Learning to Evaluate Classification Models: Building a Confusion Matrix in Python

Authored by

Mohammed looti

November 2, 2025

When developing and assessing classification models, such as logistic regression, which are fundamentally used to predict a **binary** or categorical outcome, rigorous performance evaluation is non-negotiable. Merely achieving a high accuracy score is often insufficient; a deeper mechanism is required to understand the nuances of the model's predictive capability across different classes.

The cornerstone tool for this crucial diagnostic process is the **confusion matrix**. For binary classification tasks, this structure provides an indispensable 2x2 summary table that precisely maps the relationship between the predicted outcomes generated by the model and the true **actual values** (often referred to as ground truth) observed within the test dataset. Analyzing this matrix allows data scientists to move beyond simple metrics and identify specific areas where the model is performing well or failing.

		Predicted	
		0	**1**
Actual	**0**	30	12
	1	8	56

Introduction to Classification Evaluation and the Confusion Matrix

Effective evaluation of any classification system demands a comprehensive view that extends beyond the overall accuracy percentage. We must systematically analyze where the predictive model succeeds and, more importantly, where it struggles to differentiate between the defined classes. The confusion matrix serves this purpose by segmenting all predictions into four fundamental categories: **True Positives (TP)**, **True Negatives (TN)**, **False Positives (FP)**, and **False Negatives (FN)**.

These four components are the foundational data points used to calculate virtually every advanced classification metric, including precision, recall, and the F1 score. By examining the counts of these components, we gain profound insight into potential model biases. For instance, a high count of False Positives suggests the model is overly aggressive in predicting the positive class, while an abundance of False Negatives indicates a reluctance or inability to correctly identify actual positive instances.

To automate the generation and analysis of this critical matrix within the Python ecosystem, practitioners overwhelmingly rely on the robust machine learning capabilities provided by the **scikit-learn** library. This library standardizes the process, enabling efficient and reproducible performance measurement across various modeling techniques.

Implementing the Confusion Matrix using scikit-learn

In a Python environment, the most efficient and accepted methodology for computing the confusion matrix involves utilizing the dedicated `confusion_matrix()` function. This function is conveniently located within the metrics submodule of the powerful scikit-learn (or `sklearn`) package. The function requires two essential arrays as input: the array containing the known **actual values** (ground truth) and the array containing the corresponding **predicted values** outputted by the trained model.

The standard syntax for invoking this essential function is streamlined and designed for rapid integration into any data science workflow. This direct approach allows researchers and developers to swiftly move from model training to performance assessment, making it an industry standard for core classification evaluation.

from sklearn import metrics
metrics.confusion_matrix(y_actual, y_predicted)

The following detailed example illustrates the complete process, beginning with the definition of hypothetical input data and culminating in the generation of the raw confusion matrix object for a typical logistic regression analysis conducted in Python.

Example: Setting Up Data and Generating the Matrix in Python

To effectively demonstrate the implementation, we must first simulate a scenario where a logistic regression model has been trained and subsequently used to generate predictions on a held-out test set. We define two specific arrays: `y_actual`, which holds the verified **actual values**, and `y_predicted`, which contains the probabilistic or hard-coded binary predictions (0 or 1) produced by the model.

These defined arrays represent a small test sample of 20 observations, reflecting a common convention in binary classification where 0 denotes the **negative class** and 1 signifies the **positive class**. It is crucial that the indices of both arrays align perfectly, ensuring that each prediction is compared against its correct ground truth label.

 # Define array of actual values (Ground Truth)
y_actual =

Define array of predicted values (Model Output)
y_predicted =

With these foundational arrays established, we can now seamlessly call the `confusion_matrix()` function sourced from the scikit-learn metrics module. The output is a standard NumPy array, structured to show the count of True Negatives, False Positives, False Negatives, and True Positives, respectively, in its rows and columns.

from sklearn import metrics

# Create confusion matrix object
c_matrix = metrics.confusion_matrix(y_actual, y_predicted)

# Print the resulting NumPy array
print(c_matrix)

]

Interpreting this resulting two-dimensional array provides immediate quantification of model performance: the top-left value (6) represents **True Negatives** (correctly classified as 0), the bottom-right value (8) represents **True Positives** (correctly classified as 1). The off-diagonal elements indicate errors: 4 represents **False Positives** (incorrectly classified as 1 when actually 0), and 2 represents **False Negatives** (incorrectly classified as 0 when actually 1).

Enhancing Visual Readability with Pandas Crosstab

While the raw NumPy array generated by scikit-learn is mathematically precise and efficient for computational use, its lack of explicit labels can pose a challenge for intuitive interpretation, especially when presenting results to non-technical stakeholders or including them in formal reports. To significantly improve the visual clarity and appeal of the confusion matrix, we can effectively employ the `crosstab()` function provided by the ubiquitous **pandas** library.

To leverage `crosstab()`, we must first convert our native Python lists of actual and predicted labels into pandas `Series` objects. Crucially, we assign meaningful descriptive labels, such as 'Actual' and 'Predicted', to these Series. This preparation ensures that the resulting cross-tabulation table is immediately intuitive, with clearly labeled rows and columns that map directly to the classification outcomes.

import pandas as pd

y_actual = pd.Series(y_actual, name='Actual')
y_predicted = pd.Series(y_predicted, name='Predicted')

Create cross-tabulation (confusion matrix)

print(pd.crosstab(y_actual, y_predicted))

Predicted 0 1
Actual
0 6 4
1 2 8

In this visually superior output, the columns unequivocally correspond to the **predicted values** (0 and 1), and the rows represent the **actual values** (0 and 1). This indexed structure makes the identification of True Positives, False Negatives, and the calculation of subsequent performance metrics significantly more straightforward for any user reviewing the analysis.

Calculating Core Classification Metrics

While the raw counts within the confusion matrix are informative, data science practice invariably requires the calculation of derived, normalized metrics to encapsulate overall model effectiveness into quantifiable scores. The scikit-learn package provides dedicated, optimized functions to calculate these essential metrics directly from the input arrays, including overall accuracy, precision, and recall (sensitivity).

These derived scores offer a standardized, percentage-based perspective on performance, allowing for objective comparison between different model architectures, hyperparameters, or data preprocessing steps. By automating these calculations, the risk of manual error is eliminated, ensuring reliable and reproducible results. We will now compute these three pivotal metrics using the same set of input data defined in the preceding examples.

 # Print accuracy score of model
print(metrics.accuracy_score(y_actual, y_predicted))

0.7

# Print precision value of model
print(metrics.precision_score(y_actual, y_predicted))

0.667

# Print recall value of model
print(metrics.recall_score(y_actual, y_predicted))

0.8

Deep Dive into Metric Definitions and Calculation

While the output from the scikit-learn functions provides the necessary numerical scores, a truly comprehensive analysis requires understanding the underlying mathematical relationship between these metrics and the counts within the confusion matrix (where TP=8, TN=6, FP=4, FN=2 in our example). This knowledge is essential for diagnosing why a model performs the way it does.

Below is a precise definition and manual verification of the calculation for these core classification metrics:

Accuracy: This metric quantifies the overall effectiveness of the classifier by measuring the proportion of total observations--both positive and negative classifications--that were correctly predicted across the entire dataset.

Precision: Precision measures the quality of positive predictions. It specifically addresses the question: "Out of all instances the model predicted as positive, what percentage were truly correct?"

Recall (Sensitivity): Recall measures the completeness of the positive identification. It asks: "Out of all the existing actual positive cases in the dataset, how many were successfully captured by the model?"

By using the counts derived from our generated confusion matrix, we can confirm the calculated scores:

Accuracy: (True Positives + True Negatives) / Total Observations = (6 + 8) / (6 + 4 + 2 + 8) = 14 / 20 = **0.7**

Precision: True Positives / (True Positives + False Positives) = 8 / (8 + 4) = 8 / 12 ≈ **0.667**

Recall: True Positives / (True Positives + False Negatives) = 8 / (2 + 8) = 8 / 10 = **0.8**

The strategic importance of prioritizing either Precision or Recall is entirely contingent upon the specific goals and cost function of the business problem. For example, in critical medical diagnosis systems, maximizing **Recall** (thereby minimizing dangerous False Negatives) is paramount. Conversely, in systems like spam detection or fraud alerts, maximizing **Precision** (minimizing disruptive False Positives) is often the primary objective.

Additional Resources for Further Study

To further solidify your expertise in evaluating and optimizing classification models, we highly recommend consulting the official documentation for the foundational Python libraries

demonstrated in this tutorial. These resources offer comprehensive technical details, expanded functionalities, and best practice guidelines for advanced deployment.

Key documentation sources include:

The scikit-learn Model Evaluation documentation for detailed metric definitions.

The official pandas documentation for data manipulation and visualization techniques using `crosstab()`.