1. Introduction

While high-dimensional data can indeed carry more information, it also poses significant challenges for traditional machine learning models. High-dimensional datasets may lead to the "curse of dimensionality," where model accuracy decreases due to overfitting, as well as the Hughes phenomenon (Hughes, 1968), where too many features degrade performance. Additionally, the computational complexity grows exponentially with increasing dimensions, making training inefficient.

In this assignment, we applied various dimensionality reduction (DR) techniques, including Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA), to three datasets representing different types of data: low-dimensional structured data (Wine dataset), medium-complexity image data (simplified MNIST), and high-dimensional complex colour image data (CIFAR-10). The goal is to investigate how these dimensionality reduction methods impact classification performance across different data complexities, and to understand the trade-offs between retaining dimensionality and improving classification accuracy.

2. Methodology

Datasets

3 datasets were used in this assignment for representing various data types.

- **Wine Dataset**: Contains 13 chemical data and 3 categories of wine, 178 samples.
  Representing low-dimensional structured data.
  Suitable for testing basic linear classification models.
- **Simplified MNIST**: Contains 64(8*8) pixels gray level image data, 1797 samples.
  Representing medium-complexity image data.
  Used to test the dimensionality reduction effect of medium-dimensional data
- **CIFAR-10**: Contains 32*32 pixels RGB image data, 60,000 samples.
  Representing high-dimensional complex colour image data.
  Explore dimensionality reduction and classification challenges for complex data.

Preprocessing

Data loaded and normalized into the range [0,1] to ensure uniform feature scaling, which is important for the effective performance of most machine learning models. The datasets were split into training and test sets using an 80:20 ratio to ensure a robust evaluation of the classifiers' performance while maintaining sufficient training data.

Dimensionality Reduction and Features Extraction

The Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) methods were used to reduce data dimensionality across the datasets. The CIFAR-10 dataset, due to its high dimensionality and complex data structure, posed a unique challenge, and therefore, a pre-trained ResNet18 model was used for feature extraction. ResNet18 was selected because of its balanced complexity and ability to extract robust features from image data without requiring extensive computational resources.

For each dataset, the appropriate PCA parameters were chosen based on the variance retention criteria. LDA parameters were automatically determined as the minimum of (C-1) and the input dimension.

Classifiers

To evaluate the impact of dimensionality reduction on classification performance, three classifiers were selected based on their different strengths and applicability to various data types:

- **k-Nearest Neighbors (k-NN)**: A non-parametric classifier that makes predictions based on the closest training examples in the feature space. It is simple and effective for low-dimensional data, making it a good candidate for evaluating the effects of dimensionality reduction on structured and moderately complex datasets like Wine and simplified MNIST.
- **Logistic Regression**: A linear classifier that is efficient for linearly separable data. It is particularly suitable for datasets where the relationships between features can be effectively modeled as linear combinations. Logistic regression helps evaluate the linear separability of the data after dimensionality reduction, especially following LDA.
- **Mahalanobis Distance Classifier**: This classifier calculates the distance between points and class centers using the covariance matrix. It is sensitive to feature correlations and can handle complex feature spaces well, making it ideal for high-dimensional data and post-PCA/LDA transformation.

Each classifier was trained on the reduced-dimensional datasets, and their accuracy and performance were recorded and compared.

Experimental Procedure

The experimental process follows a system pipeline where different configurations are generated based on a predefined parameter dictionary. The pipeline handles data preprocessing and experiment execution for various DR methods and classifiers. After the classifier predictions, performance metrics including accuracy and time are recorded. The above process is repeated for each dataset and the results are aggregated for further analysis.

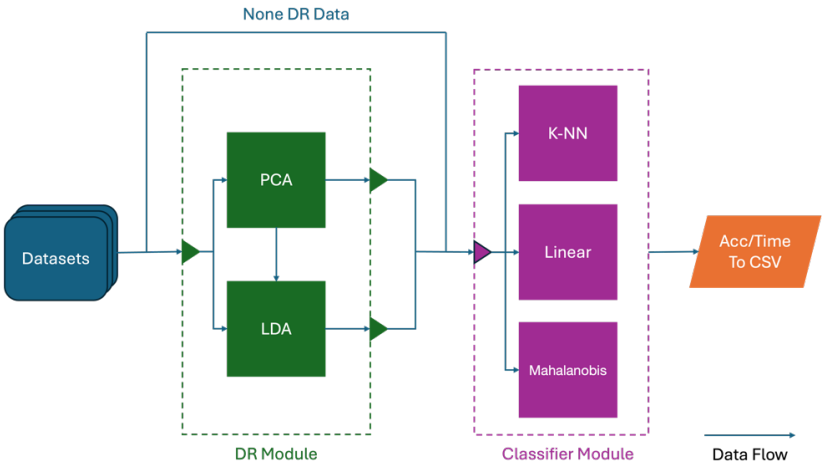

Figure 1Experiment Flow

3. Experimental Results

In this section, we present the experimental results that illustrate the impact of different dimensionality reduction techniques on classification accuracy and time consumption across the three datasets: Wine, Simplified MNIST, and CIFAR-10. Three sets of figures and tables summarize the key findings.

PCA Parameter Sensitivity

As shown in Figure 2 (see Appendix B for the detailed image) showing how varying PCA parameters (the number of retained components) impacts classification accuracy among datasets for the three classifiers. For each plot, the baseline (no dimensionality reduction) classification accuracy is marked for comparison.

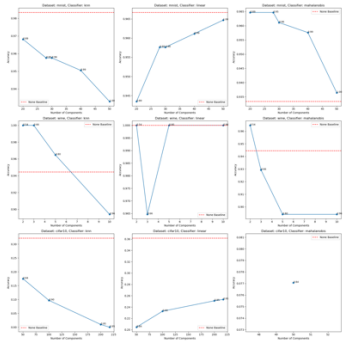

Figure 2: A brief overview of PCA sensitivity

LDA Comparison with PCA+LDA

Figure 3 displays three subplots, each representing a dataset. These plots compare the classification accuracy of applying PCA with varying parameters, using LDA-only performance as the baseline.

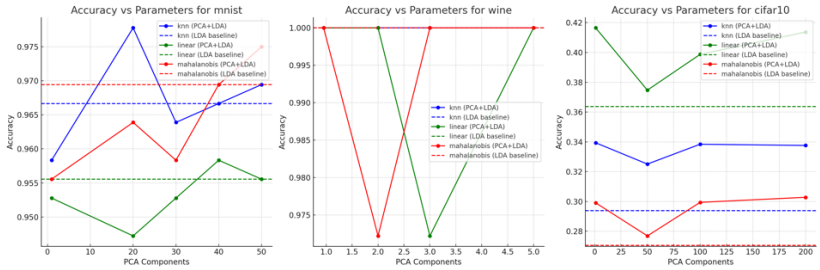

Figure 3: PCA+LDA v.s. LDA

CIFAR-10 Results: Comparison of Classifiers and Reduction Methods

Figure 4 (next page) provides a detailed comparison of different classifiers applied to the CIFAR-10 dataset, with and without dimensionality reduction. This figure consists of three subplots, each corresponding to one classifier, showing how PCA, LDA, and PCA+LDA perform relative to the baseline (no dimensionality reduction).

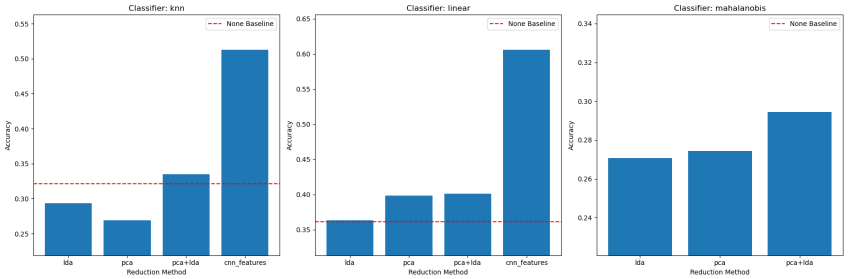

Figure 4: Accuracies on Cifar-10

Classification Time

The tables below present the classification speed (in FPS) before and after dimensionality reduction across all datasets and classifiers. The upper sub-row is FPS without DR applied. Lower sub-row is average FPS with DR applied. These results help illustrate the trade-offs between reduced dimensions and computational efficiency.

Table 1: Samples per Second before and after DR

	K-NN	Linear	Mahalanobis
Wine	11494	140	12987
	18867	136	19670
MNIST	1915	386	505
	8379	1194	5136
Cifar-10	2227	212	T.O.
	7692	5555	733

4. Discussion

This section analyzes the experimental results, focusing on the effects of different dimensionality reduction methods (PCA, LDA, PCA+LDA) and classifiers (k-NN, Logistic Regression, Mahalanobis Distance) on the three datasets. We will discuss key observations and provide explanations based on the behavior of the classifiers and the characteristics of the datasets.

PCA

k-NN and Mahalanobis Classifiers: For both k-NN and Mahalanobis classifiers, we observe that as the number of dimensions retained by PCA increases, classification accuracy decreases, especially on the MNIST and CIFAR-10 datasets. This can be explained by PCA's focus on maximizing global variance, which can cause the loss of important local structures that carries space information of image. These classifiers are sensitive to the distance between points, and PCA may distort these distances in high-dimensional spaces.

Logistic Regression: Unlike the other classifiers, logistic regression shows an improvement in performance as the number of PCA-retained dimensions increases. This is likely because logistic regression benefits from the greater linear separability introduced by PCA in higher dimensions, especially on the more complex CIFAR-10 dataset.

LDA

Improved Linear Separability: LDA significantly improves classification accuracy for logistic regression across all datasets, particularly the Wine dataset, where LDA is highly effective due to the linear separability of the classes. However, on CIFAR-10, LDA's performance is somewhat limited due to the dataset's complexity and inherent non-linear relationships.

k-NN and Mahalanobis Performance: LDA performs better than PCA for k-NN and Mahalanobis classifiers on datasets like Wine and Simplified MNIST. This is because LDA optimizes class separability, which aligns well with the distance-based decision rules used by these classifiers. However, on CIFAR-10, LDA alone does not fully capture the complexity needed for effective classification.

PCA+LDA

Combining Methods: The combination of PCA followed by LDA often outperforms PCA alone, especially for the Mahalanobis classifier on high-dimensional datasets like CIFAR-10. This suggests that reducing the dimensionality first through PCA and then applying LDA to maximize class separability provides a more stable and effective feature space for these classifiers. In addition, the reduction in dimensionality leads to faster matrix operations, which reduces the amount of computation exponentially.

5. Conclusion

For low-dimensional structured data, there may be a linear relationship between features, and there may be some redundancy in the feature dimensions. Therefore, when PCA is used to reduce the dimension to a certain range, it is possible to retain enough information for the classifier to classify while reducing the noise impact caused by redundant features (Martinez & Kak, 2001), especially for K-NN and Mahalanobis distance classifiers that are sensitive to high-dimensional noise.

For high-dimensional image data with complex structures, PCA/LDA or hybrid methods may destroy the local fine structural features of the image due to their global view and assumption of linear relationship between features. For complex abstract features, using deep learning, such as CNN, to extract features can obtain more detailed and comprehensive results.

In addition, it is undeniable that by reducing the dimensionality of the data, the calculation dimension can be effectively reduced, and the calculation process of some classifiers can be effectively accelerated, especially the Mahalanobis distance classifier that requires the calculation of the inverse matrix.

References:

Hughes, G. (1968). On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, 55-63.

Martinez, A., & Kak, A. (2001). PCA versus LDA. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 228 - 233.

Appendix A: Code

Also seen at:
https://github.com/AlvisWSY/EE6222_Assignment1/blob/main/code/Assignment.py

```python
import os
import time
import csv
import logging
import warnings
import numpy as np
from scipy.spatial.distance import cdist
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.exceptions import ConvergenceWarning
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from torch.utils.data import DataLoader
import torch
import torchvision
import torchvision.transforms as transforms

# Configure logging
logging.basicConfig(level=logging.INFO,
format='%(asctime)s %(levelname)s: %(message)s')

# Ignore convergence warnings
warnings.filterwarnings("ignore", category=ConvergenceWarning)

path = '/media/user/volume2/students/s124md209_01/WangShengyuan/6222/Assignment1/'
# Constants
RESULTS_CSV_PATH = os.path.join(path, 'result/results.csv')
DATA_DIR = os.path.join(path, 'data')

# Cache for loaded datasets
_cached_data = {}

def main():
    """
    Main function to run all experiments.
    """
```

```
    initialize_csv(RESULTS_CSV_PATH)
    config = {
        'mnist': {'n_components_list': [20, 30, 40, 50, 0.95]},
        'wine': {'n_components_list': [2, 3, 5, 0.95]},
        'cifar10': {'n_components_list': [50, 100, 200, 0.95]}
    }

    classifiers = ['knn', 'linear', 'mahalanobis']

    # Generate all experiments
    experiments = []
    for dataset_name, settings in config.items():
        experiments.extend(generate_experiments(dataset_name,
settings['n_components_list'], classifiers))

    # Include CNN feature extraction and classification for CIFAR-10
    if 'cifar10' in config:
        cnn_experiments = generate_cnn_experiments('cifar10', classifiers)
        experiments.extend(cnn_experiments)

    # Run all experiments
    for experiment in experiments:
        logging.info(f"Starting experiment: {experiment}")
        try:
            pipeline(experiment)
        except Exception as e:
            logging.error(f"Error in experiment {experiment}: {e}")
        logging.info(f"Finished experiment: {experiment}")

def generate_experiments(dataset_name, n_components_list, classifiers):
    """
    Generate experiment configurations for a given dataset.
    """
    experiments = []
    # Define reduction methods and their corresponding n_components
    reduction_methods = {
        'none': [None],
        'lda': [None],
        'pca': n_components_list,
        'pca+lda': n_components_list
    }
    for reduction_method, n_components_values in reduction_methods.items():
        for n_components in n_components_values:
            for classifier_name in classifiers:
```

```
                experiments.append({
                    'dataset_name': dataset_name,
                    'reduction_method': reduction_method,
                    'classifier_name': classifier_name,
                    'n_components': n_components
                })
    return experiments

def generate_cnn_experiments(dataset_name, classifiers):
    """
    Generate experiments using CNN features for traditional classifiers.
    """
    experiments = []
    for classifier_name in classifiers:
        experiments.append({
            'dataset_name': dataset_name,
            'reduction_method': 'cnn_features',
            'classifier_name': classifier_name,
            'n_components': None
        })
    return experiments

def initialize_csv(csv_file):
    """
    Initialize the CSV file with headers.
    """
    os.makedirs(os.path.dirname(csv_file), exist_ok=True)
    with open(csv_file, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([
            'Dataset',
            'Reduction Method',
            'Parameters',
            'Explained Variance',
            'Number of Components',
            'Classifier',
            'Accuracy',
            'Reduction Time per Sample',
            'Classification Time per Sample'
        ])
    logging.info(f"Initialized CSV file at {csv_file}")

def pipeline(params):
    """
```

```
    Pipeline to run the experiment with the given parameters.
    If CNN -> Extract features, else load and preprocess data.
    Apply dimensionality reduction -> Classify Data -> Write Results to CSV.
    """
    dataset_name = params['dataset_name']
    reduction_method = params['reduction_method']
    classifier_name = params['classifier_name']
    n_components = params['n_components']

    logging.info(f"Pipeline started for dataset: {dataset_name}, reduction:
{reduction_method}, classifier: {classifier_name}, n_components: {n_components}")

    # Handle CNN feature extraction
    if reduction_method == 'cnn_features':
        # Load data and extract CNN features
        X_train, X_test, y_train, y_test = extract_cnn_features(dataset_name)
        # Classify using traditional classifiers
        accuracy, classification_time = classify_data(
            classifier_name, X_train, y_train, X_test, y_test, dataset_name,
reduction_method
        )
        classification_avg_time = 'NA' if accuracy == 'NA' else
f"{classification_time / X_test.shape[0]:.6f}s"
        # Write results to CSV
        write_to_csv(RESULTS_CSV_PATH, [
            dataset_name,
            reduction_method,
            'N/A',
            'N/A',
            X_train.shape[1],
            classifier_name,
            accuracy,
            'N/A',
            classification_avg_time
        ])
        logging.info(f"Dataset: {dataset_name}, Reduction: {reduction_method}, "
                     f"Classifier: {classifier_name}, Accuracy: {accuracy}, "
                     f"Classification time per sample: {classification_avg_time}")
        return

    # Load and preprocess data
    X_train, X_test, y_train, y_test = load_and_preprocess_data(dataset_name)

    if reduction_method == 'none':
```

```
        # Classify without dimensionality reduction
        accuracy, total_time = classify_data(
            classifier_name, X_train, y_train, X_test, y_test, dataset_name,
reduction_method
        )
        avg_time_per_sample = 'NA' if accuracy == 'NA' else f"{total_time /
X_test.shape[0]:.6f}s"
        write_to_csv(RESULTS_CSV_PATH, [
            dataset_name,
            'none',
            'N/A',
            'N/A',
            X_train.shape[1],
            classifier_name,
            accuracy,
            'N/A',
            avg_time_per_sample
        ])
        logging.info(f"Dataset: {dataset_name}, Classifier: {classifier_name}, "
                     f"Accuracy: {accuracy}, Time per sample:
{avg_time_per_sample}")
        return

    # Apply dimensionality reduction
    X_train_reduced, X_test_reduced, explained_variance, reduction_avg_time =
apply_dimensionality_reduction(
        reduction_method, X_train, X_test, y_train, n_components
    )

    # After dimensionality reduction
    scaler = StandardScaler()
    X_train_reduced = scaler.fit_transform(X_train_reduced)
    X_test_reduced = scaler.transform(X_test_reduced)

    # Classify reduced data
    accuracy, classification_time = classify_data(
        classifier_name, X_train_reduced, y_train, X_test_reduced, y_test,
dataset_name, reduction_method
    )
    classification_avg_time = 'NA' if accuracy == 'NA' else f"{classification_time
/ X_test_reduced.shape[0]:.6f}s"

    # Write results to CSV
    write_to_csv(RESULTS_CSV_PATH, [
```

```
        dataset_name,
        reduction_method,
        n_components,
        explained_variance,
        X_train_reduced.shape[1],
        classifier_name,
        accuracy,
        f"{reduction_avg_time:.6f}s",
        classification_avg_time
    ])
    logging.info(f"Dataset: {dataset_name}, Reduction: {reduction_method}, "
                 f"Classifier: {classifier_name}, Accuracy: {accuracy}, "
                 f"Reduction time per sample: {reduction_avg_time:.6f}s, "
                 f"Classification time per sample: {classification_avg_time}")

def load_and_preprocess_data(dataset_name):
    """
    Load and preprocess data for the given dataset.
    """
    if dataset_name in _cached_data:
        return _cached_data[dataset_name]

    if dataset_name == 'mnist':
        X, y = load_mnist()
    elif dataset_name == 'wine':
        X, y = load_wine()
    elif dataset_name == 'cifar10':
        X, y = load_cifar10()
    else:
        raise ValueError("Unsupported dataset. Choose from 'mnist', 'wine',
'cifar10'.")

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )
    _cached_data[dataset_name] = (X_train, X_test, y_train, y_test)
    logging.info(f"Loaded and preprocessed dataset: {dataset_name}")
    return X_train, X_test, y_train, y_test

def load_mnist():
    """
    Load and preprocess the MNIST dataset.
    """
    from sklearn.datasets import load_digits
```

```
    digits = load_digits()
    X = digits.data.astype('float32') / 16.0  # Normalize to [0, 1]
    y = digits.target
    return X, y

def load_wine():
    """
    Load and preprocess the Wine dataset.
    """
    wine = datasets.load_wine()
    X = wine.data
    y = wine.target
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    return X, y

def load_cifar10():
    """
    Load and preprocess the CIFAR-10 dataset for traditional classifiers.
    """
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])
    dataset = torchvision.datasets.CIFAR10(
        root=DATA_DIR, train=True, download=True, transform=transform
    )
    dataloader = DataLoader(dataset, batch_size=len(dataset), shuffle=False)
    X, y = next(iter(dataloader))
    X = X.view(len(dataset), -1).numpy()
    y = y.numpy()
    X = X.astype('float32') / 2.0 + 0.5  # Normalize to [0, 1]
    return X, y

def extract_cnn_features(dataset_name):
    """
    Extract features from CNN for CIFAR-10 dataset.
    ResNet-18 model is used for feature extraction.
    """
    if dataset_name != 'cifar10':
        raise ValueError("CNN feature extraction is only implemented for CIFAR-
10.")

    # Define transformations
```

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Load datasets
trainset = torchvision.datasets.CIFAR10(
    root=DATA_DIR, train=True, download=True, transform=transform
)
testset = torchvision.datasets.CIFAR10(
    root=DATA_DIR, train=False, download=True, transform=transform
)

# Data loaders
trainloader = DataLoader(trainset, batch_size=64, shuffle=False)
testloader = DataLoader(testset, batch_size=64, shuffle=False)

# Define CNN model (using a pretrained model)
model = torchvision.models.resnet18(pretrained=True)
model = model.to('cuda:0' if torch.cuda.is_available() else 'cpu')
model.eval()

# Remove the final classification layer
features = torch.nn.Sequential(*list(model.children())[:-1])

# Function to extract features
def get_features(loader):
    X_features = []
    y_labels = []
    with torch.no_grad():
        for inputs, labels in loader:
            inputs = inputs.to('cuda' if torch.cuda.is_available() else 'cpu')
            outputs = features(inputs)
            outputs = outputs.view(outputs.size(0), -1)
            X_features.append(outputs.cpu().numpy())
            y_labels.append(labels.numpy())
    X_features = np.concatenate(X_features, axis=0)
    y_labels = np.concatenate(y_labels, axis=0)
    return X_features, y_labels

# Extract features
X_train, y_train = get_features(trainloader)
X_test, y_test = get_features(testloader)
```

```
    return X_train, X_test, y_train, y_test

def apply_dimensionality_reduction(method, X_train, X_test, y_train, n_components):
    """
    Apply the specified dimensionality reduction method to the data.
    """
    start_time = time.time()
    explained_variance = 'N/A'
    if method == 'pca':
        X_train_reduced, X_test_reduced, explained_variance = apply_pca(
            X_train, X_test, n_components
        )
    elif method == 'lda':
        X_train_reduced, X_test_reduced = apply_lda(X_train, X_test, y_train)
    elif method == 'pca+lda':
        X_train_pca, X_test_pca, explained_variance = apply_pca(X_train, X_test,
n_components)
        X_train_reduced, X_test_reduced = apply_lda(X_train_pca, X_test_pca,
y_train)
    else:
        raise ValueError("Unsupported reduction method. Choose from 'pca', 'lda',
'pca+lda'.")
    reduction_time = time.time() - start_time
    reduction_avg_time = reduction_time / (X_train.shape[0] + X_test.shape[0])
    logging.info(f"Applied {method} reduction in {reduction_time:.2f}s")
    return X_train_reduced, X_test_reduced, explained_variance, reduction_avg_time

def apply_pca(X_train, X_test, n_components):
    """
    Apply PCA to the data.
    """
    pca = PCA(n_components=n_components)
    X_train_reduced = pca.fit_transform(X_train)
    X_test_reduced = pca.transform(X_test)
    explained_variance = np.sum(pca.explained_variance_ratio_)
    logging.info(f"PCA explained variance: {explained_variance:.2f}")
    return X_train_reduced, X_test_reduced, explained_variance

def apply_lda(X_train, X_test, y_train):
    """
    Apply LDA to the data.
    """
    n_classes = len(np.unique(y_train))
    lda_n_components = min(n_classes - 1, X_train.shape[1])
```

```
    lda = LDA(n_components=lda_n_components)
    X_train_reduced = lda.fit_transform(X_train, y_train)
    X_test_reduced = lda.transform(X_test)
    logging.info(f"LDA reduced data to {lda_n_components} components")
    return X_train_reduced, X_test_reduced

def classify_data(classifier_name, X_train, y_train, X_test, y_test, dataset_name,
reduction_method):
    """
    Train and evaluate the specified classifier.
    """
    start_time = time.time()
    try:
        if classifier_name == 'knn':
            logging.info(f"Training KNN classifier on dataset: {dataset_name} with
reduction: {reduction_method}")
            classifier = KNeighborsClassifier(n_neighbors=3)
            classifier.fit(X_train, y_train)
            y_pred = classifier.predict(X_test)
        elif classifier_name == 'linear':
            logging.info(f"Training LogisticRegression on dataset: {dataset_name}
with reduction: {reduction_method}")
            # Adjust parameters for high-dimensional data
            if (dataset_name == 'cifar10') and (reduction_method == 'none' or
X_train.shape[1] > 500):
                logging.info("Using 'saga' solver for high-dimensional data")
                # Limit training data size to speed up
                max_samples = 5000
                if X_train.shape[0] > max_samples:
                    logging.info(f"Reducing training samples to {max_samples}")
                    X_train = X_train[:max_samples]
                    y_train = y_train[:max_samples]
                classifier = LogisticRegression(
                    max_iter=100,
                    solver='saga',
                    tol=1e-2,
                    multi_class='multinomial',
                    n_jobs=-1
                )
            else:
                logging.info("Using default 'lbfgs' solver")
                classifier = LogisticRegression(
                    max_iter=200,
                    solver='lbfgs',
```

```
                    multi_class='auto',
                    n_jobs=-1
                )
            classifier.fit(X_train, y_train)
            y_pred = classifier.predict(X_test)
        elif classifier_name == 'mahalanobis':
            logging.info(f"Training Mahalanobis classifier on dataset: {dataset_name} with reduction: {reduction_method}")
            # Limit sample size and dimensions for Mahalanobis classifier on CIFAR-10
            if dataset_name == 'cifar10':
                if X_train.shape[1] > 50:
                    logging.warning("Data dimension too high for Mahalanobis classifier on CIFAR-10. Skipping.")
                    return 'NA', 'NA'
                else:
                    max_samples = 5000  # Adjust based on your computational resources
                    if X_train.shape[0] > max_samples:
                        logging.warning(f"Reducing training samples to {max_samples} for Mahalanobis classifier.")
                        X_train = X_train[:max_samples]
                        y_train = y_train[:max_samples]
            y_pred = mahalanobis_classifier(X_train, y_train, X_test)
        else:
            raise ValueError("Unsupported classifier. Choose from 'knn', 'linear', 'mahalanobis'.")
        accuracy = accuracy_score(y_test, y_pred)
        total_time = time.time() - start_time
        logging.info(f"Classifier {classifier_name} achieved accuracy {accuracy:.4f} in {total_time:.2f}s")
        return accuracy, total_time
    except Exception as e:
        logging.error(f"Error in classification with {classifier_name}: {e}")
        return 'NA', 'NA'

def mahalanobis_classifier(X_train, y_train, X_test):
    """
    Classify using the Mahalanobis distance.
    """
    cov_matrix = np.cov(X_train, rowvar=False)
    inv_cov_matrix = np.linalg.pinv(cov_matrix)
    X_train_centered = X_train - np.mean(X_train, axis=0)
    X_test_centered = X_test - np.mean(X_train, axis=0)
```

```
    distances = cdist(
        X_test_centered, X_train_centered, metric='mahalanobis', VI=inv_cov_matrix
    )
    y_pred = y_train[np.argmin(distances, axis=1)]
    return y_pred

def write_to_csv(csv_file, row_data):
    """
    Write a row of data to the CSV file.
    """
    with open(csv_file, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(row_data)
    logging.debug(f"Wrote data to CSV: {row_data}")

if __name__ == '__main__':
    main()
```

Appendix B: Detailed Image for Figure 2