How to Perform Linear Discriminant Analysis in Python (Step-by-Step)

Authored by

stats writer

December 19, 2025

Linear Discriminant Analysis (LDA) is a powerful statistical technique commonly employed in machine learning for classification and dimensionality reduction. Its primary function is to find linear combinations of features that characterize or separate two or more classes of objects or events. Unlike traditional regression, LDA explicitly models the differences between classes based on the means of the variables, providing a robust method for distinguishing between groups.

Implementing LDA efficiently requires robust libraries. Fortunately, the scikit-learn library in Python provides the streamlined `LinearDiscriminantAnalysis` class, making complex modeling accessible. This comprehensive guide serves as an expert tutorial, walking you through every crucial stage: from ensuring proper data preparation to fitting the model, evaluating its performance, and visualizing the final discriminant results.

By the end of this article, you will possess a solid, working understanding of how to apply this technique to a real-world dataset, specifically classifying data using multi-class categorization methods inherent to LDA.

At its core, Linear Discriminant Analysis (LDA) is most effective when the goal is classification. It operates by identifying a set of independent, or **predictor variables**, which can be used to predict the value of a categorical response variable. A crucial requirement for utilizing LDA is that the response variable must have two or more distinct classes, making it ideal for multi-class classification problems.

We will demonstrate the implementation of LDA using a standardized, step-by-step approach optimized for clarity and reproducibility in the Python programming environment. This practical example will ensure you can quickly adapt these techniques to your own machine learning projects.

Step 1: Preparing the Environment by Loading Essential Libraries

The initial step in any machine learning project involves setting up the environment by importing the required dependencies. For this LDA tutorial, we rely heavily on **scikit-learn** for model building and evaluation, **Pandas** for efficient data manipulation, and **NumPy** for numerical operations.

Specifically, we import `LinearDiscriminantAnalysis` and several classes from `model_selection`, such as `RepeatedStratifiedKFold` and `cross_val_score`, which are critical for robust model evaluation through cross-validation. We also include `datasets` to load our sample data and `matplotlib.pyplot` for visualization in a later step.

Execute the following block of code to ensure all necessary libraries are loaded into your Python environment before proceeding:

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 LinearDiscriminantAnalysis
from sklearn import datasets
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Step 2: Data Acquisition and Preprocessing using the Iris Dataset

To demonstrate the capabilities of LDA, we will utilize the renowned Iris dataset, which is conveniently packaged within the scikit-learn library. This dataset is a classic benchmark for classification tasks, containing 150 observations of iris flowers, each categorized into one of three species based on four measured features.

The code below first loads the raw data, then converts it into a structured Pandas DataFrame. This conversion step is crucial for readability and ease of manipulation, allowing us to manage feature names and categorical targets efficiently. We rename the columns for clarity and inspect the first few rows to verify the data structure.

Upon inspection, the dataset confirms 150 total observations. Our objective is to construct an LDA model capable of classifying the flower's species based on its physical measurements. The model will utilize four crucial features as **predictor variables**, aiming to predict the categorical **response variable**, *Species*.

```
#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 four predictor variables used in our model are:

Sepal length (s_length)
Sepal width (s_width)
Petal length (p_length)
Petal width (p_width)

These features will be used to classify the response variable, *Species*, into one of the following three potential classes:

setosa
versicolor
virginica

Step 3: Separating Features and Training the Linear Discriminant Analysis Model

The next critical phase involves defining our feature matrix (X) and our target vector (y) from the prepared Pandas DataFrame. The feature matrix **X** consists of the four independent measurements (lengths and widths), while the target vector **y** holds the class labels (species).

Once the variables are separated, we instantiate the `LinearDiscriminantAnalysis` class, which is imported from the `sklearn.discriminant_analysis` module. The model is then trained--or "fitted"--to the data using the `.fit(X, y)` method. This process calculates the optimal discriminant functions that maximize the separability between the different species classes in the feature space.

The following code snippet executes the feature separation and initiates the training process for the LDA model:

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

#Fit the LDA model

```
model = LinearDiscriminantAnalysis()
model.fit(X, y)
```

Step 4: Evaluating Model Performance using Cross-Validation

After the LDA model is fitted, the next logical step is rigorous evaluation to ensure its reliability and generalization capability. We employ **Repeated Stratified K-Fold Cross-Validation**, a robust technique that partitions the dataset multiple times while maintaining the proportion of class labels in each fold. This approach minimizes bias and variance in the estimation of prediction accuracy.

For this specific evaluation, we set the parameters to use 10 folds (`n_splits=10`) and repeat this process 3 times (`n_repeats=3`). The `cross_val_score` function, provided by scikit-learn, calculates the classification **accuracy** across all iterations.

Executing the evaluation code yields a mean accuracy score. In this instance, the model achieved a high mean accuracy of **97.78%**, indicating excellent performance in classifying the known species data. This confirms that the linear boundaries established by the LDA are highly effective at separating the three iris species based on the four predictor variables.

```
#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))
```

```
0.9777777777777779
```

Applying the Model for New Predictions

Beyond statistical validation, the primary utility of a trained model is its capacity to predict the class of unseen, new observations. We can define a hypothetical new flower measurement and use the `.predict()` method of our trained model to classify it instantly.

For example, if we input measurements corresponding to sepal length, sepal width, petal length, and petal width, the model determines the most probable species class. The output shows that this particular set of dimensions is predicted to belong to the species *setosa*, demonstrating the model's practical application in real-time classification tasks.

```
#define new observation
```

new =

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

array(, dtype='<U10')

Step 5: Visualizing Class Separation with the LDA Plot

A key advantage of Linear Discriminant Analysis is its ability to serve as a dimensionality reduction technique, projecting high-dimensional data onto a lower-dimensional subspace (the linear discriminants) while maximizing class separation. Since the Iris dataset has three classes, the LDA transformation generates two discriminant dimensions (C-1 dimensions, where C is the number of classes). Visualizing the data in this new space clearly shows the effectiveness of the classification.

We use **Matplotlib** to create a scatter plot of the transformed data. The code below first refits the LDA model and then applies the `.transform(X)` method to project the original four features onto the two primary linear discriminants. Each species is assigned a distinct color (red, green, or blue), allowing us to immediately observe how well the model has separated the groups.

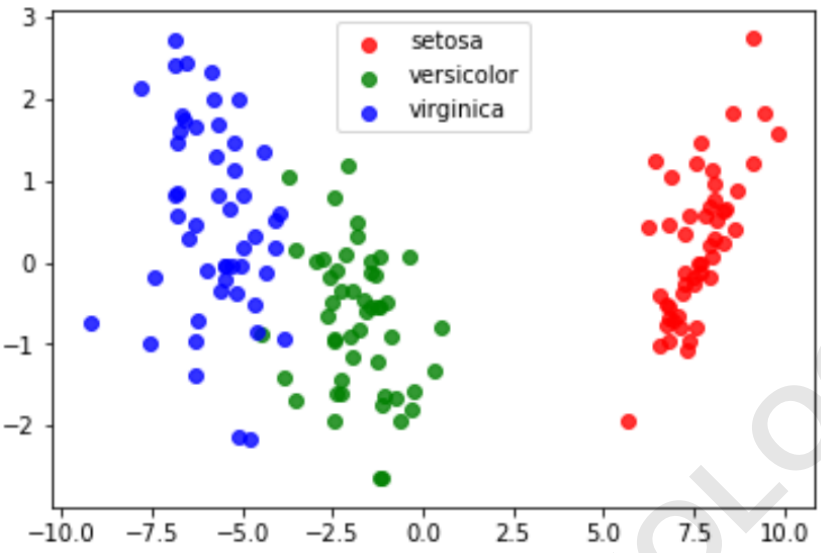
As visualized in the resulting plot, the three species--setosa, versicolor, and virginica--are highly distinct and separated along the linear discriminant axes. The clear clustering validates the high accuracy score achieved in Step 4, confirming that the LDA model is extremely effective for this specific classification problem.

#define data to plot
X = iris.data
y = iris.target
model = LinearDiscriminantAnalysis()
data_plot = model.fit(X, y).transform(X)
target_names = iris.target_names

#create LDA plot
plt.figure()
colors =
lw = 2
for color, i, target_name in zip(colors, , target_names):
plt.scatter(data_plot, data_plot, alpha=.8, color=color,
label=target_name)

```
#add legend to plot
plt.legend(loc='best', shadow=False, scatterpoints=1)

#display LDA plot
plt.show()
```



Conclusion: Leveraging LDA for Multi-Class Classification

This tutorial successfully demonstrated the end-to-end implementation of Linear Discriminant Analysis in Python using the powerful scikit-learn library. We covered essential steps, from importing specialized modules and preparing data in a Pandas DataFrame, to training a high-accuracy classification model and validating its performance through cross-validation.

LDA proves to be an exceptionally effective and interpretable tool, especially when dealing with normally distributed data and the requirement to distinctly separate multiple classes. The visualization step further solidifies the understanding of how LDA transforms complex data into a simplified, yet highly informative, lower-dimensional space.

For those interested in reviewing or utilizing the complete, runnable source code featured in this tutorial, it is available for direct access via the external link below.

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