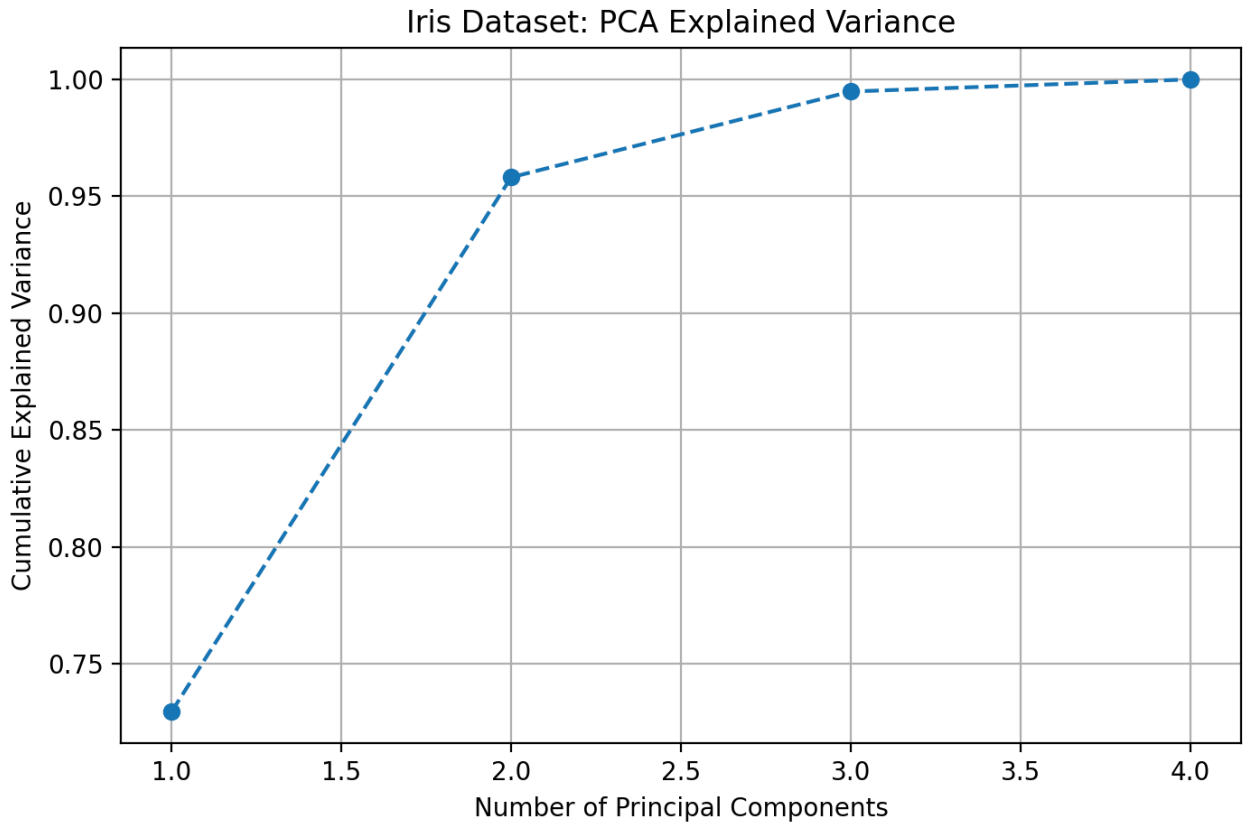
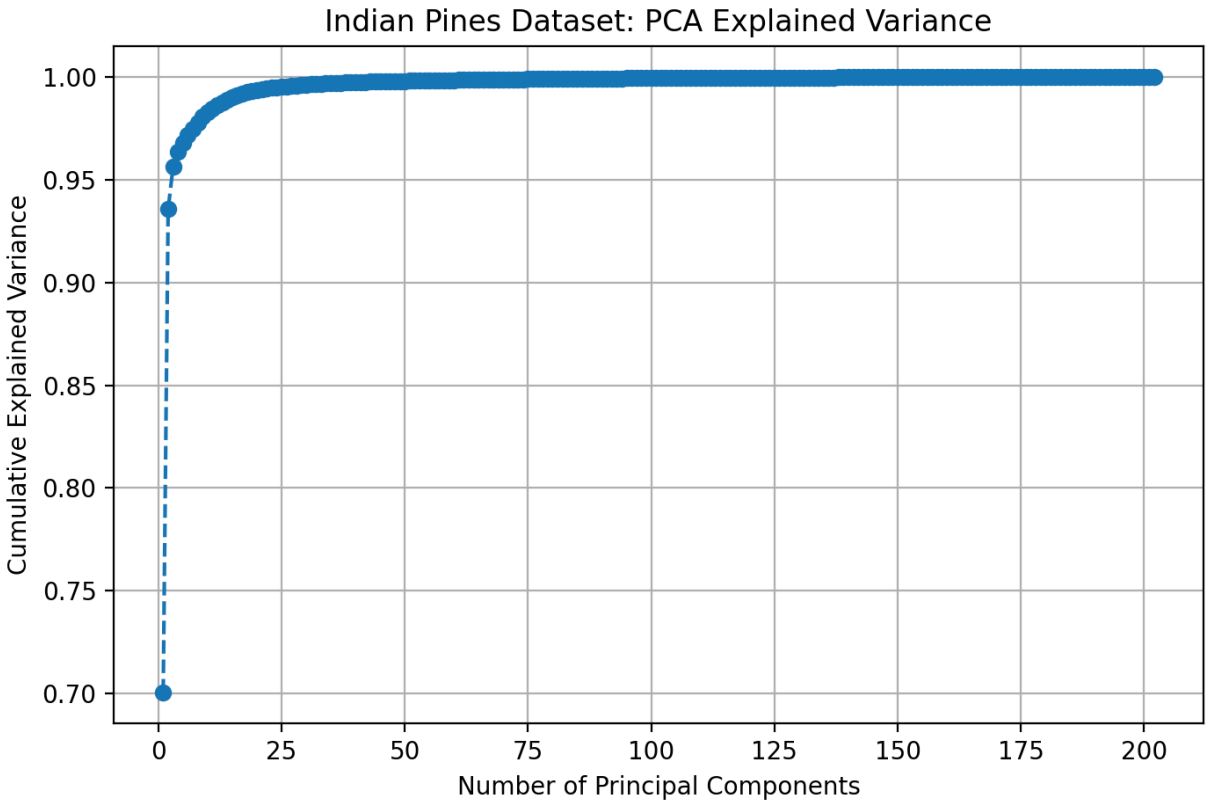
Homework 3

Casey Bramlett
CS488 Intro to Big Data
3/19/25

1a. Dimensionality Reduction with PCA

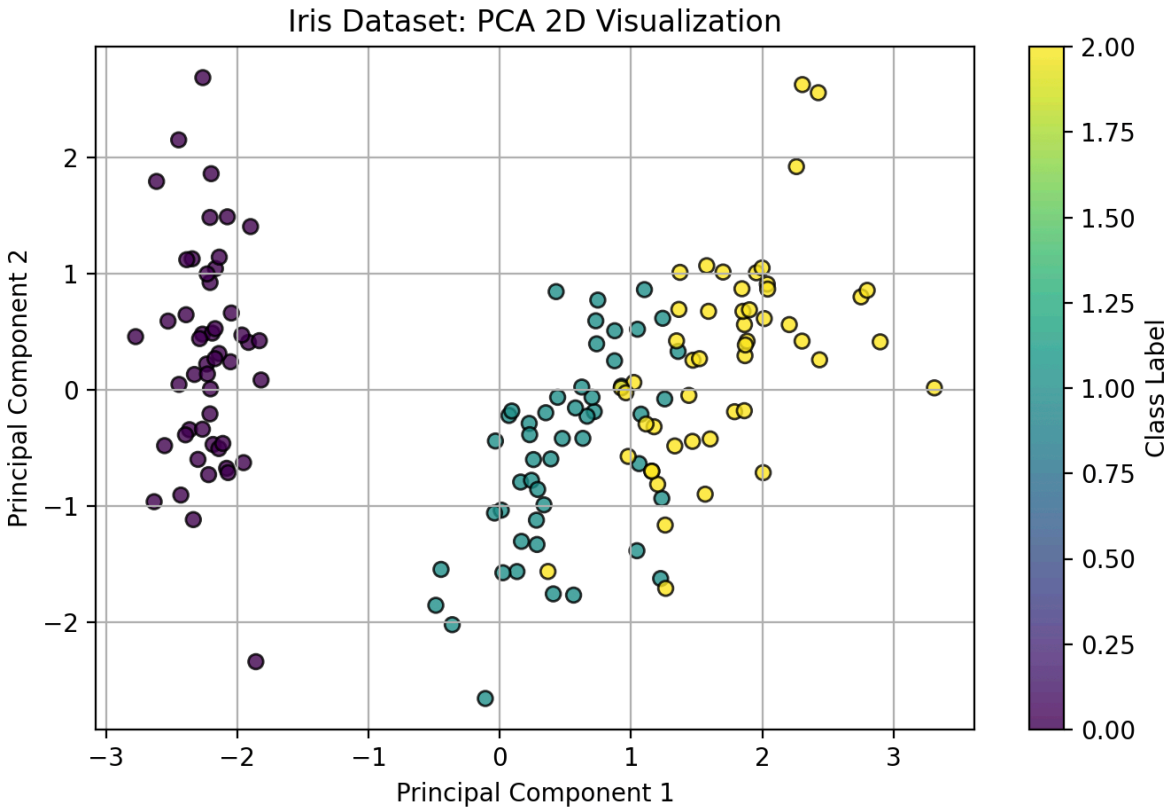
i) PCA Explained Variance Plot (5 points)

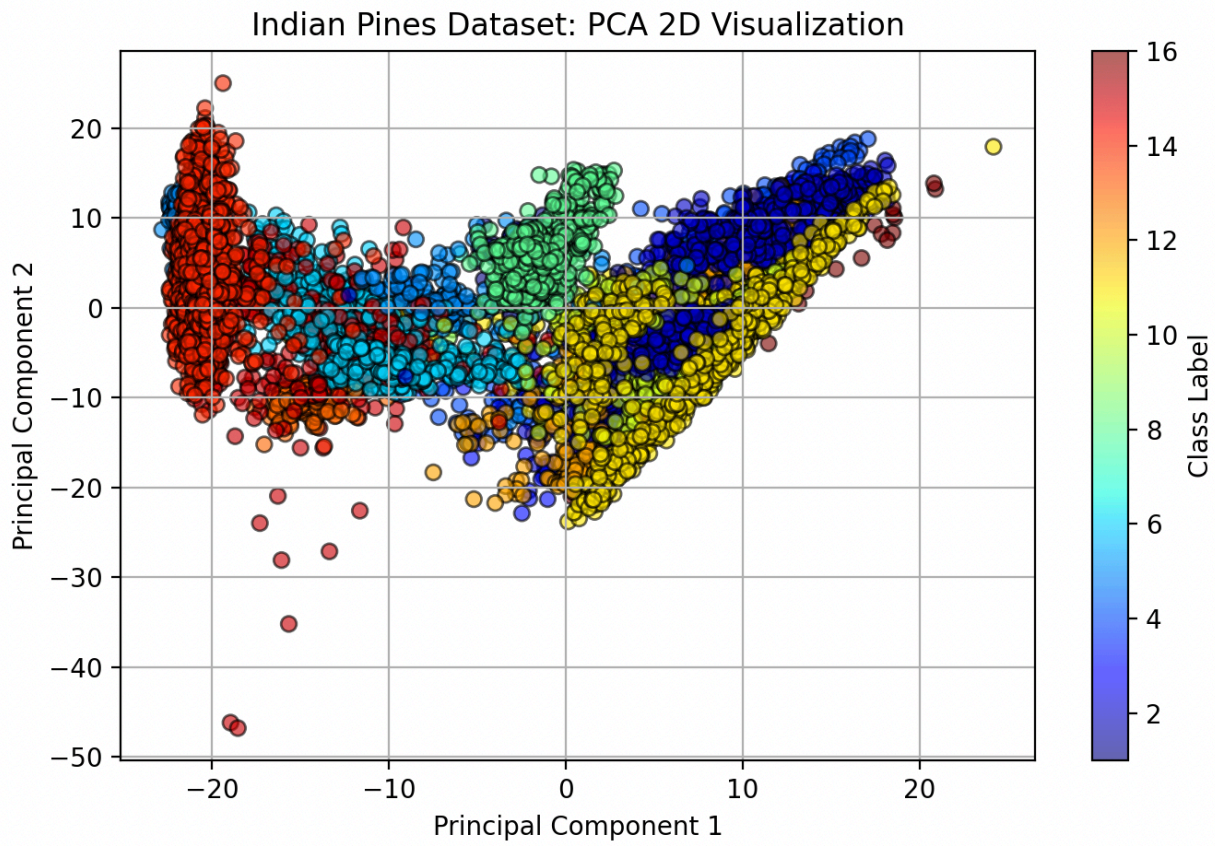




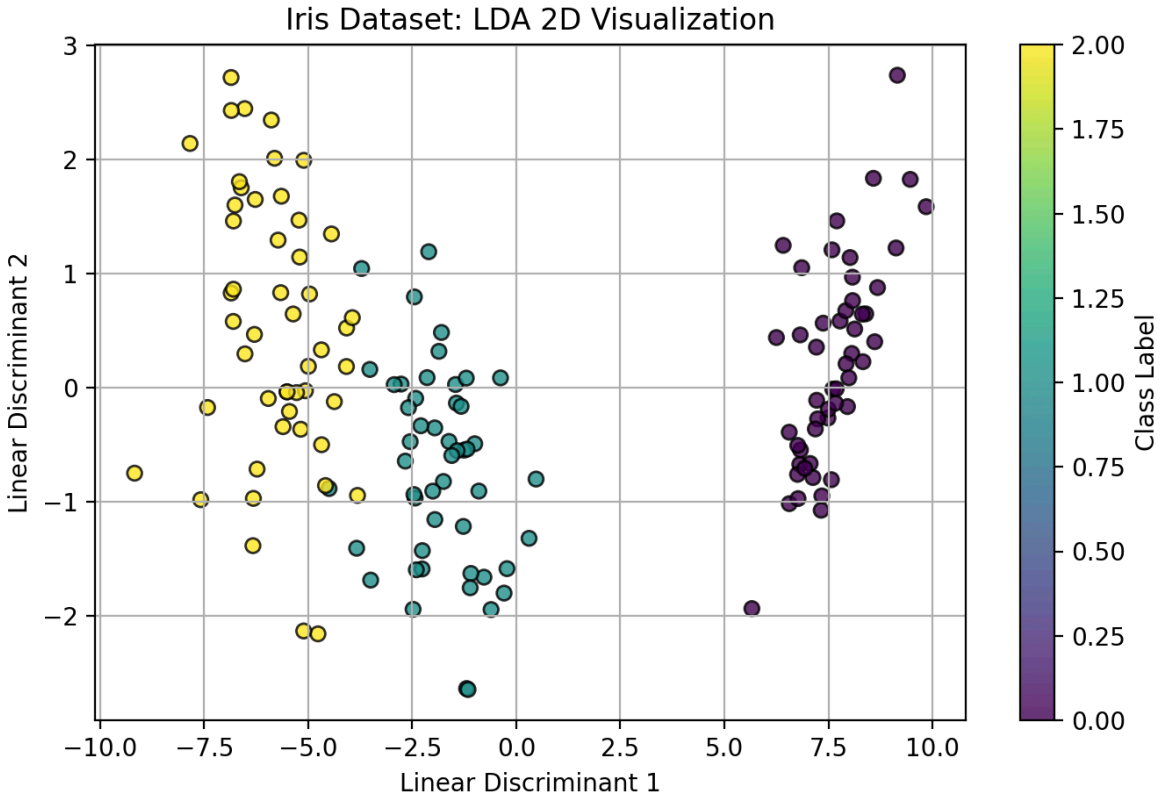
---

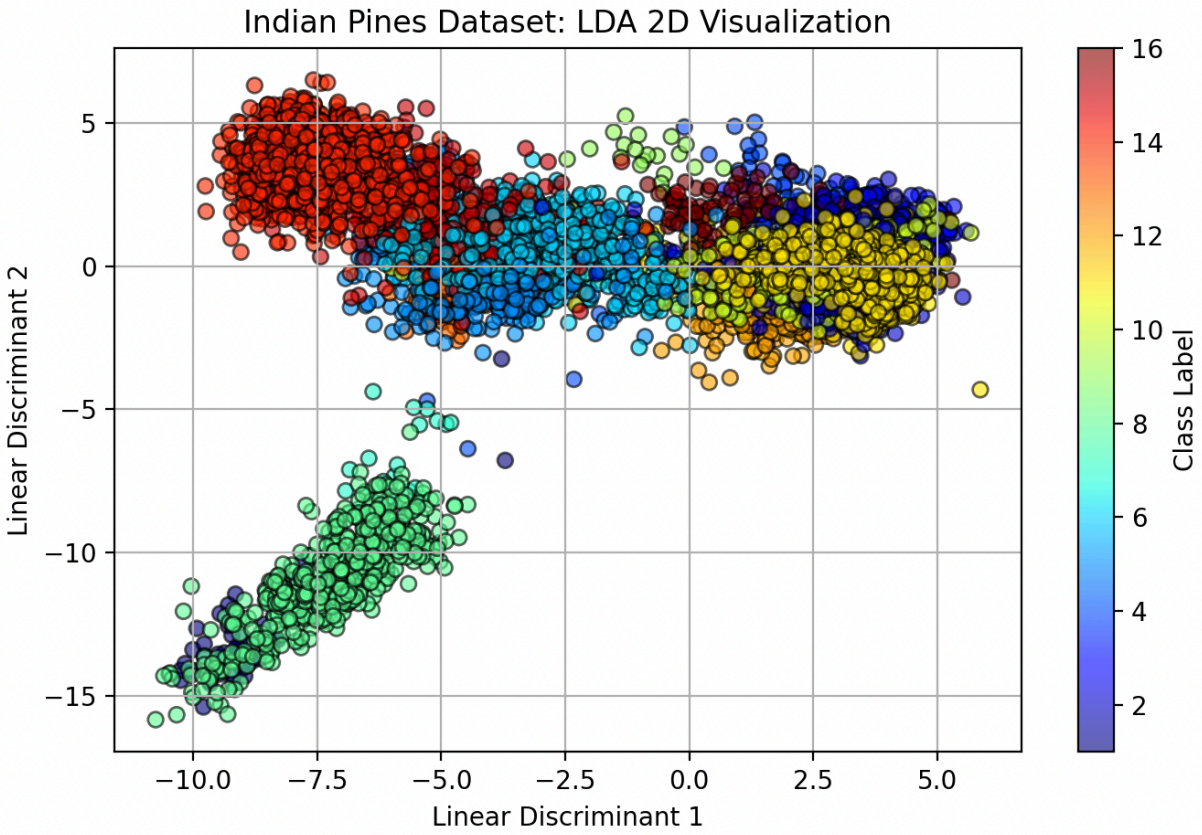
ii) PCA: 2D Reduced Data Visualization using PCA to 2 Dimensions


Iris Dataset: PCA 2D Visualization
Principal Component 2
Principal Component 1
Class Label
2
1
0
−1
−2
−3
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00


Indian Pines Dataset: PCA 2D Visualization
Principal Component 2
Principal Component 1
Class Label
20
10
0
−10
−20
−30
−40
−50
−20
−10
0
10
20
16
14
12
10
8
6
4
2

iii) LDA: 2D Reduced Data Visualization





1b. Analysis of PCA and LDA Results

Iris Dataset

- **Dimensionality Reduction Role:** PCA reduced four features to two while retaining 95%+ variance. LDA maximized class separability.
- **Data Separability:** PCA showed some class overlap, while LDA provided clearer separation.
- **Choice of Components:** K = 2 PCs captured most variance; LDA retained 2 components (C-1).
- **Best Method:** LDA outperformed PCA, as it optimized class separation, whereas PCA preserved variance without considering class labels.

Indian Pines Dataset

- **Dimensionality Reduction Role:** PCA helped reduce high-dimensional data (hundreds of bands), while LDA focused on class separation.
- **Data Separability:** PCA 2D plot showed overlapping clusters, while LDA provided better-defined groups.
- **Choice of Components:** K = 20–30 PCs retained most variance; LDA used up to 15 components (C-1).
- **Best Method:** LDA performed better for classification, though PCA is still useful for feature reduction.

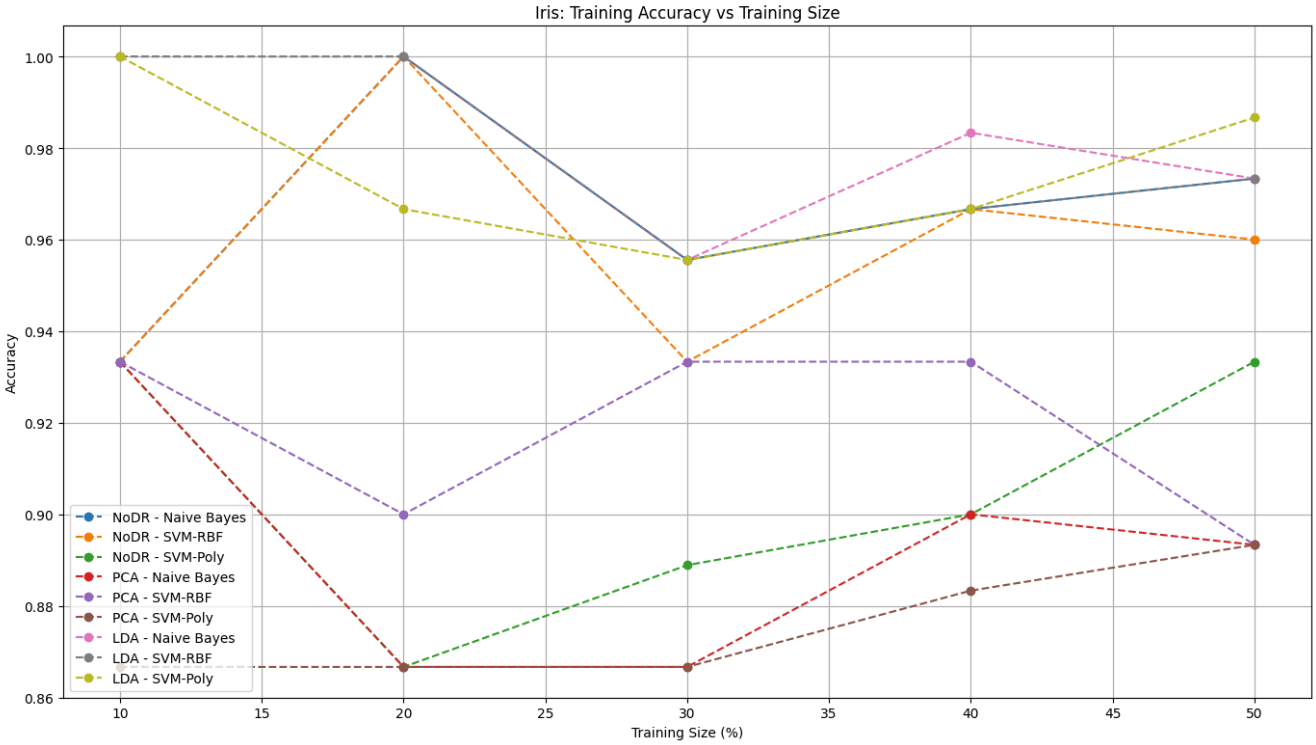
Conclusion

Dataset	Best Method	Reason
Iris	LDA	Better class separation vs. PCA's overlap.
Indian Pines	LDA	More effective for high-dimensional classification.

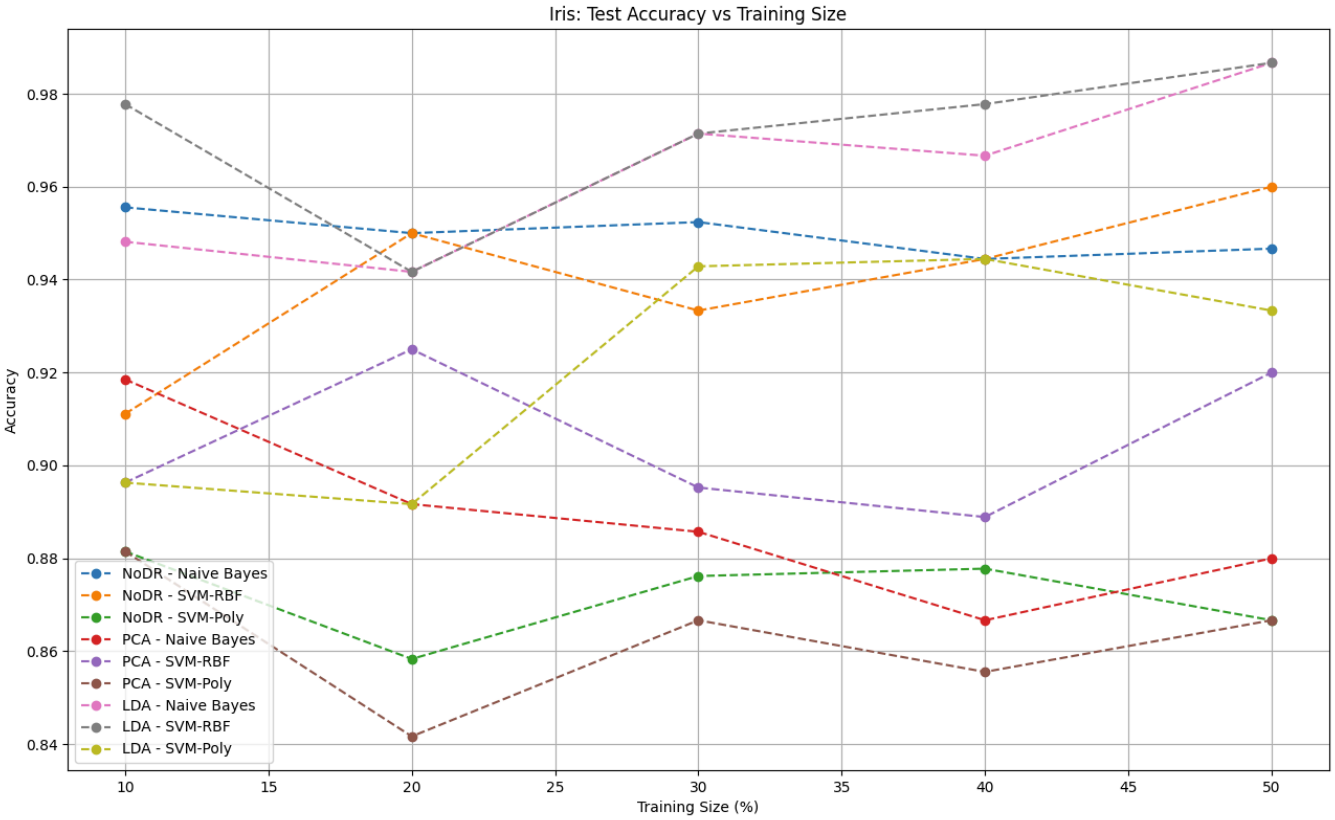
- PCA is best for feature extraction, while LDA excels at class separability.
- For Iris, 2 PCs are sufficient, and Indian Pines benefits from 20–30 PCs.
- LDA is preferred when labeled data is available, making it superior for classification.

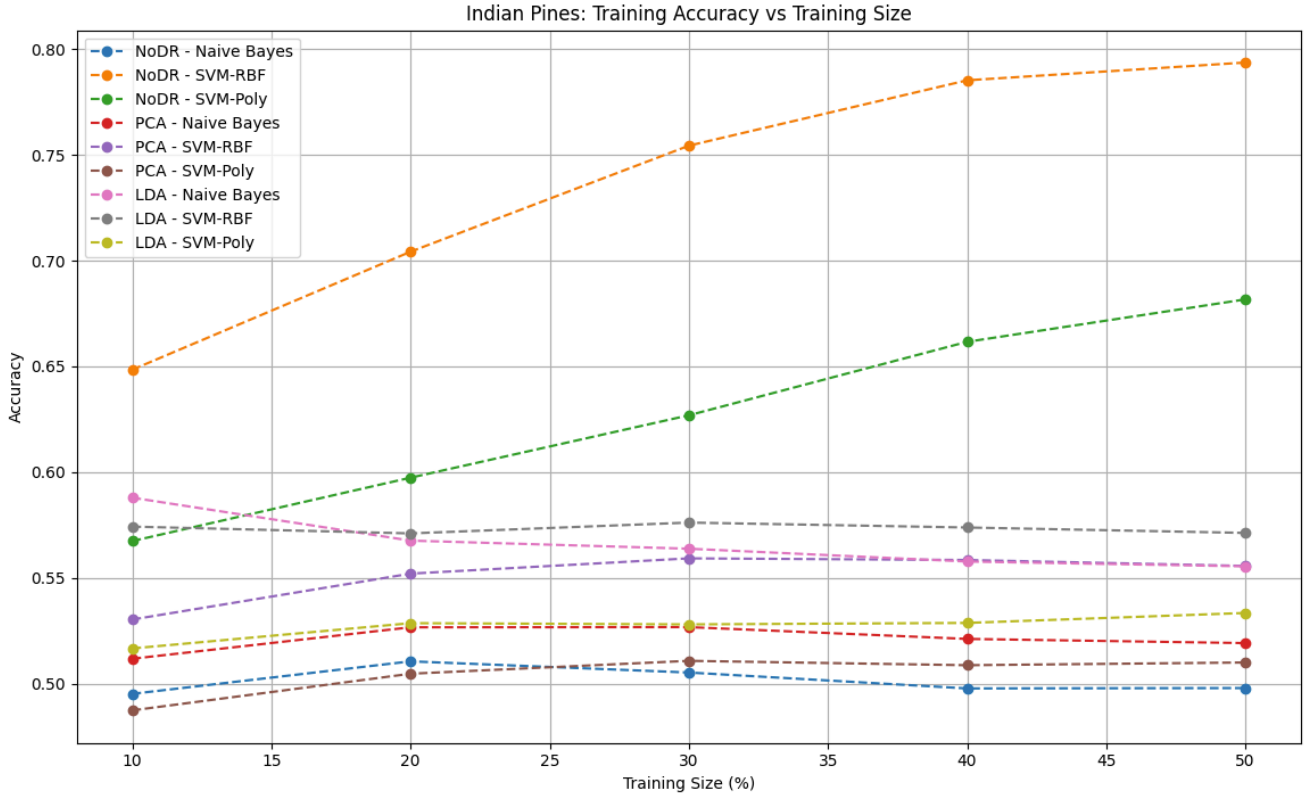
2a

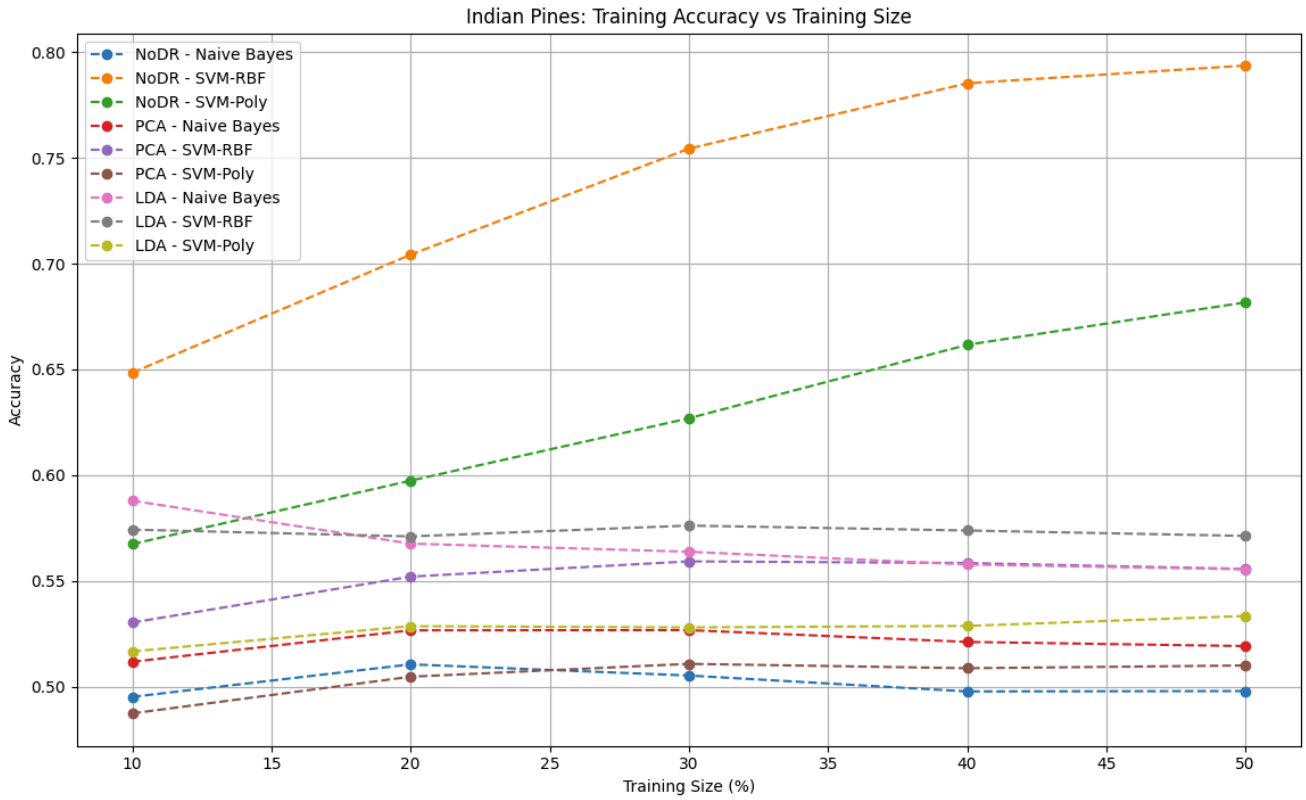
i)


Iris: Training Accuracy vs Training Size
Accuracy
Training Size (%)
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
10
15
20
25
30
35
40
45
50
NoDR - Naive Bayes
NoDR - SVM-RBF
NoDR - SVM-Poly
PCA - Naive Bayes
PCA - SVM-RBF
PCA - SVM-Poly
LDA - Naive Bayes
LDA - SVM-RBF
LDA - SVM-Poly

ii)


Iris: Test Accuracy vs Training Size
Accuracy
Training Size (%)
NoDR - Naive Bayes
NoDR - SVM-RBF
NoDR - SVM-Poly
PCA - Naive Bayes
PCA - SVM-RBF
PCA - SVM-Poly
LDA - Naive Bayes
LDA - SVM-RBF
LDA - SVM-Poly


Indian Pines: Training Accuracy vs Training Size
Accuracy
Training Size (%)
NoDR - Naive Bayes
NoDR - SVM-RBF
NoDR - SVM-Poly
PCA - Naive Bayes
PCA - SVM-RBF
PCA - SVM-Poly
LDA - Naive Bayes
LDA - SVM-RBF
LDA - SVM-Poly



Classwise Classification Accuracies for Indian Pines (30% Training) with DR

DR Method: PCA

Class	Naive Bayes	SVM-RBF	SVM-Poly
0	0.094	0.000	0.000
1	0.335	0.319	0.261
2	0.000	0.000	0.000
3	0.018	0.090	0.000
4	0.030	0.228	0.027
5	0.830	0.902	0.861
6	0.000	0.000	0.000
7	0.952	0.988	0.496
8	0.000	0.000	0.000
9	0.085	0.000	0.000
10	0.823	0.946	0.965

Class	Naive Bayes	SVM-RBF	SVM-Poly
11	0.188	0.161	0.036
12	0.909	0.902	0.881
13	0.975	0.982	0.976
14	0.063	0.026	0.048
15	0.000	0.000	0.000

DR Method: LDA

Class	Naive Bayes	SVM-RBF	SVM-Poly
0	0.562	0.000	0.000
1	0.108	0.004	0.003
2	0.583	0.568	0.466
3	0.120	0.072	0.000
4	0.272	0.607	0.683
5	0.849	0.879	0.795
6	0.700	0.000	0.000
7	0.896	1.000	0.997
8	0.643	0.643	0.143
9	0.018	0.010	0.000
10	0.776	0.895	0.948
11	0.458	0.434	0.036
12	0.664	0.804	0.000
13	0.960	0.954	0.953
14	0.241	0.000	0.030
15	0.446	0.354	0.000

Iris Dataset

- **Role of Dimensionality Reduction:**
  PCA/LDA: Since Iris is already low-dimensional, reducing to 2 dimensions preserves class separability well. Visualizations show clear clusters.

- **Impact on Classification:**
  High training and test accuracies are achieved in both DR and NoDR cases. DR does not drastically change performance but can reduce noise.
- **Best Method:**
  SVM-RBF generally shows robust performance with high sensitivity and specificity, likely due to its ability to capture non-linear boundaries.
- **Visualization:**
  Accuracy plots illustrate that even with 10–50% training data, SVM-RBF consistently maintains high test accuracy.

Indian Pines Dataset

- **Role of Dimensionality Reduction:**
  PCA vs. LDA: With hundreds of spectral bands, DR is critical. PCA reduces noise but is unsupervised, while LDA uses class labels to enhance separability.
- **Impact on Classification:**
  LDA typically improves sensitivity and specificity by better separating similar classes, as seen in the classwise accuracy tables.
- **Best Method:**
  SVM-RBF with LDA tends to outperform others, achieving higher classification accuracy on complex hyperspectral data by effectively leveraging the reduced feature space.
- **Visualization:**
  Plots reveal that as training size increases, methods with DR (especially LDA) yield improved test accuracies and more balanced per-class performance.

Final Conclusion

Dimensionality reduction is crucial for high-dimensional data like Indian Pines to improve separability and classification performance. In contrast, for low-dimensional datasets like Iris, even without DR, classifiers (especially SVM-RBF) perform exceptionally well. Visualizations confirm these trends in both overall accuracy and per-class metrics.

Appendix A

Homework3.py (for 1a-1b)

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# ================================
#           IRIS DATASET
# ================================
print("Processing Iris dataset...")

# Load Iris dataset
iris = load_iris()
X_iris = iris.data
y_iris = iris.target

# Standardize the Iris data
scaler = StandardScaler()
X_iris_scaled = scaler.fit_transform(X_iris)

# ------------------------------------------------
# i) PCA: Plot the explained variance for all PCs
# ------------------------------------------------
pca_iris = PCA()
X_iris_pca_all = pca_iris.fit_transform(X_iris_scaled)
explained_variance_ratio_iris = pca_iris.explained_variance_ratio_

plt.figure(figsize=(8, 5))
plt.plot(range(1, len(explained_variance_ratio_iris) + 1),
         np.cumsum(explained_variance_ratio_iris),
         marker='o', linestyle='--')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Iris Dataset: PCA Explained Variance')
plt.grid(True)
plt.show()

# ------------------------------------------------
# ii) PCA: Reduce data to 2 dimensions and visualize
# ------------------------------------------------
pca_iris_2D = PCA(n_components=2)
X_iris_pca_2D = pca_iris_2D.fit_transform(X_iris_scaled)
```

```
plt.figure(figsize=(8, 5))
plt.scatter(X_iris_pca_2D[:, 0], X_iris_pca_2D[:, 1],
            c=y_iris, cmap='viridis', edgecolors='k', alpha=0.8)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('Iris Dataset: PCA 2D Visualization')
plt.colorbar(label='Class Label')
plt.grid(True)
plt.show()

# ------------------------------------------------
# iii) LDA: Reduce data to 2 dimensions and visualize
# ------------------------------------------------
lda_iris = LinearDiscriminantAnalysis(n_components=2)
X_iris_lda_2D = lda_iris.fit_transform(X_iris_scaled, y_iris)

plt.figure(figsize=(8, 5))
plt.scatter(X_iris_lda_2D[:, 0], X_iris_lda_2D[:, 1],
            c=y_iris, cmap='viridis', edgecolors='k', alpha=0.8)
plt.xlabel('Linear Discriminant 1')
plt.ylabel('Linear Discriminant 2')
plt.title('Iris Dataset: LDA 2D Visualization')
plt.colorbar(label='Class Label')
plt.grid(True)
plt.show()

# ================================
#       INDIAN PINES DATASET
# ================================
print("Processing Indian Pines dataset...")

# Load the Indian Pines dataset and ground truth labels
indian_pines = scipy.io.loadmat("indianR.mat")
indian_labels = scipy.io.loadmat("indian_gth.mat")

# (Optional) Print keys to check the structure of the .mat file
print("Indian Pines keys:", indian_pines.keys())

# Extract the hyperspectral data and transpose so that each row is a sample
(pixel)
X_pines = indian_pines['X'].T

# Extract and flatten the ground truth labels
y_pines = indian_labels['gth'].flatten()
```

```
# Verify dimensions
print("X_pines shape:", X_pines.shape)  # Expected: (number of pixels,
number of spectral bands)
print("y_pines shape:", y_pines.shape)  # Expected: (number of pixels,)

# Remove zero labels (unclassified areas) if dimensions match
if X_pines.shape[0] == y_pines.shape[0]:
    mask = y_pines > 0
    X_pines = X_pines[mask]
    y_pines = y_pines[mask]
else:
    print("Dimension mismatch! Check dataset preprocessing.")

# Standardize the Indian Pines data
X_pines_scaled = scaler.fit_transform(X_pines)

# ------------------------------------------------
# i) PCA: Plot the explained variance for all PCs
# ------------------------------------------------
pca_pines = PCA()
X_pines_pca_all = pca_pines.fit_transform(X_pines_scaled)
explained_variance_ratio_pines = pca_pines.explained_variance_ratio_

plt.figure(figsize=(8, 5))
plt.plot(range(1, len(explained_variance_ratio_pines) + 1),
         np.cumsum(explained_variance_ratio_pines),
         marker='o', linestyle='--')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Indian Pines Dataset: PCA Explained Variance')
plt.grid(True)
plt.show()

# ------------------------------------------------
# ii) PCA: Reduce data to 2 dimensions and visualize
# ------------------------------------------------
pca_pines_2D = PCA(n_components=2)
X_pines_pca_2D = pca_pines_2D.fit_transform(X_pines_scaled)

plt.figure(figsize=(8, 5))
plt.scatter(X_pines_pca_2D[:, 0], X_pines_pca_2D[:, 1],
            c=y_pines, cmap='jet', edgecolors='k', alpha=0.6)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('Indian Pines Dataset: PCA 2D Visualization')
plt.colorbar(label='Class Label')
```

```
plt.grid(True)
plt.show()

# ------------------------------------------------
# iii) LDA: Reduce data to 2 dimensions and visualize
# ------------------------------------------------
lda_pines = LinearDiscriminantAnalysis(n_components=2)
X_pines_lda_2D = lda_pines.fit_transform(X_pines_scaled, y_pines)

plt.figure(figsize=(8, 5))
plt.scatter(X_pines_lda_2D[:, 0], X_pines_lda_2D[:, 1],
            c=y_pines, cmap='jet', edgecolors='k', alpha=0.6)
plt.xlabel('Linear Discriminant 1')
plt.ylabel('Linear Discriminant 2')
plt.title('Indian Pines Dataset: LDA 2D Visualization')
plt.colorbar(label='Class Label')
plt.grid(True)
plt.show()
```

Homework3_pt2.py (for 2a-2b)

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.base import clone
from sklearn.metrics import confusion_matrix

# ------------------------------
# Helper function to run experiments
# ------------------------------
def run_classification_experiment(X, y, dr_choice, train_sizes,
random_state=42):
    """
    Run classification experiments over different training sizes.
```

```
    Parameters:
      X          : data features (numpy array)
      y          : labels (numpy array)
      dr_choice  : string indicating the DR method to use:
                   'NoDR' for no dimensionality reduction,
                   'PCA' for PCA (with n_components=2),
                   'LDA' for LDA (with n_components=2).
      train_sizes: list of training sizes (fractions)
      random_state: random state for reproducibility

    Returns:
      results: dictionary with keys as training sizes. For each training
size,
               a dictionary of classifier performance metrics is stored.
    """
    # Define classifiers to test
    classifiers = {
        'Naive Bayes': GaussianNB(),
        'SVM-RBF': SVC(kernel='rbf', gamma='scale'),
        'SVM-Poly': SVC(kernel='poly', degree=3, gamma='scale')
    }

    # Dictionary to store results: keys = training size
    results = {}

    for t_size in train_sizes:
        # Split dataset (using stratification)
        X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=t_size,
                                                            stratify=y,
random_state=random_state)
        # Standardize using training set only
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        # Apply dimensionality reduction if needed
        if dr_choice == 'NoDR':
            X_train_trans, X_test_trans = X_train, X_test
        elif dr_choice == 'PCA':
            dr_model = PCA(n_components=2)
            X_train_trans = dr_model.fit_transform(X_train)
            X_test_trans = dr_model.transform(X_test)
        elif dr_choice == 'LDA':
            # For LDA, we need the labels when fitting
            dr_model = LinearDiscriminantAnalysis(n_components=2)
```

```
            X_train_trans = dr_model.fit_transform(X_train, y_train)
            X_test_trans = dr_model.transform(X_test)
        else:
            raise ValueError("dr_choice must be one of: 'NoDR', 'PCA',
'LDA'.")

        # For this training size, store results for each classifier
        results[t_size] = {}
        for clf_name, clf in classifiers.items():
            # Use clone so that each run is independent
            clf_instance = clone(clf)
            clf_instance.fit(X_train_trans, y_train)
            train_acc = clf_instance.score(X_train_trans, y_train)
            test_acc = clf_instance.score(X_test_trans, y_test)

            # Store classifier and predictions for later analysis (if
needed)
            results[t_size][clf_name] = {
                'train_acc': train_acc,
                'test_acc': test_acc,
                'clf': clf_instance,
                'X_train_trans': X_train_trans,
                'y_train': y_train,
                'X_test_trans': X_test_trans,
                'y_test': y_test
            }
    return results

# ------------------------------
# Function to plot accuracy vs. training size
# ------------------------------
def plot_accuracies(results_dict, dataset_name, metric='test_acc'):
    """
    Plot accuracy (training or test) vs. training size.

    Parameters:
      results_dict: nested dictionary with structure:
                    results_dict[dr_method][training_size][classifier]
[metric]
      dataset_name: string for dataset title
      metric      : 'train_acc' or 'test_acc'
    """
    # Extract training sizes from one of the DR method dictionaries.
    train_sizes = sorted(list(next(iter(results_dict.values())).keys()))
    plt.figure(figsize=(8, 5))
    for dr_method in results_dict:
```

```
        # Use the first training size to get classifier names
        first_ts = train_sizes[0]
        for clf_name in results_dict[dr_method][first_ts]:
            # For each classifier, gather the accuracy for each training
size.
            accs = [results_dict[dr_method][ts][clf_name][metric] for ts in
train_sizes]
            plt.plot(np.array(train_sizes)*100, accs, marker='o',
linestyle='--',
                     label=f'{dr_method} - {clf_name}')
    plt.xlabel('Training Size (%)')
    plt.ylabel('Accuracy')
    metric_title = "Training" if metric == 'train_acc' else "Test"
    plt.title(f'{dataset_name}: {metric_title} Accuracy vs Training Size')
    plt.grid(True)
    plt.legend()
    plt.show()

# --------------------------------
# Function to compute per-class accuracy given predictions and true labels
# --------------------------------
def compute_per_class_accuracy(y_true, y_pred):
    """
    Computes per-class accuracy from a confusion matrix.

    Returns:
      A dictionary with class labels as keys and per-class accuracy as
values.
    """
    cm = confusion_matrix(y_true, y_pred)
    per_class_acc = {}
    for i in range(cm.shape[0]):
        if cm[i].sum() > 0:
            per_class_acc[i] = cm[i, i] / cm[i].sum()
        else:
            per_class_acc[i] = np.nan
    return per_class_acc

# --------------------------------
# Main function
# --------------------------------
def main():
    # Define training sizes as fractions
    train_sizes = [0.1, 0.2, 0.3, 0.4, 0.5]

    # Define DR method choices (for our experiments we want: NoDR, PCA, and
```

```
LDA)
    dr_methods = ['NoDR', 'PCA', 'LDA']

    # Containers to store results for each dataset and DR method
    # results_all[dataset][dr_method][training_size][classifier]
    results_all = {'Iris': {}, 'Indian Pines': {}}

    # ================================
    #         IRIS DATASET
    # ================================
    print("Processing Iris dataset...")
    iris = load_iris()
    X_iris = iris.data
    y_iris = iris.target

    # For Iris, run experiments for each DR option
    results_all['Iris'] = {}
    for dr in dr_methods:
        print(f"Running Iris experiment with DR = {dr}")
        results_all['Iris'][dr] = run_classification_experiment(X_iris,
y_iris, dr, train_sizes)

    # Plot training and test accuracies for Iris
    plot_accuracies(results_all['Iris'], "Iris", metric='train_acc')
    plot_accuracies(results_all['Iris'], "Iris", metric='test_acc')

    # ================================
    #     INDIAN PINES DATASET
    # ================================
    print("Processing Indian Pines dataset...")
    # Load Indian Pines hyperspectral data and ground truth labels
    indian_pines = scipy.io.loadmat("indianR.mat")
    indian_labels = scipy.io.loadmat("indian_gth.mat")

    # Print keys for reference (optional)
    print("Indian Pines keys:", indian_pines.keys())

    # Extract hyperspectral data and transpose so that each row is a sample
    X_pines = indian_pines['X'].T
    # Extract and flatten the ground truth labels
    y_pines = indian_labels['gth'].flatten()

    # Remove unclassified areas (zero labels)
    if X_pines.shape[0] == y_pines.shape[0]:
        mask = y_pines > 0
        X_pines = X_pines[mask]
```

```
        y_pines = y_pines[mask]
    else:
        print("Dimension mismatch! Check dataset preprocessing.")

    # Run experiments for Indian Pines for each DR option
    results_all['Indian Pines'] = {}
    for dr in dr_methods:
        print(f"Running Indian Pines experiment with DR = {dr}")
        results_all['Indian Pines'][dr] =
run_classification_experiment(X_pines, y_pines, dr, train_sizes)

    # Plot training and test accuracies for Indian Pines
    plot_accuracies(results_all['Indian Pines'], "Indian Pines",
metric='train_acc')
    plot_accuracies(results_all['Indian Pines'], "Indian Pines",
metric='test_acc')

    # ------------------------------------------
    # Tabulate classwise accuracies for Indian Pines DR methods at 30%
training size
    # (Case i: with dimensionality reduction: PCA and LDA)
    # ------------------------------------------
    print("\nClasswise classification accuracies for Indian Pines (30%
training) with DR:")
    dr_tabulation = {}
    for dr in ['PCA', 'LDA']:
        # Create a DataFrame to store per-class accuracies for each
classifier
        per_class_df = pd.DataFrame()
        for clf_name in results_all['Indian Pines'][dr][0.3]:
            # Get the stored info for the current classifier at training
size 30%
            info = results_all['Indian Pines'][dr][0.3][clf_name]
            y_true = info['y_test']
            y_pred = info['clf'].predict(info['X_test_trans'])
            class_acc = compute_per_class_accuracy(y_true, y_pred)
            # Convert dictionary to pandas Series (index = class labels)
            per_class_series = pd.Series(class_acc, name=clf_name)
            per_class_df = pd.concat([per_class_df, per_class_series],
axis=1)
        dr_tabulation[dr] = per_class_df
        print(f"\nDR Method: {dr}")
        print(per_class_df.round(3))

if __name__ == '__main__':
```

```
    main()
```