discussion above, we find that cross-validation aims to find the best tradeoff between bias and variance. The distribution and characteristics of the data, such as the level of noise, the number of features, and the complexity of the underlying patterns, will have a significant impact on this tradeoff.

1. For datasets with **high noise**, a smaller C will be chosen to avoid to avcoid the model fitting to the noise.
2. For datasets with **clear patterns**, a larger C might work well as the model can afford to fit more closely to the data.
3. For datasets with **high dimensionality** (many features), smaller C can help to reduce the complexity of the model, focusing only on the most important features.

All this aside, remember that the cross-validation will find the best tradeoff between bias and variance, so it's not guaranteed that a dataset with high noise will definitely have a smaller C.

```
np.random.seed(42)

# Base: Generate 200 samples for each class with two features
X_class0 = np.random.uniform(low=[5, 15], high=[50, 100], size=(100, 2))
X_class1 = np.random.uniform(low=[10, 10], high=[60, 90], size=(100, 2))

# Noise: Add noise to the dataset by shifting the data randomly
X_class0_noise = X_class0 + np.random.normal(0, 15, X_class0.shape)
X_class1_noise = X_class1 + np.random.normal(0, 15, X_class1.shape)

# Clear Separation: Clear separation between classes with a wider range of feature values
X_class0_clear = np.random.uniform(low=[10, 30], high=[40, 80], size=(100, 2))
X_class1_clear = np.random.uniform(low=[30, 40], high=[70, 100], size=(100, 2))

# High Dimensionality: Increase the dimensionality by adding more features (total 10)
X_class0_dim = np.random.uniform(low=[5, 15, 25, 35, 45, 5, 10, 15, 20, 25],
                                 high=[50, 60, 70, 80, 90, 40, 50, 60, 70, 80],
                                 size=(100, 10))

X_class1_dim = np.random.uniform(low=[10, 10, 20, 30, 40, 10, 20, 30, 40, 50],
                                 high=[60, 70, 80, 90, 100, 50, 60, 70, 80, 90],
                                 size=(100, 10))

# TODO: Experiment by changing the data used for X
X = np.vstack((X_class0, X_class1))
# X = np.vstack((X_class0_noise, X_class1_noise))
# X = np.vstack((X_class0_clear, X_class1_clear))
# X = np.vstack((X_class0_dim, X_class1_dim))

y = np.array([0] * 100 + [1] * 100)

# Leave-One-Out Cross-Validation
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 20, 30]}

model = LogisticRegression(solver='liblinear')

loo = LeaveOneOut()

grid_search = GridSearchCV(model, param_grid, cv=loo)

grid_search.fit(X, y)

print(f"Best C = {grid_search.best_params_['C']}")
print(f"Best cross-validation score = {grid_search.best_score_}")
```

```
Best C = 1
Best cross-validation score = 0.625
```

Multiclass Logistic Regression

Also known as:

- Multinomial Logistic Regression

Multiclass Logistic Regression is a type of logistic regression used when the dependent variable has more than two possible outcomes or categories. Instead of predicting a binary outcome like in binary logistic regression, it is used for predicting one of several possible classes. This approach can handle multiple independent variables, which may be either continuous or categorical, and models the probability of each class occurring.

Normally, two methods are used for calculating the probability for each category, which is the **One-vs-Rest (OVR)** method or **Softmax**, allowing the data to be classified into more than two classes.

Create and Fitting Multiple-Predictor Binary Logistic Regression

Consider a $m \times n + 1$ input **augmented data matrix** $\mathbf{X}$ for our independent variables (also known as features or predictors) and $\mathbf{y}$ for our dependent variable containing the label values:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Where:

- m represents the data size (number of points = number of observations)
- n represents the data dimension (number of features)

Recap: Binary Classification Problem with Single Position Class

For the binary classification problem with single position class, each element $y^{(i)}$ in $\mathbf{y}$ typically contains the integer value of 1 to represent the **positive class**, and 0 otherwise.

The logistic regression aim to fit the input data ($\mathbf{X}$) to obtain the **best-fit** model parameters

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

After the model has been fitted, the prediction can be done by first computing the **logit** (also known as **log-odds**)

$$\mathbf{z} = \mathbf{X}\theta$$

Then the probability vector is given by:

$$\mathbf{p}(\mathbf{y}=1|\mathbf{X}) = \frac{1}{1+e^{-\mathbf{z}}}$$

The predicted label vector is then computed as:

$$\hat{\mathbf{y}} = 1\left(\frac{1}{1+e^{-\mathbf{z}}} \geq 0.5\right)$$

Multiclass Logistic Regression

While binary logistic regression predicts a single probability for the positive class, multiclass logistic regression predicts probabilities for all classes, and the class with the highest probability is chosen as the prediction.

For Multiclass Logistic Regression, each element $y^{(i)}$ in $\mathbf{y}$ can contain the integer value of $0, 1, 2, \ldots, K-1$ that represents the class labels of data point i. K is the number of classes.

We now define a $(n+1)\times K$ matrix, $\mathbf{\Theta}$. The matrix contains the parameters (weights) for each class:

$$\mathbf{\Theta} = \begin{bmatrix} \theta_{0,0} & \theta_{0,1} & \cdots & \theta_{0,K-1} \\ \theta_{1,0} & \theta_{1,1} & \cdots & \theta_{1,K-1} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{n,0} & \theta_{n,1} & \cdots & \theta_{n,K-1} \end{bmatrix}$$

where:

- n is the number of features (independent variables) in the input data
- K is the number of classes in the multiclass classification problem

Each column from the matrix $\mathbf{\Theta}$ corresponds to a set of parameters for one class. This is how we define column k of $\mathbf{\Theta}$:

$$\theta_{(\mathbf{k})} = \begin{bmatrix} \theta_{0,k} \\ \theta_{1,k} \\ \vdots \\ \theta_{n,k} \end{bmatrix}$$

representing the model parameter vector for class k, for $k = 0, 1, 2, \ldots, K-1$.

After the model has been fitted, the prediction can be done by first computing the **logit** (also known as **log-odds**), represented by z for each class k:

$$\mathbf{z_{(k)}} = \mathbf{X}\theta_{(\mathbf{k})}$$

Instead of only having a single logit in binary logistic regression, multiclass logistic regression computes multiple z, one for each class k. We can further define the logit matrix $\mathbf{Z}$ as:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{z_{(0)}} & \mathbf{z_{(1)}} & \mathbf{z_{(2)}} & \cdots & \mathbf{z_{(K-1)}} \end{bmatrix}$$

$$\mathbf{Z} = \begin{bmatrix} \mathbf{X}\theta_{(0)} & \mathbf{X}\theta_{(1)} & \mathbf{X}\theta_{(2)} & \cdots & \mathbf{X}\theta_{(\mathbf{K-1})} \end{bmatrix}$$

$$\mathbf{Z} = \mathbf{X}\begin{bmatrix} \theta_{(0)} & \theta_{(1)} & \theta_{(2)} & \cdots & \theta_{(\mathbf{K-1})} \end{bmatrix}$$

$$\mathbf{Z} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} \theta_{0,0} & \theta_{0,1} & \cdots & \theta_{0,K-1} \\ \theta_{1,0} & \theta_{1,1} & \cdots & \theta_{1,K-1} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{n,0} & \theta_{n,1} & \cdots & \theta_{n,K-1} \end{bmatrix}$$

$$\mathbf{Z} = \mathbf{X}\boldsymbol{\Theta}$$

The elements of the matrix $\mathbf{Z}$ can be broken down as given below

$$\mathbf{Z} = \begin{bmatrix} z_{(0)}^{(1)} & z_{(1)}^{(1)} & z_{(2)}^{(1)} & \cdots & z_{(K-1)}^{(1)} \\ z_{(0)}^{(2)} & z_{(1)}^{(2)} & z_{(2)}^{(2)} & \cdots & z_{(K-1)}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ z_{(0)}^{(m)} & z_{(1)}^{(m)} & z_{(2)}^{(m)} & \cdots & z_{(K-1)}^{(m)} \end{bmatrix}$$

Where:

- m represents the data size (number of points = number of observations)
- K is the number of classes in the multiclass classification problem

As mentioned just now, there is more than one approach for the computation of probability vector and predicted label vector for each class. We shall cover the following approaches:

- **One-Vs-Rest (OVR) Approach**
- **Softmax Approach**

One-Vs-Rest (OVR) Approach for Computing Prediction

As the name suggests, in the **One-Vs-Rest (OVR) Approach**, each class is treated as a separate binary classification problem. You compare the probability for **one class VS. the rest**, and you do it for all classes.

For example, if you have 3 classes—A, B, and C, the model will create 3 binary classifiers:

- Classifier 1: Class A vs. All (A vs. B and C)
- Classifier 2: Class B vs. All (B vs. A and C)
- Classifier 3: Class C vs. All (C vs. A and B)

Then, when a new sample needs to be classified, each of the binary classifiers gives a prediction. The class that has the highest confidence or score is chosen as the final prediction.

The probability vector for each class is computed in the same manner as in binary logistic regression—using the logistic sigmoid function applied to the logit $\mathbf{z_{(k)}}$:

$$\mathbf{p_{(k)}} = \mathbf{p}(\mathbf{y} = k|\mathbf{X}) = \frac{1}{}$$

$$\mathbf{P_{(k)}} \quad \mathbf{P(} \quad \quad \quad 1 + e^{-\mathbf{z_{(k)}}}$$

for $k = 0, 1, 2, \ldots, K-1$.

Since $\mathbf{z_{(k)}} = \mathbf{X\Theta}$, this results in a probability matrix P:

$$\mathbf{P} = \frac{1}{1 + e^{-\mathbf{X\Theta}}}$$

The matrix P has m rows (one for each data point) and K columns (one for each class). The elements of the matrix $\mathbf{P}$ can be broken down as given below:

$$\mathbf{P} = \begin{bmatrix} \mathbf{p_{(0)}} & \mathbf{p_{(1)}} & \mathbf{p_{(2)}} & \cdots & \mathbf{p_{(K-1)}} \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} p_{(0)}^{(1)} & p_{(1)}^{(1)} & p_{(2)}^{(1)} & \cdots & p_{(K-1)}^{(1)} \\ p_{(0)}^{(2)} & p_{(1)}^{(2)} & p_{(2)}^{(2)} & \cdots & p_{(K-1)}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{(0)}^{(m)} & p_{(1)}^{(m)} & p_{(2)}^{(m)} & \cdots & p_{(K-1)}^{(m)} \end{bmatrix}$$

where:

$$p_{(k)}^{(i)} = p(y^{(i)} = k | x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})$$

- for $i = 1, 2, \ldots, m$
- and $k = 0, 1, \ldots, K-1$
- It corresponds to the probability of input data point i belongs to class k
- The sum of the probabilities across each row will be 1 after normalization (since the data point must belong to one of the classes)

The calculated probabilities $p_{(k)}^{(i)}$ are initially unnormalized, meaning that they might not sum to 1 for each data point. To ensure that the values represent a proper probability distribution, we normalize them:

$$\hat{p}_{(k)}^{(i)} = \frac{p_{(k)}^{(i)}}{p_{(0)}^{(i)} + p_{(1)}^{(i)} + \ldots + p_{(K-1)}^{(i)}} = \frac{p_{(k)}^{(i)}}{\sum_0^{K-1} p_{(k)}^{(i)}}$$

Once it is normalized, we may find the predicted label $\hat{y}^{(i)}$ for the i-th data point by finding the class with the highest probability:

$$\hat{y}^{(i)} = \arg\max_k \left(\hat{p}_{(k)}^{(i)} \right)$$

In other words, the class k with the highest normalized probability $\hat{y}^{(i)}$ is chosen as the predicted class for the i-th data point.

Softmax Approach for Computing Prediction

Unlike the OVA approach, the software approach directly extends the binary logistic regression to handle multiple classes. It uses the softmax activation function to calculate probabilities for all classes.

multiple classes. It uses the softmax activation function to calculate probabilities for all classes.

Just like in binary logistic regression, we start with a linear model for each class. For each class k, we compute a logit (raw class scores) and then convert them into probabilities. The logit is given by the function below:

$$z_k = \mathbf{X}\theta_\mathbf{k}$$

To compute the probability, we use the **softmax function**. The Softmax function takes a vector of logits (one for each class) and squashes them into a probability distribution, ensuring that:

- The probabilities for each class sum to 1
- The probabilities are between 0 and 1

We can then calculate a matrix S, which is the result of applying the exponentiation function to the logits (the linear combination of inputs and weights). Exponentiation is used to amplify the differences between logits, making higher values even higher and smaller values even smaller. Note: This is different from the logistic sigmoid function—there is no negative $(-)$ sign!

$$\mathbf{S} = e^{\mathbf{Z}} = e^{\mathbf{X\Theta}}$$

Here, $Z = \mathbf{X\Theta}$ represents the logits for each class, where:

- X is the feature matrix
- $\mathbf{\Theta}$ is the parameter matrix.

The elements of the matrix $\mathbf{S}$ can be broken down as given below:

$$\mathbf{S} = \begin{bmatrix} s_{(0)}^{(1)} & s_{(1)}^{(1)} & s_{(2)}^{(1)} & \cdots & s_{(K-1)}^{(1)} \\ s_{(0)}^{(2)} & s_{(1)}^{(2)} & s_{(2)}^{(2)} & \cdots & s_{(K-1)}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{(0)}^{(m)} & s_{(1)}^{(m)} & s_{(2)}^{(m)} & \cdots & s_{(K-1)}^{(m)} \end{bmatrix}$$

To convert the raw exponentiated values into the final probabilities, we use the **softmax function** as denoted below:

$$\hat{p}_{(k)}^{(i)} = \frac{s_{(k)}^{(i)}}{s_{(0)}^{(i)} + s_{(1)}^{(i)} + \ldots + s_{(K-1)}^{(i)}} = \frac{s_{(k)}^{(i)}}{\sum_0^{K-1} s_{(k)}^{(i)}}$$

where:

$$s_{(k)}^{(i)} = e^{z_{(k)}^{(i)}}$$

Each element in the matrix S from each row (each data point) will be divided by the sum of the exponentiated values of all classes for that data point. This normalizes the values, so that the sum of each row (data point) equals 1. This results in $\hat{p}_{(k)}^{(i)}$, which is the final probability of the class for the data point.

We may find the predicted label $\hat{y}^{(i)}$ for the i-th data point by finding the class with the highest probability:

$$\hat{y}^{(i)} = \arg\max_k \left(\hat{p}_{(k)}^{(i)}\right)$$

In other words, the class k with the highest normalized probability $\hat{y}^{(i)}$ is chosen as the predicted class for the i-th data point.

Differences between the Two Approaches

Modelling

- **OVR** treats each class as a separate binary classification problem, so you need to train K binary classifiers for K classes
- **Softmax** computes the probabilities for all classes simultaneously using the softmax activation function, directly extending the method from binary logistic regression

Computational Efficiency

- For **OVR**, training and prediction are both computationally expensive. For each class K, you need to train a classifier; and you have to compute the outputs of all the classifiers and compare the probabilities for each input
- For **Softmax**, both the training and prediction do not take much computing power

Imbalanced Data

- In **OVR**, each binary classifier might be influenced by the majority class
- **Softmax** is generally more robust as it considers all classes together during the training

Generalization

- In **OVR**, since each classifier is trained independently, the overall performance might suffer if the classifiers do not generalize well. The problem is that OVR does not explicitly model the relationships between classes
- **Softmax** considers all classes simultaneously and models the relationships between them, which is particularly useful when there are strong correlations between classes or when classes overlap

SUMMARY

- OVR: Simple and flexible, but inefficient and struggles with imbalanced classes
- Softmax: Efficient as only one classifier is needed, but it is less flexible compared to OVR, and sensitive to outliers

One-Vs-Rest (OVR) Approach

Create and Fitting the Logistic Regression Model Using `scikit-learn`

Creating the dataset:

```
X = np.array( [ [  5, 35],
                [ 10, 50],
                [ 25, 50],
```

```
                [ 25, 90],
                [ 75, 75],
                [ 90, 60],
                [100, 40],
                [ 40, 40],
                [ 40, 15],
                [ 80, 15]
              ] )

y = np.array( [0, 0, 0, 0, 1, 1, 1, 2, 2, 2] )

print(" X", X.shape, ":\n", X)
print("\n y", y.shape, ":\n", y)
```

```
 X (10, 2) :
 [[  5  35]
 [ 10  50]
 [ 25  50]
 [ 25  90]
 [ 75  75]
 [ 90  60]
 [100  40]
 [ 40  40]
 [ 40  15]
 [ 80  15]]

 y (10,) :
 [0 0 0 0 1 1 1 2 2 2]
```

Initializing and fitting the multiclass logistic regression model:

```
model = LogisticRegression(solver = 'liblinear', C = 20, random_state = 42,
                           tol = 1e-10, max_iter = 1000,
                           multi_class = "ovr"  # "auto", "ovr" or "multinomial"
                           )

print(f"{'Solver used':28}: {model.solver}")
print(f"{'Multi-class strategy used':28}: {model.multi_class}")

model.fit(X, y)
```

```
Solver used                 : liblinear
Multi-class strategy used   : ovr
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. Use OneVsRestClassifier(Lo
```

```
LogisticRegression(C=20, max_iter=1000, multi_class='ovr', random_state=42,
                   solver='liblinear', tol=1e-10)
```

Evaluating the model accuracy with accuracy metric:

```
mean_accuracy = model.score(X, y)
print(f"\n{'mean_accuracy':23} = {mean_accuracy}")

y_pred = model.predict(X)
mean_accuracy_eq = np.sum(y == y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")
```

```
mean_accuracy           = 1.0
mean_accuracy (manual)  = 1.0
y                       = [0 0 0 0 1 1 1 2 2 2]
y_pred                  = [0 0 0 0 1 1 1 2 2 2]
```

Display parameters of the model and verifying the probability of the dataset

```
print(f"{'n_iter_':16} = {model.n_iter_}")
print(f"{'intercept_':16} = {model.intercept_}")
print(f"{'coefficient_':16} =\n", model.coef_)

print(f"\nModel Classes: ", model.classes_)

p_pred = model.predict_proba(X)
print(f"\nPredicted OVR probabilities from the model:\n{p_pred}")
```

```
n_iter_          = [14 14 13]
intercept_       = [ 6.98625961e-04 -6.11800390e+00  3.32641515e+00]
coefficient_     =
 [[-0.56336056  0.40667259]
 [ 0.06551713  0.03293791]
 [ 0.00962878 -0.12228927]]

Model Classes:  [0 1 2]

Predicted OVR probabilities from the model:
[[7.70713137e-01 7.38980623e-03 2.21897056e-01]
 [9.21655491e-01 1.98550175e-02 5.84894914e-02]
 [8.86202091e-01 4.93300119e-02 6.44678969e-02]
 [8.46963403e-01 1.52538933e-01 4.97664265e-04]
 [1.00281015e-05 9.92455552e-01 7.53441940e-03]
 [4.22836813e-12 9.53780410e-01 4.62195896e-02]
 [3.28979994e-18 7.06597596e-01 2.93402404e-01]
 [5.59680520e-03 3.00063374e-01 6.94339821e-01]
 [7.97526744e-08 5.16955191e-02 9.48304401e-01]
 [9.09309428e-18 3.09260045e-01 6.90739955e-01]]
```

```
# Compute probability using equation

p_values = np.empty((X.shape[0], 0))
for i in range(3):  # There are 3 logit planes
  # We need to sum up the 2 features using np.sum()
  logit_value = model.intercept_[i] + np.sum(model.coef_[i] * X, axis = 1).reshape((-1, 1))
  p_value = 1 / (1 + np.exp(-logit_value)) # Logistic regression surface
```

```
  p_values = np.append(p_values, p_value, axis = 1)
print(f"Predicted OVR probabilities (not normalized):\n{p_values}") # This is the one-over-rest probab

# Normalize the values
p_values_sum = np.sum(p_values, axis = 1).reshape(-1, 1)
print(f"\nSum of the predicted OVR probabilities (normalization factor):\n{p_values_sum}")

p_pred_eq = p_values / p_values_sum
print(f"\nManually computed OVR probabilities (normalized):\n{p_pred_eq}")

p_pred_eq_sum = np.sum(p_pred_eq, axis = 1).reshape(-1, 1)
print(f"\nSum of the manually computed OVR probabilities (normalized):\n{p_pred_eq_sum}")

print("\nChecking that results from both methods are very close:\n", np.abs(p_pred - p_pred_eq) < 0.00
```

```
Predicted OVR probabilities (not normalized):
[[9.99988998e-01 9.58816526e-03 2.87908178e-01]
 [9.99999587e-01 2.15427668e-02 6.34613126e-02]
 [9.98073870e-01 5.55573005e-02 7.26061516e-02]
 [1.00000000e+00 1.80100973e-01 5.87586505e-04]
 [7.88217325e-06 7.80078521e-01 5.92211785e-03]
 [3.77962489e-12 8.52558732e-01 4.13144518e-02]
 [3.96726856e-18 8.52107265e-01 3.53822773e-01]
 [1.89465868e-03 1.01578964e-01 2.35051413e-01]
 [7.29395351e-08 4.72792563e-02 8.67292322e-01]
 [1.19230588e-17 4.05508356e-01 9.05712937e-01]]

Sum of the predicted OVR probabilities (normalization factor):
[[1.29748534]
 [1.08500367]
 [1.12623732]
 [1.18068856]
 [0.78600852]
 [0.89387318]
 [1.20593004]
 [0.33852504]
 [0.91457165]
 [1.31122129]]

Manually computed OVR probabilities (normalized):
[[7.70713137e-01 7.38980623e-03 2.21897056e-01]
 [9.21655491e-01 1.98550175e-02 5.84894914e-02]
 [8.86202091e-01 4.93300119e-02 6.44678969e-02]
 [8.46963403e-01 1.52538933e-01 4.97664265e-04]
 [1.00281015e-05 9.92455552e-01 7.53441940e-03]
 [4.22836813e-12 9.53780410e-01 4.62195896e-02]
 [3.28979994e-18 7.06597596e-01 2.93402404e-01]
 [5.59680520e-03 3.00063374e-01 6.94339821e-01]
 [7.97526744e-08 5.16955191e-02 9.48304401e-01]
 [9.09309428e-18 3.09260045e-01 6.90739955e-01]]

Sum of the manually computed OVR probabilities (normalized):
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

```
 [1.]]

 Checking that results from both methods are very close:
 [[ True  True  True]
 [ True  True  True]
 [ True  True  True]
 [ True  True  True]
 [ True  True  True]
 [ True  True  True]
 [ True  True  True]
 [ True  True  True]
 [ True  True  True]
```

Plotting the result of the model in 3D:

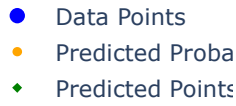
```
import numpy as np
import plotly.graph_objects as go
from sklearn.linear_model import LogisticRegression

def plot_3d_logistic_regression(X, y, model, class_idx=0):

    # Extract model coefficients and intercept
    coef0 = model.intercept_
    coef1 = model.coef_[:, 0]
    coef2 = model.coef_[:, 1]

    # Generate grid for the plot
    logit_X1, logit_X2 = np.meshgrid(np.linspace(0, 100, 100),
                                     np.linspace(0, 100, 100))

    # Create empty arrays to store the logit values for the selected class
    logit_y = coef0[class_idx] + coef1[class_idx] * logit_X1 + coef2[class_idx] * logit_X2  # Logit co

    # Create 3D scatter for original data points
    points = go.Scatter3d(
        x=X[:, 0],
        y=X[:, 1],
        z=y.reshape(-1),
        mode='markers',
        marker={"size": 8, "color": "blue"},
        name='Data Points'
    )

    # Predict probabilities for the data points
    y_pred = model.predict(X)
    p_pred = model.predict_proba(X)

    # Scatter for predicted points
    points_pred = go.Scatter3d(
        x=X[:, 0],
        y=X[:, 1],
        z=y_pred.reshape(-1),
        mode='markers',
        marker={"size": 4, "color": "green", "symbol": "diamond"},
        name='Predicted Points'
    )
```

```
# Scatter for predicted probabilities for the selected class
points_p_pred = go.Scatter3d(
    x=X[:, 0],
    y=X[:, 1],
    z=p_pred[:, class_idx],  # Probability for the selected class
    mode='markers',
    marker={"size": 5, "color": "orange"},
    name='Predicted Probability'
)

# Surface plot for logistic regression decision plane
surface_logit_plane = go.Surface(
    x=logit_X1, y=logit_X2, z=logit_y,
    opacity=0.4, colorscale='Blues',
    showscale=False,
    name=f'Class {class_idx} Logit Plane'
)

# Convert logits to probabilities
p_y = 1 / (1 + np.exp(-logit_y))  # Predicted probability for the selected class
surface_p_plane = go.Surface(
    x=logit_X1, y=logit_X2, z=p_y,
    opacity=0.5, colorscale='YlOrRd',
    showscale=False,
    name=f'Class {class_idx} Predicted Probability'
)

# Horizontal plane at y = 0.5 to separate the classes
horizontal_plane = go.Surface(
    x=logit_X1, y=logit_X2, z=0.5 * np.ones_like(logit_X1),
    opacity=0.3, colorscale='YlOrBr',
    showscale=False,
    name='Plane y = 0.5'
)

# Create the figure with all the components
fig = go.Figure(data=[horizontal_plane, surface_logit_plane, surface_p_plane,
                      points, points_p_pred, points_pred])

# Update layout with labels and camera position
title_text = "$P(y = 1 | X1, X2) = \\frac{1}{1 + e^{-(%.3f + (%.3f) X1 + (%.3f) X2 )}}$" % (coef0[
fig.update_layout(
    scene={
        "xaxis_title": "X1",
        "yaxis_title": "X2",
        "zaxis_title": "y",
        "zaxis_range": [-0.1, 1.1],
        "zaxis_dtick": 0.1
    },
    width=800,
    height=600,
    scene_camera={"eye": {"x": -1, "y": -1.5, "z": 1.5}},
    title={"text": title_text,
           "font": {"size": 20},
           "x": 0.5,
```

```
                "y": 0.9,
                "xanchor": "center",
                "yanchor": "top"}
    )

    fig.show()

print("CLASS 0")
plot_3d_logistic_regression(X, y, model, class_idx=0)
print("\nCLASS 1")
plot_3d_logistic_regression(X, y, model, class_idx=1)
print("\nCLASS 2")
plot_3d_logistic_regression(X, y, model, class_idx=2)
```

CLASS 0

$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(0.001+(-0.563)X1+(0.407)X2)}}$$

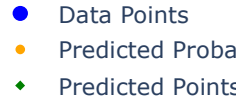


CLASS 1

$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(-6.118+(0.066)X1+(0.033)X2)}}$$

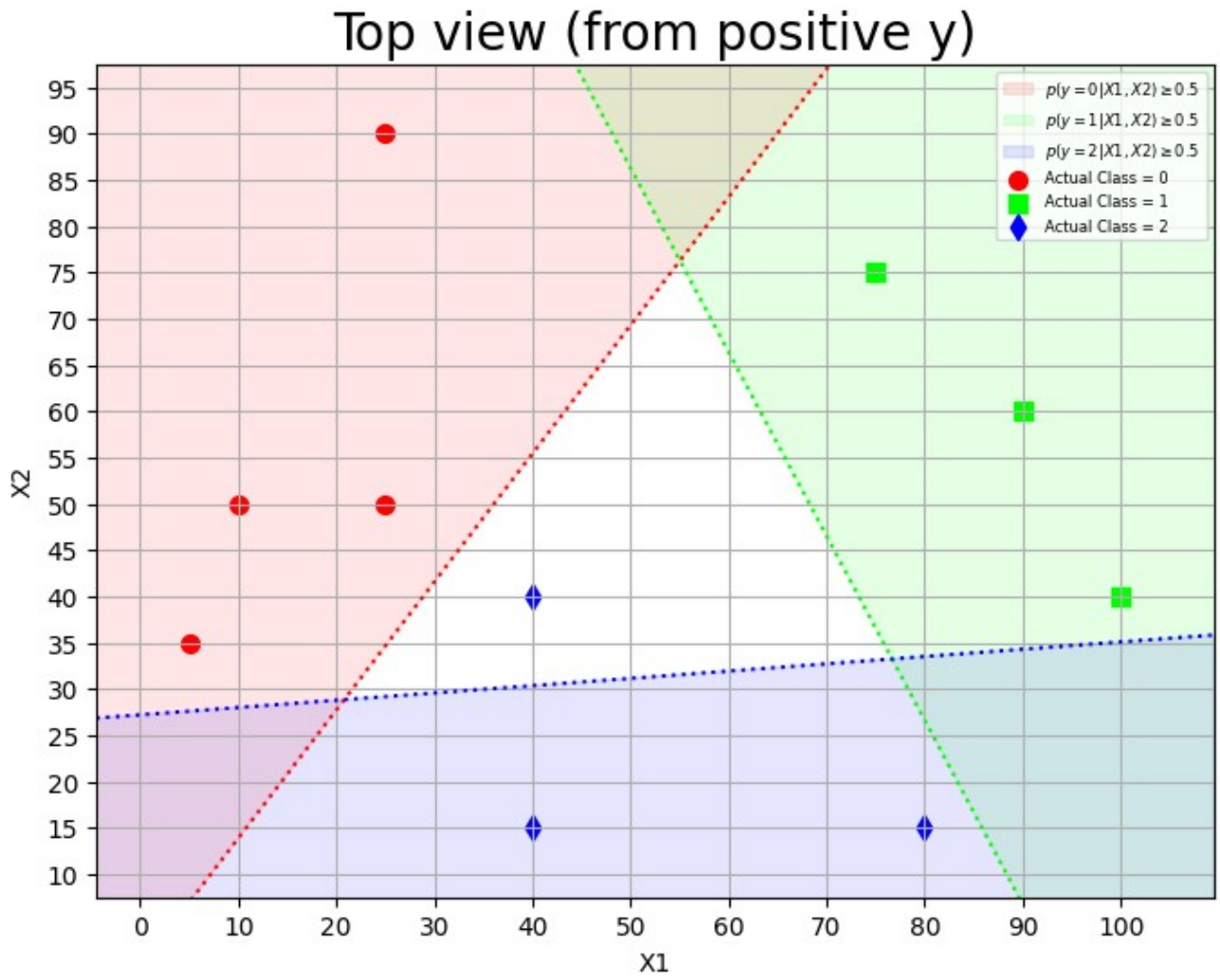
CLASS 2

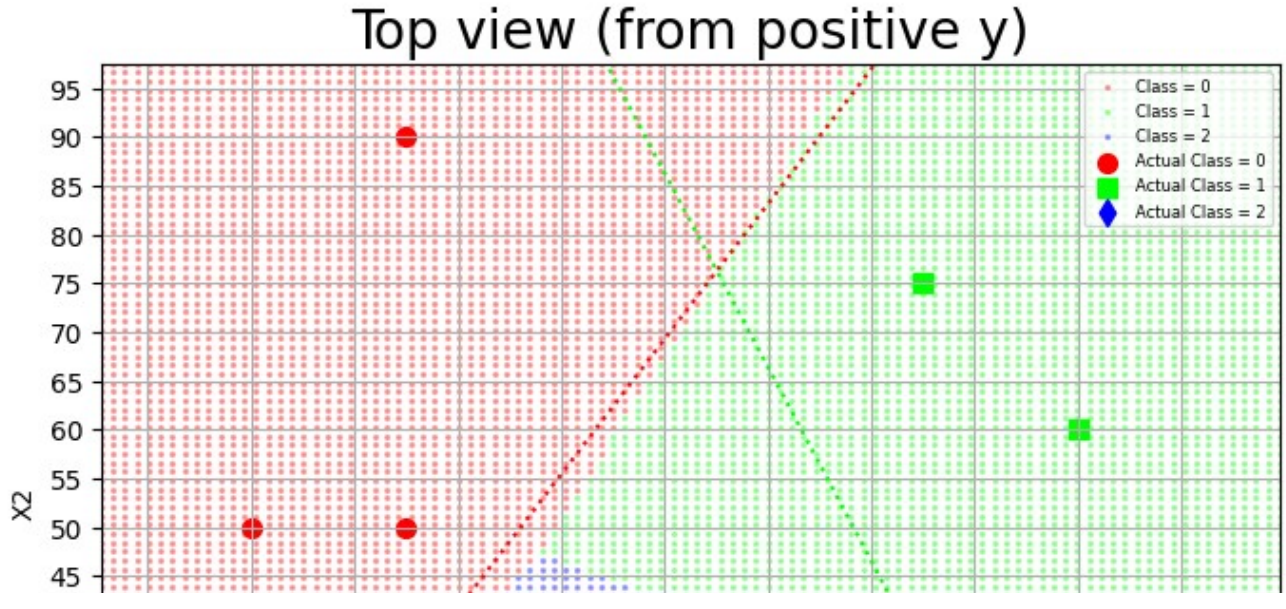
$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(3.326+(0.010)X1+(-0.122)X2)}}$$

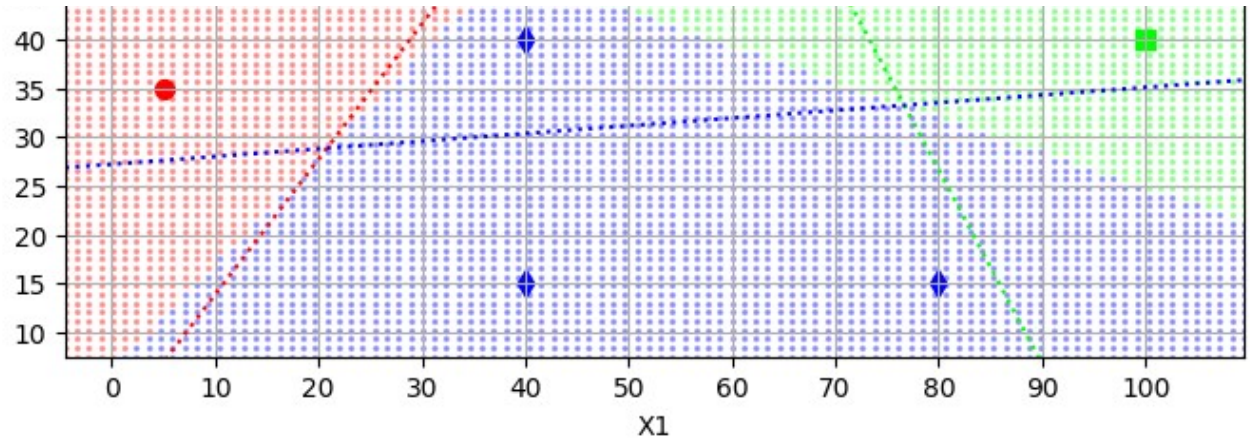


Plotting the result of the model in 2D:

```
plot_logistic_regression_top_view(X, y, model, plot_classes_all_points=False)
print()
plot_logistic_regression_top_view(X, y, model, plot_logit_region=False)
```



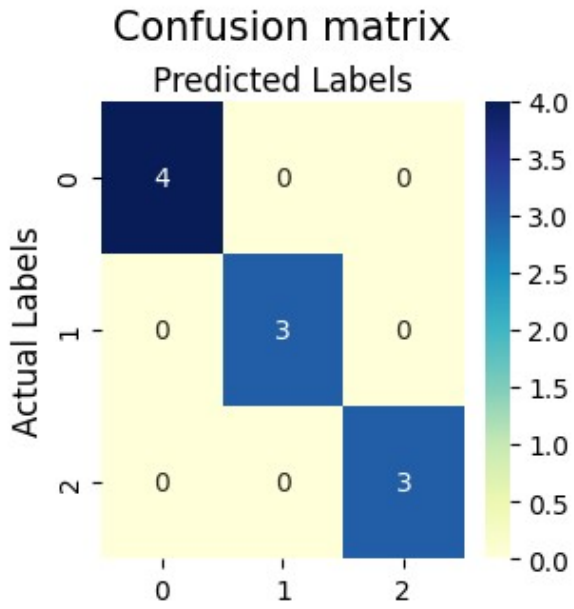




Confusion Matrix:

```
c_matrix = confusion_matrix(y, y_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
[[4 0 0]
 [0 3 0]
 [0 0 3]]
```

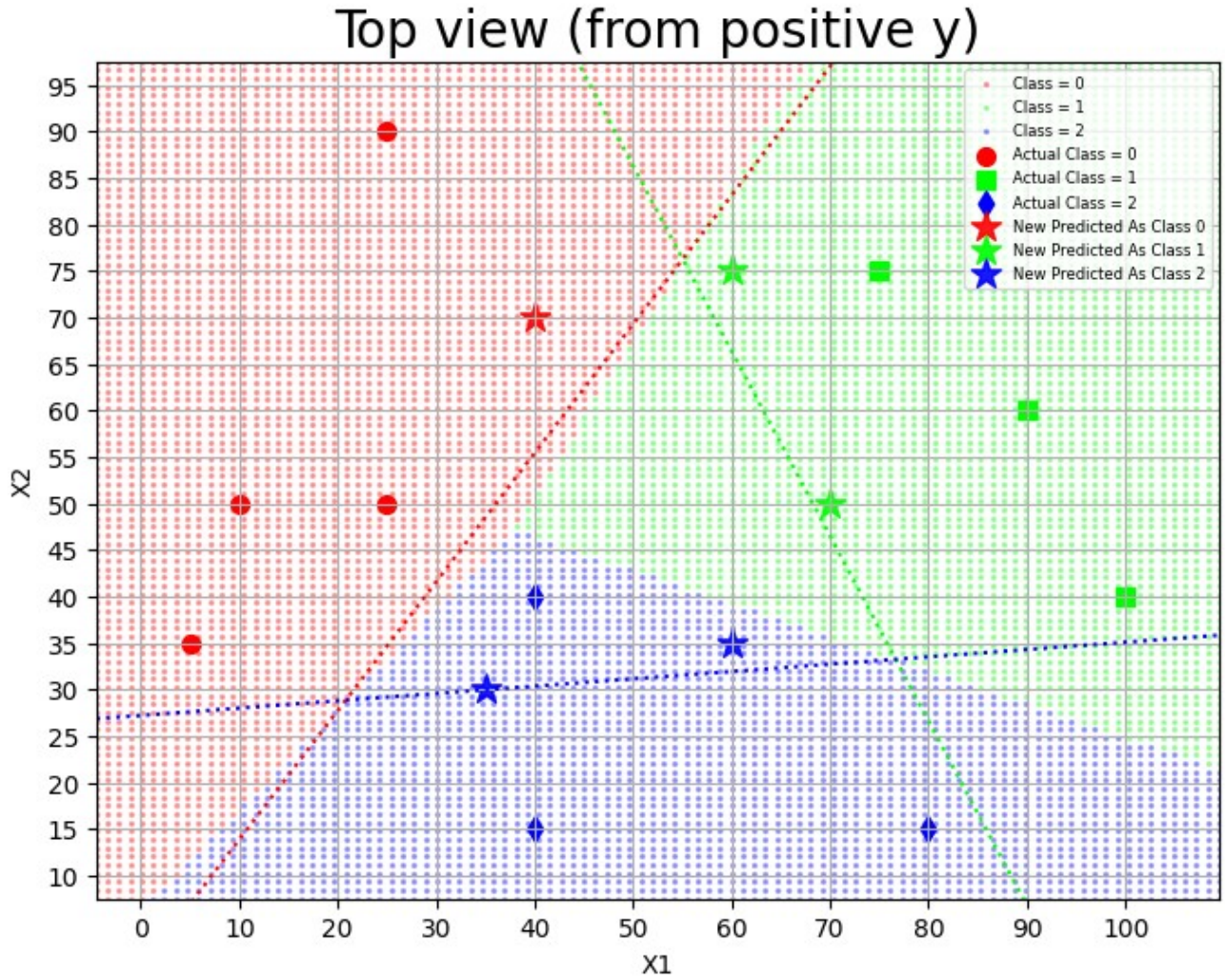


Predict New Data Points

```
X_new = np.array( [ [ 35, 30],
                    [ 40, 70],
                    [ 70, 50],
                    [ 60, 35],
                    [ 60, 75]
                  ] )
print(" X_new", X_new.shape, ":\n", X)

y_pred_new = model.predict(X_new)
```

```
print("\ny_pred_new =\n", y_pred_new)

plot_logistic_regression_top_view(X, y, model, X_new = X_new,
                                  plot_logit_region=False)
```

```
X_new (5, 2) :
[[  5  35]
[ 10  50]
[ 25  50]
[ 25  90]
[ 75  75]
[ 90  60]
[100  40]
[ 40  40]
[ 40  15]
[ 80  15]]

y_pred_new =
 [2 0 1 2 1]
```



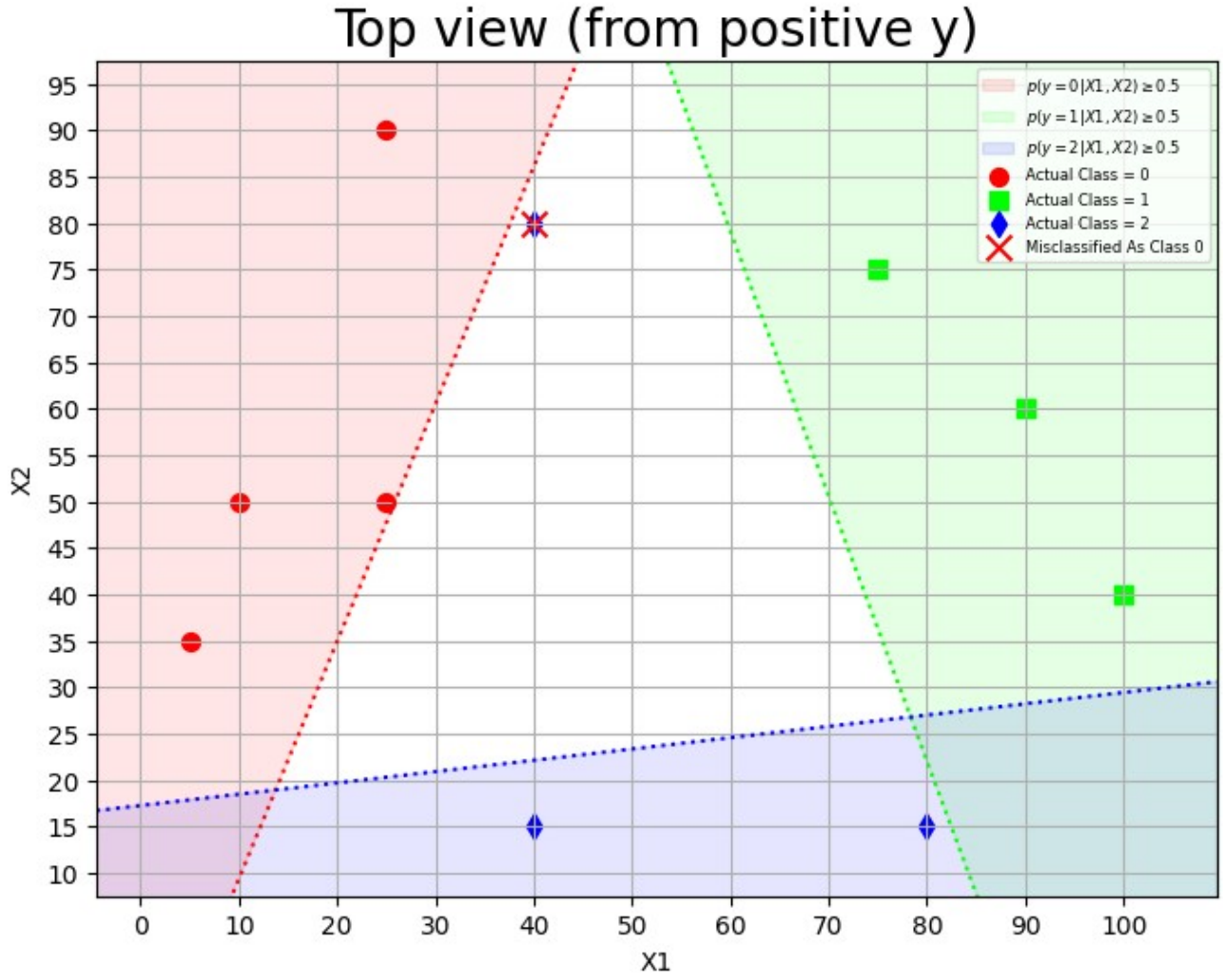
TO DO: Let's Experiment!

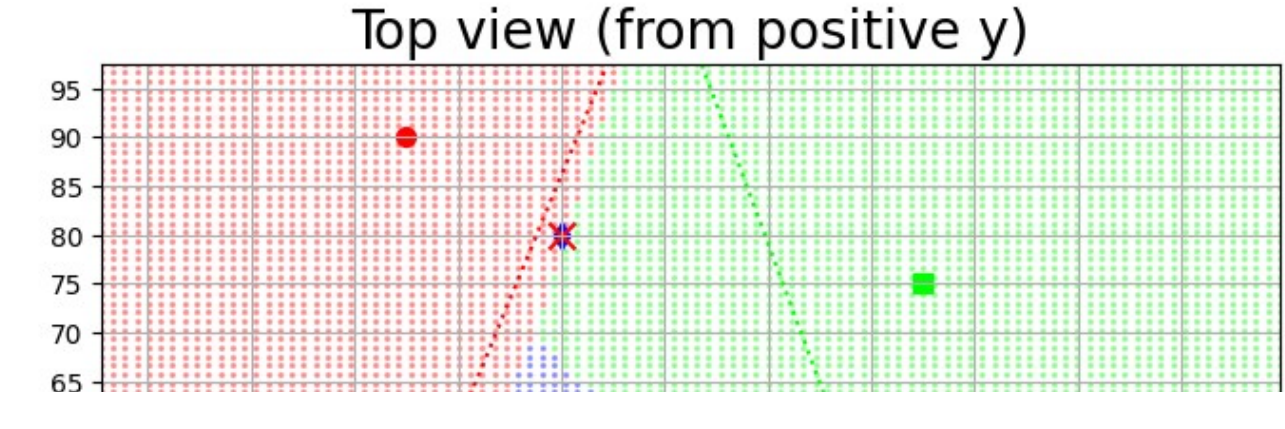
Try changing some points and see the results.

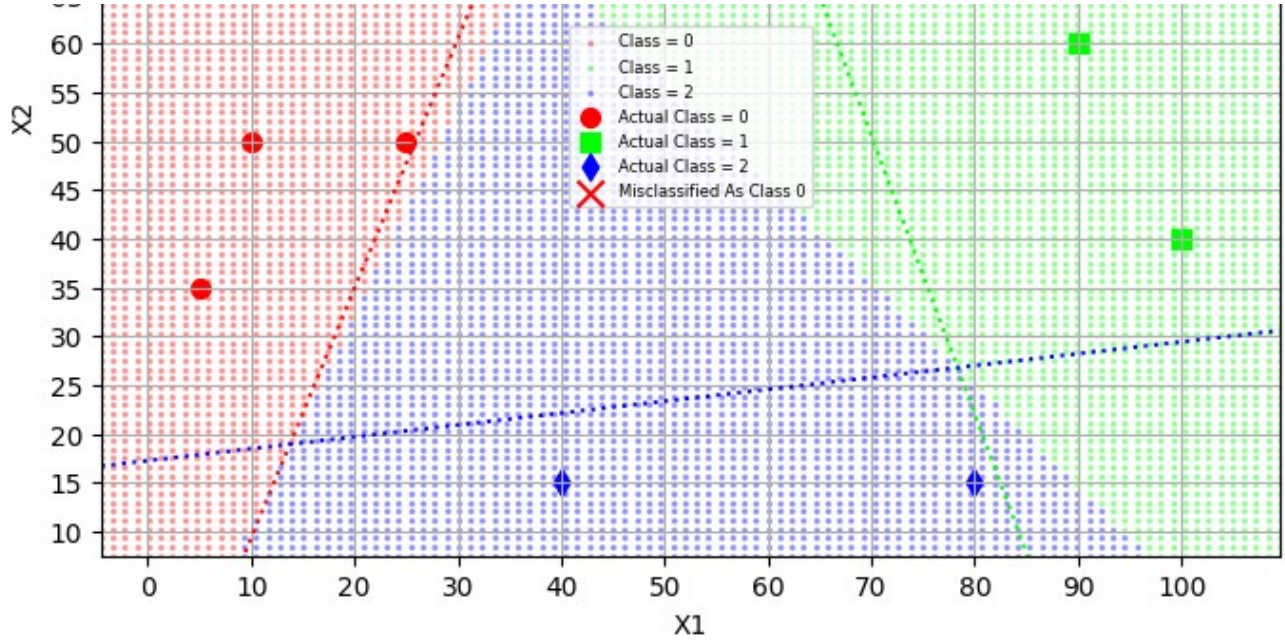
```
# TO DO: Try changing some points and see the result
X[7] = [40, 80]
```

```
X[7] = [40, 80]

model.fit(X, y)

mean_accuracy = model.score(X, y)
print(f"mean_accuracy = {mean_accuracy}")

plot_logistic_regression_top_view(X, y, model, plot_classes_all_points=False)
print()
plot_logistic_regression_top_view(X, y, model, plot_logit_region=False)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. Use OneVsRestClassifier(Lo

mean_accuracy = 0.9
```

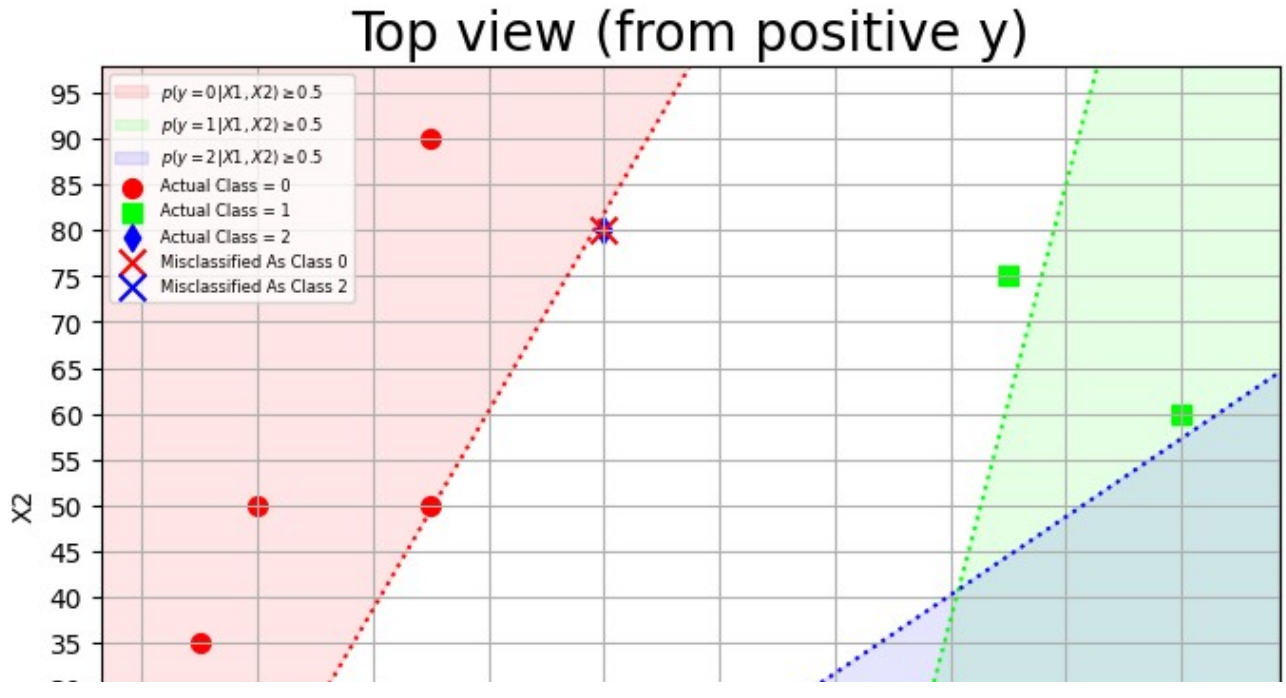


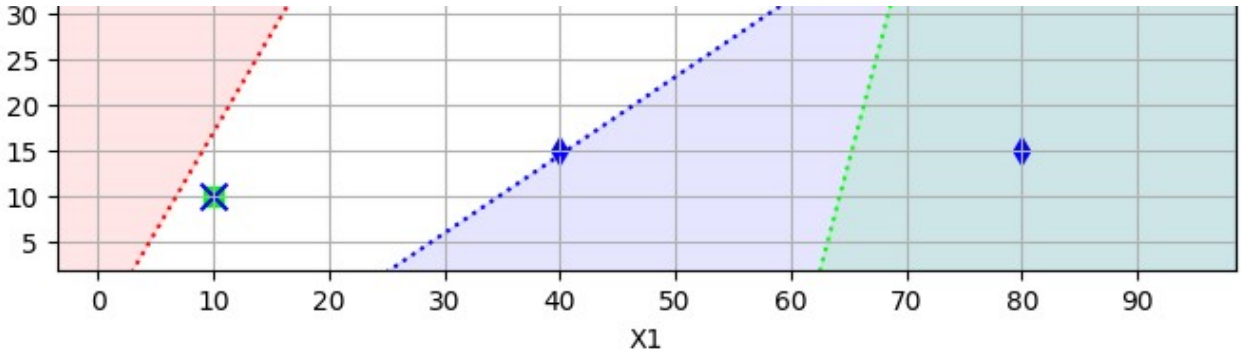


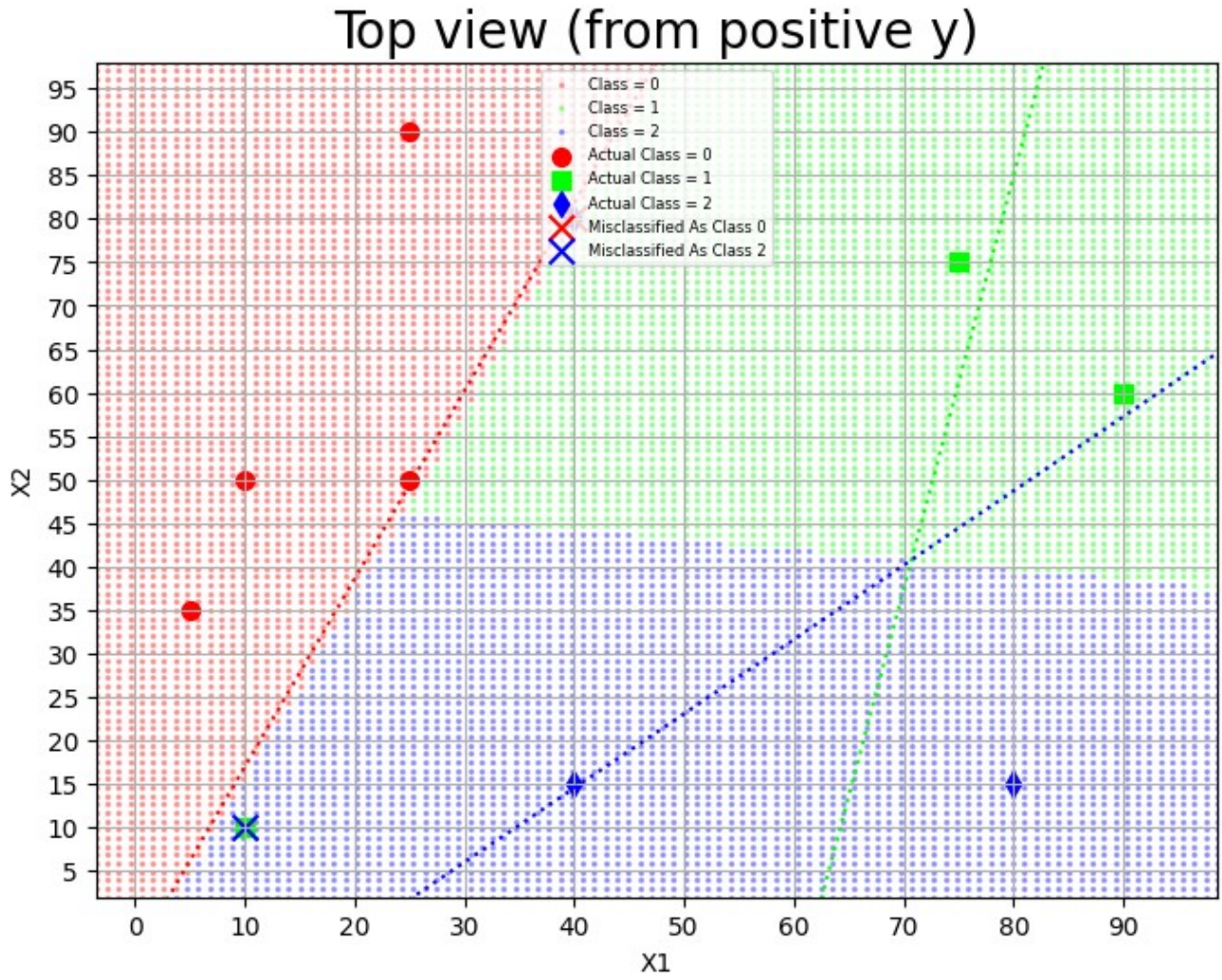



```
# TO DO: Try changing some points and see the result
X[6] = [10, 10]

model.fit(X, y)

mean_accuracy = model.score(X, y)
print(f"mean_accuracy = {mean_accuracy}")

plot_logistic_regression_top_view(X, y, model, plot_classes_all_points=False)
print()
plot_logistic_regression_top_view(X, y, model, plot_logit_region=False)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. Use OneVsRestClassifier(Lo

mean_accuracy = 0.8
```







Multinomial Approach Using Softmax

Create and Fitting the Logistic Regression Model Using `scikit-learn`

Creating the dataset:

```
X = np.array( [ [  5, 35],
                [ 10, 50],
                [ 25, 50],
                [ 25, 90],
```

```
                [ 75, 75],
                [ 90, 60],
                [100, 40],
                [ 40, 40],
                [ 40, 15],
                [ 80, 15]
              ] )

y = np.array( [0, 0, 0, 0, 1, 1, 1, 2, 2, 2] )

print(" X", X.shape, ":\n", X)
print("\n y", y.shape, ":\n", y)
```

```
 X (10, 2) :
 [[  5  35]
 [ 10  50]
 [ 25  50]
 [ 25  90]
 [ 75  75]
 [ 90  60]
 [100  40]
 [ 40  40]
 [ 40  15]
 [ 80  15]]

 y (10,) :
 [0 0 0 0 1 1 1 2 2 2]
```

Initializing and fitting the multiclass logistic regression model:

```
# use default solver, cannot use "liblinear"
# use multinomial option for multi_class
model = LogisticRegression(C = 20, random_state = 42,
                           tol = 1e-10, max_iter = 1000,
                           multi_class = "multinomial"  # "auto", "ovr" or "multinomial"
                           )

print(f"{'Solver used':28}: {model.solver}")
print(f"{'Multi-class strategy used':28}: {model.multi_class}")

model.fit(X, y)
```

```
Solver used                 : lbfgs
Multi-class strategy used   : multinomial
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. From then on, it will alwa
```

```
LogisticRegression(C=20, max_iter=1000, multi_class='multinomial',
                   random_state=42, tol=1e-10)
```

Evaluating the model accuracy with accuracy metric:

```
mean_accuracy = model.score(X, y)
print(f"\n{'mean_accuracy':23} = {mean_accuracy}")

y_pred = model.predict(X)
mean_accuracy_eq = np.sum(y == y_pred) / len(y)
print(f"{'mean_accuracy (manual)':23} = {mean_accuracy_eq}")

print(f"{'y':23} = {y}")

print(f"{'y_pred':23} = {y_pred}")
```

```
mean_accuracy           = 1.0
mean_accuracy (manual)  = 1.0
y                       = [0 0 0 0 1 1 1 2 2 2]
y_pred                  = [0 0 0 0 1 1 1 2 2 2]
```

Display parameters of the model and verifying the probability of the dataset

```
print(f"{'n_iter_':16} = {model.n_iter_}")
print(f"{'intercept_':16} = {model.intercept_}")
print(f"{'coefficient_':16} =\n", model.coef_)

print(f"\nModel Classes: ", model.classes_)

p_pred = model.predict_proba(X)
print(f"\nPredicted OVR probabilities from the model:\n{p_pred}")
```

```
n_iter_          = [80]
intercept_       = [  7.05760184 -21.77232965  14.71472781]
coefficient_     =
 [[-0.43994053  0.22274504]
 [ 0.35445805  0.10526885]
 [ 0.08548248 -0.32801389]]

Model Classes:  [0 1 2]

Predicted OVR probabilities from the model:
[[9.99875654e-01 2.62184105e-13 1.24345611e-04]
 [9.99999556e-01 2.38993387e-12 4.44377873e-07]
 [9.98824565e-01 3.57206295e-07 1.17507773e-03]
 [9.99999997e-01 3.25581341e-09 3.18357356e-13]
 [2.96418643e-10 9.99999906e-01 9.37798879e-08]
 [3.40073170e-16 9.99998897e-01 1.10274415e-06]
 [1.15072059e-20 9.99565903e-01 4.34096909e-04]
 [1.30022712e-03 2.25260745e-04 9.98474512e-01]
 [1.36424753e-09 4.45698284e-09 9.99999994e-01]
 [1.01687932e-18 2.09668909e-04 9.99790331e-01]]
```

```
# Compute probability using equation - Using Softmax Method

logit_values = np.empty((X.shape[0], 0))
for i in range(3):  # There are 3 logit planes
  # We need to sum up over the 2 features using np.sum()
  logit_value = model.intercept_[i] + np.sum(model.coef_[i] * X, axis = 1).reshape((-1, 1))
```

```
  # NOT USING THIS: p = 1 / (1 + np.exp(-logit)) # Logistic regression surface
  logit_values = np.append(logit_values, logit_value, axis = 1)
print(f"All logit values:\n{logit_values}")

# Below is Softmax Method
e_values = np.exp(logit_values)
print(f"\nExponentiated logit values (not normalized):\n{e_values}")

e_values_sum = np.sum(e_values, axis = 1).reshape(-1, 1)
print(f"\nSum of the exponentiated logit values (normalization factor):\n{e_values_sum}")

p_pred_eq = e_values / e_values_sum
print(f"\nManually computed Softmax probabilities (normalized):\n{p_pred_eq}")

p_pred_eq_sum = np.sum(p_pred_eq, axis = 1).reshape(-1, 1)
print(f"\nSum of the manually computed Softmax probabilities (normalized):\n{p_pred_eq_sum}")

print("\nChecking that results from both methods are very close:\n", np.abs(p_pred - p_pred_eq) < 0.00
```

```
All logit values:
[[ 12.65397548 -16.31562962   3.66165415]
 [ 13.79544837 -12.9643066   -0.83114175]
 [  7.19634037  -7.64743587   0.45109552]
 [ 16.10614187  -3.43668183 -12.66946001]
 [ -9.23206037  12.70718787  -3.47512746]
 [-19.17234393  16.44502584   2.72731814]
 [-28.02665001  17.88422931  10.14242075]
 [ -1.63021801  -3.38325364   5.01347167]
 [ -7.19884394  -6.01497491  13.21381887]
 [-24.79646528   8.16334705  16.63311826]]

Exponentiated logit values (not normalized):
[[3.13005324e+05 8.20752263e-08 3.89256784e+01]
 [9.80137721e+05 2.34246537e-06 4.35551710e-01]
 [1.33453790e+03 4.77266336e-04 1.57003124e+00]
 [9.88117379e+06 3.21712582e-02 3.14574437e-06]
 [9.78514187e-05 3.30112197e+05 3.09578878e-02]
 [4.71581573e-09 1.38670467e+07 1.52918214e+01]
 [6.73256498e-13 5.84819848e+07 2.53978740e+04]
 [1.95886865e-01 3.39368566e-02 1.50426060e+02]
 [7.47449405e-04 2.44190962e-03 5.47884005e+05]
 [1.70228377e-11 3.50991484e+03 1.67367634e+07]]

Sum of the exponentiated logit values (normalization factor):
[[3.13044249e+05]
 [9.80138157e+05]
 [1.33610841e+03]
 [9.88117383e+06]
 [3.30112228e+05]
 [1.38670620e+07]
 [5.85073827e+07]
 [1.50655884e+02]
 [5.47884008e+05]
 [1.67402733e+07]]

Manually computed Softmax probabilities (normalized):
[[9.99875654e-01 2.62184105e-13 1.24345611e-04]
 [9.99999556e-01 2.38993387e-12 4.44377873e-07]
 [9.98824565e-01 3.57206295e-07 1.17507773e-03]
 [9.99999997e-01 3.25581341e-09 3.18357356e-13]
```

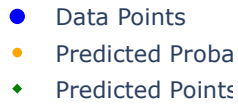
```
 [2.96418643e-10 9.99999906e-01 9.37798879e-08]
 [3.40073170e-16 9.99998897e-01 1.10274415e-06]
 [1.15072059e-20 9.99565903e-01 4.34096909e-04]
 [1.30022712e-03 2.25260745e-04 9.98474512e-01]
 [1.36424753e-09 4.45698284e-09 9.99999994e-01]
 [1.01687932e-18 2.09668909e-04 9.99790331e-01]]

Sum of the manually computed Softmax probabilities (normalized):
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
```

Plotting the result of the model in 3D:

```
print("CLASS 0")
plot_3d_logistic_regression(X, y, model, class_idx=0)
print("\nCLASS 1")
plot_3d_logistic_regression(X, y, model, class_idx=1)
print("\nCLASS 2")
plot_3d_logistic_regression(X, y, model, class_idx=2)
```

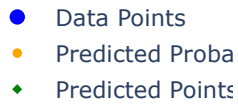
```
CLASS 0
```

$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(7.058+(-0.440)X1+(0.223)X2)}}$$



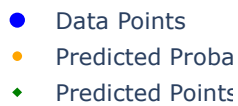
CLASS 1

$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(-21.772+(0.354)X1+(0.105)X2)}}$$



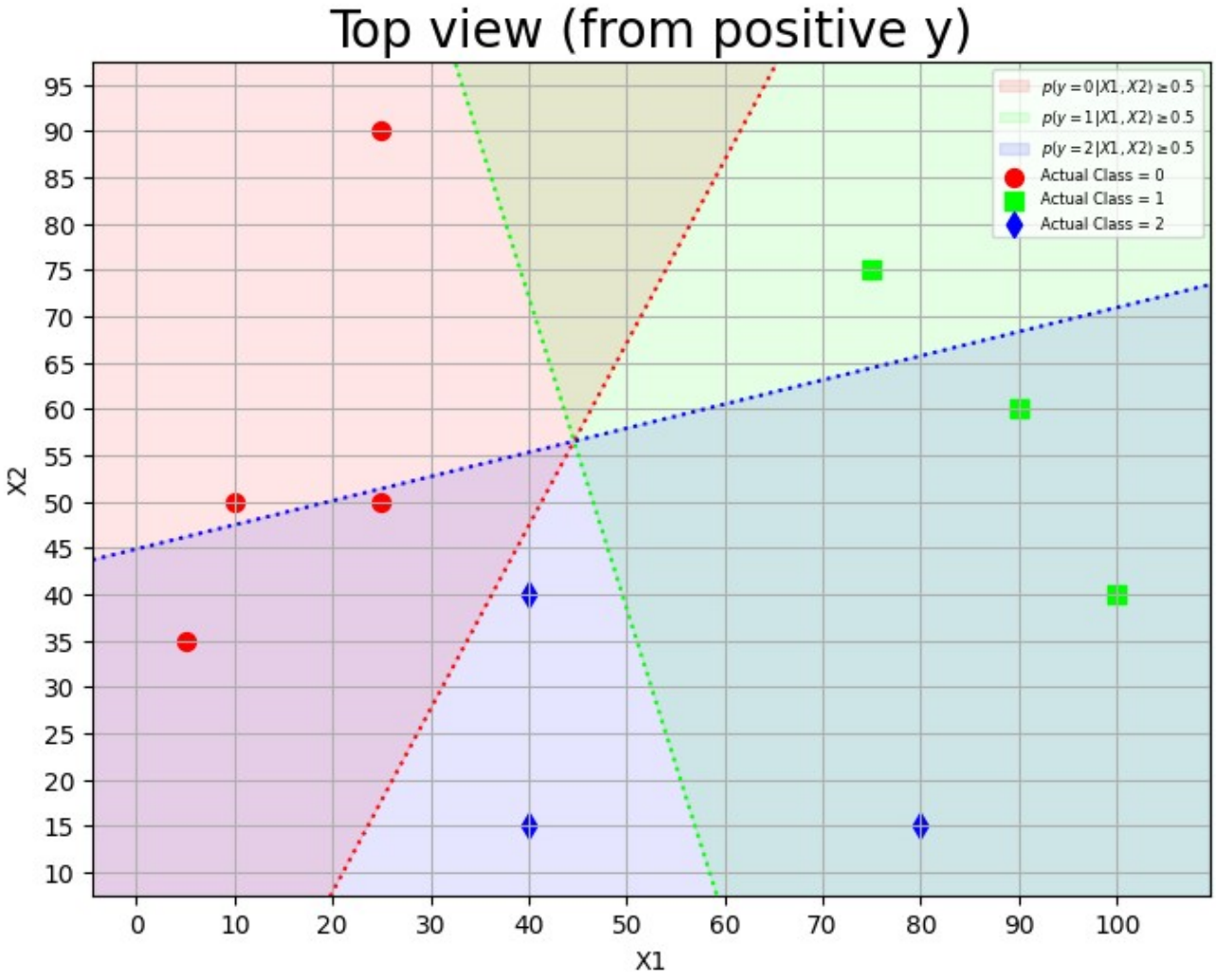
CLASS 2

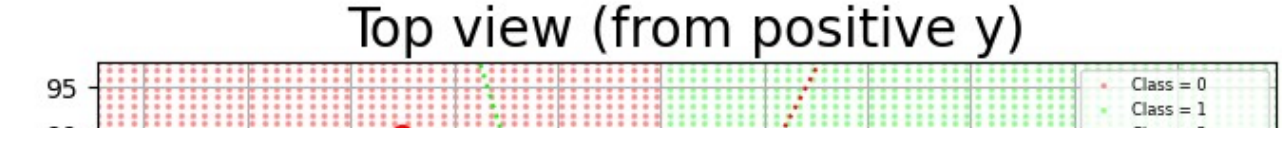
$$P(y = 1|X1, X2) = \frac{1}{1+e^{-(14.715+(0.085)X1+(-0.328)X2)}}$$

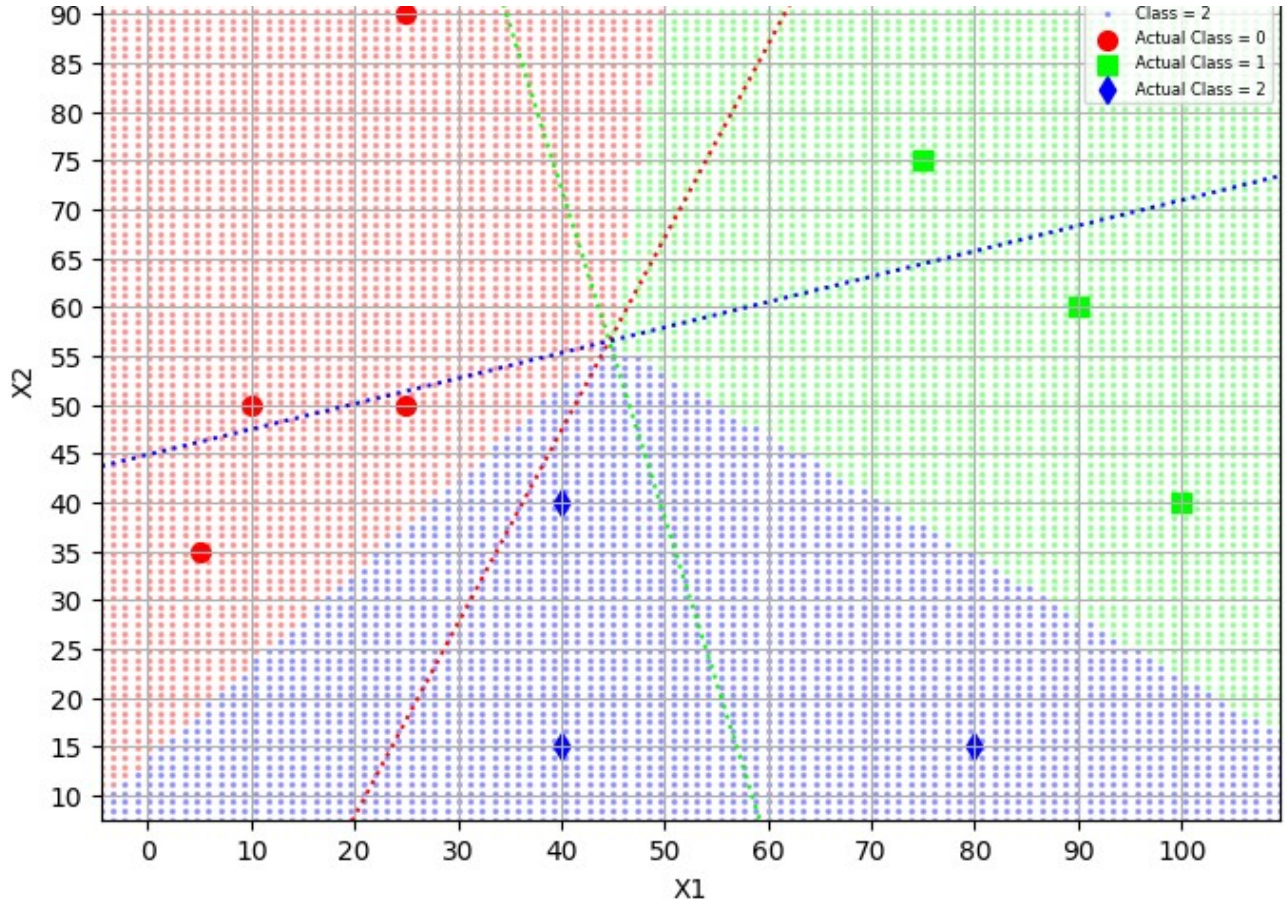


Plotting the result of the model in 2D:

```
plot_logistic_regression_top_view(X, y, model, plot_classes_all_points=False)
print()
plot_logistic_regression_top_view(X, y, model, plot_logit_region=False)
```



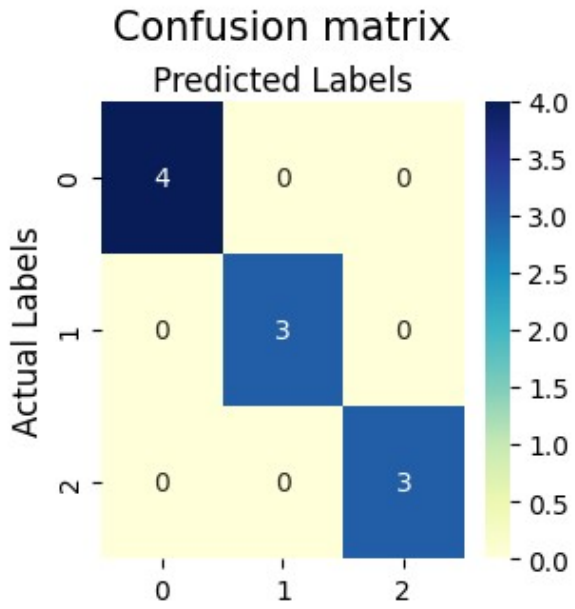




Confusion Matrix:

```
c_matrix = confusion_matrix(y, y_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
[[4 0 0]
 [0 3 0]
 [0 0 3]]
```

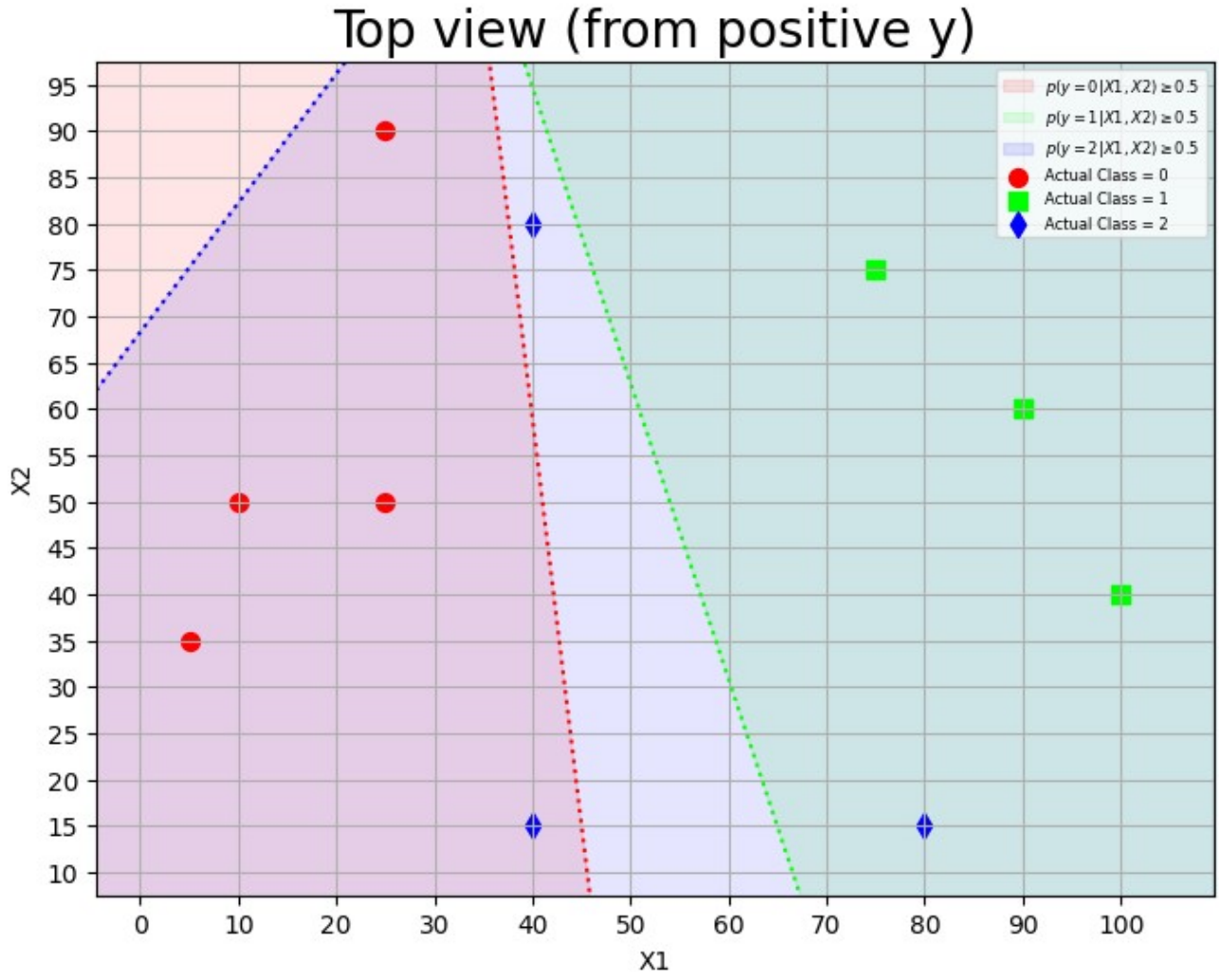


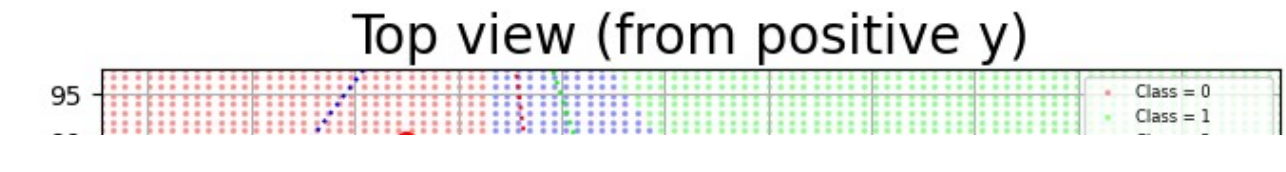
TO DO: Let's Experiment!

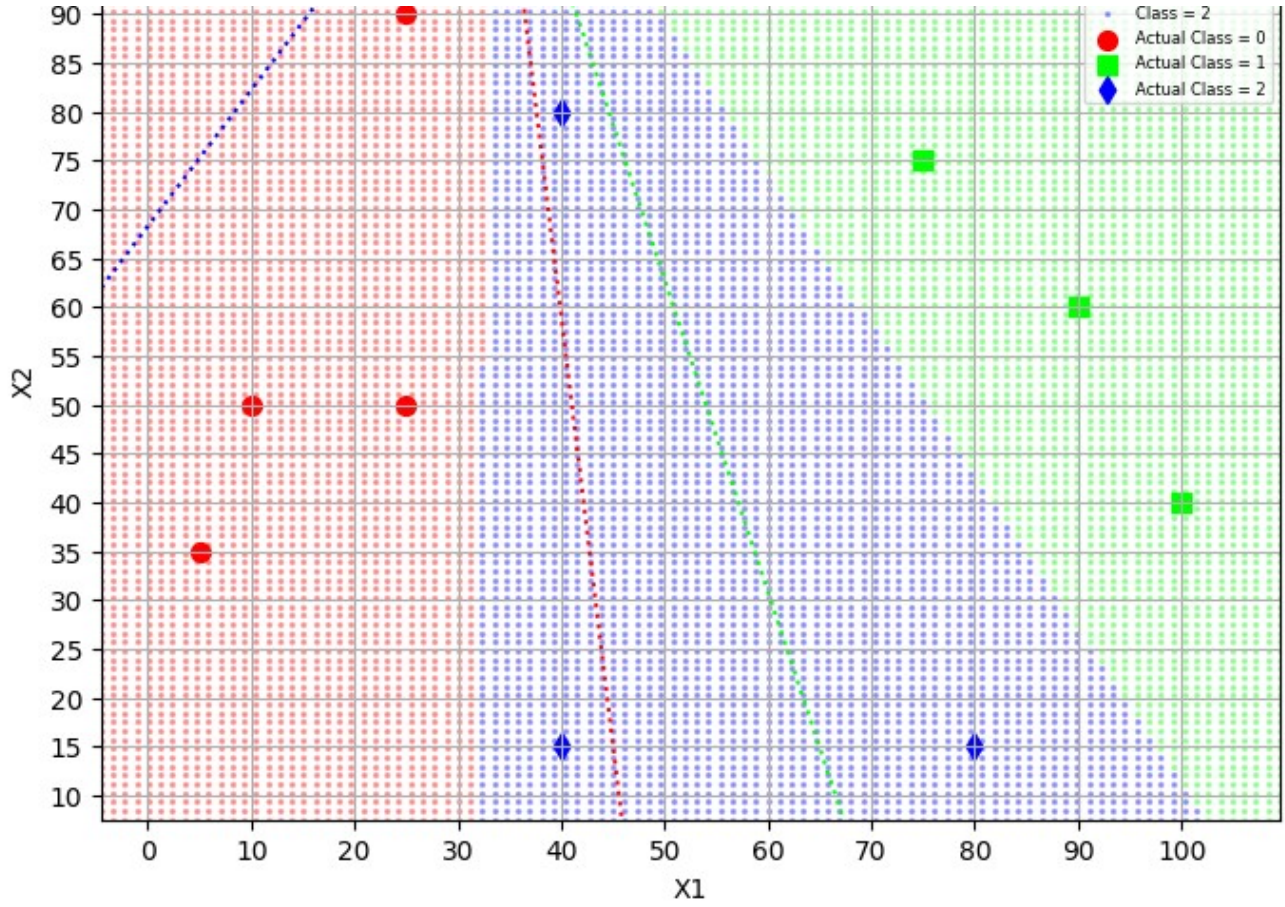
Try changing some points and see the results.

```
# TO DO: Try changing some points and see the result
X[7] = [40, 80]

model.fit(X, y)

mean_accuracy = model.score(X, y)
print(f"mean_accuracy = {mean_accuracy}")

plot_logistic_regression_top_view(X, y, model, plot_classes_all_points=False)
print()
plot_logistic_regression_top_view(X, y, model, plot_logit_region=False)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. From then on, it will alwa

mean_accuracy = 1.0
```

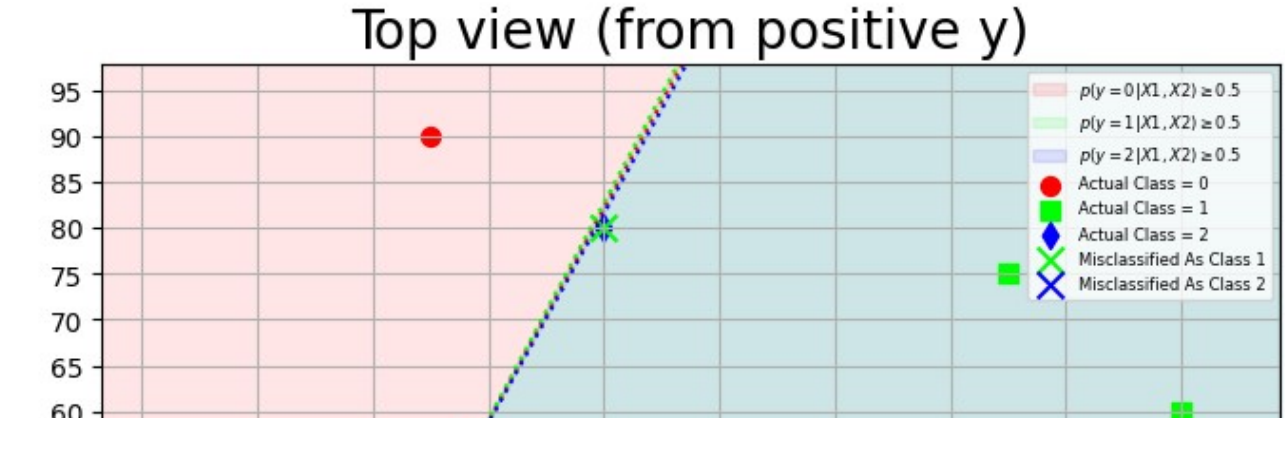


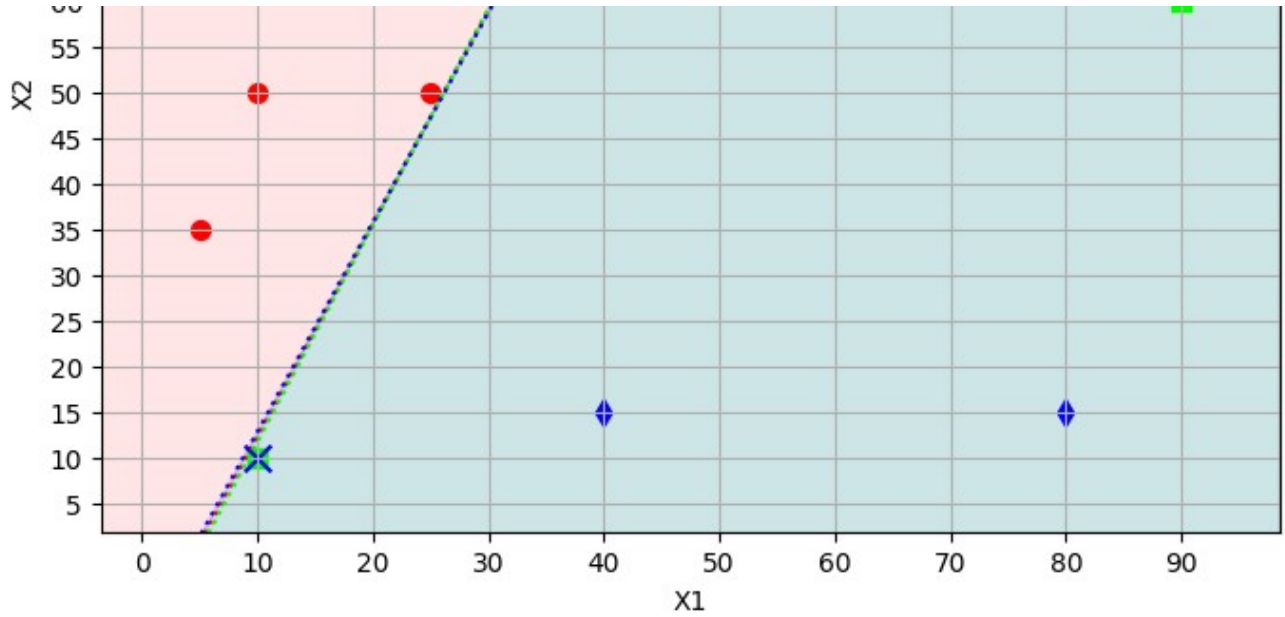


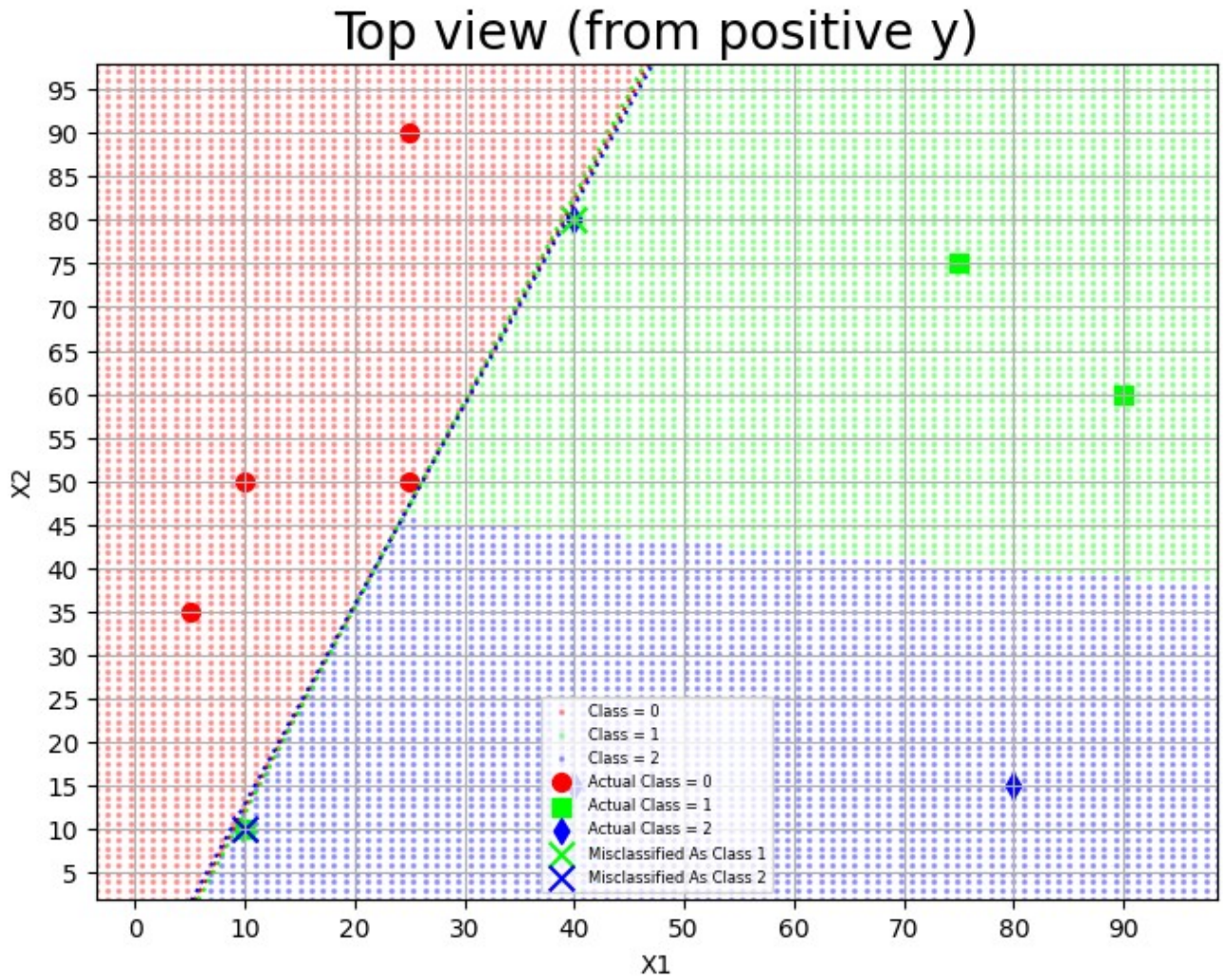



```
# TO DO: Try changing some points and see the result
X[6] = [10, 10]

model.fit(X, y)

mean_accuracy = model.score(X, y)
print(f"mean_accuracy = {mean_accuracy}")

plot_logistic_regression_top_view(X, y, model, plot_classes_all_points=False)
print()
plot_logistic_regression_top_view(X, y, model, plot_logit_region=False)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. From then on, it will alwa

mean_accuracy = 0.8
```







Real World Logistic Regression Application: Handwriting Recognition

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
```

```
from sklearn.metrics import classification_report, confusion_matrix

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
X, y = load_digits(return_X_y = True)

print(X)
print(y)
print(X.shape)
print(y.shape)
```

```
[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]]
[0 1 2 ... 8 9 8]
(1797, 64)
(1797,)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.2,
                                                    random_state = 0)

print(X_train)
print(X_test)
print(y_train)
print(y_test)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
[[ 0.  0.  0. ... 16. 16.  6.]
 [ 0.  3. 12. ... 16.  2.  0.]
 [ 0.  1. 10. ...  0.  0.  0.]
 ...
 [ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  4. ...  0.  0.  0.]
 [ 0.  0.  6. ... 11.  0.  0.]]
[[ 0.  0. 11. ... 13. 16.  8.]
 [ 0.  1. 15. ...  1.  0.  0.]
 [ 0.  2. 13. ... 16. 16.  3.]
 ...
 [ 0.  1.  9. ...  4.  0.  0.]
 [ 0.  0.  0. ... 15.  2.  0.]
 [ 0.  0.  0. ... 12.  0.  0.]]
[6 5 3 ... 7 7 8]
[2 8 2 6 6 7 1 9 8 5 2 8 6 6 6 6 1 0 5 8 8 7 8 4 7 5 4 9 2 9 4 7 6 8 9 4 3
 1 0 1 8 6 7 7 1 0 7 6 2 1 9 6 7 9 0 0 5 1 6 3 0 2 3 4 1 9 2 6 9 1 8 3 5 1
 2 8 2 2 9 7 2 3 6 0 5 3 7 5 1 2 9 9 3 1 7 7 4 8 5 8 5 5 2 5 9 0 7 1 4 7 3
 4 8 9 7 9 8 2 6 5 2 5 8 4 8 7 0 6 1 5 9 9 9 5 9 9 5 7 5 6 2 8 6 9 6 1 5 1
 5 9 9 1 5 3 6 1 8 9 8 7 6 7 6 5 6 0 8 8 9 8 6 1 0 4 1 6 3 8 6 7 4 5 6 3 0
 3 3 3 0 7 7 5 7 8 0 7 8 9 6 4 5 0 1 4 6 4 3 3 0 9 5 9 2 1 4 2 1 6 8 9 2 4
 0 2 7 6 2 2 2 1 6 0 2 6 2 2 2 0 7 6 1 1 0 7 2 7 8 5 5 7 5 2 2 7 2 7 5 5 7
```

```
 9 3 7 6 2 3 3 1 6 9 3 6 3 2 2 0 7 6 1 1 9 7 2 7 8 5 5 7 5 2 3 7 2 7 5 5 7
 0 9 1 6 5 9 7 4 3 8 0 3 6 4 6 3 2 6 8 8 8 4 6 7 5 2 4 5 3 2 4 6 9 4 5 4 3
 4 6 2 9 0 1 7 2 0 9 6 0 4 2 0 7 9 8 5 4 8 2 8 4 3 7 2 6 9 1 5 1 0 8 2 1 9
 5 6 8 2 7 2 1 5 1 6 4 5 0 9 4 1 1 7 0 8 9 0 5 4 3 8 8]
(1437, 64)
(360, 64)
(1437,)
(360,)
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)

print("X_train.shape =", X_train.shape)

X_test = scaler.transform(X_test)
print("X_test.shape =", X_test.shape)
```

```
X_train.shape = (1437, 64)
X_test.shape = (360, 64)
```

```
model = LogisticRegression(solver='liblinear', C=0.05, multi_class='ovr',
                           random_state=0)

model.fit(X_train, y_train)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning:

'multi_class' was deprecated in version 1.5 and will be removed in 1.7. Use OneVsRestClassifier(Lo
```

LogisticRegression

```
LogisticRegression(C=0.05, multi_class='ovr', random_state=0,
                   solver='liblinear')
```

```
print("y_train =", y_train)

y_train_pred = model.predict(X_train)
print("y_train_pred =", y_train_pred)

print("y_test =", y_test)

y_test_pred = model.predict(X_test)
print("y_test_pred =", y_test_pred)
```

```
y_train = [6 5 3 ... 7 7 8]
y_train_pred = [6 5 3 ... 7 7 8]
y_test = [2 8 2 6 6 7 1 9 8 5 2 8 6 6 6 6 1 0 5 8 8 7 8 4 7 5 4 9 2 9 4 7 6 8 9 4 3
 1 0 1 8 6 7 7 1 0 7 6 2 1 9 6 7 9 0 0 5 1 6 3 0 2 3 4 1 9 2 6 9 1 8 3 5 1
 2 8 2 2 9 7 2 3 6 0 5 3 7 5 1 2 9 9 3 1 7 7 4 8 5 8 5 5 2 5 9 0 7 1 4 7 3
 4 8 9 7 9 8 2 6 5 2 5 8 4 8 7 0 6 1 5 9 9 9 5 9 9 5 7 5 6 2 8 6 9 6 1 5 1
 5 9 9 1 5 3 6 1 8 9 8 7 6 7 6 5 6 0 8 8 9 8 6 1 0 4 1 6 3 8 6 7 4 5 6 3 0
 3 3 3 0 7 7 5 7 8 0 7 8 9 6 4 5 0 1 4 6 4 3 3 0 9 5 9 2 1 4 2 1 6 8 9 2 4
 9 3 7 6 2 3 3 1 6 9 3 6 3 2 2 0 7 6 1 1 9 7 2 7 8 5 5 7 5 2 3 7 2 7 5 5 7
 0 9 1 6 5 9 7 4 3 8 0 3 6 4 6 3 2 6 8 8 8 4 6 7 5 2 4 5 3 2 4 6 9 4 5 4 3
 4 6 2 9 0 1 7 2 0 9 6 0 4 2 0 7 9 8 5 4 8 2 8 4 3 7 2 6 9 1 5 1 0 8 2 1 9
 5 6 8 2 7 2 1 5 1 6 4 5 0 9 4 1 1 7 0 8 9 0 5 4 3 8 8]
```

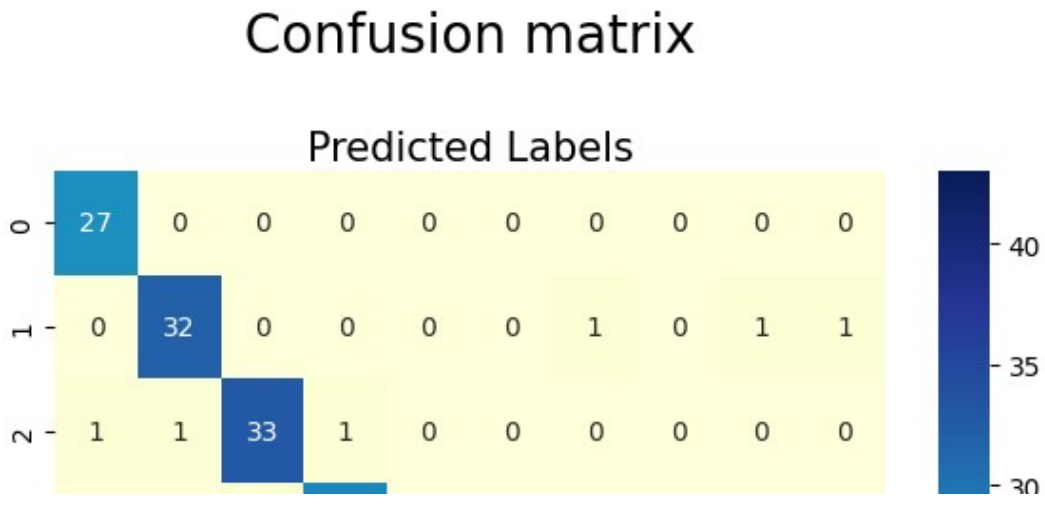
```
y_test_pred = [2 8 2 6 6 7 1 9 8 5 2 8 6 6 6 6 1 0 5 8 8 7 8 4 7 5 4 9 2 9 4 7 6 8 9 4 3
 1 0 1 8 6 7 7 9 0 7 6 2 1 9 6 7 9 0 0 5 1 6 3 0 2 3 4 1 9 2 6 9 1 8 3 5 1
 2 1 2 2 9 7 2 3 6 0 5 3 7 5 1 2 9 9 3 1 7 7 4 8 5 8 5 5 2 5 9 0 7 1 4 7 3
 4 8 9 7 7 8 0 1 5 2 5 3 4 1 7 0 6 1 5 9 9 9 5 9 9 5 7 5 6 2 8 6 7 6 1 5 1
 5 9 9 1 5 3 6 1 8 9 7 7 6 7 6 5 6 0 8 8 9 3 6 1 0 4 1 6 3 8 6 7 4 9 6 3 0
 3 3 3 0 7 7 5 7 8 0 7 8 9 6 4 5 0 1 4 6 4 3 3 0 9 5 9 2 1 4 2 1 6 8 9 2 4
 9 3 7 6 2 3 3 1 6 9 3 6 3 2 2 0 7 6 1 1 3 7 2 7 8 5 5 7 5 3 2 7 2 7 5 5 7
 0 9 1 6 5 9 7 4 3 8 0 3 6 4 6 3 1 6 8 8 8 4 6 7 5 2 4 5 3 2 4 6 9 4 5 4 3
 4 6 2 9 0 6 7 2 0 9 6 0 4 2 0 7 5 8 5 7 8 2 8 4 3 7 2 6 8 1 5 1 0 8 2 8 9
 5 6 2 2 7 2 1 5 1 6 4 5 0 9 4 1 1 7 0 8 9 0 5 4 3 8 8]
```

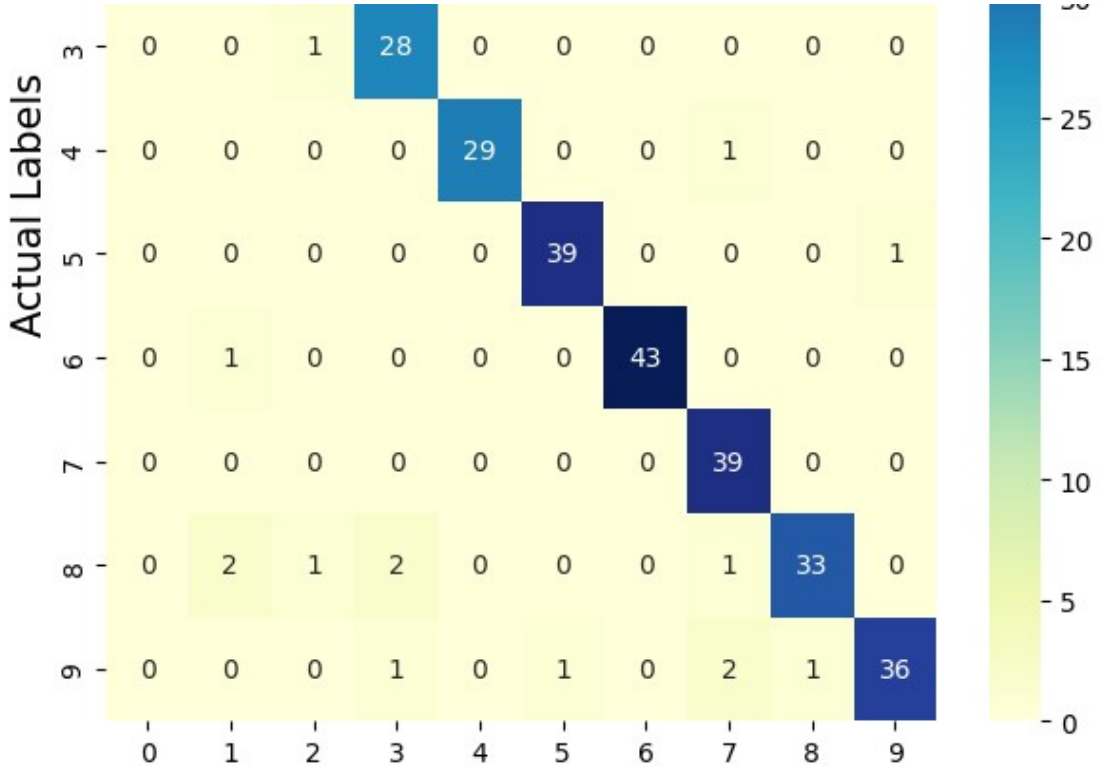
```
mean_accuracy_train = model.score(X_train, y_train)
print(f"mean_accuracy_train = {mean_accuracy_train}")

mean_accuracy_test = model.score(X_test, y_test)
print(f"mean_accuracy_test = {mean_accuracy_test}")

mean_accuracy_train_eq = np.sum(y_train == y_train_pred) / len(y_train)
print(mean_accuracy_train_eq)

mean_accuracy_test_eq = np.sum(y_test == y_test_pred) / len(y_test)
print(mean_accuracy_test_eq)
```

```
mean_accuracy_train = 0.964509394572025
mean_accuracy_test = 0.9416666666666667
0.964509394572025
0.9416666666666667
```

```
c_matrix = confusion_matrix(y_test, y_test_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix, figure_size = (6, 6), title_font_size = 20, font_size = 15)
```

```
[[27  0  0  0  0  0  0  0  0  0]
 [ 0 32  0  0  0  0  1  0  1  1]
 [ 1  1 33  1  0  0  0  0  0  0]
 [ 0  0  1 28  0  0  0  0  0  0]
 [ 0  0  0  0 29  0  0  1  0  0]
 [ 0  0  0  0  0 39  0  0  0  1]
 [ 0  1  0  0  0  0 43  0  0  0]
 [ 0  0  0  0  0  0  0 39  0  0]
 [ 0  2  1  2  0  0  0  1 33  0]
 [ 0  0  0  1  0  1  0  2  1 36]]
```






```
report = classification_report(y_test, y_test_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        27
           1       0.89      0.91      0.90        35
           2       0.94      0.92      0.93        36
           3       0.88      0.97      0.92        29
           4       1.00      0.97      0.98        30
           5       0.97      0.97      0.97        40
           6       0.98      0.98      0.98        44
           7       0.91      1.00      0.95        39
           8       0.94      0.85      0.89        39
           9       0.95      0.88      0.91        41

    accuracy                           0.94       360
   macro avg       0.94      0.94      0.94       360
weighted avg       0.94      0.94      0.94       360
```

```
report = my_classification_report(y_test, y_test_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        27
           1       0.89      0.91      0.90        35
           2       0.94      0.92      0.93        36
           3       0.88      0.97      0.92        29
           4       1.00      0.97      0.98        30
           5       0.97      0.97      0.97        40
           6       0.98      0.98      0.98        44
           7       0.91      1.00      0.95        39
           8       0.94      0.85      0.89        39
```

```
9       0.95      0.88      0.91        41
```

Real World Logistic Regression Application: Diabetes Prediction

Mount Google Drive

```
# Uncomment the below if you need to read data from your Google Drive
# Change the notebook_path to where you run the Jupyter Notebook from.

from google.colab import drive
import os

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
notebook_path = r"/content/drive/MyDrive/MLLabs/Lab08_LogisticRegression"
os.chdir(notebook_path)
!pwd
```

```
/content/drive/MyDrive/MLLabs/Lab08_LogisticRegression
```

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
```

```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']

# load dataset
pima = pd.read_csv("pima-indians-diabetes.csv", header = 0, names = col_names)
pima.head()
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
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


```
# split dataset in features and target variable
```

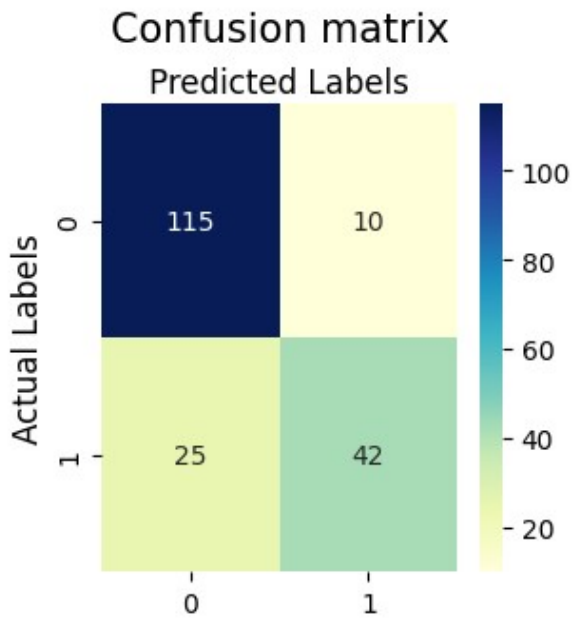
```
# split dataset in features and target variable
feature_cols = ['pregnant', 'insulin', 'bmi', 'age', 'glucose', 'bp', 'pedigree']

X = pima[feature_cols]  # Features
y = pima.label          # Target variable
```

```
# split X and y into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state = 16)
```

```
# instantiate the model (using the default parameters)
model = LogisticRegression(random_state = 16, max_iter = 200)

model.fit(X_train, y_train)

# Test using test set
y_test_pred = model.predict(X_test)
```

```
c_matrix = confusion_matrix(y_test, y_test_pred)
print(c_matrix)

plot_confusion_matrix(c_matrix)
```

```
[[115  10]
 [ 25  42]]
```




```
target_names = ['Without Diabetes', 'With diabetes']

report = classification_report(y_test, y_test_pred, target_names = target_names)
print(report)
```

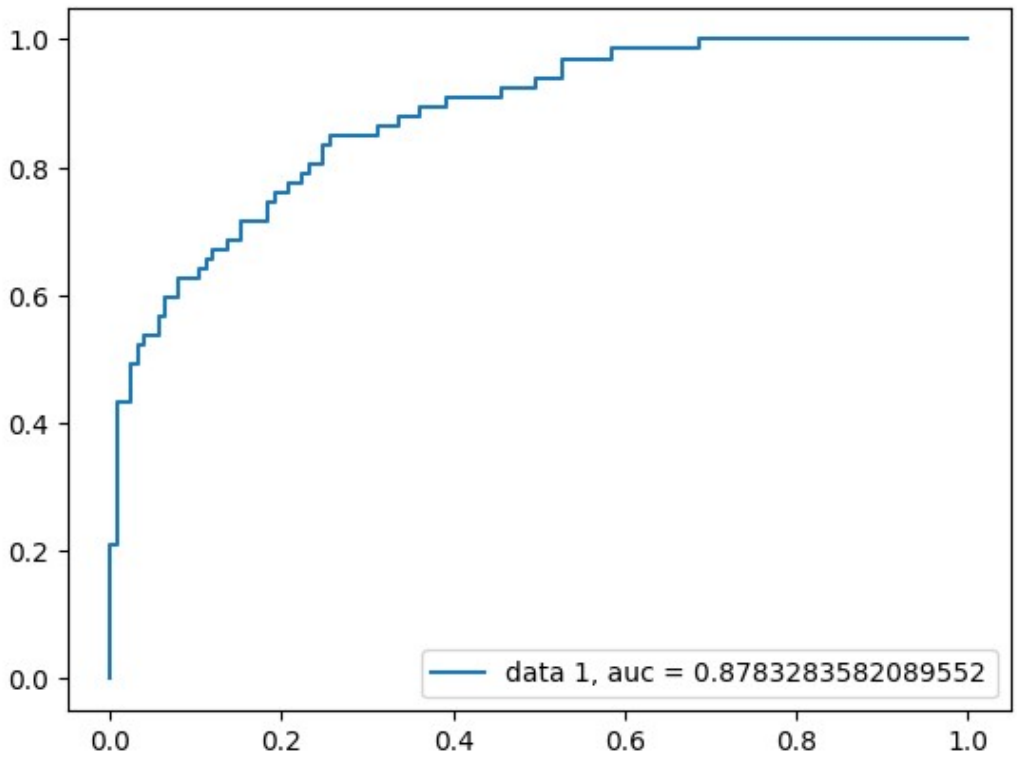
```
                    precision    recall  f1-score   support

     Without Diabetes       0.82      0.92      0.87       125
        With diabetes       0.81      0.63      0.71        67

             accuracy                           0.82       192
            macro avg       0.81      0.77      0.79       192
         weighted avg       0.82      0.82      0.81       192
```

```
weighted avg       0.82      0.82      0.81       192
```

```
report = my_classification_report(y_test, y_test_pred, target_names = target_names)
print(report)
```

```
                     precision    recall  f1-score   support

     Without Diabetes       0.82      0.92      0.87       125
        With diabetes       0.81      0.63      0.71        67
```

```
from sklearn.metrics import roc_curve, roc_auc_score

y_pred_proba = model.predict_proba(X_test)[::, 1]

fpr, tpr, _ = roc_curve(y_test,  y_pred_proba)

auc = roc_auc_score(y_test, y_pred_proba)

plt.plot(fpr, tpr, label="data 1, auc = " + str(auc))

plt.legend(loc = 4)

plt.show()
```



Logistic Regression With `StatsModels`: Example

```
import numpy as np
import statsmodels.api as sm
```

```
X = np.arange(10).reshape(-1, 1)
y = np.array([0, 1, 0, 0, 1, 1, 1, 1, 1, 1])

X = sm.add_constant(X)

print(X)
print(y)
print(X.shape)
print(y.shape)
```

```
[[1. 0.]
 [1. 1.]
 [1. 2.]
 [1. 3.]
 [1. 4.]
 [1. 5.]
 [1. 6.]
 [1. 7.]
 [1. 8.]
 [1. 9.]]
[0 1 0 0 1 1 1 1 1 1]
(10, 2)
(10,)
```

```
model = sm.Logit(y, X)  # NOTE: X is 2nd argument

result = model.fit(method = 'newton')
```

```
Optimization terminated successfully.
         Current function value: 0.350471
         Iterations 7
```

```
print(result.params)
```

```
[-1.972805    0.82240094]
```

```
result.predict(X)
```

```
array([0.12208792, 0.24041529, 0.41872657, 0.62114189, 0.78864861,
       0.89465521, 0.95080891, 0.97777369, 0.99011108, 0.99563083])
```

```
(result.predict(X) >= 0.5).astype(int)
```

```
array([0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

```
result.pred_table()
```

```
array([[2., 1.],
       [1., 6.]])
```

```
result.summary()
```

Logit Regression Results

Dep. Variable: y **No. Observations:** 10

Model:	Logit	**Df Residuals:**	8
Method:	MLE	**Df Model:**	1
Date:	Thu, 06 Feb 2025	**Pseudo R-squ.:**	0.4263
Time:	13:14:31	**Log-Likelihood:**	-3.5047
converged:	True	**LL-Null:**	-6.1086
Covariance Type:	nonrobust	**LLR p-value:**	0.02248

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
const	-1.9728	1.737	-1.136	0.256	-5.377	1.431
x1	0.8224	0.528	1.557	0.119	-0.213	1.858

```
result.summary2()
```

Model:	Logit	Method:	MLE
Dependent Variable:	y	Pseudo R-squared:	0.426
Date:	2025-02-06 13:14	AIC:	11.0094
No. Observations:	10	BIC:	11.6146
Df Model:	1	Log-Likelihood:	-3.5047
Df Residuals:	8	LL-Null:	-6.1086
Converged:	1.0000	LLR p-value:	0.022485
No. Iterations:	7.0000	Scale:	1.0000

	Coef.	**Std.Err.**	**z**	**P>|z|**	**[0.025**	**0.975]**
const	-1.9728	1.7366	-1.1360	0.2560	-5.3765	1.4309
x1	0.8224	0.5281	1.5572	0.1194	-0.2127	1.8575

END.