How to Easily Generate a Confusion Matrix in Python with scikit-learn

Authored by

stats writer

December 3, 2025

Evaluating the performance of classification models is a critical step in the machine learning workflow. Unlike regression tasks, where metrics like Mean Squared Error suffice, classification requires specialized tools to assess how well a model discriminates between classes. The primary tool for this assessment in binary and multi-class problems is the Confusion Matrix. To effectively implement and utilize this matrix within your data science projects, familiarity with the Python ecosystem, particularly the powerful sklearn.metrics library, is essential.

The process of generating a **Confusion Matrix** in Python is streamlined through the Scikit-learn framework. This library provides the necessary functions to compare the actual values (true labels) from a test dataset against the values predicted by your trained model. Beyond the matrix itself, sklearn.metrics also offers the **classification_report** function, which delivers a concise summary of key performance indicators, including Precision, Recall, and the F1-score for every class. Furthermore, while Scikit-learn provides the numerical data, libraries like Matplotlib or Seaborn are often utilized to generate compelling, easily interpretable visualizations of the resulting matrix.

The Role of the Confusion Matrix in Classification

Understanding the fundamental nature of the problem being solved is key to performance evaluation. For instance, when dealing with a binary response variable--a scenario where the outcome can only be one of two states (e.g., success/failure, yes/no)--we often employ Logistic Regression. While labeled as a regression technique, its application is purely for classification purposes. Evaluating the quality and predictive power of such a classification model is paramount to ensuring its reliability in real-world deployment.

The most common and fundamental technique for evaluating these models is the construction of the Confusion Matrix. This matrix provides a complete picture of the model's performance across all possible outcomes. It is typically structured as a 2x2 table for binary classification, although it scales up for multi-class problems. The matrix systematically compares the predicted values generated by the statistical model against the known, actual values present in the test dataset, exposing where the model excels and where it makes critical errors.

Each cell within the 2x2 matrix represents a specific interaction between the predicted and actual classes: **True Positives (TP)**, **True Negatives (TN)**, **False Positives (FP)**, and **False Negatives (FN)**. These four values serve as the building blocks for nearly all advanced classification metrics. By aggregating these results, the matrix provides a comprehensive, granular assessment that simple overall metrics like raw Accuracy often obscure. A careful examination of these four components allows data scientists to tailor optimization efforts based on whether minimizing false alarms or missed detections is the higher priority for the specific application.

Defining the Confusion Matrix Structure

The structure of the **Confusion Matrix** is standardized to allow for consistent interpretation across different models and datasets. The rows of the matrix represent the actual classes (the ground truth), while the columns represent the classes predicted by the model. This layout ensures that we can quickly discern the type and frequency of errors made by the classifier. The diagonal elements--True Positives and True Negatives--indicate correct predictions, while the off-diagonal elements represent misclassifications.

Let's formally define the four components of a binary Confusion Matrix:

True Positives (TP): The number of instances correctly predicted as belonging to the positive class. (Actual = 1, Predicted = 1)
True Negatives (TN): The number of instances correctly predicted as belonging to the negative class. (Actual = 0, Predicted = 0)
False Positives (FP): Also known as a Type I error. The number of instances incorrectly predicted as positive when they were actually negative. (Actual = 0, Predicted = 1)
False Negatives (FN): Also known as a Type II error. The number of instances incorrectly predicted as negative when they were actually positive. (Actual = 1, Predicted = 0)

Visualizing this structure helps solidify understanding. The image below depicts the standard layout, illustrating how the model's predictions align with the reality of the test data. This foundation is crucial before moving on to the practical implementation in Python, as interpreting the generated numerical output directly relates back to these definitions.

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

Generating the Basic Confusion Matrix using Scikit-learn

In Python, the industry standard library for machine learning--Scikit-learn (often imported as sklearn)--provides a robust and efficient way to calculate the required metrics. Specifically, we leverage the **confusion_matrix()** function found within the sklearn.metrics package. This function requires two essential inputs: the array of actual values (**y_actual**) and the array of predicted values (**y_predicted**).

The simplicity of the implementation belies the power of the function. By merely feeding the true labels and the model's output into **confusion_matrix()**, the library handles all the complex counting and tabulation required to structure the matrix correctly. The output is a NumPy array representing the 2x2 (or NxN) structure of the matrix, ready for immediate interpretation or further processing.

To use this capability, ensure you have Scikit-learn installed in your environment. The basic syntax for calling the function is straightforward, demonstrating the minimal code required to access this crucial analytical tool:

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

The forthcoming example will demonstrate this process in detail, showing how to define the input arrays and interpret the resulting matrix array, laying the groundwork for more advanced analysis and visualization.

Example: Implementing the Confusion Matrix in Python

To illustrate the practical application of the **confusion_matrix()** function, let us define a representative scenario. Suppose we have trained a Logistic Regression model designed to predict a binary outcome (e.g., whether a customer will churn or not). We have run this model on a test set and generated a series of predictions. We must now compare these predictions against the known ground truth to assess the model's performance.

We begin by defining two arrays. The first array, **y_actual**, contains the true outcomes for the 20 samples in our test set. The second array, **y_predicted**, holds the corresponding classifications made by our model. These arrays serve as the fundamental data required for calculating the matrix elements. Note the presence of both correct and incorrect predictions across the two lists, which the matrix calculation will summarize.

#define array of actual values
y_actual =

#define array of predicted values
y_predicted =

Using the **confusion_matrix()** function from sklearn, we can now rapidly generate the necessary numerical structure that summarizes these 20 predictions. The process involves importing the necessary module, invoking the function with our data, and printing the resulting NumPy array to

the console. This array immediately gives us the counts for TP, TN, FP, and FN based on the standard matrix layout (where the rows are true labels and columns are predicted labels, and often 0=Negative, 1=Positive).

from sklearn import metrics

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

#print confusion matrix
print(c_matrix)

]

Interpreting this output array **,]** reveals the following distribution: 6 True Negatives (correctly predicted 0s), 4 False Positives (incorrectly predicted 1s), 2 False Negatives (incorrectly predicted 0s), and 8 True Positives (correctly predicted 1s). This raw numerical representation is the core output required for subsequent calculations of performance metrics.

Visualizing the Matrix with Pandas Crosstab

While the NumPy array output provided by Scikit-learn is mathematically precise, it lacks the intuitive labels required for quick analysis, especially when sharing results with non-technical stakeholders. A more human-readable format can be generated using the statistical features available in the pandas library, specifically the **crosstab()** function. This function allows us to create a contingency table that clearly labels the rows (Actual) and columns (Predicted), transforming the raw array into a structured dataframe.

To use **crosstab()** effectively, the actual and predicted lists must first be converted into pandas Series, ensuring they are properly labeled before tabulation. This transformation ensures that the final printed output is not just a matrix of numbers, but a descriptive table that explicitly defines what each row and column represents, making the interpretation instantaneous and unambiguous.

import pandas as pd

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

#create confusion matrix
print(pd.crosstab(y_actual, y_predicted))

Predicted 0 1

Actual
0 6 4
1 2 8

As clearly demonstrated by the pandas output, the columns definitively show the predicted values for the response variable (0 or 1), and the rows show the actual values (0 or 1). This visual enhancement is particularly useful when dealing with multi-class problems where simply reading index numbers in a NumPy array becomes cumbersome. The clarity provided by **crosstab()** allows for efficient communication of results and error analysis.

Interpreting Performance Metrics: Accuracy, Precision, and Recall

The real value of the Confusion Matrix lies in its ability to serve as the foundation for calculating specialized performance metrics that move beyond simple raw counts. Scikit-learn provides dedicated functions within the **metrics** module to calculate Accuracy, Precision, and Recall directly from the actual and predicted labels. These metrics offer differing perspectives on model quality, allowing analysts to select the most appropriate evaluation metric based on the business objective.

Using the same **y_actual** and **y_predicted** arrays from our previous example, we can calculate these three core metrics instantly. This demonstrates the efficiency of using built-in library functions over manually writing the calculation formulas, ensuring consistency and correctness across different experiments. The **accuracy_score** function provides the overall ratio of correct predictions, while **precision_score** and **recall_score** focus specifically on the performance concerning the positive class (often the class of interest).

#print accuracy 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

It is crucial to have a clear understanding of what each calculated metric represents in the context

of classification errors. They address different aspects of model correctness, and choosing which metric to optimize depends heavily on the cost associated with False Positives versus False Negatives in the specific application. For example, in medical diagnosis, maximizing Recall (minimizing missed detections) is usually prioritized over maximizing Precision.

Refresher on Core Classification Metrics

For clarity and reference, here is a concise refresher on the mathematical definitions of Accuracy, Precision, and Recall, expressed in terms of the components of the **Confusion Matrix**:

Accuracy: Percentage of total predictions that were correct (TP + TN) / Total Samples. This is the simplest measure of overall correctness.
Precision: Measures the quality of the positive predictions. It is the ratio of correct positive predictions (TP) relative to the total number of instances predicted as positive (TP + FP). A high precision means low false alarms.
Recall: Measures the completeness of the positive predictions. It is the ratio of correct positive predictions (TP) relative to the total number of actual positive instances (TP + FN). A high recall means fewer missed detections.

Applying these formulas to the results derived from our sample Confusion Matrix **,]**, where TN=6, FP=4, FN=2, and TP=8, we can confirm the calculations performed by sklearn. This manual calculation provides validation and deepens the understanding of how the metrics relate to the raw counts:

Accuracy: (6 + 8) / (6 + 4 + 2 + 8) = 14 / 20 = **0.7**
Precision: 8 / (8 + 4) = 8 / 12 ≈ **0.667**
Recall: 8 / (2 + 8) = 8 / 10 = **0.8**

These calculations confirm that our model, while achieving 70% overall accuracy, is better at finding positive instances (Recall = 80%) than it is at ensuring that its positive predictions are always correct (Precision = 66.7%). This trade-off analysis is the fundamental purpose of the Confusion Matrix and its derived metrics in classification modeling.