ECE 661 Homework 10

Michael Goldberg

December 6 2024

1 Theoretical Question

1.1 Overfitting

The ultimate goal of a machine learning algorithm is to find the solution that reduces the error the most across the training set yet generalizes well to the testing set. Overfitting refers to a found solution that cannot generalize well to the test set, as it has become overly reliant on the structure of the training set. For example, given an upwardly-trending set of points on a 2D scatter plot, one can draw a best-fit line through the data - this would be the desired solution, such that any test point can be predicted with the best-fit line. However, the overfit solution would be a high-degree polynomial that intersects all of the scatter points - while this provides us with very low training error, it cannot generalize to test data that lies outside of the polynomial.

1.2 Reparametrization Trick

I understand this in the following manner. The encoder maps the input data to a latent space defined by a mean and standard deviation that describes a probability distribution for the latent variable. However, this latent variable is sampled from the output of the encoder. This sampling makes backpropagation difficult, as a process with randomness like this is nondifferentiable.

Therefore, the trick is to introduce the latent variable as:

$$x = \mu + \epsilon\sigma \tag{1}$$

where μ is the mean, ϵ is a randomly sampled value between 0 and 1, and σ is the standard deviation. This allows us to still have a randomly sampled process, while remaining differentiable with respect to μ and σ for backpropagation.

2 Programming Tasks

The following task was to perform face recognition by using the PCA and LDA algorithms on a labelled dataset.

PCA I designed my own implementation of PCA and LDA. Let's start with PCA. The first step was to load in each image and vectorize them by flattening them into a single dimension. The mean of these vectors were found and subtracted from each value, transforming the vector to be zero-mean. Then, the data was normalized.

Once this was done, the goal of PCA is to reduce the dimensionality of the feature vector to a lower-dimensional space. This allows for rich encoding of the features for use within a classifier, ensuring computational inexpensiveness. Therefore, PCA suggests to encode the image vectors by using the eigenvectors that correspond to the p-largest eigenvalues in the eigendecomposition of the covariance matrix of the input data. However, the covariance matrix of the input data is incredibly large, yet a computational trick can be performed for less expensive eigendecomposition. Let's say the collection of all of the input images is in a data matrix called X, which is of size $N \times C$, where N is the number of pixels in the input image (in my case 64x64 = 16384), and C is the number of images (in my case 630). The following submatrix can be computed:

$$s = X^T * X \tag{2}$$

, which is of size $C \times C$, which is much smaller than the covariance matrix of size $N \times N$. We can then perform eigendecomposition on s, and find the p-largest eigenvectors, called v_s. However, these are within the submatrix space, and are not true eigenvectors of the covariance matrix. Therefore, we must map them back using:

$$v = X * v_s \tag{3}$$

Then, I normalized these true eigenvectors to form the feature vectors for the image.

2.1 LDA

LDA was more intensive, as this requires a process that uses the between- and within-class scatter matrices. First, the dimension of the input data is first reduced by running PCA to get the PCA eigenvectors, and mapping the input data to the PCA space through:

$$X_r = v * X \tag{4}$$

where v is the PCA eigenvectors, X is the input data, and X_r is the dimensionally reduced data. This then allows us to compute within-class and between-class variance using the following formulas:

Between-class variance:

$$S_B = \frac{1}{|M|} \sum_{i=1}^{|M|} (m_i - m)(m_i - m)^T \tag{5}$$

where M is the total number of unique faces within the training data, m_i is the single class mean and m is the global mean of the reduced data.

Within-class variance:

$$S_W = \frac{1}{|M|} \sum_{i=1}^{|M|} \frac{1}{|M_i|} \sum_{k=1}^{|M_i|} (x_k^i - m_i)(x_k^i - m_i)^T \tag{6}$$

where x_k^i is the kth image vector in the ith class.

We now want to solve the generalized eigenvalue problem, producing eigenvectors that satisfy the equation

$$S_B * v = \lambda * S_w * v \tag{7}$$

With these eigenvectors determined, we can then again collect the p-largest eigenvectors and treat that as our feature vector. Below is a comparison between the PCA and LDA methods for facial detection on the test set. First, we want to see how well these perform given different choices of p. The following plots show this relationship:

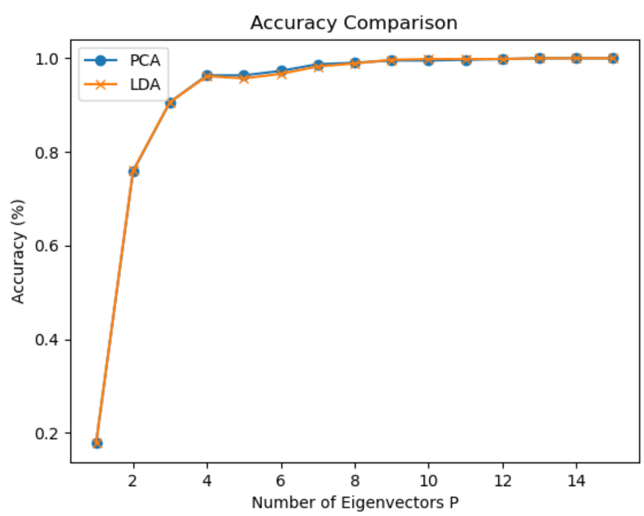
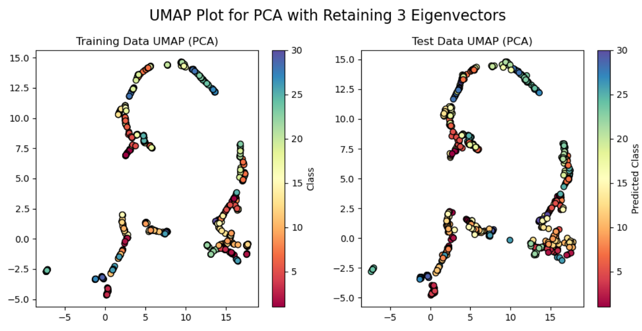


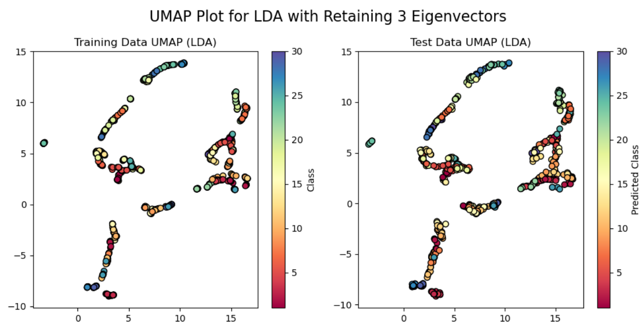
Figure 1: PCA and LDA Acuracy as a Function of p

It can be seen that both perform very well, and reach 100 accuracy when p becomes greater than 10. However, there are small mismatches between the two, with PCA generally performing better for this dataset.

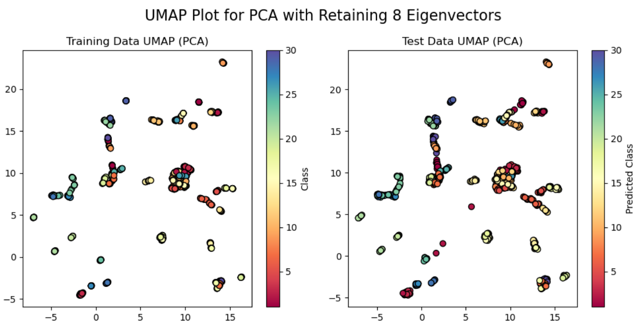
Next, let's look at how the different encodings look when projected to a 2D space for visualization:



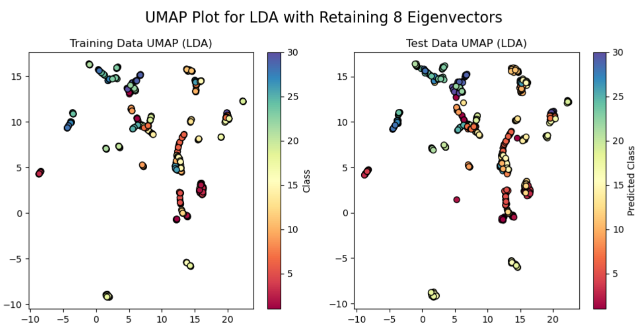
(a) PCA UMAP, p=3



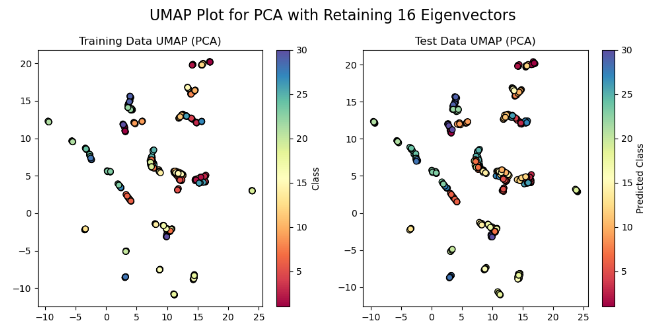
(b) LDA UMAP, p=3



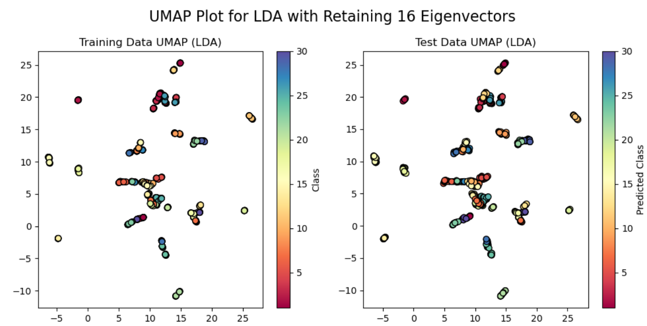
(a) PCA UMAP, p=8



(b) LDA UMAP, p=8



(a) PCA UMAP, p=16



(b) LDA UMAP, p=16

We can clearly see that the different strategies encode the rich features differently for the same value of p and same input data. However, when p increases to larger values, the 2D representations of the manifolds tend to converge to look more and more similar between the two approaches.

2.2 Autoencoder

Here, I used a variational autoencoder (AE) to encode the input data. I wanted to compare performance with PCA and LDA to see if this machine-learning-based approach had richer embeddings.

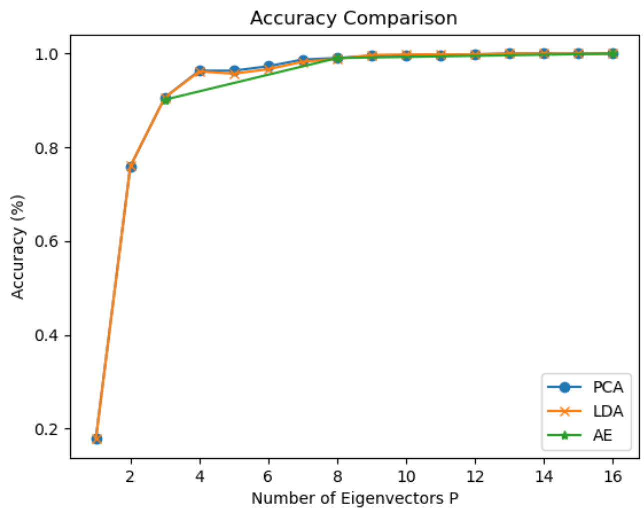
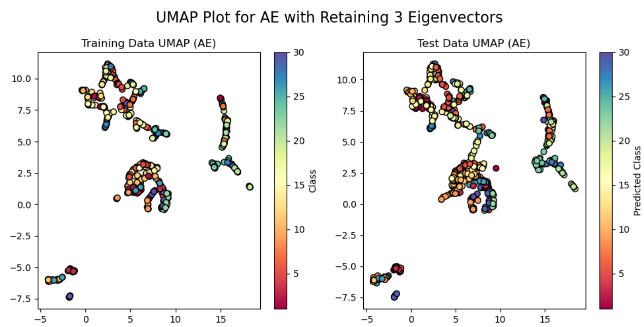
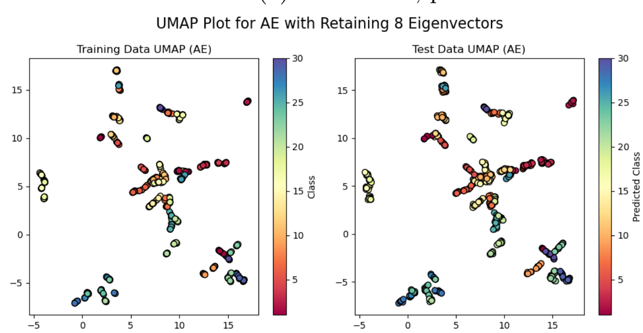


Figure 5: PCA and LDA and AE Accuracy as a Function of p

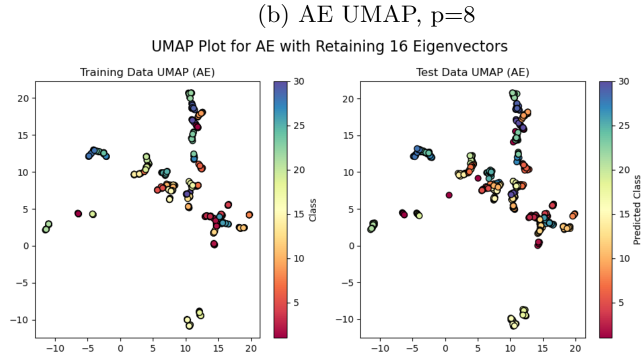
It can be seen that all three approaches match very closely to one another in terms of classification accuracy, with all tending towards 100 accuracy by the point that $p = 10$. Below are the UMAP projections for the AE approach at the same values of p (3, 8, 16):



(a) AE UMAP, p=3



(b) AE UMAP, p=8



(c) AE UMAP, p=16

It can again be seen that the autoencoder has a vastly different projection than the other two methods, yet as p tends to larger values, it converges to look more similar to PCA and LDA UMAP representations.

2.3 AdaBoost Cascade Classifier

We now are looking at a task that correctly identifies if an image is of a car or not. We used a cascade classifier built from weak adaboost classifiers to form an overall strong classifier. My approach was as follows:

First, I transformed the input images into integral images, and passed them through a vertical and horizontal haar feature detector to extract small vertical and horiontal features, reducing the dimensionality of the dataset for training. These features were then fed into a cascade classifier of multiple stages. Each stage, an adaboost classifier was trained on the data. Any particular feature vector was classified as true positive, true negative, false positive, and false negative determined from learned weights and learned threshold values. These threshold values were updated on each iteration of the adaboost training loop to discriminate more between positive and negative classification.

The cascade then determined how many samples from the input data pass the current stage, and only kept those for the next iteration. This repeats for a specified number of stages, until a strong classifier is built. The test data could then be passed through this final classifier and the false positive and false negative detection rates could be determined. Here are the results for a 2-stage classifier:

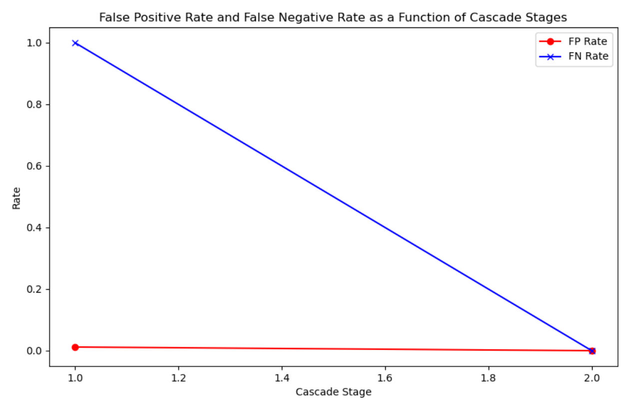


Figure 7: Cascaded Adaboost Classifier False Positive and False Negative Rate

3 Code

```
import os
import numpy as np
import cv2
import matplotlib.pyplot as plt
import time
import scipy
import umap

from sklearn.metrics import accuracy_score
from autoencoder import get_data


base_directory = 'FaceRecognition/'
```

```python
train_directory = base_directory + 'train'
test_directory = base_directory + 'test'


def load_and_vectorize(directory):
    image_vectors = []
    normalized_vectors = []
    labels = []

    for filename in os.listdir(directory):
        if filename.endswith(".png"):
            # Read in grayscale image
            image_path = os.path.join(directory, filename)
            image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

            # Vectorizing image and collecting label
            image_vector = image.flatten()
            image_vectors.append(image_vector)

            labels.append(int(filename.split("_")[0]))

    # Normalization and centering to be zero-mean
    image_vectors = np.array(image_vectors)
    mean = np.mean(image_vectors, axis=0)
    centered_data = image_vectors - mean

    normalized_data = centered_data / np.linalg.norm(centered_data,
     axis=1, keepdims=True)

    return normalized_data.T, labels

def pca(data, p):
    # Compute submatrix, computational trick
    submatrix = data.T @ data

    # Eignedecomposition of the submatrix
    sub_eigenvalues, sub_eigenvectors = np.linalg.eigh(submatrix)

    # Sort in descending order, choose eigenvectors that correspond
     to the p-largest eigenvalues
    descending_sorted_indices = np.argsort(sub_eigenvalues)[::-1]
    sub_eigenvectors = sub_eigenvectors[:,
    descending_sorted_indices][:, :p]

    # Mapping back to be true eigenvectors of the covariance matrix
     + nnormalization
    eigenvectors = data @ sub_eigenvectors
    eigenvectors = eigenvectors / np.linalg.norm(eigenvectors, axis
    =0)

    return eigenvectors

def lda(data, labels, p):
    # Perform PCA for dimensionality reduction
    pca_eigenvectors = pca(data, p)
    reduced_data = pca_eigenvectors.T @ data

```

```python
    # Compute within-class and between-class scatter matrices in PCA space

    # Calculating within-class and between-class scatter matrices
    S_W_reduced = np.zeros((reduced_data.shape[0], reduced_data.shape[0]))
    S_B_reduced = np.zeros((reduced_data.shape[0], reduced_data.shape[0]))


    unique_classes = np.unique(labels)
    overall_mean = np.mean(reduced_data, axis=1)

    # Calculation of within-class and between-class scatter
    for class_ in unique_classes:
        class_data = reduced_data[:, np.array(labels) == class_]
        class_mean = np.mean(class_data, axis=1)
        class_scatter = (class_data - class_mean[:, np.newaxis]) @ (class_data - class_mean[:, np.newaxis]).T
        S_W_reduced += class_scatter

        mean_difference = (class_mean - overall_mean).reshape(-1, 1)
        S_B_reduced += class_data.shape[1] * (mean_difference @ mean_difference.T)


    # Eigendecomposition and keeping the eigenvectors corresponding to the p-largest eigenvalues
    lda_eigenvalues, lda_eigenvectors = scipy.linalg.eigh(S_B_reduced, S_W_reduced + np.eye(S_W_reduced.shape[0]) * 1e-6)
    descending_sorted_indices = np.argsort(lda_eigenvalues)[::-1]
    lda_eigenvectors = lda_eigenvectors[:, descending_sorted_indices][:, :p]

    # Step 4: Map LDA eigenvectors back to original space + normalization
    lda_eigenvectors = pca_eigenvectors @ lda_eigenvectors
    lda_eigenvectors = lda_eigenvectors / np.linalg.norm(lda_eigenvectors, axis=0)


    return lda_eigenvectors

def project_to_subspace(data, pca_feature_set):
    return pca_feature_set.T @ data

def nearest_neighbor(train_projected, train_labels, test_projected):
    predictions = []

    for test_sample in test_projected.T:
        # Calculate L2 norm to all training samples
        distances = np.linalg.norm(train_projected.T - test_sample, axis=1)

        # Find the index of the closest training sample
```

```
        nearest_index = np.argmin(distances)

        # Assign label
        predictions.append(train_labels[nearest_index])

    return predictions


def plot_umap(train_data, train_labels, test_data, test_labels,
    predicted_labels, method, p):
    # Apply UMAP to reduce the data to 2D
    reducer = umap.UMAP(n_components=2, random_state=42)

    # Apply UMAP on training data
    train_umap = reducer.fit_transform(train_data.T)
    # Apply UMAP on test data
    test_umap = reducer.transform(test_data.T)

    plt.figure(figsize=(10, 5))

    # Plot training data
    plt.subplot(1, 2, 1)
    plt.scatter(train_umap[:, 0], train_umap[:, 1], c=train_labels,
     cmap='Spectral', edgecolors='k', s=40)
    plt.title(f"Training Data UMAP ({method})")
    plt.colorbar(label="Class")

    # Plot test data
    plt.subplot(1, 2, 2)
    plt.scatter(test_umap[:, 0], test_umap[:, 1], c=
    predicted_labels, cmap='Spectral', edgecolors='k', s=40)
    plt.title(f"Test Data UMAP ({method})")
    plt.colorbar(label="Predicted Class")

    plt.suptitle(f"UMAP Plot for {method} with Retaining {p}
    Eigenvectors", fontsize=16)
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    train_data, train_labels = load_and_vectorize(train_directory)
    test_data, test_labels = load_and_vectorize(test_directory)

    print("train data shape: ", train_data.shape)

    time.sleep(100)

    p_set = range(1, 17)
    ae_p_set = [3, 8, 16]
    pca_accuracies = []
    lda_accuracies = []
    ae_accuracies = []

    plot_umaps = True

    for p in p_set:
```

```python

    # Perform PCA

    # Collect feature set
    pca_feature_set = pca(train_data, p)
    # Project to PCA subspace
    pca_train_feature_vector = project_to_subspace(train_data, pca_feature_set)
    pca_test_feature_vector = project_to_subspace(test_data, pca_feature_set)
    # print("lda_train_feature_vector shape: ", pca_train_feature_vector.shape)
    # Predict labels with nearest neighbor algorithm
    pca_predicted_labels = nearest_neighbor(pca_train_feature_vector, train_labels, pca_test_feature_vector)
    # Calculate accuracy through #correct_predictions / #total_images
    pca_accuracy = accuracy_score(test_labels, pca_predicted_labels)


    # Perform LDA
    # Collect feature set
    lda_feature_set = lda(train_data, train_labels, p)
    # Project to LDA subspace
    lda_train_feature_vector = project_to_subspace(train_data, lda_feature_set)
    lda_test_feature_vector = project_to_subspace(test_data, lda_feature_set)
    # Predict labels with nearest neighbor algorithm
    lda_predicted_labels = nearest_neighbor(lda_train_feature_vector, train_labels, lda_test_feature_vector)
    # Calculate accuracy through #correct_predictions / #total_images
    lda_accuracy = accuracy_score(test_labels, lda_predicted_labels)

    pca_accuracies.append(pca_accuracy)
    lda_accuracies.append(lda_accuracy)

    print(f"p = {p}: PCA Accuracy = {pca_accuracy:.2%}, LDA Accuracy = {lda_accuracy:.2%}")

    if (p == 3 or p == 8 or p == 16):
        # Load autoencoder vectors and labels
        ae_train_feature_vector, ae_train_labels, ae_test_feature_vector, ae_test_labels = get_data(training=False, p=p)
        # Converting to be of shape (p, C) rather than (C, p)
        ae_train_feature_vector = ae_train_feature_vector.T
        ae_test_feature_vector = ae_test_feature_vector.T

        # Use nearest neighbors to predict test labels with autoencoder embeddings
```

```python
            ae_predicted_labels = nearest_neighbor(ae_train_feature_vector, ae_train_labels, ae_test_feature_vector)
            # Calculate autoencoder accuracy
            ae_accuracy = accuracy_score(test_labels, ae_predicted_labels)
            ae_accuracies.append(ae_accuracy)

            print(f"p = {p}: AE Accuracy = {ae_accuracy:.2%}")

            if(plot_umaps):
                plot_umap(pca_train_feature_vector, train_labels, pca_test_feature_vector, test_labels, pca_predicted_labels, "PCA", p)
                plot_umap(lda_train_feature_vector, train_labels, lda_test_feature_vector, test_labels, lda_predicted_labels, "LDA", p)
                plot_umap(ae_train_feature_vector, ae_train_labels, ae_test_feature_vector, ae_test_labels, ae_predicted_labels, "AE", p)

    plt.plot(p_set, pca_accuracies, "-o", label="PCA")
    plt.plot(p_set, lda_accuracies, "-x", label="LDA")
    plt.plot(ae_p_set, ae_accuracies, "-*", label="AE")
    plt.title("Accuracy Comparison")
    plt.xlabel("Number of Eigenvectors P")
    plt.ylabel("Accuracy (%)")
    plt.legend()
    plt.show()
```

Listing 1: PCA, LDA, and Autoencoder Classification

```python
import os

import numpy as np
import torch
from torch import nn, optim
from PIL import Image
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms


class DataBuilder(Dataset):
    def __init__(self, path):
        self.path = path
        self.image_list = [f for f in os.listdir(path) if f.endswith('.png')]
        self.label_list = [int(f.split('_')[0]) for f in self.image_list]
        self.len = len(self.image_list)
        self.aug = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
        ])

    def __getitem__(self, index):
```

```python
        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)
        )
```

```python

    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 get_data(training=False, p=3):
    #################################
    TRAIN_DATA_PATH = 'FaceRecognition/train'
    EVAL_DATA_PATH = 'FaceRecognition/test'
```

```python
    LOAD_PATH = f'weights/model_{p}.pt'
    OUT_PATH = LOAD_PATH
    ###################################

    model = Autoencoder(p)

    if training:
        epochs = 100
        log_interval = 1
        trainloader = DataLoader(
            dataset=DataBuilder(TRAIN_DATA_PATH),
            batch_size=12,
            shuffle=True,
        )
        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, weights_only=True))
        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)

        return X_train, y_train, X_test, y_test
```

Listing 2: Autoencoder Helper Script

```python
import os
import cv2
```

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

num_iterations = 20

class WeakClassifier:
    def __init__(self, feature, threshold, polarity):
        self.feature = feature
        self.threshold = threshold
        self.polarity = polarity

    def predict(self, features):
        feature_value = features[self.feature]
        return 1 if (self.polarity == 1 and feature_value >= self.threshold) or (self.polarity == -1 and feature_value < self.threshold) else -1


class AdaBoost:
    def __init__(self, T):
        self.T = T
        self.alphas = []
        self.classifiers = []

    def fit(self, X, y):
        w = np.ones(len(X)) / len(X)  # Initial weight for each sample

        for t in tqdm(range(self.T), desc="Training AdaBoost", ncols=100):
            best_classifier = None
            min_error = float('inf')

            for feature_index in range(len(X[0])):
                # Sorting features
                feature_values = [features[feature_index] for features in X]
                sorted_indices = np.argsort(feature_values)
                sorted_features = np.array(feature_values)[sorted_indices]
                sorted_weights = w[sorted_indices]
                sorted_labels = np.array(y)[sorted_indices]

                # Calculating next threshold value
                T_plus = np.sum(w * (y == 1))
                T_minus = np.sum(w * (y == -1))

                for i in range(1, len(X)):
                    threshold = (sorted_features[i - 1] + sorted_features[i]) / 2
                    S_plus = np.sum(sorted_weights[:i] * (sorted_labels[:i] == 1))
                    S_minus = np.sum(sorted_weights[:i] * (sorted_labels[:i] == -1))

                    error_pos_1 = S_plus + (T_minus - S_minus)
```

```
                    error_neg_1 = S_minus + (T_plus - S_plus)

                    # Choosing minimum between two error metrics
                    error = min(error_pos_1, error_neg_1)

                    if error < min_error:
                        min_error = error
                        best_classifier = WeakClassifier(feature_index, threshold, 1)

            # Find new parameters to compute next weak classifier
            alpha = 0.5 * np.log((1 - min_error) / (min_error + 1e-10))
            self.alphas.append(alpha)
            self.classifiers.append(best_classifier)

            predictions = np.array([best_classifier.predict(features) for features in X])
            w = w * np.exp(-alpha * y * predictions)
            w = w / np.sum(w)  # Normalize weights


    def predict(self, X):
        strong_preds = np.zeros(len(X))
        for alpha, classifier in zip(self.alphas, self.classifiers):
            predictions = np.array([classifier.predict(features) for features in X])
            strong_preds += alpha * predictions
        return np.sign(strong_preds)


class CascadeClassifier:
    def __init__(self, false_positive_target, true_detection_target, num_stages):
        self.false_positive_target = false_positive_target
        self.true_detection_target = true_detection_target
        self.num_stages = num_stages
        self.stages = []
        self.fp_rates = []
        self.fn_rates = []

    def train(self, X, y):

        for stage_index in range(self.num_stages):
            print(f"Training stage {stage_index + 1}/{self.num_stages}...")
            # Perform adaboost to fit weak classifier to data
            adaboost = AdaBoost(T=num_iterations)
            adaboost.fit(X, y)
            self.stages.append(adaboost)

            # Get predictions on the current stage's data
            predictions = adaboost.predict(X)

            # Compute False Positive and False Negative rates
            fp = np.sum((predictions == 1) & (y == 0))  # Positive
```

```
classified as negative
            fn = np.sum((predictions == -1) & (y == 1))  # Negative classified as positive

            fp_rate = fp / np.sum(y == 0) if np.sum(y == 0) > 0 else 0
            fn_rate = fn / np.sum(y == 1) if np.sum(y == 1) > 0 else 0

            self.fp_rates.append(fp_rate)
            self.fn_rates.append(fn_rate)

            print(f"Stage {stage_index + 1}: FP rate = {fp_rate}, FN rate = {fn_rate}")

            # If the stage does not meet target FP and FN rates, stop training
            if fp_rate > self.false_positive_target or fn_rate < self.true_detection_target:
                print(f"Stage {stage_index + 1} did not meet target rates. Stopping early.")
                break

            # Keep only the samples that pass the current stage
            passed_indices = (predictions == 1)
            X = np.array(X)[passed_indices.astype(int)]
            y = np.array(y)[passed_indices.astype(int)]

            if len(X) == 0:
                break

        self.plot_performance()

    def predict(self, X):
        for stage in self.stages:
            predictions = stage.predict(X)
            if np.any(predictions == -1):  # Reject if any stage fails
                return -1
        return 1

    def plot_performance(self):
        # Plotting FP and FN rates
        stages = np.arange(1, len(self.fp_rates) + 1)

        plt.figure(figsize=(10, 6))

        # Plot FP rate
        plt.plot(stages, self.fp_rates, label="FP Rate", color="red", marker='o')
        # Plot FN rate
        plt.plot(stages, self.fn_rates, label="FN Rate", color="blue", marker='x')

        # Labels and title
        plt.xlabel("Cascade Stage")
        plt.ylabel("Rate")
```

```
        plt.title("False Positive Rate and False Negative Rate as a Function of Cascade Stages")

        # Show a legend
        plt.legend()

        # Show the plot
        plt.show()


def extract_haar_features(integral_image):
    img_height, img_width = integral_image.shape
    features = []

    # Horizontal 1x2 feature
    for y in range(img_height):
        for x in range(img_width - 1):
            left = integral_image[y, x]
            right = integral_image[y, x + 1]
            horizontal_feature = right - left
            features.append(horizontal_feature)

    # Vertical 2x1 feature
    for y in range(img_height - 1):
        for x in range(img_width):
            top = integral_image[y, x]
            bottom = integral_image[y + 1, x]
            vertical_feature = bottom - top
            features.append(vertical_feature)

    return np.array(features)


def compute_integral_image(image):
    return image.cumsum(axis=0).cumsum(axis=1)


def load_data(positive_dir, negative_dir):
    images = []
    labels = []

    # Loading data and labels
    for filename in os.listdir(positive_dir):
        img = cv2.imread(os.path.join(positive_dir, filename), cv2.IMREAD_GRAYSCALE)
        images.append(img)
        labels.append(1)

    for filename in os.listdir(negative_dir):
        img = cv2.imread(os.path.join(negative_dir, filename), cv2.IMREAD_GRAYSCALE)
        images.append(img)
        labels.append(0)

    images = np.array(images)
    labels = np.array(labels)

```

```python
    return images, labels


# Load training data
train_dir = "CarDetection/train"
positive_train_dir = os.path.join(train_dir, "positive")
negative_train_dir = os.path.join(train_dir, "negative")

positive_train_images, positive_train_labels = load_data(
    positive_train_dir, negative_train_dir)

train_integral_images = [compute_integral_image(img) for img in
    positive_train_images]
train_features = [extract_haar_features(integral_image) for
    integral_image in train_integral_images]

# Train cascade classifier
cascade_classifier = CascadeClassifier(false_positive_target=0.1,
    true_detection_target=0.9, num_stages=5)
cascade_classifier.train(train_features, positive_train_labels)

# Test the classifier
test_dir = "CarDetection/test"
positive_test_dir = os.path.join(test_dir, "positive")
negative_test_dir = os.path.join(test_dir, "negative")

positive_test_images, positive_test_labels = load_data(
    positive_test_dir, negative_test_dir)

test_integral_images = [compute_integral_image(img) for img in
    positive_test_images]
test_features = [extract_haar_features(integral_image) for
    integral_image in test_integral_images]

test_predictions = cascade_classifier.predict(test_features)
# test_predictions = [cascade_classifier.predict(features) for
    features in test_features]
accuracy = np.mean(np.array(test_predictions) == np.array(
    positive_test_labels))
print(f"Cascade Classifier Test Accuracy: {accuracy * 100:.2f}%")
```

Listing 3: Cascaded Adaboost Classifier