ECE661 Fall2024 HW10

Runlin Duan
duan92@purdue.edu

1 Theory Question

Understanding Overfitting

Overfitting occurs when a model learns the training data too well, capturing noise and fluctuations that do not generalize to new data. This results in high performance on the training set but poor performance on unseen data. For example, we achieved a training accuracy of 95 %, but the model only got a testing accuracy of 90 % . A potential reason for this is the overfitting of the model. Overfitting is a fundamental issue discussed in various contexts within machine learning because it significantly impacts the model's ability to function effectively in real-world settings. Techniques such as regularization, cross-validation, and pruning are employed to prevent overfitting, aiming to simplify the model and ensure it performs well on both seen and unseen data.

The Reparameterization Trick in Variational Autoencoders

In Variational Autoencoders (VAEs), the reparameterization trick is crucial for enabling gradient descent through stochastic nodes. The encoder in a VAE outputs parameters to a probability distribution, typically Gaussian, characterized by the mean (μ) and the logarithm of the variance ($\log \sigma^2$):

- **Deterministic Component:** The encoder outputs the mean (μ) and log-variance ($\log \sigma^2$) of the latent variables, describing the center and dispersion of the latent space points.
- **Stochastic Component:** Instead of sampling z directly from $\mathcal{N}(\mu, \sigma^2)$, the trick introduces an auxiliary noise variable ϵ from a standard normal distribution $\mathcal{N}(0, 1)$.
- **Reparameterization:** The latent variable z is computed as:

  $$z = \mu + \sigma \odot \epsilon$$

  where σ is the standard deviation, computed from the exponential of half the log-variance to ensure positivity. This formulation maintains randomness in z while allowing gradients to be backpropagated through μ and σ.

This method separates the randomness from the learnable parameters, allowing efficient gradient-based optimization of the variational lower bound, which is critical for the training of VAEs. The reparameterization trick thus enables VAEs to model complex distributions and generate new data points effectively.

2 Task 1: Face Recognition using PCA and LDA

2.1 PCA

1. **Data Loading:** Images are loaded and converted into a 1D vector using the `load_data` function.

$$\text{vectorized\_image}_i = \text{reshape}(\text{image}_i, m \times n)$$

2. **Normalization:** Each vectorized image is normalized to have unit magnitude.

$$\tilde{x}_i = \frac{x_i}{\|x_i\|}$$

3. **Centering Data:** Compute the mean vector $\overline{x}$ and subtract it from each image vector to center the data.

$$X = [\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_N]$$

$$X_{\text{centered}} = X - \overline{x}$$

where$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} \tilde{x}_i$ is the mean vector from all normalized image vectors.

4. **Covariance Matrix:** Calculate the covariance matrix C.

$$C = X_{\text{centered}} X_{\text{centered}}^T$$

5. **SVD Computation:** To reduce complexity using the computation trick mentioned in the reference, we apply SVD on the smaller matrix $X_{\text{centered}}^T X_{\text{centered}}$, yielding eigenvalues and eigenvectors.

$$X_{\text{centered}}^T X_{\text{centered}} v = \lambda v$$

6. **Eigenvectors of C:** Compute eigenvectors of C as:

$$u = X_{\text{centered}} v$$

7. **Normalization:** Normalize the eigenvectors to ensure unit length.

$$\hat{u} = \frac{u}{\|u\|}$$

8. **Dimension Reduction:** Select the top P eigenvectors corresponding to the largest eigenvalues.

$$W = [\hat{u}_1, \hat{u}_2, \ldots, \hat{u}_P]$$

9. **Projection:** Project the data onto the subspace defined by W_P.

$$y_i = W^T(x_i - \overline{x})$$

2.2 LDA

1. **Class Means:** Compute the overall mean m and class-specific means m_i for each class.
$$m_i = \frac{1}{|C_i|} \sum_{k \in C_i} x_k$$

2. **Between-Class Scatter:**
$$S_B = \sum_{i=1}^{C} n_i (m_i - m)(m_i - m)^T$$
where n_i is the number of samples in class i.

3. **Within-Class Scatter:**
$$S_W = \sum_{i=1}^{C} \sum_{k \in C_i} (x_k - m_i)(x_k - m_i)^T$$

4. **Fisher's Criterion:** Solve the eigenvalue problem for maximizing the discriminant:
$$J(w) = \frac{w^T S_B w}{w^T S_W w}, \quad S_B w = \lambda S_W w$$

5. **Applying Yu and Yang Algorithm:**
   (a) **Eigendecomposition of S_B:** Perform eigendecomposition on S_B to find the matrix V of eigenvectors, with $V^T S_B V = \Lambda$ where Λ is diagonal.
   (b) **Discard Near-Zero Eigenvalues:** Discard eigenvectors corresponding to near-zero eigenvalues in Λ, retaining the most significant directions in a reduced matrix Y.
   (c) **Matrix Z Construction:** Form $Z = Y D_B^{-1/2}$, normalizing S_B in the reduced dimensionality space: $Z^T S_B Z = I_{M \times M}$.
   (d) **Diagonalize $Z^T S_W Z$:** Perform eigendecomposition on $Z^T S_W Z$ to obtain U such that $U^T Z^T S_W Z U = D_W$.
   (e) **Discard Largest Eigenvalues of D_W:** Eliminate the eigenvectors associated with the largest eigenvalues of D_W, focusing on minimizing within-class scatter.
   (f) **Form Final LDA Eigenvectors:** Construct the LDA transformation matrix $W^T = \hat{U}^T Z^T$, maximizing the Fisher criterion with eigenvectors retained from U.

6. **Dimension Reduction:** Same as PCA, we select the top P eigenvectors as our main components.

2.3 Nearest Neighbor Classification

1. **Compute Distances:** For each test sample x_j, compute the Euclidean distance to all training samples.

$$d(x_j, x_i) = \|x_j - x_i\|$$

2. **Assign Label:** Assign the label of the nearest training sample:

$$\hat{y}_j = y_{\arg\min_i d(x_j, x_i)}$$

2.4 Visualization with UMAP

Embedding PCA and LDA Spaces

1. **UMAP Reduction:** Apply UMAP to reduce the PCA and LDA subspaces to 2 dimensions.

2. **Scatter Plots:** Create scatter plots for the training and testing data in both PCA and LDA spaces.

$$\text{UMAP}(X) = \{x_1, x_2, \ldots, x_N\} \subseteq R^2$$

PCA and LDA Visualization

- **Training Space:** Colors represent ground-truth labels.
- **Testing Space:** Colors represent predicted labels.

2.5 Results

2.5.1 Accuracy Plot

Here, we provide the result of the plot showing the classification accuracy as a function of the subspace dimensionality p for both PCA and LDA. The classification accuracy is defined as:

$$\text{accuracy} = \frac{\text{\# of test images correctly classified}}{\text{total \# of test images}}$$

We provide the accuracy of the PCA and LDA in the Fig 1

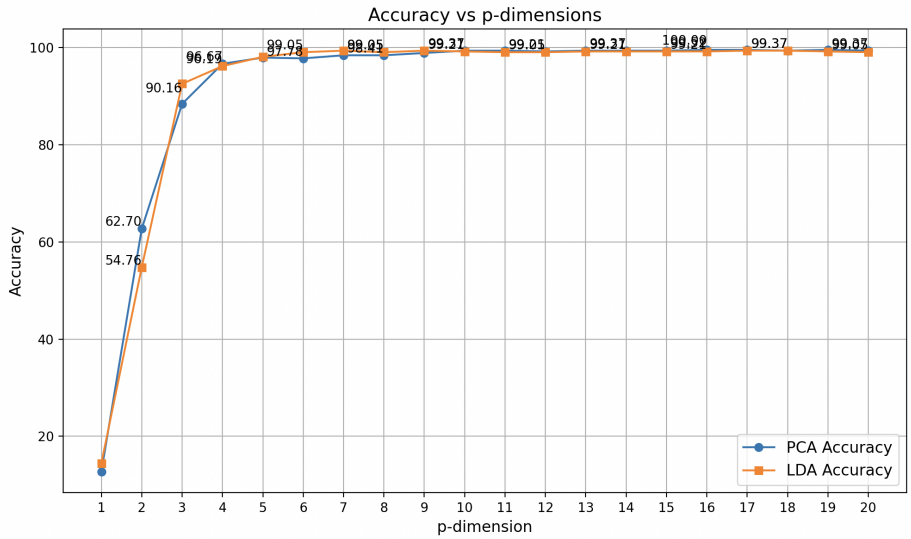

Figure 1: The accuracy of the PCA and LDA with different p-dimensions

2.5.2 UMAP Plot

We provide the UMAP plot with different p values, including $p = [3, 8, 16]$; we pick up these values for comparison with the p of autoencoder in the later section.

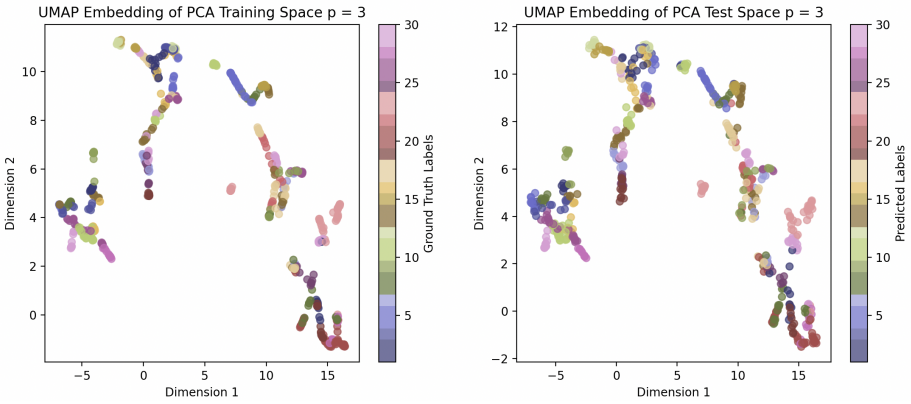

Figure 2: UMAP of PCA with the p=3

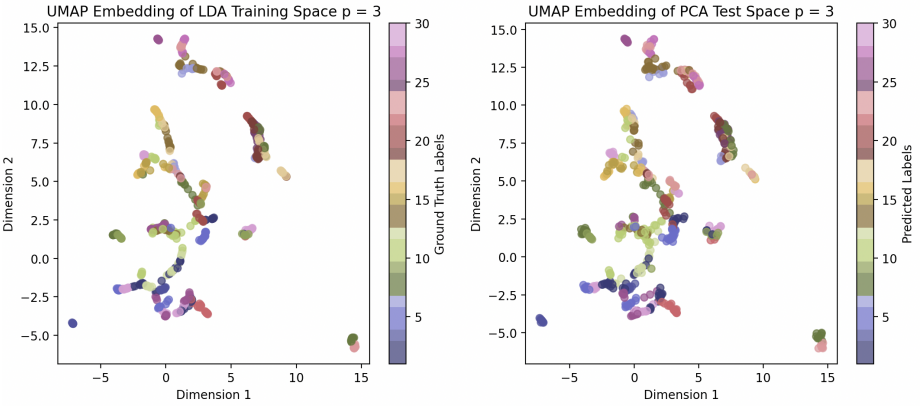

Figure 3: UMAP of LDA with the p=3

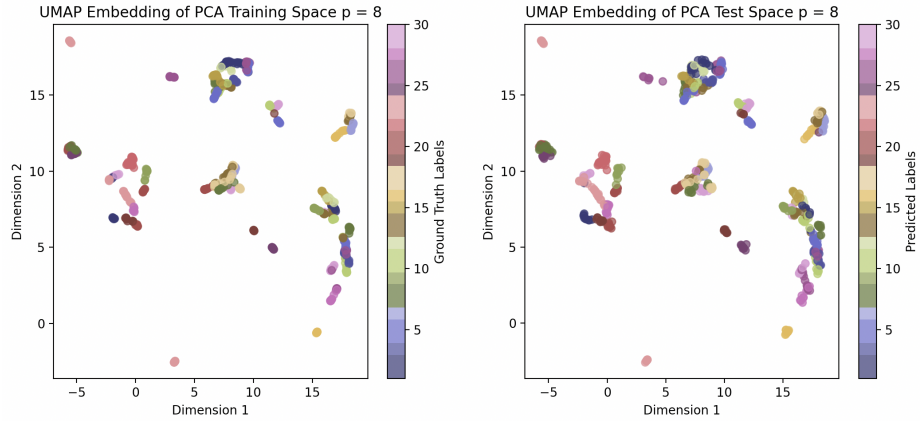

Figure 4: UMAP of PCA with the p=8

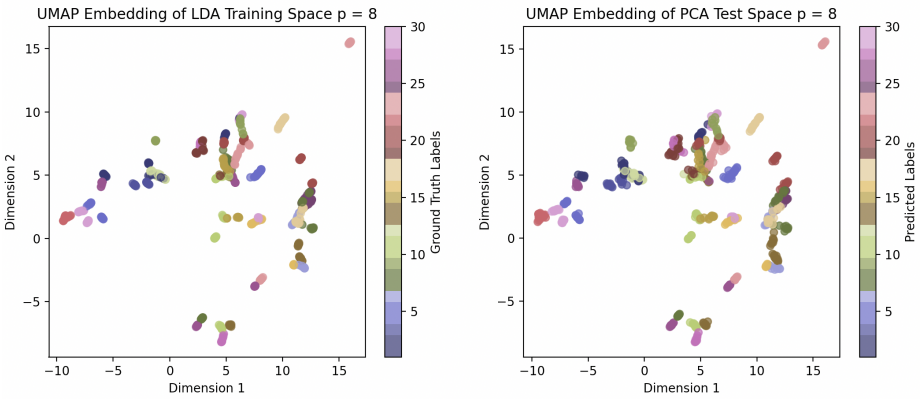

Figure 5: UMAP of LDA with the p=8

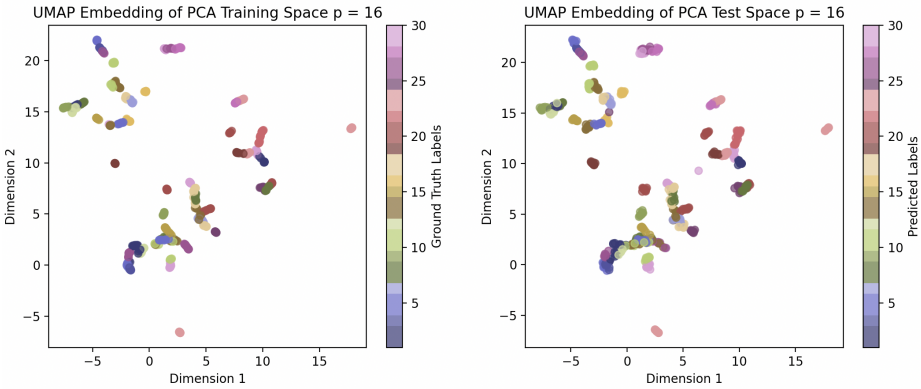

Figure 6: UMAP of PCA with the p=16

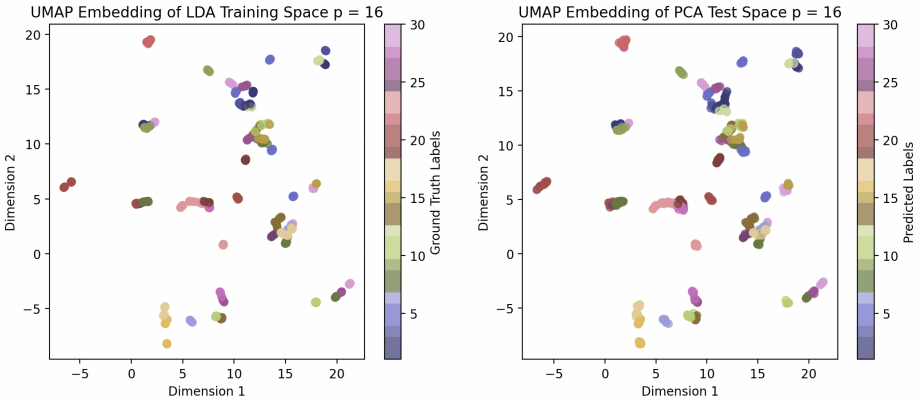

Figure 7: UMAP of LDA with the p=16

3 Task 2: Face Recognition using an Autoencoder

Autoencoders are neural networks used for unsupervised learning of compressed encodings from given data. They are typically used for dimensionality reduction. In this task, we use the pre-trained autoencoder to get the feature vectors for given images.

3.1 Implementation Details

Our implementation of an autoencoder is built using Pytorch and has been designed to handle different complexities of input data.

1. **Framework:** The model is implemented in Pytorch, utilizing its comprehensive support for building and training neural networks.
2. **Model Training:** The autoencoder has been trained on $P = \{3, 8, 16\}$.
3. **Low-dimensional Projection:** We will use the provided code to reduce the dimension of a given image. Each image is processed to predict a low-dimensional projection of length P, serving as a reduced representation of the original data.
4. **Nearest neighbor Classifier:** After encoding, the nearest neighbor classifier is employed to classify images based on the reduced representations provided by the autoencoder.
5. **Accuracy Measurement:** The classifier's accuracy is assessed on a test dataset to evaluate the effectiveness of the learned representations.

3.2 Results

3.2.1 Accuracy Plot

Here, we provide the result of the plot showing the classification accuracy as a function of the subspace dimensionality p for PCA, LDA, and autoencode in the same plot, in Fig 8.

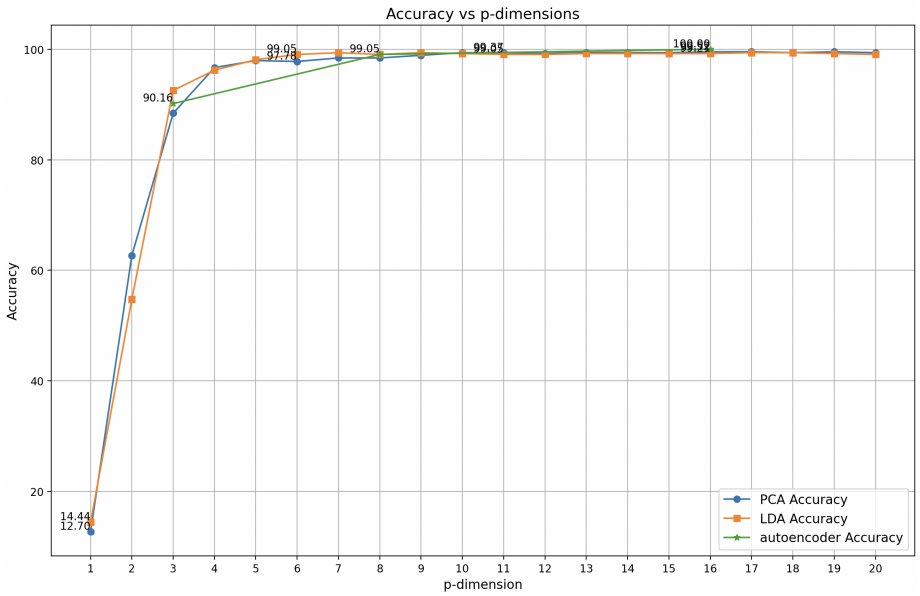

Figure 8: Classification accuracy as a function of the subspace dimensionality p for PCA, LDA, and autoencode

3.2.2 UMAP Plot

We provide the UMAP plot of the autoencoder with different p values, where $p = [3, 8, 16]$.

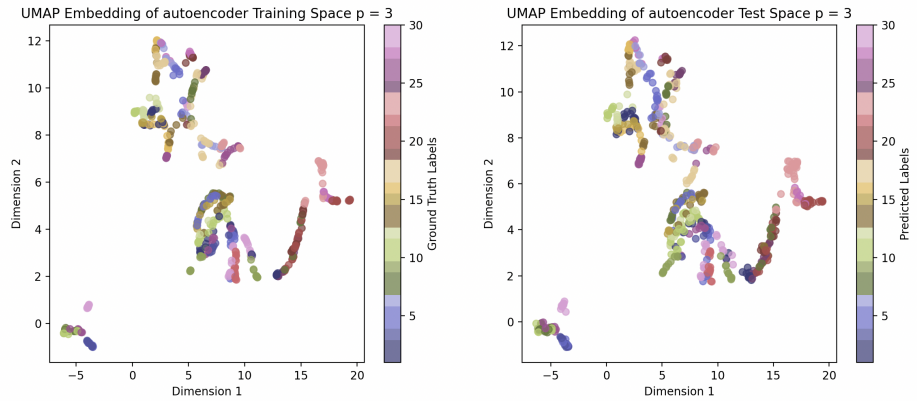

Figure 9: UMAP of Autoencoder with the p=3

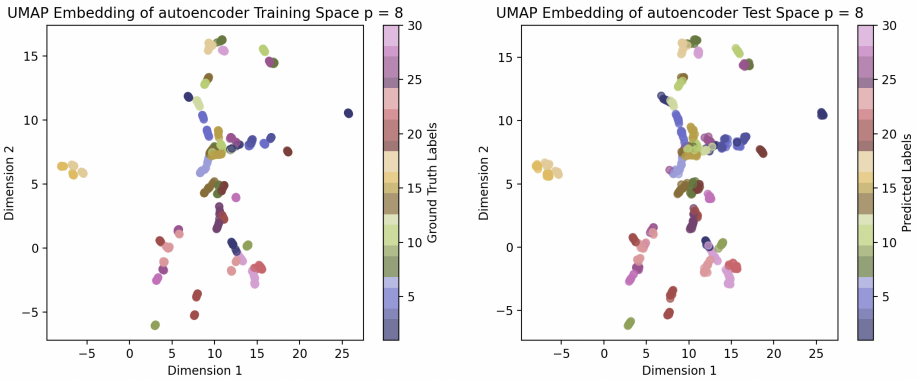

Figure 10: UMAP of Autoencoder with the p=8

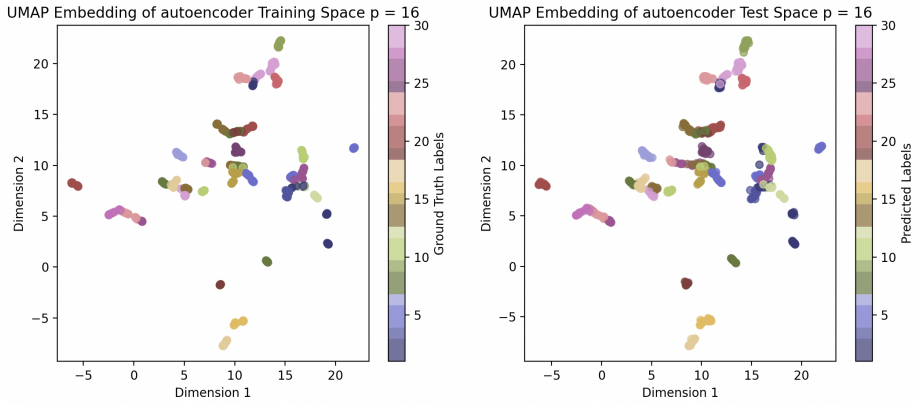

Figure 11: UMAP of Autoencoder with the p=16

3.3 Discissions for Taks1 and Taks2

3.3.1 Classification Accuracy

The results reveal the following key observations about the performance of PCA, LDA and Autoencoder on the face-detection task:

- At low-dimension, PCA and LDA exhibit low performance, with accuracies below 20%. At $p = 2$, PCA outperforms LDA, achieving an accuracy of 62.7% compared to LDA's 54.74%.
- All models reach peak accuracies of over 90 % around $p = 5$ and maintain stable performance until a slight decline begins at $p = 17$ for PCA and LDA.
- LDA achieves its highest accuracy of 100% at $p = 15$, Autoencoder achieves 100% at $p = 16$,while PCA reaches its maximum accuracy of 99.37% at $p = 13$.
- LDA converges faster than PCA and Autoencoder
- It seems that the Autoencoder demonstrates slightly better overall performance, particularly at higher dimensions.

These observations suggest that LDA may be better suited for this task, especially when high accuracy is required in lower-dimensional spaces.

3.3.2 UMAP Plot

The UMAP plots reveal important insights about the effects of different p-dimensions and feature extraction methods on the classification task. The key observations are summarized below:

- **Effect of Increasing p-Dimension:**
  - As the p-dimension increases, the categories in the UMAP plots become more discrete, and the distances between classes grow larger.
  - Higher p-dimensions result in feature vectors that are more separated, making it easier to classify samples.
  - At $p = 3$, samples from each class are close to each other, with noticeable overlap between classes. However, at $p = 16$, the overlap is significantly reduced, demonstrating improved class separability.
- **Comparison of Feature Extraction Methods:**
  - UMAP visualizations for PCA, LDA, and Autoencoders reveal distinct patterns, indicating that each method extracts unique feature representations.
  - Despite their differences, all three methods effectively capture the key information necessary for class separation.
  - The observed patterns in the UMAP plots confirm that each method preserves essential class-related information in its own way.
- **Conclusion:**
  - The UMAP plots validate that PCA, LDA, and Autoencoders are all reasonable and effective methods for feature extraction in this classification task.
  - Each method provides meaningful feature representations that support the accurate classification of the data.

4 Task 3: Object Detection using Cascaded AdaBoost Classifiers

Training Phase

Feature Extraction

- Utilize Haar-like features for image analysis, suitable for capturing edge, line, and textural information.

  The Haar filters are represented as follows:

$$\text{Horizontal: } \begin{bmatrix} -1 & -1 & 1 & 1 \end{bmatrix}$$

$$\text{Vertical: } \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

- **Integral Image Calculation**: Compute integral images to enable rapid feature calculation, facilitating the summation of pixel values within rectangular areas efficiently.
- **Feature Generation**: Apply Haar filters at every possible position and scale within the training images. Calculate each feature as:

$$\text{value(ABCD)} = I(D) - I(B) - I(C) + I(A)$$

where $I(x)$ represents the integral image value at point x, and A, B, C, D are the corners of the Haar filter.

Construct Weak Classifier

Our implementation of the AdaBoost starts by constructing weak classifiers.

1. **Weight Normalization**: The weights are normalized to ensure their sum equals one, maintaining weight distribution integrity across the dataset.
2. **Initialization**: Variables are initialized to track the minimum error and the best classifier parameters (feature and threshold).
3. **Feature Iteration**: For each feature:
   (a) **Sorting**: Data is sorted by feature values to facilitate cumulative weight calculations.
   (b) **Error Calculation**: We calculate the total weights of positive (T_p) and negative (T_n) labels. Errors are computed as we update the cumulative weights S_p and S_n:

$$\text{Error}_1 = S_n + (T_p - S_p) \quad \text{and} \quad \text{Error}_2 = S_p + (T_n - S_n)$$

   where Error_1 assumes samples below the threshold are negative, and Error_2 assumes they are positive.
   (c) **Threshold Determination**: If the feature value changes, we compute the threshold as the midpoint between consecutive differing feature values and update the error calculations.
4. **Classifier Selection**: The classifier that achieves the lowest error is selected, specifying the feature, threshold, and classification rule (above or below the threshold).

Cascade Construction

The cascade function begins by distributing initial weights equally across all samples. The cascade aims to maximize the True Positive Rate (TPR) while minimizing the False Positive Rate (FPR) to achieve specified thresholds.

1. **Initialization**: Start with equal weights for all training samples, combining positive and negative features and labels into a single dataset.
2. **Classifier Training**: For each classifier in the cascade, up to a predefined maximum:
   - Train the weak classifier on the current dataset using weighted instances.
   - Calculate α for the classifier, where ϵ refers to the error of the classifer: and α is defined as:
   $$\alpha = \ln\left(\frac{1}{\frac{\epsilon}{1-\epsilon}}\right)$$
   - Update the weights for the next iteration, increasing the weights of misclassified samples.
3. **Cascade Evaluation**: Aggregate the predictions from all classifiers, weighted by their respective α values. Adjust the classification threshold to the minimum score among all correctly classified positive samples:
   $$\text{Threshold} = \min(\text{Aggregated Predictions of Positive Samples})$$
4. **Performance Metrics**: Evaluate the TPR and FPR after adding each classifier:
   $$\text{TPR} = \frac{\text{Number of Correctly Classified Positives}}{\text{Total Positives}}$$
   $$\text{FPR} = \frac{\text{Number of Incorrectly Classified Negatives}}{\text{Total Negatives}}$$
   If the cascade meets the required TPR and FPR thresholds, terminate the cascade early.
5. **Refinement**: Focus subsequent training on challenging-to-classify negative examples by removing those that are correctly classified above the threshold.

Staging Cascade Classifiers

The cascade configuration is designed to optimize performance through precise parameters:

- **Maximum Stages**: Set to 10 to balance refinement and overfitting.
- **Target True Positive Rate (TPR)**: Set at 0.99 to ensure accurate positive classifications.
- **Target False Positive Rate (FPR)**: Maintained at 0.50 to limit incorrect classifications.
- **Maximum Weak Classifiers per Stage**: Limited to 50 to manage complexity and processing demands efficiently.

4.1 Results

We provide the false positive rate for the training process in the given Fig 12. The false positive and false negative rates over stages during the testing is provided in Fig. 13

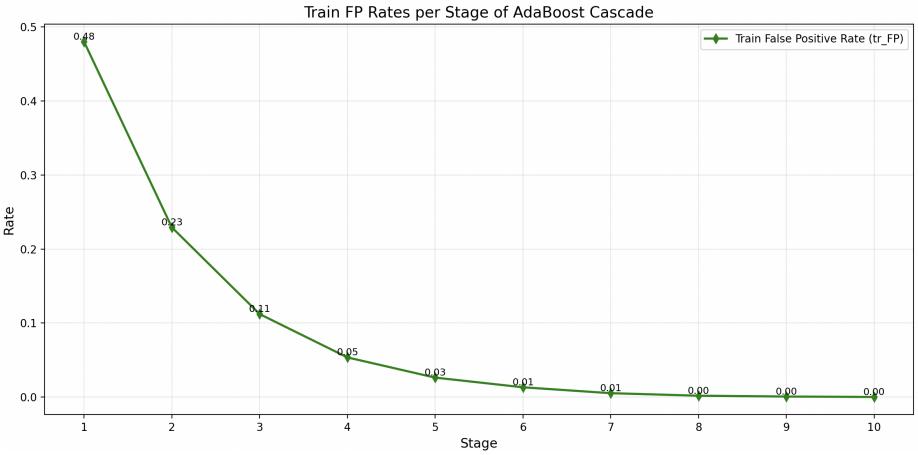

Figure 12: Train FP rates per Stage of AdaBoost Cascade

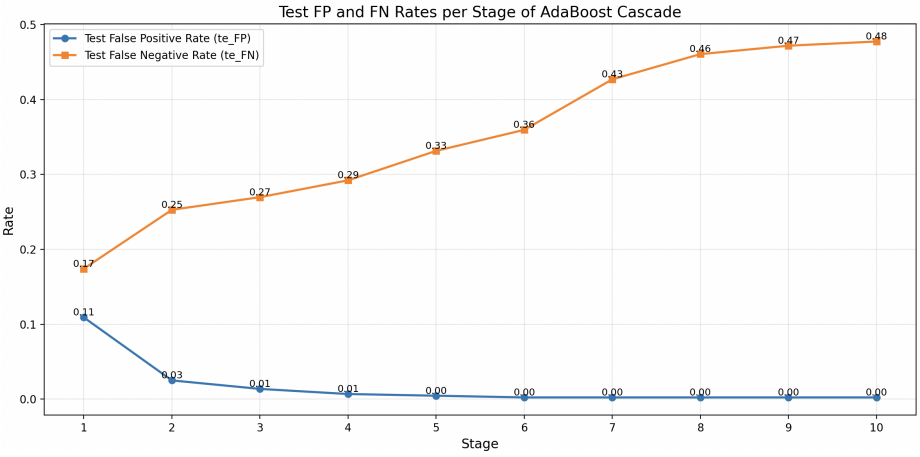

Figure 13: Test FP rates per Stage of AdaBoost Cascade

4.2 Discissions

The AdaBoost cascade demonstrates a clear trend of decreasing false positive rates (FPR) during training, with the training FPR sharply reducing from 48% at the first stage to 0% by the tenth stage. This decline signifies robust learning

and optimization capabilities of the model. Conversely, the false negative rates (FNR) increase from 17.4% to 47.7% over the same stages, indicating a trade-off between decreasing FPR and increasing FNR (Figure **??**).

As the training stage increases, the FPR rate drops down to zero at stage 8; during the testing, we achieve earlier zero FPR at stage 5. During the testing, note that we continue removing the correctly classified sample while the stage increases.

The result achieves the goal of minimizing the FPR while as a trade of increased FNR.

5 Code

```
1
2  #Task 1 2
3
4
5  import os
6  import cv2
7  import numpy as np
8  # from scipy.linalg import eigh
9  import matplotlib.pyplot as plt
10 import umap
11 from matplotlib import cm
12 import matplotlib.colors as mcolors
13 import autoencoder
14
15 def load_data(folder_type):
16
17     if folder_type not in ["train", "test"]:
18         raise ValueError("folder_type must be 'train' or
           ↪  'test'")
19
20     folder_path = f"facerecognition/{folder_type}"
21     if not os.path.exists(folder_path):
22         raise FileNotFoundError(f"The folder '{folder_path}'
           ↪  does not exist.")
23     images = []
24     labels = []
25     for file_name in os.listdir(folder_path):
26         if file_name.endswith(".png"):
27             # Extract label from file name
28             label = int(file_name.split("_")[0])
29             # print('label',label)
30             labels.append(label)
31             # Load the image
```

```
            image_path = os.path.join(folder_path, file_name)
            image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
            image = image.reshape(1, -1)[0]

            if image is None:
                raise ValueError(f"Image at '{image_path}' could not be loaded.")
            images.append(image)
    images_array = np.array(images, dtype=np.uint8)
    labels_array = np.array(labels, dtype=np.int32)

    return images_array, labels_array

def normalization(x):
    # Compute the L2 norm of each row. Keepdims=True makes the output shape (n_rows, 1)
    norms = np.linalg.norm(x, axis=1, keepdims=True)
    norms[norms == 0] = 1e-10
    normalized_x = x / norms

    return normalized_x

def PCA(X, num_components):
    # Transpose X so that columns are samples (features in rows)
    X = X.T
    # 1. Center the data
    mean_vector = np.mean(X, axis=1, keepdims=True)  # mean computed across rows (for each feature)
    X_centered = X - mean_vector
    # 2. Compute X^T * X (now correctly as X_centered is features in rows)
    XTX = np.dot(X_centered.T, X_centered)
    # 3. Apply SVD on the smaller matrix X^T * X
    U, S, VT = np.linalg.svd(XTX)
    # 4. Compute eigenvectors of the original covariance matrix XX^T
    # Since U contains the eigenvectors of X^T * X, we need to multiply by X_centered to get the eigenvectors of XX^T
    W_hat = np.dot(X_centered, VT)
    # print(W_hat)
    # W_hat = np.dot(X_centered, U)
    # 5. Normalize the eigenvectors to have unit length
    W_hat = normalization(W_hat)
    # W_hat / np.linalg.norm(W_hat, axis=0)
    # 6. Select the top 'num_components' eigenvectors
    principal_components = W_hat[:, :num_components]
```

```
72
73      return principal_components, mean_vector
74
75  def LDA(data, labels, num_components):
76      # Ensure labels is a 1D array
77      labels = labels.flatten()
78      # print(labels)
79      # Calculate the mean of each class
80      class_labels = np.unique(labels)
81
82      # print(class_labels)
83      mean_cl = np.array([np.mean(data[labels == cl], axis=0) for
        ↪  cl in class_labels])
84      mean_cl = mean_cl.T
85
86      # print('mean_cl',mean_cl)
87      # print('mean_cl,shape',mean_cl.shape)
88
89
90      X = data.T
91      # 1. Center the data
92      overall_mean_vec = np.mean(mean_cl, axis=1, keepdims=True)
93      # overall_mean_vec = np.mean(X, axis=1, keepdims=True)
94
95      Sw_dumy = X-overall_mean_vec
96      mean_i = mean_cl-overall_mean_vec
97      # print('mean_i',mean_i)
98      # Apply computation tricks
99      S_B = np.dot(mean_i.T,mean_i)
100     # Singular Value Decomposition of the between-class scatter
        ↪  matrix S_B
101     U_B, Sigma_B, V_B_T = np.linalg.svd(S_B)
102     # print(Sigma_B.shape)
103
104     d = (1/np.sqrt(Sigma_B))[:-2]
105     D_b = np.diag(d)
106
107     Y_dumy = np.dot(mean_i, V_B_T)
108     # Normalize the eigenvectors to have unit length
109     Y = normalization(Y_dumy)[:, :-2]
110     Z = np.dot(Y,D_b)
111     zt_sw = np.dot(np.transpose(Z), Sw_dumy)
112     # apply trick
113     S_W = np.dot(zt_sw, zt_sw.T)
114     # print('S_W',S_W.shape)
115     U_B, Sigma_B, V_B_T = np.linalg.svd(S_W)
```

```
116
117     # np.linalg.svd(sw)
118
119     # multiply by X to get eigenvecs of XX'
120     W = np.dot(Z, V_B_T)
121     W = normalization(W)
122     # retain first p eigen vecs
123     W_p = W[:, :num_components]
124
125     return W_p
126
127 def NN_classifier(train_space, test_space, train_labels,
    ↪  test_labels):
128
129     # Number of test samples
130     num_test_samples = test_space.shape[1]
131
132     # Placeholder for predicted labels
133     predicted_labels = np.zeros(num_test_samples,
        ↪  dtype=train_labels.dtype)
134
135     # Iterate over each test sample
136     for i in range(num_test_samples):
137         # Compute L2 distance from the i-th test sample to all
            ↪  training samples
138         distances = np.linalg.norm(train_space - test_space[:,
            ↪  i].reshape(-1, 1), axis=0)
139
140         # Find the index of the closest training sample
141         nearest_index = np.argmin(distances)
142
143         # Assign the label of the nearest training sample
144         predicted_labels[i] = train_labels[nearest_index]
145
146     # Calculate accuracy
147     correct_predictions = np.sum(predicted_labels ==
        ↪  test_labels)
148     accuracy = (correct_predictions / num_test_samples) * 100
149
150     return accuracy,predicted_labels
151
152 def PCA_LDA_Auto_plot():
153     train_data, train_labels = load_data("train")
154     test_data, test_labels = load_data("test")
155
156     print('train_labels',train_labels.shape)
```

```
157
158     train_data_n = normalization(train_data)
159     test_data_n = normalization(test_data)
160
161     # num_components = 5
162
163     accuracy_PCA = []
164     accuracy_LDA = []
165
166     for i in range(1,21):
167         num_components = i
168
169         # print('run pca')
170         PCA_pc, mean_vector = PCA(train_data_n, num_components)
171         LDA_W_p = LDA(train_data_n,train_labels,num_components)
172
173
174         train_data_center = train_data.T - mean_vector
175         test_data_center = test_data.T -mean_vector
176
177         PCA_train_space = np.dot(PCA_pc.T, train_data_center)
178         LDA_train_space = np.dot(LDA_W_p.T, train_data_center)
179
180         # print('PCA_train_space.shape',PCA_train_space.shape)
181
182         PCA_test_space = np.dot(PCA_pc.T, test_data_center)
183         LDA_test_space = np.dot(LDA_W_p.T, test_data_center)
184
185         # print('PCA_test_space.shape',PCA_test_space.shape)
186
187         accuracy_P,predicted_labels_p =
            ↪  NN_classifier(PCA_train_space, PCA_test_space,
            ↪  train_labels, test_labels)
188         accuracy_L,predicted_labels_p =
            ↪  NN_classifier(LDA_train_space, LDA_test_space,
            ↪  train_labels, test_labels)
189
190         accuracy_PCA.append(accuracy_P)
191         accuracy_LDA.append(accuracy_L)
192
193
194     autoencoder_acc = []
195     for p in [3, 8, 16]:
196         train_data,train_label,test_data,test_label =
            ↪  autoencoder.autoencoder_features(p)
197         # print(train_data.shape)
```

```
198         accuracy_a,predicted_labels_a =
            ↪ NN_classifier(train_data.T, test_data.T,
            ↪ train_label, test_label)
199         autoencoder_acc.append(accuracy_a)
200
201     # print(autoencoder_acc)
202
203     # import matplotlib.pyplot as plt
204
205     # Assuming accuracy_PCA and accuracy_LDA are populated as in
        ↪ your code
206     num_components = list(range(1, 21))  # x-axis: 1 to 20
207
208     # Plot the accuracies
209     plt.figure(figsize=(10, 6))
210     plt.plot(num_components, accuracy_PCA, marker='o',
        ↪ label='PCA Accuracy')
211     plt.plot(num_components, accuracy_LDA, marker='s',
        ↪ label='LDA Accuracy')
212     # plt.plot([3, 8, 16], autoencoder_acc, marker='*',
        ↪ label='autoencoder Accuracy')
213
214
215     for i, (x, y) in enumerate(zip(num_components,
        ↪ accuracy_PCA)):
216         if (i-1) % 2 == 0:  # Add text for every 5th point
217             plt.text(x, y, f"{y:.2f}", fontsize=10, ha='right',
                ↪ va='bottom')
218     for i, (x, y) in enumerate(zip(num_components,
        ↪ accuracy_LDA)):
219         if (i-1) % 2 == 0:  # Add text for every 5th point
220             plt.text(x, y, f"{y:.2f}", fontsize=10, ha='right',
                ↪ va='bottom')
221
222
223     for x, y in zip([3, 8, 16], autoencoder_acc):
224         plt.text(x, y, f"{y:.2f}", fontsize=10, ha='right',
            ↪ va='bottom')
225
226     # Add labels, title, and legend
227     plt.xticks(ticks=range(1, 21, 1))  # Set x-axis ticks from 1
        ↪ to 20
228     plt.xlabel('p-dimension', fontsize=12)
229     plt.ylabel('Accuracy', fontsize=12)
230     plt.title('Accuracy vs p-dimensions', fontsize=14)
231     plt.legend(fontsize=12)
```

```
232     plt.grid(True)
233
234     # Show the plot
235     plt.tight_layout()
236     plt.show()
237
238 def auto_encoder_plot():
239
240     autoencoder_acc = []
241     for p in [3, 8, 16]:
242         train_data,train_label,test_data,test_label =
            ↪ autoencoder.autoencoder_features(p)
243         # print(train_data.shape)
244         accuracy_a,predicted_labels_a =
            ↪ NN_classifier(train_data.T, test_data.T,
            ↪ train_label, test_label)
245         autoencoder_acc.append(accuracy_a)
246
247     print(autoencoder_acc)
248
249     # import matplotlib.pyplot as plt
250
251     # Assuming accuracy_PCA and accuracy_LDA are populated as in
        ↪ your code
252     num_components = list(range(1, 21))  # x-axis: 1 to 20
253
254
255     # Plot the accuracies
256     plt.figure(figsize=(10, 6))
257     # plt.plot(num_components, accuracy_PCA, marker='o',
        ↪ label='PCA Accuracy')
258     # plt.plot(num_components, accuracy_LDA, marker='s',
        ↪ label='LDA Accuracy')
259     plt.plot([3, 8, 16], autoencoder_acc, marker='*', label='LDA
        ↪ Accuracy')
260
261     # Add labels, title, and legend
262     plt.xticks(ticks=range(1, 21, 1))  # Set x-axis ticks from 1
        ↪ to 20
263     plt.xlabel('Number of Components', fontsize=12)
264     plt.ylabel('Accuracy', fontsize=12)
265     plt.title('Accuracy vs Number of Components', fontsize=14)
266     plt.legend(fontsize=12)
267     plt.grid(True)
268
269     # Show the plot
```

```
    plt.tight_layout()
    plt.show()

def UMAP_PCA_LDA():
    train_data, train_labels = load_data("train")
    test_data, test_labels = load_data("test")

    train_data_n = normalization(train_data)
    test_data_n = normalization(test_data)

    accuracy_PCA = []
    accuracy_LDA = []

    # for i in range(1, 21):
    num_components = 16

    # Run PCA and LDA to get the spaces
    PCA_pc, mean_vector = PCA(train_data_n, num_components)
    LDA_W_p = LDA(train_data_n, train_labels, num_components)

    train_data_center = train_data.T - mean_vector
    test_data_center = test_data.T - mean_vector

    PCA_train_space = np.dot(PCA_pc.T, train_data_center)
    LDA_train_space = np.dot(LDA_W_p.T, train_data_center)

    PCA_test_space = np.dot(PCA_pc.T, test_data_center)
    LDA_test_space = np.dot(LDA_W_p.T, test_data_center)

    # Get prediction results using NN classifier
    accuracy_P, predicted_labels_PCA =
    ↪  NN_classifier(PCA_train_space, PCA_test_space,
    ↪  train_labels, test_labels)
    accuracy_L, predicted_labels_LDA =
    ↪  NN_classifier(LDA_train_space, LDA_test_space,
    ↪  train_labels, test_labels)

    accuracy_PCA.append(accuracy_P)
    accuracy_LDA.append(accuracy_L)

    # # p = [3,8,16]
    # p = 3
    # train_data,train_label,test_data,test_label =
    ↪  autoencoder.autoencoder_features(p)
```

```
    # accuracy_a,predicted_labels_a =
    ↪  NN_classifier(train_data.T, test_data.T, train_label,
    ↪  test_label)

    # Apply UMAP to reduce PCA and LDA spaces to 2D
    # if num_components == 5:  # Example: Apply UMAP only for
    ↪  p=5
    print('run umap')
    umap_model = umap.UMAP(n_components=2, random_state=42)
    print('PCA_train_space',PCA_train_space.shape)
    # UMAP on PCA spaces
    PCA_train_umap = umap_model.fit_transform(PCA_train_space.T)
    PCA_test_umap = umap_model.transform(PCA_test_space.T)

    print('PCA_train_umap',PCA_train_umap.shape)
    # UMAP on LDA spaces
    LDA_train_umap = umap_model.fit_transform(LDA_train_space.T)
    LDA_test_umap = umap_model.transform(LDA_test_space.T)

    # Define a colormap for labels 1 to 30
    num_labels = 30
    cmap = cm.get_cmap('tab20b', num_labels)  # Ensure the
    ↪  colormap has 30 colors
    norm = mcolors.Normalize(vmin=1, vmax=num_labels)  #
    ↪  Normalize for the range 1-30

    # Plot PCA UMAP embeddings
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    sc = plt.scatter(PCA_train_umap[:, 0], PCA_train_umap[:, 1],
    ↪  c=train_labels, cmap=cmap, norm=norm, alpha=0.7)
    plt.title(f"UMAP Embedding of PCA Training Space p =
    ↪  {num_components}")
    plt.xlabel("Dimension 1")
    plt.ylabel("Dimension 2")
    cb = plt.colorbar(sc, label="Ground Truth Labels")
    # cb.set_ticks(range(0, num_labels,5))  # Set ticks from 1
    ↪  to 30
    # cb.set_ticklabels(range(1, num_labels + 1))  # Label ticks
    ↪  from 1 to 30
    # plt.show()

    # plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 2)
    sc = plt.scatter(PCA_test_umap[:, 0], PCA_test_umap[:, 1],
    ↪  c=predicted_labels_PCA, cmap=cmap, norm=norm, alpha=0.7)
```

```
345     plt.title(f"UMAP Embedding of PCA Test Space p =
        ↪ {num_components}")
346     plt.xlabel("Dimension 1")
347     plt.ylabel("Dimension 2")
348     cb = plt.colorbar(sc, label="Predicted Labels")
349
350     plt.show()
351
352     plt.figure(figsize=(12, 6))
353     plt.subplot(1, 2, 1)
354     sc = plt.scatter(LDA_train_umap[:, 0], LDA_train_umap[:, 1],
        ↪ c=train_labels, cmap=cmap, norm=norm, alpha=0.7)
355     plt.title(f"UMAP Embedding of LDA Training Space p =
        ↪ {num_components}")
356     plt.xlabel("Dimension 1")
357     plt.ylabel("Dimension 2")
358     cb = plt.colorbar(sc, label="Ground Truth Labels")
359
360     plt.subplot(1, 2, 2)
361     sc = plt.scatter(LDA_test_umap[:, 0], LDA_test_umap[:, 1],
        ↪ c=predicted_labels_LDA, cmap=cmap, norm=norm, alpha=0.7)
362     plt.title(f"UMAP Embedding of PCA Test Space p =
        ↪ {num_components}")
363     plt.xlabel("Dimension 1")
364     plt.ylabel("Dimension 2")
365     cb = plt.colorbar(sc, label="Predicted Labels")
366
367     plt.show()
368
369 def UMAP_autoencoder():
370
371     # p = [3,8,16]
372     num_components = 16
373     train_data,train_label,test_data,test_label =
        ↪ autoencoder.autoencoder_features(num_components)
374     accuracy_a,predicted_labels_a = NN_classifier(train_data.T,
        ↪ test_data.T, train_label, test_label)
375
376     # Apply UMAP to reduce PCA and LDA spaces to 2D
377     # if num_components == 5:  # Example: Apply UMAP only for
        ↪ p=5
378     print('run umap')
379     umap_model = umap.UMAP(n_components=2, random_state=42)
380     print('train_data',train_data.shape)
381     # UMAP on PCA spaces
382     encoder_train_umap = umap_model.fit_transform(train_data)
```

```
    encoder_test_umap = umap_model.transform(test_data)

    # Define a colormap for labels 1 to 30
    num_labels = 30
    cmap = cm.get_cmap('tab20b', num_labels)  # Ensure the colormap has 30 colors
    norm = mcolors.Normalize(vmin=1, vmax=num_labels)  # Normalize for the range 1-30

    # Plot PCA UMAP embeddings
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    sc = plt.scatter(encoder_train_umap[:, 0], encoder_train_umap[:, 1], c=train_label, cmap=cmap, norm=norm, alpha=0.7)
    plt.title(f"UMAP Embedding of autoencoder Training Space p = {num_components}")
    plt.xlabel("Dimension 1")
    plt.ylabel("Dimension 2")
    cb = plt.colorbar(sc, label="Ground Truth Labels")

    plt.subplot(1, 2, 2)
    sc = plt.scatter(encoder_test_umap[:, 0], encoder_test_umap[:, 1], c=predicted_labels_a, cmap=cmap, norm=norm, alpha=0.7)
    plt.title(f"UMAP Embedding of autoencoder Test Space p = {num_components}")
    plt.xlabel("Dimension 1")
    plt.ylabel("Dimension 2")
    cb = plt.colorbar(sc, label="Predicted Labels")

    plt.show()

# UMAP_autoencoder()
# auto_encoder_plot()
PCA_LDA_Auto_plot()
# UMAP_PCA_LDA()



#TASK 3
import numpy as np
import cv2
import glob
import os
import matplotlib.pyplot as plt
```

```
421
422 def load_data(folder_type):
423     # Specify the base path for the folders
424     base_path = 'CarDetection'
425
426     # Construct full paths for the positive and negative image
    ↪  directories
427     pos_path = os.path.join(base_path, folder_type, 'positive')
428     neg_path = os.path.join(base_path, folder_type, 'negative')
429
430     # Function to load images from a directory
431     def load_images_from_folder(folder):
432         images = []
433         for filename in os.listdir(folder):
434             img_path = os.path.join(folder, filename)
435             # Read the image using cv2
436             img = cv2.imread(img_path, cv2.IMREAD_COLOR)
437             if img is not None:
438                 # Optionally, convert the image from BGR to RGB
439                 img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
440                 images.append(img)
441             else:
442                 print(f"Failed to load image at {img_path}")
443         return np.array(images)
444
445     # Load positive and negative images
446     image_array_pos = load_images_from_folder(pos_path)
447     image_array_neg = load_images_from_folder(neg_path)
448
449     # Generate label arrays
450     label_array_pos = np.ones(len(image_array_pos))
451     label_array_neg = np.zeros(len(image_array_neg))
452
453     return image_array_pos, image_array_neg, label_array_pos,
    ↪  label_array_neg
454
455
456 def integral_image(image):
457     """ Compute the integral image using a data type that can
    ↪  handle larger numbers to avoid overflow. """
458     # Ensure the data type is large enough to handle the sums
459     return np.cumsum(np.cumsum(image.astype(np.int64), axis=0),
    ↪  axis=1)
460
461 def haar_feature(ii, top_left, width, height, orientation):
462     x, y = top_left
```

```
463     try:
464         if orientation == 'horizontal':
465             mid = y + height // 2
466             if y + 2 * height > ii.shape[1]:
467                 return 0
468             white = ii[x + width, mid] - ii[x, mid]
469             black = ii[x + width, y + height] - ii[x, y +
                ↪ height]
470         else:
471             mid = x + width // 2
472             if x + 2 * width > ii.shape[0]:
473                 return 0
474             white = ii[mid, y + height] - ii[mid, y]
475             black = ii[x + width, y + height] - ii[x, y +
                ↪ height]
476 
477         feature = white - black
478         if feature > np.iinfo(np.int32).max:  # Check if feature
            ↪ value is too large
479             print("Feature computation overflow: ", feature)
480         return feature
481     except IndexError:
482         # print("IndexError with parameters:", x, y, width,
            ↪ height, orientation)
483         return 0
484 
485 def extract_features(images):
486     """ Extract Haar features from a set of images, each image
        ↪ is 20x40 pixels. """
487     features = []
488     for image in images:
489         ii = integral_image(image)
490         img_features = []
491         # Define feature sizes and positions here:
492         for width in range(1, 21, 4):  # Adjust for plausible
            ↪ widths
493             for height in range(1, 41, 4):  # Adjust for
                ↪ plausible heights
494                 for x in range(21 - width):  # Avoid going out
                    ↪ of x bounds
495                     for y in range(41 - height):  # Avoid going
                        ↪ out of y bounds
496                         # Horizontal feature, check bounds
                            ↪ inside the function
497                         if y + 2 * height <= 40:  # Adjusted
                            ↪ bounds for horizontal feature
```

```
                                feature_val = haar_feature(ii, (x,
                                ↪  y), width, height, 'horizontal')
                                img_features.append(feature_val)
                            # Vertical feature, check bounds inside
                            ↪  the function
                            if x + 2 * width <= 20:  # Adjusted
                            ↪  bounds for vertical feature
                                feature_val = haar_feature(ii, (x,
                                ↪  y), width, height, 'vertical')
                                img_features.append(feature_val)
        features.append(img_features)

    return np.array(features)

def weak_classifier(weights, feats, labels):
    # Normalize the weights
    weights /= np.sum(weights)

    # Initialize variables to track the best classifier
    min_error = np.inf
    best_weak_cl = None
    best_pred = None

    # Iterate over each feature
    n_samples, n_features = feats.shape
    for feature_index in range(n_features):

        # Sort features and related labels and weights
        sorted_indices = np.argsort(feats[:, feature_index])
        sorted_feats = feats[sorted_indices, feature_index]
        sorted_weights = weights[sorted_indices]
        sorted_labels = labels[sorted_indices]

        # Calculate total positive and negative weights
        Tp = np.sum(sorted_weights[sorted_labels == 1])
        Tn = np.sum(sorted_weights[sorted_labels == 0])
        print('running weak_classifier TP',Tp)
        # Initialize cumulative sums
        Sp = 0  # Cumulative sum of weights for positive labels
        Sn = 0  # Cumulative sum of weights for negative labels
        err1 = Tp  # Initial error if all are predicted as
        ↪  positive (threshold at -inf)
        err2 = Tn  # Initial error if all are predicted as
        ↪  negative (threshold at +inf)

        for i in range(n_samples - 1):
```

```
            if sorted_labels[i] == 1:
                Sp += sorted_weights[i]
            else:
                Sn += sorted_weights[i]

            # Compute errors for both polarities
            err1 = Sn + (Tp - Sp)  # Samples below threshold are negative
            err2 = Sp + (Tn - Sn)  # Samples below threshold are positive

            # Check if we can split at this point
            if sorted_feats[i] != sorted_feats[i + 1]:
                threshold = (sorted_feats[i] + sorted_feats[i + 1]) / 2.0
                if err1 < min_error:
                    min_error = err1
                    best_weak_cl = [feature_index, threshold, '>=']
                    best_pred = np.where(feats[:, feature_index] >= threshold, 1, 0)
                if err2 < min_error:
                    min_error = err2
                    best_weak_cl = [feature_index, threshold, '<']
                    best_pred = np.where(feats[:, feature_index] < threshold, 1, 0)

    return best_weak_cl, min_error, best_pred

def update_weights(weights, predictions, labels, alpha):
    """ Update weights of the training samples. """
    weights *= np.exp(alpha * (predictions != labels))
    return weights / weights.sum()

def cascade(train_pos_feature, train_neg_feature, label_pos, label_neg, req_tpr, req_fpr, max_num_weak_clf):
    n_pos = train_pos_feature.shape[0]
    n_neg = train_neg_feature.shape[0]

    features = np.vstack((train_pos_feature, train_neg_feature))
    labels = np.concatenate((label_pos, label_neg))
    weights = np.ones(n_pos + n_neg) / (n_pos + n_neg)
    classifiers = []

    print('running cascade')
```

```
576
577     for _ in range(max_num_weak_clf):
578         clf, error, predictions = weak_classifier(weights,
            ↪  features, labels)
579         alpha = 0.5 * np.log((1 - error) / max(error, 1e-10))
580         classifiers.append((clf, alpha))
581
582         weights = update_weights(weights, predictions, labels,
            ↪  alpha)
583
584         # Calculate cascade output
585         agg_predictions = np.zeros(n_pos + n_neg)
586         for clf, alpha in classifiers:
587             predictions = np.where(features[:,
                ↪  clf['feature_index']] < clf['threshold'], 1, -1)
                ↪  if clf['inequality'] == 'lt' else
                ↪  np.where(features[:, clf['feature_index']] >=
                ↪  clf['threshold'], 1, -1)
588             agg_predictions += alpha * predictions
589
590         # Determine classification threshold
591         threshold = np.min(agg_predictions[:n_pos])  # Min among
            ↪  positives
592
593         # Evaluate
594         classified_pos = (agg_predictions[:n_pos] >= threshold)
595         classified_neg = (agg_predictions[n_pos:] >= threshold)
596
597         TPR = np.mean(classified_pos)
598         FPR = np.mean(classified_neg)
599
600         print('TPR',TPR)
601         print('FPR',FPR)
602
603         if TPR >= req_tpr and FPR <= req_fpr:
604             break  # Early stopping
605
606     # Refinement of dataset
607     misclassified_neg_indices = np.where(agg_predictions[n_pos:]
        ↪  < threshold)[0]
608     refined_neg_features =
        ↪  train_neg_feature[misclassified_neg_indices]
609     refined_neg_labels = label_neg[misclassified_neg_indices]
610
611     return refined_neg_features, refined_neg_labels,
        ↪  classifiers, {'TPR': TPR, 'FPR': FPR}
```

```
612
613 def load_or_initialize_cascade(filename):
614     """ Attempt to load a previously saved cascade model, or
        ↪  initialize new structures if none exist. """
615     if os.path.exists(filename):
616         with np.load(filename) as data:
617             return list(data['classifiers']),
                ↪  list(data['alphas']), list(data['fprs'])
618     else:
619         return [], [], []
620
621 def save_cascade(filename, classifiers, alphas, fprs):
622     """ Save the cascade model to a compressed file. """
623     np.savez_compressed(filename, classifiers=classifiers,
        ↪  alphas=alphas, fprs=fprs)
624
625 def training_function(tr_pos_ft, tr_neg_ft, tr_pos_lbl,
    ↪  tr_neg_lbl, max_stages, req_tpr, req_fpr, max_num_weak_clf):
626     filename = 'classifier_stages.npz'
627     classifier_stages, weight_stages, false_positive_rates =
        ↪  load_or_initialize_cascade(filename)
628
629     if len(classifier_stages) >= max_stages:
630         print("Loaded pre-trained model with sufficient
            ↪  stages.")
631         return classifier_stages, weight_stages,
            ↪  false_positive_rates
632
633     current_neg_ft = tr_neg_ft
634     current_neg_lbl = tr_neg_lbl
635
636     while len(classifier_stages) < max_stages:
637         refined_neg_ft, refined_neg_lbl, stage_classifiers,
            ↪  metrics = cascade(
638             tr_pos_ft, current_neg_ft, tr_pos_lbl,
                ↪  current_neg_lbl, req_tpr, req_fpr,
                ↪  max_num_weak_clf
639         )
640
641         classifier_stages.append(stage_classifiers)
642         weight_stages.extend([clf[1] for clf in
            ↪  stage_classifiers])  # Assuming clf tuple contains
            ↪  (classifier, alpha)
643         false_positive_rates.append(metrics['FPR'])
644
```

```
        # Refine dataset by removing correctly classified negatives
        current_neg_ft = refined_neg_ft
        current_neg_lbl = refined_neg_lbl

        print('false_positive_rates',false_positive_rates)
        # Early stopping if FPR is already below required or no negatives left
        if metrics['FPR'] <= req_fpr or len(current_neg_ft) == 0:
            print("Early stopping: FPR threshold met or no negatives left.")
            break

    save_cascade(filename, classifier_stages, weight_stages, false_positive_rates)
    return classifier_stages, weight_stages, false_positive_rates

def load_trained_cascade(filename):
    """ Load the trained cascade model from a compressed file. """
    data = np.load(filename)
    return data['classifiers'], data['alphas'],data['fprs']

def apply_classifier(features, classifier):
    """ Apply a weak classifier to the given features. """
    feature_index, threshold, polarity = classifier
    if polarity == 'lt':
        return features[:, feature_index] < threshold
    else:
        return features[:, feature_index] >= threshold

def testing_function(trained_cascade, test_positive_features, test_negative_features):
     # = trained_cascade
    classifiers, alphas,_ = load_trained_cascade(trained_cascade)
    # Combine test features
    test_features = np.vstack((test_positive_features, test_negative_features))
    # Labels for computing metrics
    test_labels = np.hstack((np.ones(len(test_positive_features)), np.zeros(len(test_negative_features))))

```

```
    false_positive_rates = []
    false_negative_rates = []
    # Aggregate predictions from each stage
    stage_predictions = np.zeros(test_features.shape[0])

    for i, stage in enumerate(classifiers):
        stage_sum = np.zeros(test_features.shape[0])
        for clf, alpha in zip(stage, alphas[i]):
            predictions = apply_classifier(test_features, clf)
            stage_sum += alpha * (2 * predictions - 1)  # Convert boolean to {-1, 1}

        # Calculate the threshold for binary classification
        threshold = np.sum(alphas[i]) * 0.5
        stage_output = (stage_sum > threshold).astype(int)

        # Calculate FPR and FNR
        positive_test_predictions = stage_output[:len(test_positive_features)]
        negative_test_predictions = stage_output[len(test_positive_features):]

        FPR = np.mean(negative_test_predictions == 1)
        FNR = np.mean(positive_test_predictions == 0)

        false_positive_rates.append(FPR)
        false_negative_rates.append(FNR)
        # Update stage_predictions for final output
        stage_predictions = stage_output

    # Reporting results
    print("False Positive Rates by stage:", false_positive_rates)
    print("False Negative Rates by stage:", false_negative_rates)

    return false_positive_rates, false_negative_rates

def plot_adaboost_rates(te_FP_rates, te_FN_rates, tr_FP_rates):
    # Define stages with an increment of 1 starting from 1
    stages = list(range(1, len(te_FP_rates) + 1))

    # Plot te_FP and te_FN rates on the first plot
    plt.figure(figsize=(12, 6))
    plt.plot(stages, te_FP_rates, label='Test False Positive Rate (te_FP)', marker='o', linestyle='-', linewidth=2)
```

```
    plt.plot(stages, te_FN_rates, label='Test False Negative Rate (te_FN)', marker='s', linestyle='-', linewidth=2)

    # Annotate te_FP and te_FN rates
    for i, rate in enumerate(te_FP_rates):
        plt.text(stages[i], rate, f"{rate:.2f}", fontsize=9, ha='center', va='bottom')
    for i, rate in enumerate(te_FN_rates):
        plt.text(stages[i], rate, f"{rate:.2f}", fontsize=9, ha='center', va='bottom')

    # Configure x-axis with integer steps
    plt.xticks(stages)  # Ensure the x-axis only shows integers at each step

    # Add labels, title, legend, and grid
    plt.xlabel('Stage', fontsize=12)
    plt.ylabel('Rate', fontsize=12)
    plt.title('Test FP and FN Rates per Stage of AdaBoost Cascade', fontsize=14)
    plt.legend(fontsize=10)
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.show()

    # Plot tr_FP rates on a separate plot
    plt.figure(figsize=(12, 6))
    plt.plot(stages, tr_FP_rates, label='Train False Positive Rate (tr_FP)', marker='d', linestyle='-', linewidth=2, color='green')

    # Annotate tr_FP rates
    for i, rate in enumerate(tr_FP_rates):
        plt.text(stages[i], rate, f"{rate:.2f}", fontsize=9, ha='center', va='bottom')

    # Configure x-axis with integer steps
    plt.xticks(stages)  # Ensure the x-axis only shows integers at each step

    # Add labels, title, legend, and grid
    plt.xlabel('Stage', fontsize=12)
    plt.ylabel('Rate', fontsize=12)
    plt.title('Train FP Rates per Stage of AdaBoost Cascade', fontsize=14)
    plt.legend(fontsize=10)
```

```
755     plt.grid(alpha=0.3)
756     plt.tight_layout()
757     plt.show()
758 
759 def adaboost():
760 
761     train_pos, train_neg,tr_label_pos,tr_label_neg =
        ↪  load_data('train')
762     test_pos, test_neg,_,_ = load_data('test')
763 
764     # print(train_pos.shape)
765 
766     # train_pos_feature = extract_features(train_pos)
767     # train_neg_feature = extract_features(train_neg)
768     # test_pos_feature = extract_features(test_pos)
769     # test_neg_feature = extract_features(test_neg)
770 
771 
772     #
        ↪  np.save('./task3npy/train_pos_feature.npy',train_pos_feature)
773     #
        ↪  np.save('./task3npy/train_neg_feature.npy',train_neg_feature)
774     #
        ↪  np.save('./task3npy/test_neg_feature.npy',test_neg_feature)
775     #
        ↪  np.save('./task3npy/test_pos_feature.npy',test_pos_feature)
776 
777     # print('saved features')
778 
779     # np.save('train_pos_feature.npy',train_pos_feature)
780     train_pos_feature =
        ↪  np.load('./task3npy/train_pos_feature.npy')
781     train_neg_feature =
        ↪  np.load('./task3npy/train_neg_feature.npy')
782     test_neg_feature =
        ↪  np.load('./task3npy/test_neg_feature.npy')
783     test_pos_feature =
        ↪  np.load('./task3npy/test_pos_feature.npy')
784 
785     # 710*14160
786     print('loaded features')
787 
788     train_pos_label = np.ones((train_pos.shape[0]))
789     train_neg_label = np.zeros((train_neg.shape[0]))
790     test_pos_label = np.ones((test_pos.shape[0]))
791     test_neg_label = np.zeros((test_neg.shape[0]))
```

```
792
793     max_stages = 20
794     # Required true positive rate (TPR) at each stage
795     target_tpr = 0.99
796     # Required false positive rate (FPR) at each stage
797     target_fpr = 0.50
798     # Maximum number of weak classifiers per stage
799     max_num_wclf = 50
800
801     print('start training')
802     classifier_stages, weight_stages, false_positive_rates =
        ↪  training_function(train_pos_feature, train_neg_feature,
        ↪  tr_label_pos, tr_label_neg, max_stages, target_tpr,
        ↪  target_fpr, max_num_wclf)
803
804     filename = 'classifier_stages.npz'
805     te_FP_rates, te_FN_rates = testing_function(filename,
        ↪  test_pos_feature, test_neg_feature)
806
807     plot_adaboost_rates(te_FP_rates, te_FN_rates)
808
809 adaboost()
810
811
812
813
814
815
816
817
818
```