ECE 661 - Assignment 10

Ali Almuallem, aalmuall@purdue.edu

December 2024

1 Theory Questions

1.1 Overfitting

Overfitting occurs when a model (often a machine learning model) learns the training data very well but fail to generalize to the testing data. This can be due to the scarcity of the training data, or to the capacity of the model. In other words, overfitting can be seen as memorizing the training data but failing to generalize to unseen data. If a model is too complex or too simple for the underlying data, it may fail to generalize. [2]

In machine learning, however, there is some recent research that argue that overparametrization may lead to overcome the outfitting phenomnon in what is known as *"Double Descent"* [3] which is explained below.

1.1.1 Do models overfit in the "modern" overparametrized models?

There is recent research that suggests that as the model complexity gets bigger, the model will initially overfit to the training data (the under-parametrized region in Fig. 1), but as model gets "over-parametrized", it will reach a point where it will actually gets better in generalization and avoids overfitting. This hypothesis is known in the literature as "over-parametrization", which as the name suggests, make large models (larger than usually necessary for a given problem) and performs well on the training and testing data. This challenges the usual conventional thinking that very large models will always yield to overfitting. [4]

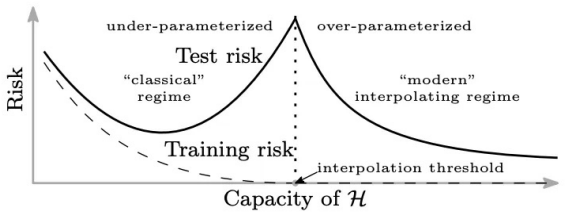

Figure 1: Overparametrization. To the left of the interpolation threshold is the classical thinking of model complexity and overfitting, to the right, the modern hypothesis that suggests that sufficiently large models do not suffer from overfitting. Figure from: [4]

1.2 Reparameterization Trick

The reparameterization trick is a technique used in VAEs and variational inference in general to overcome a backpropagation problem through a random sampling step. The issue arises in VAEs when the latent vector z samples from a distribution, i.e.: $z \sim \mathcal{N}(\mu, \sigma)$, that steps is non-differentable, and therefore the loss cannot be calculated and backpropagated.

The trick therefore solves this issue by defining a latent vector z according to the following equation:

$$z = \mu + \sigma * \epsilon \tag{1}$$

Where $\epsilon \sim \mathcal{N}(0, 1)$. Note however, that drawing a sample from, $z \sim \mathcal{N}(\mu, \sigma)$ or $\epsilon \sim \mathcal{N}(0, 1)$ is nontdifferntiable. However, imploying the trick in the equation above makes z differentiable despite having a non-learnable parameter ϵ.

The role of the new parameter ϵ is said to give a differentiable degree of freedom to the decoder (generator) when it generates a new image from the latent vector z. Others describe the new parameter ϵ as adding noise to the sampled z. [2], [1]

2 Face Recognition using PCA and LDA

The dataset provided has 30 persons faces. Each person has 21 images of their face from slightly different directions. The first four faces can be seen in Fig. 2 *Note*: we resized the images to 32×32 for faster computations.

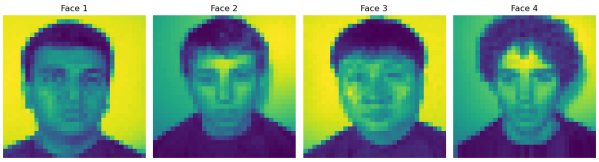

Figure 2: First four persons faces

2.1 Principal Component Analysis: PCA

PCA is a statistical method commonly used for dimentionality reduction tasks. PCA aims to find a lower-dimensional representation of an underlying high-dimensional data. It tries to find directions (prinicipal components) that maximizes the variation in the data so that it can be easily dismantled or identified.

PCA achieves that using the following steps:

- Given N vectorized images $\vec{x}_i \quad i = 1, 2, ..., N$ where each image is represented as a column in the matrix.
- Compute the mean image vector $\vec{m} = \frac{1}{N} \sum_{i=1}^{N} \vec{x}_i$
- Subtract the mean from the images: $X = \begin{bmatrix} \vec{x}_1 - \vec{m} & \vec{x}_2 - \vec{m} & \cdots & \vec{x}_N - \vec{m} \end{bmatrix}$
- To avoid variations due to differences in illumination, normalize the each image in $\vec{x}_i$
- Calculate the covariance $C = \frac{1}{N} \sum_{i=1}^{N} \left\{ (\vec{x}_i - \vec{m})(\vec{x}_i - \vec{m})^T \right\}$
- The above calculation is expensive since it's XX^T so if X is of size 4096×1000, the resulting C is of size $4096 \times 4096 = +16 million$ which is huge. So, we employ a computational trick described in [2]. For the interest of conciseness, we refer the reader to the reference for the details.
- We calculate the eigenvectors $W_K = \begin{bmatrix} \vec{w}_1 & \vec{w}_2 & \cdots & \vec{w}_K \end{bmatrix}$ of our covariance matrix.

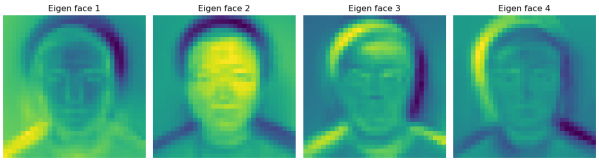

Figure 3: The first 4 eigen faces

- Each vector $\vec{w}_i$ acts as a feature for classification.
- The data can be then projected onto the new space: $\vec{y} = W_K^T (\vec{x} - \vec{m})$

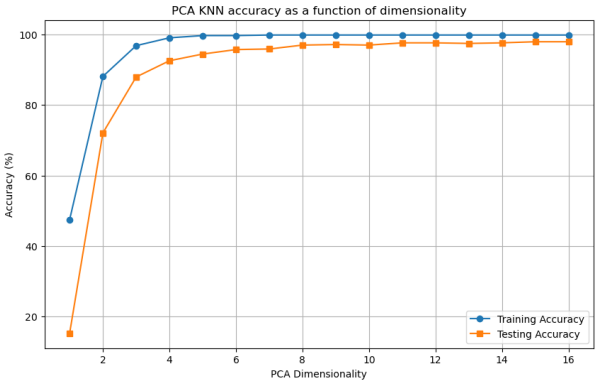

Figure 4: PCA accuracy as a function of dimensionality p.

2.2 UMAP for PCA

The UMAP for PCA shows a good overall seaparation between classes that increases with the dimensionality (value of P). However, there are still many classes that are very close to each other or cannot be disentagled, at least visually by inspecting the plot. We will see that this is slightly different with LDA.

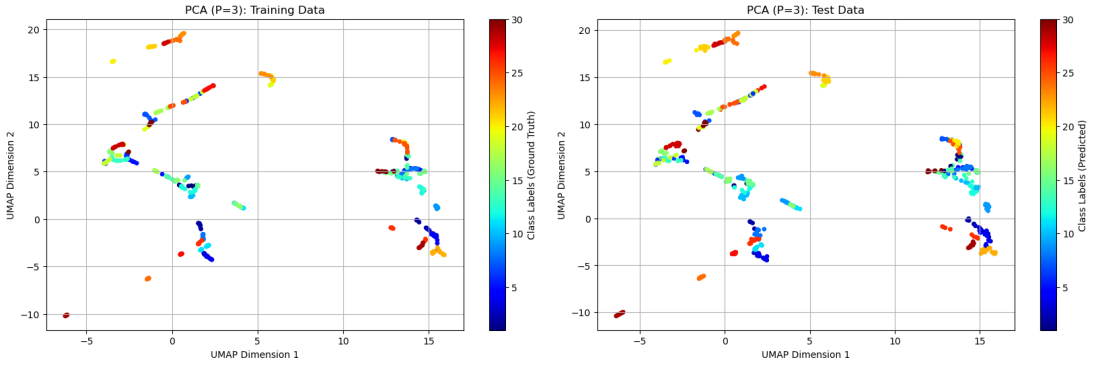

Figure 5: UMAP for PCA with P=3. Left: training. Right: Testing

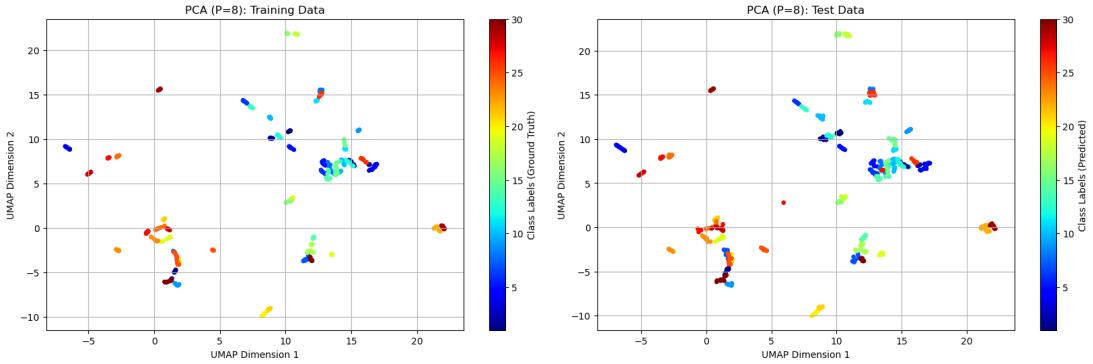

Figure 6: UMAP for PCA with P=8. Left: training. Right: Testing

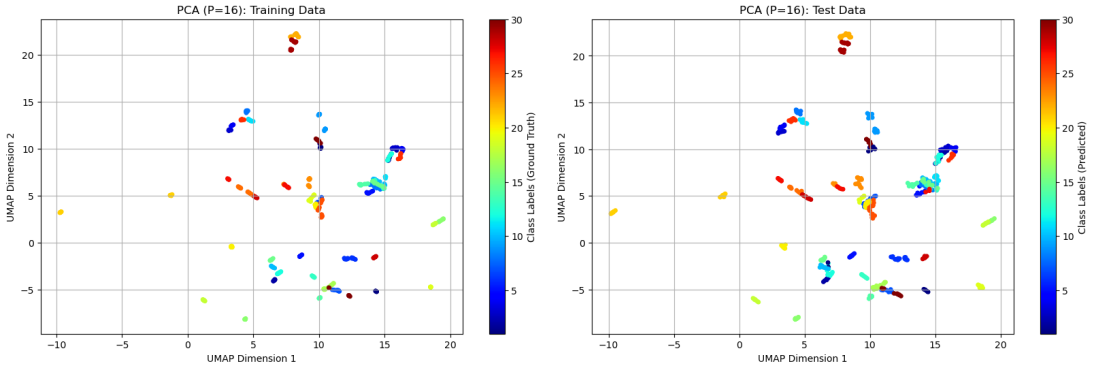

Figure 7: UMAP for PCA with P=16. Left: training. Right: Testing

2.3 Linear Discriminant Analysis: LDA

Just like PCA, LDA is another dimensionality reduction method, but unlike PCA, LDA tries to find the direction of a vector space that maximizes the separability between classes, i.e.: it tries to maximizes the descrimination between classes. Instead of maximizing the overall variance, LDA aims to maximizes the between-class s scatter and minimize the within-class scatter.

The steps are generally as follow:

- Let $\vec{m}$ be the global mean over all images.
- Define the between-classes scatter as: $S_B = \frac{1}{|C|}\sum_{i=1}^{|C|}(\vec{m}_i - \vec{m})(\vec{m}_i - \vec{m})^T$
- Define the within-class scatter as: $S_W = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|C_i|}\sum_{k=1}^{|C_i|}(\vec{x}_k^i - \vec{m}_i)(\vec{x}_k^i - \vec{m}_i)^T$
- If $\vec{w}$ is the underlying vector space, then $\vec{w^T}S_B\vec{w}$ is the projection of the between-classes scatter on $\vec{w}$ and $\vec{w^T}S_w\vec{w}$ it the projection of the within-class scatter on $\vec{w}$.
- Use the Fisher Discriminant Function to maximizes the between-class and minimize the within-class scatter as follows: $J(\vec{w}) = \frac{\vec{w}^T S_B \vec{w}}{\vec{w}^T S_W \vec{w}}$
- The Fisher function must satisfy $S_B\vec{w} = \lambda S_W\vec{w}$ for some constant λ.
- If the within-class scatter is assumed non-singular, the problem can be trasnformed into an eigen-decomposition problem as follows: $S_W^{-1}S_B\vec{w} = \lambda\vec{w}$
- Additional steps might be needed if it is singular. For the sake of conciseness, we refer the reader for [2] for more details.

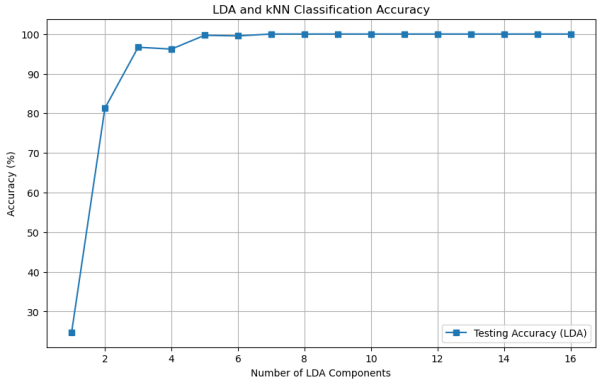

Figure 8: LDA result as a function of dimensionality.

2.4 Overall observation of PCA and LDA

In general, both methods performed similarity with LDA performing a notch better. It could be that PCA needs more dimenstionality. It could also be the nature of PCA objective, as it does not work on the variance between and within classes, but rather the overall variance, which makes LDA better in some situations. However, the obtained results still achieved pretty good accuracy with both methods.

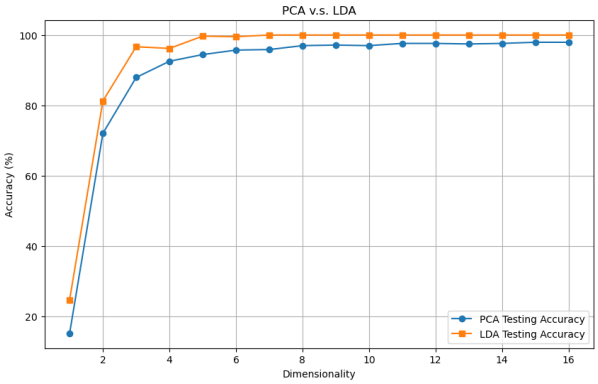

Figure 9: A comparison between LDA and PCA

2.5 UMAP for LDA

The LDA UMAP plots are much more discriminatory than PCA. The seaparation is "cleaner", i.e.: there is only slight or no overlap between classes. Looking at P=3 graphs, we can see only few classes that overlap, and as the dimensionality gets larger $P > 3$, those overlaps disappear. This is supported by the accuracy plots provided earlier, which shows a very high accuracy on the testing dataset as P increases.

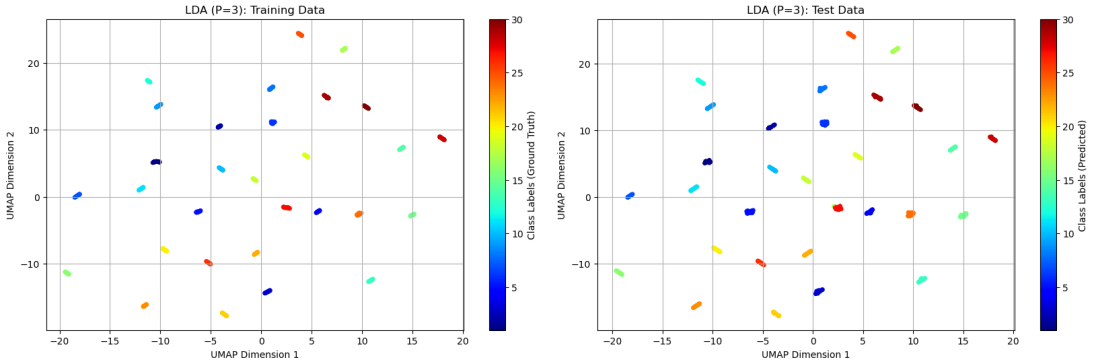

Figure 10: UMAP for LDA with P=3. Left: training. Right: Testing

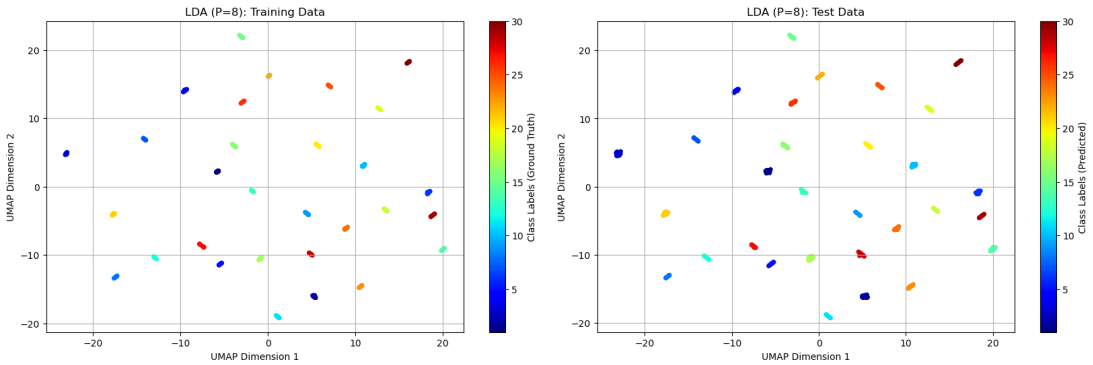

Figure 11: UMAP for LDA with P=8. Left: training. Right: Testing

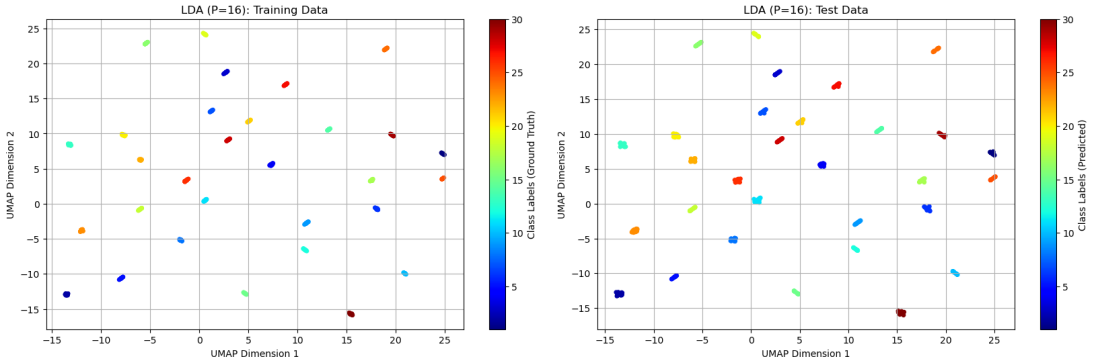

Figure 12: UMAP for LDA with P=16. Left: training. Right: Testing

3 Face Recognition using an Autoencoder

We utilized the provided code and obtain the testing accuracies on different network that were pre-trained on different dimensionalities, namely: $P = 3, 8, 16$. As expected, the accuracy increased as the dimensionality insreases, reaching 100% with $P = 16$ as seen in Fig. 13. We will compare the autoencoder results against PCA and LDA in a later subsection, and detail the accuracies in Table. 1

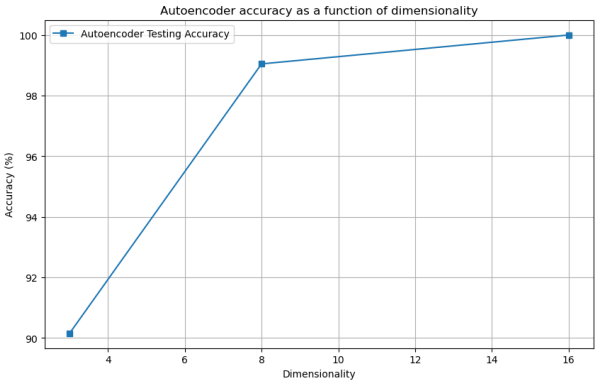

Figure 13: The pre-trained autoencoder result as a function of dimensionality

3.1 UMAP for Autoencoder

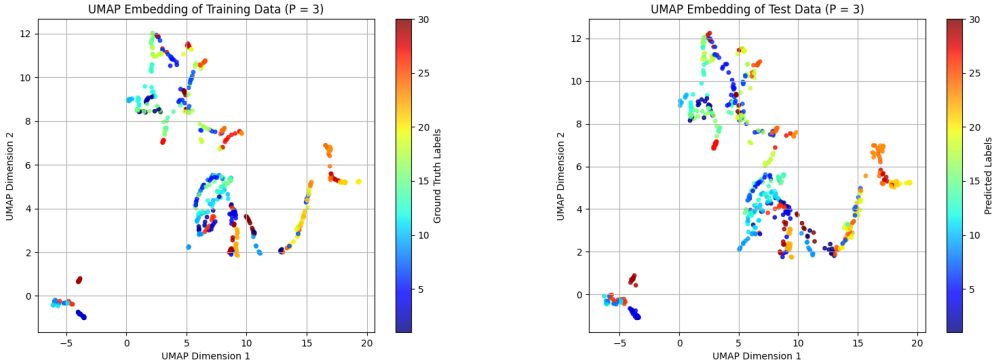

Figure 14: UMAP for Autoencoder with P=3. Left: training. Right: Testing

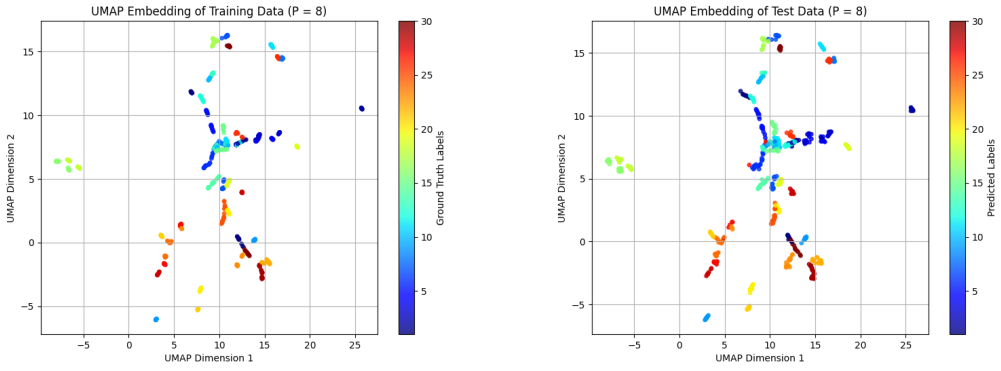

Figure 15: UMAP for Autoencoder with P=8. Left: training. Right: Testing

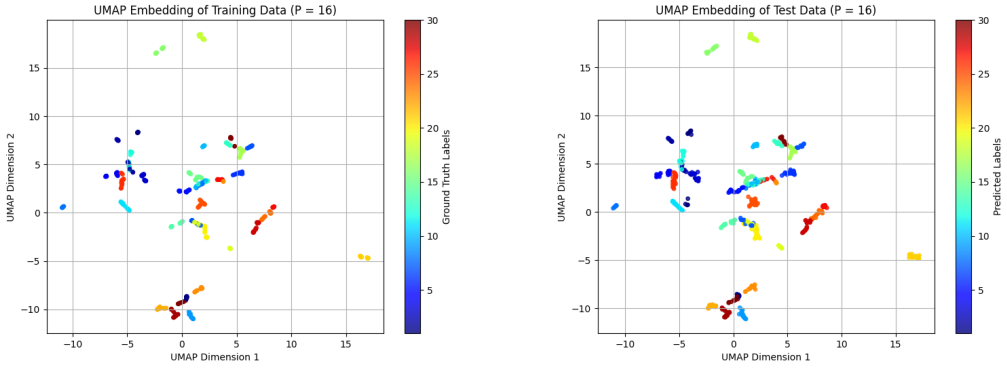

Figure 16: UMAP for Autoencoder with P=16. Left: training. Right: Testing

3.2 Overall comparison: LDA v.s. Autoencoder v.s. PCA

We compare the results of the provided pre-trained autoencoder against our implemented LDA and PCA on the testing data for P values of 3, 8, and 16. As we can see from Fig. 17, the LDA performed the best, especially when the dimensionality was low (P=3), but it has almost the same accuracy as the autoencoder when the dimensionality is sufficiently large (P=16). The PCA in our experiment performed well but slighly lower than the other two methods, which suggests that it could be an issue with the PCA not having the same discrimination power as the LDA and autoencoder, as it does not seek to "descriminate" in its objective function like the LDA does.

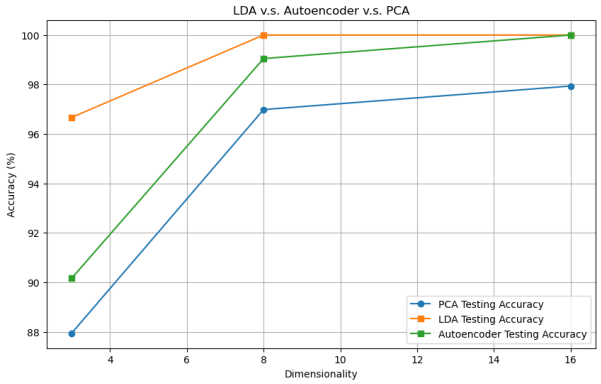

Figure 17: The testing accuracy result for P= 3, 8, 16 for LDA, Autoencoder, and PCA.

	P = 3	P = 8	P = 16
Autoencoder	90.159%	99.048%	**100%**
PCA	87.937%	96.984%	97.937%
LDA	**96.667%**	**100%**	**100%**

Table 1: Accuracy results with different dimensionality P. Highest values are in **bold**

4 Object Detection using Cascaded AdaBoost Classifiers

AdaBoost, short for Adaptive Boosting, is a learning algorithm that combines multiple "weak" classifiers into a single "strong" classifier. A weak classifier here is one that performs slighly better than random chance, while a strong classifier achieves high accuracy rates and low false positive and false negative rates.

The idea behind the cascaded classifers relies on the exponential rule, where if we have a true positive rate of $x_{TP} = 0.99$ and a false positive rate of $x_{FP} = 0.3$ as a start, having 10 classifiers would yield: $x_{TP}^{10} = 0.99^{10} \approx 0.90$ and $x_{FP}^{10} = 0.30^{10} \approx 0.000006$.

The general outline for how AdaBoost works is as follows. For detailed steps and intuition, we refer the reader to the comprehensive guide by Prof. Kak [2]:

- Compute image features. There are multiple ways to do that. I choose to implement HAAR like features with windw of sizes: 2, 4, 8, 16.
- We initilize a weight matrix uniformly w. They are then updated for each sample to determine wether it should be considered in the next iterations.
- Get a weak classifier that specifically targets the training samples that were misclassified by the previous weak classifier (if any). We'll denote it as h_t
- Apply h_t to all training data. The classifier can bee seen as a mapping $h_t : X \mapsto \{-1, 1\}$
- If D_t is the probability distribution, we use the following equation to measure the weighted misclssification rate $Prob\{h_t(x_i) \neq y_i$ where y_i is the label

$$\epsilon_t = \frac{1}{2} \sum_{i=1}^{m} D_t(x_i) \cdot |h_t(x_i) - y_i| \tag{2}$$

- We calculate the trust factor for the current classifier

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{3}$$

- Update the probability distribution for the next iteration

$$D_{t+1}(x_i) = \frac{D_t(x_i) e^{-\alpha_t y_i h_t(x_i)}}{Z_t} \tag{4}$$

  Where Z_t is a normalization factor. So it is set such that

$$D_{t+1}(x_i) = \frac{D_t(x_i) e^{-\alpha_t y_i h_t(x_i)}}{Z_t} \tag{5}$$

- Iterate back to the 3rd step by getting another weak classifer.

4.1 AdaBoost Classification performance

The data for the AdaBoost classification tasks consis of images of cars and other images that do not contain cars as seen in Fig. 18. The dataset images are pretty small (40×20, which could have impacted the accuracy of our feature extraction routine. We expand on that in the following subsections.

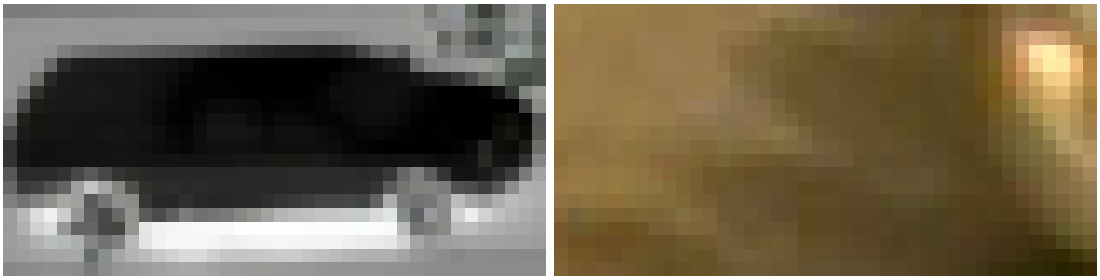

Figure 18: **Left: Positive** sample (car present). **Right: Negative** sample (car not present)

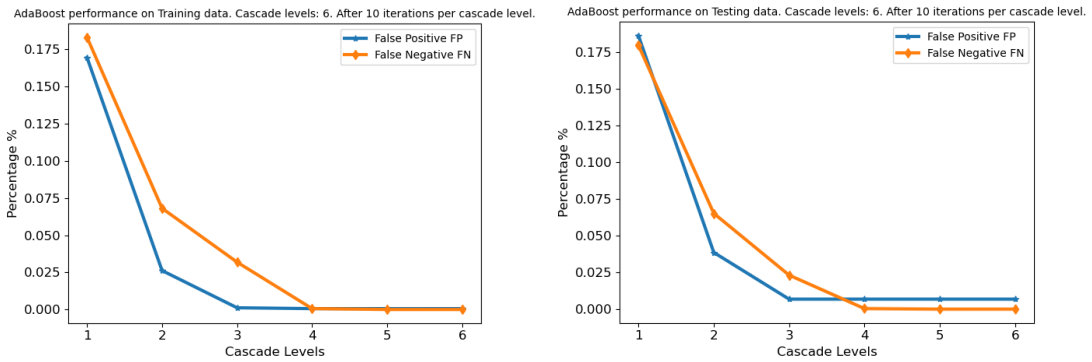

Figure 19: AdaBoost performance. **Left: training** dataset. **Right: Testing** dataset

4.2 Observation on AdaBoost

The plots above in Fig. 19 shows clearly that the classification performance improves substantially as the cascade levels increases. However, it is noticable that the performance plateaus around level four in my implementation. This could be due to a number of reason among which is the dataset which consist of very small low quality images which might have been hard to extract the features from. Another possibility is my feature extraction method which consisted of a HAAR-like windowing technique. The choose of windows sizes might influence the classification performance.

Non-theless, the lowest false positive and false negatives rates achieved are still remarkable as seen in Table.2.

	Training	Testing
FP	0.0005688	0.0068182
FN	0.000000	0.000000

Table 2: Lowest false positive (FP) and false negative (FN) rates achieved on both training and testing datasets.

.

References

[1] Gregory Gundersen. The Reparameterization Trick — gregorygundersen.com. `https://gregorygundersen.com/blog/2018/04/29/reparameterization/`, 2018. [Accessed 30-11-2024].

[2] Avinash Kak. Ece 661: Computer vision. `https://engineering.purdue.edu/kak/computervision/`, 2024.

[3] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent. *OpenAI Blog*, 2019.

[4] Jaideep Ray. What are Over-parameterized models? - medium.com. `https://medium.com/better-ml/what-are-over-parameterized-models-40c53c304367`, 2022. Accessed: 30-11-2024.

5 Code

Listing 1: Assignment 10 Code

```

import numpy as np
import matplotlib.pyplot as plt
import cv2
import umap
import os
import pickle


def reading_images(images_directory, num_image_per_class, num_classes,
    resize = True, resize_to = 32):
    """Read images from a directory
    Images are of this format: xx_yy.png, where xx is the class number,
        and yy is the index of the image per class
    Input:
    - images_directory: the folder where images reside
    - num_image_per_class: number of images per class
    - num_classes: the number of classes

    returns:
    - labels: numpy array of labels
    - images: numpy array of the flatten images. Each image is a column
        vector
    """
    labels = []#np.linspace(1, num_classes, num_classes, dtype= int,
        endpoint= True)

    #Classes starts at 1
    for class_num in range(1, num_classes+1):

        #Images start at 1
        for image_num in range(1, num_image_per_class+1):
            current_class_num = str(class_num) if class_num>=10 else
                "0"+str(class_num)
            current_image_num = str(image_num) if image_num>=10 else
                "0"+str(image_num)
            current_path = images_directory +"/"+current_class_num+"_"+
                current_image_num+".png"

            #Opening image
            current_image = cv2.imread(current_path)

            # Resize the image
            if (resize):
                current_image = cv2.resize(current_image, (resize_to,
                    resize_to),
                                           interpolation = cv2.
                                               INTER_AREA)

            #Convert to grayscale
            current_image = cv2.cvtColor(current_image, cv2.
                COLOR_BGR2GRAY)

            current_image = np.array(current_image).flatten()
```

```
            current_image = np.expand_dims(current_image, 1)

            #If it's the first class and image, create the numpy array
            if (class_num==1 and image_num==1):
                all_images = current_image
            else:
                all_images = np.concatenate((all_images, current_image)
                    , axis = 1)
            labels.append(class_num)
    return all_images, np.array(labels)



# Define the kNN function
def knn_classify(trainX, trainY, testX, k):
    predictions = []
    for test_point in testX:
        distances = np.linalg.norm(trainX - test_point, axis=1)  #
            Compute distances
        neighbors = np.argsort(distances)[:k]  # Find k nearest
            neighbors
        neighbor_labels = trainY[neighbors]  # Get their labels
        prediction = np.bincount(neighbor_labels).argmax()  # Majority
            vote
        predictions.append(prediction)
    return np.array(predictions)


#Compute PCA
def manual_pca(X, num_components):
    mean_vector = np.mean(X, axis=0)  # Compute mean
    X_centered = X - mean_vector  # Center the data
    covariance_matrix = np.dot(X_centered.T, X_centered) / (X_centered.
        shape[0] - 1)  # Covariance matrix
    eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)  #
        Eigen decomposition
    idx = np.argsort(-eigenvalues)  # Sort eigenvalues in descending
        order
    eigenvectors = eigenvectors[:, idx[:num_components]]  # Select top
        eigenvectors
    X_reduced = np.dot(X_centered, eigenvectors)  # Project data to the
         new basis
    return X_reduced, mean_vector, eigenvectors

# Compute accuracy
def compute_accuracy(true_labels, predicted_labels):
    return np.mean(true_labels == predicted_labels) * 100

#LDA
# Define the LDA function
def manual_lda(trainX, trainY, num_components):
    classes = np.unique(trainY)
    overall_mean = np.mean(trainX, axis=0)

    # Initialize within-class scatter matrix and between-class scatter
        matrix
    Sw = np.zeros((trainX.shape[1], trainX.shape[1]))
    Sb = np.zeros((trainX.shape[1], trainX.shape[1]))
```

```
93
94      for c in classes:
95          class_samples = trainX[trainY == c]
96          class_mean = np.mean(class_samples, axis=0)
97          Sw += np.dot((class_samples - class_mean).T, (class_samples -
                class_mean))
98          n_class_samples = class_samples.shape[0]
99          mean_diff = (class_mean - overall_mean).reshape(-1, 1)
100         Sb += n_class_samples * np.dot(mean_diff, mean_diff.T)
101
102     eigvals, eigvecs = np.linalg.eig(np.linalg.pinv(Sw).dot(Sb))
103     idx = np.argsort(-eigvals.real)  # Sort eigenvalues in descending
            order
104     eigvecs = eigvecs[:, idx[:num_components]].real  # Select top
            eigenvectors
105
106     # Project data onto the LDA components
107     trainX_lda = np.dot(trainX, eigvecs)
108     return trainX_lda, eigvecs
109
110
111 #Plot UMAP
112 def plot_umap_embeddings(trainX_proj, trainY, testX_proj, testY_pred,
        title_train, title_test):
113     """
114     Function to plot UMAP embeddings for training and test data.
115     """
116     umap_reducer = umap.UMAP(n_components=2, random_state=42)
117
118     # Fit UMAP on training data
119     trainX_umap = umap_reducer.fit_transform(trainX_proj.real)
120     # Transform test data using the same UMAP
121     testX_umap = umap_reducer.transform(testX_proj.real)
122
123     # Plot training data
124     plt.figure(figsize=(10, 6))
125     scatter = plt.scatter(trainX_umap[:, 0], trainX_umap[:, 1], c=
            trainY, cmap='jet', s=15)
126     plt.title(title_train)
127     plt.xlabel("UMAP Dimension 1")
128     plt.ylabel("UMAP Dimension 2")
129     plt.colorbar(scatter, label="Class Labels (Ground Truth)")
130     plt.grid()
131     plt.show()
132
133     # Plot test data
134     plt.figure(figsize=(10, 6))
135     scatter = plt.scatter(testX_umap[:, 0], testX_umap[:, 1], c=
            testY_pred, cmap='jet', s=15)
136     plt.title(title_test)
137     plt.xlabel("UMAP Dimension 1")
138     plt.ylabel("UMAP Dimension 2")
139     plt.colorbar(scatter, label="Class Labels (Predicted)")
140     plt.grid()
141     plt.show()
142
143
144 #Reading data
```

```
trainX, trainY = reading_images('FaceRecognition/train', 21, 30, True, 32)
testX, testY = reading_images('FaceRecognition/test', 21, 30, True, 32)

trainX = trainX.T
trainY = trainY.T
testX = testX.T
testY = testY.T


#plot the first four faces
fig, axes = plt.subplots(1, 4, figsize=(12, 4))  # Create a 1x4 grid of subplots

for i in range(4):
    # Reshape and take the real part of the eigenvector
    image = trainX[i*21,:].reshape(32, 32).real

    # Plot the image
    axes[i].imshow(image)
    axes[i].axis('off')  # Turn off axis for a cleaner look
    axes[i].set_title(f"Face {i+1}")

plt.tight_layout()  # Adjust layout to avoid overlap
plt.show()


#PCA calculation

# Parameters
P = 16  # Number of principal components
K = 3   # Number of neighbors for KNN

# Step 1: Perform manual PCA on the training data
trainX_pca, train_mean, eigenvectors = manual_pca(trainX, P)

# Step 2: Project the test data using the same eigenvectors
testX_centered = testX - train_mean
testX_pca = np.dot(testX_centered, eigenvectors)

# Step 3: Classify using kNN and compute accuracies
train_predictions = knn_classify(trainX_pca, trainY, trainX_pca, K)
test_predictions = knn_classify(trainX_pca, trainY, testX_pca, K)

train_accuracy = compute_accuracy(trainY, train_predictions)
test_accuracy = compute_accuracy(testY, test_predictions)

# Print results
print(f"Training Accuracy: {train_accuracy:.2f}%")
print(f"Testing Accuracy: {test_accuracy:.2f}%")

# Step 4: Analyze accuracy for different values of P
p_values = range(1, P + 1)
train_accuracies = []
test_accuracies = []

for p in p_values:
    trainX_pca, train_mean, eigenvectors = manual_pca(trainX, p)
```

```
    testX_pca = np.dot(testX - train_mean, eigenvectors)
    train_predictions = knn_classify(trainX_pca, trainY, trainX_pca, K)
    test_predictions = knn_classify(trainX_pca, trainY, testX_pca, K)
    train_accuracies.append(compute_accuracy(trainY, train_predictions))
    test_accuracies.append(compute_accuracy(testY, test_predictions))

# Plot PCA results
plt.figure(figsize=(10, 6))
plt.plot(p_values, train_accuracies, label="Training Accuracy", marker='o')
plt.plot(p_values, test_accuracies, label="Testing Accuracy", marker='s')
plt.xlabel("Number of Principal Components (P)")
plt.ylabel("Accuracy (%)")
plt.title("PCA and kNN Classification Accuracy")
plt.legend()
plt.grid()
plt.show()


#plot the first four eigen faces
fig, axes = plt.subplots(1, 4, figsize=(12, 4))  # Create a 1x4 grid of subplots

for i in range(4):
    # Reshape and take the real part of the eigenvector
    image = eigenvectors[:, i].reshape(32, 32).real

    # Plot the image
    axes[i].imshow(image)
    axes[i].axis('off')  # Turn off axis for a cleaner look
    axes[i].set_title(f"Eigen face {i+1}")

plt.tight_layout()  # Adjust layout to avoid overlap
plt.show()



#2. LDA calculation

# Parameters for LDA
L = 2  # Number of LDA components

# Perform LDA on training data
trainX_lda, lda_eigvecs = manual_lda(trainX, trainY, L)

# Project test data using LDA
testX_lda = np.dot(testX, lda_eigvecs)

# Classify using kNN
train_predictions_lda = knn_classify(trainX_lda, trainY, trainX_lda, K)
test_predictions_lda = knn_classify(trainX_lda, trainY, testX_lda, K)

train_accuracy_lda = compute_accuracy(trainY, train_predictions_lda)
test_accuracy_lda = compute_accuracy(testY, test_predictions_lda)

print(f"LDA Training Accuracy: {train_accuracy_lda:.2f}%")
```

```
print(f"LDA Testing Accuracy: {test_accuracy_lda:.2f}%")

# Analyze accuracy for different LDA dimensions
lda_dimensions = range(1, 16 + 1)
train_accuracies_lda = []
test_accuracies_lda = []

for dim in lda_dimensions:
    trainX_lda, lda_eigvecs = manual_lda(trainX, trainY, dim)
    testX_lda = np.dot(testX, lda_eigvecs)
    train_predictions_lda = knn_classify(trainX_lda, trainY, trainX_lda, K)
    test_predictions_lda = knn_classify(trainX_lda, trainY, testX_lda, K)
    train_accuracies_lda.append(compute_accuracy(trainY, train_predictions_lda))
    test_accuracies_lda.append(compute_accuracy(testY, test_predictions_lda))

# Plot LDA results
plt.figure(figsize=(10, 6))
plt.plot(lda_dimensions, train_accuracies_lda, label="Training Accuracy (LDA)", marker='o')
plt.plot(lda_dimensions, test_accuracies_lda, label="Testing Accuracy (LDA)", marker='s')
plt.xlabel("Number of LDA Components")
plt.ylabel("Accuracy (%)")
plt.title("LDA and kNN Classification Accuracy")
plt.legend()
plt.grid()
plt.show()


# Plot PCA vs LDA testing results
plt.figure(figsize=(10, 6))
plt.plot(lda_dimensions, test_accuracies, label="PCA Testing Accuracy", marker='o')
plt.plot(lda_dimensions, test_accuracies_lda, label="LDA Testing Accuracy", marker='s')
plt.xlabel("Dimensionality")
plt.ylabel("Accuracy (%)")
plt.title("PCA v.s. LDA")
plt.legend()
plt.grid()
plt.show()


#Already obtained the autoencoder numbers by running the code provided for P=3, 8, 16
autoencoder_accuracies = [90.1587, 99.0476, 100.00]
PCA_3_8_16_accuracy = [test_accuracies[2], test_accuracies[7], test_accuracies[-1]]
LDA_3_8_16_accuracy = [test_accuracies_lda[2], test_accuracies_lda[7], test_accuracies_lda[-1]]


# Plot LDA v.s. PCA v.s. Autoencoder results
plt.figure(figsize=(10, 6))
```

```
plt.plot([3, 8, 16], PCA_3_8_16_accuracy, label="PCA Testing Accuracy", marker='o')
plt.plot([3, 8, 16], LDA_3_8_16_accuracy, label="LDA Testing Accuracy", marker='s')
plt.plot([3, 8, 16], autoencoder_accuracies, label="Autoencoder Testing Accuracy", marker='s')
plt.xlabel("Dimensionality")
plt.ylabel("Accuracy (%)")
plt.title("LDA v.s. Autoencoder v.s. PCA")
plt.legend()
plt.grid()
plt.show()


# Plot Autoencoder results alone
plt.figure(figsize=(10, 6))
plt.plot([3, 8, 16], autoencoder_accuracies, label="Autoencoder Testing Accuracy", marker='s')
plt.xlabel("Dimensionality")
plt.ylabel("Accuracy (%)")
plt.title("Autoencoder accuracy as a function of dimensionality")
plt.legend()
plt.grid()
plt.show()




#Plot the UMAP

# Experiment parameters
P_values = [3, 8,  16]  # Example values for PCA
L_values = [3, 8, 16]  # Example values for LDA

# Generate plots for PCA and LDA
for P in P_values:
    # PCA Projection
    trainX_pca, _, eigenvectors_pca = manual_pca(trainX, P)
    testX_pca = np.dot(testX - train_mean, eigenvectors_pca)

    # kNN Predictions
    testY_pred_pca = knn_classify(trainX_pca, trainY, testX_pca, K)

    # Plot UMAP for PCA embeddings
    plot_umap_embeddings(
        trainX_proj=trainX_pca,
        trainY=trainY,
        testX_proj=testX_pca,
        testY_pred=testY_pred_pca,
        title_train=f"PCA (P={P}): Training Data",
        title_test=f"PCA (P={P}): Test Data"
    )

for L in L_values:
    # LDA Projection
    trainX_lda, lda_eigvecs = manual_lda(trainX, trainY, L)
    testX_lda = np.dot(testX, lda_eigvecs)

    # kNN Predictions
```

```
    testY_pred_lda = knn_classify(trainX_lda, trainY, testX_lda, K)

    # Plot UMAP for LDA embeddings
    plot_umap_embeddings(
        trainX_proj=trainX_lda,
        trainY=trainY,
        testX_proj=testX_lda,
        testY_pred=testY_pred_lda,
        title_train=f"LDA (P={L}): Training Data",
        title_test=f"LDA (P={L}): Test Data"
    )

#3. AdaBoost

def calculate_features(img):
    #  to grayscale
    if len(img.shape) > 2:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Compute the integral image
    integral_img = cv2.integral(img)

    window_sizes = [2, 4, 8, 16]
    features = []

    # Loop over windows
    for win_size in window_sizes:
        for y in range(0, img.shape[0] - win_size + 1, win_size):
            for x in range(0, img.shape[1] - win_size + 1, win_size):
                #  horizontal feature
                mid_x = x + win_size // 2
                sum_left = integral_img[y + win_size, mid_x] - integral_img[y, mid_x] \
                        - (integral_img[y + win_size, x] - integral_img[y, x])
                sum_right = integral_img[y + win_size, x + win_size] - integral_img[y, x + win_size] \
                         - (integral_img[y + win_size, mid_x] - integral_img[y, mid_x])
                two_rect_horizontal = sum_right - sum_left
                features.append(two_rect_horizontal)

                # vertical feature
                mid_y = y + win_size // 2
                sum_top = integral_img[mid_y, x + win_size] - integral_img[mid_y, x] \
                       - (integral_img[y, x + win_size] - integral_img[y, x])
                sum_bottom = integral_img[y + win_size, x + win_size] - integral_img[y + win_size, x] \
                          - (integral_img[mid_y, x + win_size] - integral_img[mid_y, x])
                two_rect_vertical = sum_bottom - sum_top
                features.append(two_rect_vertical)

    return np.array(features)


```

```
406 def concatenate_feats(pos_feats, neg_feats):
407     return np.concatenate((pos_feats, neg_feats), axis=0)
408
409
410 def get_labels_from_feats(pos_feats, neg_feats):
411     """
412     Return positvie and negative labels, un-concatenated given positive
            and negative features
413     """
414     positive_labels = np.ones((pos_feats.shape[0]))
415     negative_labels = np.zeros((neg_feats.shape[0]))
416
417     return positive_labels, negative_labels
418
419
420 def concatenate_labels(pos_labels, neg_labels):
421     combined_labels = np.concatenate((pos_labels, neg_labels), axis =
           0)
422     combined_labels = combined_labels.astype(np.uint8)
423
424     return combined_labels
425
426 #
427 def get_features(data_path, training = True):
428     """
429     Input:
430     - data_path = "CarDetection"
431     - training = True if the features needed are the training features.
            False for testing data features
432     Returns:
433     - positive_feat: positive data features
434     - negative_feat: negative data features
435     - positive_labels
436     - negative_labels
437     """
438     positive_feat = []
439     negative_feat = []
440
441     for category in ["positive", "negative"]:
442         print("Getting the features of " + category + " training data"
               if training
443                else "Getting the features of " + category + " testing
                      data")
444         current_path = data_path
445         current_path += "/train/" + category if training else "/test/"
               +category
446         print ("Current path: ", current_path)
447
448         for file_name in os.listdir(current_path):
449             #Read the images
450             current_image = cv2.imread(current_path+"/"+file_name)
451             if category == "positive":
452                 positive_feat.append(calculate_features(current_image))
453             else:
454                 negative_feat.append(calculate_features(current_image))
455
456     #Create the labels
457     positive_feat = np.array(positive_feat)
```

```
458     negative_feat = np.array(negative_feat)
459
460     positive_labels, negative_labels = get_labels_from_feats(
            positive_feat, negative_feat)
461
462     return positive_feat, negative_feat, positive_labels,
            negative_labels
463
464
465 #Weak classifer
466 def get_weak_classifier(combined_feats, combined_labels, weights):
467
468     #Init error to large value
469     classifier_err = 1e15
470
471     for feat in range(combined_feats.shape[1]):
472         current_feat = combined_feats[:, feat]
473
474         #Sort the features for Ranking
475         sorted_ind = np.argsort(current_feat)
476
477         #Sort the features, labels, and weights
478         current_feat_sorted = current_feat[sorted_ind]
479         current_labels_sorted = combined_labels[sorted_ind]
480         current_weights_sorted = weights[sorted_ind]
481
482         #Declare pos, neg weights
483         num_feat = combined_feats.shape[0]
484         pos_weights = np.zeros((num_feat, 1))
485         neg_weights = np.zeros((num_feat, 1))
486
487         #Assign pos, neg weights
488
489         pos_weights[current_labels_sorted == 1, 0] =
                current_weights_sorted[current_labels_sorted == 1]
490         neg_weights[current_labels_sorted == 0, 0] =
                current_weights_sorted[current_labels_sorted == 0]
491
492         #Error polarity
493         pos_weights_cumsum = np.cumsum(pos_weights)
494         neg_weights_cumsum = np.cumsum(neg_weights)
495
496         pos_weights_sum = np.sum(pos_weights)
497         neg_weights_sum = np.sum(neg_weights)
498
499         error_polarity1 = pos_weights_cumsum + neg_weights_sum -
                neg_weights_cumsum
500         error_polarity2 = neg_weights_cumsum + pos_weights_sum -
                pos_weights_cumsum
501
502         #Unsqueeze
503         error_polarity1 = np.reshape(error_polarity1, [-1, 1])#np.
                expand_dims(error_polarity1, 1)
504         error_polarity2 = np.reshape(error_polarity2, [-1, 1])#np.
                expand_dims(error_polarity2, 1)
505
506         #Stack errors
```

```
        total_error = np.concatenate((error_polarity1, error_polarity2)
            , axis = 1)

        #Get the index where error is minimum
        total_min_error = np.argmin(total_error)
        error_min_index = np.unravel_index(total_min_error, total_error
            .shape)

        current_min_error = total_error[error_min_index]

        #Check if the new error is lower than the previous classifer
            error
        #If yes, assign it as the new classifer error
        if current_min_error < classifier_err:
            classifier_err = current_min_error

            new_threshold = current_feat_sorted[error_min_index[0]]

            polarity = 0
            prediction = current_feat < new_threshold
            if error_min_index[1]==0:
                polarity = 1
                prediction = current_feat >= new_threshold


            best_classifier = [feat, new_threshold, polarity,
                classifier_err, prediction]

    return best_classifier


def get_cascaded_AdaBoost(combined_feats, combined_labels, epochs,
    levels):
    """
    Input:
    - combined_feats = an array of combined features for positive and
        negative
    - combined_labels = for positive and negative
    - epochs = how many epochs or iterations per cascade
    - levels = how many cascade levels to run
    """

    #Init trust factor to something very small
    best_trust = -1e16

    #Initi weights
    pos_w_total =  np.sum(combined_labels == 1)
    neg_w_total = np.sum(combined_labels == 0)
    pos_w = np.repeat(1 / pos_w_total, pos_w_total)
    neg_w = np.repeat(1 / neg_w_total, neg_w_total)

    weights = np.concatenate((pos_w, neg_w), axis = 0)


    for epoch in range(epochs):

        weights_sum = np.sum(weights)
        weights = weights / weights_sum #Normalizing the weights
```

```
559
560         #Get a weak classifier
561         current_weak_classifer = get_weak_classifier(combined_feats=
                combined_feats, combined_labels= combined_labels, weights=
                weights)
562
563         current_feat_index = current_weak_classifer[0]
564         current_threshold = current_weak_classifer[1]
565         current_polarity = current_weak_classifer[2]
566         current_classifer_err = current_weak_classifer[3]
567         current_prediction = current_weak_classifer[4]
568
569         #Set the negative labels to -1
570         current_prediction = np.where(current_prediction == 0, -1,
                current_prediction)
571
572
573         weights_expanded = np.expand_dims(weights, 0)
574         #Calculate absolute prediction error
575         label_err = np.abs(current_prediction - combined_labels)
576
577
578         current_epsilon = np.matmul(weights_expanded, label_err.T)
579         current_epsilon = np.expand_dims(current_epsilon, 1).squeeze()
580         current_epsilon *= 0.5
581
582         #Update trust factor and weights
583         trust_factor = np.log((1-current_epsilon)/current_epsilon)
584         trust_factor *= 0.5
585
586         #Updating the weights based on the trust factor
587         weights = weights * np.exp(- trust_factor * combined_labels *
                current_prediction)
588
589         #Computing the Falsse Positives FP, and False Negatives FN
590         #Get the false positive and false negative rates
591         total_negatives = np.sum(combined_labels==0)
592         total_positives = np.sum(combined_labels==1)
593
594         misclassified_negatives = np.sum((combined_labels == 0) & (
                current_prediction == 1))
595         misclassified_positives = np.sum((combined_labels == 1) & (
                current_prediction == -1))
596
597         FP = misclassified_negatives / total_negatives
598         FN = misclassified_positives / total_positives
599
600         print ("Training cascase level: %d. Epoch: %d. Trust: %.3f.
                Epsilon: %.3f. FP: %.7f. FN: %.7f." %
601             (levels+1, epoch, trust_factor, current_epsilon, FP, FN)
                    )
602
603         #Check the trust factor
604         if (trust_factor > best_trust):
605             best_trust = trust_factor
606             best_prediction = current_prediction
607             best_FP = FP
608             best_FN = FN
```

```
609
610             best_classifier = [current_feat_index, current_threshold,
611                                current_polarity, current_classifer_err,
                                       current_prediction, FP, FN,
                                   current_epsilon, trust_factor]
612
613     positive_feats = combined_feats[: np.sum(combined_labels==1),:]
614     negative_feats = combined_feats[np.sum(combined_labels==1):, :]
615     negative_feats_mask = np.where(best_prediction[np.sum(
           combined_labels==1):]==1)
616
617     negative_feats = negative_feats[negative_feats_mask, :][0]
618
619     combined_feats = concatenate_feats(positive_feats, negative_feats)
620
621     current_pos_labels, current_neg_labels = get_labels_from_feats(
           positive_feats, negative_feats)
622
623     combined_labels  = concatenate_labels(current_pos_labels,
           current_neg_labels)
624
625     return combined_feats, combined_labels, best_FP, best_FN,
           best_classifier
626
627
628
629 def predict_ada(cascade_classifiers, features, labels):
630     feats_number = features.shape[0]
631     final_labels = np.ones((feats_number, 1))
632     class_indices = np.arange(features.shape[0]).reshape(-1, 1)
633     FPRs, FNRs = [], []
634
635     for level, classifier in enumerate(cascade_classifiers.values()):
636         feature_id, threshold, polarity, trust_factor = classifier[0],
               classifier[1], classifier[2], classifier[-1]
637
638         feature_column = features[:, feature_id].reshape(-1, 1)
639         predictions = (feature_column >= threshold) if polarity == 1
               else (feature_column < threshold)
640         predictions = np.where(predictions, 1, -1)
641
642         weighted_preds = trust_factor * predictions
643         final_preds = weighted_preds >= trust_factor
644
645         total_negatives = np.sum(labels == 0)
646         total_positives = np.sum(labels == 1)
647
648         FPR = np.sum(final_preds[total_positives:] == 1) /
               total_negatives if total_negatives else 0
649         FNR = 1 - np.sum(final_preds[:total_positives] == 1) /
               total_positives if total_positives else 0
650
651         features = features[final_preds.flatten() == 1]
652         labels = labels[final_preds.flatten() == 1]
653         negative_indices = class_indices[final_preds.flatten() == 0]
654         final_labels[negative_indices] = -1
655         class_indices = class_indices[final_preds.flatten() == 1]
656
```

```
        FPRs.append(FPR if level == 0 else FPRs[-1] * FPR)
        FNRs.append(FNR if level == 0 else FNRs[-1] * FNR)

    return FPRs, FNRs, final_labels

def plot_adaboost(FPs, FNs, title=None):

    plt.rcParams.update({
        'font.size': 10,
        'figure.titlesize': 14,
        'legend.fontsize': 10,
        'axes.titlesize': 10,
        'axes.labelsize': 12,
        'xtick.labelsize': 12,
        'ytick.labelsize': 12
    })

    plt.figure()
    plt.plot(FPs, '-*', label='False Positive FP', linewidth=3)
    plt.plot(FNs, '-d', label='False Negative FN', linewidth=3)
    plt.legend()

    plt.xlabel('Cascade Levels')
    plt.ylabel('Percentage %')
    plt.xticks(np.arange(len(FPs)), [str(ix) for ix in range(1, len(FPs
        ) + 1)])
    plt.title(title)
    plt.tight_layout()
    plt.show()


def excute_adaboost(number_of_cascades = 10, iterations = [1, 5, 25],
    data_path = "CarDetection", train = True):

    """
    Inputs:
    - number_of_cascades: how many levels
    - iterations: iterations per cascade
    - data_path: parent data directory "CarDetection"
    - train: if True, excute adaboost on training dataset, else do
        inference on testing data
    NOTE: the code has to be run on training mode first to save the
        classifier that will be used for testing.

    Returns:
    None. Plots will be shown and accuracies will be printed.
    """
    for epoch in iterations:
        current_FPR = 1
        current_FNR = 1
        FPs = []
        FNs = []
        tol = 1e-6
        best_classifer={}

        tolerance_counter = 0
        if train:
            print("Performing Adaboost on Training data")
```

```
711             train_positive_feats, train_negative_feats,
                    train_positive_labels, train_negative_labels =
                    get_features(data_path, train)
712             combined_train_feats = concatenate_feats(
                    train_positive_feats, train_negative_feats)
713             combined_train_labels = concatenate_labels(
                    train_positive_labels, train_negative_labels)
714
715
716             for level in range(number_of_cascades):
717                 current_Ada = get_cascaded_AdaBoost(
                        combined_train_feats, combined_train_labels, epoch,
                        level)
718
719                 combined_train_feats, combined_train_labels, best_FPR,
                        best_FNR, best_weak_classifier = current_Ada
720
721                 best_classifer[str(level + 1)] = best_weak_classifier
722                 current_FPR *= best_FPR
723                 current_FNR *= best_FNR
724                 FPs.append(current_FPR)
725                 FNs.append(current_FNR)
726
727                 print("Current cumulative FP: %.7f, cumulative FN %.7F"
                        % (current_FPR, current_FNR))
728
729                 #Increasing the tolerance
730                 tolerance_counter = level + 1
731
732                 if current_FPR <= tol:
733                     print("We reached FP tolerance level:", tol, 'after
                            cascade level =', level + 1)
734
735                 if np.sum(combined_train_labels == 0) == 0:
736                     print("No images with negative labels remained
                            after cascade level: ", level+1)
737                     break
738
739             #Plot the FP and FN rates
740             current_title = "AdaBoost performance on Training data.
                    Cascade levels: " + str(number_of_cascades) + ". After "
                     + str(epoch) + " iterations per cascade level."
741             plot_adaboost(FPs= FPs, FNs= FNs, title = current_title)
742
743             #Save the best classifer
744             with open("Results/AdaboostMine" + '/
                    Adaboost_classifer_level_' + str(epoch) + '_' + str(
                    number_of_cascades) + '.pkl', 'wb') as f:
745                 pickle.dump(best_classifer, f)
746
747         #Test data
748         else:
749             print("Performing Adaboost on Testing data")
750             #loading the best classifer
751             with open("Results/AdaboostMine" + '/
                    Adaboost_classifer_level_' + str(epoch) + '_' + str(
                    number_of_cascades) + '.pkl', 'rb') as f:
752                 best_classifer = pickle.load(f)
```

```
753
754             #Get the test features and labels
755             test_positive_feats, test_negative_feats,
                    test_positive_labels, test_negative_labels =
                    get_features(data_path, False)
756             combined_test_feats = concatenate_feats(test_positive_feats
                    , test_negative_feats)
757             combined_test_labels = concatenate_labels(
                    test_positive_labels, test_negative_labels)
758
759             #Get the FP and FN
760             FPs, FNs, _ = predict_ada(best_classifer,
                    combined_test_feats, combined_test_labels)
761             print("Current Testing cumulative FP: %.7f, cumulative FN
                    %.7F" % (np.array(FPs).min(), np.array(FNs).min()))
762
763             #current_title = "Test Data - Iteration: " + str(epoch) +".
                     Reached minimum at cascade level: "+str(
                    number_of_cascades)
764             current_title = "AdaBoost performance on Testing data.
                    Cascade levels: " + str(number_of_cascades) + ". After "
                     + str(epoch) + " iterations per cascade level."
765             plot_adaboost(FPs = FPs, FNs = FNs, title = current_title)
766
767
768 #Training data AdaBoost
769 excute_adaboost(number_of_cascades= 6, iterations=[1, 3, 10], data_path
        = "CarDetection", train = True)
770
771
772 #Performing Adaboost on testing data
773 excute_adaboost(number_of_cascades= 6, iterations=[1, 3, 10], data_path
        = "CarDetection", train = False)
```

Listing 2: Assignment 10 Autoencoder Code

```
1  import os
2
3  import numpy as np
4  import torch
5  from torch import nn, optim
6  from PIL import Image
7  from torch.autograd import Variable
8  from torch.utils.data import Dataset, DataLoader
9  from torchvision import transforms
10 import umap
11 import matplotlib.pyplot as plt
12 from sklearn.neighbors import KNeighborsClassifier
13
14 class DataBuilder(Dataset):
15     def __init__(self, path):
16         self.path = path
17         self.image_list = [f for f in os.listdir(path) if f.endswith('.
               png')]
18         self.label_list = [int(f.split('_')[0]) for f in self.
               image_list]
19         self.len = len(self.image_list)
20         self.aug = transforms.Compose([
21             transforms.Resize((64, 64)),
```

```
            transforms.ToTensor(),
        ])

    def __getitem__(self, index):
        fn = os.path.join(self.path, self.image_list[index])
        x = Image.open(fn).convert('RGB')
        x = self.aug(x)
        return {'x': x, 'y': self.label_list[index]}

    def __len__(self):
        return self.len


class Autoencoder(nn.Module):

    def __init__(self, encoded_space_dim):
        super().__init__()
        self.encoded_space_dim = encoded_space_dim
        ### Convolutional section
        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(3, 8, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.LeakyReLU(True)
        )
        ### Flatten layer
        self.flatten = nn.Flatten(start_dim=1)
        ### Linear section
        self.encoder_lin = nn.Sequential(
            nn.Linear(4 * 4 * 64, 128),
            nn.LeakyReLU(True),
            nn.Linear(128, encoded_space_dim * 2)
        )
        self.decoder_lin = nn.Sequential(
            nn.Linear(encoded_space_dim, 128),
            nn.LeakyReLU(True),
            nn.Linear(128, 4 * 4 * 64),
            nn.LeakyReLU(True)
        )
        self.unflatten = nn.Unflatten(dim=1,
                                      unflattened_size=(64, 4, 4))
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(32, 16, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(True),
```

```
            nn.ConvTranspose2d(8, 3, 3, stride=2,
                               padding=1, output_padding=1)
        )

    def encode(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        mu, logvar = x[:, :self.encoded_space_dim], x[:, self.
            encoded_space_dim:]
        return mu, logvar

    def decode(self, z):
        x = self.decoder_lin(z)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x)
        return x

    @staticmethod
    def reparameterize(mu, logvar):
        std = logvar.mul(0.5).exp_()
        eps = Variable(std.data.new(std.size()).normal_())
        return eps.mul(std).add_(mu)


class VaeLoss(nn.Module):
    def __init__(self):
        super(VaeLoss, self).__init__()
        self.mse_loss = nn.MSELoss(reduction="sum")

    def forward(self, xhat, x, mu, logvar):
        loss_MSE = self.mse_loss(xhat, x)
        loss_KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp
            ())
        return loss_MSE + loss_KLD


def train(epoch):
    model.train()
    train_loss = 0

    for batch_idx, data in enumerate(trainloader):
        optimizer.zero_grad()
        mu, logvar = model.encode(data['x'])
        z = model.reparameterize(mu, logvar)
        xhat = model.decode(z)
        loss = vae_loss(xhat, data['x'], mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print('====> Epoch: {} Average loss: {:.4f}'.format(
        epoch, train_loss / len(trainloader.dataset)))

def calculate_accuracy(true_labels, distances, num_classes, k_neighbors
    =1):
    temp_predictions = []
```

```
135
136     # Loop through each data point
137     for sample_idx in range(distances.shape[0]):
138         sample_distances = distances[sample_idx, :].copy()
139         class_votes = np.zeros(num_classes)
140         class_total_distances = np.zeros(num_classes)
141
142         # Find the k nearest neighbors
143         for _ in range(k_neighbors):
144             nearest_idx = np.argmin(sample_distances)
145             nearest_class = true_labels[nearest_idx] - 1
146             class_votes[nearest_class] += 1
147             class_total_distances[nearest_class] += sample_distances[
                    nearest_idx]
148             sample_distances[nearest_idx] = np.inf  # Exclude this
                    index from future searches
149
150
151         # Determine the predicted class
152
153         avg_class_distances = class_total_distances / np.max(
                class_votes)
154         zero_indices = np.where(avg_class_distances ==0)
155         avg_class_distances[zero_indices] = np.inf
156         predicted_class = np.argmin(avg_class_distances) + 1
157         temp_predictions.append(predicted_class)
158
159     # Convert predictions to a numpy array
160     temp_predictions = np.array(temp_predictions, dtype=int).squeeze()
161
162     # Calculate accuracy
163     matches = (true_labels == temp_predictions).astype(int)
164     correct_predictions = np.sum(matches)
165     accuracy_percentage = (correct_predictions / true_labels.shape[0])
            * 100
166
167     return accuracy_percentage
168 ##################################
169 # Change these
170 p = 3  # [3, 8, 16]
171 training = False
172 TRAIN_DATA_PATH = 'FaceRecognition/train'
173 EVAL_DATA_PATH = 'FaceRecognition/test'
174 LOAD_PATH = f'weights/model_{p}.pt'
175 OUT_PATH = 'autoencoderOutput'
176 ##################################
177
178 model = Autoencoder(p)
179
180 if training:
181     epochs = 100
182     log_interval = 1
183     trainloader = DataLoader(
184         dataset=DataBuilder(TRAIN_DATA_PATH),
185         batch_size=12,
186         shuffle=True,
187     )
188     optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

```
    vae_loss = VaeLoss()
    for epoch in range(1, epochs + 1):
        train(epoch)
    torch.save(model.state_dict(), os.path.join(OUT_PATH, f'model_{p}.pt'))
else:
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH),
        batch_size=1,
    )
    model.load_state_dict(torch.load(LOAD_PATH))
    model.eval()

    X_train, y_train = [], []
    for batch_idx, data in enumerate(trainloader):
        mu, logvar = model.encode(data['x'])
        z = mu.detach().cpu().numpy().flatten()
        X_train.append(z)
        y_train.append(data['y'].item())
    X_train = np.stack(X_train)
    y_train = np.array(y_train)

    testloader = DataLoader(
        dataset=DataBuilder(EVAL_DATA_PATH),
        batch_size=1,
    )
    X_test, y_test = [], []
    for batch_idx, data in enumerate(testloader):
        mu, logvar = model.encode(data['x'])
        z = mu.detach().cpu().numpy().flatten()
        X_test.append(z)
        y_test.append(data['y'].item())
    X_test = np.stack(X_test)
    y_test = np.array(y_test)

    ###################################
    # Your code starts here
    samples = X_test.shape[0]
    current_distance = np.zeros((samples, samples))
    for i in range(samples):
        squared_diff = np.square(X_train - X_test[i, :])
        ith_distance = np.sqrt(np.sum(squared_diff, axis=1)).T
        current_distance[i,:] = ith_distance

    #Calculate the accuracy
    #How many images per class
    imagesPerClass = 21
    total_accuracy = calculate_accuracy(y_test, current_distance, 30, 1)

    print ("With P = %d. Accuracy = %.4f" % (p, total_accuracy))
    ###################################



def plot_umap_embeddings(X_train, y_train, X_test, y_test, y_test_pred, p):
    """
```

```
    Function to plot UMAP embeddings for training and test data.
    """
    # Initialize UMAP reducer
    reducer = umap.UMAP(n_components=2, random_state=42)

    # Fit UMAP on training data
    X_train_umap = reducer.fit_transform(X_train)

    # Transform test data
    X_test_umap = reducer.transform(X_test)

    # Plot training data
    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(
        X_train_umap[:, 0],
        X_train_umap[:, 1],
        c=y_train,
        cmap="jet",
        s=15,
        alpha=0.8,
    )
    plt.title(f"UMAP Embedding of Training Data (P = {p})")
    plt.colorbar(scatter, label="Ground Truth Labels")
    plt.xlabel("UMAP Dimension 1")
    plt.ylabel("UMAP Dimension 2")
    plt.grid()
    plt.show()

    # Plot test data
    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(
        X_test_umap[:, 0],
        X_test_umap[:, 1],
        c=y_test_pred,
        cmap="jet",#"tab10",
        s=15,
        alpha=0.8,
    )
    plt.title(f"UMAP Embedding of Test Data (P = {p})")
    plt.colorbar(scatter, label="Predicted Labels")
    plt.xlabel("UMAP Dimension 1")
    plt.ylabel("UMAP Dimension 2")
    plt.grid()
    plt.show()


if not training:
    # UMAP and Visualization
    knn_classifier = KNeighborsClassifier(n_neighbors=1)
    knn_classifier.fit(X_train, y_train)
    y_test_pred = knn_classifier.predict(X_test)

    # Generate UMAP plots
    plot_umap_embeddings(X_train, y_train, X_test, y_test, y_test_pred,
        p)
```