Linear Discriminant Analysis - A Comprehensive Guide

Intuition

The idea behind Linear Discriminant Analysis (LDA) is to dimensionally reduce the input feature matrix while preserving as much *class-discriminatory* information as possible. LDA tries to express the dependent variable as a linear combination of other features. Unlike ANOVA (Analysis of Variance) which analyzes differences among group means, LDA uses continuous independent features and a categorical output variable.

LDA attempts to distance the k classes as much as possible through a linear combination of its features by maximizing the between-class scatter while minimizing within-class scatter. LDA makes an assumption that for each of the k classes, the class conditional densities $p(x \mid Y = k) = f_k(x)$ are multivariate Gaussian distributions.

Why does LDA make Gaussian assumptions?

One, it makes it mathematically easy to derive a closed-form solution for LDA. Two, through Maximum Likelihood Estimation (MLE), LDA's parameters can be estimated under Gaussian assumptions.

LDA assumes all classes have the same covariance matrix.
Simplification - Assuming equal covariance matrices for all k classes simplifies the LDA model. The assumption leads to **linear decision boundaries** between the classes aligning with LDA's linear combinations of features objective.

How is LDA different from PCA?

Principal Component Analysis (PCA) is different in its motives - it tries to maximize variance in the transformed features by finding directions/principal components that captures the highest overall variance. LDA works on maximizing class separability - it is class conscious and is a supervised method, whereas PCA is unsupervised. PCA is majorly used for dimensionality reduction whereas LDA is effective for classification tasks.

What does LDA do?

LDA tries to maximize the 'Between class distance' and minimize the 'intra class distance' for each class. Doing so will transform the original input space to a transformed lower dimensional space.

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)\right)$$

where

- $f_k(x)$ is the probability density function for class k
- x is the input vector
- p is the number of features

- μ_k is the mean vector for class k
- Σ_k is the covariance matrix for class k
- $|\Sigma_k|$ is the determinant of Σ_k
- Σ_k^{-1} is the inverse of Σ_k

Σ_1 = Σ_2 = Σ is an assumption. Analyzing the decision boundary between the two classes where the discriminant functions are equal. The total between-class variance is calculated based on the below equation.

$$(\mathbf{m}_i - \mathbf{m})^2 = \mathbf{W}^T \mathbf{S}_{B_i} \mathbf{W}$$

The below equation represents the discriminant's function for class k. The formula encapsulates the essence of LDA's classification mechanism, combining priors of the equation, covariance information, and distance of a data point from the class mean. By evaluating this function for each class and selecting the class with the highest score, LDA makes the classification decision. This below equation is after logarithm of $f_k(\mathbf{x})$ since $\log$ is montonic and preserves the order of the original equation.

$$\delta_k(\mathbf{x}) = \log(\pi_k) - \frac{1}{2}\log(|\Sigma_k|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)$$

$$\delta_1(\mathbf{x}) = \delta_2(\mathbf{x})$$

$$\log(\pi_1) - \frac{1}{2}\log(|\Sigma|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_1) = \log(\pi_2) - \frac{1}{2}\log(|\Sigma|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_2)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_2)$$

$$\log\left(\frac{\pi_1}{\pi_2}\right) - \frac{1}{2}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)^T \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) + \mathbf{x}^T \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) = 0$$

This result is linear in x and discriminates class 1 from class 2. $\mathbf{x}^T \Sigma \mathbf{x}$ cancels out with the equal covariance assumption making the discriminant linear.

Constructing the Lower Dimensional Space

After calculating between-class variance, S_B and within-class variance, S_W, the lower dimensional matrix post transformation, W can be calculated based on *Fisher's criterion* (maximizing $\frac{S_B}{S_W}$).

$$\arg\max_{\mathbf{W}} \frac{\mathbf{W}^T \mathbf{S}_B \mathbf{W}}{\mathbf{W}^T \mathbf{S}_W \mathbf{W}}$$

$$\mathbf{S}_W \mathbf{W} = \lambda \mathbf{S}_B \mathbf{W}$$

$$(\mathbf{S}_W^{-1} \mathbf{S}_B - \lambda \mathbf{I}) \mathbf{W} = \mathbf{0}$$

$$\mathbf{W} = \mathbf{S}_W^{-1} \mathbf{S}_B$$

Solution to the problem can be calculated from the eigenvalues $\lambda = \{\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_M\}$ and eigenvectors $v = \{v_1, v_2, v_3, \ldots, v_M\}$ of the transformed matrix W. The eigenvectors show the directions of the transformed space and the eigenvalues shows the scaling factor - the discriminative power of the corresponding eigenvectors. Larger the eigenvalue, the more variance it accounts for in the direction of its corresponding eigenvector.

Building the LDA Model

Let's build an LDA classification model for binary classification.

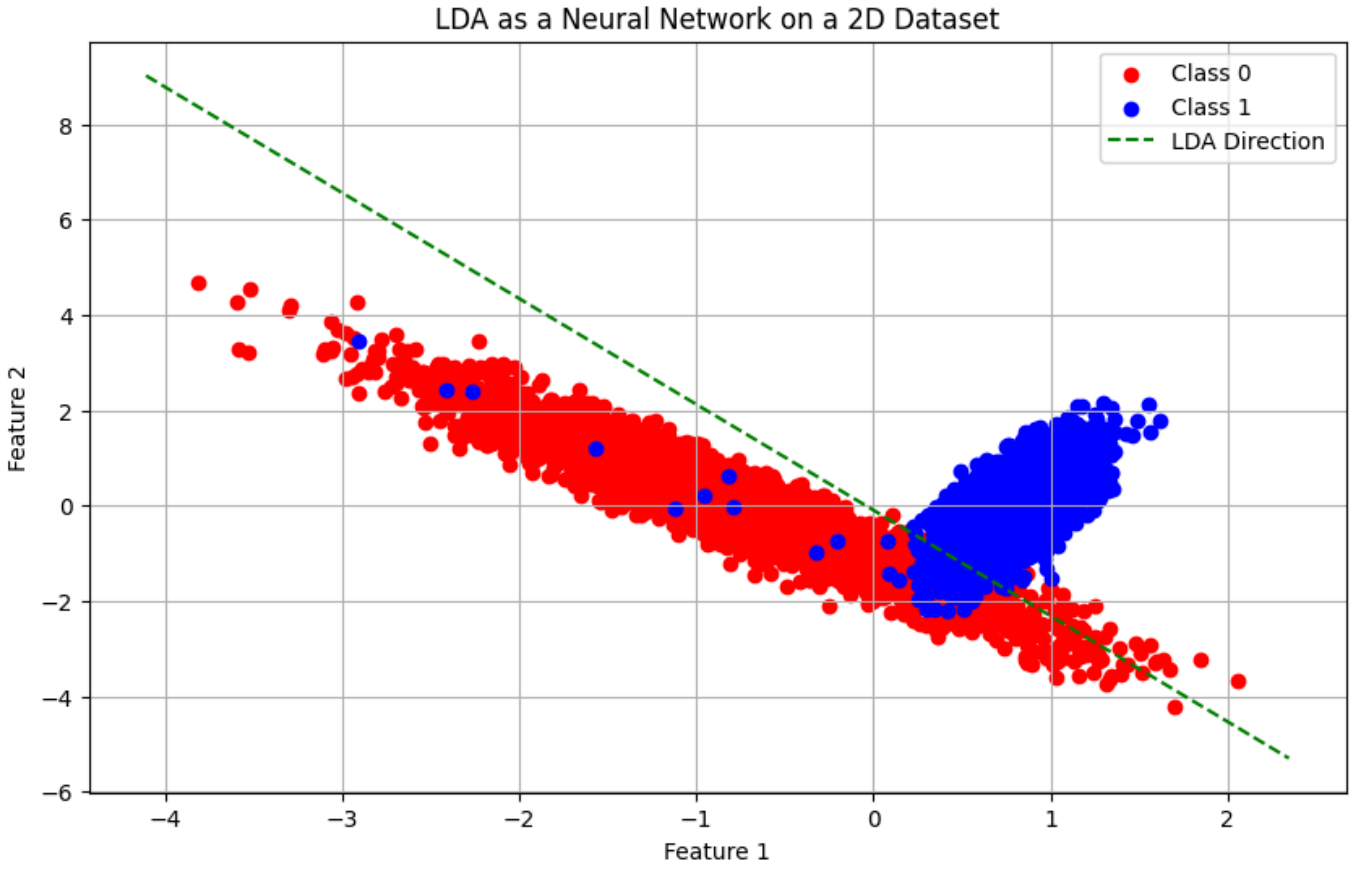
```
import numpy as np
from sklearn.preprocessing import StandardScaler

class LinearDiscriminant:
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.eigenvectors = None
        self.scaler = StandardScaler()

    def scale_features(self, X):
        return self.scaler.fit_transform(X)

    def fit(self, X, y):
        X = self.scale_features(X)  # Scale features before fitting
        M = X.shape[1]  # Number of Features
        classes = np.unique(y)  # Number of Classes
        mu_overall = np.mean(X, axis=0)  # Overall mean
        S_W = np.zeros((M, M))
        S_B = np.zeros((M, M))

        for k in classes:
            X_k = X[y == k]
            mu_k = np.mean(X_k, axis=0)
            S_W += np.dot((X_k - mu_k).T, (X_k - mu_k))  # Within class scatter

            n_k = X_k.shape[0]
            mu_delta = (mu_k - mu_overall).reshape(M, 1)
            S_B += n_k * np.dot(mu_delta, mu_delta.T)  # Between class scatter


        A = np.linalg.pinv(S_W).dot(S_B)
        eigenvalues, eigenvectors = np.linalg.eig(A)
        self.eigenvectors = eigenvectors[:, np.argsort(eigenvalues)[::-1]]  # Sort
eigenvectors by eigenvalues (descending order)
        self.eigenvalues = eigenvalues[np.argsort(eigenvalues)[::-1]]

    def transform(self, X, dimensions):
        X = self.scale_features(X)
        if dimensions is not None and dimensions < X.shape[1]:
            X_transformed = np.dot(X, self.eigenvectors[:, :dimensions])
        else:
            X_transformed = np.dot(X, self.eigenvectors)

        explained_variance = np.var(X_transformed, axis=0) / np.var(X, axis=0).sum()

        return X_transformed, explained_variance

        ```
```
```

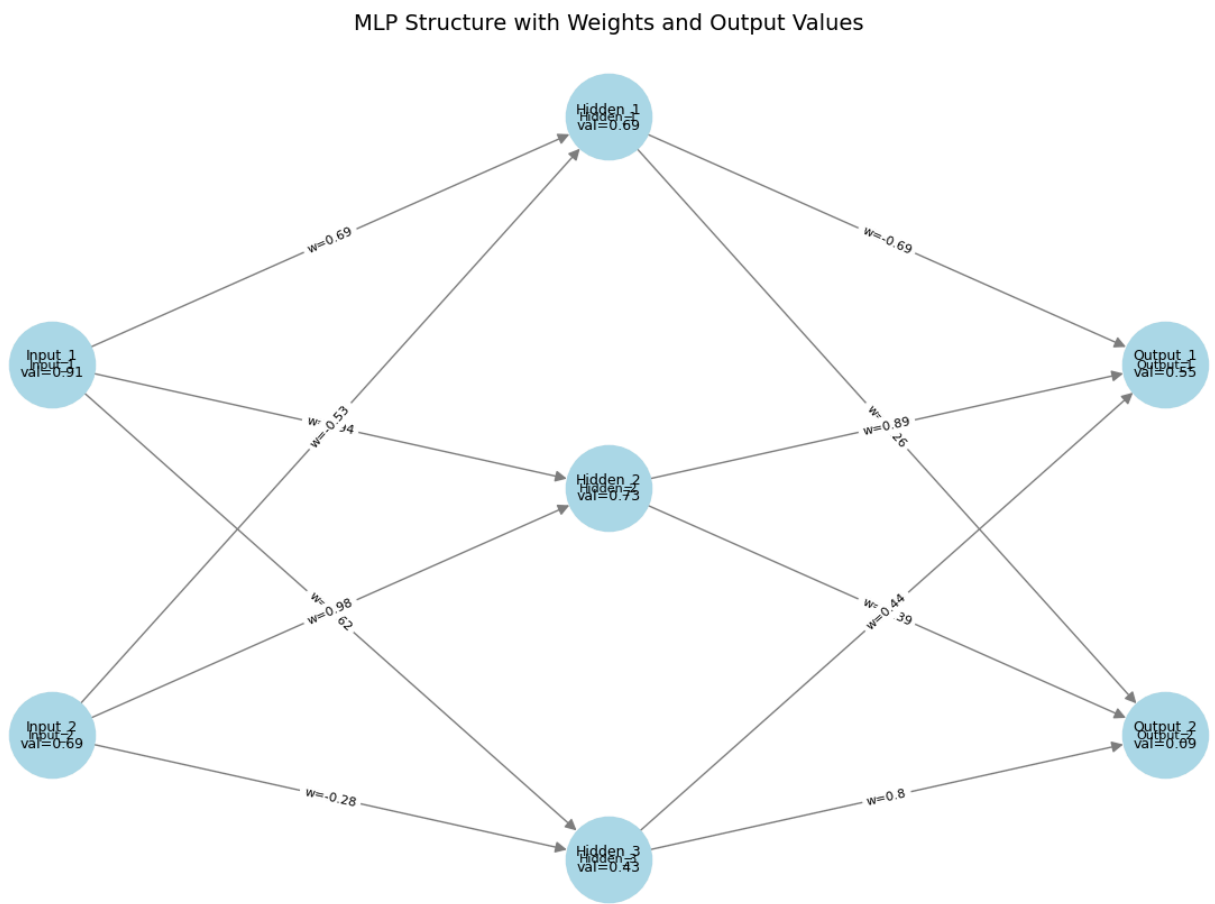
Formulating LDA as a Neural Network

The LDA classification model can be reformulated as a neural network with a single hidden layer. The input layer corresponds to the input data feature space and the output is a Softmax layer. The hidden layer is a single dense layer with a linear activation function. This layer performs the linear transformation defined by LDA. Weights of this layer correspond to the LDA directions (eigenvectors of the transformed matrix). Output Softmax layer converts the linear projection into class probabilities.

Training objective: Minimize cross-entropy loss, which is the same as maximizing class separability. Unlike LDA which computes a closed-form solution, the neural network can be trained using gradient-based optimizers.

```python
class LDANeuralNet(nn.Module):
    def __init__(self):
        super(LDANeuralNet, self).__init__()
        self.lda_layer = nn.Linear(2, 1)
        self.softmax = nn.Softmax(dim = 1)

    def forward(self, x):
        x = self.lda_layer(x)
        x = self.softmax(torch.cat([-x, x], dim=1))  # Two-class softmax
        return x

model = LDANeuralNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Train the model
num_epochs = 500
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)

    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        with torch.no_grad():
            test_outputs = model(X_test_tensor)
            test_loss = criterion(test_outputs, y_test_tensor)
            print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}, Test Loss:
{test_loss.item():.4f}')
```


LDA as a Neural Network on a 2D Dataset
Class 0
Class 1
LDA Direction
Feature 1
Feature 2


MLP Structure with Weights and Output Values
Input_1
val=0.91
Input_2
val=0.69
Hidden_1
val=0.69
Hidden_2
val=0.73
Hidden_3
val=0.43
Output_1
val=0.55
Output_2
val=0.69
w=0.69
w=-0.69
w=0.89
w=-0.28
w=0.8

References

[1] Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd ed.). Springer. Chapter 4.3 discusses LDA in detail.
[2] Tharwat, A., Gaber, T., Ibrahim, A., & Hassanien, A. E. (20xx). Linear discriminant analysis: A detailed tutorial. AI Communications, 00, 1-22. DOI: 10.3233/AIC-170729