# Learning Logistic Regression with Python: A Step-by-Step Guide

Authored by

**Mohammed looti**

November 6, 2025

## Understanding the Core Principles of Logistic Regression

**Logistic Regression** stands as a cornerstone algorithm in machine learning and statistics, specifically designed for problems where the outcome, or **dependent variable**, is categorical and binary. This means the model aims to predict one of two possible states (e.g., success/failure, 0/1, or in our case, Default/No Default). Crucially, unlike linear regression which predicts a continuous numerical value, logistic regression estimates the probability of an event belonging to the positive class.

The process of fitting a logistic regression model involves determining the optimal set of coefficients for the predictor variables. This optimization is typically achieved using a technique called **Maximum Likelihood Estimation** (MLE). MLE works by selecting the coefficients that maximize the likelihood of observing the actual outcome data given the input features. The successful application of MLE ensures the model parameters are statistically robust and accurately reflect the relationship between the predictors and the outcome probability.

Mathematically, logistic regression transforms the continuous linear combination of predictors into a probability through the use of the sigmoid function. Before this transformation, the model relates the input variables to the **log odds** of the outcome (P(X)) via a standard linear equation. This intermediate step, known as the logit transformation, provides the foundation for the model's structure:

**log = β0 + β1X1 + β2X2 + ... + βpXp**

Within this fundamental logit equation, each component plays a specific role in modeling the probability:

**Xj**: Represents the jth predictor variable (or feature) that influences the outcome.
**βj**: Is the corresponding coefficient estimate, quantifying the change in the log odds associated with a one-unit change in Xj.

To translate the predicted log odds back into a meaningful probability score P(X), which ranges between 0 and 1, we must apply the inverse of the logit function, commonly known as the sigmoid function. This step is essential as it maps the potentially infinite range of the linear combination onto a finite, interpretable probability scale:

p(X) = eβ0 + β1X1 + β2X2 + ... + βpXp / (1 + eβ0 + β1X1 + β2X2 + ... + βpXp)

Once the model calculates P(X) for a given observation, a classification threshold (usually 0.5) is applied. If the calculated probability is greater than or equal to 0.5, the observation is classified as '1' (the positive outcome); otherwise, it is classified as '0' (the negative outcome). The remainder of this guide provides a detailed, practical walkthrough on how to implement and evaluate this

powerful **logistic regression** technique using **Python** and the highly regarded scikit-learn library.

## Step 1: Preparing the Environment by Importing Essential Python Libraries

The initial stage of any data science project involves setting up the computational environment. Before we can begin manipulating data, training our model, or generating visualizations, we must import the necessary collection of Python packages. These libraries provide the specialized functions required for efficient handling of complex datasets and implementation of machine learning algorithms.

Our implementation relies on several core components of the Python data science ecosystem. We utilize **Pandas** for its robust data frame capabilities, essential for loading and structuring our raw data. **NumPy** provides the foundational support for high-performance numerical operations and array manipulation. The **Scikit-learn** library is the primary engine, offering tools for model selection, training, and evaluation, including the dedicated `LogisticRegression` class. Finally, **Matplotlib** is included to facilitate the creation of diagnostic plots, such as the vital ROC curve.

The following code block executes the required imports, ensuring all necessary modules are available for the subsequent steps of data preparation and model building:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import matplotlib.pyplot as plt
```

## Step 2: Acquiring and Exploring the Dataset

To demonstrate the practical application of logistic regression, we will use a well-known public dataset. For this tutorial, we have selected the **Default** dataset, which is frequently used in statistical learning literature and serves as excellent material for a binary classification exercise focused on predicting financial risk. This dataset is derived from the supplementary materials accompanying the seminal textbook, Introduction to Statistical Learning.

The data is readily available as a CSV file hosted on GitHub, allowing for direct and seamless ingestion into our Python environment. We use the `pandas.read_csv` function to load the data directly into a DataFrame object named `data`. Following the loading process, it is standard practice to inspect the first few rows of the DataFrame to verify successful data acquisition, confirm the column names, and understand the initial structure and data types of the features we will be

working with.

The output below shows the commands used to load the data and display the initial observations, confirming that the dataset contains the necessary financial and demographic information required for our prediction task:

**#import dataset from CSV file on Github**
**url = "https://raw.githubusercontent.com/Statology/Python-Guides/main/default.csv"**
**data = pd.read_csv(url)**

#view first six rows of dataset
data

default student balance income
0 0 0 729.526495 44361.625074
1 0 1 817.180407 12106.134700
2 0 0 1073.549164 31767.138947
3 0 0 529.250605 35704.493935
4 0 0 785.655883 38463.495879
5 0 1 919.588530 7491.558572

#find total observations in dataset
len(data.index)

10000

The dataset encompasses 10,000 unique customer observations, each described by four key variables:

**default:** The binary target variable (0 indicating No Default, 1 indicating Default). This is what we aim to predict.
**student:** A categorical indicator, encoded numerically (0 for Non-Student, 1 for Student).
**balance:** The average credit card balance, measured as a continuous numerical feature.
**income:** The individual's annual income, also a continuous numerical feature.

Our goal is to leverage the features--student status, bank balance, and income--to construct a **logistic regression** model capable of calculating the probability of default for any given individual.

## Step 3: Data Partitioning: Creating Training and Testing Sets

A fundamental step in building reliable machine learning models is the process of data partitioning. To ensure that our model's performance evaluation is unbiased and reflects its ability to generalize

to unseen data, we must separate the dataset into distinct subsets. We use the **training set** to fit the model parameters, and the completely independent **testing set** to evaluate its predictive power post-training.

We begin by clearly defining the input features (predictors) and the target variable (response). In this context, the feature matrix **X** consists of the columns 'student', 'balance', and 'income', while the target vector **y** is solely the 'default' column. This separation is crucial before the split occurs.

Next, we utilize scikit-learn's powerful `train_test_split` function. We specify that 70% of the data will be allocated for training and 30% for testing (`test_size=0.3`). To guarantee that anyone running this code will obtain the exact same split, thereby ensuring reproducibility, we set a fixed `random_state`. This practice is considered best standard procedure in applied machine learning workflows.

**#define the predictor variables (X) and the response variable (y)**
**X = data]**
**y = data**

#split the dataset into training (70%) and testing (30%) sets
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=0)

## Step 4: Model Implementation and Training with Scikit-learn

With the data successfully partitioned into training and testing subsets, we are ready to implement and train the **logistic regression** model. Scikit-learn simplifies this process significantly through its unified API, which involves three primary steps: instantiation, fitting, and prediction.

We first instantiate the `LogisticRegression` class, creating an object named `log_regression`. This object encapsulates the necessary algorithms and parameters. By default, scikit-learn handles the internal optimization process, often employing advanced solvers to manage the Maximum Likelihood Estimation efficiently.

The crucial training phase occurs when we invoke the `.fit()` method, passing the training features (`X_train`) and the training target labels (`y_train`). During this step, the model learns the relationship between the predictors and the probability of default, optimizing the coefficient values (β values) that define the model. Once fitting is complete, the model is considered "trained." We then immediately use the trained model's `.predict()` method to generate classification outcomes (`y_pred`) for the independent test set observations.

**#instantiate the model object**
**log_regression = LogisticRegression()**

```
#fit the model using the training data
log_regression.fit(X_train,y_train)

#use the trained model to make predictions on the test data
y_pred = log_regression.predict(X_test)
```

## Step 5: Diagnostic Analysis: Interpreting Model Performance

Evaluating the performance of a classification model is the most vital step, particularly when dealing with potentially imbalanced datasets like financial default data. We must move beyond simple counts of correct predictions and utilize sophisticated metrics to understand where the model succeeds and where it fails. We start by generating the **confusion matrix**, which provides a detailed breakdown of all classification results against the true outcomes.

The confusion matrix is calculated using the actual test labels (`y_test`) and the model's predictions (`y_pred`). This matrix organizes predictions into four categories: True Positives, True Negatives, False Positives, and False Negatives, giving us a complete view of the model's classification behavior across both classes.

**cnf_matrix = metrics.confusion_matrix(y_test, y_pred)**
**cnf_matrix**

```
array(,
])
```

Interpreting the results for the 3,000 observations in the test set reveals critical insights into the model's predictive patterns:

**True Negatives (2886):** Observations that did not default were correctly predicted as non-default.
**False Positives (1):** Observations that did not default were incorrectly predicted as default (Type I Error).
**False Negatives (113):** Observations that defaulted were incorrectly predicted as non-default (Type II Error).
**True Positives (0):** Observations that defaulted were correctly predicted as default.

Following the detailed analysis provided by the confusion matrix, we calculate the overall **accuracy**. While accuracy offers a straightforward percentage of correctly classified instances, it can be highly misleading when the dataset exhibits significant class imbalance--as is often the case with default prediction where non-defaulters far outnumber defaulters.

**print("Accuracy:",metrics.accuracy_score(y_test, y_pred))l**

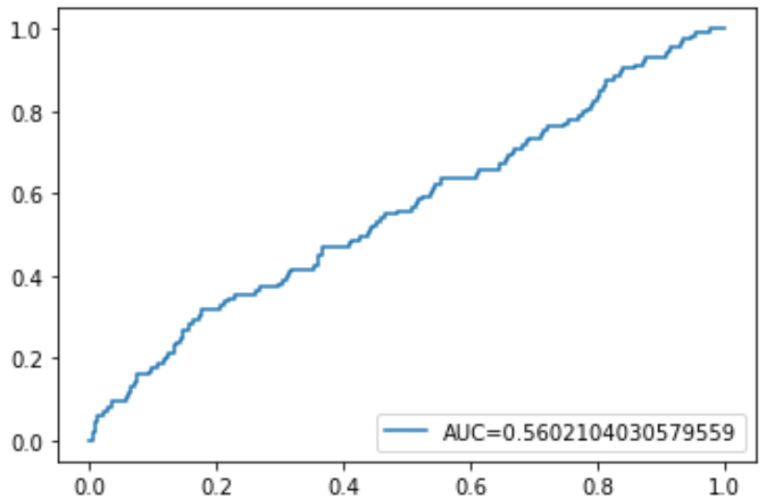
Accuracy: 0.962

An **accuracy** score of 96.2% seems excellent at first glance. However, the confusion matrix exposes a severe limitation: the model completely failed to identify any True Positives (0). This suggests that the high accuracy is simply achieved by predicting the majority class (non-default) most of the time. In high-stakes scenarios like finance, failing to identify actual defaults (high False Negatives) is often more costly than misclassifying a non-defaulter (False Positives).

To obtain a more comprehensive, threshold-independent measure of discriminative capability, we rely on the **Receiver Operating Characteristic (ROC) Curve** and the associated **Area Under the Curve (AUC)**. The ROC curve plots the True Positive Rate against the False Positive Rate across all possible classification thresholds. The AUC score summarizes the entire curve; a value close to 1.0 indicates that the model has excellent ability to distinguish between the positive and negative classes, regardless of the specific threshold chosen.

```
#Calculate probability scores for the positive class (class 1)
y_pred_proba = log_regression.predict_proba(X_test)
fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)

#Create and display the ROC curve plot
plt.plot(fpr,tpr,label="AUC="+str(auc))
plt.legend(loc=4)
plt.show()
```




The resulting **AUC** score provides the final assessment of the model's ranking ability. Even when

the default classification threshold (0.5) leads to zero True Positives, the AUC can still indicate that the model correctly assigns higher probabilities to the positive instances than to the negative instances, suggesting that tuning the classification threshold or addressing the class imbalance might significantly improve the practical utility of the model.

*The complete **Python** source code used in this tutorial can be found here.*

*The complete Python source code used in this tutorial can be found here.*