“In The Name Of God”

**HW#4 Report**

Professor: Dr.Azmifar

Presenters:

Narges Dehghan

Masome Jafari

Table of Contents

**Abstract3**

**PCA** **4**

**Fisher LDA** 10

**Conclusion 15**

Abstract

This assignment encompasses a comprehensive exploration of implement PCA and Fisher methods and how to visualize the resulted probability density functions., fundamental techniques in the realm of machine learning and data analysis. The primary objectives are twofold: to understand the fundamental concepts and application of these methods and to apply this knowledge to real-world datasets.

**Principal Component Analysis (PCA)**

we aim to implement Principal Component Analysis (PCA) from scratch for facial emotion recognition.

In this part we present an analysis of facial expressions using the JAFFE dataset, which consists of images of Japanese female facial expressions. The goal is to explore and visualize the dataset, identify patterns in facial expressions, and perform dimensionality reduction to capture essential features.

**Data Overview**

The dataset is located in the 'jaffe' folder and contains images of facial expressions. The feelings associated with each image are extracted from the filenames and stored in a DataFrame. The images are resized to 64x64 pixels and flattened for further processing.

**Exploratory Data Analysis (EDA)**

**Original vs. Normalized Images**

The dataset is explored by comparing original and normalized images for each unique facial expression. The normalization involves subtracting the mean face from each image. This step aims to highlight facial expression features by removing background and lighting variations.

**Eigenfaces**

Eigenfaces are calculated by performing Principal Component Analysis (PCA) on the normalized images. The top 10 eigenfaces are visualized, providing insights into the most significant facial expression patterns present in the dataset.

**Dimensionality Reduction**

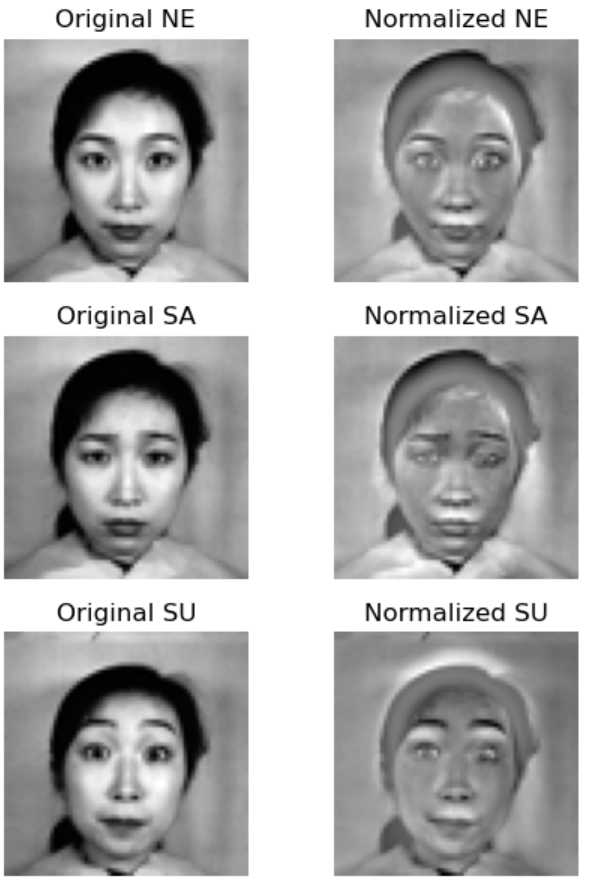
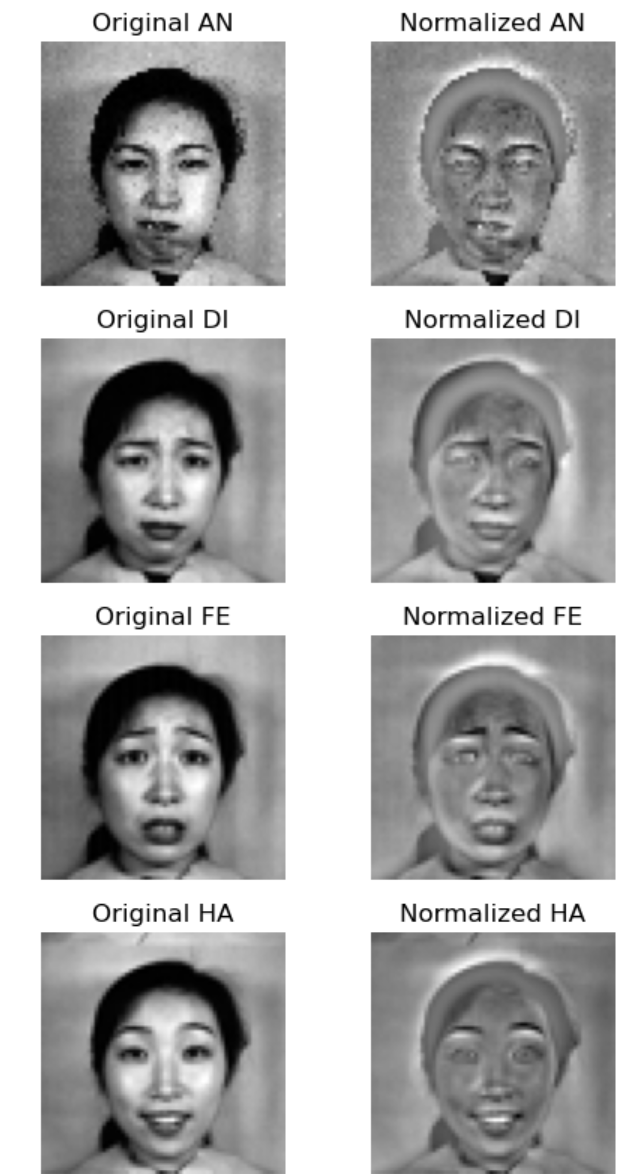
**2D and 3D Projections**

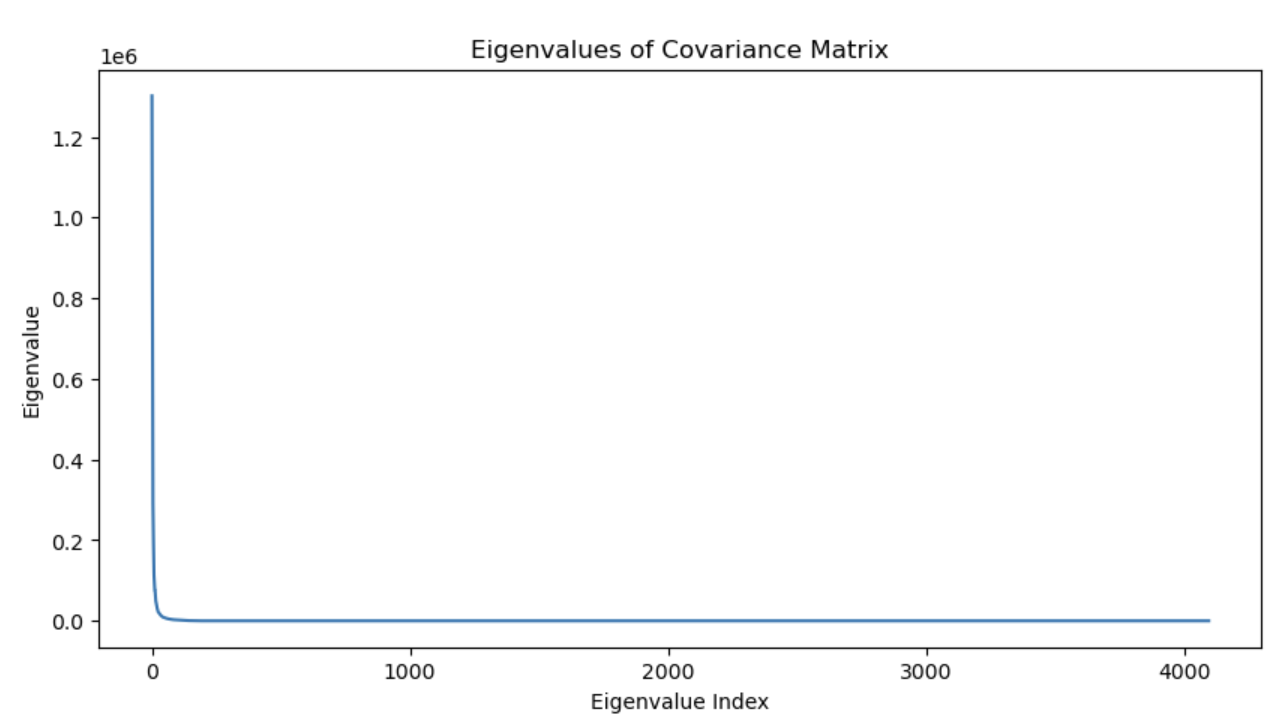
The dataset is projected onto lower-dimensional subspaces (2D and 3D) using the top eigenfaces. Scatter plots are created to visualize the distribution of facial expressions in these reduced spaces. Each point on the plot corresponds to an image, and different emotions are represented by distinct colors.

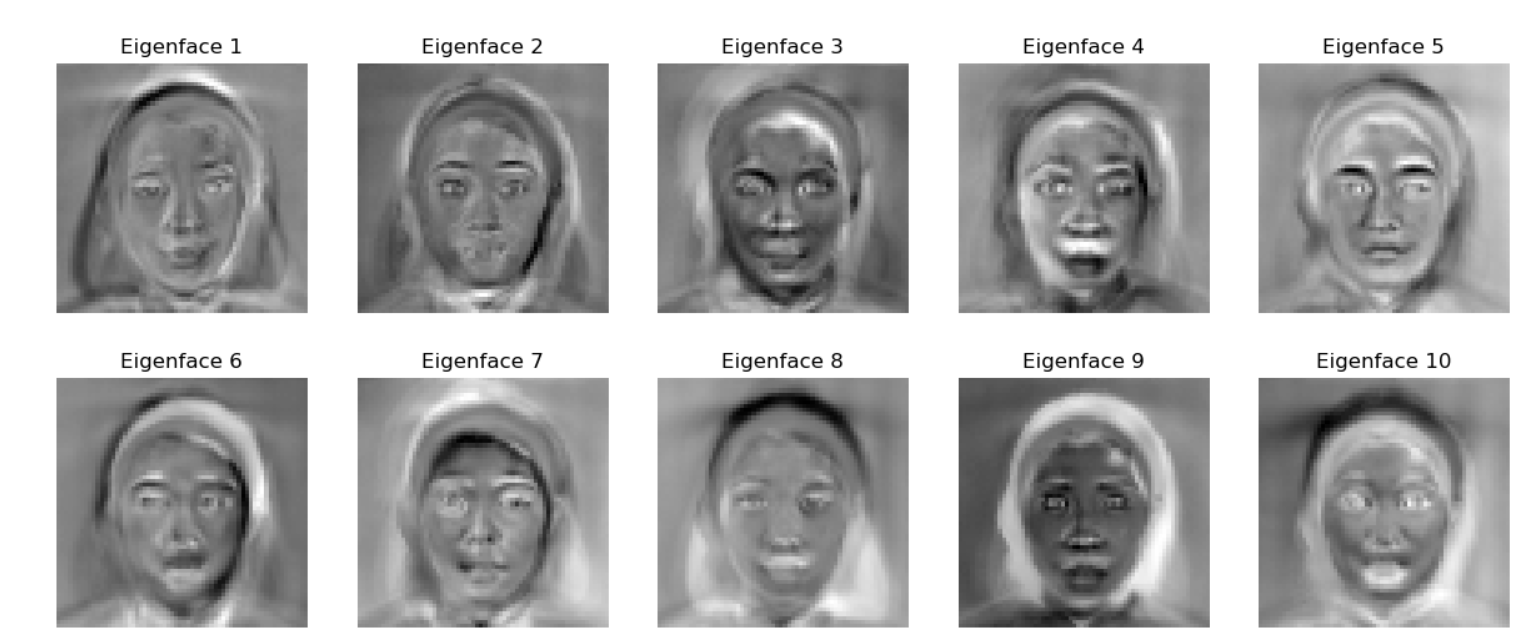
**Facial Expression Reconstruction**

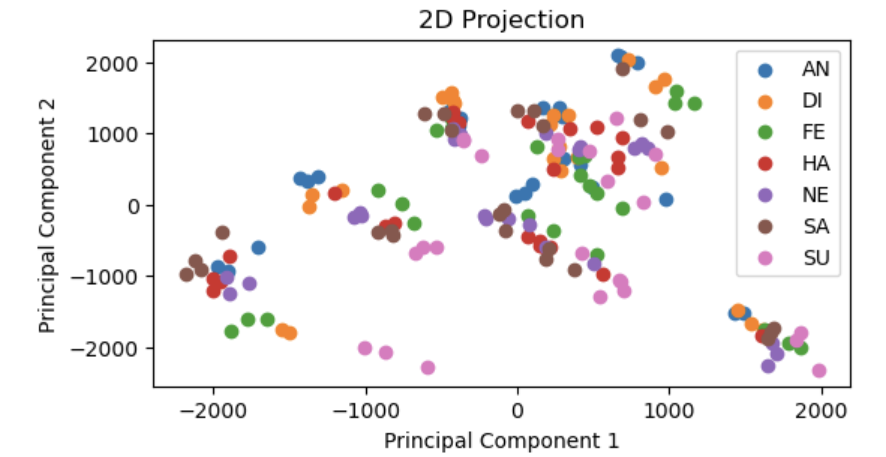
Random images are selected for reconstruction to demonstrate the ability to capture essential facial expression features. The original and reconstructed images are compared, showing the effectiveness of the dimensionality reduction and reconstruction process.

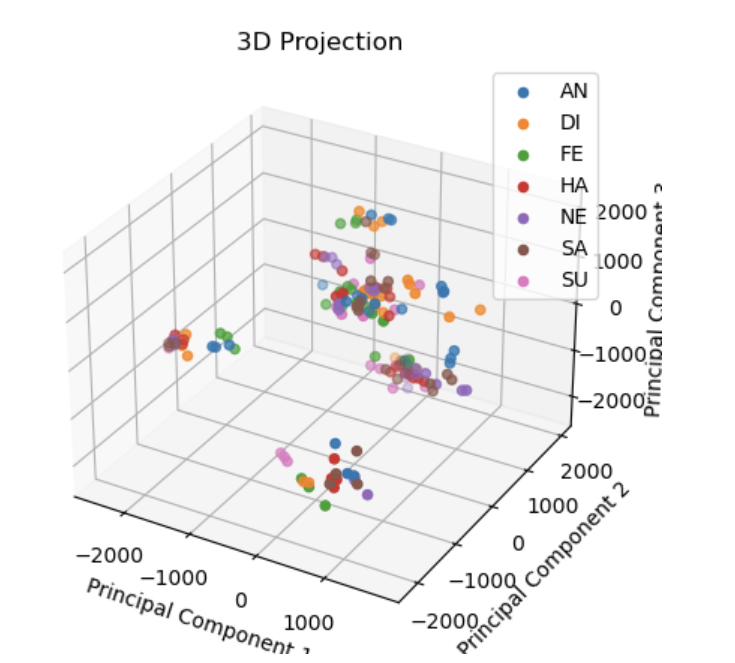
This analysis provides valuable insights into the facial expression patterns present in the JAFFE dataset. The use of eigenfaces and dimensionality reduction techniques allows for a compact representation of the data, aiding in the visualization and understanding of facial expressions. The presented scatter plots and reconstructed images offer a comprehensive view of the dataset's structure and potential applications in facial expression recognition. Further exploration and analysis can be performed to enhance the understanding of facial expressions and improve recognition models.







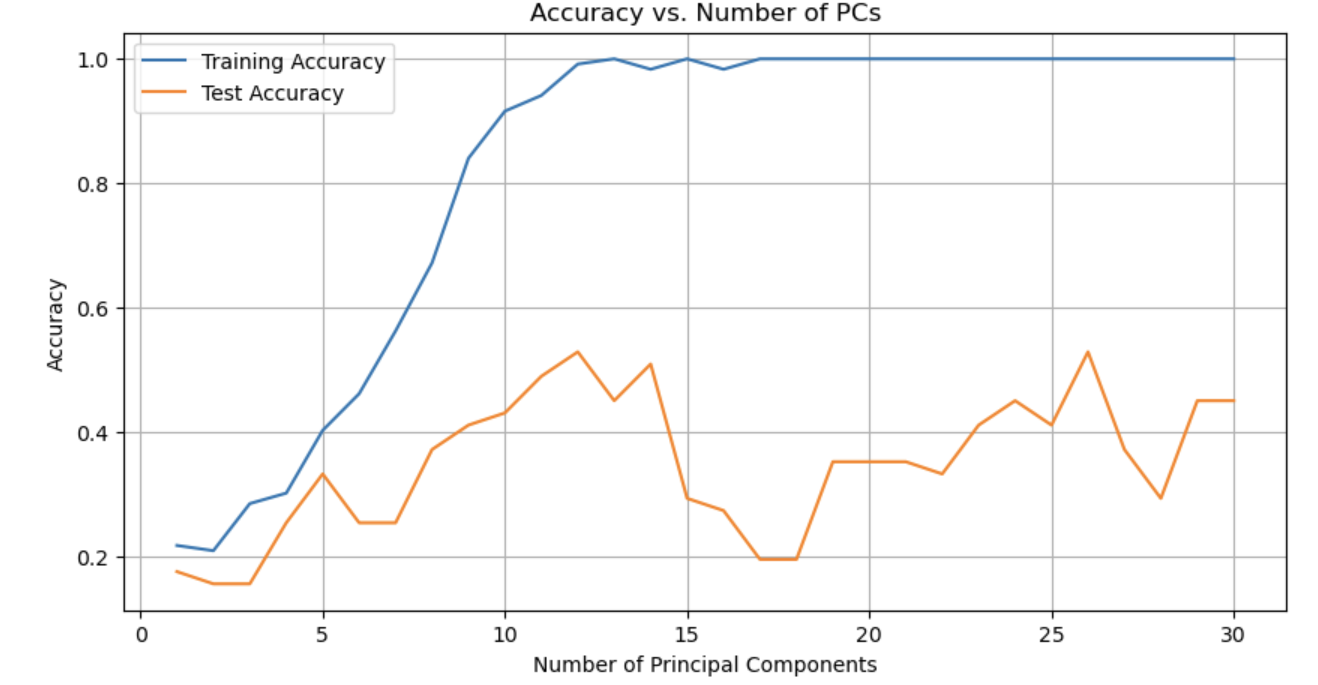






We achieve : Training Accuracy: 0.91 || Test Accuracy: 0.43

For QDA part, And the plot for the Bonus part.



**Fisher Linear Discriminant Analysis (Fisher LDA)**

**Abstract**

**Report: Implementation of Image Loading and Preprocessing**

The `load\_images` function was implemented to load and preprocess images from a specified directory. It takes a base directory path and a list of file paths as input and returns a NumPy array containing the preprocessed images.The function iterates over each file path, constructs the full file path, reads the image using `cv2.imread` in grayscale mode, resizes the image to a fixed size of 64x64 pixels using `cv2.resize`, and flattens the image into a one-dimensional array. The preprocessed image is then added to the `images` list.Finally, the function returns a NumPy array containing all the preprocessed images.

Second sell consists of further commends:

* mounts the Google Drive to the Colab notebook.
* The `os` library is imported for file and directory operations.
* The `base\_dir` is set to the root directory of the mounted Google Drive.
* and the `folder\_path` is set to the specific directory where the images are stored.
* The `os.listdir` function is used to obtain the list of file names within the specified folder path.
* The first two characters of each file name are extracted as labels for the images.
* File paths are created by joining the folder path with each file name.
* The `load\_images()` function (not shown) is called to load and preprocess the images using the constructed file paths.

Next Cell is as follow:

* The list of labels is converted into a NumPy array using `np.array(labels)`.
* The `LabelEncoder` class from `sklearn.preprocessing` is used to transform the labels into numeric representations.
* The `fit\_transform()` method of `label\_encoder` is applied to the labels array to perform the label encoding.
* The encoded labels are stored in the variable `numeric\_labels`.
* The shape of the `numeric\_labels` array is reshaped to have a single column using `reshape(-1, 1)`.
* The `np.squeeze()` function is utilized to remove any single-dimensional entries from the shape of the array, resulting in a flattened numeric labels array.

the whiten data function centers the input data, computes the covariance matrix, calculates the eigenvalues and eigenvectors, and then performs whitening by scaling the data along the eigenvector directions. This process results in uncorrelated features with unit variance.

The function takes whitened data, which is the preprocessed dataset with normalized features, and labels, which are the corresponding class labels for each sample. The whitened data is assumed to have zero mean and unit variance along each feature.

The scatter matrices play a crucial role in LDA by quantifying the spread of data between different classes (SB) and within each class (SW). SB measures the variations between class means and the mean of the entire dataset, while SW accounts for the variations within each class.

The function takes two input matrices: the between-class scatter matrix (SB) and the within-class scatter matrix (SW). These scatter matrices capture the statistical characteristics of the data distribution with respect to class separability.

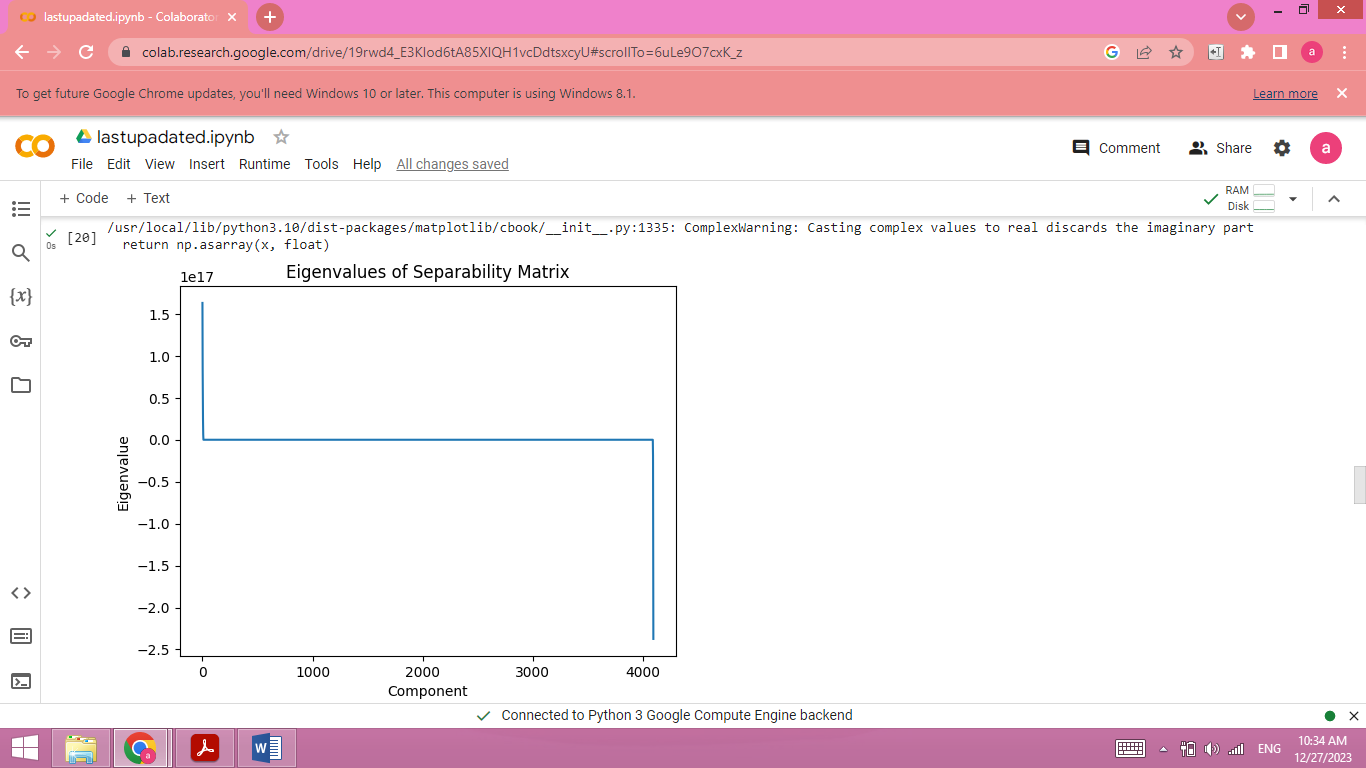
The separability matrix is computed by taking the dot product of the inverse of SW and SB. It represents the transformation that maximizes the separation between different classes while minimizing the variation within each class.

Eigen decomposition is then applied to the separability matrix, yielding eigenvalues and eigenvectors. The eigenvalues indicate the importance of each eigenvector in explaining the separability of the data.

To facilitate further analysis, the eigenvalues are sorted in descending order, and the corresponding eigenvectors are reordered accordingly. This sorting allows for selecting the most discriminative features or projections.

The function returns the sorted eigenvalues and eigenvectors

The function takes the sorted eigenvalues as input and creates a line plot using the Matplotlib library. The x-axis represents the components or eigenvalue indices, while the y-axis represents the corresponding eigenvalues.



The function takes three inputs: the input data, the eigenvectors, and the desired number of components for the projection. The input data is typically a matrix or array containing the samples or observations, while the eigenvectors represent the directions or axes of the subspace onto which the data will be projected.

The projection is performed by taking the dot product of the data with the selected eigenvectors. This operation effectively projects each data point onto the subspace spanned by the eigenvectors, resulting in a lower-dimensional representation of the data

The function takes three inputs: the data to be split, the corresponding labels, and the desired test ratio. The test ratio represents the proportion of the total data that should be allocated to the testing set, while the remaining data becomes the training set.

The number of samples in the data is determined, and the number of test samples is calculated based on the test ratio. This allows for flexibility in adjusting the size of the testing set according to specific requirements.

The function generates an array of shuffled indices using np.random.permutation, ensuring that the samples are randomly ordered. This randomness is crucial to avoid any biases that might arise from the original ordering of the data.

Using the shuffled indices, the data and labels are split into training and testing sets. The training set comprises the samples after the first num\_test\_samples indices, while the testing set consists of the first num\_test\_samples indices.

The function then returns the split data and labels as separate variables: train\_data, train\_labels, test\_data, and test\_labels.

The class priors are calculated by counting the occurrences of each unique label in the training labels and normalizing them by dividing by the total number of training labels. This provides the prior probabilities of each class.

For each unique label, the function computes the class mean and covariance using the corresponding samples from the training data. The class mean is calculated as the average of the samples, while the class covariance is obtained using the np.cov function.

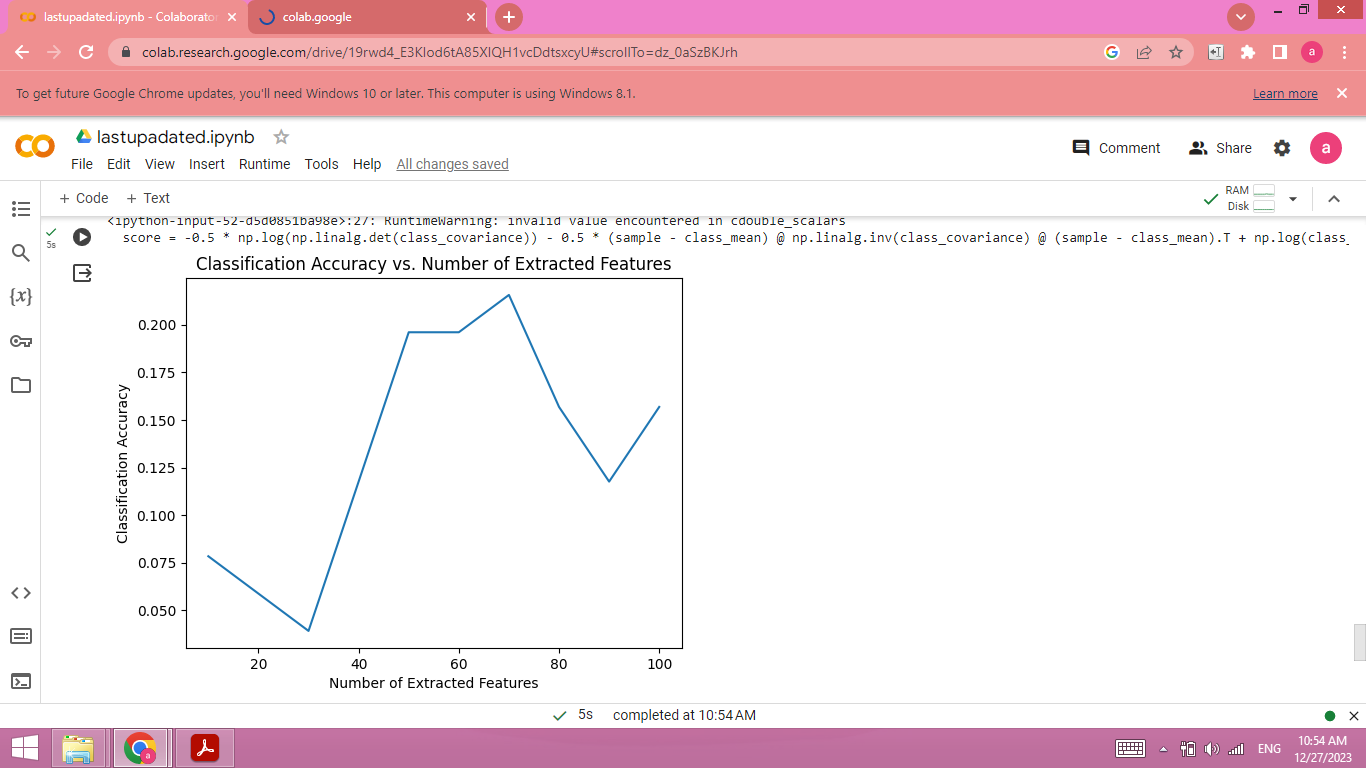
To handle potential issues with covariance matrices, the function checks if a covariance matrix is empty. If empty, a small positive value is added to the diagonal elements to ensure invertibility and stability. Otherwise, the small positive value is added to the diagonal elements to prevent numerical instability.

The classification process involves iterating over each sample in the test data. For each sample, the function computes a score for each class label using the Bayesian classifier formula. The score takes into account the class mean, covariance, class prior, and the sample itself. The class label with the highest score is selected as the predicted label for that sample.

The accuracy of the classifier is calculated by comparing the predicted labels with the true test labels and taking the mean of the comparison result.

The function returns the accuracy, representing the classification accuracy of the Bayesian classifier on the test data.

We aims to evaluate the classification accuracy of the Bayes classifier with varying numbers of extracted features. It performs the following step.



**Conclusion**

Both codes serve as educational examples of how to implement and apply fundamental machine learning algorithms. They highlight the importance of data preprocessing, parameter optimization, and result visualization in the context of regression and classification tasks. These codes can be a starting point for more advanced modeling and analysis of the respective domains they address