Quadratic Discriminant Analysis in Python (Step-by-Step)

Authored by

stats writer

December 19, 2025

Quadratic Discriminant Analysis (QDA) is recognized as a sophisticated parametric classification technique widely employed in the field of machine learning. Its core purpose is to optimally segregate observations into distinct classes by modeling the probability distribution of the predictor variables within each category. Critically, QDA assumes that the data within each class follows a Gaussian distribution but permits each class to possess its own unique covariance matrix, leading to the formation of non-linear, or quadratic, decision boundaries that are necessary for complex data separation. Effective implementation of QDA in Python relies heavily on the robust functionalities provided by the Scikit-learn library. The typical implementation pipeline requires a series of organized steps: importing necessary packages, preparing and loading the data, fitting the QDA model, generating predictions, and rigorously evaluating the model's performance metrics. Understanding and accurately executing this sequence of steps enables practitioners to apply this powerful classification tool to virtually any multivariate dataset, yielding reliable and often highly accurate classification results.

QDA serves as a powerful extension and generalization of Linear Discriminant Analysis (LDA). While both methodologies aim to model class-conditional densities and utilize Bayes' theorem for classification, LDA operates under the restrictive assumption that all classes share an identical covariance structure. When this assumption holds true, LDA produces linear decision boundaries. Conversely, QDA relaxes this constraint, allowing the covariance matrices to vary freely between classes. This flexibility is paramount when dealing with real-world data where the features within different classes exhibit varying degrees of correlation and variance, ultimately enabling QDA to delineate class separations using complex, curved surfaces, thus significantly improving classification accuracy in non-linear scenarios. This tutorial is designed to provide a comprehensive, step-by-step framework for successfully executing quadratic discriminant analysis within a Python environment, ensuring clarity and practical application throughout the process.

Theoretical Foundations and Prerequisites

Before diving into the Python implementation, it is essential to appreciate the theoretical underpinnings that guide the application of QDA. QDA is inherently a parametric method, meaning it assumes a specific functional form for the distribution of the data, which in this case is the multivariate Gaussian distribution for each class. The classification rule is derived by minimizing the total probability of misclassification. This is achieved by estimating the mean vector and the covariance matrix for each individual class based on the training data. The resulting decision boundary between any two classes, say k and l, is determined by the set of points where the posterior probability of belonging to class k equals the posterior probability of belonging to class l, leading mathematically to a quadratic equation defining the boundary surface.

Successful implementation requires a dataset with clearly defined predictor variables (features)

and a categorical response variable that needs classification. It is important to remember that QDA performs optimally when the number of observations is reasonably large compared to the number of features, as accurately estimating the large number of parameters within the class-specific covariance matrices demands substantial data. Furthermore, while QDA is robust, data scaling and handling of outliers remain crucial preprocessing steps to ensure the Gaussian assumptions are not severely violated. For this tutorial, we will use a well-known, clean dataset that minimizes the need for extensive preprocessing, allowing us to focus specifically on the QDA modeling procedure itself.

Step 1: Loading Essential Python Libraries

The initial step in any machine learning project within Python is to ensure all necessary libraries are imported into the working environment. For Quadratic Discriminant Analysis using the Scikit-learn framework, we require modules for model selection, the specific discriminant analysis function, data handling, and visualization capabilities. We must import functions related to splitting data for training and testing, tools for rigorous cross-validation, the QDA estimator itself, and foundational libraries like NumPy and Pandas for numerical operations and data structuring, respectively.

The following code block meticulously imports all required components.

tags are used here to maintain the exact formatting of the code, which is essential for execution. These imports establish the foundation upon which the entire modeling process will be built, ensuring that all subsequent functions, such as data loading and model fitting, are correctly recognized and executed by the Python interpreter.

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn import datasets
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Step 2: Data Acquisition and Formatting

For demonstrating the practical application of QDA, we will leverage the renowned Iris dataset, a classic benchmark in classification tasks, readily available within the Scikit-learn library's datasets module. While the dataset is loaded natively by Scikit-learn, converting it into a Pandas DataFrame

is highly recommended. This conversion provides numerous benefits, including easier data manipulation, clearer column naming, and enhanced readability, which significantly simplifies the subsequent steps of variable assignment and data exploration. The code below performs the loading of the raw data and transforms it into the flexible DataFrame structure, assigning appropriate column names for clarity.

The transformation process involves concatenating the feature measurements (sepal and petal dimensions) and the numerical target classes into a unified NumPy array, which is then cast into a Pandas DataFrame. Furthermore, the numerical target codes (0, 1, 2) are mapped back to their descriptive species names (setosa, versicolor, virginica) to make the resulting classification predictions intuitively understandable. This preparation phase is crucial for ensuring data quality and organization before model training commences. After structuring the data, we perform a quick inspection of the first few rows using the head() method and confirm the total number of observations to gain immediate insights into the dataset's scale and composition.

#load *iris* dataset
iris = datasets.load_iris()

#convert dataset to pandas DataFrame
df = pd.DataFrame(data = np.c_, iris],
columns = iris +)
df = pd.Categorical.from_codes(iris.target, iris.target_names)
df.columns =

#view first six rows of DataFrame
df.head()

s_length s_width p_length p_width target species
0 5.1 3.5 1.4 0.2 0.0 setosa
1 4.9 3.0 1.4 0.2 0.0 setosa
2 4.7 3.2 1.3 0.2 0.0 setosa
3 4.6 3.1 1.5 0.2 0.0 setosa
4 5.0 3.6 1.4 0.2 0.0 setosa

#find how many total observations are in dataset
len(df.index)

150

The output confirms that the dataset contains a total of 150 complete observations, providing a solid foundation for training our classification model. We can observe the predictor variables--sepal

length, sepal width, petal length, and petal width--alongside the numerical target and the final categorical species label.

Step 3: Defining Predictor and Response Variables

With the data successfully loaded and formatted, the next critical step is to explicitly delineate the roles of the variables: which ones will serve as predictors, and which one is the response variable we intend to classify. For this specific demonstration, the objective is to construct a Quadratic Discriminant Analysis model capable of classifying the species of a given flower based on its morphological measurements.

We designate the four physical measurements as the set of independent variables, typically denoted as X. These predictor variables provide the features that the QDA model will use to estimate the probability distributions for each class.

Sepal length

Sepal width

Petal length

Petal width

The dependent variable, y, is the *Species* column. This response variable represents the outcome we are attempting to predict and takes on one of the following three distinct classes, requiring the QDA model to learn the complex quadratic boundaries separating them:

setosa

versicolor

virginica

Step 4: Initializing and Fitting the QDA Model

The fourth step involves the technical creation and training of the QDA model. Using the variables defined in the previous step, we isolate the features (X) and the target (y). The actual QDA model instance is created using the QuadraticDiscriminantAnalysis class imported from Scikit-learn's discriminant_analysis module. Initialization is typically straightforward as QDA in Scikit-learn does not require complex hyperparameter tuning for its basic implementation.

Once the model object is instantiated, the crucial step of fitting the model begins. The fit(X, y) method processes the training data, calculating the class-specific mean vectors and covariance

matrices. This process is essentially the learning phase where the model internalizes the data structure necessary to determine the quadratic decision boundaries that define optimal class separation. The efficiency of Scikit-learn ensures this fitting process is rapid and robust for standard datasets.

#define predictor and response variables
X = df]
y = df

#Fit the QDA model
model = QuadraticDiscriminantAnalysis()
model.fit(X, y)

Step 5: Evaluating Model Performance through Cross-Validation

A fitted model is only useful if its performance can be reliably measured and validated against unseen data. A standard and highly effective technique for robust performance assessment is cross-validation, specifically Repeated Stratified K-Fold cross-validation. This method ensures that the model's accuracy score is not overly dependent on a single, arbitrary train-test split, providing a more generalized estimate of performance across the entire dataset. Stratification ensures that the proportion of classes remains consistent across all folds, which is vital for balanced classification problems.

In this example, we define a cross-validation strategy using 10 folds and 3 repetitions. This means the dataset will be split 10 times, and the entire 10-fold process will be repeated three times, yielding 30 distinct accuracy scores. These scores are then averaged to produce a final, highly reliable estimate of the model's predictive accuracy. We utilize the cross_val_score function from Scikit-learn, specifying the QDA model, the data (X and y), the scoring metric (accuracy), and the defined cross-validation object (cv).

#Define method to evaluate model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

#evaluate model
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print(np.mean(scores))

0.97333333333334

The output of the evaluation demonstrates an impressive mean accuracy of approximately **97.33%**.

This high score indicates that the QDA model, trained on the features of the Iris dataset, is exceptionally effective at correctly classifying the three flower species. This result confirms the suitability of the QDA approach for this specific non-linear classification problem.

Step 6: Utilizing the Model for Novel Predictions

The final objective of any classification model is its application to classify new, previously unseen data points. Having established the QDA model's high accuracy through rigorous cross-validation, we can now confidently utilize the trained model object to make predictions on novel observations. This step demonstrates the practical utility of the derived quadratic decision boundaries.

We define a synthetic new observation representing the measurements of an unknown flower, structured as a list of four features: sepal length, sepal width, petal length, and petal width. This observation is then passed to the model's predict() method. The model applies the learned classification rules--the class means and covariance matrices--to determine the highest posterior probability, assigning the observation to the most likely species category.

#define new observation
new =

#predict which class the new observation belongs to
model.predict()

array(, dtype='<U10')

Based on the input measurements , the QDA model confidently predicts that this new observation belongs to the species setosa. This successful prediction validates the operational capability of the fitted model for real-world application scenarios.

Conclusion and Next Steps

This comprehensive guide has demonstrated the entire process of performing Quadratic Discriminant Analysis in Python using the powerful Scikit-learn library. We began with the theoretical justification for using QDA over LDA when dealing with non-linear class separations, meticulously prepared the necessary libraries and data, defined the roles of the predictor and response variables, trained the QDA model, and rigorously validated its exceptional performance using repeated stratified cross-validation. Finally, we showcased the practical application of the trained model by successfully classifying a new, unseen data point.

QDA proves to be an indispensable tool in the data scientist's arsenal, particularly when the underlying data distributions suggest unequal variance across classes, demanding non-linear

boundaries. For those seeking to further explore or replicate this analysis, the complete Python script used throughout this tutorial is made available for download and review. Mastering this workflow allows for the confident application of QDA to more complex, high-dimensional datasets where accurate classification is paramount.

You can find the complete Python code used in this tutorial here.