PRML_(CSL2050) Programming Assignment 3

Report

Name-Soumen Kumar
Roll No.-B22ES006

Question1-Implementing Perceptron-based Classification

In this problem, the goal is to implement Perceptron to solve the linear classification problem.

Task-0: Generate a synthetic 4-dimensional dataset:

The **generate_dataset** function generates synthetic data tailored for binary classification. It randomly generates weights for a linear function, including a bias term and four feature weights drawn from a uniform distribution between -1 and 1. Random input vectors are then created, each composed of four integer-valued features ranging from 0 to 10, simulating typical dataset features. For each input vector, the linear function f(x) is evaluated, determining the label based on whether f(x) is greater than or equal to zero. The resulting labels indicate positive or negative examples. Finally, the function prints the generated weights for reference, providing insight into the dataset's characteristics. Overall, generate_dataset encapsulates the process of generating synthetic data, enabling further analysis and experimentation in binary classification tasks.

```
# Generation of synthetic data
def generate_dataset(num_samples, seed=None):
    # Set seed for reproducibility
    np.random.seed(seed)

    # Generate random weights as floats
    weights = np.random.uniform(-1, 1, size=5).astype(np.float32)  #
Including bias term

    # Generate random input vectors
    X = np.random.randint(0, 10, size=(num_samples, 4), dtype=np.int32)

    # Calculate f(x) for each input vector
    f_values = np.dot(X, weights[1:]) + weights[0]
```

```
    # Assign labels based on f(x) using thresholding
    y = np.where(f_values >= 0, 1, 0)

    print("Generated Weights (w0, w1, w2, w3, w4):", weights)  # Print
generated weights

    return X, y, weights
```

Saving Data to Files (save_to_file and save_test_data_without_labels functions):

- This part of the code defines functions for saving datasets to text files.
- **save_to_file** function saves datasets with both input features and labels.
- **save_test_data_without_labels** function saves test data without labels, which is useful for predictions or evaluation.

```
def save_to_file(X, y, filename):
    with open(filename, 'w') as file:
        file.write(f"{len(X)}\n")
        for i in range(len(X)):
            line = ' '.join(map(str, X[i].tolist() + [y[i]])) + '\n'
            file.write(line)

def save_test_data_without_labels(X, filename):
    with open(filename, 'w') as file:
        file.write(f"{len(X)}\n")
        for i in range(len(X)):
            line = ' '.join(map(str, X[i].tolist())) + '\n'
            file.write(line)
```

Splitting Dataset (**split_dataset function)**:

- This function splits the generated dataset into training and testing sets with a 70:30 split.
- It shuffles the dataset to ensure randomness.
- The split datasets are returned for further processing.

```
def split_dataset(X, y):
    num_samples = len(X)
    num_train_samples = int(num_samples * 0.7)

    # Shuffle the dataset
    indices = np.arange(num_samples)
    np.random.shuffle(indices)
```

```
    X_shuffled = X[indices]
    y_shuffled = y[indices]

    # Split the dataset
    X_train = X_shuffled[:num_train_samples]
    y_train = y_shuffled[:num_train_samples]
    X_test = X_shuffled[num_train_samples:]
    y_test = y_shuffled[num_train_samples:]

    return X_train, y_train, X_test, y_test
```

Main Execution:

- The script sets the current directory for file operations.
- It generates a synthetic dataset with 1000 samples using a specified seed for reproducibility.
- Data is saved to files: data.txt for the entire dataset, train.txt and test.txt for training and testing sets, respectively, and test_data_without_labels.txt for test data without labels.
- Finally, it splits the dataset into training and testing sets and saves them into separate files.

These parts together facilitate the creation of a synthetic dataset, its storage for later use, and the division of the dataset into train and test sets for machine learning tasks. They form essential steps in preparing data for model training and evaluation.

Task-1: Write training code for the perceptron learning algorithm. (Do not forget to normalize the data both during training and testing). (Deliverable: train.py)

The Perceptron class encapsulates the functionality for implementing a perceptron model. Upon initialization, the object is set with default parameters, including the learning rate and the number of epochs for training(0.01,700). The train method trains the perceptron model using the provided training data. During training, the algorithm iterates over the training data for a specified number of epochs, adjusting the model's weights based on prediction errors to minimize classification errors. The predict method is responsible for predicting the class label for a given input vector, utilizing the trained perceptron model. Additionally, the class includes a normalize_data method, which ensures that the input data is normalized to have a unit norm, enhancing the model's stability and convergence during training.

Utility Functions:

- **load_data**: Reads data from a file and returns feature vectors and corresponding labels.
- **save_weights**: Saves the trained weights to a file.

Main Execution:

- It checks if the correct number of command-line arguments is provided (expecting the filename of the training data).
- It loads the training data from the file specified in the command line argument.
- It creates an instance of the Perceptron class and trains it using the loaded training data.
- Trained weights are saved to a file named "weights.txt" using the save_weights function.
- Upon completion, it prints a message indicating that training is over and weights are saved.

This code can be used to train a perceptron model on a given dataset and save the learned weights for later use

```
class Perceptron:
    def __init__(self, learning_rate=0.01, num_epochs=700):
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.weights = None

    def train(self, X_train, y_train):
        # Normalize data
        X_train_normalized = self.normalize_data(X_train)

        # Initialize weights
        num_features = X_train_normalized.shape[1]
        self.weights = np.zeros(num_features + 1)  # Additional weight
        for bias term

        # Train the perceptron
        for _ in range(self.num_epochs):
            for i in range(len(X_train_normalized)):
                prediction = self.predict(X_train_normalized[i])
                error = y_train[i] - prediction
                self.weights[:-1] += self.learning_rate * error *
X_train_normalized[i]
```

```
                self.weights[-1] += self.learning_rate * error  # Update
bias term

    def predict(self, x):
        return 1 if np.dot(x, self.weights[:-1]) + self.weights[-1] >= 0
else 0

    def normalize_data(self, X):
        return X / np.linalg.norm(X)

def load_data(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()
        num_samples = int(lines[0])
        data = np.zeros((num_samples, 4))
        labels = np.zeros(num_samples)
        for i, line in enumerate(lines[1:]):
            values = line.strip().split()
            data[i] = list(map(int, values[:4]))
            labels[i] = int(values[4])
    return data, labels

def save_weights(weights, filename):
    with open(filename, 'w') as file:
        file.write(" ".join(map(str, weights)))
```

Task-2: Write testing and evaluation code. Report accuracy.

This code segment is designed to evaluate the performance of a trained perceptron model using test data.

The Perceptron class extends its functionality by including the load_weights method, which facilitates the loading of weights from a file and assigns them to the perceptron object. This method streamlines initialising the perceptron with pre-trained weights, enabling seamless integration with previously trained models or weight configurations. Subsequently, the predict method predicts class labels for input feature vectors using the loaded weights. Before making predictions, this method incorporates a bias term into the input feature vectors, a crucial step in the perceptron algorithm for accommodating varying offsets in the data distribution. By integrating the bias term, the perceptron can effectively capture non-linear relationships between features and labels, enhancing its predictive capabilities

Utility Functions:

- **load_data**: Loads test data from a file. It determines the number of features based on whether the data has labels or not.
- **load_weights**: Loads weights from a file.
- **evaluate_model**: Calculates the accuracy of the model predictions compared to the true labels.

Main Execution:

- It checks if the correct number of command-line arguments is provided (expecting filenames of the test data and the weights).
- It loads test data from the file specified in the first command-line argument.
- It loads weights from the file specified in the second command-line argument.
- It creates an instance of the Perceptron class and assigns the loaded weights to it.
- It predicts labels for the test data using the perceptron model.
- Finally, it prints the predicted labels for the test data.

This script can be executed from the command line to assess the performance of a trained perceptron model on unseen test data. It provides a straightforward evaluation of the model's accuracy.

```
class Perceptron:
    def __init__(self):
        self.weights = None

    def load_weights(self, weights_filename):
        # Load weights from a file and assign them to the perceptron
object
        with open(weights_filename, 'r') as file:
            self.weights = np.array(list(map(float,
file.readline().split())))

    def predict(self, X):
        # Predict class labels for input feature vectors using the loaded
weights
        X_with_bias = np.c_[X, np.ones(X.shape[0])]  # Add bias term
        return np.where(np.dot(X_with_bias, self.weights) >= 0, 1, 0)

def load_data(filename, has_labels=True):
    # Load data from a file
    with open(filename, 'r') as file:
        lines = file.readlines()
```

```
        num_samples = int(lines[0])
        num_features = 4 if has_labels else 5
        start_line = 1 if has_labels else 0
        data = np.zeros((num_samples, num_features))
        labels = np.zeros(num_samples, dtype=int) if has_labels else None
        for i, line in enumerate(lines[start_line:]):
            values = line.strip().split()
            data[i] = list(map(float, values[:num_features]))  # Load as
floats
            if has_labels:
                labels[i] = int(values[-1])
    if has_labels:
        return data, labels
    else:
        return data

def load_weights(weights_filename):
    # Load weights from a file
    with open(weights_filename, 'r') as file:
        weights = np.array(list(map(float, file.readline().split())))
    return weights

def evaluate_model(y_pred, y_true):
    # Evaluate the model's performance
    num_correct = np.sum(y_pred == y_true)
    accuracy = num_correct / len(y_true)
    return accuracy
```

Accuracy-

Overall Accuracy (94%):

- This indicates the accuracy of the perceptron model across the entire test dataset.
- It means that out of all the samples in the test dataset, 94% of them were correctly classified by the perceptron model.

Task-3: Report results with training using 20%, 50%, and 70% of synthetic Data.

SubSet Size	Accuracy
20%	**93.33%**
50%	**92.67%**
70%	**93.33%**

Accuracy for 20% of the Data (93.33%):

- This accuracy value represents the performance of the perceptron model when trained on a randomly selected 20% subset of the train data and tested on test data
- It means that when considering only 20% of the training dataset, the perceptron model correctly classified approximately 93.33% of the samples in that subset.

Accuracy for 50% of the Data (92.67%):

- Similarly, this accuracy value represents the performance of the perceptron model when trained on a randomly selected 50% subset of the train data and tested on test data
- It means that when considering only 50% of the training dataset, the perceptron model correctly classified around 92.67% of the samples in that subset.

Accuracy for 70% of the Data (93.33%):

- This accuracy value represents the performance of the perceptron model when trained on a randomly selected 70% subset of the train data and tested on test data
- It indicates that when considering 70% of the training dataset, the perceptron model achieved an accuracy of approximately 93.33%.

These accuracy values provide insights into how the model performs across different portions of the train data. The slight variation in accuracy across subsets may be due to differences in the distribution of samples or specific characteristics of those subsets. Overall, achieving high accuracy across various subsets indicates the robustness of the perceptron model in classifying unseen data.

Regarding the use of data.txt(how synthetic data is used)-
Firstly, two files, train.txt and test.txt, are created from data.txt using 70:30 split…train.txt contains labels, whereas test.txt doesn't contain labels…first, the model is trained using train.txt and then tested by test.txt to obtain an overall accuracy of 94%..after that as per task 3 the train.txt is splitted internally(temp files creation) into 20%,50%,70% of total data on a random basis. Then, the model is trained using that. After this, the model is tested using test.txt to obtain an accuracy of around 94% in all three cases.

Qno2:Eigenfaces –
a dimensionality reduction technique based on PCA for face recognition.

Task-1: Data Preprocessing (a) Load the LFW dataset using the Scikit-learn's fetch LFW people function. (b) Split the dataset into training and testing sets using an 80:20 split ratio.

Importing Libraries:

- **from sklearn.datasets import fetch_lfw_people**:
  This imports the function fetch_lfw_people from the sklearn.datasets module, which is used to load the LFW dataset.
- **from sklearn.model_selection import train_test_split**:
  This imports the train_test_split function from the sklearn.model_selection module is used to split the dataset into training and testing sets.

Loading LFW Dataset:

- **lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)**:
  This function fetches the LFW dataset, ensuring that each person in the dataset has at least 70 images. The resize parameter resizes each image to 40% of its original size.

Extracting Images and Labels:

- **X_faces = lfw_people.images**:
  This extracts the images from the loaded dataset.
- **y = lfw_people.target**:
  This extracts the corresponding target labels (identifiers of the person in each image).

Splitting the Dataset:

- **X_train, X_test, y_train, y_test = train_test_split(X_faces, y, test_size=0.2, random_state=84):**
  This function splits the dataset into training and testing sets with an 80:20 ratio. It randomly shuffles the data before splitting, ensuring that both sets represent the overall dataset. The random_state parameter ensures reproducibility by fixing the seed for random number generation.

Printing Shapes of Sets:

- **print("Shape of X_train:", X_train.shape):**
  This prints the shape of the training set (number of samples, image height, image width).

- **print("Shape of y_train:", y_train.shape)**: This prints the shape of the training labels.
- **print("Shape of X_test:", X_test.shape)**: This prints the shape of the testing set.
- **print("Shape of y_test:", y_test.shape**): This prints the shape of the testing labels.

```
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split

# Load LFW dataset
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# Extract the images and corresponding target labels
X_faces = lfw_people.images
y = lfw_people.target

# Split the dataset into training and testing sets with 80:20 split ratio
X_train, X_test, y_train, y_test = train_test_split(X_faces, y,
test_size=0.2, random_state=84)

# Print the shapes of the training and testing sets
print("Shape of X_train:", X_train.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_test:", y_test.shape)
```

Task-2: Eigenfaces Implementation (a) Implement Eigenfaces using Principal Component Analysis (PCA). Set an appropriate value for n_components and explain your choice in the report

The PCA class encapsulates key functionalities for principal component analysis. Upon initialisation, the object is configured with the number of components to retain (n_components), and the components attribute is initialised to None. The fit method fits the PCA model to the input data. This process involves several crucial steps: firstly, it computes the covariance matrix of the training data; next, it computes the eigenvectors and eigenvalues of the covariance matrix. Subsequently, the eigenvectors are sorted based on their corresponding eigenvalues, with the top n_components eigenvectors selected as principal components. Additionally, the

method computes the explained variance ratio, providing insight into the proportion of variance captured by each principal component and sets the necessary attributes accordingly. On the other hand, the transform method transforms the input data into a reduced-dimensional space using the principal components obtained during fitting. This transformation enables dimensionality reduction while preserving the essential information contained in the original data.

```
class PCA:
    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None

    def fit(self, X):
        # Calculate the covariance matrix of the training data
        cov_matrix = np.cov(X.T)

        # Compute the eigenvectors and eigenvalues of the covariance
matrix
        eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

        # Sort the eigenvectors based on their corresponding eigenvalues
        sorted_indices = np.argsort(eigenvalues)[::-1]
        sorted_eigenvectors = eigenvectors[:, sorted_indices]

        # Choose the top n_components eigenvectors as principal components
        self.components = sorted_eigenvectors[:, :self.n_components]

        # Compute explained variance ratio
        total_variance = np.sum(eigenvalues)
        self.explained_variance_ratio_ =
eigenvalues[sorted_indices][:self.n_components] / total_variance

        # Set the attribute n_components_ to the actual number of
components
        self.n_components_ = self.components.shape[1]

    def transform(self, X):
        # Project the data onto the principal components
        return np.dot(X, self.components)
```

Data processing:

- **X_train_flat**: Reshapes the 2D images into 1D vectors to prepare them for PCA.
- Normalises pixel values to the range [0, 1] by dividing by 255.0.

PCA Initialization and Fitting:

- **n_components**: Defines the desired number of principal components.
- **PCA**: Initializes the PCA object with the specified number of components.
- **PCA.fit**: Fits the PCA model to the training data, computing the principal components.

Transforming Data:

- **X_train_pca**: Transforms the training data into the reduced-dimensional space using the fitted PCA model.

Explained Variance Ratio Plot:

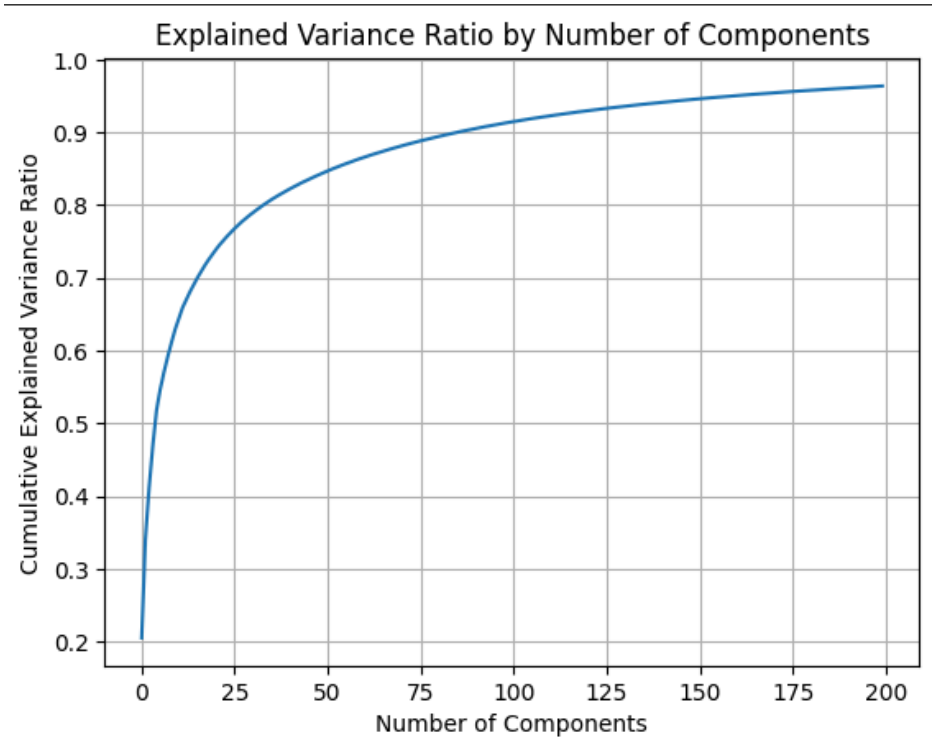
- Plots the cumulative explained variance ratio against the number of components.
- This plot helps visualise how much variance in the data is explained by each additional component.

Explained Variance Ratio Printout:

- Prints the explained variance ratio for each component.
- This provides insight into the contribution of each component to the total variance in the data.

Number of Components Selected:

- Prints the number of components selected based on the explained variance ratio.
- This indicates how many principal components are retained based on a specified threshold or criterion.



```
Explained Variance Ratio for each Component:
Component 1: 0.206
Component 2: 0.136
Component 3: 0.071
Component 4: 0.058
Component 5: 0.049
Component 6: 0.029
Component 7: 0.023
Component 8: 0.021
Component 9: 0.019
Component 10: 0.018

Number of components selected based on explained variance ratio: 200
```

Explained Variance Ratio for each Component:

Each line represents the explained variance ratio for a specific principal component.

- For example, "Component 1: 0.206" indicates that the first principal component explains approximately 20.6% of the total variance in the data.
- Similarly, "Component 2: 0.136" indicates that the second principal component explains approximately 13.6% of the total variance, and so on.

Number of components selected based on explained variance ratio (200):

- This value represents the total number of principal components selected based on the cumulative explained variance ratio.
- In this case, 200 components are selected, which means these 200 components collectively explain a significant portion of the variance in the data.
- By retaining these 200 components, we aim to capture as much information as possible from the original dataset while reducing its dimensionality.

Task-3: Model Training (a) Choose a classifier for Eigenfaces (e.g., K-Nearest Neighbors) and train the classifier using the transformed training data.

KNN Classifier:

- **__init__**: Initializes the KNN classifier with a default value of n_neighbors set to 5.
- **fit**: Stores the training data (X_train) and corresponding labels (y_train).
- **predict**: Predicts the class labels for the given test data (X_test) based on the nearest neighbors in the training data. It computes the distances between each test point and all training points, finds the nearest neighbors, and predicts the class based on majority voting among the nearest neighbors.

Initializing KNN Classifier:

- Initializes a KNN classifier object with 5 neighbors.

Training KNN Classifier:

- Trains the KNN classifier using the transformed training data (X_train_pca).

Transforming Testing Data for KNN:

- Reshapes and normalizes the testing data (X_test) similar to the training data.
- Transforms the testing data into the reduced-dimensional space using PCA (X_test_pca).

Linear Regression Classifier:

- **__init__**: Initializes the Linear Regression classifier.
- **fit**: Fits the Linear Regression model to the training data.
- **predict**: Predicts the class labels for the given test data using the trained Linear Regression model.

Training and Evaluating Linear Regression Classifier:

- Initializes a Linear Regression classifier object.
- Trains the Linear Regression classifier using the transformed training data.
- Predicts class labels for the testing data using the trained Linear Regression model (y_pred_lr).

Decision Tree Classifier:

- **__init__**: Initializes the Decision Tree classifier.
- **fit:** Fits the Decision Tree model to the training data.
- **predict**: Predict the class labels for the given test data using the trained Decision Tree model.

Training and Evaluating Decision Tree Classifier:

- Initialises a Decision Tree classifier object.
- Trains the Decision Tree classifier using the transformed training data.
- Predicts class labels for the testing data using the trained Decision Tree model (y_pred_dt).

This code demonstrates the training and evaluation of different classifiers (KNN, Linear Regression, and Decision Tree) using the transformed training data (X_train_pca) and testing data (X_test_pca) obtained through PCA dimensionality reduction. Each classifier follows a similar pattern of initialisation, training, and prediction, providing different approaches to the classification task.

Task-4: Model Evaluation: (a) Use the trained Eigenfaces classifier to make predictions on the Eigenfaces-transformed testing data. (b) Calculate and report accuracy. (c) Visualise a subset of Eigenfaces and report the observations. (Report on what type of test images the model is failing, and mention ways to improve the model)

Predicting Labels and Calculating Accuracy:

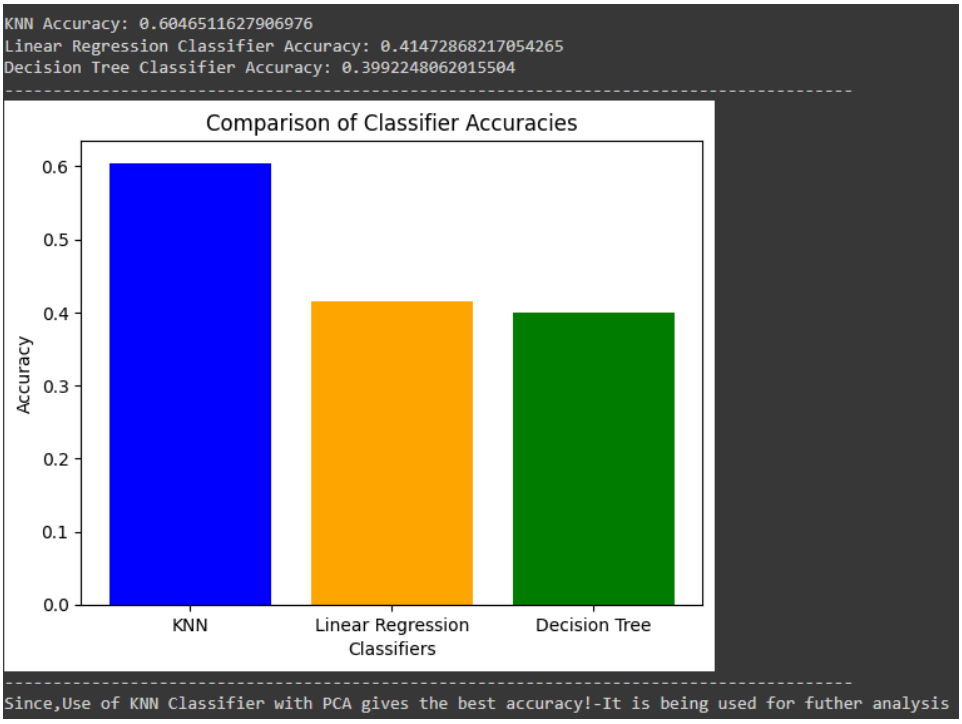
- Labels are predicted for the transformed testing data using each classifier: KNN, Linear Regression, and Decision Tree.
- Accuracy is calculated for each classifier by comparing the predicted and actual labels (y_test).

Plotting Comparison Graph:

- A bar plot is generated to compare the accuracies of different classifiers.
- Each classifier is represented on the x-axis, and the corresponding accuracy is represented on the y-axis.
- This visualisation helps in comparing the performance of classifiers at a glance.

Selecting the Best Classifier:

- Based on the comparison of accuracies, the KNN classifier with PCA dimensionality reduction is identified as the best-performing classifier.
- A message is printed indicating the selection of the KNN classifier for further analysis.

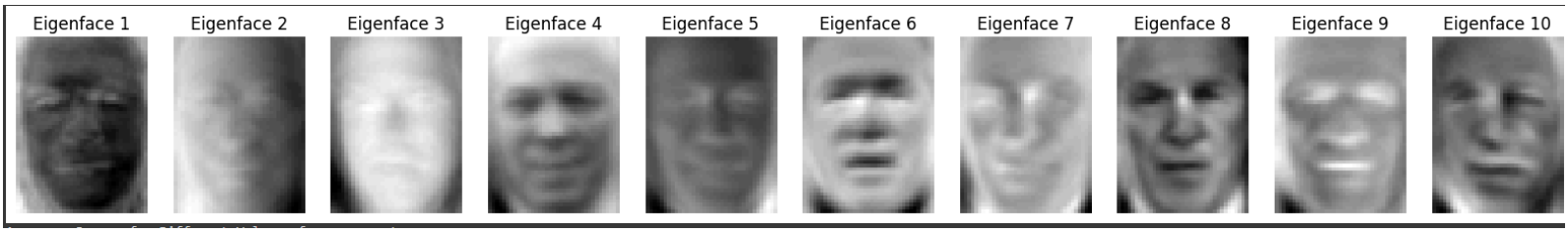


Visualising Eigenfaces:

- A subset of eigenfaces (principal components) obtained from PCA is visualised.
- Each eigenface represents a principal component that captures important patterns or features in the data.
- The eigenfaces help in understanding the underlying structure of the data and how it is represented in the reduced-dimensional space.

Observations and Recommendations(Ways to Improve the Model):

- While the overall accuracy of the KNN classifier with PCA is high, it's essential to analyse where the model may fail.
- One way to improve the model is to analyse the types of test images where the model is failing. This could involve examining misclassified images to identify patterns or features challenging for the model to recognise.
- Additionally, considering **data augmentation techniques such as rotating, flipping, or adding noise** to the training images could help the model generalise better to unseen variations in the test data.
- **Fine-tuning hyperparameters of the classifiers**, such as the number of neighbours in KNN or the maximum depth in Decision Trees, could also lead to performance improvements.
- Lastly, **experimenting with different dimensionality reduction techniques or adjusting the number of principal components** in PCA may also impact and improve model performance.



Task-5: Experiment with different values of n_components in PCA and observe the impact on the performance metrics (accuracy).

List of n_components to Experiment With:

- We define a list of n_components_values containing the values of n_components that we want to experiment with. These values represent the number of principal components to retain during dimensionality reduction.
-

Dictionary to Store Accuracy for Each Value of n_components:

- We initialise an empty dictionary accuracy_scores to store the accuracy obtained for each value of n_components.

Loop Through Different Values of n_components:

- We iterate over each value of n_components in the n_components_values list.
- For each value of n_components, we:
  - Initialise a PCA object with the current value of n_components.
  - Fit the PCA model on the flattened training data (X_train_flat).
  - Transform both the training and testing data into the reduced-dimensional space using PCA.
  - Train the KNN classifier using the transformed training data.
  - Predict labels for the transformed testing data using the trained KNN classifier.
  - Calculate the accuracy of the classifier on the testing data.
  - Store the accuracy in the accuracy_scores dictionary with the current value of n_components as the key.

Print Accuracy Scores:

- After iterating through all values of n_components, we print the accuracy scores obtained for each value.
- This provides insight into how the accuracy varies with different numbers of principal components.

This experimentation allows us to understand the impact of varying the number of principal components on the performance of the KNN classifier. It helps determine an optimal value of n_components that balances model complexity and predictive performance.

```
Accuracy Scores for Different Values of n_components:
n_components=50: Accuracy=0.5930
n_components=100: Accuracy=0.5930
n_components=150: Accuracy=0.5930
n_components=200: Accuracy=0.6047
n_components=250: Accuracy=0.5969
```

—----------