ECE 66100 Homework #10

by

Adrien Dubois (dubois6@purdue.edu)

December 6, 2024

Contents

1 Theory Questions:

1.1 Question 1: Overfitting to training data

What is overfitting to training data and why is it important to control. My understanding of overfitting is when the model starts to memorize the training data compared to learning the overall distribution. This can often occur because the training dataset is too small, or not representative of the overall distrubtion of the data. In this way, it is very important to consistently measure both the training loss and evaluation loss. Comparing these two can give a good indication of when to stop training the model so that it does not overfit to the training data, since the true accuracy measurement should always be performed on a significant amount (at least 10% of the total dataset) of unseen testing samples.

1.2 Question 2: Reparametrization Trick

How do you understand the reparametrization trick used in variational autoencoders. The reparametrization trick is required when using variational autoencoders since we cannot backprogate through the distribution parametrized by the (μ, σ) parameters. Instead, we sample $\boldsymbol{z}$ from a standard normal distrubtion with $(\mu = 0, \sigma = 1)$ and compute the latent space variable as:

$$z = \mu_{learned} + \sigma_{learned} \times \boldsymbol{z}$$

This seperates the randomness in the latent space from the learned parameters, so that we can backpropogate solely throug hthe deterministic parameters, instead of not being able to backpropogate through a randomly sampled distribution. This is required because the sampling operation is non-differentiable. So, with this reparametrization trick, backpropogation of the gradients is possible and we can use the reconstruction and KL divergence losses to learn the best parameters.

2 PCA:

The process for principle component analysis is as follows:

- Vectorize each black and white image such that it goes from a shape of (H, W, 1) array to $(H \times W)$.
- We then normalize the vector by dividing each image vector by its L2 Norm.

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

- We can then compute the global mean from each image and substract it from the image vector:

$$\vec{m} = \frac{1}{N} \sum_{i=1}^{N} \vec{x_i}$$

$$X = [x_0 - \vec{m} \| x_1 - \vec{m} \| ... \| x_N - \vec{m}]$$

- Next, we know that the covariance matrix is $C = X^T X$. However, for large image vectors, with matrix would be much too large when calculating the eigen decomposition of C. Therefore, we estime this matrix from a smaller covariance matrix: $C = XX^T$.
- Decompose the C matrix into its eigevalues λ, and eigenvectors $\boldsymbol{v}$.
- Retain the top $\boldsymbol{p}$ eigenvectors according to the largest eigenvalues. This allows us to create our projection matrix. Note that since np.linalg.eig already sorts the eigenvectors for you, I return the top p eigenvectors through indexing in the $\boldsymbol{v}$ array from the end.

$$W_p = X^T \times [\boldsymbol{v_{-p}} \| \boldsymbol{v_{-p+1}} \| ... \| \boldsymbol{v_{-1}}]$$

- Normalize the projection matrix:

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

- Lastly, our feature vectors are calculated as follows:

$$y_i = X \times \hat{W_p}$$

Then we train a classifer using the PCA embeddings for each image. I did this by storing each embedding vector and its corresponding class into an array. I can then search for the closest embedding to a probe test image PCA embedding and assign it the class of that nearby train embedding. This technique was utilized for all such testing in PCA, LDA and autoencoders for this report.

3 LDA:

The idea behind linear discriminant analysis is to find the directions in the underlying latent space that maximally discriminate between the classes. This is done through two metrics: the within and between class class scatters. We can calculate the p dimensional LDA embedding as follows:

- Compute the overall mean of all image vectors:

$$\vec{m} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

- Compute the per-class means:

$$\vec{m_c} = \frac{1}{N_c}\sum_{i=1}^{N_c} x_i^c$$

- The between class scatter is defined as:

$$S_B = \frac{1}{N}\sum_{i=1}^{N}\{(\mathbf{m}_c - \mathbf{m})(\mathbf{m}_c - \mathbf{m})^T\}$$

- On the other hand, the within class scatter is defined as:

$$S_W = \frac{1}{N}\sum_{c=1}^{N}\frac{1}{|N_c|}\sum_{k=1}^{|N_c|}\{(\mathbf{x}_k^c - \mathbf{m}_c)(\mathbf{x}_k^c - \mathbf{m}_c)^T\}$$

- It is important to note for debugging purposes that both the S_W and S_B matrices are of shape: (feature dimension, feature dimension)
- Our goal with LDA is to maximize the Fischer Discriminant function:

$$J(\mathbf{w}) = \frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

- Instead of direclty optimizing over this cost, it is equivalent to instead find the few largest eigenvalues of the matrix $S_W^{-1} S_B$. However, there is a high probability that the S_W matrix is singular and therefore non-invertible. We therefore need to use Yu and Yang's approach for LDA calculation to get around this problem.
- We first calculate the eigevalues λ, and eigenvectors $\boldsymbol{v}$ of the S_B matrix.
- We then remove all eigevalues that are near 0, and their corresponding eigenvectors. The eigenvectors are also normalized columnwise.
- Next, we create a diagonal matrix of the eigenvalues D_B and calculate the low dimensional projection matrix of S_B:

$$Z = [\hat{v_0}\|\hat{v_1}\|...\|\hat{v_n}]\, D_B^{-0.5}$$

- We then require the following eigen decomposition:

$$\text{eig-values}, U = \text{eig-decomp}\left(Z^T S_W Z\right)$$

- Retain the top p columns of U and normalize these vectors.
- The projection matrix W_p is then equal to:

$$W_p = \left(\hat{U}^T Z^T\right)$$

- and the final feature vector is equal to:

$$y_i = (\hat{x}_i - \vec{m}) \times W_p$$

4 Autoencoders:

An autoencoder works through a pair of models: the encoder and decoders. The encoder's role is to project the initial image vector into a lower dimensional space, while the decoder will reproject the latent space vector to the original image dimensionality. You can then train both models together to recreate the original image values in the decoder output. This will push also push the encoder to learn the optimal latent space representation that maintains the most information of the original image so that it can be reconstructed by the decoder.

For this assignment, I have tested and demonstrate results using a latent space dimension of 3, 8 and 16. I also trained autoencoders with latent space dimensions from 1 to 16 and will be reporting accuracy for each such model, but will not include the UMAP representation, nor the confusion matrices. The accuracy for each model was also calculated using the same approach described in the PCA section.

5 Cascading AdaBoost Classifiers:

Instead of including a step by step theory based approach for explaining the Adaboost classifiers, I will use a functional/class based approach that is more closely related to the actual programming task. Please refer to other previous solutions such as 2020, solution 1 by Brian Helfrecht for a more theoreitical step by step approach without any bias to a certain programming style.

5.1 Data Preprocessing

The purpose of the data preprocessing is to take input images of shape (128, 128) into low level feature vectors. I did this as follows:

1. Convert every image to (64×64), grayscale pytorch tensors or numpy arrays.
2. We then need to apply Haar filters of sizes 0, 2, 4, 6, 8, ..., 64. For each "Haar size", I am referring to the dimension of the convolutional kernel applied when performing haar filtering. For example, a Haar size of 6 would have two kernels:

$$\text{haar\_dx} = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 \end{bmatrix}, \quad \text{haar\_dy} = \text{haar\_dx}^T$$

   We can then horizontally stack the features extracted each horizontal and vertical filter to construct a high dimenisonal feature representation of the image. It is important to note that by a Haar size of 0, I also include the actual image in the feature vector.

This pre-processing proceedure is applied to the training and testing images. It is also important to note that I create a very large vertically stacked array of all the positive samples and negative samples in each data set partition. I also randomize these arrays and their corresponding labels (1 for positive sample, -1 for negative samples) accordingly to ensure realistic training and testing.

5.2 Weak Classifier Class:

- For each img in the array of image feature vectors, we associate a weight. This weight is initialize as a uniform distribution across all samples.
- Next, for each feature vector (a column in the (num-samples, feature-dim) img array), we do the following:
  - Sort the feature column. Also sort the weight matrix, and labels according to the ascending sorting procedure on the features.
  - We then define four metrics used to qualify the classification error for both polarities. I remind you that this weak classifier works by assuming all features below a certain threshold are class 1, and the rest are class 2. So the polarity will switch whether the first class refers to the positive or negative samples. The metrics used are:
    * S^+ The cumulative sum of all weights for positive images whose value is less than the current threshold.
    * S^- The cumulative sum of all weights for negative images whose value is less than the current threshold.

    * T^{+} The sum of the weights for positive images
    * T^{-} The sum of the weights for negative images
  - We can then calculate the errors for each polarity:

$$e^{+1} = S^{+} + T^{-} - S^{-}$$

$$e^{-1} = S^{-} + T^{+} - S^{+}$$

- The four metrics above combine into the overall prediction error: $\epsilon = min\left(e^{+1}, e^{-1}\right)$
- So, depending on which error was the minimum, we associate the polarity accordingly. We also store the feature index and feature value that created the minimum error.
- This procedure is repeated for every feature vector (column) in the image matrix. The optimal values for the feature, threshold, polarity and error are what define a weak classifier.

5.3 Strong Classifier:

A strong classifier is defined by one cascade of weak classifiers.

- We first initialize one weak classifier.
- We then calculate the following parameters:

$$\beta = \frac{\epsilon}{1-\epsilon}$$

$$\alpha = ln\left(\frac{1}{\beta}\right)$$

- We then have to aclaulte our predicted labels through our classifier by thresholding the features based on what you calculated for the weak classifier. Make sure to take into account the polarity when assigning labels.
- We can then updates each weight in the weight matrix as follows, where δ equals 1 if the prediction was incorrect, and 0 if the prediction was correct. Make sure to also normalize the updated weights so that they still sum up to 1.

$$w_i = old\left(w_i\right) \times \beta^{1-\delta}$$

- Next, we need to calculate the accuracy of the weak classifiers so far. In this way, we loop through each weak classifier and use the alpha parameter as a scalar that informs us how much each classifier impacts the final result. Therefore:

$$\text{label}(x) = \sum_{t=0}^{T-1} \alpha_t \times \text{pred}_t$$

  where T is the total number of weak classifiers, and $pred_t$ is a one if $feature * polarity >= threshold * polarity$ else it is a -1.
- We can then get the sign of each output to get the final predicted label.
- Next we need to compare the predicted labels to the ground truth labels to calculate the accuracy.
- If the accuracy is 100%, we can stop adding to the cascade. Othewise, we keep adding new weak classifiers up to a user specified limit.

5.4 Strong classifier Cascade:

When one strong classifier is not enough to reduce the false positive rate to below 1%, we need to keep adding new cascades. This creates a cascade of strong classifiers.

- In this way, we first define a strong classifier.
- We then get the predicted labels in the same way that was performed for a strong classifier.
- Now, instead of calculating accuracy, we calculate the false positive rate. If this rate is less than 1% we stop our strong classifier cascade.
- If the false positive rate is not low enough, we need to update our training data. In this way, we look for correct nagtive predictions and remove them from the image and label arrays. This will help us hone in on reducing the false positive rate with a mix of positive samples, and incorrectly predicted negative samples.
- Lastly, regardless of the false positive rate, if we have removed all the incorrectly predicted negative images from the image array, or if we have reach a user defined limit, we stop adding new strong classifier cascades.

5.5 Adaboost Testing:

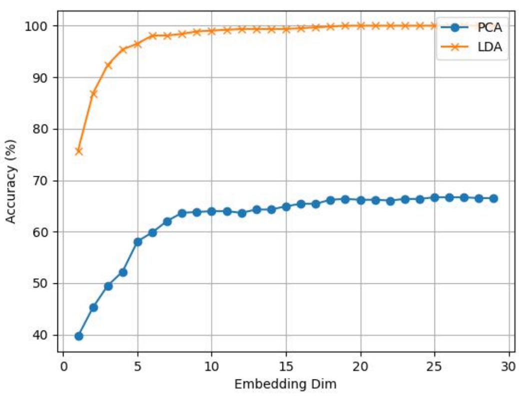
For testing Adaboost:

- I first loop through each of the cascades to calculate the predicted labels.
- For each cascade, I compute the true negative, true positive, false positive and false negative predictions and report the false positive rate and false negative rate.
- Additionally, to get the overall prediction acurracy, I keep track of the true negatives and false positives in an array, and I use a logical or operation over each cascade for both of these arrays to get the final predictions. This works because the initial predictions are only accurate on a subset of the data (due to the data removal process used during training). So we must take into account what each cascade predicts overall to get the final accuracy.

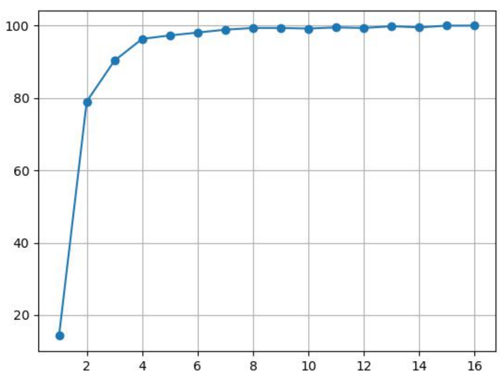
6 Results:

6.1 Classification Accuracy as a function of p:

Included below are graphs of the accuracy of each technique with respect to p, the dimension of the feature vector reduced represenation.



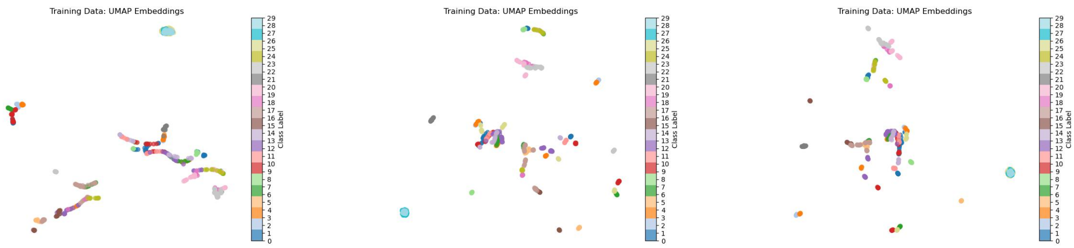
(a) PCA and LDA with Yuyang accuracy graph



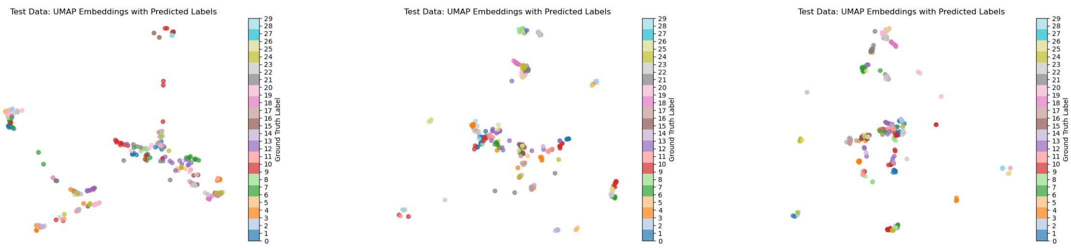
(b) Autoencoder accuracy graph

6.2 UMAP plots for PCA, LDA and Autoencoder:

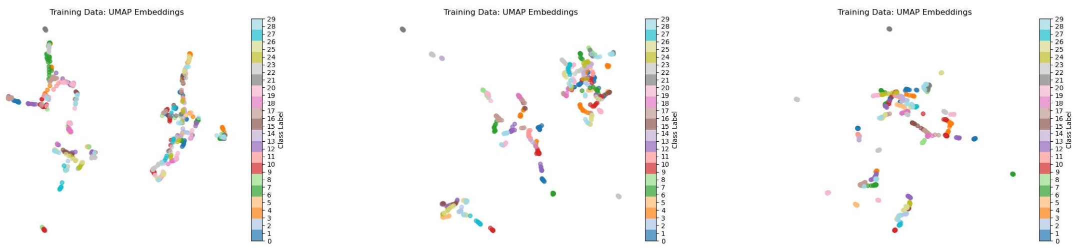
Please note that, for this section, I only plot a subset of the overall testing data in the UMAP plots due to the requirement to balance out my classes before running the UMAP fit transform. Terefore, when testing my PCA output for low values of p some classes were not present in the testing data and are therefore not included in the UMAP plot to allow other classes to be printed normally. I did this by only printing classes that have at least 11 samples in the testing data (only for done PCA).



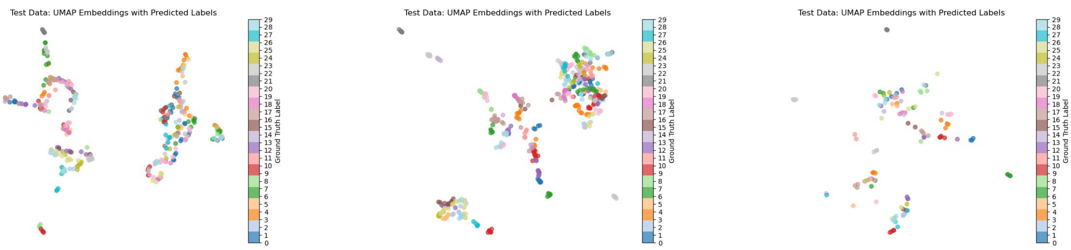
(a) PCA UMAP display of train embeddings with p=3, 8, 16



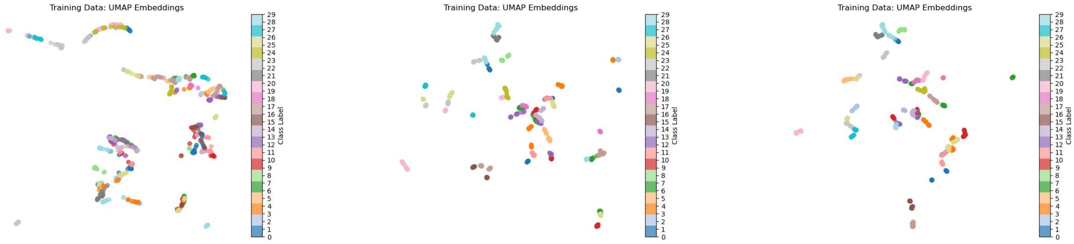
(b) PCA UMAP display of test embeddings with p=3, 8, 16



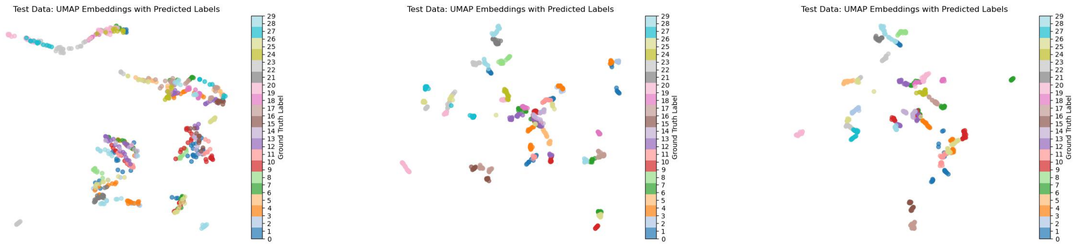
(a) LDA UMAP display of train embeddings with p=3, 8, 16



(b) LDA UMAP display of test embeddings with p=3, 8, 16

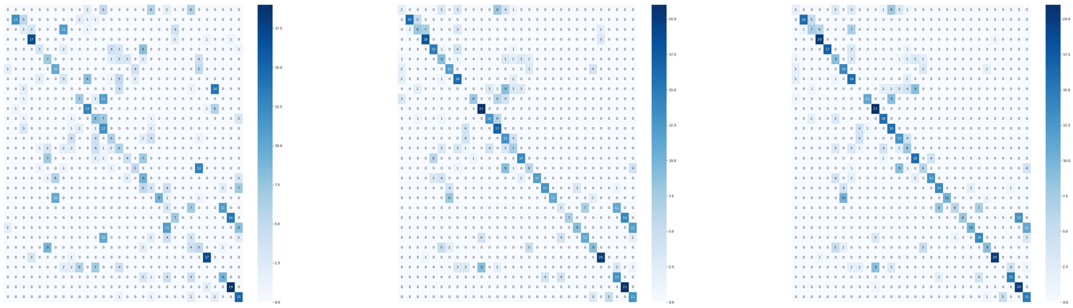


(a) Autoencoder UMAP display of train embeddings with p=3, 8, 16

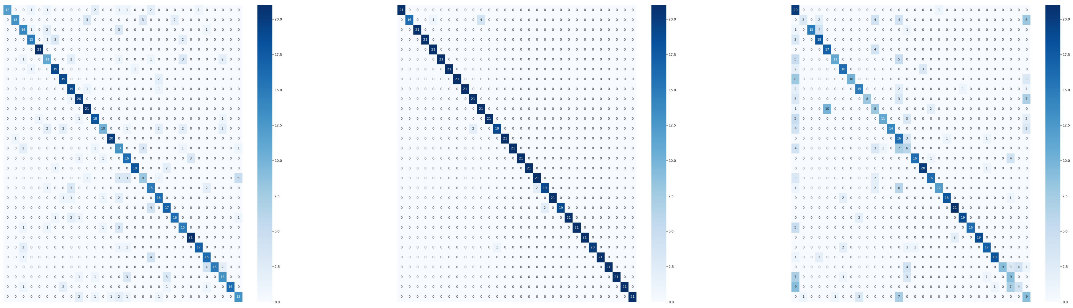


(b) Autoencoder UMAP display of test embeddings with p=3, 8, 16

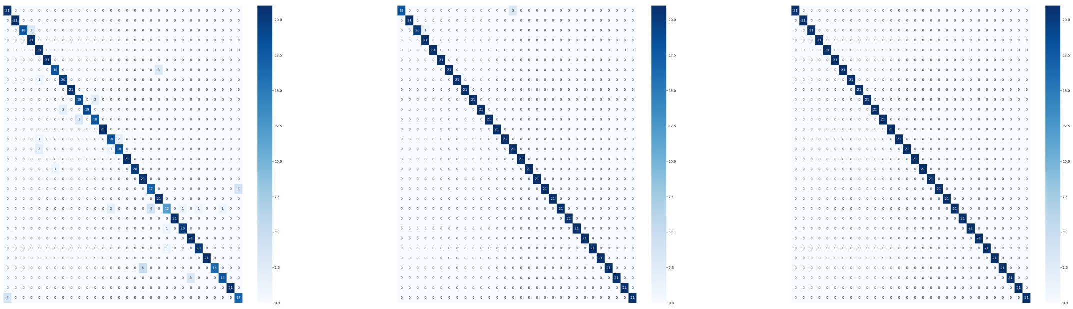
6.3 Confusion Matrix for PCA, LDA and Autoencoder:



(a) Confusion matrix of PCA with p=3, 8, 16



(b) Confusion matrix of LDA with p=3, 8, 16



(c) Confusion matrix of AutoEncoder testing with p=3, 8, 16

6.4 Comparison:

6.4.1 PCA vs LDA:

Using the computation trick, PCA was much faster than LDA with Yuyang for dimensionality reduction. This was due to the need to calculate the S_B and S_W parameters. However, as shown in the accuracy graphs, LDA had much higher accuracy than PCA. Not only did it's accuracy rise much more sharply than PCA, but it also had a higher starting point, and a higher finishing accuracy of 100%. On the other hand, PCA was much less accurate than LDA with a maximum accuracy of around 67% over the embedding dimensions analyzed. It additionally suffered to class imbalance in the testing predictions compared to LDA which was able to more accurately span all classes. I also interally compared these results to sklearn's PCA fit transform function and found that the results were consistent with mine making me believe that this issue is inherent to PCA for this dataset with 64 by 64 images.

6.4.2 Autoencoders vs PCA & LDA:

The autoencoder outperformed PCA and LDA on almost all metrics. It accuracy was much higher across the board, and the autoencoder was able to reach 100% accuracy with an embedding dimension as low as 9, while LDA required an embedding dimension of 18 to do the same. Additionally, the UMAP displayed embeddings show much greater seperation compared to PCA and LDA which would imply that the retainec features in the latent space are more disciminative compared to PCA and LDA. The only downside of the autoencoder was the need for a GPU to train the model efficiently, and the fact that the training still took much longer than LDA and PCA even when training such a small model on an Nvidia A5000 with a large batch size. Techniques such as early stopping could have been utilized to reduce the training time once the loss flattened out. Additionally, reporting a validation loss with the training loss would have helped to find the optimal stopping time before the model overfit to the training data.

6.5 AdaBoost results:

6.5.1 Training Results:

The following data are the results of training my AdaBoost cascades with up to 5 weak classifiers per cascade and up to 3 cascades. While I am aware that the instructions asked for the data to be represented graphically, I have tabulated it instead as I believe it more clearly displays the results.

Overall my results showed:

- Accuracy: **96.7045**
- Best Strong Classifier's False Negative Rate: 0.08
- Best Strong Classifier's False Positive Rate: 0.05

Cascade ID	False Negative Rate	False Positive Rate	Final Accuracy
1	0.07	0.04	0.89
2	0.01	0.08	0.91
3	0.00	0.07	0.93

Table 1: False positive rate at each cascade

Weak Classifier ID	Feature	Threshold	Polarity	Error	Alpha	Accuracy
1	6993	-1.099	-1	0.139	1.827	0.861
2	1500	0.227	-1	0.226	1.231	0.861
3	5694	-0.041	-1	0.281	0.940	0.886
4	6747	-0.015	1	0.282	0.933	0.880
5	5697	0.040	1	0.325	0.731	0.893
1	769	0.137	1	0.104	2.150	0.894
2	6347	0.095	1	0.287	0.909	0.894
3	4720	0.000	-1	0.300	0.848	0.894
4	637	0.243	1	0.302	0.840	0.913
5	4267	-0.073	-1	0.302	0.838	0.906
1	7252	-0.319	-1	0.076	2.495	0.924
2	5758	-0.054	-1	0.279	0.949	0.924
3	4718	-0.005	-1	0.299	0.850	0.924
4	133	0.243	1	0.284	0.924	0.928
5	2851	0.125	-1	0.284	0.927	0.934

Table 2: Parameters for each weak classifier in the cascade and the corresponding classification accuracy

6.5.2 Testing Results:

Overall accuracy of 96.7045%.

Cascade ID	False Negative Rate	False Positive Rate
1	0.95	0.05
2	0.13	0.87
3	0.08	0.92

Table 3: False positive rate at each cascade

As you can see, due to our training strategy of removing data after each cascade, the different cascades were specialized at seperate different types of data. In this way, my first cascade was very powerful at detecting true negatives as it already detected 417 out of the 440 true negatives through the first pass. However, the first cascade only classified 23 out of the 440 possible true positives. On the other hand, the final cascade was very strong as seperating out the remaining positive samples with 404 out of 440 positive samples detected, and much weaker at detecting negative samples. This is reflected in the shifting false positive and false negative rates as well.

7 Source Code Listing:

```
# %%
import os
import numpy as np
import torch
import umap
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

# %%
##################################
# Change these
p = 3  # [3, 8, 16]
training = False
TRAIN_DATA_PATH = '/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/train/'
EVAL_DATA_PATH = '/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/test/'
LOAD_PATH = f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/model_{p}.pt"
OUT_PATH = '/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/exp/'
##################################

# %%
class DimReducerBuilder(Dataset):
    def __init__(self, path, option):
        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)
        if option == "bw":
            self.aug = transforms.Compose([
                transforms.Resize((64, 64)),
                transforms.Grayscale(num_output_channels=1),
                transforms.ToTensor(),
            ])
        else:
            self.aug = transforms.Compose([
                transforms.Resize((64, 64)),
                transforms.ToTensor(),
            ])

    def __len__(self):
        return self.len

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

        # Flatten and normalize the image data:
```

```python
        img_vec = torch.reshape(x, shape=(x.shape[0], -1))
        img_vecn = img_vec / torch.norm(img_vec, dim=1, keepdim=True)
        # img_vecn = img_vec / img_vec.shape[1]
        return {'img_vecn': img_vecn.squeeze(), 'y': self.label_list[index]}


# %% [markdown]
# ### Task 1: PCA

# %%
from sklearn.decomposition import PCA

# %%
def get_id_of_nearest_embedding(training_set, probe_embedding, num_classes=30,
    num_samples=21):
    index_to_class_id = np.repeat(np.arange(num_classes), num_samples)
    training_set = np.reshape(training_set.copy(), newshape=(training_set.shape[0]*
    training_set.shape[1], -1))
    # We first calculate the euclidean distance between the probe and all trained
    embeddings:
    distances = np.linalg.norm(training_set - probe_embedding, axis=1)
    nearest_neighbor_idx = np.argmin(distances)

    # Then we return the index with the smallest distance
    return index_to_class_id[nearest_neighbor_idx]

# %%
def get_PCA_vecs(img_vec, p, speedup=True):
    overall_mean = np.mean(img_vec, axis=0)

    # Substract by the mean:
    img_meaned = img_vec - overall_mean

    # PCA:
    # Return p eigenvectors from the covariance matrix:
    # Calculate the covariance matrix:
    if speedup == False:
        print("IMg_meanign", img_meaned.shape)
        _, _, Vt = np.linalg.svd(img_meaned @ img_meaned.T, full_matrices=False)

        W_p = img_meaned.T @ Vt

        # Normalize the principal components
        W_p = W_p / np.linalg.norm(W_p, axis=1, keepdims=True)

        # Extract the top-p right singular vectors (principal components)
        W_phat = W_p[:, :p]  # Shape is (CHW, p)
    else:
        C = img_meaned @ img_meaned.T # Shape is (B, B)

        # Take the eig vecs of this, they are already in reverse order:
        _, eigvec_c = np.linalg.eigh(C)

        W_p = img_meaned.T @ eigvec_c  # Shape is (CHW, p)
        W_phat = W_p / np.linalg.norm(W_p, axis=1, keepdims=True) # Normalized
        W_phat = W_phat[:, -p:]

    # Project the image vectors into the p dimensional space
    pca = img_meaned @ W_phat
    return pca

# %% [markdown]
# ### Task 1: LDA

# %%
def calculate_global_parameters_for_LDA(train_loader, num_classes=30, feature_dim=4096):
    num_classes = 30
    overall_mean = 0
    class_means = np.zeros((num_classes, feature_dim))
    num_samples = 0
    class_sample_counts = np.zeros((num_classes))

    for batch in train_loader:
```

```
        img_vecn = batch["img_vecn"].numpy()
        labels = batch["y"] # Shape is (B,)

        # Accumulate means:
        overall_mean += np.sum(img_vecn, axis=0) # shape is: (4096) -> (CHW)
        num_samples += img_vecn.shape[0]

        # Set up the dataset wide information required
        for label in np.unique(labels):
            # idx of class_means = label + 1
            class_samples = img_vecn[labels == label]
            class_sample_counts[label - 1] += class_samples.shape[0]
            class_means[label - 1] += np.sum(class_samples, axis=0)

    # Finalize the means:
    overall_mean = np.array(overall_mean / num_samples)  # Shape: (4096,)
    for label in range(num_classes):
        class_means[label] /= class_sample_counts[label]  # Shape: (4096,)

    # Compute S_B, the outer product for the between-class scatter:
    S_B = np.zeros((feature_dim, feature_dim)) # Shape is (4096, 4096)
    for i in range(num_classes):
        mean_diff = np.expand_dims(class_means[i] - overall_mean, axis=1) # Shape is (4096, 1)
        S_B += mean_diff @ mean_diff.T

    # Compute Within-Class Scatter Matrix (S_W)
    S_W = np.zeros((feature_dim, feature_dim)) # Shape is (4096, 4096)
    for batch in train_loader:
        img_vecn = batch["img_vecn"].numpy()
        labels = batch["y"] # Shape is (B,)

        # Set up the dataset wide information required
        for label in np.unique(labels):
            # idx of class_means = label + 1
            class_samples = img_vecn[labels == label]

            for sample in class_samples:
                diff = np.expand_dims((sample - class_means[label - 1]), axis=1)  # Shape: (4096, 1)
                S_W += diff @ diff.T  # Outer product shape is (4096, 4096)
    return S_B, S_W

# %%
def get_projection_matrix(S_W, S_B, p, lda_option, num_labels=30):
    if lda_option == "YUYANG":
        # Retain eigvecs where the values are not close to 0
        eigvals, eigvecs = np.linalg.eigh(S_B)
        idx = eigvals > 1e-6  # Filter eigenvalues
        top_eigvals, top_eigvecs = eigvals[idx], eigvecs[:, idx]

        # Normalize the eigenvectors:
        sb_eigvecn = top_eigvecs / np.linalg.norm(top_eigvecs, axis=1, keepdims=True) # Shapes is (CHW, K_Y)

        eig_val_mat = np.diag(top_eigvals) #np.eye(num_labels - 1) * top_eigvals
        # We can then construct a low dimensional projection of S_B with sb_eigvecn
        D_B = np.sqrt(np.linalg.inv(eig_val_mat))
        Z = np.dot(sb_eigvecn, D_B)

        # Use eigendecomp to diagonalize Z
        _, U = np.linalg.eigh(Z.T @ S_W @ Z)
        # Get the top eigenvectors and normalize them
        U_top = U[:, -p:]
        U_topn = U_top / np.linalg.norm(U_top, axis=1, keepdims=True)

        # Generate the projection matrix
        proj_mat = (U_topn.T @ Z.T).T
    else:
        # LDA Optimization
        # Get the eigenvalue/vectors of S_W^-1 S_B
        _, eigvecs = np.linalg.eig(np.linalg.inv(S_W) @ S_B)

```

```python
        # Get the top eigvecs since np already sorts them
        proj_mat = eigvecs[:, -p:]  # Columns are the eigenvectors

    return proj_mat

# %% [markdown]
# ### Nearest neighbor classifier:

# %%
# Run through training dataset and create the mean embedding for all the images
    belonging to that class
def train_classifier(train_loader, lda_proj_mat, dim_reducer, p, num_classes=30):
    class_embs = [[] for _ in range(num_classes)]

    for batch in train_loader:
        img_vecn = batch["img_vecn"].numpy()
        labels = batch["y"] # Shape is (B,)

        if img_vecn.shape[0] >= p:

            if dim_reducer == "PCA":
                embs = get_PCA_vecs(img_vecn, p)

            elif dim_reducer == "LDA":
                embs = img_vecn @ lda_proj_mat  # Shape is (B, p)

            else:
                raise ValueError("Wrong input type: dim_reducer should be PCA or LDA")

            # Train Classifier embeddings
            for label in np.unique(labels):
                # idx of class_means = label + 1
                class_embedding = embs[labels == label]

                for sample in class_embedding:
                    class_embs[label - 1].append(sample)

    return np.array(class_embs).astype(np.float32)

# %%
def run_testing_script(test_loader, lda_proj_mat, class_embs, dim_reducer, num_classes
    =30, num_samples=21):
    predicted_label = []
    true_label = []
    test_embs_list = [[] for _ in range(num_classes)]

    for batch in test_loader:
        img_vecn = batch["img_vecn"].numpy()
        labels = batch["y"] # Shape is (B,)

        if img_vecn.shape[0] >= p:
            if dim_reducer == "PCA":
                embs = get_PCA_vecs(img_vecn, p)

            elif dim_reducer == "LDA":
                # img_vecn = img_vec / np.linalg.norm(img_vec, axis=1, keepdims=True) #
    Shapes is (B, CHW)
                # The projection matrix is just the top eigenvectors?
                embs = img_vecn @ lda_proj_mat  # Shape is (B, p)

            else:
                raise ValueError("Wrong input type: dim_reducer should be PCA or LDA")
            # Compare nearest embeddings to get predicted label
            for embedding, label in zip(embs, labels):
                embedding = np.array(np.expand_dims(embedding, axis=0), dtype=np.float32
    )
                index = get_id_of_nearest_embedding(class_embs, embedding, num_classes,
    num_samples)

                test_embs_list[index.item()].append(embedding)
                predicted_label.append(index.item() + 1)
                true_label.append(label)
```

```
    return np.array(predicted_label, dtype=np.float32), np.array(true_label, dtype=np.float32), test_embs_list

# %%
accuracy_list_lda = []

# %%
# batch_size = 630
# num_classes = 30
# dim_reducer = "LDA"
# lda_option = "YUYANG"
# train_loader = DataLoader(dataset=DimReducerBuilder(TRAIN_DATA_PATH, option="bw"), batch_size=batch_size, shuffle=False)
# S_B, S_W = calculate_global_parameters_for_LDA(train_loader, num_classes=num_classes, feature_dim=4096)
# for p in range(3, 32):
#     if p % 24 == 0:
#         print(p)
#     lda_proj_mat = get_projection_matrix(S_B=S_B, S_W=S_W, p=p, lda_option=lda_option)

#     # Run through test set and find the nearest embedding and assign that label to the image
#     class_embs = train_classifier(train_loader, dim_reducer=dim_reducer, p=p, num_classes=num_classes, lda_proj_mat=lda_proj_mat)

#     test_loader = DataLoader(dataset=DimReducerBuilder(EVAL_DATA_PATH, option="bw"), batch_size=batch_size, shuffle=False)
#     pred_labels, true_labels, test_embs = run_testing_script(test_loader, lda_proj_mat, class_embs, dim_reducer=dim_reducer)
#     accuracy = np.count_nonzero(pred_labels == true_labels) / len(pred_labels)
#     accuracy_list_lda.append(np.round(accuracy * 100, 4))

# %%
p = 16 # [3, 8, 16]
batch_size = 630
num_classes = 30

# %%
dim_reducer = "PCA"
lda_option = "YUYANG" #"YUYANG"
train_loader = DataLoader(dataset=DimReducerBuilder(TRAIN_DATA_PATH, option="bw"), batch_size=batch_size, shuffle=False)


# %%
lda_proj_mat = None
if dim_reducer == "LDA":
    # Compute S_W and S_B:
    S_B, S_W = calculate_global_parameters_for_LDA(train_loader, num_classes=num_classes, feature_dim=4096)
    # Also decides whether we compute YUYANG or not:
    lda_proj_mat = get_projection_matrix(S_B=S_B, S_W=S_W, p=p, lda_option=lda_option)

# %%
# Run through test set and find the nearest embedding and assign that label to the image
class_embs = train_classifier(train_loader, dim_reducer=dim_reducer, p=p, num_classes=num_classes, lda_proj_mat=lda_proj_mat)

# %%
test_loader = DataLoader(dataset=DimReducerBuilder(EVAL_DATA_PATH, option="bw"), batch_size=batch_size, shuffle=False)
pred_labels, true_labels, test_embs = run_testing_script(test_loader, lda_proj_mat, class_embs, dim_reducer=dim_reducer)
accuracy = np.count_nonzero(pred_labels == true_labels) / len(pred_labels)
np.round(accuracy * 100, 4)

# %% [markdown]
# ### Graph the embeddings:

# %%
test_embs_to_print = []
for sample in test_embs:
    if len(sample) >= 10:
```

```
        test_embs_to_print.append(sample)

# %%
# Match the shapes of the train and test embeddings:
min_num_training_class_samples, min_num_test_class_samples = np.inf, np.inf
for train_sample, test_sample in zip(class_embs, test_embs_to_print):
    if len(train_sample) < min_num_training_class_samples:
        min_num_training_class_samples = len(train_sample)
    if len(test_sample) < min_num_test_class_samples:
        min_num_test_class_samples = len(test_sample)

print("Minimum training samples per class", min_num_training_class_samples)
print("Minimum testing samples per class", min_num_test_class_samples)

graph_train_embs = np.zeros((num_classes, min_num_training_class_samples, p))
graph_test_embs = np.zeros((num_classes, min_num_test_class_samples, p))

for i, (train_sample, test_sample) in enumerate(zip(class_embs, test_embs_to_print)):
    test_sample = np.squeeze(test_sample)
    graph_train_embs[i] = np.array(train_sample[:min_num_training_class_samples], dtype=np.float32)
    graph_test_embs[i] = np.array(test_sample[:min_num_test_class_samples], dtype=np.float32)

# Reshape embeddings and generate labels
train_embeddings = graph_train_embs.reshape(-1, p)
train_labels = np.repeat(np.arange(num_classes), min_num_training_class_samples)

test_embeddings = graph_test_embs.reshape(-1, p)
test_labels = np.repeat(np.arange(num_classes), min_num_test_class_samples)

# Reduce to 2D with UMAP
umap_reducer = umap.UMAP(n_components=2)
train_umap = umap_reducer.fit_transform(train_embeddings)
test_umap = umap_reducer.transform(test_embeddings)


# %%
# Plot training data with different colors for each class
plt.figure(figsize=(8, 6))
scatter = plt.scatter(train_umap[:, 0], train_umap[:, 1], c=train_labels, cmap='tab20', alpha=0.7)
plt.colorbar(scatter, ticks=range(num_classes), label="Class Label")
plt.title("Training Data: UMAP Embeddings")
plt.axis("off")
plt.savefig(f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/PCA/UMAP/train_umap_{p}.jpg")
plt.close()

# Plot test data with predicted labels
plt.figure(figsize=(8, 6))
scatter = plt.scatter(test_umap[:, 0], test_umap[:, 1], c=test_labels, cmap='tab20', alpha=0.7)
plt.colorbar(scatter, ticks=range(num_classes), label="Ground Truth Label")
plt.title("Test Data: UMAP Embeddings with Predicted Labels")
plt.axis("off")
plt.savefig(f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/PCA/UMAP/test_umap_{p}.jpg")
plt.close()

# Generate and plot the confusion matrix:
cm = confusion_matrix(true_labels, pred_labels)

# Create a heatmap using Seaborn
plt.figure(figsize=(16, 16))
sns.heatmap(cm, annot=True, fmt='.0f', cmap='Blues')

# Add labels and title
plt.ylabel('Actual Class')
plt.xlabel('Predicted Class')
plt.axis("off")
plt.savefig(f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/PCA/Confusion_Mat/conf_mat_{p}.jpg")
```

```
plt.close()

# %%
# Generate x-axis values (e.g., epoch numbers or iteration indices)
x_values = list(range(1, len(accuracy_list) + 1))
plt.plot(x_values, accuracy_list, marker='o', linestyle='-', label='PCA')
plt.plot(x_values, accuracy_list_lda, marker='x', linestyle='-', label='LDA')
plt.legend(loc="upper right")
plt.xlabel("Embedding Dim")
plt.ylabel("Accuracy (%)")
plt.grid(True)

plt.savefig("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/Accuracy/PCA.jpg")

# %%
import os
import numpy as np
import torch
from torch import nn, optim
import umap
from PIL import Image
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

# %%
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device

# %% [markdown]
# ### Accuracies:
#
# p = 3: 0%
#

# %%
class DataBuilder(Dataset):
    def __init__(self, path, option):
        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)
        if option == "bw":
            self.aug = transforms.Compose([
                transforms.Resize((64, 64)),
                transforms.Grayscale(num_output_channels=1),
                transforms.ToTensor(),
            ])
        else:
            self.aug = transforms.Compose([
                transforms.Resize((64, 64)),
                transforms.ToTensor(),
            ])

    def __len__(self):
        return self.len

    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 get_id_of_nearest_embedding(training_set, probe_embedding):
    # We first calculate the euclidean distance between the probe and all trained
    embeddings:
    distances = np.linalg.norm(training_set - probe_embedding, axis=1)
```

```python

    # Then we return the index with the smallest distance
    return np.argmin(distances)

# %%
def train(epoch, vae_loss, model, optimizer, trainloader):
    model.train()
    train_loss = 0

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

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

# %%
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)


# %% [markdown]
# ### VAE Training:

# %%
##################################
# Change these
p = 8
batch_size = 24
training = False
TRAIN_DATA_PATH = '/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/train/'
EVAL_DATA_PATH = '/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/test/'
LOAD_PATH = f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/exp/model_{p}.pt"
OUT_PATH = '/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/exp/'
##################################

# %%
train_loader = DataLoader(dataset=DataBuilder(TRAIN_DATA_PATH, option="bw"), batch_size=
    batch_size, shuffle=True)


# %%
accuracy_list = []

# %%
p = 16
LOAD_PATH = f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/exp/model_{p}.pt"
train_loader = DataLoader(dataset=DataBuilder(TRAIN_DATA_PATH, option="bw"), batch_size=
    batch_size, shuffle=True)
training=False
model = Autoencoder(p).to(device)

if training:
    epochs = 100
    log_interval = 1
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH, option=""),
        batch_size=32,
    )
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    vae_loss = VaeLoss()
    for epoch in range(1, epochs + 1):
        model = train(epoch, vae_loss, model, optimizer, trainloader)
    torch.save(model.state_dict(), os.path.join(OUT_PATH, f'model_{p}.pt'))
```

```python
else:
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH, option=""),
        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'].to(device))
        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, option=""),
        batch_size=1,
    )
    X_test, y_test = [], []
    for batch_idx, data in enumerate(testloader):
        mu, logvar = model.encode(data['x'].to(device))
        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)


# %%
train_embs = [[] for _ in range(30)]
test_embs = [[] for _ in range(30)]

for i, (train_emb, train_label, test_emb, test_label) in enumerate(zip(X_train, y_train, X_test, y_test)):
    train_embs[train_label - 1].append(train_emb)
    test_embs[test_label - 1].append(test_emb)


# %%
train_embs = np.array(train_embs, dtype=np.float32)
test_embs = np.array(test_embs, dtype=np.float32)

# %%
# Now that all the array is reordered, I can add them to the faiis search
num_classes, num_samples, embedding_dim = train_embs.shape
flattened_train_embs = train_embs.reshape(-1, embedding_dim)
print("Flattened", flattened_train_embs.shape)

# Create a mapping from flattened indices to class IDs
index_to_class_id = np.repeat(np.arange(num_classes), num_samples)
print("indices", index_to_class_id.shape)



# %%
true_labels = []
pred_labels = []
for test_emb, test_label in zip(X_test, y_test):
    search_emb = np.array(np.expand_dims(test_emb, axis=0), dtype=np.float32)
    index = get_id_of_nearest_embedding(flattened_train_embs, search_emb)
    true_labels.append(test_label)
    pred_labels.append(index_to_class_id[index.item()] + 1)
true_labels = np.array(true_labels)
pre_labels = np.array(pred_labels)
accuracy = np.count_nonzero(pred_labels == true_labels) / len(pred_labels)
print("Accuracy: ", np.round(accuracy * 100, 4), "\n")
accuracy_list.append(np.round(accuracy * 100, 4))

# %%
# Generate x-axis values (e.g., epoch numbers or iteration indices)
```

```
x_values = list(range(1, len(accuracy_list) + 1))
plt.plot(x_values, accuracy_list, marker='o', linestyle='-', label='Accuracy')
plt.grid(True)

plt.savefig("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/Accuracy/Autoencoder.jpg
    ")

# %%
num_classes, num_samples, embedding_dim = train_embs.shape
flattened_train_embs = train_embs.reshape(-1, embedding_dim)
train_labels = np.repeat(np.arange(num_classes), num_samples)

num_classes, num_samples, embedding_dim = test_embs.shape
flattened_test_embs = test_embs.reshape(-1, embedding_dim)
test_labels = np.repeat(np.arange(num_classes), num_samples)

# %%
# Reduce to 2D with UMAP
umap_reducer = umap.UMAP(n_components=2)
train_umap = umap_reducer.fit_transform(flattened_train_embs)
test_umap = umap_reducer.transform(flattened_test_embs)

# %%
# Plot training data with different colors for each class
plt.figure(figsize=(8, 6))
scatter = plt.scatter(train_umap[:, 0], train_umap[:, 1], c=train_labels, cmap='tab20',
    alpha=0.7)
plt.colorbar(scatter, ticks=range(num_classes), label="Class Label")
plt.title("Training Data: UMAP Embeddings")
plt.axis("off")
plt.savefig(f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/AutoEncoder/UMAP/
    train_umap_{p}.jpg")
plt.close()

# Plot test data with predicted labels
plt.figure(figsize=(8, 6))
scatter = plt.scatter(test_umap[:, 0], test_umap[:, 1], c=test_labels, cmap='tab20',
    alpha=0.7)
plt.colorbar(scatter, ticks=range(num_classes), label="Ground Truth Label")
plt.title("Test Data: UMAP Embeddings with Predicted Labels")
plt.axis("off")
plt.savefig(f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/AutoEncoder/UMAP/
    test_umap_{p}.jpg")
plt.close()

# Generate and plot the confusion matrix:
cm = confusion_matrix(true_labels, pred_labels)

# Create a heatmap using Seaborn
plt.figure(figsize=(16, 16))
sns.heatmap(cm, annot=True, fmt='.0f', cmap='Blues')

# Add labels and title
plt.ylabel('Actual Class')
plt.xlabel('Predicted Class')
plt.axis("off")
plt.savefig(f"/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/Results/AutoEncoder/
    Confusion_Mat/conf_mat_{p}.jpg")
plt.close()

# %%

# %%
import os
import numpy as np
import torch
import torch.nn.functional as F
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt

# %%
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device

# %%
def apply_haar_filter(img_bw, haar_size):
    if haar_size % 2 == 1:
        # Odd -> add 1
        # Else this is already the largest even number > 4 sigma
        haar_size += 1
    haar_dx = np.vstack((-1*np.ones((haar_size, 1)), np.ones((haar_size, 1)) ))
    haar_dy = torch.tensor(-1*haar_dx.copy().T)
    haar_dx = torch.tensor(haar_dx)

    # haar_dx = torch.tensor([[-1] * haar_size + [1] * haar_size], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    # haar_dy = -haar_dx.transpose(2, 3)

    # dx = cv2.filter2D(img_bw, -1, haar_dx)
    # dy = cv2.filter2D(img_bw, -1, haar_dy)

    dx = F.conv2d(img_bw, haar_dx, padding='same')
    dy = F.conv2d(img_bw, haar_dy, padding='same')

    return np.hstack((dx, dy))

def apply_haar_filter(img_bw, haar_size):
    haar_dx_np = np.vstack((-1 * np.ones((haar_size, 1)), np.ones((haar_size, 1))))
    haar_dy_np = -haar_dx_np.T

    # Convert Haar kernels to PyTorch tensors with proper shape
    haar_dx = torch.tensor(haar_dx_np, dtype=torch.float32).unsqueeze(0).unsqueeze(0)  # Shape: (1, 1, H, W)
    haar_dy = torch.tensor(haar_dy_np, dtype=torch.float32).unsqueeze(0).unsqueeze(0)  # Shape: (1, 1, H, W)

    # Ensure img_bw has the correct dimensions (batch_size, channels, height, width)
    if len(img_bw.shape) == 3:
        img_bw = img_bw.unsqueeze(0)  # Add batch dimension if missing

    # Apply Haar filters using F.conv2d
    dx = F.conv2d(img_bw, haar_dx, padding='same')
    dy = F.conv2d(img_bw, haar_dy, padding='same')
    return torch.hstack((dx, dy))

# %%
class DataBuilder(Dataset):
    def __init__(self, path, option=False):
        self.path = path
        self.image_list = [f for f in os.listdir(path) if f.endswith('.png')]
        self.len = len(self.image_list)
        self.aug = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.Grayscale(num_output_channels=1),
            transforms.ToTensor(),
        ])

    def __len__(self):
        return self.len

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

        low_feature_vector = [torch.squeeze(torch.reshape(x, shape=(x.shape[0], -1)))]
        for haar_size in range(2, x.shape[1], 2):
            haar_img = apply_haar_filter(x, haar_size)
            pooled_img = F.avg_pool2d(haar_img, kernel_size=(haar_size, haar_size))
            pooled_img_flat = torch.squeeze(torch.reshape(pooled_img, shape=(pooled_img.shape[0], -1)))
            low_feature_vector.append(pooled_img_flat)
        feature_vecs = torch.hstack(low_feature_vector)

```

```
        return feature_vecs


# %%
pos_train_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/train/positive/"
neg_train_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/train/negative/"
pos_test_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/test/negative/"
neg_test_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW10/test/negative/"
num_pos_train = len(os.listdir(pos_train_path))
num_neg_train = len(os.listdir(neg_train_path))
num_pos_test = len(os.listdir(pos_test_path))
num_neg_test = len(os.listdir(neg_test_path))

# %%
pos_train_loader = DataLoader(dataset=DataBuilder(pos_train_path), batch_size=num_pos_train, shuffle=True)
neg_train_loader = DataLoader(dataset=DataBuilder(neg_train_path), batch_size=num_neg_train, shuffle=True)
pos_test_loader = DataLoader(dataset=DataBuilder(pos_test_path), batch_size=num_pos_test, shuffle=True)
neg_test_loader = DataLoader(dataset=DataBuilder(neg_test_path), batch_size=num_neg_test, shuffle=True)

# %%
for pos_train, neg_train, pos_test, neg_test in zip(pos_train_loader, neg_train_loader, pos_test_loader, neg_test_loader):
    print("Pos Train: ", pos_train.shape)
    print("Neg Train: ", neg_train.shape)
    # Create a matrix of all training images, positive or negative:
    train_imgs = np.vstack((pos_train, neg_train))
    train_labels = np.hstack(( np.ones(pos_train.shape[0]), -1*np.ones(neg_train.shape[0])) )

    # Randomize the order of the training images. Maintain a constant pairing of label to img though
    shuffle_indices = np.random.permutation(train_imgs.shape[0])
    train_imgs = train_imgs[shuffle_indices]
    train_labels = train_labels[shuffle_indices]

    # Do the same for thesting images:
    test_imgs = np.vstack((pos_test, neg_test))
    test_labels = np.hstack(( np.ones(pos_test.shape[0]), -1*np.ones(neg_test.shape[0])) )
    shuffle_indices = np.random.permutation(test_imgs.shape[0])
    test_imgs = test_imgs[shuffle_indices]
    test_labels = test_labels[shuffle_indices]

    print("Train imgs", train_imgs.shape)
    print("Test imgs", test_imgs.shape)

# %%
class WeakClassifier():
    def __init__(self):
        # These are the terms we need to calculate for the weak-classifier:
        self.best_feature = None
        self.best_threshold = None
        self.best_polarity = None
        self.min_error = np.inf

    def get_params(self, imgs, labels, weight_mat):
        # Normalize each weight matrix:
        weight_mat = weight_mat / np.sum(weight_mat)

        # We need to loop through each feature in the img matrix. Each feature is counted as a column in that matrix:
        for feature_idx in range(imgs.shape[1]):
            # Extract current feature (the column)
            features = imgs[:, feature_idx]

            # Sort the feature, weights, and labels
            sorted_indices = np.argsort(features)
            sorted_features = features[sorted_indices]
            sorted_weights = weight_mat[sorted_indices]
```

```
            sorted_labels = labels[sorted_indices]

            # We need to use multiplication here to preserve the original shape
            # However, we don't want the opposite labels to affect the cumulative summ
            S_plus = np.cumsum(sorted_weights * (sorted_labels == 1))
            S_minus = np.cumsum(sorted_weights * (sorted_labels == -1))
            T_plus = np.sum(sorted_weights * (sorted_labels == 1))
            T_minus = np.sum(sorted_weights * (sorted_labels == -1))

            # Calculate the polarity errors:
            e_1 = S_plus + T_minus - S_minus
            e_neg1 = S_minus + T_plus - S_plus

            # Calculate classification error:
            for i, feature in enumerate(sorted_features):
                # Since Error = min(e_1, e_neg1) we will always compute both and keep the trailing
                # minimum along both calculations
                if e_1[i] < self.min_error:
                    self.min_error = e_1[i]
                    self.best_feature = feature_idx
                    self.best_threshold = feature
                    self.best_polarity = 1
                if e_neg1[i] < self.min_error:
                    self.min_error = e_neg1[i]
                    self.best_feature = feature_idx
                    self.best_threshold = feature
                    self.best_polarity = -1
        return (self.best_feature, self.best_threshold, self.best_polarity, self.min_error)


# %%
class ClassifierCascade():
    def __init__(self):
        self.classifier_list = []
        self.alpha_list = []
        self.cascades = []
        self.max_num_classifiers_per_cascade = 5
        self.max_cascades = 3

    def run_strong_classifier(self, imgs, labels):
        # I associate a uniform initial weight with each image initially:
        weight_mat = np.ones(imgs.shape[0], dtype=np.float32) / imgs.shape[0]

        # Define the cascade:
        for classifier_idx in range(self.max_num_classifiers_per_cascade):
            # Every new iteration, we add in a new weak classifier until we have reached the maximum number, or they have the correct accuracy.
            self.classifier_list.append(WeakClassifier().get_params(imgs, labels, weight_mat))
            feature, threshold, polarity, error = self.classifier_list[classifier_idx]

            # Update algorithm parameters
            beta = error / (1 - error)
            alpha = np.log(1 / beta)
            self.alpha_list.append(alpha)

            print(f"Weak classifier id {(classifier_idx + 1):.3f} feature: {feature:.3f}, threshold: {threshold:.3f}, polarity: {polarity:.3f}, error: {error:.3f}, alpha: {alpha:.3f}")

            # Update weights accordingly:
            # You need to find your predictions (feature vs threshold feature value)
            # However, also make sure to take into account the polarity
            pred_labels = np.where((imgs[:, feature] * polarity) >= (threshold * polarity), 1, -1)
            wrong_preds = pred_labels != labels

            # Multiply by 1 where predictions are incorrect, else by beta.
            # Also normalize weights to make sure they sum to 1
            weight_mat = weight_mat * np.where(wrong_preds, 1, beta)
            weight_mat /= np.sum(weight_mat)
```

```python

            # Calculate the accuracy of all weak classifiers so far:
            # To do so we first need to get the predictions by passing the input through all the weak classifiers
            # We can then get the sign of this predictions to assign it to a class label (1 or -1)
            predictions_so_far = np.zeros_like(labels)
            # Get the predictions for each classifier, alpha is a weight factor for how much each classifier contributes
            for (f, th, p, _), alpha in zip(self.classifier_list, self.alpha_list):
                predictions_so_far += alpha * np.where((imgs[:, f] * p) >= (th * p), 1, -1)
            final_pred_labels = np.sign(predictions_so_far)

            # Calculate accuracy
            accuracy = np.count_nonzero(final_pred_labels == labels) / len(final_pred_labels)
            print(f"Iteration {classifier_idx + 1}: Accuracy = ", np.round(accuracy, 3))

            # Evaluate if there are enough classifiers:
            if accuracy > 1.00:
                break
        return self.classifier_list, self.alpha_list

    def run_cascades(self, imgs, labels):
        # This function will run multiple cascades until the false positive rate reaches 0
        for cascade_idx in range(self.max_cascades):
            # We first train a strong classifier
            classifier_params, alphas = ClassifierCascade().run_strong_classifier(imgs, labels)
            self.cascades.append((classifier_params, alphas))

            # We then need to evaluate the most recent cascade as we did before
            predictions_so_far = np.zeros_like(labels)
            for (f, th, p, _), alpha in zip(classifier_params, alphas):
                predictions_so_far += alpha * np.where((imgs[:, f] * p) >= (th * p), 1, -1)
            cascade_pred_labels = np.sign(predictions_so_far)

            # false_positives = np.mean((cascade_pred_labels == 1) & (labels == -1))
            false_positives = np.count_nonzero((cascade_pred_labels == 1) & (labels == -1)) / len(cascade_pred_labels)
            false_negatives = np.count_nonzero((cascade_pred_labels == -1) & (labels == 1)) / len(cascade_pred_labels)
            accuracy = np.count_nonzero(cascade_pred_labels == labels) / len(cascade_pred_labels)
            print(f"Cascade id: {cascade_idx + 1}, False positive rate: {false_positives:.2f}, False negative rate: {false_negatives:.2f} Final Accuracy: {accuracy:.2f}")

            # Now that we know the false positive rate, we can either terminate, or keep going by removing correctly labeled negatives:
            if false_positives + false_negatives < 0.01:
                break
            # Remove all the correctly classified negative images from the dataset
            idx_to_keep = (labels == 1) | ((cascade_pred_labels == 1) & (labels == -1))
            imgs = imgs[idx_to_keep]
            labels = labels[idx_to_keep]

            # We also need to stop if there are no more imgs left (not removing any)
            if len(idx_to_keep) == len((labels == 1)):
                break

        return self.cascades

    def test_cascade(self, imgs, labels):
        total_num_images = imgs.shape[0]
        final_true_negatives = np.zeros_like(labels)
        final_true_positives = np.zeros_like(labels)

        # Get the accuracy:
        for cascade_idx, (classifier_params, alphas) in enumerate(self.cascades):
            # Get the predictions on the test dataset:
```

```
            predictions_so_far = np.zeros_like(labels)
            for (f, th, p, _), alpha in zip(classifier_params, alphas):
                predictions_so_far += alpha * np.where((imgs[:, f] * p) >= (th * p), 1, -1)
            cascade_pred_labels = np.sign(predictions_so_far)

            # Compute performance metrics for the current cascade
            true_negative_mask = (cascade_pred_labels == -1) & (labels == -1)
            true_positive_mask = (cascade_pred_labels == 1) & (labels == 1)
            final_true_negatives = np.logical_or(final_true_negatives, true_negative_mask)
            final_true_positives = np.logical_or(final_true_positives, true_positive_mask)

            false_positives = np.count_nonzero((cascade_pred_labels == 1) & (labels == -1))
            false_negatives = np.count_nonzero((cascade_pred_labels == -1) & (labels == 1))
            true_negatives = np.count_nonzero(true_negative_mask)
            true_positives = np.count_nonzero(true_positive_mask)
            tot_negative = np.count_nonzero(labels == -1)
            tot_positive = np.count_nonzero(labels == 1)

            print(f"True Positives: {true_positives}, True Negatives: {true_negatives}, Total Image Count {total_num_images}")
            false_positive_rate = false_positives / (tot_negative)
            false_negative_rate = false_negatives / (tot_positive)
            accuracy = (true_positives + true_negatives) / len(cascade_pred_labels)
            percent_of_dataset_kept = len(cascade_pred_labels) / total_num_images
            print(f"Cascade id: {cascade_idx + 1}, False positive rate: {false_positive_rate:.2f}, False negative rate: {false_negative_rate:.2f}, Accuracy: {np.round(accuracy*100, 4)}, over {np.round(percent_of_dataset_kept*100, 4)} of the imgs.")

        # Print final accuracy:
        final_tp = np.count_nonzero(final_true_positives)
        final_tn = np.count_nonzero(final_true_negatives)
        final_accuracy = (final_tp + final_tn) / total_num_images
        print(f"Final Accuracy: {np.round(final_accuracy*100, 4)}")


# %%
classifier_cascade = ClassifierCascade()
cascade_params = classifier_cascade.run_cascades(train_imgs.copy(), train_labels.copy())

# %%
classifier_cascade.test_cascade(test_imgs.copy(), test_labels.copy())

# %%
```