Logistic Regression

Agenda

- Introduction to Logistic Regression
    - Types of Logistic Regression
- Why do we use Logistic Regression rather than Linear Regression?
- Why are we not using Step Function?
- Logistic Function(sigmoid Function)
    - Key Properties of Logistic Regression Equation
- Simulation of Logistic Function
- Negative Log Likelihood
    - Likelihood
    - Negative Loglikelihood
    - Minimization of Negative loglikelihood
    - Gradient Descent
    - Logistic Regression Formulation
- Simulation
    - Define Sigmoid Function
    - Define Hypothesis
    - Define Error(Cost Function)
    - Define Gradient
    - Define Gradient Descent
- Simulation Using Sklearn
    - Model Performance
        - Accuracy Evaluation
        - Confusion Matrix
        - Evaluation Metrics
            - Precision
            - Recall
- Logistic Regression Assumptions
    - The dependent/response variable is binary or dichotomous
    - Little or no multicollinearity between the predictor/explanatory variables
    - Linear relationship of independent variables to log odds
    - Prefers large sample size

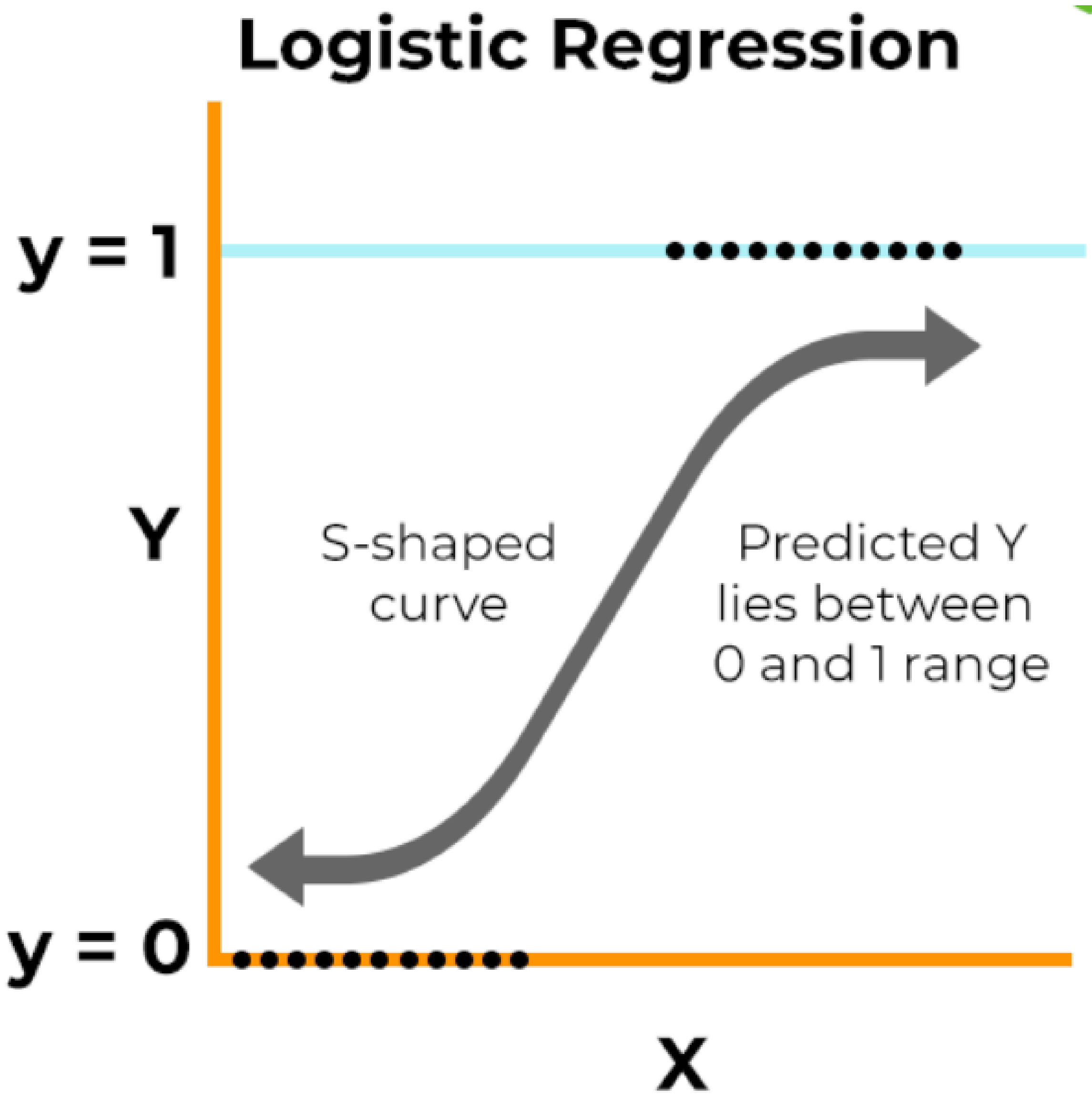
- Problem with extreme outliers
- Independent Observations

Introduction to Logistic Regression

Logistic Regression is a supervised machine learning algorithm that accomplishes binary classification tasks by predicting the probability of an outcome, event, or observation.

Logistic Regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. It is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

It is used when our dependent variable is dichotomous or binary. It just means a variable that has only 2 outputs, for example, A person will survive this accident or not, The student will pass this exam or not. The outcome can either be yes or no (2 outputs). This regression technique is similar to linear regression and can be used to predict the Probabilities for classification problems.



Key Assumptions for Implementing Logistic Regression

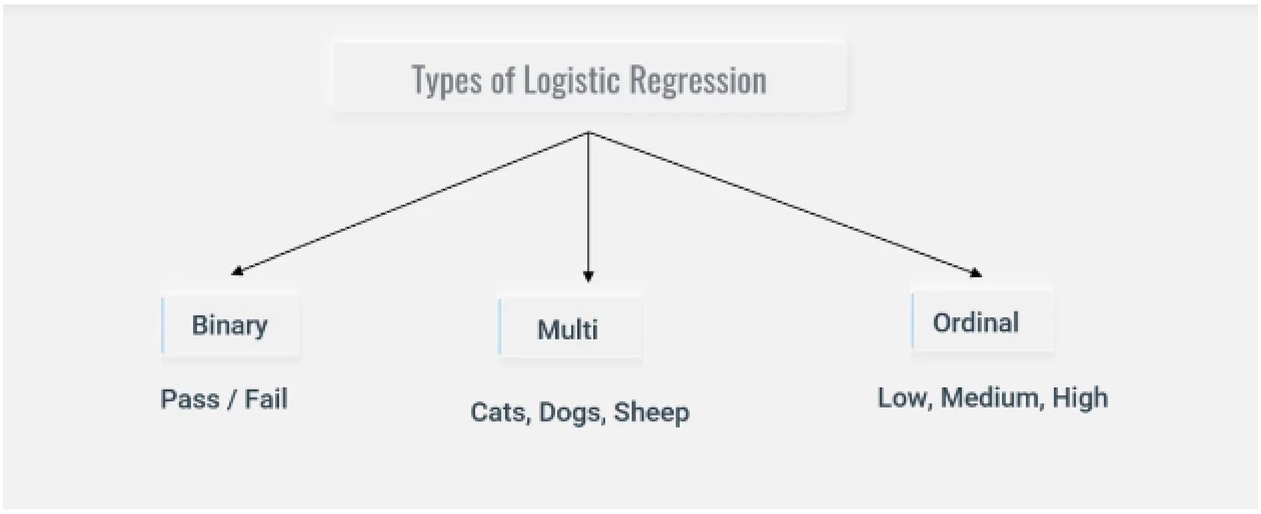
Examples

Some examples of such classifications and instances where the binary response is expected or implied are:

1. Determine the probability of heart attacks: With the help of a logistic model, medical practitioners can determine the relationship between variables such as the weight, exercise, etc., of an individual and use it to predict whether the person will suffer from a heart attack or any other medical complication.
2. Possibility of enrolling into a university: Application aggregators can determine the probability of a student getting accepted to a particular university or a degree course in a college by studying the relationship between the estimator variables, such as GRE, GMAT, or TOEFL scores.

3. Identifying spam emails: Email inboxes are filtered to determine if the email communication is promotional/spam by understanding the predictor variables and applying a logistic regression algorithm to check its authenticity.

Types of Logistic Regression



Binary Logistic Regression

Binary logistic regression predicts the relationship between the independent and binary dependent variables. Some examples of the output of this regression type may be, success/failure, 0/1, or true/false.

Examples:

- Deciding on whether or not to offer a loan to a bank customer: Outcome = yes or no.
- Evaluating the risk of cancer: Outcome = high or low.
- Predicting a team's win in a football match: Outcome = yes or no.

Multinomial Logistic Regression

A categorical dependent variable has two or more discrete outcomes in a multinomial regression type. This implies that this regression type has more than two possible outcomes.

Examples:

- Let's say you want to predict the most popular transportation type for 2040. Here, transport type equates to the dependent variable, and the possible outcomes can be electric cars, electric trains, electric buses, and electric bikes.
- Predicting whether a student will join a college, vocational/trade school, or corporate industry.

- Estimating the type of food consumed by pets, the outcome may be wet food, dry food, or junk food.

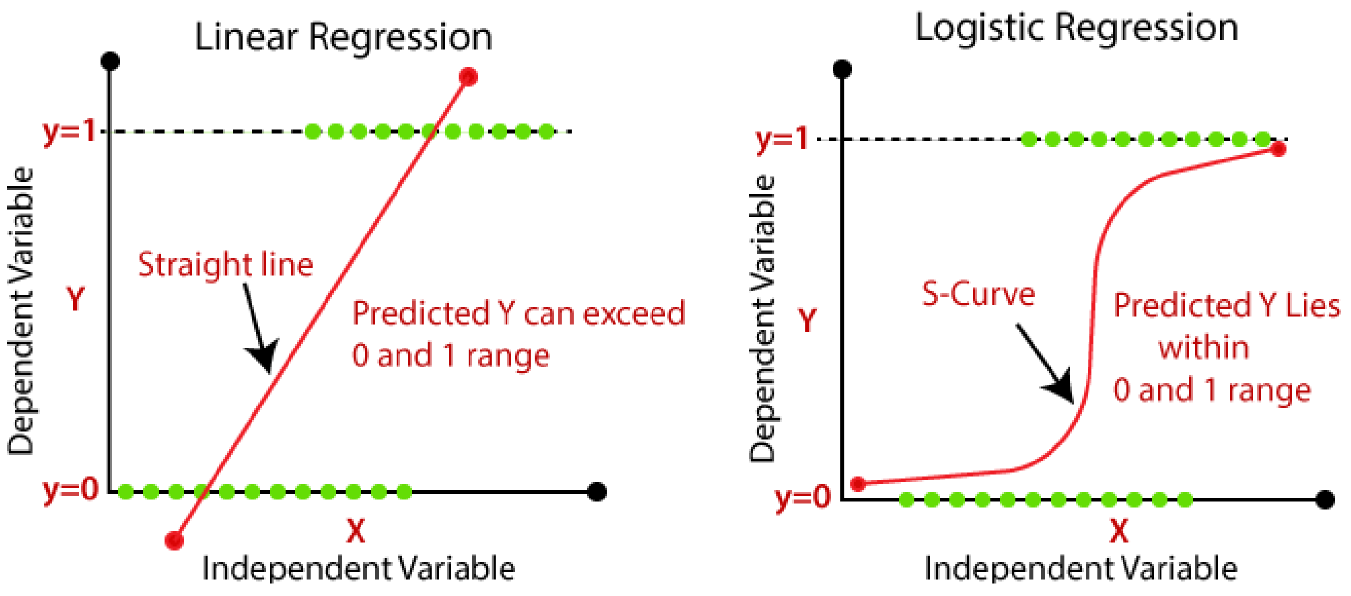
Ordinal Logistic Regression

Ordinal logistic regression applies when the dependent variable is in an ordered state (i.e., ordinal). The dependent variable (y) specifies an order with two or more categories or levels.

Examples: Dependent variables represent,

- Formal shirt size: Outcomes = XS/S/M/L/XL
- Survey answers: Outcomes = Agree/Disagree/Unsure
- Scores on a math test: Outcomes = Poor/Average/Good

Why do we use Logistic Regression rather than Linear Regression?



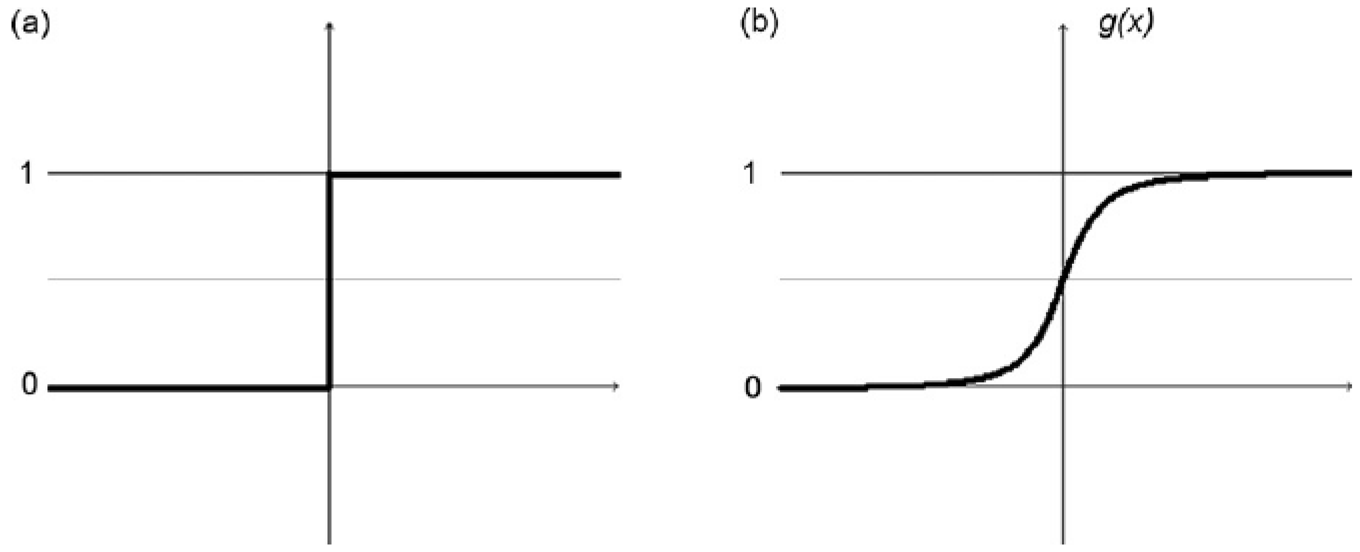
Logistic Regression is preferred over Linear Regression for categorical outcomes:

1. Nature of the Dependent Variable: Logistic Regression is specifically designed to handle categorical dependent variables. It's suitable for binary outcomes (e.g., whether a customer will buy a product or not) or multi-class outcomes (e.g., predicting which category a customer will choose: low, medium, or high).
2. Output Interpretability: In Logistic Regression, the output is a probability estimate. It models the probability that a given input belongs to a particular category. This makes it easier to interpret the results in terms of likelihoods. For example, if the model predicts a probability of 0.8 for a certain class, it means that the model is 80% certain that the input belongs to that class.

3. Sigmoid Activation Function: Logistic Regression uses a sigmoid (logistic) activation function to map the linear combination of input features to the range [0, 1]. This function ensures that the output of the model lies within this range, making it suitable for representing probabilities.
4. Assumption of Linearity of Log Odds: Logistic Regression models the natural logarithm of the odds ratio, which assumes a linear relationship between the independent variables and the log odds of the outcome. While the relationship between the independent variables and the probability of the outcome may not be linear, the relationship between the independent variables and the log odds is assumed to be linear in logistic regression. This assumption is better suited for modeling categorical outcomes.
5. Robustness to Outliers: Logistic Regression is more robust to outliers compared to Linear Regression. Outliers in the dependent variable can significantly affect the results of linear regression, whereas logistic regression is less affected because it models probabilities rather than absolute values.
6. Performance Evaluation Metrics: Evaluation metrics used for logistic regression, such as accuracy, precision, recall, and F1-score, are more appropriate for classification tasks compared to the metrics used for linear regression, such as mean squared error (MSE) or R-squared.

So, Logistic Regression is preferred over Linear Regression for categorical outcomes because it is specifically designed for such scenarios, provides interpretable probability estimates, and is robust to outliers.

Why are we not using step function?



We are not using the step function because it presents certain challenges that make it unsuitable for our purposes. One of the main issues with the step function is its discontinuous nature. This discontinuity causes problems in predicting outcomes as it may result in abrupt changes or jumps in the predictions based on small changes in input variables. Additionally, deciding where to place the step in the function (i.e., determining the threshold) can be arbitrary and may not accurately reflect the underlying data distribution. The step function lacks the smoothness and flexibility required for effectively modeling the relationship between input variables and binary outcomes.

Logistic Function (Sigmoid Function)

The logistic function, also known as the sigmoid function, is a mathematical function that maps input values to a range between 0 and 1.

The logistic function is defined as:

$$P = \frac{1}{1 + e^{-\left(\beta_0 + \beta_1 x\right)}}$$

How similar it is too linear regression?

The equation of the best fit line in linear regression is:

$$y = \beta_0 + \beta_1 x$$

When working with logistic regression, we start with a linear relationship between the input features and the log odds of the outcome.

This relationship is represented by the equation:

$$P = \beta_0 + \beta_1 x$$

To model this relationship, we want to transform p in such a way that it can take on values across the entire real number line, while ensuring that the output remains within the range of probabilities (0 to 1).

We choose to transform p into the odds ratio, which is defined as the ratio of the probability of success to the probability of failure:

$$\frac{P}{1 - P} = \beta_0 + \beta_1 x$$

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

In logistic regression, we aim to transform the linear relationship between the input features and the outcome into a format that allows us to model probabilities effectively. One common transformation we employ is converting probabilities into odds, which represent the ratio of the probability of success to the probability of failure.

The reason we choose to work with odds lies in their inherent properties: they are always positive and have a range from 0 to positive infinity. This characteristic makes odds a convenient choice for modeling purposes.

The odds ratio has a range from 0 to positive infinity. This transformation allows us to work with values that are always positive, making it easier to handle mathematically.

By taking the natural logarithm of the odds ratio, we obtain the log odds (also known as the logit):

$$\log\left(\frac{P}{1-P}\right) = \beta_0 + \beta_1 x$$

Now, we want a function of P because we want to predict probability.

To do so we will multiply by exponent on both sides and then solve for P.

$$\exp[\log(\frac{p}{1-p})] = \exp(\beta_0 + \beta_1 x)$$

$$e^{\ln[\frac{p}{1-p}]} = e^{(\beta_0 + \beta_1 x)}$$

$$\frac{p}{1-p} = e^{\left(\beta_0 + \beta_1 x\right)}$$

$$p = e^{\left(\beta_0 + \beta_1 x\right)} - pe^{\left(\beta_0 + \beta_1 x\right)}$$

$$p = p[\frac{e^{\left(\beta_0+\beta_1 x\right)}}{p} - e^{\left(\beta_0+\beta_1 x\right)}]$$

$$1 = \frac{e^{\left(\beta_0+\beta_1 x\right)}}{p} - e^{\left(\beta_0+\beta_1 x\right)}$$

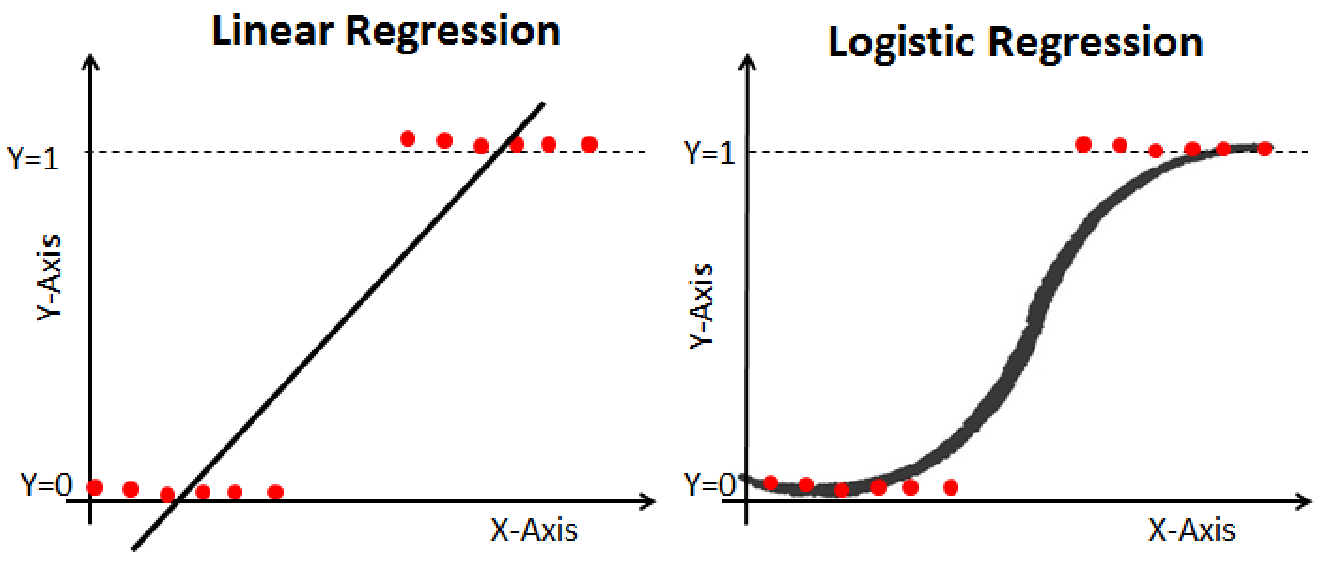
$$p[1+e^{\left(\beta_0+\beta_1 x\right)}] = e^{\left(\beta_0+\beta_1 x\right)}$$

$$p = \frac{e^{\left(\beta_0+\beta_1 x\right)}}{1+e^{\left(\beta_0+\beta_1 x\right)}}$$

$$Now\ dividing\ by\ e^{\left(\beta_0+\beta_1 x\right)},\ we\ will\ get$$

$$p = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}}\quad This\ is\ our\ sigmoid\ function.$$

This is logistic function, also called a sigmoid function. The graph of a sigmoid function is as shown below. It squeezes a straight line into an S-curve.

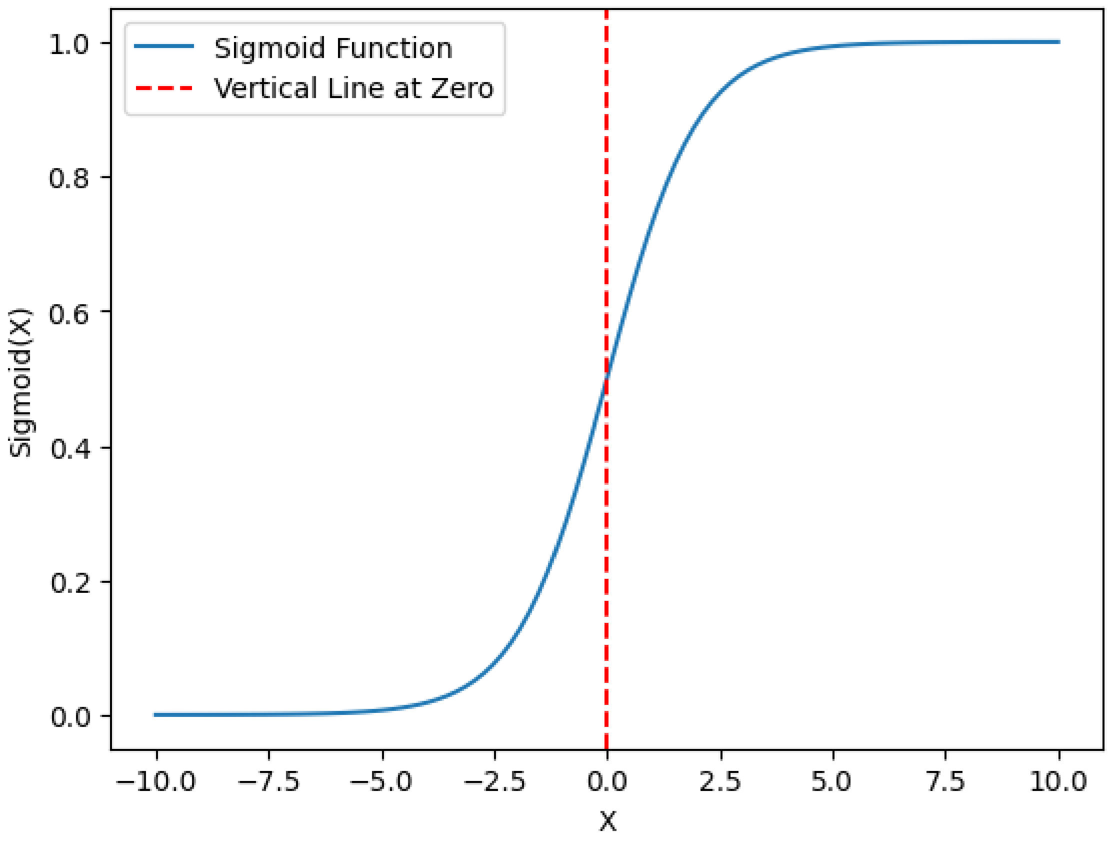


Key properties of the Logistic Regression Equation

- Logistic regression’s dependent variable obeys ‘Bernoulli distribution’
- Estimation/prediction is based on ‘maximum likelihood.’
- Logistic regression does not evaluate the coefficient of determination (or R squared) as observed in linear regression’. Instead, the model’s fitness is assessed through a concordance.

Simulation of Logistic Function

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


def sigmoid(x):
    """
    Compute the sigmoid function for the input x.

    Parameters:
    x (numeric or array-like): Input values.

    Returns:
    numeric or array-like: Output of the sigmoid function.
    """
    return 1 / (1 + np.e ** -x)


# Generate data for plotting sigmoid function
x = np.linspace(-10, 10, 100)
z = sigmoid(x)

# Plot sigmoid function
plt.plot(x, z, label='Sigmoid Function')

# Plot vertical line at zero
plt.axvline(x=0, color='r', linestyle='--', label='Vertical Line at Zero')

# Labeling the plot
plt.xlabel("X")
plt.ylabel("Sigmoid(X)")
plt.legend()

# Show plot
plt.show()
```



2D Data

Two-Dimensional Space:

In logistic regression, we typically deal with multiple features, but for the sake of simplicity, we'll consider a scenario with just two features. Let's call them x_1 and x_2, representing two dimensions in our dataset.

Data Points:

Each data point in our dataset is represented as a pair (x_1, x_2), where x_1 and x_2 are the values of the two features for that particular data point.

Plotting Data Points:

We can plot our data points on a two-dimensional Cartesian coordinate system, with x_1 values on the x-axis and x_2 values on the y-axis. Each data point will be represented as a dot on this plot.

Class Labels:

In logistic regression, each data point belongs to one of two classes, typically labeled as 0 or 1. On our plot, we can use different colors or markers to distinguish between data points belonging to different classes.

Linear Boundary:

- The logistic regression algorithm aims to find a linear boundary that separates the two classes in our dataset. This boundary is represented as a line on our plot.
- Mathematically, the linear boundary can be described by the equation $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$

Decision Boundary:

The linear boundary serves as the decision boundary for logistic regression. Points on one side of the boundary are classified as belonging to one class (e.g., class 0), while points on the other side belong to the other class (e.g., class 1).

Sigmoid Function:

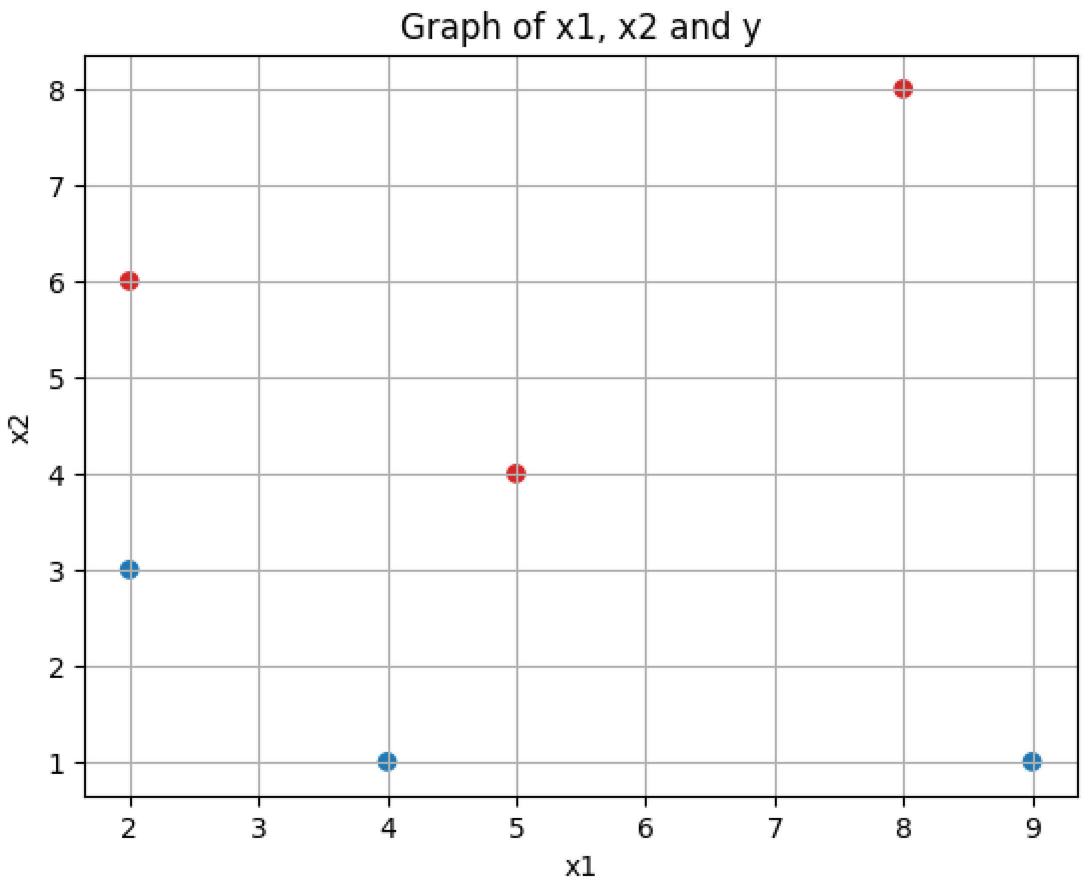
The sigmoid function is used to convert the output of the linear equation (also known as the hypothesis function) into a probability score between 0 and 1. This probability represents the likelihood that a given data point belongs to a particular class.

By visualizing the logistic regression process in two dimensions, we can gain a better understanding of how the algorithm learns to classify data points and draw a decision boundary to separate different classes effectively.

```
# Define the input array
a = np.array([[2, 3], [4, 1], [5, 4], [8, 8], [9, 1], [2, 6]])

# Define the target array
b = np.array([0, 0, 1, 1, 0, 1])


# Create a DataFrame with columns "x1" and "x2" from the input array
df = pd.DataFrame(a, columns=["x1", "x2"])

# Add a new column "y" to the DataFrame with values from the target array
df["y"] = b

# Display the DataFrame
df
```

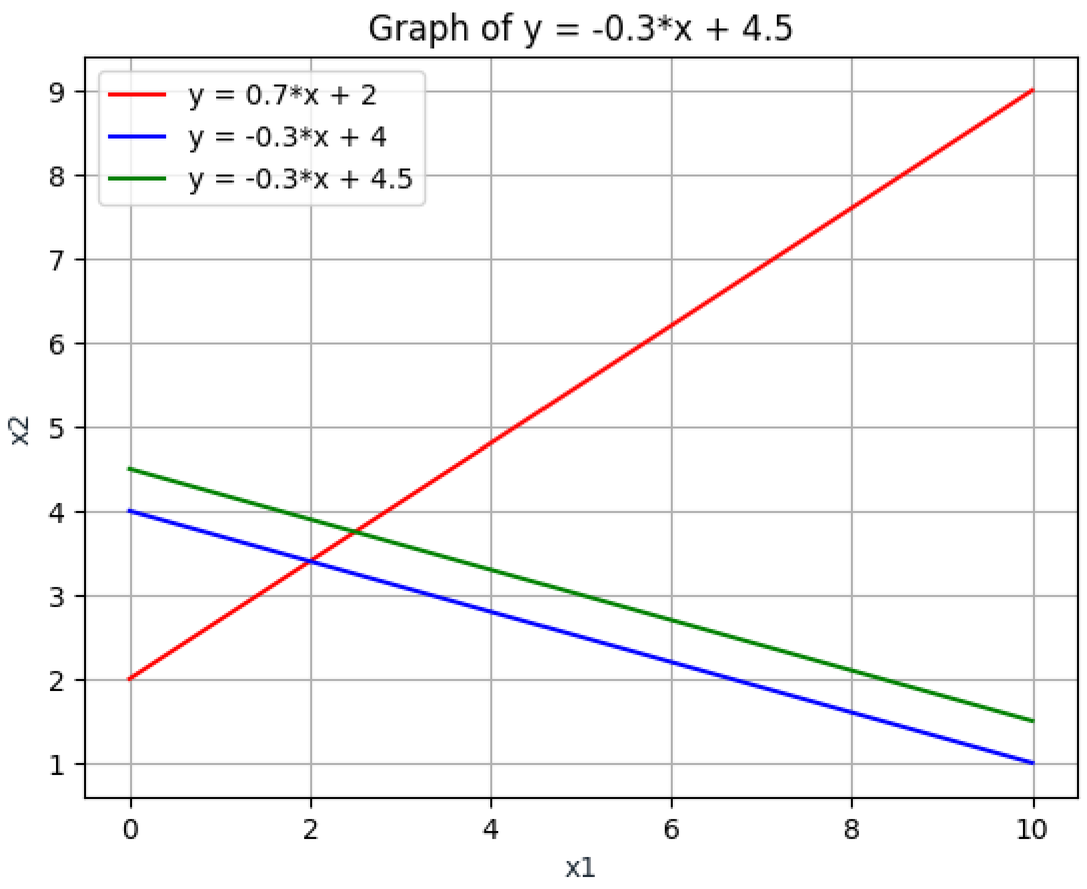
	x1	x2	y
0	2	3	0
1	4	1	0
2	5	4	1
3	8	8	1
4	9	1	0
5	2	6	1

```
# Define colors for each class
colors = {0: 'tab:blue', 1: 'tab:red'}

# Create a scatter plot of x1 vs x2, coloring points based on the y values
plt.scatter(a[:, 0], a[:, 1], c=df["y"].map(colors))

# Set plot title and labels
plt.title("Graph of x1, x2 and y")
plt.xlabel("x1")
plt.ylabel("x2")

# Add grid lines
plt.grid()

# Display the plot
plt.show()
```



Plotting Some Random Lines

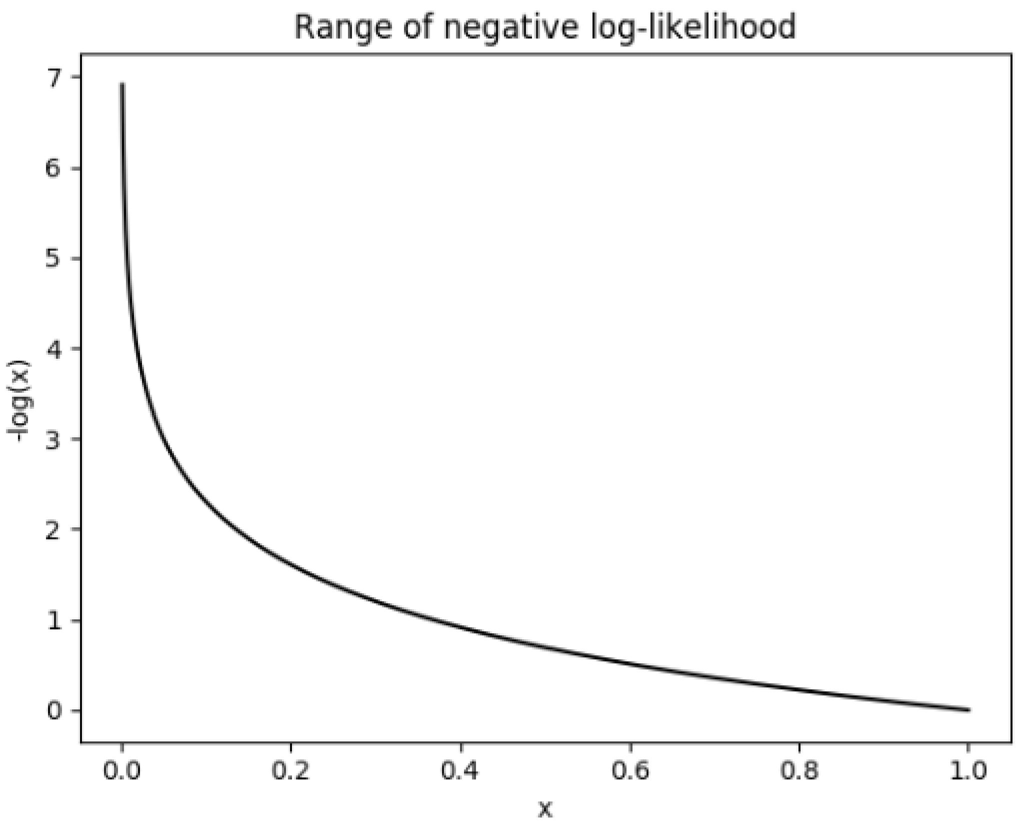
```
# Define x values for lines
x = np.linspace(0, 10, 100)

# Plot lines
y = 0.7*x + 2
plt.plot(x, y, '-r', label="y = 0.7*x + 2")

y = -0.3*x + 4
plt.plot(x, y, '-b', label="y = -0.3*x + 4")

y = -0.3*x + 4.5
plt.plot(x, y, '-g', label="y = -0.3*x + 4.5")

# Set plot title and labels
plt.title("Graph of y = -0.3*x + 4.5")
plt.xlabel('x1', color='#1C2833')
plt.ylabel('x2', color='#1c2833')

# Add legend
plt.legend(loc='upper left')

# Add grid lines
plt.grid()

# Display the plot
plt.show()
```



Negative Log Likelihood

Likelihood

Likelihood: Likelihood represents the probability of a dependent variable Y taking a specific value (typically 0 or 1) given a set of input variables X. It assesses how well a model predicts the observed data.



Likelihood Representation: The likelihood can be mathematically represented as follows:

```
Likelihood = Y.(Y_hat) + (1 − Y).(1 −Y_hat)
```

This formula reflects that if the actual value Y is 1, we want our prediction Y_hat to be close to 1, and if Y is 0, we want Y_hat to be close to 0.

Alternative Representation: The likelihood equation can also be represented in terms of powers:

Likelihood = $(Y\_hat)^{Y}.(1 - Y\_hat)^{(1 - Y)}$

This representation emphasizes the relationship between the predicted probability Y_hat and the actual outcome Y

Negative Log Likelihood

Cost Function of Logistic Regression

Negative Log Likelihood: The negative log likelihood is simply the negative of the log likelihood equation. It's used to simplify calculations and make optimization easier, as minimizing the negative log likelihood is equivalent to maximizing the likelihood.

Negative Log Likelihood = -(Y.log(y_hat) + (1-Y).log(1-Y_hat))

- The negative log likelihood (NLL) is a measure of how well a logistic regression model predicts the probabilities of different classes.
- It's derived from the likelihood function, which represents the probability of observing the given data under the model.
- The NLL is the negative of the log likelihood, and it's used as the cost function to be minimized during the training of the logistic regression model.

By using the negative log likelihood, we aim to find model parameters (such as coefficients in logistic regression) that minimize the discrepancy between the predicted probabilities and the actual outcomes. This optimization process ensures that our model provides the best fit to the observed data.

Negative loglikelihood Function

```
def Negloglikelihood(y, y_hat):
    """
    Calculate the negative log-likelihood loss function for binary classification

    Parameters:
    y (array-like): True binary labels (0 or 1).
    y_hat (array-like): Predicted probabilities of the positive class.

    Returns:
    float: Negative log-likelihood loss.
    """
    # Compute the negative log-likelihood
    loss = -(np.sum(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat)))
    return loss
```

Comparing Different Lines

```
# compute points on x2
df['x2_green'] = -0.3*df["x1"] + 4.5
df['x2_blue'] = -0.3*df["x1"] + 4
df['x2_red'] = 0.7*df["x1"] + 2

# find the distance between each line to the corresponding point
df["dist_green"] = df['x2'] - df['x2_green']
df["dist_blue"] = df['x2'] - df['x2_blue']
df["dist_red"] = df['x2'] - df['x2_red']

# convert distance to probability using sigmoid
df['prob_green'] = sigmoid(df['dist_green'])
df['prob_blue'] = sigmoid(df['dist_blue'])
df['prob_red'] = sigmoid(df['dist_red'])

# compute the neagative loglikelihood for each line
df['loglike_green'] = Negloglikelihood(df['y'], df['prob_green'])
df['loglike_blue'] = Negloglikelihood(df['y'], df['prob_blue'])
df['loglike_red'] = Negloglikelihood(df['y'], df['prob_red'])


print("Negative loglikelihood of Green: ", sum(df['loglike_green']))
print("Negative loglikelihood of Blue: ", sum(df['loglike_blue']))
print("Negative loglikelihood of Red: ", sum(df['loglike_red']))
```

```
Negative loglikelihood of Green:  7.435901492126054
Negative loglikelihood of Blue:  8.97039753455413
Negative loglikelihood of Red:  16.931328011440748
```

When comparing different lines (or decision boundaries) for separating classes in a logistic regression model, the **negative log likelihood (NLL)** serves as a crucial metric for evaluating their performance. Each line is represented by a set of parameters θ, which determine its position and orientation in the feature space. The logistic regression model assigns probabilities to each data point belonging to a particular class based on its position relative to the decision boundary represented by the line.

To compute the NLL for each line, we first use the logistic regression model to **predict the probabilities of the data points** belonging to the positive class (class 1). Then, we **compare these predicted probabilities to the actual** class labels in the dataset. The NLL quantifies the discrepancy between the predicted probabilities and the true labels. Specifically, it penalizes the model more heavily for making inaccurate predictions with higher confidence (i.e., assigning a high probability to the wrong class).

The line that minimizes the negative log likelihood is considered the best line for separating the classes because it maximizes the likelihood of observing the given dataset under the logistic regression model. In other words, it provides the most accurate representation of the relationship between the features and the class labels. Therefore, in your example, the green line, which corresponds to the set of parameters θ that yield the minimum NLL, is identified as the best line for the given dataset. This line optimally separates the classes based on the logistic regression model's estimation of the class probabilities, offering the most effective decision boundary for classification.

Comparison with Linear Regression

Linear regression and logistic regression are two commonly used techniques in machine learning for regression and classification tasks, respectively. One fundamental difference between them lies in their underlying assumptions and cost functions.

- In linear regression, the goal is to model the relationship between the input features X and the continuous target variable Y. The model assumes that the relationship between X and Y can be approximated by a linear function. The cost function used in linear regression is the mean squared error (MSE), which measures the average squared difference between the predicted and actual values of Y. The objective of linear regression is to minimize this cost function, often achieved through techniques like ordinary least squares or gradient descent.
- On the other hand, logistic regression is primarily used for binary classification tasks, where the goal is to predict the probability that an instance belongs to a particular class (usually denoted as class 1). Unlike linear regression, logistic regression models the probability of the target variable belonging to a certain class using a logistic (sigmoid) function. The output of the logistic function is constrained between 0 and 1, representing the probability of class membership.
- The cost function for logistic regression is the negative log likelihood (NLL), which quantifies the error between the predicted probabilities and the true class labels. The NLL penalizes the model for assigning low probabilities to the true class labels and high probabilities to the incorrect class labels. By minimizing the NLL, logistic regression optimizes the parameters of the model to accurately predict the probability of class membership for each instance in the dataset.

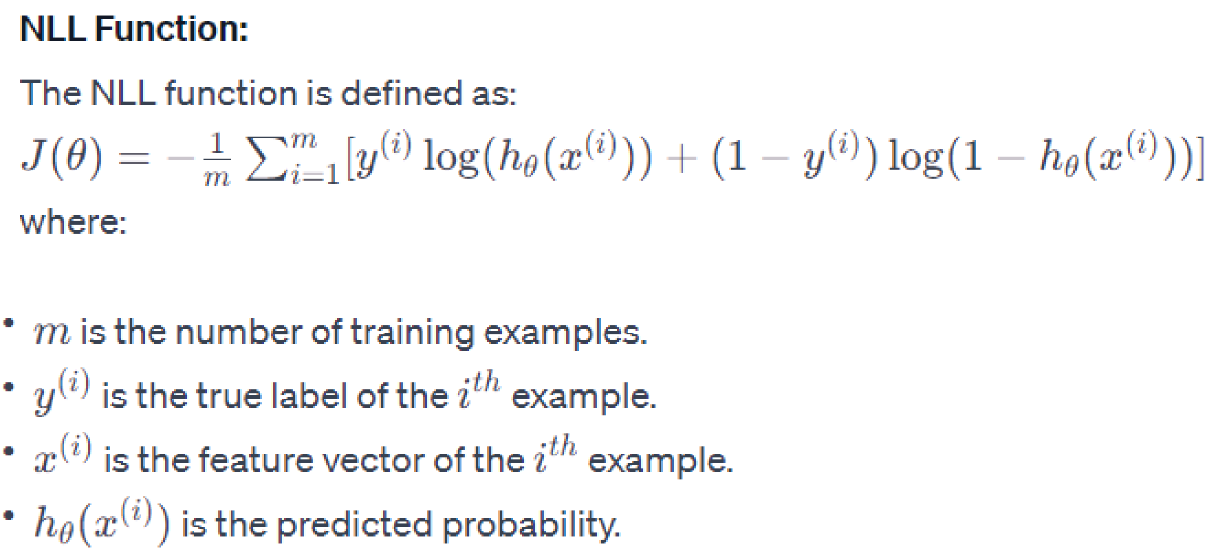
While both linear regression and logistic regression aim to model relationships between features and target variables, they differ in their assumptions, objectives, and cost functions. Linear regression uses the mean squared error to minimize the discrepancy between predicted and actual continuous values, whereas logistic regression uses the negative log likelihood to optimize the model parameters for accurate probability estimation in binary classification tasks.

Minimization of the Negative Log Likelihood

The primary objective in logistic regression is to minimize the negative log likelihood (NLL) function, which is the cost function used to measure the error between the predicted probabilities and the actual labels.

To minimize the NLL, partial derivatives of the NLL function with respect to each parameter θ_j are computed.

The NLL function is given by:



NLL Function:

The NLL function is defined as:

$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$

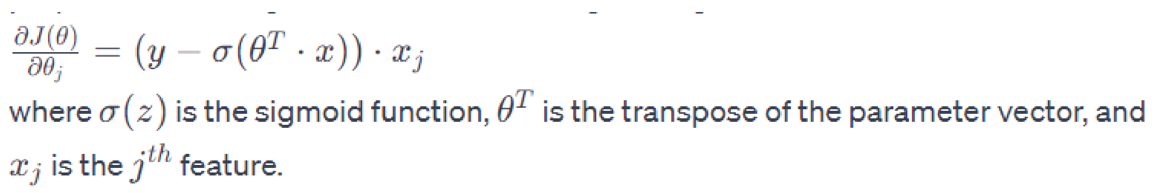
where:

- m is the number of training examples.
- $y^{(i)}$ is the true label of the i^{th} example.
- $x^{(i)}$ is the feature vector of the i^{th} example.
- $h_\theta(x^{(i)})$ is the predicted probability.

To minimize the NLL, partial derivatives of the NLL function with respect to each parameter θ_j are calculated. The partial derivative is computed using the chain rule of calculus.

Gradient Descent

The derivative of the NLL function with respect to θ_j is derived using the chain rule and properties of the sigmoid function, resulting in the gradient:



$\frac{\partial J(\theta)}{\partial \theta_j} = (y - \sigma(\theta^T \cdot x)) \cdot x_j$

where $\sigma(z)$ is the sigmoid function, θ^T is the transpose of the parameter vector, and x_j is the j^{th} feature.

Minimizing the NLL function in logistic regression involves computing partial derivatives, utilizing gradient descent for parameter updates, and deriving the gradient with respect to each parameter. These steps collectively optimize the model parameters to best fit the training data and improve predictive performance.

Logistic Regression Formulation

Logistic Function:

After obtaining the optimal parameters θ, predictions are made using the logistic function:

$$\hat{y} = \frac{1}{1+e^{-\theta^T \cdot x}}$$

Logit Link Function:

The logistic regression equation is derived from the logit link function:

$$\theta^T \cdot x = \log\left(\frac{P}{1-P}\right)$$

where:

- P is the probability of the positive class.
- $1 - P$ is the probability of the negative class.
- $\theta^T \cdot x$ is the linear combination of input features weighted by parameters θ.

The logistic regression equation resembles a linear regression equation but is used for classification purposes. It models the log odds ratio of the probability of the positive class to the negative class. The logit link function transforms the linear combination of input features into a probability between 0 and 1, allowing for binary classification based on a threshold (typically 0.5).

Simulation

Define Sigmoid Function

```
def sigmoid(z):
    """
    Compute the sigmoid function for the input z.

    Parameters:
    z (numeric or array-like): Input values.

    Returns:
    numeric or array-like: Output of the sigmoid function.
    """
    # Compute the sigmoid function
    result = 1.0 / (1.0 + np.exp(-1.0 * z))
    return result
```

Define Hypothesis

```
def hypothesis(X, theta):
    '''
    X - np.array(m, n)
    theta - np.array(n, 1)
    '''
    """
    Compute the hypothesis function for logistic regression.

    Parameters:
    X (array-like): Input features matrix of shape (m, n).
    theta (array-like): Model parameters (coefficients) of shape (n, 1).

    Returns:
    array-like: Predicted probabilities of the positive class.
    """
    # Compute the dot product of X and theta, and pass it through the sigmoid fun
    z = np.dot(X, theta)
    return sigmoid(z)
```

Define Error(Cost Function)

```
def error(X, y, theta):
    '''
      X - (m, n)
      y - (m, 1)
      theta - (n, 1)

      return (n, 1)
    '''
    """
    Compute the cross-entropy error (log loss) for logistic regression.

    Parameters:
    X (array-like): Input features matrix of shape (m, n).
    y (array-like): True binary labels of shape (m, 1).
    theta (array-like): Model parameters (coefficients) of shape (n, 1).

    Returns:
    float: Cross-entropy error.
    """
    # Calculate the predicted probabilities using the hypothesis function
    hypo = hypothesis(X, theta)

    # Compute the cross-entropy error
    err = np.mean((y * np.log(hypo) + (1 - y) * np.log(1 - hypo)))

    # Return the negative of the error (since it's often minimized)
    return -err
```

Define Gradient

```
import numpy as np

def gradient(X, y, theta):
    """
    Compute the gradient of the cross-entropy error with respect to the model par

    Parameters:
    X (array-like): Input features matrix of shape (m, n).
    y (array-like): True binary labels of shape (m, 1).
    theta (array-like): Model parameters (coefficients) of shape (n, 1).

    Returns:
    array-like: Gradient of the error with respect to theta, shape (n, 1).
    """
    # Calculate the predicted probabilities using the hypothesis function
    hypo = hypothesis(X, theta)

    # Compute the gradient
    grad = np.dot(X.T, (hypo - y))

    # Normalize the gradient by the number of samples
    grad /= X.shape[0]

    return grad
```

Define Gradient Descent

```
import numpy as np

def gradient_descent(X, y, lr=0.5, max_iter=30):
    """
    Perform gradient descent to optimize the parameters for logistic regression.

    Parameters:
    X (array-like): Input features matrix of shape (m, n).
    y (array-like): True binary labels of shape (m, 1).
    lr (float): Learning rate (default is 0.5).
    max_iter (int): Maximum number of iterations (default is 30).

    Returns:
    tuple: Tuple containing the optimized parameters (theta) and a list of errors
    """
    # Initialize parameters theta with zeros
    theta = np.zeros((X.shape[1], 1))

    # List to store errors during optimization
    error_list = []

    # Gradient descent loop
    for _ in range(max_iter):
        # Compute the error
        e = error(X, y, theta)
        error_list.append(e)

        # Compute the gradient
        grad = gradient(X, y, theta)

        # Update parameters using the update rule
        theta = theta - lr * grad

    return theta, error_list
```

```
# Assuming X is a 2D array with 10 rows and 5 columns
X = np.zeros((10, 5))  # Example: Initialize X with zeros
theta = np.zeros((X.shape[1], 1))
theta
```

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

```
from sklearn.datasets import make_classification

# Generate synthetic data for binary classification
X, y = make_classification(n_samples=500,
                           n_features=2,
                           n_redundant=0,
                           n_clusters_per_class=1,
                           random_state=5)
```

```
X.shape
```

```
(500, 2)
```

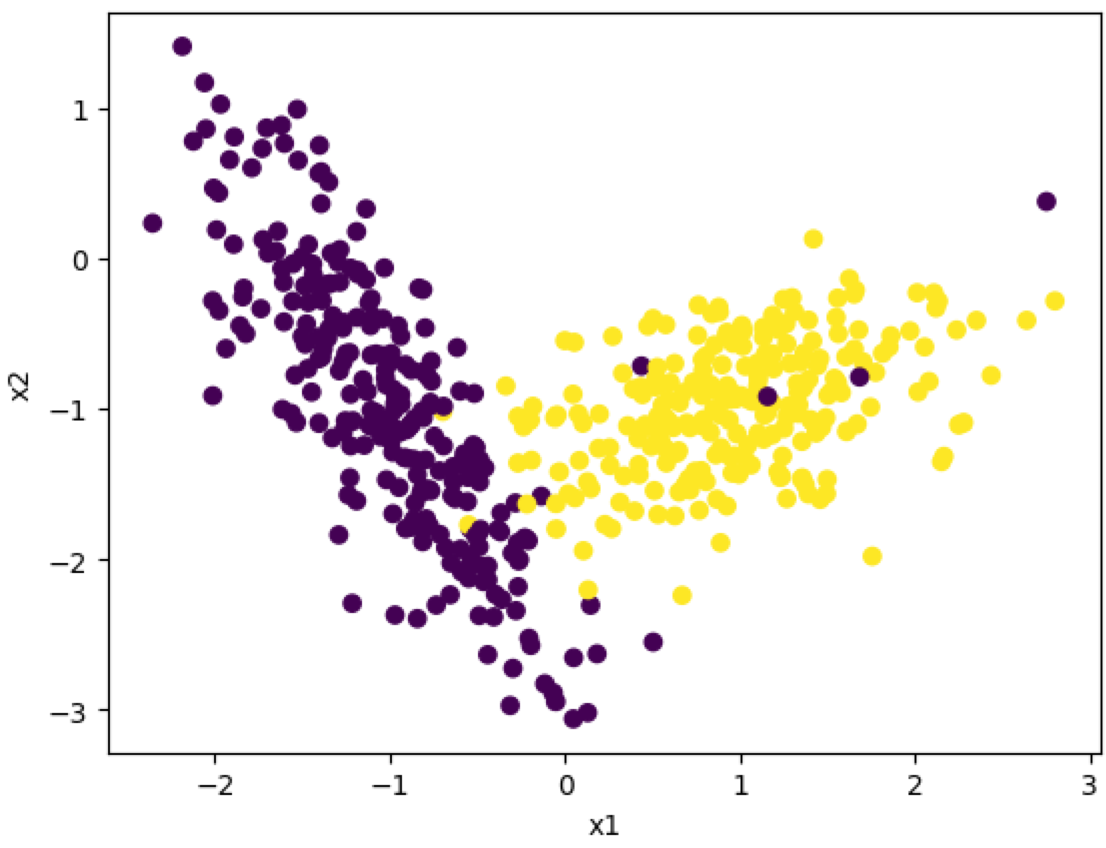
```
y.shape
```

```
(500,)
```

```
y
```

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

```
import matplotlib.pyplot as plt

# Create a scatter plot of the feature values, coloring points based on class lab
plt.scatter(X[:, 0], X[:, 1], c=y)

# Set labels for x-axis and y-axis
plt.xlabel("x1")
plt.ylabel("x2")

# Show the plot
plt.show()
```



```
import numpy as np

# Create a column vector of ones with shape (500, 1)
ones = np.ones((500, 1))

# Stack the column vector of ones horizontally (along axis 1) with the original f
X_ = np.hstack((ones, X))

# Display the first 5 rows of the modified feature matrix
print(X_[:5])
```

```
[[ 1.          1.22167239 -0.4757541 ]
 [ 1.         -0.2292072  -1.85663378]
 [ 1.         -1.34913896  0.50458721]
 [ 1.          0.31402206 -1.62029248]
 [ 1.          1.13807877 -0.99148158]]
```

```
from sklearn.model_selection import train_test_split

# Split the dataset into training and testing sets
# X_train: Training features, X_test: Testing features, y_train: Training labels,
X_train, X_test, y_train, y_test = train_test_split(X_, y, test_size=0.2, random_
```

```
# Use gradient_descent function to optimize parameters for logistic regression
opt_theta, error_list = gradient_descent(X_train, y_train)

# Display the optimized parameters
opt_theta
```

```
array([[ 1.97220933, -1.97220933, -1.97220933, ...,  1.97220933,
         1.97220933,  1.97220933],
       [-0.17941749,  0.17941749,  0.17941749, ..., -0.17941749,
        -0.17941749, -0.17941749],
       [-1.35149631,  1.35149631,  1.35149631, ..., -1.35149631,
        -1.35149631, -1.35149631]])
```

Simulation Using Sklearn

```
from sklearn.linear_model import LogisticRegression  # Importing the LogisticRegr

# Create a logistic regression model instance
logistic = LogisticRegression()

# Fit the model to the training data
logistic.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
logistic.intercept_
```

```
array([1.46952407])
```

```
logistic.coef_
```

```
array([[-3.17062293e-06,  3.89591876e+00,  8.50972735e-01]])
```

Model Performance

Accuracy Evaluation

Accuracy, a fundamental metric for assessing classification model performance, was discussed. It measures the proportion of correctly predicted instances out of the total number of instances.

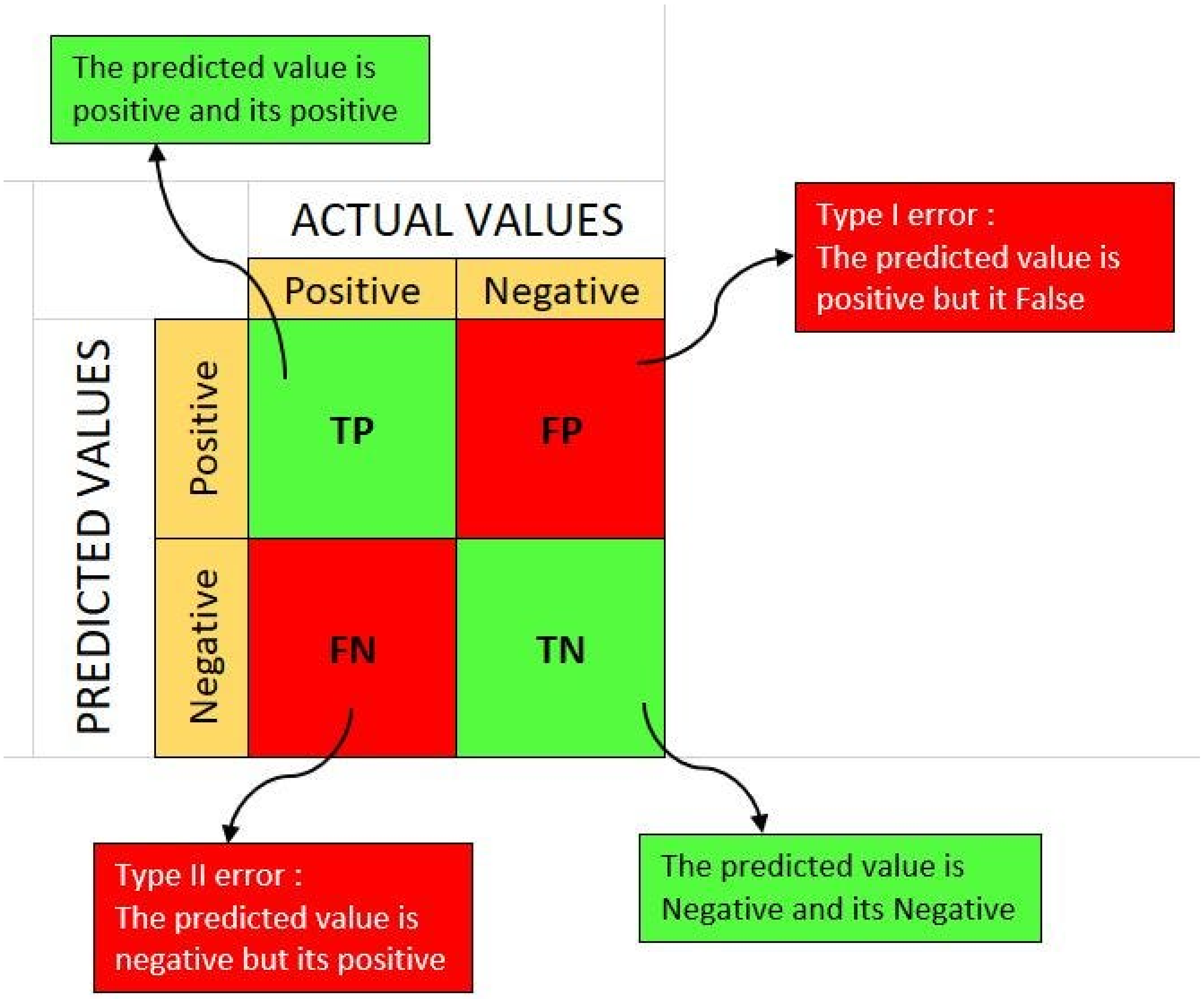
```
# Model Performance on test dataset
```

```
# Predict the class labels for the testing data using the trained logistic regres
y_pred = logistic.predict(X_test)
```

```
# Accuracy
```

Confusion Matrix

- The confusion matrix is a table that allows visualization of the performance of a classification model by comparing actual (true) values with predicted values.
- It serves as a fundamental tool for evaluating the performance of classification models, particularly in scenarios where the outcome is binary (e.g., positive/negative, yes/no).


The predicted value is positive and its positive
ACTUAL VALUES
Positive
Negative
PREDICTED VALUES
Positive
Negative
TP
FP
FN
TN
Type I error : The predicted value is positive but it False
Type II error : The predicted value is negative but its positive
The predicted value is Negative and its Negative

Components of the Confusion Matrix:

- True Positives (TP):
  - True positives represent the cases where the model correctly predicts the positive class.
  - In medical diagnosis, for example, a true positive would occur when the model correctly identifies a patient with a particular disease as having the disease.
- True Negatives (TN):

    - True negatives indicate the cases where the model correctly predicts the negative class.
    - Using the medical diagnosis example, a true negative would occur when the model correctly identifies a healthy individual as not having the disease.
- False Positives (FP):
    - False positives occur when the model incorrectly predicts the positive class.
    - In medical diagnosis, a false positive would be when the model wrongly identifies a healthy individual as having the disease.
- False Negatives (FN):
    - False negatives represent cases where the model incorrectly predicts the negative class.
    - For instance, in medical diagnosis, a false negative would occur when the model fails to identify a patient with the disease as having the disease.

Significance in Model Evaluation:

- The confusion matrix provides a comprehensive overview of a model's performance by breaking down its predictions into four categories.
- It enables stakeholders to assess both the strengths and weaknesses of the model, identifying areas where it performs well and areas requiring improvement.
- Understanding the distribution of true positives, true negatives, false positives, and false negatives helps in determining the effectiveness of the model in differentiating between classes.

Evaluation Metrics

Accuracy

- Accuracy Accuracy is one of the most intuitive metrics for evaluating classification models. It measures the proportion of correctly predicted observations to the total observations. While it's simple to compute and interpret, accuracy can be misleading in cases where there is a class imbalance.

```
Accuracy= TP+TN/FP+FN+TP+TN
```

Precision

- Precision is a performance metric that assesses the accuracy of positive predictions made by a classification model.
- Mathematically, precision is calculated as the ratio of true positives to the sum of true positives and false positives:

```
Precision = TP/(TP + FP)
```

- Precision focuses on the relevance of positive predictions, indicating how many of the predicted positive cases are truly positive.
- It is particularly important in scenarios where false positives are costly or undesirable, such as medical diagnoses or fraud detection.

Recall

- Recall, also known as sensitivity or true positive rate, measures the proportion of actual positive instances that are correctly identified by the model.
- Mathematically, recall is calculated as the ratio of true positives to the sum of true positives and false negatives:

```
Recall = TP/(TP + FN)
```

- Recall emphasizes the model's ability to capture all positive instances in the dataset, irrespective of the number of false negatives.
- It is crucial in situations where missing positive cases can have significant consequences, such as disease detection or anomaly identification.

F1 Score

- The F1 Score provides a more balanced measure than accuracy, particularly in cases of imbalanced datasets. It is the harmonic mean of Precision and Recall, giving equal weight to both false positives and false negatives. The F1 score is useful when you care about the performance on both the positive and negative classes and when the cost of false positives and false negatives is similar.

```
F1=2×(Precision×Recall/Precision+Recall)
```

```
from sklearn.metrics import accuracy_score, confusion_matrix  # Importing accurac
```

```
# Calculate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
```

```
# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)
```

```
from sklearn.metrics import precision_score, recall_score, f1_score  # Importing
```

```
precision_score(y_test, y_pred)
# Calculate precision score
```

```
0.9411764705882353
```

```
# Calculate recall score
recall_score(y_test, y_pred)
```

```
0.96
```

```
f1_score(y_test, y_pred)
```

```
0.9504950495049505
```

```
y_pred
```

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

```
# Predict class probabilities for the testing data using the trained logistic reg
prob = logistic.predict_proba(X_test)
```

```
prob[1]
```

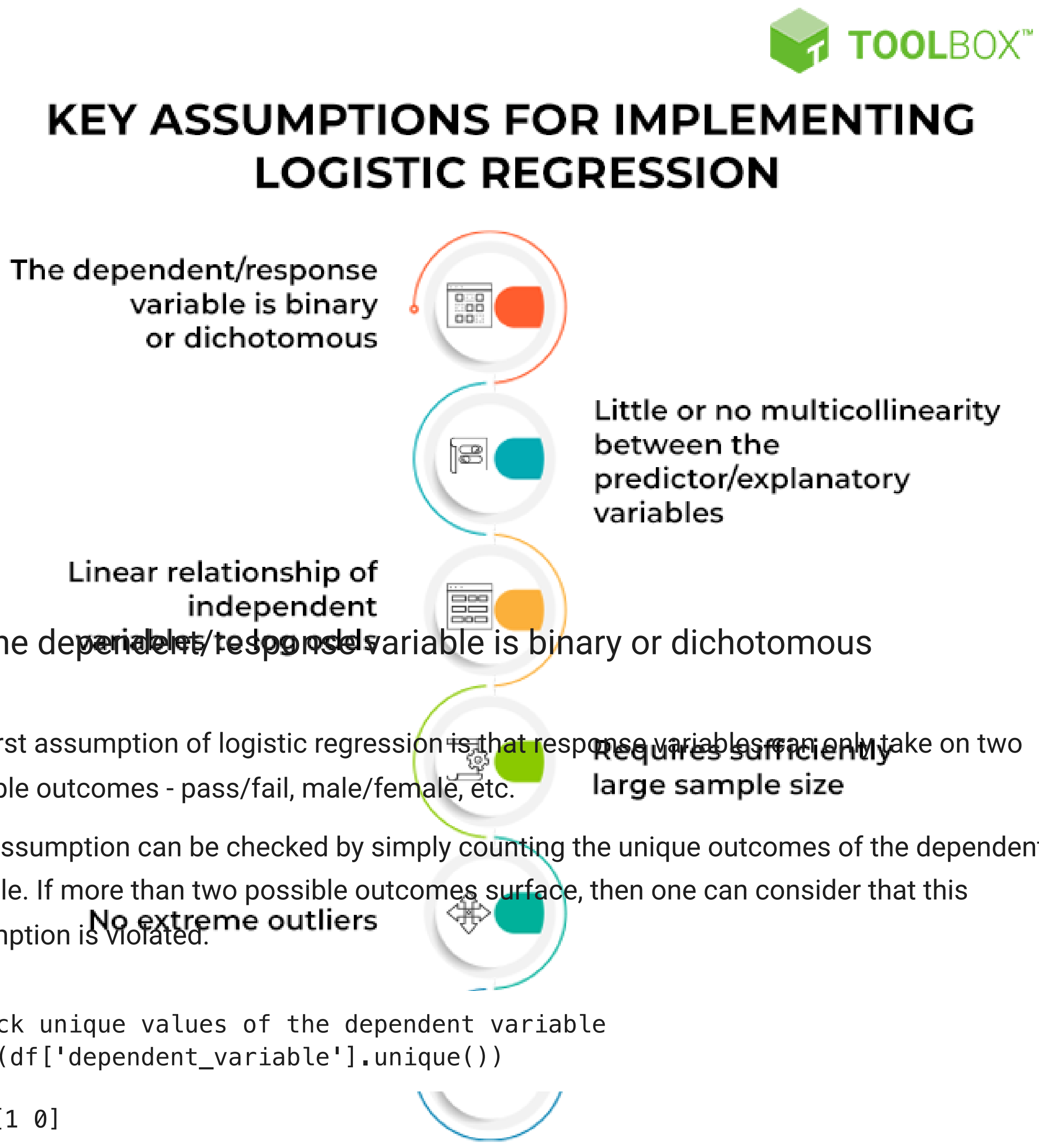
```
array([0.01821833, 0.98178167])
```

Simulation of Logistic Regression Assumptions

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
# Set random seed for reproducibility
np.random.seed(42)

# Number of samples
n_samples = 1000

# Generate independent variables
X1 = np.random.normal(0, 1, n_samples)
X2 = np.random.normal(0, 1, n_samples)
X3 = np.random.normal(0, 1, n_samples)

# Generate correlated independent variables
X4 = 0.5*X1 + 0.5*X2 + np.random.normal(0, 0.2, n_samples)  # Some correlation be
X5 = -0.5*X1 + 0.5*X3 + np.random.normal(0, 0.2, n_samples) # Some correlation be

# Generate dependent variable
# Assume a linear relationship with some noise
log_odds = 0.5 + 0.3*X1 + 0.4*X2 - 0.2*X3 + 0.1*X4 + 0.2*X5 + np.random.normal(0,
# Convert log odds to probabilities using logistic function
probabilities = 1 / (1 + np.exp(-log_odds))
# Convert probabilities to binary outcome
y = np.random.binomial(1, probabilities)

# Create DataFrame
df = pd.DataFrame({'dependent_variable': y,
                   'independent_variable_1': X1,
                   'independent_variable_2': X2,
                   'independent_variable_3': X3,
                   'independent_variable_4': X4,
                   'independent_variable_5': X5})
```

Logistic Regression Assumptions


TOOLBOX™
KEY ASSUMPTIONS FOR IMPLEMENTING LOGISTIC REGRESSION
The dependent/response variable is binary or dichotomous
Little or no multicollinearity between the predictor/explanatory variables
Linear relationship of independent variables to log odds
Requires sufficiently large sample size
No extreme outliers

The dependent/response variable is binary or dichotomous

The first assumption of logistic regression is that response variables can only take on two possible outcomes - pass/fail, male/female, etc.

This assumption can be checked by simply counting the unique outcomes of the dependent variable. If more than two possible outcomes surface, then one can consider that this assumption is violated.

```
# Check unique values of the dependent variable
print(df['dependent_variable'].unique())
```

```
[1 0]
```

Little or no multicollinearity between the predictor/explanatory variables

This assumption implies that the predictor variables (or the independent variables) should be independent of each other. Multicollinearity relates to two or more highly correlated independent variables. Such variables do not provide unique information in the regression model and lead to wrongful interpretation.

The assumption can be verified with the variance inflation factor (VIF), which determines the correlation strength between the independent variables in a regression model.

```
# Calculate Variance Inflation Factor (VIF) for multicollinearity
def calculate_vif(X):
```