Classification: Linear Discriminant Analysis (LDA)

Rafiq Islam

2024-10-17

Table of contents

## Introduction

Linear Discriminant Analysis (LDA) is a supervised machine learning algorithm commonly used for classification tasks. It is widely applied when dealing with datasets where the number of predictors (features) exceeds the number of observations, or when multicollinearity is a concern. LDA works by projecting data onto a lower-dimensional space, maximizing the separation between classes.

## Mathematical Foundation of LDA

Let’s assume we have a dataset consisting of data points and features, and each data point belongs to one of distinct classes. The goal of LDA is to find a new space (called a discriminant space) in which the classes are maximally separated, i.e. we want to **maximize the separability between classes** while **minimizing the variation within each class**. This can be mathematically expressed as finding a projection that maximizes the ratio of between-class variance to within-class variance.

For each class (where ):

* is the **mean vector** of class .
* is the **overall mean** of the entire dataset.

**Class Mean**: For each class , the mean is calculated as:

where is the number of data points in class , and represents individual data points.

**Overall Mean**: The mean of the entire dataset is:

To understand how well classes are separated, we need two key measures:

1. **Within-Class Scatter Matrix**   
   The within-class scatter matrix measures how the data points of each class deviate from the class mean. It captures the **spread of data points within each class**. For class , the scatter matrix is calculated as:

* This formula is saying that for each class , we calculate the distance of every point from the mean of its class , and then sum these squared distances across all classes.

1. **Between-Class Scatter Matrix**   
   The between-class scatter matrix measures how the **class means deviate from the overall mean**. It captures how well-separated the classes are.

* In this case, for each class , we calculate the distance between the mean of class and the overall mean , then scale this by the number of points in class .

LDA aims to find a transformation that maximizes the separation between classes. This is done by finding a linear projection such that the **between-class scatter is maximized** and the **within-class scatter is minimized**. Mathematically, the optimization problem becomes:

* captures the between-class variance (how well-separated the classes are in the new projection).
* captures the within-class variance (how tightly packed the points of the same class are in the new projection).

This ratio is known as the **Fisher’s discriminant ratio**. The goal is to find that maximizes this ratio. To maximize the Fisher’s discriminant ratio, we need to solve the following generalized eigenvalue problem:

Here, is the vector that defines the linear combination of features that maximizes class separation, and is an eigenvalue that represents how much variance is explained by that direction.

The solution to this equation gives us the eigenvectors (directions) and eigenvalues (variances) of the transformed space. We select the top eigenvectors corresponding to the largest eigenvalues to form the projection matrix .

## Dimensionality Reduction

The LDA transformation reduces the dimensionality of the data by projecting it onto a subspace spanned by the eigenvectors with the largest eigenvalues. For a dataset with classes, LDA can reduce the data to at most dimensions because has rank . If we have two classes, LDA will reduce the data to a one-dimensional subspace. For three classes, LDA can project the data onto a two-dimensional subspace, and so on.

Now before diving into the python code, let’s do some math by hand so that we can understand the skeleton of the process. Let’s create a small dataset with 6 features and 4 observations divided into 3 classes. We will use this dataset to manually go through the Linear Discriminant Analysis (LDA) process step by step.

### Dataset

| Observation | Feature 1 | Feature 2 | Feature 3 | Feature 4 | Feature 5 | Feature 6 | Class |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 |  |
|  | 3 | 4 | 5 | 6 | 7 | 8 |  |
|  | 6 | 5 | 4 | 3 | 2 | 1 |  |
|  | 7 | 6 | 5 | 4 | 3 | 2 |  |

Now, we’ll walk through the mathematical steps of LDA for this small dataset.

### 1. Compute Class Means for each class:

* Class (mean of and ):
* Class (only one observation ):
* Class (only one observation ):

### 2. Compute Overall Mean :

We compute the overall mean , which is the average of all observations from all classes:

### 3. **Compute the Within-Class Scatter Matrix** :

For each class , the within-class scatter matrix is computed as:

For , the within-class scatter matrix is:

$$
(x\_1 - \mu\_1) = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \\ 4.5 \\ 5.5 \\ 6.5 \\ 7.5 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}; \hspace{6mm} (x\_2 - \mu\_1) = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \\ 4.5 \\ 5.5 \\ 6.5 \\ 7.5 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}
$$

For class , the scatter matrix is:

For classes and , there is only one data point in each, so there is no within-class scatter:

Thus, the total within-class scatter matrix is:

### 4. Compute the Between-Class Scatter Matrix :

For each class , the between-class scatter matrix is computed as:

For class (where ):

Thus, for :

For (where ):

The between-class scatter matrix for is:

For (where ):

The between-class scatter matrix for is:

Total Between-Class Scatter Matrix :

Adding the matrices gives:

### 5. Solve the Eigenvalue Problem:

We now solve the eigenvalue problem:

The solution to this eigenvalue problem gives us the eigenvalues (which quantify the amount of variance captured in each direction) and the eigenvectors (which give the directions of maximum class separation). The eigenvector corresponding to the largest eigenvalue defines the direction of the first discriminant axis, which is the direction that maximally separates the classes.

The eigenvalues of the matrix are:

The two largest eigenvalues are:

The corresponding eigenvectors for the two largest eigenvalues are:

By projecting the data onto the eigenvector , we transform the original dataset into a lower-dimensional space where class separability is maximized. For this dataset, since there are 3 classes, LDA will find up to discriminant axes. Let’s see how.

The matrix formed by the two largest eigenvectors is:

This matrix represents the projection directions corresponding to the two largest eigenvalues in the Linear Discriminant Analysis process. With the eigenvectors and , we can now project our original dataset onto the new 2D subspace. Now, let represent our original dataset (where each row corresponds to an observation and each column to a feature). The projection of the original data onto the new 2D subspace is given by:

Where:

* is the matrix (4 observations, 6 features),
* is the matrix of eigenvectors.

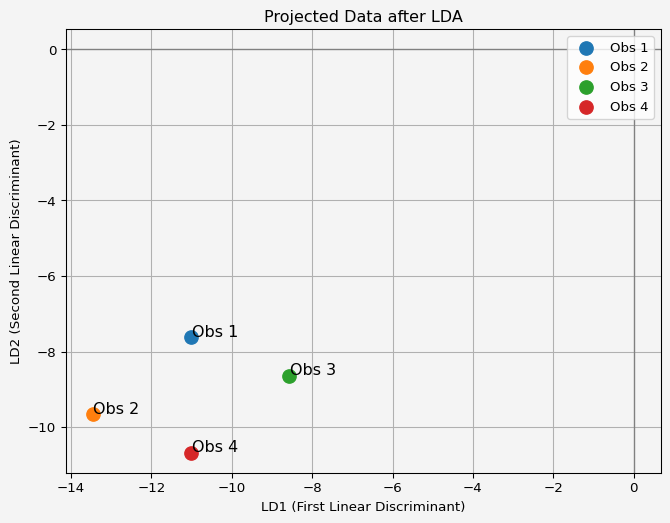
After multiplying by , we obtain the projected data matrix , which is a matrix (4 observations, 2 features):

This matrix represents the data in the new 2D space where class separability is maximized. So for our data

### Step 6: Visualizing the Results

If we were to plot the projected data in this new 2D space, we would see the observations from different classes are better separated, which is the ultimate goal of LDA. The two axes of this 2D space correspond to the two linear discriminants that maximize the separation between the classes.

import numpy as np  
import matplotlib.pyplot as plt  
  
X = np.array([[2, 3, 4, 5, 6, 7],  
 [3, 4, 5, 6, 7, 8],  
 [6, 5, 4, 3, 2, 1],  
 [7, 6, 5, 4, 3, 2]])  
  
W = np.array([[-0.408, -0.848],  
 [-0.408, -0.237],  
 [-0.408, -0.237],  
 [-0.408, -0.237],  
 [-0.408, -0.237],  
 [-0.408, -0.237]])  
  
Y = np.dot(X, W)  
  
# Visualize the projection  
plt.figure(figsize=(8, 6))  
for i in range(Y.shape[0]):  
 plt.scatter(Y[i, 0], Y[i, 1], label=f'Obs {i+1}', s=100)  
 plt.text(Y[i, 0] + 0.02, Y[i, 1] + 0.02, f'Obs {i+1}', fontsize=12)  
  
plt.title("Projected Data after LDA")  
plt.xlabel('LD1 (First Linear Discriminant)')  
plt.ylabel('LD2 (Second Linear Discriminant)')  
plt.axhline(0, color='gray', lw=1)  
plt.axvline(0, color='gray', lw=1)  
plt.grid(True)  
plt.legend(loc='upper right')  
plt.gcf().patch.set\_facecolor('#f4f4f4')  
plt.gca().set\_facecolor('#f4f4f4')  
plt.show()



### Summary of the Process of Eigenvalue Problem

1. **Eigenvalue Calculation**: We found the eigenvalues and to be the largest, indicating the directions with the most class separability. We did find only two eigenvaleus since total class is 3.
2. **Eigenvector Calculation**: We computed the eigenvectors and corresponding to these eigenvalues. These eigenvectors define the directions in the original feature space along which the class separation is maximized.
3. **Projection**: We projected the original dataset onto the new 2D subspace spanned by the eigenvectors. This resulted in a new dataset in 2D, where the different classes are more separable.

This completes the detailed walkthrough of solving the eigenvalue problem in LDA for our example dataset.

### Final Summary

* **Within-class scatter matrix** quantifies the spread of data points within each class, and we calculated it for each class.
* **Between-class scatter matrix** quantifies the separation between the class means, and we calculated it using the mean of each class and the overall mean.
* Solving the **eigenvalue problem** gives us the directions (eigenvectors) that maximize class separation.

This is how LDA works step by step, using a small dataset as an example.

## Python Code Example

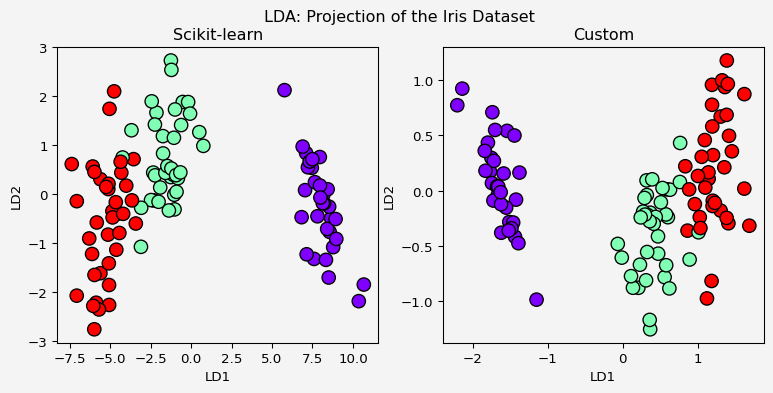
Let’s now revisit the Python code, with an understanding of the math behind LDA. First build our own classifier

class CustomLDA:  
 def \_\_init\_\_(self,n\_components = None) -> None:  
 """  
 Parameters:  
 n\_components: int, optional (default=None)  
 Number of components to keep. If None, all components are kept  
 """  
 self.n\_components = n\_components  
 self.eigenvalues = None  
 self.eigenvectors = None   
 self.mean\_vectors = None   
 self.class\_means = None  
   
 def fit(self, X, y):  
 """  
 Parameters:  
 X: ndarray of shape (n\_samples, n\_features)  
 y: ndarray of shape (n\_samples,)  
 Target labels (must be categorical)  
 """  
 n\_features = X.shape[1]  
 class\_labels = np.unique(y)  
  
 # Step1: Compute the class means mu\_k for each class   
 self.mean\_vectors = []  
 for c in class\_labels:  
 self.mean\_vectors.append(np.mean(X[y==c], axis=0))  
   
 # Step 2: Compute the within-class scatter matrix S\_W   
 S\_W = np.zeros((n\_features, n\_features))  
 for c in class\_labels:  
 class\_scatter = np.cov(X[y==c].T, bias=True) # Covariance matrix for each class  
 S\_W += class\_scatter \* (X[y==c].shape[0])  
  
 # Step 3: Compute the between-class scatter matrix S\_B  
 overall\_mean = np.mean(X, axis=0)  
 S\_B = np.zeros((n\_features, n\_features))  
  
  
 for i,mean\_vector in enumerate(self.mean\_vectors):  
 n = X[y == class\_labels[i]].shape[0]  
 mean\_differences = (mean\_vector -overall\_mean).reshape(n\_features,1)  
 S\_B += n\*(mean\_differences).dot(mean\_differences.T)  
   
 # Step 4: Solve the Eigenvalue problem   
 eigvalues, eigvectors = np.linalg.eig(np.linalg.pinv(S\_W).dot(S\_B))  
  
 # Step 5: Sort the Eigenvalues and corresponding eigenvectors   
 eigvalues\_sort\_idx = np.argsort(np.abs(eigvalues))[::-1]  
 self.eigenvalues = eigvalues[eigvalues\_sort\_idx]  
 self.eigenvectors = eigvectors[:,eigvalues\_sort\_idx]  
  
 # Step 6: Keep only the top n\_components  
 if self.n\_components:  
 self.eigenvectors = self.eigenvectors[:,:self.n\_components]  
   
 self.class\_means = np.dot(self.mean\_vectors, self.eigenvectors)  
   
 def transform(self,X):  
 """  
 Project the data onto the LDA components   
  
 Parameters:  
 X: ndarray of shape (n\_samples, n\_features)  
  
 Returns:  
 X\_transformed: ndarray of shape (n\_samples, n\_features)  
 """  
 return np.dot(X,self.eigenvectors)  
   
 def fit\_transform(self, X, y):  
 """  
 Fit the LDA model and transform the data.  
   
 Parameters:  
 X : ndarray of shape (n\_samples, n\_features)  
 Training data.  
 y : ndarray of shape (n\_samples,)  
 Target labels (must be categorical).  
   
 Returns:  
 X\_transformed : ndarray of shape (n\_samples, n\_components)  
 Transformed data after fitting.  
 """  
 self.fit(X, y)  
 return self.transform(X)  
   
 def predict(self, X):  
 """  
 Predict the class labels for new data points.  
  
 Parameters:  
 X : ndarray of shape (n\_samples, n\_features)  
 New data to classify.  
  
 Returns:  
 Predictions: ndarray of shape (n\_samples,)  
 Predicted class labels  
 """  
 X\_projected = self.transform(X)  
  
 predictions = []  
 for x in X\_projected:  
 distances = np.linalg.norm(x-self.class\_means, axis=1)  
 predictions.append(np.argmin(distances))  
   
 return np.array(predictions)  
  
   
 def explained\_variance\_ratio(self):  
 """  
 Return the percentage of variance explained by each of the selected components  
  
 Returns:  
 explained\_variance: ndarray of shape (n\_components,)  
 Percentage of variance explained by each selected components  
 """  
 total = np.sum(self.eigenvalues)  
  
 return [(i/total) for i in self.eigenvalues[:self.n\_components]]

Next we apply both the custom classifier and the classifier from the scikit-learn library.

import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis as LDA  
from sklearn.datasets import load\_iris  
from sklearn.metrics import accuracy\_score  
  
# Load the dataset  
iris = load\_iris()  
X = iris.data  
y = iris.target  
  
# Standardize the dataset (optional but often improves performance)  
scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
  
# Split into training and test sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)  
  
# Apply LDA from the scikit-learn library  
lda1 = LDA(n\_components=2) # Reduce to 2 dimensions  
X\_train\_lda1 = lda1.fit\_transform(X\_train, y\_train)  
X\_test\_lda1 = lda1.transform(X\_test)  
  
# Apply LDA from the custom built classifier  
lda2 = CustomLDA(n\_components=2) # Reduce to 2 dimensions  
X\_train\_lda2 = lda2.fit\_transform(X\_train, y\_train)  
X\_test\_lda2 = lda2.transform(X\_test)  
  
# Visualize the LDA-transformed data  
fig, axes = plt.subplots(1,2, figsize=(9.5,4))  
  
axes[0].scatter(X\_train\_lda1[:, 0], X\_train\_lda1[:, 1], c=y\_train, cmap='rainbow', edgecolor='k', s=100)  
axes[0].set\_xlabel('LD1')  
axes[0].set\_ylabel('LD2')  
axes[0].set\_title('Scikit-learn')  
axes[1].scatter(X\_train\_lda2[:, 0], X\_train\_lda2[:, 1], c=y\_train, cmap='rainbow', edgecolor='k', s=100)  
axes[1].set\_xlabel('LD1')  
axes[1].set\_ylabel('LD2')  
axes[1].set\_title('Custom')  
for ax in axes:  
 ax.set\_facecolor('#f4f4f4')  
plt.gcf().patch.set\_facecolor('#f4f4f4')  
fig.suptitle('LDA: Projection of the Iris Dataset')  
plt.show()

/opt/hostedtoolcache/Python/3.10.15/x64/lib/python3.10/site-packages/matplotlib/cbook.py:1762: ComplexWarning: Casting complex values to real discards the imaginary part  
 return math.isfinite(val)  
/opt/hostedtoolcache/Python/3.10.15/x64/lib/python3.10/site-packages/matplotlib/collections.py:197: ComplexWarning: Casting complex values to real discards the imaginary part  
 offsets = np.asanyarray(offsets, float)



Next, apply LDA as a classifiers for the actual classification

lda\_classifier1 = LDA()  
lda\_classifier1.fit(X\_train, y\_train)  
y\_pred1 = lda\_classifier1.predict(X\_test)  
  
lda\_classifier2 = CustomLDA()  
lda\_classifier2.fit(X\_train, y\_train)  
y\_pred2 = lda\_classifier2.predict(X\_test)  
  
  
# Check accuracy  
accuracy1 = accuracy\_score(y\_test, y\_pred1)  
accuracy2 = accuracy\_score(y\_test, y\_pred2)  
print(f'sklearn LDA Classifier Accuracy: {accuracy1 \* 100:.2f}% and \ncustom LDA Classifier Accuracy: {accuracy2 \* 100:.2f}%')

sklearn LDA Classifier Accuracy: 100.00% and   
custom LDA Classifier Accuracy: 95.56%

Not too bad, huh! Let’s see the confusion matrix for our custom classifier

from sklearn.metrics import confusion\_matrix  
  
conf\_mat = confusion\_matrix(y\_test, y\_pred2)  
  
print(pd.DataFrame(  
 conf\_mat,   
 columns=['Pred: Setosa','Pred: Virginica', 'Pred: Versicolor'],  
 index=['Actual: Setosa','Actual: Virginica', 'Actual: Versicolor']  
))

Pred: Setosa Pred: Virginica Pred: Versicolor  
Actual: Setosa 19 0 0  
Actual: Virginica 0 11 2  
Actual: Versicolor 0 0 13

## Conclusion

Linear Discriminant Analysis (LDA) is a powerful technique for dimensionality reduction and classification. Its goal is to find directions (linear combinations of the original features) that best separate the classes by maximizing between-class variance while minimizing within-class variance.

### Disclaimer

For the mathematical explanation, I used generative AI to produce the matrices and vectors and their manipulations. So it won’t be surprising if a calculation mistake is found. The custom python class was created by the help of ChatGPT4

## References

* Fisher, R.A. (1936). “The Use of Multiple Measurements in Taxonomic Problems.” *Annals of Eugenics*, 7(2), 179–188.
* Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.
* Strang, G. (2016). *Introduction to Linear Algebra* (5th ed.). Wellesley-Cambridge Press.
* Lay, D. C. (2011). *Linear Algebra and Its Applications* (4th ed.). Pearson.

**Share on**

**You may also like**