**Eigenface Generator**

**Description**

This code provides the EigenfaceGenerator class, which generates eigenfaces from a set of facial images. Eigenfaces are the principal components of the distribution of faces, representing the directions of greatest variance in the data. They form a basis set for representing faces and are commonly used for facial recognition and dimensionality reduction. The class also includes methods for visualizing the generated eigenfaces, the mean face, and the explained variance ratio.

**Usage**

1. Installation

Ensure you have the required libraries installed:

*pip install scikit-learn Pillow matplotlib tqdm*

1. Import the Class

*from src.modules.eigenface import EigenfaceGenerator*

1. Prepare Your Data
   * Image Loading: Load your facial images into a list of PIL Image objects.
   * Resizing: Crucially, ensure all images have the same dimensions. Use PIL.Image.resize((width, height)) to resize them if necessary before passing them to the EigenfaceGenerator. Inconsistent image sizes will cause errors.
   * Grayscale Conversion: The EigenfaceGenerator handles grayscale conversion internally, so you can pass either color or grayscale images.
2. Create an EigenfaceGenerator Object

*eigenface\_generator = EigenfaceGenerator(images, n\_components=5)*

* images: A list of PIL Image objects (not NumPy arrays directly). The class handles conversion to NumPy arrays internally.
* n\_components: (Optional) The number of principal components (eigenfaces) to generate. Defaults to 5. Must be less than or equal to the number of images, and less than the number of pixels.

1. Generate Eigenfaces

*eigenface\_generator.generate()*

This method performs the core PCA calculation:

* Converts images to grayscale (if not already).
* Flattens the images into 1D NumPy arrays.
* Performs Principal Component Analysis (PCA) on the flattened images.
* Reshapes the principal components into 2D eigenfaces.
* Calculates the mean face.

These steps are internal to the class; the user does not need to perform them manually.

1. Retrieve Results
   * Eigenfaces:

*eigenfaces = eigenface\_generator.get\_eigenfaces()*

* + Mean Face:

*mean\_face = eigenface\_generator.get\_mean\_face()*

* + PCA Object:

*pca = eigenface\_generator.get\_pca\_object()*

1. Visualization (Plotting)

The EigenfaceGenerator class provides methods for visualizing the results:

* Plot Eigenfaces:

*eigenface\_generator.plot\_eigenfaces(*

*output\_folder="path/to/save",*

*show\_plot=False)*

* Plot Mean Face

*eigenface\_generator.plot\_mean\_face(*

*output\_folder="path/to/save",*

*show\_plot=False)*

* Plot Explained Variance:

*eigenface\_generator.plot\_explained\_variance(*

*output\_folder="path/to/save",*

*show\_plot=False)*

1. Analysis

The EigenfaceGenerator class also provides a method for analyzing the eigenfaces and their relationships:

Analyze Eigenfaces:

eigenface\_generator.analyze\_eigenfaces(output\_folder="path/to/save")

This method performs the following analysis:

1. Checks for static components in the eigenfaces.
2. Calculates the cosine similarity between all pairs of eigenface vectors.
3. Generates a heatmap of the cosine similarity matrix.
4. Prints the number of eigenface vectors generated per user.

The analysis results are saved to an "eigenface\_analysis.txt" file in the specified output folder.

**Class methods :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Arguments** | **Returns** |
| init(self, images, n\_components=5) | Initializes the EigenfaceGenerator object. | images: List of PIL Image objects.  n\_components: (Optional) Number of principal components. Defaults to 5. | None |
| generate(self) | Performs PCA to generate eigenfaces and the mean face. | None | None |
| get\_eigenfaces(self) | Returns the generated eigenfaces. | None | List of 2D NumPy arrays  (the eigenfaces). |
| get\_mean\_face(self) | Returns the calculated mean face. | None | 2D NumPy array  (the mean face). |
| get\_pca\_object(self) | Returns the scikit-learn PCA object. | None | sklearn.decomposition.PCA object. |
| plot\_eigenfaces(self, output\_folder, show\_plot=False) | Plots and saves the eigenfaces | output\_folder: Path to save the plot.  show\_plot:(Optional) Whether to display the plot | None |
| plot\_mean\_face(self, output\_folder, show\_plot=False) | Plots and saves the mean face | output\_folder: Path to save the plot.  show\_plot:(Optional) Whether to display the plot | None |
| plot\_explained\_variance(self, output\_folder, show\_plot=False) | Plots and saves the explained variance | output\_folder: Path to save the plot.  show\_plot:(Optional) Whether to display the plot | None |
| plot\_analysis(self, output\_folder, show\_plots=False) | Plots and saves eigenfaces, the mean face and the explained variance ratio | output\_folder: Path to save the plot.  show\_plots: (Optional) Whether to display all plots. | None |

**Testing**

To run the provided unit tests (located in tests/test\_eigenface.py):

1. Open a terminal.
2. Navigate to the project root directory.
3. Run the tests using one of the following commands:
   * To run all tests in the tests directory:

*python -m unittest discover tests*

* + To run only the test\_eigenface.py tests:

*python -m unittest tests.test\_eigenface*

Important: Run unittest using python -m unittest from the project root. This ensures Python's module import system works correctly. Do *not* run the test file directly (e.g., python tests/test\_eigenface.py).

**Next Steps**

* Implement PEEP: Reprogram the PEEP (Privacy using Eigenface Perturbation) method with the three algorithms:
  + Laplacian mechanism (initial mechanism used by PEEP)
  + Gaussian mechanism
  + Uniform mechanism

PEEP applies differential privacy to the eigenfaces to protect the privacy of the facial images. This involves adding noise to the eigenfaces in a controlled manner, making it difficult to reconstruct the original facial images from the perturbed eigenfaces.

* Add Facial Recognition: Add a facial recognition system. This could involve:
  + Training a classifier (e.g., Support Vector Machine, k-Nearest Neighbors) on a set of labeled facial images (represented by their projections onto the eigenface space).
  + Using the trained classifier to predict the identity of a new facial image.
  + Evaluating the accuracy of the facial recognition system (e.g., using metrics like accuracy, precision, recall, F1-score).
* Experiment with Epsilon: Experiment with different values of epsilon (the privacy parameter in differential privacy) to find the optimal balance between privacy and recognition accuracy.
  + A higher epsilon value provides stronger privacy guarantees but may reduce recognition accuracy.
  + A lower epsilon value provides weaker privacy guarantees but may improve recognition accuracy.
* Handle different image formats: Add the possibility to handle jpg, jpeg and other usual image types
* Add exceptions: Add exceptions to handle different errors that could occur (wrong path, image not found,...)