# Import library

**Importing necessary libraries:**

This code imports various libraries such as scikit-learn (for SVM, metrics, decomposition), PIL (for image processing), numpy (for numerical operations), os (for operating system-related tasks), cv2 (OpenCV for image processing), matplotlib (for plotting), seaborn (for visualizations), and h5py (for storing data).

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.decomposition import PCA

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

import matplotlib.pyplot as plt

import seaborn as sns

from PIL import Image

import numpy as np

import os

import cv2

import math

import glob

import h5p**y**

# Preprocessing

**Preprocessing for test data:**

This code reads the test images from the 'Test' folder, converts them to grayscale, resizes them to (150, 130) pixels, and saves them back to the same location.

folders = glob.glob('Test/\*')

imagenames\_list = []

for folder in folders:

    for f in glob.glob(folder+'/\*.jpg'):

        imagenames\_list.append(f)

for image in imagenames\_list:

    Preimage = Image.open(image).convert('L')

    new\_image = Preimage.resize((150, 130))

    new\_image.save(image)

1. **Preprocessing for training data (similar to the previous step):**

folders = glob.glob('Train/\*')

imagenames\_list = []

for folder in folders:

    for f in glob.glob(folder+'/\*.jpg'):

        imagenames\_list.append(f)

for image in imagenames\_list:

    Preimage = Image.open(image).convert('L')

    new\_image = Preimage.resize((150, 130))

    new\_image.save(image)

# Load data and Create label

**Loading training data:**

This code loads the training data by iterating over the folders and images in the 'Train' directory. It reads each image, converts it to grayscale, resizes it to (150, 130) pixels, and appends it to the 'train' list along with its class label.

folderName = []

train = []

dataPath = "C:/Users/selle/Downloads/temple recognition/Test"

IMG\_WIDTH = 150

IMG\_HEIGHT = 130

for i in range(1, 4):

    folderName.append(str(i))

    i+=1

def createData():

    for folder in folderName:

        path = os.path.join(dataPath, folder)

        class\_num = folderName.index(folder)

        for image in os.listdir(path):

            imageArray = cv2.imread(os.path.join(path, image), cv2.IMREAD\_GRAYSCALE)

            newArray = cv2.resize(imageArray, (IMG\_WIDTH, IMG\_HEIGHT))

            train.append([newArray, class\_num])

createData()

1. **Loading test data (similar to the previous step):**
2. folderName = []
3. test = []
4. dataPath = "C:/Users/selle/Downloads/temple recognition/Train"
5. IMG\_WIDTH = 150
6. IMG\_HEIGHT = 130
7. for i in range(1, 4):
8. folderName.append(str(i))
9. i+=1
11. def createData():
12. for folder in folderName:
13. path = os.path.join(dataPath, folder)
14. class\_num = folderName.index(folder)
15. for image in os.listdir(path):
16. imageArray = cv2.imread(os.path.join(path, image), cv2.IMREAD\_GRAYSCALE)
17. newArray = cv2.resize(imageArray, (IMG\_WIDTH, IMG\_HEIGHT))
18. test.append([newArray, class\_num])
20. createData()

The code you provided is responsible for preparing the training and testing data for a machine learning model. Here's a breakdown of what each section does:

X\_train = []

Y\_train = []

X\_test = []

Y\_test = []

In the first four lines, empty lists are initialized to store the features and labels for both the training and testing data.

for features, label in train:

    X\_train.append(features)

    Y\_train.append(label)

The next loop iterates over the `train` dataset, which likely contains a collection of feature-label pairs. For each pair, the features are appended to the `X\_train` list, and the label is appended to the `Y\_train` list.

for features, label in test:

    X\_test.append(features)

    Y\_test.append(label)

Similarly, the subsequent loop operates on the `test` dataset. It iterates over the feature-label pairs, appends the features to the `X\_test` list, and the label to the `Y\_test` list.

X\_train = np.array(X\_train)

Y\_train = np.array(Y\_train)

X\_test = np.array(X\_test)

Y\_test = np.array(Y\_test)

Finally, after the loops, the `X\_train`, `Y\_train`, `X\_test`, and `Y\_test` lists are converted into NumPy arrays using the `np.array()` function. This conversion allows for efficient manipulation and processing of the data using the NumPy library.

By the end of this code snippet, you will have four NumPy arrays: `X\_train`, `Y\_train`, `X\_test`, and `Y\_test`, which contain the training features, training labels, testing features, and testing labels, respectively. These arrays can be used as input for training and evaluating a machine learning model.

# Store train data to h5py

hf = h5py.File('xtrain.h5', 'w')

hf.create\_dataset('X\_train', data=X\_train)

In the first section, an HDF5 file named "xtrain.h5" is created in write mode (`'w'`). Then, a dataset named 'X\_train' is created inside this file, and the training data (`X\_train`) is stored in this dataset.

# Store test data to h5py

hf = h5py.File('xtest.h5', 'w')

hf.create\_dataset('X\_test', data=X\_test)

In the second section, a similar process is performed to store the testing data. An HDF5 file named "xtest.h5" is created, and a dataset named 'X\_test' is created inside this file to store the testing data (`X\_test`).

# Load h5py file

h5\_train = h5py.File('xtrain.h5', 'r+')

h5\_test = h5py.File('xtest.h5', 'r+')

In the third section, the previously created HDF5 files are loaded in read-write mode (`'r+'`) using the h5py.File() function. These files are assigned to `h5\_train` and `h5\_test` variables, respectively.

X\_train = np.array(h5\_train["/X\_train"])

X\_test = np.array(h5\_test["/X\_test"])

print(X\_train.shape)

print(X\_test.shape)

Finally, the data is loaded from the HDF5 files into NumPy arrays. The datasets 'X\_train' and 'X\_test' inside the respective HDF5 files are accessed using their paths (`"/X\_train"` and `"/X\_test"`), and then converted to NumPy arrays using `np.array()`. The shapes of the loaded arrays are printed to verify the data's dimensions.

After executing this code, `X\_train` and `X\_test` will contain the training and testing data, respectively, loaded from the HDF5 files.

# PCA

1. `image\_grid(D, H, W, cols=10, scale=1)`: This function displays a grid of images stored in the array `D`. `H` and `W` represent the height and width of the images, respectively. `cols` specifies the number of columns in the grid, and `scale` is an optional parameter to adjust the size of the grid.

2. Displaying training images:

image\_grid(X\_train, H, W)

plt.show()

This code uses the `image\_grid` function to display a grid of training images stored in the `X\_train` array. The images are visualized using `plt.imshow` and the grayscale color map. The `plt.show()` function is used to display the grid.

3. Displaying testing images:

image\_grid(X\_test, H, W)

plt.show()

Similar to the previous section, this code displays a grid of testing images stored in the `X\_test` array.

4. Finding the mean image:

mean\_image = np.mean(X\_train, axis=0)

plt.imshow(np.reshape(mean\_image, [H, W]), cmap=plt.get\_cmap("gray"))

plt.show()

Here, the mean image is computed by taking the average of all images in the `X\_train` array along the specified axis (axis=0). The mean image is then reshaped and displayed using `plt.imshow` with the grayscale color map.

5. Performing Principal Component Analysis (PCA):

pca = PCA(n\_components=n\_components, svd\_solver='randomized', whiten=True).fit(X\_train)

eigenfaces = pca.components\_

PCA is applied to the training data (`X\_train`) using `sklearn.decomposition.PCA`. The `n\_components` parameter specifies the number of principal components to retain. The `eigenfaces` variable contains the top `n\_components` eigenfaces extracted from the dataset.

6. Displaying the eigenfaces:

image\_grid(eigenfaces[:,:], H, W)

plt.show()

The eigenfaces (principal components) obtained from the PCA are displayed using the `image\_grid` function.

7. Feature extraction using PCA:

def featureExtraction(train, test, n):

    pca = PCA(n\_components=n)

    pca.fit(train)

    X\_pca\_train = pca.transform(train)

    X\_pca\_test = pca.transform(test)

    return X\_pca\_train, X\_pca\_test

X\_pca\_train, X\_pca\_test = featureExtraction(X\_train, X\_test, 40)

This function performs feature extraction using PCA on the training and testing data. It creates a `PCA` object with `n` components, fits it to the training data, and then transforms both the training and testing data using the learned PCA model. The transformed data, `X\_pca\_train` and `X\_pca\_test`, are returned.

8. Plotting cumulative explained variance:

plt.plot(np.cumsum(pca.explained\_variance\_ratio\_))

plt.xlabel('number of components')

plt.ylabel('cumulative explained variance')

plt.show()

This code generates a plot showing the cumulative explained variance as a function of the number of components used in PCA. The explained variance ratio for each component is accessed via `pca.explained\_variance\_ratio\_`.

9. Calculating distances:

intra\_class\_dist = np.sum(np.power((X\_pca\_train[1,:] - X\_pca\_test[1,:]),2))

inter\_class\_dist = np.sum(np.power((X\_pca\_train[1,:] - X\_pca\_test[6,:]),2))

print("Intra-class distance: %d" % (intra\_class\_dist))

print("Inter-class distance: %d" % (inter\_class\_dist))

This section computes the distances between two samples in the transformed feature space (`X\_pca\_train` and `X\_pca\_test`). The `np.power` function is used to compute the element-wise squared differences between the two samples, and `np.sum` sums up the squared differences. The distances are then printed as the intra-class distance and inter-class distance.

**SVM/SVC**

The code trains a Support Vector Classifier (SVC) with an RBF kernel on the transformed feature vectors `X\_pca\_train` and corresponding labels `Y\_train`. It then uses the trained classifier to make predictions on the transformed test data `X\_pca\_test`. Here's a breakdown of the remaining code:

1. Printing model parameters:

print("Model parameters:")

clf\_trained.get\_params()

This code prints the parameters used in the trained SVC model.

2. Calculating and printing accuracy score:

a\_score = metrics.accuracy\_score(Y\_test, pred)

print("Accuracy:", a\_score)

The accuracy score of the classifier is calculated using `metrics.accuracy\_score` by comparing the predicted labels `pred` with the true labels `Y\_test`. The accuracy score is then printed.

3. Printing classification report:

print(classification\_report(Y\_test, pred))

This code prints a classification report, which includes precision, recall, F1-score, and support for each class based on the predicted labels `pred` and the true labels `Y\_test`.

4. Creating and plotting confusion matrix:

mat = confusion\_matrix(Y\_test, pred)

accurancy = accuracy\_score(Y\_test, pred)

plt.figure(figsize=(11,11))

ax = sns.heatmap(mat[0:10, 0:10], annot=True, fmt=".3f", linewidths=.5, square=True, cmap='Reds')

plt.ylabel('Actual label')

plt.xlabel('Predicted label')

all\_sample\_title = 'Accuracy Score : {0}'.format(accurancy\*100)

plt.title(all\_sample\_title, size=15)

bottom, top = ax.get\_ylim()

ax.set\_ylim(bottom + 0.5, top - 0.5)

plt.show()

This code calculates the confusion matrix using `confusion\_matrix` based on the true labels `Y\_test` and the predicted labels `pred`. The accuracy score is also calculated. Then, a heatmap of the confusion matrix is created and displayed using `sns.heatmap`.

5. Printing misclassified samples:

print("Actual     Predicted ")

for p in range(len(Y\_test)):

  if Y\_test[p] != pred[p]:

    print(Y\_test[p], "      ", pred[p])

This code loops over the test samples and prints the true label and predicted label for any misclassified samples where `Y\_test[p] != pred[p]`.

These sections of code provide insights into the performance and evaluation of the trained classifier, including accuracy, classification report, confusion matrix, and misclassified samples.