110705067 洪薇欣

Methods:

A. Dimensionality Reduction:

1. FLD/LDA (Fisher's Linear Discriminant/ Linear Discriminant Analysis):

- LDA **maximize the separability** between different classes.
- **Maximizing Between-Class Scatter and Minimizing Within-Class Scatter.**
- LDA is a **supervised method** that takes the class labels into account .

Steps:

- **Compute the Mean Vectors**: For each class, compute the **mean of the data points**.
- **Compute the Scatter Matrices**:

  Within-class scatter matrix: Measures how much the data points within each class deviate **from the class mean**.

$$S_W = \sum_{i=1}^{k} \sum_{x \in C_i} (x - \mu_i)(x - \mu_i)^T$$

  where C_i is the set of data points in class i, and μ_i is the mean of class i.

  Between-class scatter matrix: Measures how much the class means deviate from the **overall mean of the data**.

$$S_B = \sum_{i=1}^{k} N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

  where μ_i is the mean of class i, μ is the overall mean of all classes, and N_i is the number of data points in class i.

- **Compute the Linear Discriminants**: Solve the generalized eigenvalue problem $S_W^{-1} S_B$ to find the eigenvectors (linear discriminants). These **eigenvectors represent the directions that maximize class separability**.
- **Choose the Top Discriminants**: Sort the eigenvectors by their corresponding eigenvalues. The **eigenvectors with the highest eigenvalues** will give the directions that **maximize the separability**.
- **Project the Data**: Once the top discriminants are chosen, project the data onto these new axes (lower-dimensional space).

2. PCA (Principal Component Analysis):

- It transforms a dataset with many features into a **smaller set of features** (principal components) while **preserving as much of the original data's variability as possible**.
- **Principal Components are the directions in the data where the variance is maximized (finding the directions where the data is most spread out).**
- PCA is an **unsupervised** method which does not take class labels into account.

Steps:

- **Standardize the Data:** If features are on different scales, PCA is sensitive to this.
- **Compute the Covariance Matrix:** Find how the features vary with respect to each other.
- **Find the Eigenvalues and Eigenvectors: Eigenvectors represent the directions of maximum variance**

(principal components), and eigenvalues show how much variance is captured in each direction.

- **Sort Eigenvalues:** Sort the eigenvectors by eigenvalues in descending order to choose **the top components.**
- **Project the Data:** Use the selected eigenvectors to form a projection matrix. Multiply the data by this matrix to get the new, reduced set of features.

B. Classifiers:

1. Naive Bayes Log-Likelihood:

a probabilistic machine learning model based on Bayes' Theorem, with a naive assumption that **features are conditionally independent** given the class.

2. Perceptron Classifier:

A perceptron takes multiple input signals, assigns weights to them, sums them up (along with a bias term), and then applies a step function to produce a binary output.

The most fundamental limitation is that perceptron can only correctly classify **linearly separable data**.

Experiments and Results:

A. Datasets:

Breast Cancer Coimbra: 116 samples, 9 features, **2 classes (small & imbalanced)**

Ionosphere: 351 samples, 34 features, **2 classes (high dimensional)**

Iris: 150 samples, 4 features, **3 classes (3 balanced classes)**

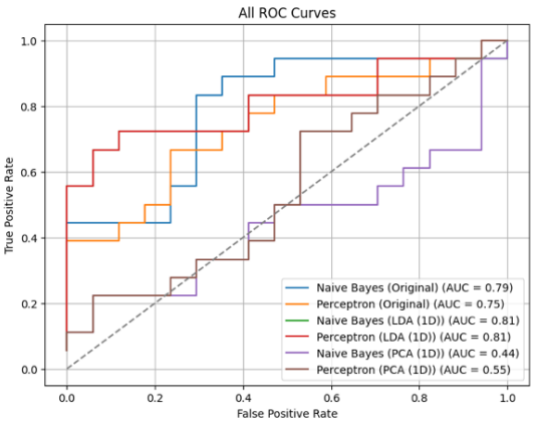
Wine: 178 samples, 13 features, **3 classes (more complex)**

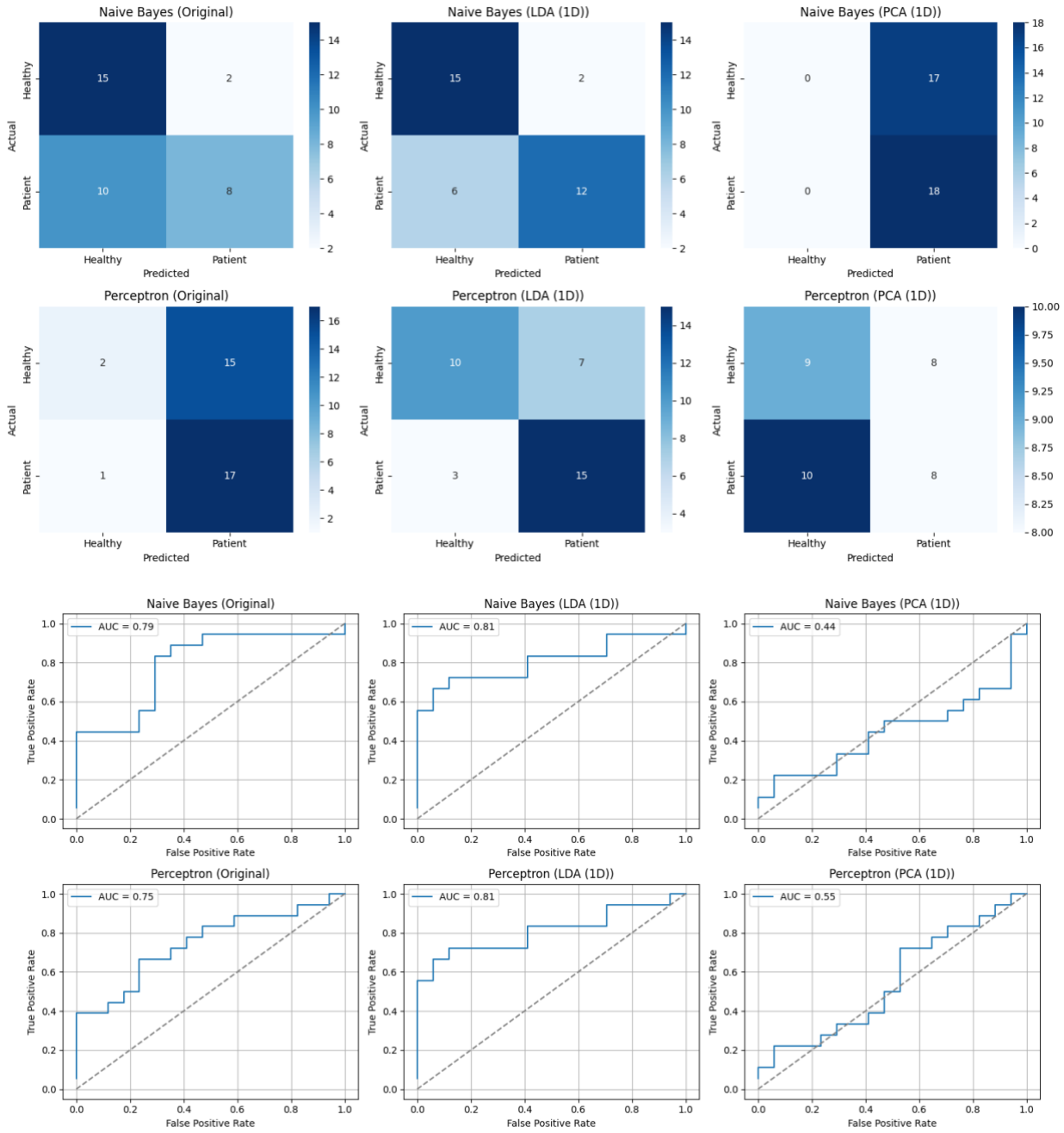
B. Dataset Splitting: randomly split datasets into training set and validation set with ratio **7:3**.

C. Results:

1. Breast Cancer Coimbra – LDA(to 1D)/PCA(to 1D) vs. Naïve Bayes/Perceptron:

```
=== Evaluation Summary Table ===
                    Label  Accuracy  Precision    Recall  F1 Score       AUC
Naive Bayes (Original)  0.657143   0.800000 0.444444  0.571429 0.790850
 Perceptron (Original)  0.542857   0.531250 0.944444  0.680000 0.745098
Naive Bayes (LDA (1D))  0.771429   0.857143 0.666667  0.750000 0.807190
 Perceptron (LDA (1D))  0.714286   0.681818 0.833333  0.750000 0.810458
Naive Bayes (PCA (1D))  0.514286   0.514286 1.000000  0.679245 0.444444
 Perceptron (PCA (1D))  0.485714   0.500000 0.444444  0.470588 0.549020
```





■ **Naive Bayes:**

LDA improved accuracy from 0.6571 to 0.7714 and AUC from 0.7908 to 0.8072, indicating better class separation after dimensionality reduction.

PCA significantly worsened performance (AUC dropped to 0.4444, lower than random).

■ **Perceptron:**

LDA improved performance: accuracy increased from 0.5429 to 0.7143, and AUC rose to 0.8105.

PCA reduced performance (accuracy dropped to 0.4857, AUC to 0.5490), reinforcing the idea that PCA's 1D projection was not useful for classification in this dataset.

■ **Overall:**

LDA is a supervised method that uses class labels to maximize class separability, making it especially effective for classification tasks. Therefore, **LDA benefited both classifiers, especially the Perceptron**, due to its **reliance on linear separability**. **Naive Bayes assumes feature independence**, and since **LDA produces linearly combined features**, this assumption is slightly violated, which might lead to less improvement comparing to perceptron. PCA in 1D may **eliminate**

vital class-discriminative information. PCA is unsupervised and **preserves variance without considering class boundaries**—leading to poorer performance when reduced to just one dimension. PCA hurt both models in 1D, which is consistent with the idea that **maximizing variance doesn't necessarily align with maximizing discriminative power** between classes. The results are as expected, LDA is expected to perform better than PCA in classification tasks, especially in low-dimensional projections.

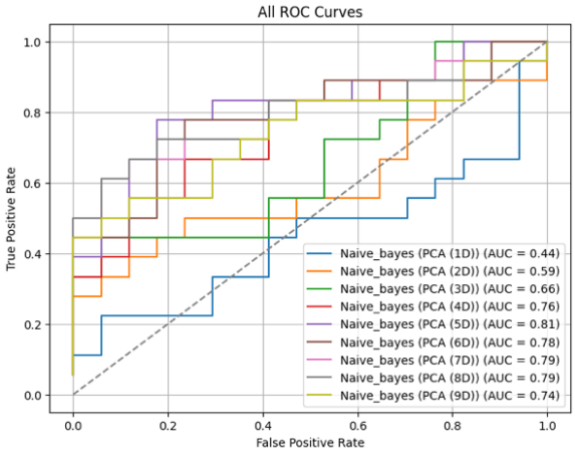
2. Breast Cancer Coimbra – PCA (1D~9D) vs. Naïve Bayes/Perceptron:

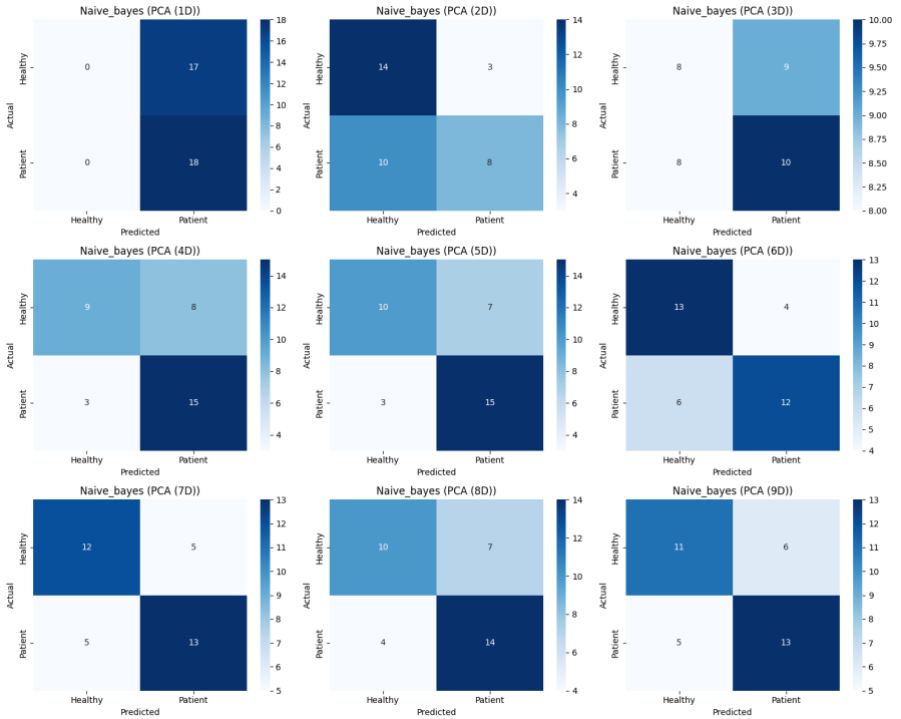
The previous experiments have worse performance using PCA is likely to be due to the value I set for num_components is too low. **num_components is number of principal axes (directions of highest variance in data) to keep. Lower num_components in PCA retain less total variance.** This reduces data size for faster computation and lower storage needs, but it may also discard important patterns and lead to information loss. And this might lead to the machine learning models perform worse.

So, the following experiments examine the results of using different values for num_components. Notice that the value of num_components should be any number from **1 up to min(n_samples, n_features)**.

- **Naïve Bayes:**

```
=== Evaluation Summary Table ===
                    Label  Accuracy  Precision    Recall  F1 Score       AUC
Naive_bayes (PCA (1D))  0.514286   0.514286 1.000000  0.679245 0.444444
Naive_bayes (PCA (2D))  0.628571   0.727273 0.444444  0.551724 0.591503
Naive_bayes (PCA (3D))  0.514286   0.526316 0.555556  0.540541 0.660131
Naive_bayes (PCA (4D))  0.685714   0.652174 0.833333  0.731707 0.758170
Naive_bayes (PCA (5D))  0.714286   0.681818 0.833333  0.750000 0.810458
Naive_bayes (PCA (6D))  0.714286   0.750000 0.666667  0.705882 0.784314
Naive_bayes (PCA (7D))  0.714286   0.722222 0.722222  0.722222 0.787582
Naive_bayes (PCA (8D))  0.685714   0.666667 0.777778  0.717949 0.790850
Naive_bayes (PCA (9D))  0.685714   0.684211 0.722222  0.702703 0.741830
```





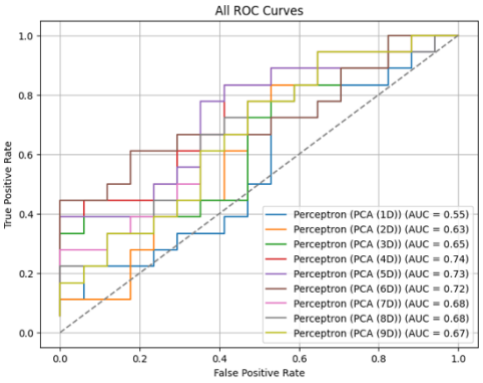
Extremely poor AUC (0.4444) and low accuracy (0.5143) for **1D**. **Poor performance due to high info loss**. Confirms that 1D is insufficient to capture class-discriminative information in this dataset.

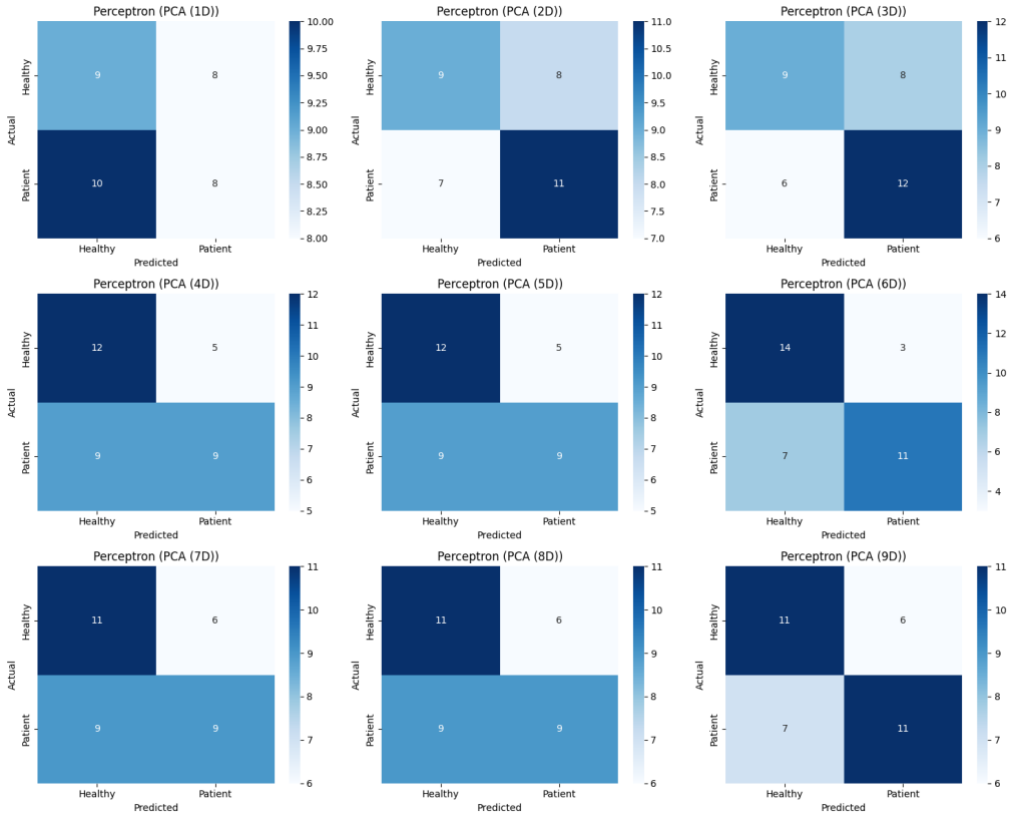
Significant gains in accuracy, AUC, and F1 Score from 2D to 5D. Peak AUC and accuracy occur at 5 components (AUC = 0.8105, Accuracy = 0.7143). This suggests that around 5 dimensions r**etain most of the discriminative variance for Naive Bayes**.

Performance plateaus or slightly drops after 6–9 components. This is expected because **Naive Bayes assumes feature independence, and adding more components may introduce noise or correlated features** not aligned with its assumptions.

■ **Perceptron:**

```
=== Evaluation Summary Table ===
                    Label  Accuracy  Precision    Recall  F1 Score       AUC
Perceptron (PCA (1D))  0.485714   0.500000 0.444444  0.470588 0.549020
Perceptron (PCA (2D))  0.571429   0.578947 0.611111  0.594595 0.627451
Perceptron (PCA (3D))  0.600000   0.600000 0.666667  0.631579 0.653595
Perceptron (PCA (4D))  0.600000   0.642857 0.500000  0.562500 0.738562
Perceptron (PCA (5D))  0.600000   0.642857 0.500000  0.562500 0.732026
Perceptron (PCA (6D))  0.714286   0.785714 0.611111  0.687500 0.722222
Perceptron (PCA (7D))  0.571429   0.600000 0.500000  0.545455 0.683007
Perceptron (PCA (8D))  0.571429   0.600000 0.500000  0.545455 0.676471
Perceptron (PCA (9D))  0.628571   0.647059 0.611111  0.628571 0.666667
```





Gradual improvement in metrics as components increase. In **1D**, accuracy is poor (0.4857), similar to random guessing, suggesting that class-separating information is **too compressed**. By 3D, performance is becoming acceptable, indicating that 3 dimensions start capturing enough class-discriminative variance.

Peak performance at 6 components. **6D appears to balance variance retention and generalization best for Perceptron**, which is a linear classifier sensitive to feature space orientation.

Performance drops slightly or stagnates for 7D-9D. AUC drops compared to 6D (even at 9D, AUC = 0.6667 vs. 0.7222 at 6D). **Indicates potential overfitting or noise inclusion** from less informative components.

- **Overall:**

Naive Bayes performs better overall with PCA in terms of F1 score, AUC, and recall. Perceptron has slightly better precision at its peak, but its overall scores plateau lower than Naive Bayes. Naive Bayes improves steadily up to 5–6 components and then holds performance quite well up to 9D. Perceptron also improves up to 6D, but starts to drop off after that, with F1 and accuracy falling or stagnating.

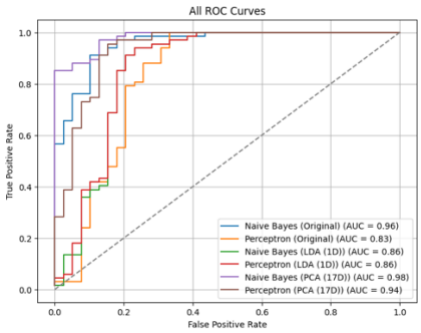
Naïve Bayes benefits more from PCA because PCA brings the data distribution closer to Naive Bayes' assumptions (independence and Gaussianity). PCA removes correlation between features, but doesn't guarantee class separation. **PCA doesn't necessarily making the data more linearly separable, which is what the Perceptron needs.**

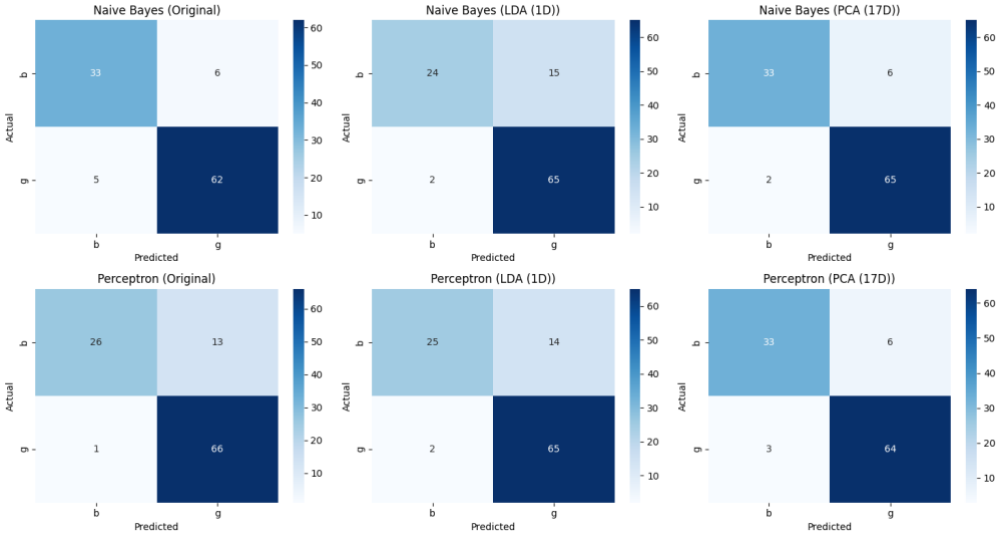
- **Conclusion:**

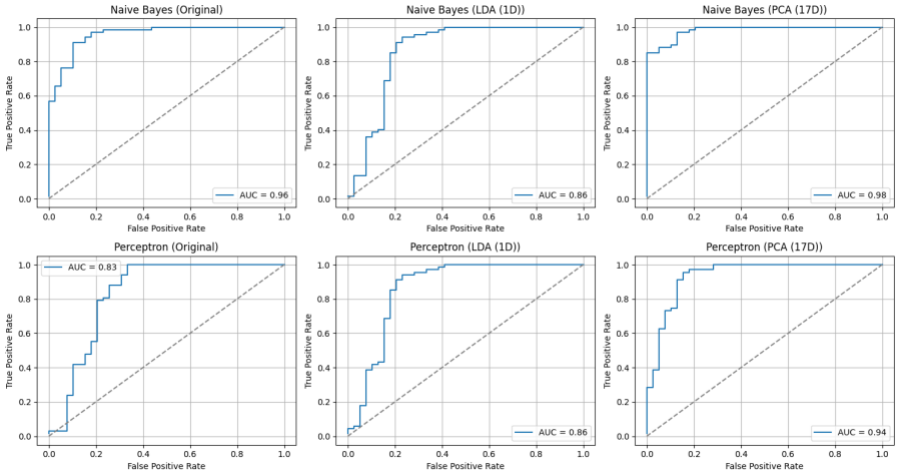
It looks like PCA performs best at mid-range dimensions because it balances data simplification with information retention. Too few dimensions lose critical class info; too many reintroduce noise and redundancy.

3. Ionosphere – LDA(to 1D)/PCA(to 17D) vs. Naïve Bayes/Perceptron :

```
=== Evaluation Summary Table ===
                  Label  Accuracy  Precision    Recall  F1 Score       AUC
 Naive Bayes (Original)  0.896226   0.911765 0.925373  0.918519 0.957137
  Perceptron (Original)  0.867925   0.835443 0.985075  0.904110 0.833142
 Naive Bayes (LDA (1D))  0.839623   0.812500 0.970149  0.884354 0.861462
  Perceptron (LDA (1D))  0.849057   0.822785 0.970149  0.890411 0.863758
Naive Bayes (PCA (17D))  0.924528   0.915493 0.970149  0.942029 0.981630
 Perceptron (PCA (17D))  0.915094   0.914286 0.955224  0.934307 0.936471
```







■ **LDA:**

The performance of **Naive Bayes with LDA is slightly worse** than the original features. While the recall is still high (0.9701), the precision drops a little, and the accuracy is lower than before (0.8396). The AUC also drops to 0.8615, indicating that the class separation is not as strong after applying LDA.

The **Perceptron model's with LDA performance is similarly reduced compared to the original**. While recall remains high, precision increases slightly (0.8228), and accuracy and AUC are also slightly worse compared to the original. However, **it performs a little better than Naive Bayes after LDA.**

■ **PCA:**

For Naive Bayes, PCA clearly improves performance over both the original data and LDA. The accuracy improves significantly to 0.9245, precision and F1 Score are also higher, and the AUC jumps to 0.9816.

For Perceptron, PCA also improves the performance significantly compared to the original dataset. The accuracy increases to 0.9151, precision improves, and AUC increases to 0.9365.

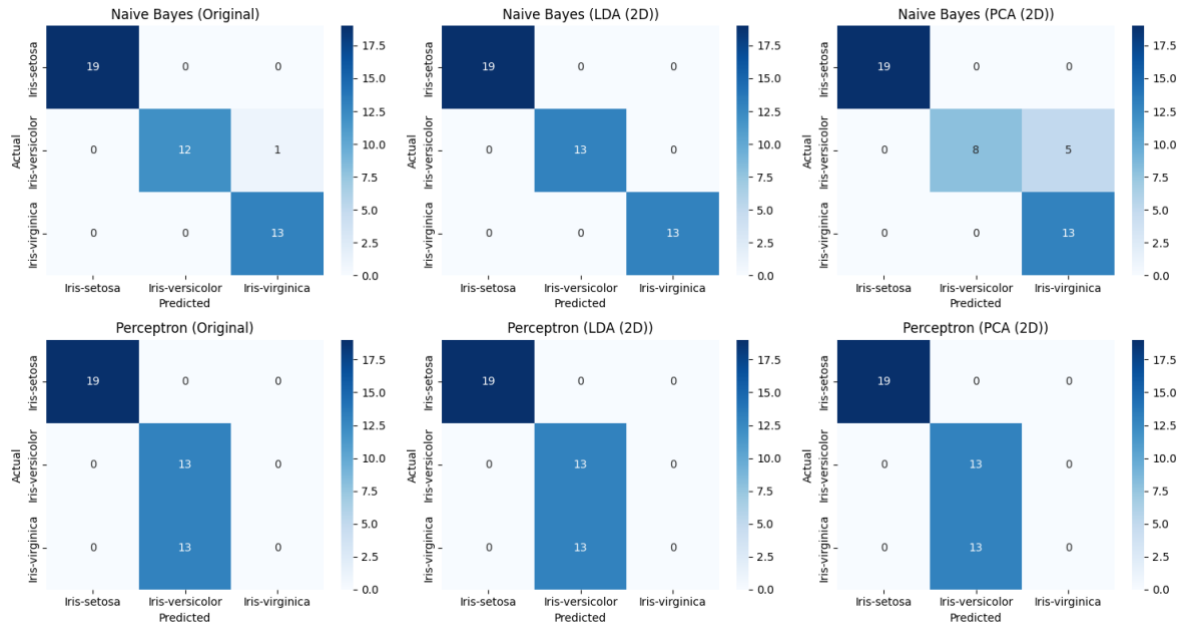
■ **Overall:**

PCA helps Naive Bayes more. Naive Bayes assumes **feature independence** and can benefit from PCA, which **removes correlations between features (PCA** creates uncorrelated components) and reduces noise, improving generalization for probabilistic models like Naive Bayes.

LDA helps Perceptron more, because **Perceptron is a linear classifier, and LDA is a linear method** that finds a projection maximizing class separability. The alignment between LDA's class-separating direction and Perceptron's linear boundary helps improve learning.

4. Iris – LDA(to 2D)/PCA(to 2D) vs. Naïve Bayes/Perceptron:

```
=== Evaluation Summary Table ===
                 Label  Accuracy  Precision   Recall  F1 Score
Naive Bayes (Original)  0.977778   0.976190 0.974359  0.974321
 Perceptron (Original)  0.711111   0.500000 0.666667  0.555556
Naive Bayes (LDA (2D))  1.000000   1.000000 1.000000  1.000000
 Perceptron (LDA (2D))  0.711111   0.500000 0.666667  0.555556
Naive Bayes (PCA (2D))  0.888889   0.907407 0.871795  0.866871
 Perceptron (PCA (2D))  0.711111   0.500000 0.666667  0.555556
```



■ **Naive Bayes:**

Original data already performs excellently (97.8% accuracy).

LDA improves this to perfect classification—all classes are 100% correctly identified in 2D.

PCA results in a performance drop (accuracy 88.9%, F1 0.867), likely because variance retained by PCA does not align perfectly with class separability.

■ **Perceptron:**

Perceptron performs poorly across the board (71.1% accuracy), and does not benefit from dimensionality reduction (neither PCA nor LDA improves or worsens performance).

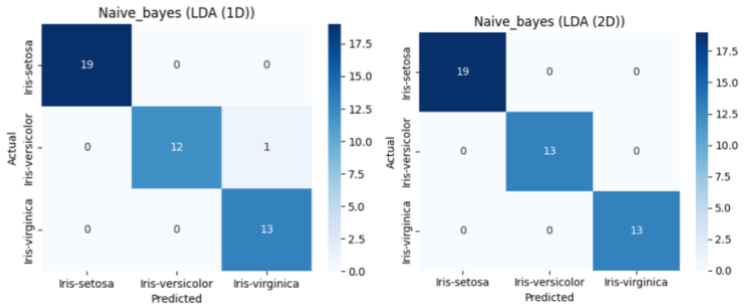
This is possibly due to: **Lack of convergence during training, Inherent limitations in fitting non-linearly separable data, Inadequate data scaling or preprocessing**.

■ **Overall:**

LDA has better performance on Naïve Bayes than PDA in this case, as expected. Perceptron fails to leverage any benefit from LDA or PCA, likely due to **underfitting or an inability to handle the class boundaries in this form**.

5. Iris – LDA(1D- 2D) vs. Naïve Bayes:

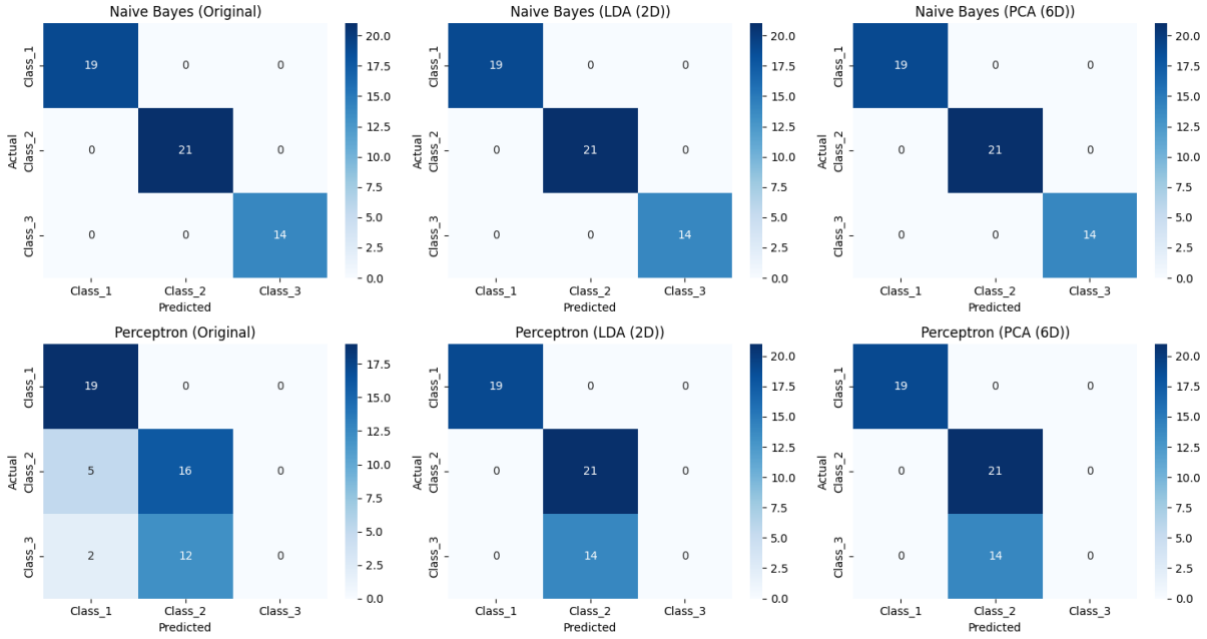
LDA can produce at most (C - 1) linear discriminants for C classes. Since Iris has 3 classes, LDA supports a **maximum of 2 components**. Using only **1 component (1D) compresses the data too much, potentially overlapping class distributions slightly**. In 2D, the model **captures both discriminative axes, leading to perfect separation. Naive Bayes assumes feature independence and normally distributed data. LDA projects data to maximize class separability while maintaining Gaussian class-conditional densities, so LDA output often aligns well with Naive Bayes' assumptions.**




```
=== Evaluation Summary Table ===
                        Label  Accuracy  Precision   Recall  F1 Score
Naive_bayes (LDA (1D))  0.977778    0.97619 0.974359  0.974321
Naive_bayes (LDA (2D))  1.000000    1.00000 1.000000  1.000000
```

6. Wine – LDA(to 2D)/PCA(to 6D) vs. Naïve Bayes/Perceptron:

```
=== Evaluation Summary Table ===
                        Label  Accuracy  Precision   Recall  F1 Score  AUC
Naive Bayes (Original)  1.000000  1.000000 1.000000  1.000000 None
 Perceptron (Original)  0.648148  0.434066 0.587302  0.499169 None
Naive Bayes (LDA (2D))  1.000000  1.000000 1.000000  1.000000 None
 Perceptron (LDA (2D))  0.740741  0.533333 0.666667  0.583333 None
Naive Bayes (PCA (6D))  1.000000  1.000000 1.000000  1.000000 None
 Perceptron (PCA (6D))  0.740741  0.533333 0.666667  0.583333 None
```



- **Naive Bayes:**

The original, after LDA, after PCA all have 100% accuracy. Naive Bayes achieved 100% accuracy because the Wine dataset has **well-separated classes**, and its Gaussian and independence assumptions are approximately satisfied. Additionally**, both LDA and PCA preserved or even enhanced class separability**, allowing the model to perform well. Finally, **the validation set was relatively small, making perfect performance more statistically likely compared to larger datasets**.

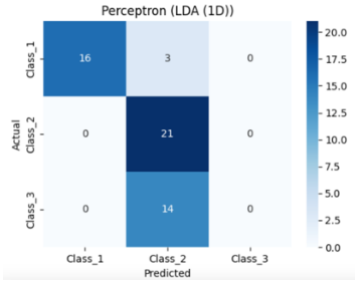
- **Perceptron:**

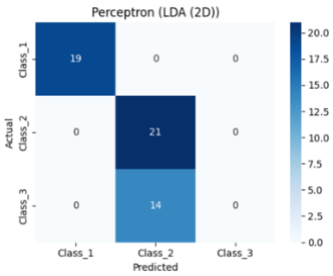
Accuracy improved from 64.8% to 74.1% for LDA. Class 3 is still poorly classified (all predicted as class 2). LDA projects the data into a space that maximizes class separation, which helps linear classifiers like Perceptron. However, the **improvement is limited due to possible overlaps between class 2 and class 3 even in LDA space**. **PCA performs as well as LDA for Perceptron here. Also, improvement for PCA, suggesting PCA preserved enough structure of the data to assist Perceptron.**

LDA is expected to be better for Perceptron because it is designed to enhance class separability, which directly benefits a linear classifier that relies on separating classes with a decision boundary. PCA, while useful for dimensionality reduction, does not optimize for class separability and thus may not help the Perceptron model as much. However, in this case, both **LDA and PCA resulted in the same performance for Perceptron**. This could be because **PCA retained enough of the important structure in the data** (by keeping 6 components) that helped Perceptron perform just as well as LDA, despite PCA not being optimized for class separability.

7. Wine – LDA(1D-2D) vs. Perceptron:

Accuracy improved from 68.52% → 74.07% going from 1D to 2D. Precision, recall, F1 also improved. LDA with 2 components should outperform 1 component because more components retain more discriminative information. 2D LDA allows separation across two orthogonal discriminative directions, which helps especially in overlapping class distributions. **LDA + Perceptron doesn't perform (Class 3 is still entirely misclassified) well this might be due to LDA projects for mean separation**, not full class separation. LDA maximizes between-class mean distances but doesn't explicitly reduce within-class variance. **If two classes (like class 2 and 3 in your case) have overlapping distributions, LDA may not separate them effectively**. LDA actually did improve the performance, LDA 2D improves class 1's separation (from 16→19 correct). So, the problem might also be due to perceptron. Perceptron is a hard linear classifier (no probability output, no margin), and **struggles with overlapping or not linearly separable classes**. The **residual overlap between class 2 and 3 makes it hard for Perceptron to classify correctly.**






```
=== Evaluation Summary Table ===
                    Label  Accuracy  Precision   Recall  F1 Score
Perceptron (LDA (1D))  0.685185   0.517544 0.614035  0.542050
Perceptron (LDA (2D))  0.740741   0.533333 0.666667  0.583333
```

Appendix (Code)

- Experiment 1,3,4,6:

```
import numpy as np
import matplotlib.pyplot as plt
from urllib.request import urlopen
import pandas as pd
from io import StringIO
from sklearn.model_selection import train_test_split
import seaborn as sns

# =======================
# Dataset Loaders
# =======================
def load_iris_dataset():
    """Load Iris dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"
    column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, :-1].values
        y = df.iloc[:, -1].values
        return X, y, column_names[:-1], np.unique(y)
    except:
        print("Error loading Iris dataset from URL. Using synthetic data instead.")
        # Create synthetic Iris data if URL fails
        from sklearn.datasets import load_iris
        iris = load_iris()
        X = iris.data
        y = np.array(['Iris-setosa', 'Iris-versicolor', 'Iris-
virginica'])[iris.target]
        return X, y, iris.feature_names, np.unique(y)

def load_breast_cancer_coimbra_dataset():
    """Load Breast Cancer Coimbra dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/00451/dataR2.csv"
    try:
        response = urlopen(url)
```

```
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data))
        X = df.iloc[:, :-1].values  # Features
        y = df.iloc[:, -1].values   # Classification column
        # Convert class to strings for consistency
        y = np.array(['Healthy' if label == 1 else 'Patient' for label in y])
        feature_names = df.columns[:-1].tolist()
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Breast Cancer Coimbra dataset from URL. Generating
synthetic data instead.")
        # Generate synthetic data if URL fails
        np.random.seed(42)
        n_samples = 116  # Actual dataset size
        n_features = 9   # Actual number of features
        X = np.random.randn(n_samples, n_features)
        y = np.array(['Healthy' if i < 58 else 'Patient' for i in range(n_samples)])
        feature_names = [
            'Age', 'BMI', 'Glucose', 'Insulin', 'HOMA', 'Leptin',
            'Adiponectin', 'Resistin', 'MCP.1'
        ]
        class_names = np.unique(y)
        return X, y, feature_names, class_names

def load_ionosphere_dataset():
    """Load Ionosphere dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/ionosphere/ionosphere.data"
    column_names = [f'feature_{i}' for i in range(34)] + ['class']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, :-1].values
        y = df.iloc[:, -1].values
        feature_names = column_names[:-1]
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Ionosphere dataset from URL. Using synthetic data
instead.")
```

```
        # Create synthetic ionosphere data if URL fails
        X = np.random.randn(351, 34)
        y = np.random.choice(['g', 'b'], size=351)
        feature_names = [f'feature_{i}' for i in range(34)]
        return X, y, feature_names, np.unique(y)

def load_wine_dataset():
    """Load Wine dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data"
    column_names = ['class', 'alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash',
'magnesium',
                'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
'proanthocyanins',
                'color_intensity', 'hue', 'od280/od315_of_diluted_wines',
'proline']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, 1:].values  # Features
        y = df.iloc[:, 0].values   # Class column
        # Convert class to strings for consistency
        y = np.array([f'Class_{int(label)}' for label in y])
        feature_names = column_names[1:]
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Wine dataset from URL. Using synthetic data instead.")
        # Create synthetic wine data if URL fails
        from sklearn.datasets import load_wine
        wine = load_wine()
        X = wine.data
        y = np.array([f'Class_{i+1}' for i in wine.target])
        return X, y, wine.feature_names, np.unique(y)

# ======================
# FLD / LDA Projection
# ======================
def lda_projection(X, y, num_components=1):
    """
    LDA projection to reduce data to 'num_components' dimensions.
```

```
    - num_components: The number of components to project onto.
      (1 for 2 classes, 2 for 3 classes)
    """
    class_labels = np.unique(y)
    mean_total = np.mean(X, axis=0)
    Sw = np.zeros((X.shape[1], X.shape[1]))
    Sb = np.zeros((X.shape[1], X.shape[1]))

    # Calculate Sw and Sb
    for c in class_labels:
        X_c = X[y == c]
        mean_c = np.mean(X_c, axis=0)
        Sw += (X_c - mean_c).T @ (X_c - mean_c)
        mean_diff = (mean_c - mean_total).reshape(-1, 1)
        Sb += X_c.shape[0] * (mean_diff @ mean_diff.T)

    # Compute the eigenvalues and eigenvectors
    eigvals, eigvecs = np.linalg.eig(np.linalg.pinv(Sw).dot(Sb))

    # Sort eigenvalues in descending order and pick the top 'num_components'
    sorted_indices = np.argsort(eigvals)[::-1]  # Indices of eigenvalues in
descending order
    top_eigenvectors = eigvecs[:, sorted_indices[:num_components]]  # Select top
'num_components' eigenvectors

    return top_eigenvectors.real, Sb, Sw

def project_data(X, W):
    """
    Project data onto the lower-dimensional subspace defined by W.
    - W: Matrix containing eigenvectors (e.g., top eigenvectors for LDA).
    """
    return X @ W


# ======================
# PCA Implementation
# ======================
def pca_fit(X, num_components):
    X_centered = X - np.mean(X, axis=0)
    cov = np.cov(X_centered, rowvar=False)
    eigvals, eigvecs = np.linalg.eigh(cov)
    idx = np.argsort(eigvals)[::-1]
```

```
    return eigvecs[:, idx[:num_components]]

def pca_transform(X, components):
    return (X - np.mean(X, axis=0)) @ components

# ======================
# Naive Bayes Classifier
# ======================
class NaiveBayesClassifier:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.mean = {}
        self.var = {}
        self.priors = {}
        for c in self.classes:
            X_c = X[y == c]
            self.mean[c] = X_c.mean(axis=0)
            self.var[c] = X_c.var(axis=0) + 1e-6  # add small value to prevent div by
zero
            self.priors[c] = X_c.shape[0] / X.shape[0]

    def _log_likelihood(self, x, c):
        prior = np.log(self.priors[c])
        likelihood = -0.5 * np.sum(np.log(2 * np.pi * self.var[c]))
        likelihood -= 0.5 * np.sum(((x - self.mean[c])**2) / self.var[c])
        return prior + likelihood

    def predict(self, X):
        preds = []
        for x in X:
            probs = [self._log_likelihood(x, c) for c in self.classes]
            preds.append(self.classes[np.argmax(probs)])
        return np.array(preds)

    def predict_log_proba(self, X):
        log_proba = []
        for x in X:
            log_probs = np.array([self._log_likelihood(x, c) for c in self.classes])
            log_probs -= np.max(log_probs)  # for numerical stability
            probs = np.exp(log_probs)
            probs /= probs.sum()
```

```
            # Avoid log(0) by adding a small epsilon to probs
            log_proba.append(np.log(probs + 1e-10))  # add small value to avoid
log(0)
        return np.array(log_proba)


    def predict_proba(self, X):
        return np.exp(self.predict_log_proba(X))


# ========================
# Perceptron Classifier
# ========================
class PerceptronClassifier:
    def __init__(self, lr=0.01, epochs=1000):
        self.lr = lr
        self.epochs = epochs

    def fit(self, X, y):
        self.classes = np.unique(y)
        y_bin = np.where(y == self.classes[0], -1, 1)
        self.w = np.zeros(X.shape[1])
        self.b = 0

        for _ in range(self.epochs):
            for i in range(X.shape[0]):
                if y_bin[i] * (np.dot(X[i], self.w) + self.b) <= 0:
                    self.w += self.lr * y_bin[i] * X[i]
                    self.b += self.lr * y_bin[i]

    def predict(self, X):
        preds = np.dot(X, self.w) + self.b
        return np.where(preds > 0, self.classes[1], self.classes[0])

# ========================
# Evaluation Metrics
# ========================
def manual_confusion_matrix(y_true, y_pred, labels):
    matrix = np.zeros((len(labels), len(labels)), dtype=int)
    label_to_index = {label: i for i, label in enumerate(labels)}
    for t, p in zip(y_true, y_pred):
        matrix[label_to_index[t], label_to_index[p]] += 1
    return matrix
```

```python
def manual_accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

def manual_precision(y_true, y_pred, positive_class):
    tp = np.sum((y_pred == positive_class) & (y_true == positive_class))
    fp = np.sum((y_pred == positive_class) & (y_true != positive_class))
    return tp / (tp + fp + 1e-10)

def manual_recall(y_true, y_pred, positive_class):
    tp = np.sum((y_pred == positive_class) & (y_true == positive_class))
    fn = np.sum((y_pred != positive_class) & (y_true == positive_class))
    return tp / (tp + fn + 1e-10)

def manual_f1(precision, recall):
    return 2 * precision * recall / (precision + recall + 1e-10)

def macro_avg_metrics(y_true, y_pred, labels):
    precisions, recalls, f1s = [], [], []
    for label in labels:
        p = manual_precision(y_true, y_pred, label)
        r = manual_recall(y_true, y_pred, label)
        f1 = manual_f1(p, r)
        precisions.append(p)
        recalls.append(r)
        f1s.append(f1)
    return {
        "Precision": np.mean(precisions),
        "Recall": np.mean(recalls),
        "F1 Score": np.mean(f1s)
    }

def manual_roc_auc(y_true, y_score, pos_label=1):
    desc_score_indices = np.argsort(-y_score)
    y_true_sorted = np.array(y_true)[desc_score_indices]
    y_score_sorted = np.array(y_score)[desc_score_indices]

    tpr = []
    fpr = []
    P = np.sum(y_true == pos_label)
    N = len(y_true) - P
    tp = 0
```

```
    fp = 0

    for i in range(len(y_score_sorted)):
        if y_true_sorted[i] == pos_label:
            tp += 1
        else:
            fp += 1
        tpr.append(tp / P)
        fpr.append(fp / N)

    auc = np.trapezoid(tpr, fpr)
    return fpr, tpr, auc


def plot_confusion_matrices(cm_data):
    n = len(cm_data)
    orig_rows = 3
    orig_cols = 2

    assert n == orig_rows * orig_cols, "cm_data 的長度應該為 6（3 列×2 行）"


    reordered = []
    for col in range(orig_cols):
        for row in range(orig_rows):
            reordered.append(cm_data[row * orig_cols + col])


    rows, cols = orig_cols, orig_rows
    fig, axs = plt.subplots(rows, cols, figsize=(cols * 5, rows * 4))

    for ax, (cm, labels, title) in zip(axs.flat, reordered):
        sns.heatmap(cm, annot=True, fmt='d', xticklabels=labels, yticklabels=labels,
cmap="Blues", ax=ax)
        ax.set_title(title)
        ax.set_xlabel("Predicted")
        ax.set_ylabel("Actual")

    plt.tight_layout()
    plt.show()

def plot_individual_roc_curves(roc_data):
    n = len(roc_data)
```

```
    orig_rows = 3
    orig_cols = 2

    assert n == orig_rows * orig_cols, "roc_data should contain exactly 6 entries
(3 rows × 2 cols)"

    # Reorder for column-wise display like confusion matrices
    reordered = []
    for col in range(orig_cols):
        for row in range(orig_rows):
            reordered.append(roc_data[row * orig_cols + col])

    rows, cols = orig_cols, orig_rows
    fig, axs = plt.subplots(rows, cols, figsize=(cols * 5, rows * 4))

    for ax, (fpr, tpr, label, auc) in zip(axs.flat, reordered):
        ax.plot(fpr, tpr, label=f"AUC = {auc:.2f}")
        ax.plot([0, 1], [0, 1], linestyle='--', color='gray')
        ax.set_title(label)
        ax.set_xlabel("False Positive Rate")
        ax.set_ylabel("True Positive Rate")
        ax.legend()
        ax.grid(True)

    plt.tight_layout()
    plt.show()

# =====================
# Evaluation Function
# =====================
def evaluate(y_true, y_pred, y_score=None, label=None, roc_curves=None):
    labels = np.unique(np.concatenate((y_true, y_pred)))
    cm = manual_confusion_matrix(y_true, y_pred, labels)

    print("Confusion Matrix:")
    print(cm)

    acc = manual_accuracy(y_true, y_pred)
    print(f"Accuracy: {acc:.4f}")

    metrics = {
        "Label": label,
```

```
        "Accuracy": acc,
        "Precision": None,
        "Recall": None,
        "F1 Score": None,
        "AUC": None
    }

    if len(labels) == 2:
        pos_class = labels[1]
        prec = manual_precision(y_true, y_pred, pos_class)
        rec = manual_recall(y_true, y_pred, pos_class)
        f1 = manual_f1(prec, rec)

        print(f"Precision: {prec:.4f}, Recall: {rec:.4f}, F1 Score: {f1:.4f}")
        metrics.update({"Precision": prec, "Recall": rec, "F1 Score": f1})

        if y_score is not None:
            fpr, tpr, auc = manual_roc_auc(y_true, y_score, pos_label=pos_class)
            print(f"AUC: {auc:.4f}")
            metrics["AUC"] = auc

            if roc_curves is not None:
                roc_curves.append((fpr, tpr, label, auc))

    else:
        avg_metrics = macro_avg_metrics(y_true, y_pred, labels)
        print(f"Macro Precision: {avg_metrics['Precision']:.4f}, "
              f"Recall: {avg_metrics['Recall']:.4f}, "
              f"F1 Score: {avg_metrics['F1 Score']:.4f}")
        metrics.update(avg_metrics)

    return cm, metrics



# =====================
# Main Pipeline
# =====================
def run_pipeline(load_dataset_func, num_components_pca, num_components_lda):
    X, y, feature_names, class_names = load_dataset_func()
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,
random_state=42)
```

```
    classifiers = {
        "Naive Bayes": NaiveBayesClassifier(),
        "Perceptron": PerceptronClassifier()
    }

    cm_data = []
    metric_rows = []
    roc_curves = []

    def evaluate_stage(X_train_sub, X_val_sub, label, y_score_func=None):
        for name, clf in classifiers.items():
            print(f"\n[{name} - {label}]")
            clf.fit(X_train_sub, y_train)
            y_pred = clf.predict(X_val_sub)
            y_score = y_score_func(clf, X_val_sub) if y_score_func else None

            cm, metrics = evaluate(
                y_val, y_pred, y_score,
                label=f"{name} ({label})",
                roc_curves=roc_curves  # Collect ROC data for later plotting
            )
            cm_data.append((cm, class_names, metrics["Label"]))
            metric_rows.append(metrics)

    # Original Data
    evaluate_stage(
        X_train, X_val, "Original",
        y_score_func=lambda clf, X: (
            clf.predict_proba(X)[:, 1] if isinstance(clf, NaiveBayesClassifier) else
np.dot(X, clf.w) + clf.b
        )
    )

    # LDA Projection
    print("\n=== FLD / LDA Projection ===")
    w, Sb, Sw = lda_projection(X_train, y_train, num_components_lda)
    X_train_lda = project_data(X_train, w)
    X_val_lda = project_data(X_val, w)
    evaluate_stage(
        X_train_lda, X_val_lda, f"LDA ({num_components_lda}D)",
        y_score_func=lambda clf, X: (
```

```
            clf.predict_proba(X)[:, 1] if isinstance(clf, NaiveBayesClassifier) else
np.dot(X, clf.w) + clf.b
        )
    )

    # PCA Projection
    print("\n=== PCA Projection ===")
    components = pca_fit(X_train, num_components_pca)
    X_train_pca = pca_transform(X_train, components)
    X_val_pca = pca_transform(X_val, components)
    evaluate_stage(
        X_train_pca, X_val_pca, f"PCA ({num_components_pca}D)",
        y_score_func=lambda clf, X: (
            clf.predict_proba(X)[:, 1] if isinstance(clf, NaiveBayesClassifier) else
np.dot(X, clf.w) + clf.b
        )
    )

    # Plot all ROC curves together (only for binary classification)
    if len(np.unique(y)) == 2 and roc_curves:
        plt.figure(figsize=(8, 6))
        for fpr, tpr, label, auc in roc_curves:
            plt.plot(fpr, tpr, label=f"{label} (AUC = {auc:.2f})")
        plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
        plt.xlabel("False Positive Rate")
        plt.ylabel("True Positive Rate")
        plt.title("All ROC Curves")
        plt.legend()
        plt.grid(True)
        plt.show()

    # Plot individual ROC curves (all classifiers together in subplots)
    if len(np.unique(y)) == 2 and len(roc_curves) == 6:
        plot_individual_roc_curves(roc_curves)

    # Plot confusion matrices
    plot_confusion_matrices(cm_data)

    # Print metrics
    df_metrics = pd.DataFrame(metric_rows)
    print("\n=== Evaluation Summary Table ===")
    print(df_metrics.to_string(index=False))
```

```
run_pipeline(load_breast_cancer_coimbra_dataset,num_components_pca=1,
num_components_lda=1)
run_pipeline(load_ionosphere_dataset, num_components_pca=17, num_components_lda=1)
run_pipeline(load_iris_dataset, num_components_pca=2,num_components_lda=2)
run_pipeline(load_wine_dataset,num_components_pca=6,num_components_lda=2)
```

- **Experiment 2,5,7:**

```
import numpy as np
import matplotlib.pyplot as plt
from urllib.request import urlopen
import pandas as pd
from io import StringIO
from sklearn.model_selection import train_test_split
import seaborn as sns

# ======================
# Dataset Loaders
# ======================
def load_iris_dataset():
    """Load Iris dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"
    column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, :-1].values
        y = df.iloc[:, -1].values
        return X, y, column_names[:-1], np.unique(y)
    except:
        print("Error loading Iris dataset from URL. Using synthetic data instead.")
        # Create synthetic Iris data if URL fails
        from sklearn.datasets import load_iris
        iris = load_iris()
        X = iris.data
        y = np.array(['Iris-setosa', 'Iris-versicolor', 'Iris-
virginica'])[iris.target]
        return X, y, iris.feature_names, np.unique(y)
```

```
def load_breast_cancer_coimbra_dataset():
    """Load Breast Cancer Coimbra dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/00451/dataR2.csv"
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data))
        X = df.iloc[:, :-1].values  # Features
        y = df.iloc[:, -1].values   # Classification column
        # Convert class to strings for consistency
        y = np.array(['Healthy' if label == 1 else 'Patient' for label in y])
        feature_names = df.columns[:-1].tolist()
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Breast Cancer Coimbra dataset from URL. Generating
synthetic data instead.")
        # Generate synthetic data if URL fails
        np.random.seed(42)
        n_samples = 116  # Actual dataset size
        n_features = 9   # Actual number of features
        X = np.random.randn(n_samples, n_features)
        y = np.array(['Healthy' if i < 58 else 'Patient' for i in range(n_samples)])
        feature_names = [
            'Age', 'BMI', 'Glucose', 'Insulin', 'HOMA', 'Leptin',
            'Adiponectin', 'Resistin', 'MCP.1'
        ]
        class_names = np.unique(y)
        return X, y, feature_names, class_names

def load_ionosphere_dataset():
    """Load Ionosphere dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/ionosphere/ionosphere.data"
    column_names = [f'feature_{i}' for i in range(34)] + ['class']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, :-1].values
        y = df.iloc[:, -1].values
```

```
        feature_names = column_names[:-1]
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Ionosphere dataset from URL. Using synthetic data
instead.")
        # Create synthetic ionosphere data if URL fails
        X = np.random.randn(351, 34)
        y = np.random.choice(['g', 'b'], size=351)
        feature_names = [f'feature_{i}' for i in range(34)]
        return X, y, feature_names, np.unique(y)

def load_wine_dataset():
    """Load Wine dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data"
    column_names = ['class', 'alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash',
'magnesium',
                    'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
'proanthocyanins',
                    'color_intensity', 'hue', 'od280/od315_of_diluted_wines',
'proline']
    try:
        response = urlopen(url)
        data = response.read().decode('utf-8')
        df = pd.read_csv(StringIO(data), header=None, names=column_names)
        X = df.iloc[:, 1:].values  # Features
        y = df.iloc[:, 0].values   # Class column
        # Convert class to strings for consistency
        y = np.array([f'Class_{int(label)}' for label in y])
        feature_names = column_names[1:]
        class_names = np.unique(y)
        return X, y, feature_names, class_names
    except:
        print("Error loading Wine dataset from URL. Using synthetic data instead.")
        # Create synthetic wine data if URL fails
        from sklearn.datasets import load_wine
        wine = load_wine()
        X = wine.data
        y = np.array([f'Class_{i+1}' for i in wine.target])
        return X, y, wine.feature_names, np.unique(y)
```

```
# ========================
# FLD / LDA Projection
# ========================
def lda_projection(X, y, num_components=1):
    """
    LDA projection to reduce data to 'num_components' dimensions.
    - num_components: The number of components to project onto.
      (1 for 2 classes, 2 for 3 classes)
    """
    class_labels = np.unique(y)
    mean_total = np.mean(X, axis=0)
    Sw = np.zeros((X.shape[1], X.shape[1]))
    Sb = np.zeros((X.shape[1], X.shape[1]))

    # Calculate Sw and Sb
    for c in class_labels:
        X_c = X[y == c]
        mean_c = np.mean(X_c, axis=0)
        Sw += (X_c - mean_c).T @ (X_c - mean_c)
        mean_diff = (mean_c - mean_total).reshape(-1, 1)
        Sb += X_c.shape[0] * (mean_diff @ mean_diff.T)

    # Compute the eigenvalues and eigenvectors
    eigvals, eigvecs = np.linalg.eig(np.linalg.pinv(Sw).dot(Sb))

    # Sort eigenvalues in descending order and pick the top 'num_components'
    sorted_indices = np.argsort(eigvals)[::-1]  # Indices of eigenvalues in
descending order
    top_eigenvectors = eigvecs[:, sorted_indices[:num_components]]  # Select top
'num_components' eigenvectors

    return top_eigenvectors.real, Sb, Sw

def project_data(X, W):
    """
    Project data onto the lower-dimensional subspace defined by W.
    - W: Matrix containing eigenvectors (e.g., top eigenvectors for LDA).
    """
    return X @ W


# ========================
# PCA Implementation
```

```python
# ========================
def pca_fit(X, num_components):
    X_centered = X - np.mean(X, axis=0)
    cov = np.cov(X_centered, rowvar=False)
    eigvals, eigvecs = np.linalg.eigh(cov)
    idx = np.argsort(eigvals)[::-1]
    return eigvecs[:, idx[:num_components]]

def pca_transform(X, components):
    return (X - np.mean(X, axis=0)) @ components

# ========================
# Naive Bayes Classifier
# ========================
class NaiveBayesClassifier:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.mean = {}
        self.var = {}
        self.priors = {}
        for c in self.classes:
            X_c = X[y == c]
            self.mean[c] = X_c.mean(axis=0)
            self.var[c] = X_c.var(axis=0) + 1e-6  # add small value to prevent div by
zero
            self.priors[c] = X_c.shape[0] / X.shape[0]

    def _log_likelihood(self, x, c):
        prior = np.log(self.priors[c])
        likelihood = -0.5 * np.sum(np.log(2 * np.pi * self.var[c]))
        likelihood -= 0.5 * np.sum(((x - self.mean[c])**2) / self.var[c])
        return prior + likelihood

    def predict(self, X):
        preds = []
        for x in X:
            probs = [self._log_likelihood(x, c) for c in self.classes]
            preds.append(self.classes[np.argmax(probs)])
        return np.array(preds)

    def predict_log_proba(self, X):
        log_proba = []
```

```
        for x in X:
            log_probs = np.array([self._log_likelihood(x, c) for c in self.classes])
            log_probs -= np.max(log_probs)  # for numerical stability
            probs = np.exp(log_probs)
            probs /= probs.sum()

            # Avoid log(0) by adding a small epsilon to probs
            log_proba.append(np.log(probs + 1e-10))  # add small value to avoid
log(0)
        return np.array(log_proba)


    def predict_proba(self, X):
        return np.exp(self.predict_log_proba(X))


# ======================
# Perceptron Classifier
# ======================
class PerceptronClassifier:
    def __init__(self, lr=0.01, epochs=1000):
        self.lr = lr
        self.epochs = epochs

    def fit(self, X, y):
        self.classes = np.unique(y)
        y_bin = np.where(y == self.classes[0], -1, 1)
        self.w = np.zeros(X.shape[1])
        self.b = 0

        for _ in range(self.epochs):
            for i in range(X.shape[0]):
                if y_bin[i] * (np.dot(X[i], self.w) + self.b) <= 0:
                    self.w += self.lr * y_bin[i] * X[i]
                    self.b += self.lr * y_bin[i]

    def predict(self, X):
        preds = np.dot(X, self.w) + self.b
        return np.where(preds > 0, self.classes[1], self.classes[0])

# ======================
# Evaluation Metrics
# ======================
```

```python
def manual_confusion_matrix(y_true, y_pred, labels):
    matrix = np.zeros((len(labels), len(labels)), dtype=int)
    label_to_index = {label: i for i, label in enumerate(labels)}
    for t, p in zip(y_true, y_pred):
        matrix[label_to_index[t], label_to_index[p]] += 1
    return matrix

def manual_accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

def manual_precision(y_true, y_pred, positive_class):
    tp = np.sum((y_pred == positive_class) & (y_true == positive_class))
    fp = np.sum((y_pred == positive_class) & (y_true != positive_class))
    return tp / (tp + fp + 1e-10)

def manual_recall(y_true, y_pred, positive_class):
    tp = np.sum((y_pred == positive_class) & (y_true == positive_class))
    fn = np.sum((y_pred != positive_class) & (y_true == positive_class))
    return tp / (tp + fn + 1e-10)

def manual_f1(precision, recall):
    return 2 * precision * recall / (precision + recall + 1e-10)

def macro_avg_metrics(y_true, y_pred, labels):
    precisions, recalls, f1s = [], [], []
    for label in labels:
        p = manual_precision(y_true, y_pred, label)
        r = manual_recall(y_true, y_pred, label)
        f1 = manual_f1(p, r)
        precisions.append(p)
        recalls.append(r)
        f1s.append(f1)
    return {
        "Precision": np.mean(precisions),
        "Recall": np.mean(recalls),
        "F1 Score": np.mean(f1s)
    }

def manual_roc_auc(y_true, y_score, pos_label=1):
    desc_score_indices = np.argsort(-y_score)
    y_true_sorted = np.array(y_true)[desc_score_indices]
    y_score_sorted = np.array(y_score)[desc_score_indices]
```

```
    tpr = []
    fpr = []
    P = np.sum(y_true == pos_label)
    N = len(y_true) - P
    tp = 0
    fp = 0

    for i in range(len(y_score_sorted)):
        if y_true_sorted[i] == pos_label:
            tp += 1
        else:
            fp += 1
        tpr.append(tp / P)
        fpr.append(fp / N)

    auc = np.trapezoid(tpr, fpr)
    return fpr, tpr, auc


def plot_confusion_matrices(cm_data):
    n = len(cm_data)
    # Dynamically determine rows and columns based on the number of confusion
matrices
    orig_rows = int(np.ceil(np.sqrt(n)))  # Square grid
    orig_cols = int(np.ceil(n / orig_rows))  # Calculate columns based on rows

    # Create subplots grid based on the number of matrices
    fig, axs = plt.subplots(orig_rows, orig_cols, figsize=(orig_cols * 5, orig_rows
* 4))

    # If axs is a single plot (not an array), make it iterable by putting it in a
list
    if isinstance(axs, np.ndarray):
        axs = axs.flatten()
    else:
        axs = [axs]

    # Plot each confusion matrix
    for i, (cm, labels, title) in enumerate(cm_data):
        ax = axs[i]
        sns.heatmap(cm, annot=True, fmt='d', xticklabels=labels, yticklabels=labels,
cmap="Blues", ax=ax)
```

```
        ax.set_title(title)
        ax.set_xlabel("Predicted")
        ax.set_ylabel("Actual")

    # Hide any unused axes (if there are more axes than confusion matrices)
    for j in range(i + 1, len(axs)):
        axs[j].axis('off')

    plt.tight_layout()
    plt.show()


def plot_individual_roc_curves(roc_data):
    n = len(roc_data)
    orig_rows = 3
    orig_cols = 2

    assert n == orig_rows * orig_cols, "roc_data should contain exactly 6 entries
(3 rows × 2 cols)"

    # Reorder for column-wise display like confusion matrices
    reordered = []
    for col in range(orig_cols):
        for row in range(orig_rows):
            reordered.append(roc_data[row * orig_cols + col])

    rows, cols = orig_cols, orig_rows
    fig, axs = plt.subplots(rows, cols, figsize=(cols * 5, rows * 4))

    for ax, (fpr, tpr, label, auc) in zip(axs.flat, reordered):
        ax.plot(fpr, tpr, label=f"AUC = {auc:.2f}")
        ax.plot([0, 1], [0, 1], linestyle='--', color='gray')
        ax.set_title(label)
        ax.set_xlabel("False Positive Rate")
        ax.set_ylabel("True Positive Rate")
        ax.legend()
        ax.grid(True)

    plt.tight_layout()
    plt.show()

# =====================
# Evaluation Function
```

```python
# ====================
def evaluate(y_true, y_pred, y_score=None, label=None, roc_curves=None):
    labels = np.unique(np.concatenate((y_true, y_pred)))
    cm = manual_confusion_matrix(y_true, y_pred, labels)

    print("Confusion Matrix:")
    print(cm)

    acc = manual_accuracy(y_true, y_pred)
    print(f"Accuracy: {acc:.4f}")

    metrics = {
        "Label": label,
        "Accuracy": acc,
        "Precision": None,
        "Recall": None,
        "F1 Score": None,
        "AUC": None
    }

    if len(labels) == 2:
        pos_class = labels[1]
        prec = manual_precision(y_true, y_pred, pos_class)
        rec = manual_recall(y_true, y_pred, pos_class)
        f1 = manual_f1(prec, rec)

        print(f"Precision: {prec:.4f}, Recall: {rec:.4f}, F1 Score: {f1:.4f}")
        metrics.update({"Precision": prec, "Recall": rec, "F1 Score": f1})

        if y_score is not None:
            fpr, tpr, auc = manual_roc_auc(y_true, y_score, pos_label=pos_class)
            print(f"AUC: {auc:.4f}")
            metrics["AUC"] = auc

            if roc_curves is not None:
                roc_curves.append((fpr, tpr, label, auc))

    else:
        avg_metrics = macro_avg_metrics(y_true, y_pred, labels)
        print(f"Macro Precision: {avg_metrics['Precision']:.4f}, "
              f"Recall: {avg_metrics['Recall']:.4f}, "
              f"F1 Score: {avg_metrics['F1 Score']:.4f}")
```

```
        metrics.update(avg_metrics)

    return cm, metrics



# =====================
# Main Pipeline
# =====================
def run_pipeline(load_dataset_func, num_components_pca, num_components_lda,
classifier_type, reduction_type):
    X, y, feature_names, class_names = load_dataset_func()
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,
random_state=42)

    classifiers = {
        "Naive Bayes": NaiveBayesClassifier(),
        "Perceptron": PerceptronClassifier()
    }

    # Select classifier based on user input
    if classifier_type == 'naive_bayes':
        selected_classifier = classifiers["Naive Bayes"]
    elif classifier_type == 'perceptron':
        selected_classifier = classifiers["Perceptron"]
    else:
        raise ValueError("Invalid classifier type. Choose 'naive_bayes' or
'perceptron'.")

    cm_data = []
    metric_rows = []
    roc_curves = []

    def evaluate_stage(X_train_sub, X_val_sub, label, y_score_func=None):
        print(f"\n[{classifier_type.capitalize()} - {label}]")
        selected_classifier.fit(X_train_sub, y_train)
        y_pred = selected_classifier.predict(X_val_sub)
        y_score = y_score_func(selected_classifier, X_val_sub) if y_score_func else
None

        cm, metrics = evaluate(
            y_val, y_pred, y_score,
            label=f"{classifier_type.capitalize()} ({label})",
```

```
            roc_curves=roc_curves  # Collect ROC data for later plotting
        )
        cm_data.append((cm, class_names, metrics["Label"]))
        metric_rows.append(metrics)

    # Loop for PCA if reduction_type is 'pca'
    if reduction_type == 'pca':
        print(f"\n=== PCA Projection with {num_components_pca} Components ===")
        for num_pca in range(1, num_components_pca + 1):
            components = pca_fit(X_train, num_pca)  # Fit PCA for num_pca components
            X_train_pca = pca_transform(X_train, components)
            X_val_pca = pca_transform(X_val, components)
            evaluate_stage(
                X_train_pca, X_val_pca, f"PCA ({num_pca}D)",
                y_score_func=lambda clf, X: (
                    clf.predict_proba(X)[:, 1] if isinstance(clf,
NaiveBayesClassifier) else np.dot(X, clf.w) + clf.b
                )
            )

    # Loop for LDA if reduction_type is 'lda'
    elif reduction_type == 'lda':
        for num_lda in range(1, num_components_lda + 1):
            print(f"\n=== FLD / LDA Projection with {num_lda} Components ===")
            w, Sb, Sw = lda_projection(X_train, y_train, num_lda)
            X_train_lda = project_data(X_train, w)
            X_val_lda = project_data(X_val, w)
            evaluate_stage(
                X_train_lda, X_val_lda, f"LDA ({num_lda}D)",
                y_score_func=lambda clf, X: (
                    clf.predict_proba(X)[:, 1] if isinstance(clf,
NaiveBayesClassifier) else np.dot(X, clf.w) + clf.b
                )
            )

    else:
        raise ValueError("Invalid reduction type. Choose 'pca' or 'lda'.")

    # Plot confusion matrices
    plot_confusion_matrices(cm_data)

    # Plot all ROC curves together (only for binary classification)
```

```
    if len(np.unique(y)) == 2 and roc_curves:
        plt.figure(figsize=(8, 6))
        for fpr, tpr, label, auc in roc_curves:
            plt.plot(fpr, tpr, label=f"{label} (AUC = {auc:.2f})")
        plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
        plt.xlabel("False Positive Rate")
        plt.ylabel("True Positive Rate")
        plt.title("All ROC Curves")
        plt.legend()
        plt.grid(True)
        plt.show()

    # Plot individual ROC curves (all classifiers together in subplots)
    if len(np.unique(y)) == 2 and len(roc_curves) == 6:
        plot_individual_roc_curves(roc_curves)

    # Print metrics
    df_metrics = pd.DataFrame(metric_rows)
    print("\n=== Evaluation Summary Table ===")
    print(df_metrics.to_string(index=False))
```

```
run_pipeline(load_breast_cancer_coimbra_dataset, num_components_pca=9,
num_components_lda=1, classifier_type='naive_bayes', reduction_type='pca')

run_pipeline(load_breast_cancer_coimbra_dataset, num_components_pca=9,
num_components_lda=1, classifier_type='perceptron', reduction_type='pca')

run_pipeline(load_iris_dataset, num_components_pca=0, num_components_lda=2,
classifier_type='naive_bayes', reduction_type='lda')

run_pipeline(load_wine_dataset, num_components_pca=0, num_components_lda=2,
classifier_type='perceptron', reduction_type='lda')
```