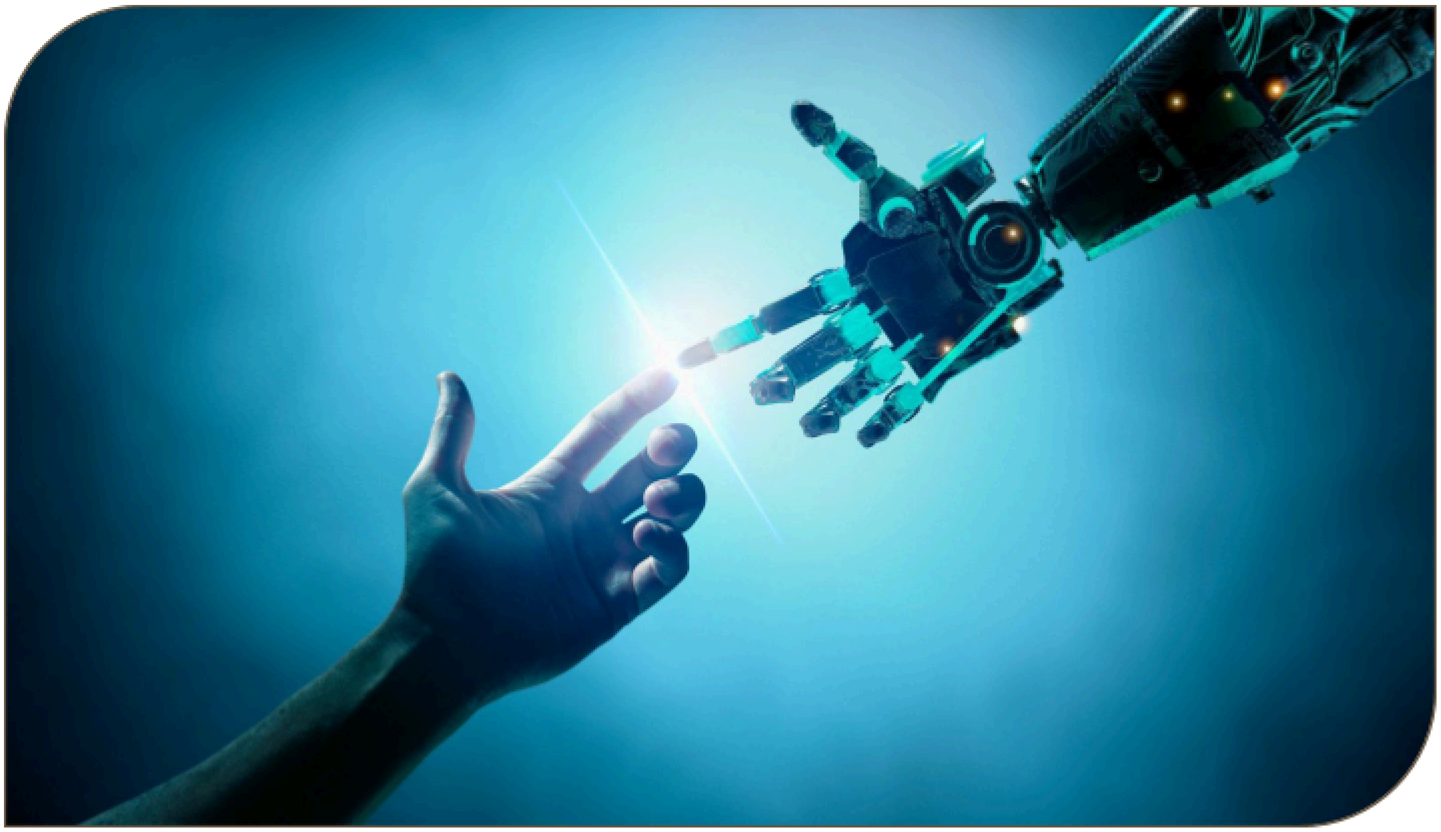


PATTERN RECOGNITION

Face recognition

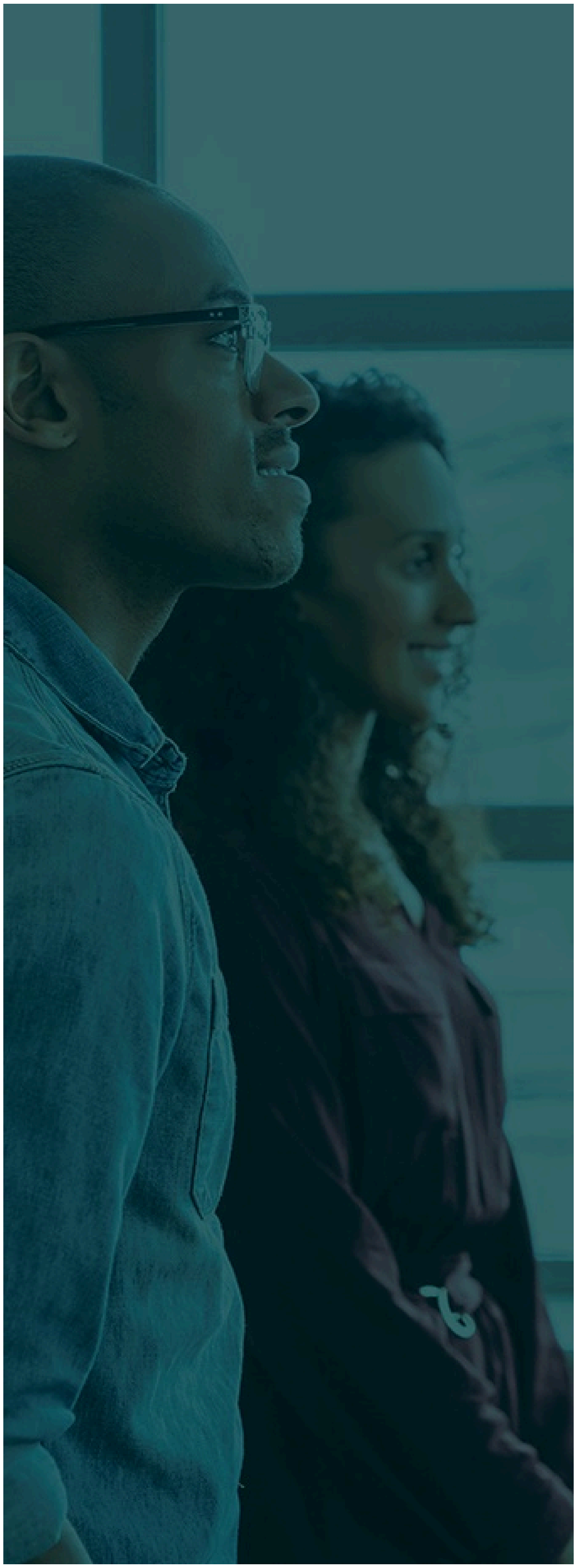


TABLE OF CONTENTS

Problem Statement

Intend to perform face recognition. Face recognition means that for a given image you can tell the subject id. The training set consists of different 40 subject ids such that each subject id has 10 images to train from.

1. Data Preparation: Download, convert, and stack images into a data matrix with corresponding labels.

2. Dataset Splitting: Divide data into training and test sets, ensuring equal representation of subjects.

3. PCA Classification: Reduce dimensionality using PCA, project data, and classify using nearest neighbor.

4. LDA Classification: Implement LDA for multiclass, project data, and classify using nearest neighbor.

5. Classifier Tuning: Evaluate K-NN classifier with varying neighbors for both PCA and LDA.

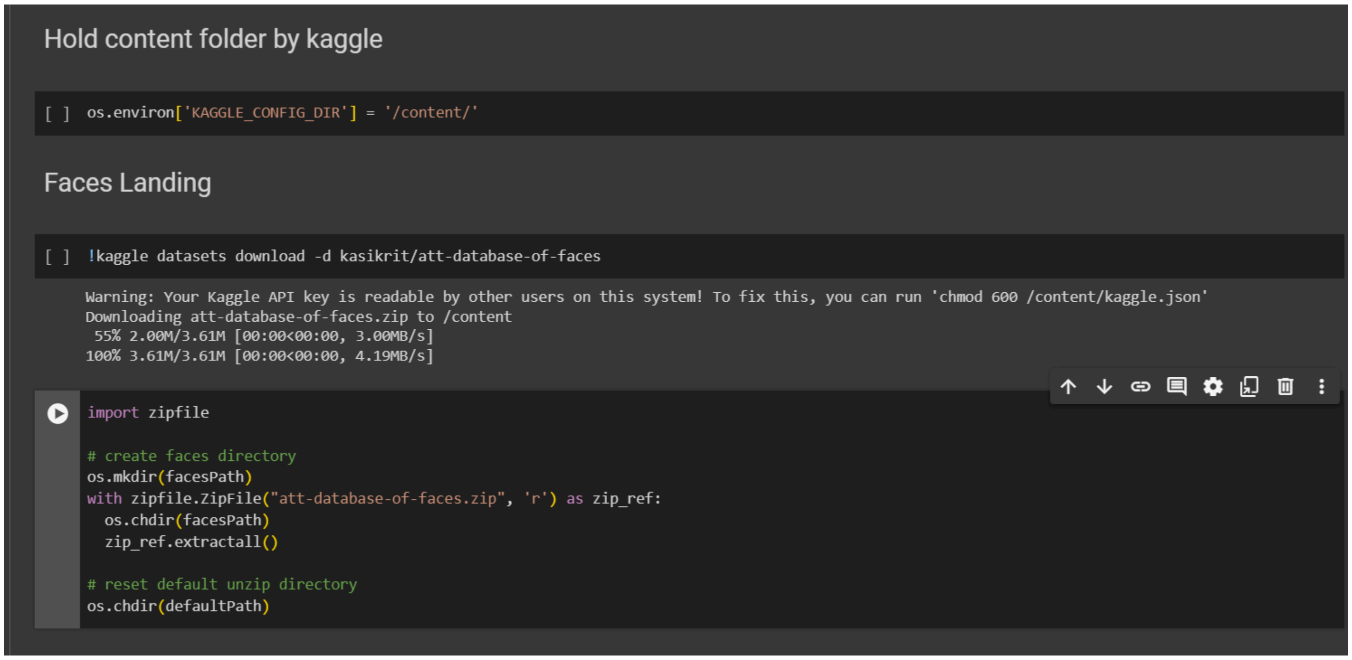
6. Comparison with Non-Face Images: Assess performance against non-face images, considering varying dataset sizes.

7. Create a comparison between Basic PCA and one of its variations,Kernel PCA and Basic LDA and one of its variations.

1. DATASET CONFIGURATION

We have downloaded the "AT&T Database of Faces" dataset from Kaggle.

- **Downloading the dataset and understanding it's format:**



Hold content folder by kaggle

```
[ ] os.environ['KAGGLE_CONFIG_DIR'] = '/content/'
```

Faces Landing

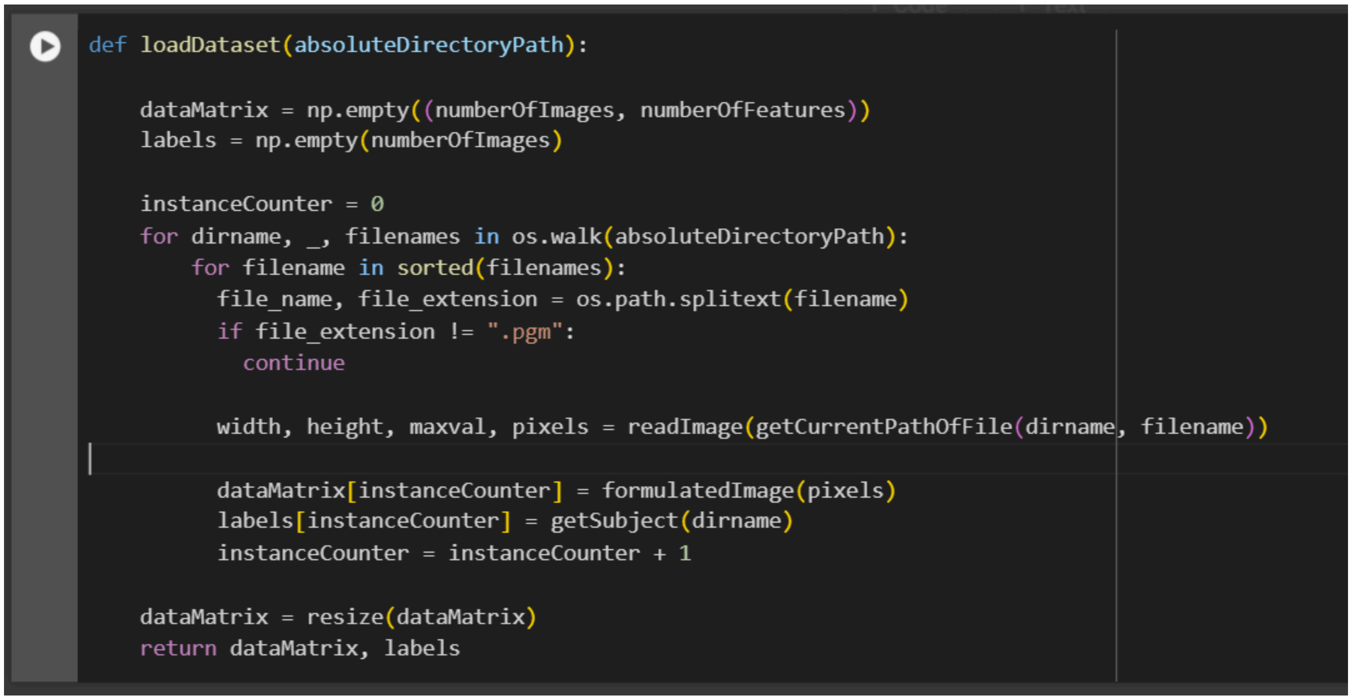
```
[ ] !kaggle datasets download -d kasikrit/att-database-of-faces
```

```
Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /content/kaggle.json'
Downloading att-database-of-faces.zip to /content
 55% 2.00M/3.61M [00:00<00:00, 3.00MB/s]
100% 3.61M/3.61M [00:00<00:00, 4.19MB/s]
```

```
import zipfile

# create faces directory
os.mkdir(facesPath)
with zipfile.ZipFile("att-database-of-faces.zip", 'r') as zip_ref:
  os.chdir(facesPath)
  zip_ref.extractall()

# reset default unzip directory
os.chdir(defaultPath)
```

  - We've used the Kaggle CLI to download the dataset specifying it's slug (kasikrit/att-database-of-faces).
  - Then we made a folder for face images and extracted the zip file of the dataset in it.
  - We have 40 labels and for each label we have 10 images, so by total we have 400 images and each image is 92x112 pixels.

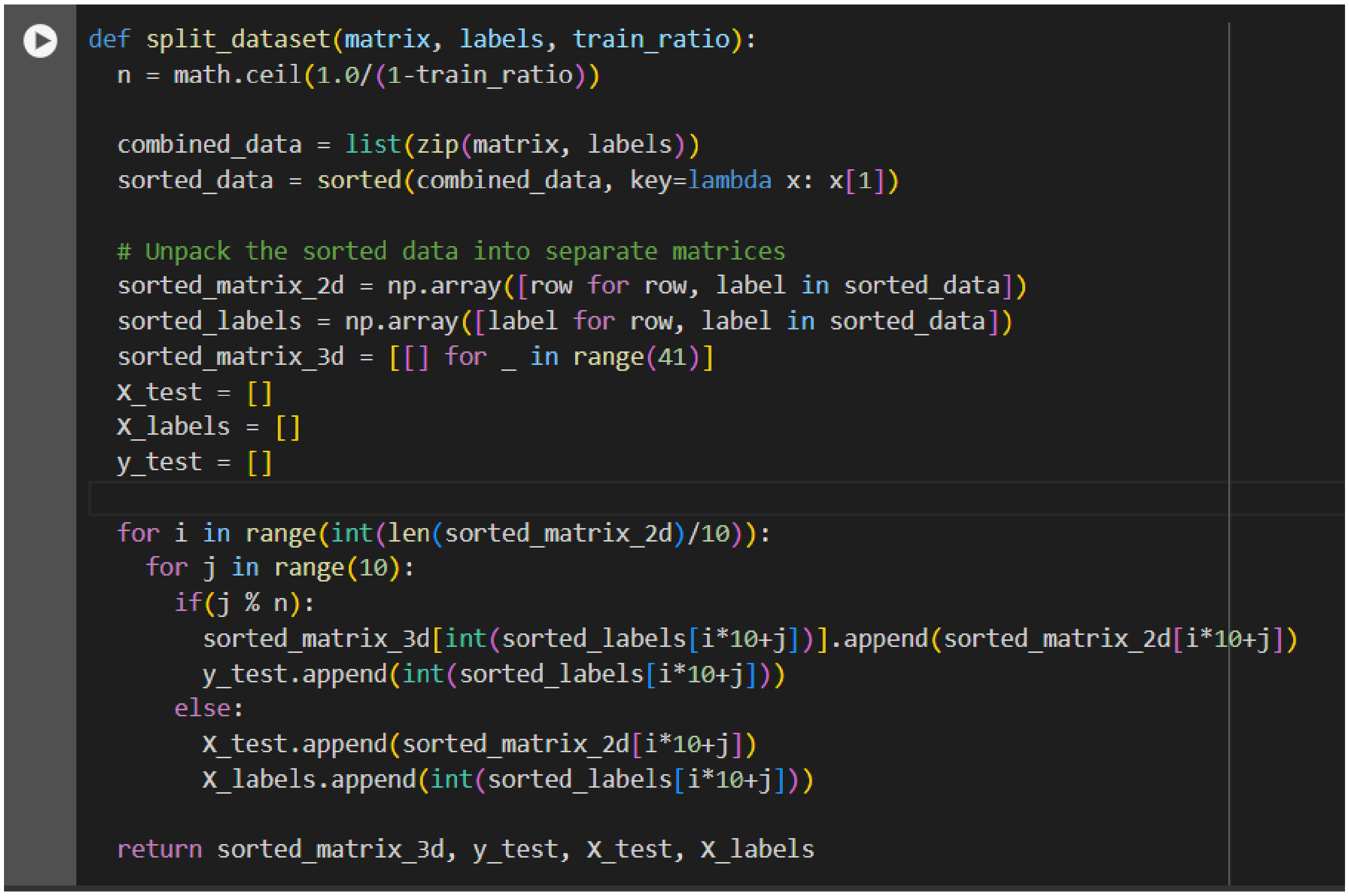
- **Generate the Data Matrix and the Label vector:**



```python
def loadDataset(absoluteDirectoryPath):

    dataMatrix = np.empty((numberOfImages, numberOfFeatures))
    labels = np.empty(numberOfImages)

    instanceCounter = 0
    for dirname, _, filenames in os.walk(absoluteDirectoryPath):
        for filename in sorted(filenames):
          file_name, file_extension = os.path.splitext(filename)
          if file_extension != ".pgm":
            continue

          width, height, maxval, pixels = readImage(getCurrentPathOfFile(dirname, filename))

          dataMatrix[instanceCounter] = formulatedImage(pixels)
          labels[instanceCounter] = getSubject(dirname)
          instanceCounter = instanceCounter + 1

    dataMatrix = resize(dataMatrix)
    return dataMatrix, labels
```

  - We've declared a data matrix and labels np arrays, then we iterated on each pgm file and read the image and added it into the data matrix and its label.
  - Data matrix is an np array that hold the images from the dataset of size number of images (400) * number of features (10304).
  - Labels is an np array that holds the labels of each image of size 400, where label[i] range from 1 to 40.
  - We've resized each image by taking every 2x2 square pixels and replacing them with one pixel equal to their average. This trick helps us to reduce the number of features from 10304 into 2576 without losing the important information about the image, also it reduces the time of execution of both algorithms.

- **Generate the Data Matrix and the Label vector:**

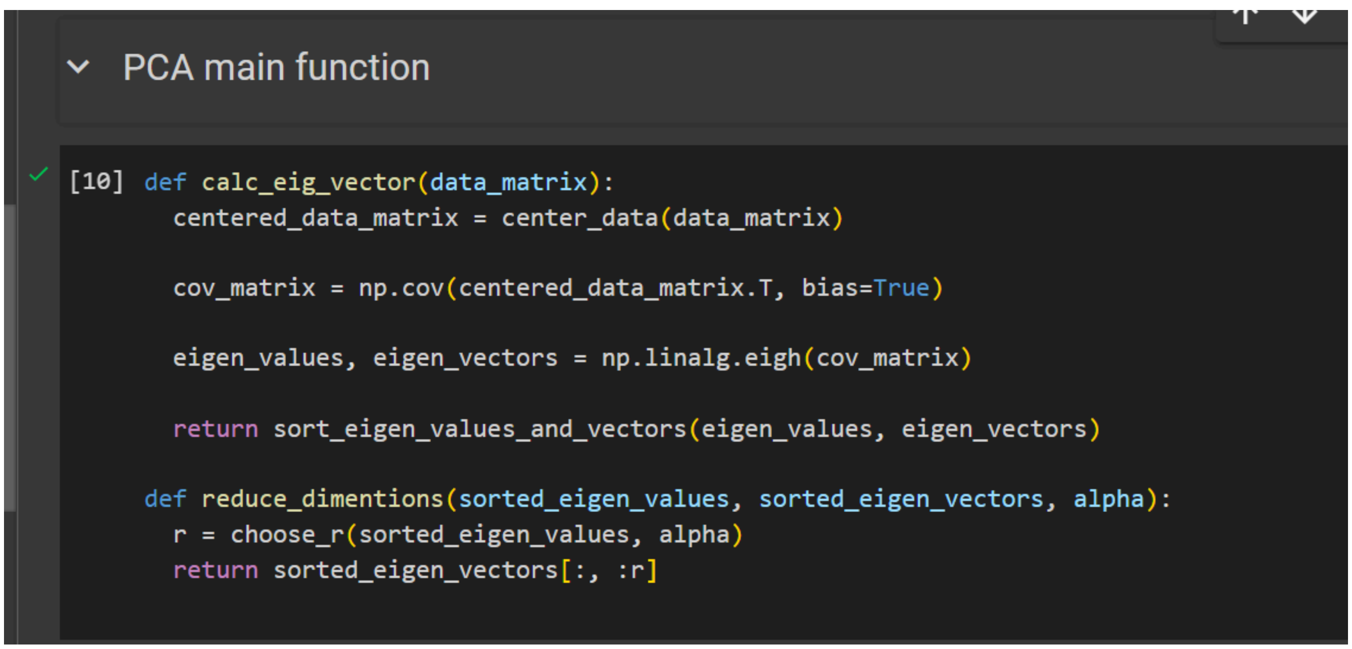


```
def split_dataset(matrix, labels, train_ratio):
  n = math.ceil(1.0/(1-train_ratio))

  combined_data = list(zip(matrix, labels))
  sorted_data = sorted(combined_data, key=lambda x: x[1])

  # Unpack the sorted data into separate matrices
  sorted_matrix_2d = np.array([row for row, label in sorted_data])
  sorted_labels = np.array([label for row, label in sorted_data])
  sorted_matrix_3d = [[] for _ in range(41)]
  X_test = []
  X_labels = []
  y_test = []

  for i in range(int(len(sorted_matrix_2d)/10)):
    for j in range(10):
      if(j % n):
        sorted_matrix_3d[int(sorted_labels[i*10+j])].append(sorted_matrix_2d[i*10+j])
        y_test.append(int(sorted_labels[i*10+j]))
      else:
        X_test.append(sorted_matrix_2d[i*10+j])
        X_labels.append(int(sorted_labels[i*10+j]))

  return sorted_matrix_3d, y_test, X_test, X_labels
```

- We’ve created a function that takes the data matrix, the corresponding labels, and a train ration.
- Given the ratio the function will split the data matrix into training set and test set with the given ratio, for ratio 0.5 the data will splitted into even rows for training and odd for testing, accordingly the labels will be splitted too.
- The function is generic, where it takes a ratio so as to be used in the bonus part too with train ratio = 0.7

2. Classification using PCA

- **Code:**



PCA main function

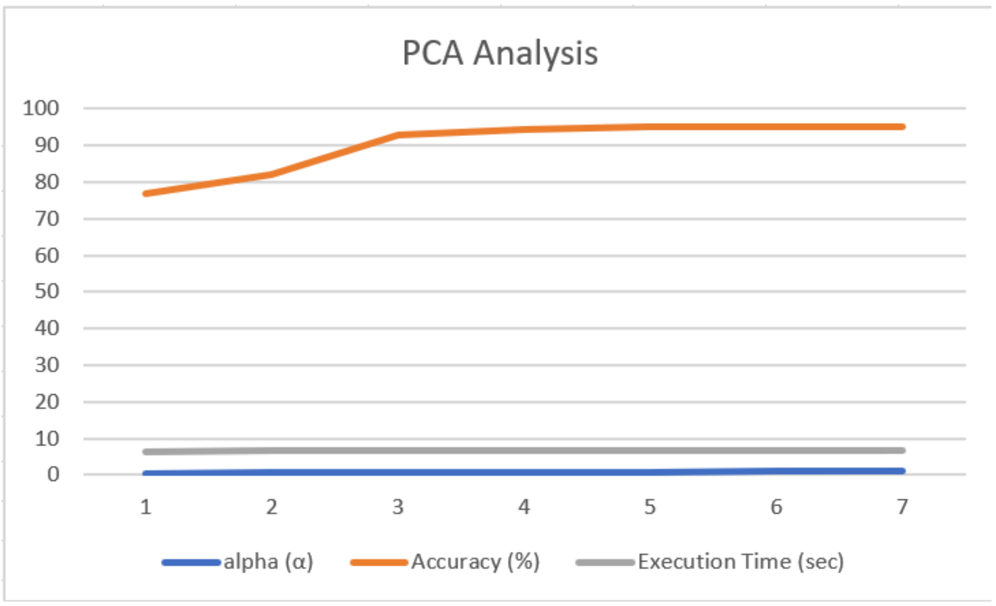
```
[10] def calc_eig_vector(data_matrix):
       centered_data_matrix = center_data(data_matrix)

       cov_matrix = np.cov(centered_data_matrix.T, bias=True)

       eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)

       return sort_eigen_values_and_vectors(eigen_values, eigen_vectors)

     def reduce_dimentions(sorted_eigen_values, sorted_eigen_vectors, alpha):
       r = choose_r(sorted_eigen_values, alpha)
       return sorted_eigen_vectors[:, :r]
```

- First, we center the data using the mean vector.
- compute the covariance matrix by using the builtin function in python and making bias = True, since python uses a non-biased version to compute this matrix.
- Calculate the eigenvalues and eigenvectors by taking advantage that the covariance matrix is symmetric so we use eigh function that are faster than eig.
- Then we sort the eigenvectors according to their eigenvalues and pick the first r components that meet the condition of alpha.

- **Analysis of PCA:**

  - By splitting the data matrix into training and test sets by ratio 0.5.
  - By using the first nearest neighbor as our classifier (k = 1).

alpha (α)	Accuracy (%)	Execution Time (sec)
0.5	77	6.695
0.6	82	6.6978
0.7	93	6.7181
0.8	94.5	6.7245
0.85	95	6.74151
0.9	95	6.7884
0.95	95	6.79033



- **Comments:**

    - The accuracy of PCA is dependent on alpha values where as the value of alpha increases, the accuracy generally increases and the reason for this is by retaining more principal components (higher alpha values), more variance in the data is preserved, resulting in better classification performance.

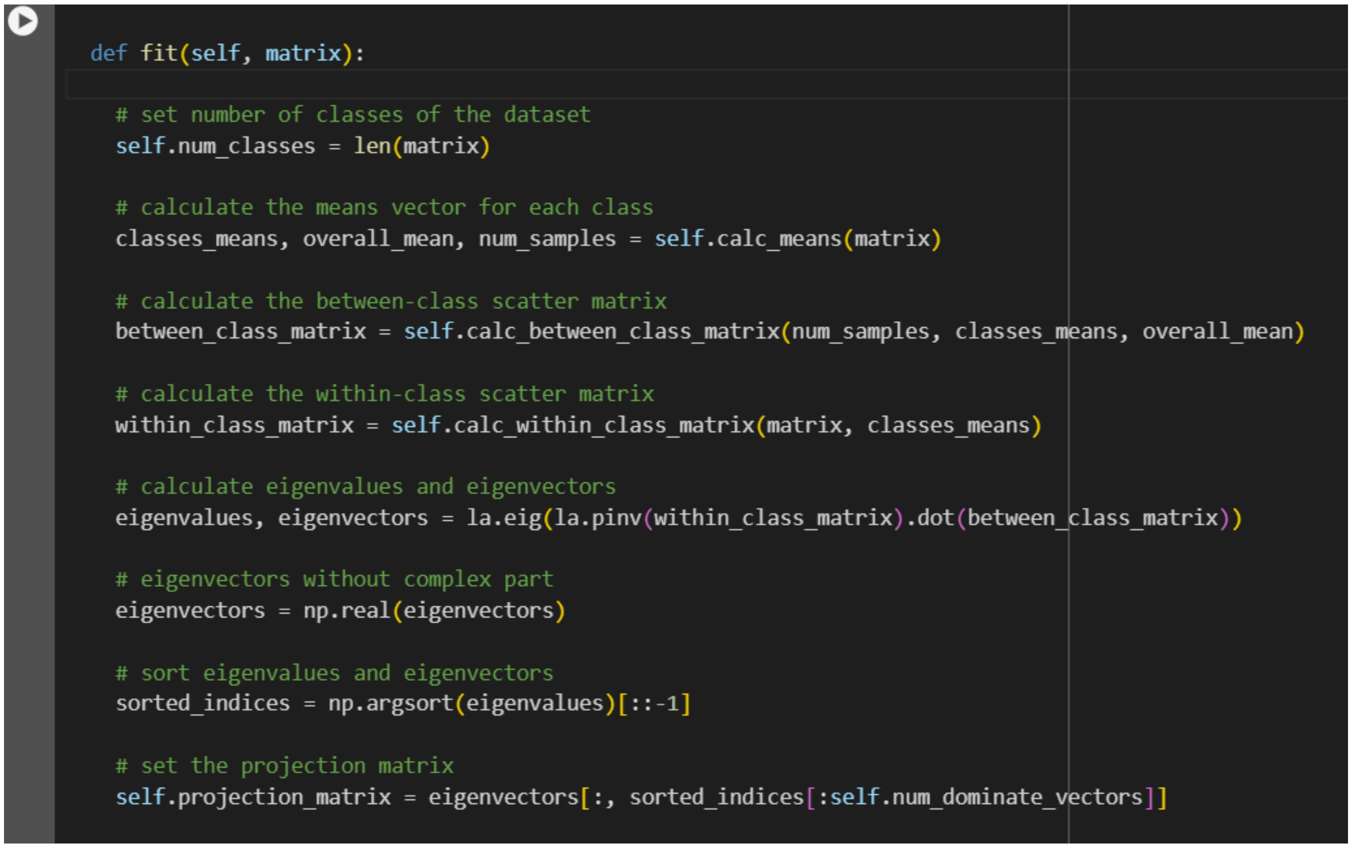
    - The execution time generally increases as the value of alpha increases. This is expected because retaining more principal components requires more computational resources for eigenvector calculation, dimensionality reduction, and classification.

    - The accuracy improvement starts to plateau around alpha values of 0.8. Beyond this point, Beyond this point, there's minimal increase in accuracy so the accuracy saturates after alpha=0.85 because the dimension remained have no dominant effect and this may be because the dataset have might not contain significant information beyond a certain point, and retaining more principal components does not lead to further accuracy improvement.

- **Relation between alpha and accuracy:**

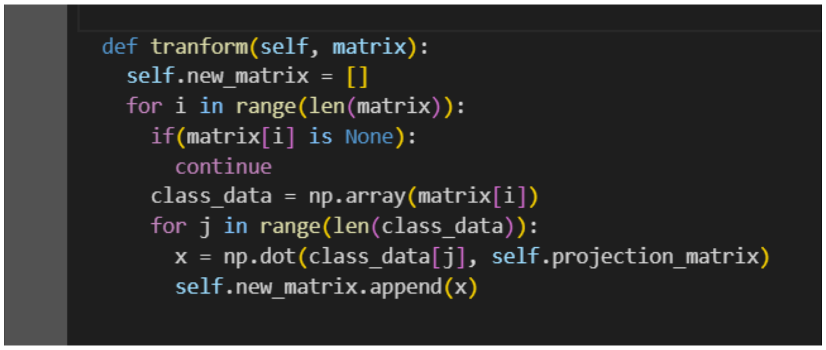
    - As declared above as the alpha increases the accuracy also increases till a certain alpha where the accuracy remains unchanged and that's because the remaining eigenvectors have no dominant effect.

3. Classification using LDA



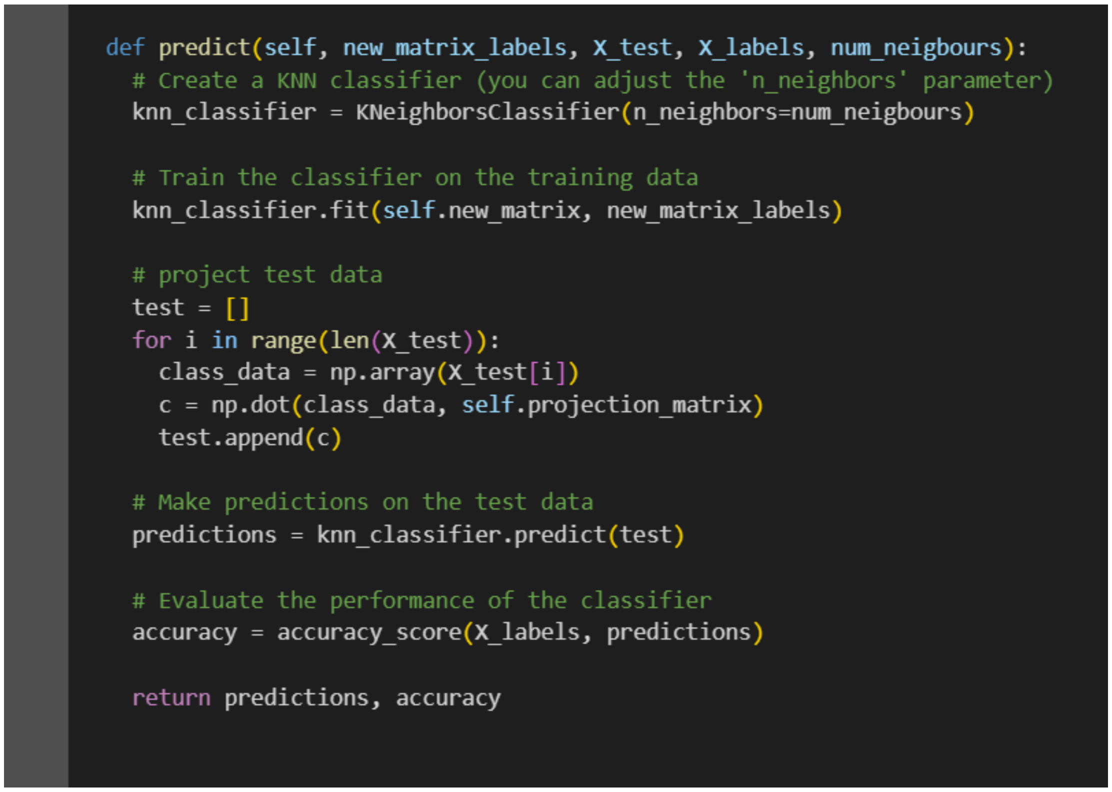
```
def fit(self, matrix):

  # set number of classes of the dataset
  self.num_classes = len(matrix)

  # calculate the means vector for each class
  classes_means, overall_mean, num_samples = self.calc_means(matrix)

  # calculate the between-class scatter matrix
  between_class_matrix = self.calc_between_class_matrix(num_samples, classes_means, overall_mean)

  # calculate the within-class scatter matrix
  within_class_matrix = self.calc_within_class_matrix(matrix, classes_means)

  # calculate eigenvalues and eigenvectors
  eigenvalues, eigenvectors = la.eig(la.pinv(within_class_matrix).dot(between_class_matrix))

  # eigenvectors without complex part
  eigenvectors = np.real(eigenvectors)

  # sort eigenvalues and eigenvectors
  sorted_indices = np.argsort(eigenvalues)[::-1]

  # set the projection matrix
  self.projection_matrix = eigenvectors[:, sorted_indices[:self.num_dominate_vectors]]
```

- **Code:**
    - Get the number of classes from the matrix size.
    - Calculate the mean vector for each class, the overall mean, and the number of samples vector.
    - Calculate the between class matrix, within class matrix.
    - Calculate the eigenvalues and its corresponding eigenvectors of the matrix resulting from the pinv(within class matrix) multiplied by the between class matrix.
    - The eigenvectors are calculated using the np.linalg.eigh function not the eig function since the resulting matrix may not be symmetric.
    - Then the eigenvectors are sorted according to their corresponding eigenvalues.
    - The projection matrix now equals the first 39 sorted eigenvectors.



```
def tranform(self, matrix):
  self.new_matrix = []
  for i in range(len(matrix)):
    if(matrix[i] is None):
      continue
    class_data = np.array(matrix[i])
    for j in range(len(class_data)):
      x = np.dot(class_data[j], self.projection_matrix)
      self.new_matrix.append(x)
```

- Transform function is used to project the training data into the new reduced dimensions by multiplying the projection matrix with the image array.



```
def predict(self, new_matrix_labels, X_test, X_labels, num_neigbours):
  # Create a KNN classifier (you can adjust the 'n_neighbors' parameter)
  knn_classifier = KNeighborsClassifier(n_neighbors=num_neigbours)

  # Train the classifier on the training data
  knn_classifier.fit(self.new_matrix, new_matrix_labels)

  # project test data
  test = []
  for i in range(len(X_test)):
    class_data = np.array(X_test[i])
    c = np.dot(class_data, self.projection_matrix)
    test.append(c)

  # Make predictions on the test data
  predictions = knn_classifier.predict(test)

  # Evaluate the performance of the classifier
  accuracy = accuracy_score(X_labels, predictions)

  return predictions, accuracy
```

- The predict function is used to project the testing data and classify each image by using the first nearest neighbor using the built in KNeigborsClassifier
- Then the accuracy is calculated on the given labels and the predictions returned by the knn classifier.

- **Analysis of LDA:**

  - By splitting the data matrix into training and test sets by ration 0.5.
  - By using the first nearest neighbor as our classifier (k = 1).

Accuracy (%)	**Execution Time (sec)**
97	**51.22**

- **Comments:**

  - The achieved accuracy of 97% is quite high, indicating that the Linear Discriminant Analysis (LDA) model trained on the dataset is effective in classifying the data. This suggests that the features extracted by LDA are discriminative and capture important information for classification.

  - The execution time is high when the training process is computationally high, this is caused by two functions which are pinv to calculate the inverse and eig to calculate the eigenvectors.

- **PCA vs LCA:**

    - We’ve maintained the comparison at the same number of components = 39.

	Accuracy (%)	Execution Time (sec)
PCA	95.5	6.75
LDA	97	51.22

- **Comments:**

    - LDA achieves a slightly higher accuracy of 97% compared to PCA, which achieves 95.5% accuracy. This suggests that LDA might be better at capturing the discriminative information necessary for classification compared to PCA in this specific context, and that’s because PCA focuses on maximizing variance while reducing dimensionality, while LDA emphasizes class separability and decrease variance in each class, so it would be better in classification and the accuracy emphasizes this.

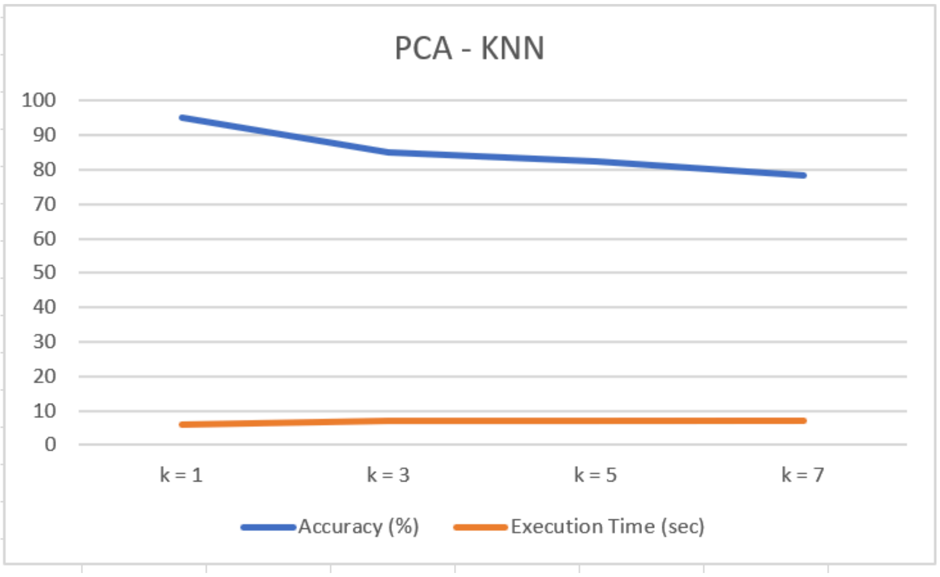
    - PCA demonstrates significantly faster execution times, with only 6.75 seconds compared to LDA's 51.22 seconds. This highlights the computational efficiency advantage of PCA over LDA, i think this is mainly because of using the eigh function which is faster than eig function as it works on a symmetric matrix.

4. Classifier Tuning

- **Tie break strategy:**
    - We've used the default tie breaking strategy provided by the KNeighborsClassifier which breaks the tie by considering the class with the smallest index (i.e., the class that appears first in the sorted list of class labels).

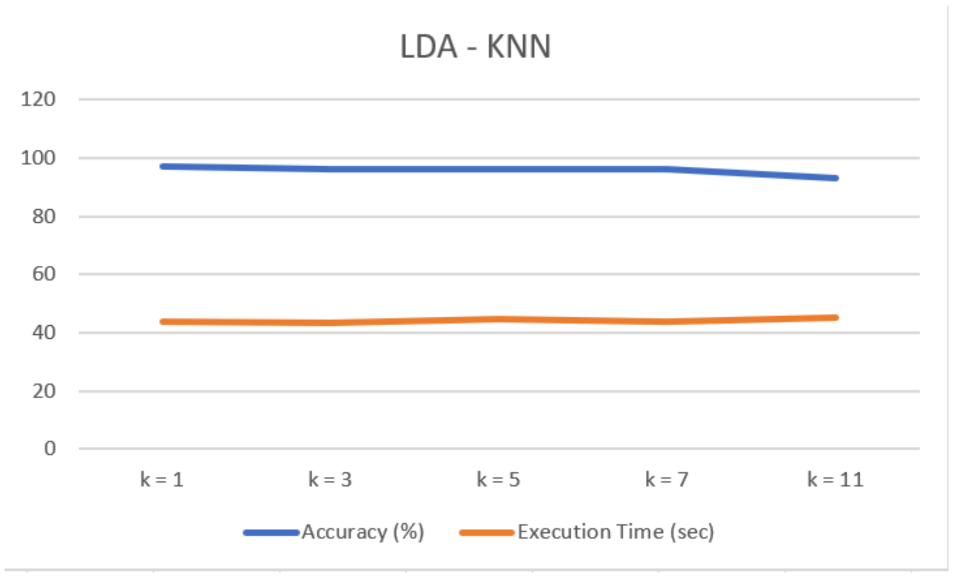
- **PCA:**
    - These runs are based on alpha = 0.9, and train ratio = 0.5.

number of neighbors	Accuracy (%)	Execution Time (sec)
k = 1	95	5.7708
k = 3	85	6.9559
k = 5	82.5	7.0805
k = 7	78.5	7.1367



- **LDA**:
    - These runs are based on train ratio = 0.5.

number of neighbors	Accuracy (%)	Execution Time (sec)
k = 1	97	43.9459
k = 3	96	43.5205
k = 5	96	44.7723
k = 7	96	43.8533
k = 11	93	44.99997



- **Comments:**

    - By increasing K, for both algorithms the accuracy decreases and also the execution time increases.

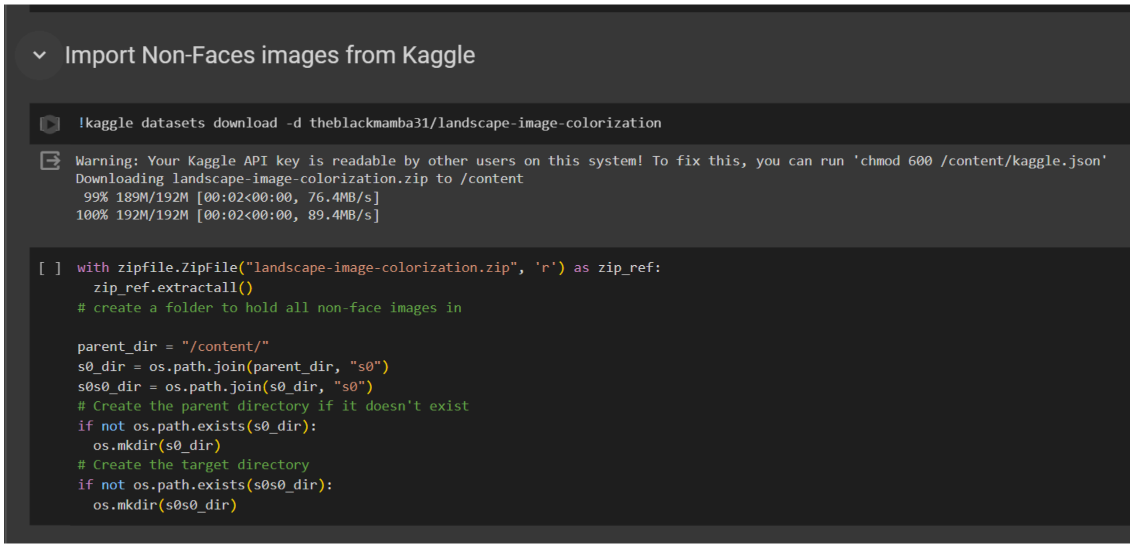
    - LDA generally outperforms PCA in terms of accuracy across different values of k since PCA achieves slightly lower accuracy compared to LDA for all values of k. PCA's accuracy ranges from 78.5% to 95%, while LDA's accuracy ranges from 93% to 97%.

    - The reason for above point is that the LDA objective is to group each cluster images (minimize variance between them) so by default in LDA each image is surrounded by high percentage with images from same class, while in PCA the objective is just to increase the variance and this may group images from different clusters together.

5. Face vs Non-Face images

-
- **Download Dataset:**

- We've downloaded a landscape image dataset from kaggle.



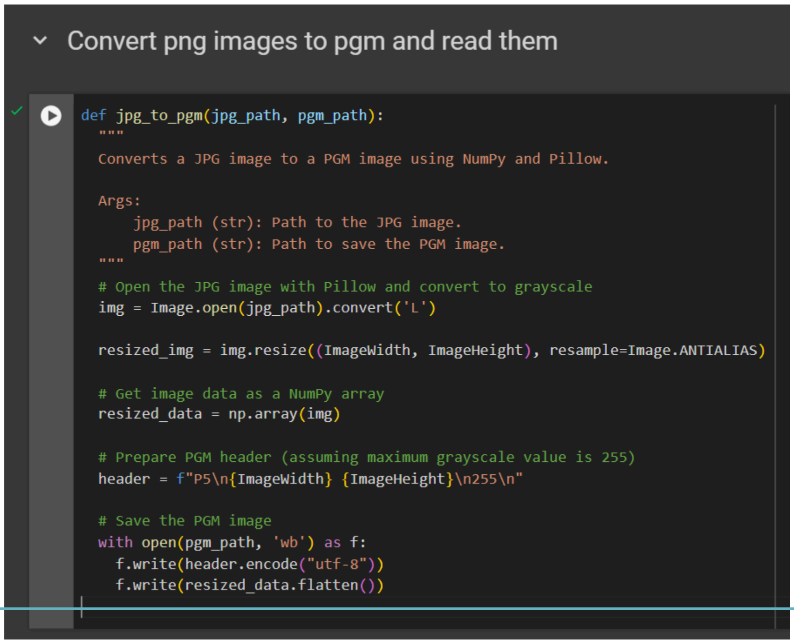
Import Non-Faces images from Kaggle

```
!kaggle datasets download -d theblackmamba31/landscape-image-colorization
```

```
Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /content/kaggle.json'
Downloading landscape-image-colorization.zip to /content
 99% 189M/192M [00:02<00:00, 76.4MB/s]
100% 192M/192M [00:02<00:00, 89.4MB/s]
```

```python
with zipfile.ZipFile("landscape-image-colorization.zip", 'r') as zip_ref:
  zip_ref.extractall()
# create a folder to hold all non-face images in

parent_dir = "/content/"
s0_dir = os.path.join(parent_dir, "s0")
s0s0_dir = os.path.join(s0_dir, "s0")
# Create the parent directory if it doesn't exist
if not os.path.exists(s0_dir):
  os.mkdir(s0_dir)
# Create the target directory
if not os.path.exists(s0s0_dir):
  os.mkdir(s0s0_dir)
```

- Here we convert the image from jpg into pgm and resize it into 92x112.

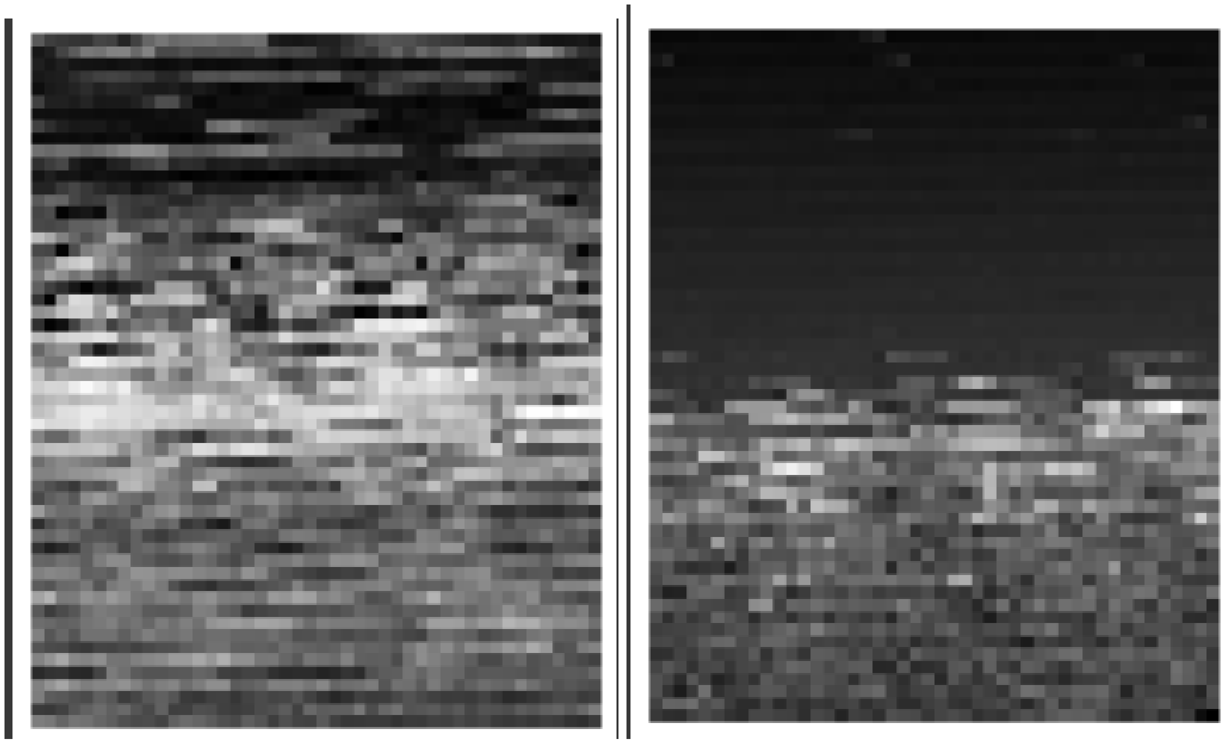


Convert png images to pgm and read them

```python
def jpg_to_pgm(jpg_path, pgm_path):
  """
  Converts a JPG image to a PGM image using NumPy and Pillow.

  Args:
      jpg_path (str): Path to the JPG image.
      pgm_path (str): Path to save the PGM image.
  """
  # Open the JPG image with Pillow and convert to grayscale
  img = Image.open(jpg_path).convert('L')

  resized_img = img.resize((ImageWidth, ImageHeight), resample=Image.ANTIALIAS)

  # Get image data as a NumPy array
  resized_data = np.array(img)

  # Prepare PGM header (assuming maximum grayscale value is 255)
  header = f"P5\n{ImageWidth} {ImageHeight}\n255\n"

  # Save the PGM image
  with open(pgm_path, 'wb') as f:
    f.write(header.encode("utf-8"))
    f.write(resized_data.flatten())
```

- **Failure cases:**

  - These two images fail during classification while running PCA.



- Note that the two images share the same characteristics and that's a reason why both images are misclassified due to the rise of Misclassification Patterns.

- **Dominant eigenvectors:**

  - Number of dominant eigenvectors in LDA equals to the number of classes minus one (# of classes - 1).
  - In our case we have two classes which are face or non-face so the number of dominant eigenvectors equals one (= 1).

- **Analysis of Non-Faces vs Faces:**

- The number of face images for training is fixed to 280 images.
- These runs for PCA.

Number of non-face images	**Accuracy (%)**
84	**99.16**
140	**100**
196	**100**
280	**100**

- These runs for LDA.

Number of non-face images	**Accuracy (%)**
84	**87**
140	**82**
196	**84**
280	**83**

- **For large numbers:**

- **The accuracy for LDA have decreased while increasing the number of non-face images, while in PCA the accuracy increases.**

6. Bonus

a. Using Different train-test-ratio such that training ratio = 70% and testing ratio = 30%.

It's totally simplified using a function call to with the given ratio:

1. PCA:

alpha	0.5	0.6	0.7	0.8	0.85	0.9
percentage at ratio = 0.5	0.77	0.82	0.93	0.945	0.95	0.95
percentage at ratio = 0.7	0.8416667	0.94166667	0.966667	0.975	0.975	0.975

- From the previous table, we deduce obviously that when the test ratio increased from 0.5 to 0.7 the accuracy of the test data increased which intuitively appears correct since Increasing the training dataset size generally allows the model to learn better representations of the data and capture more of its underlying structure. With more data, the model has a better chance of learning patterns that are generalizable to unseen data.

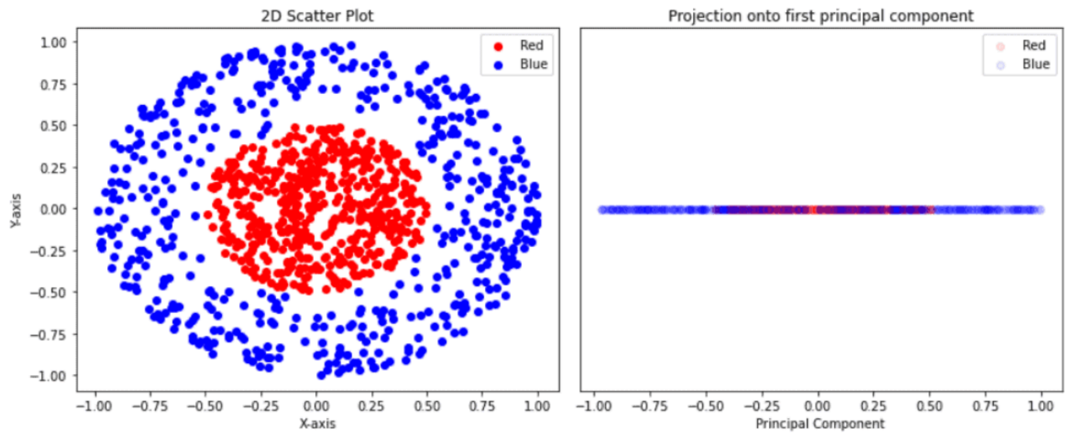
2. LDA:
   - with ratio = 0.5 gets accuracy = 97% but with ratio = 0.7 gets accuracy = 0.99.
   - Same as PCA, With a higher train-test ratio (e.g., 0.7), more data is used for training the model. This can lead to better generalization as the model has more examples to learn from, potentially capturing the underlying patterns more effectively.

b. Comparisons with variation of PCA and LDA:

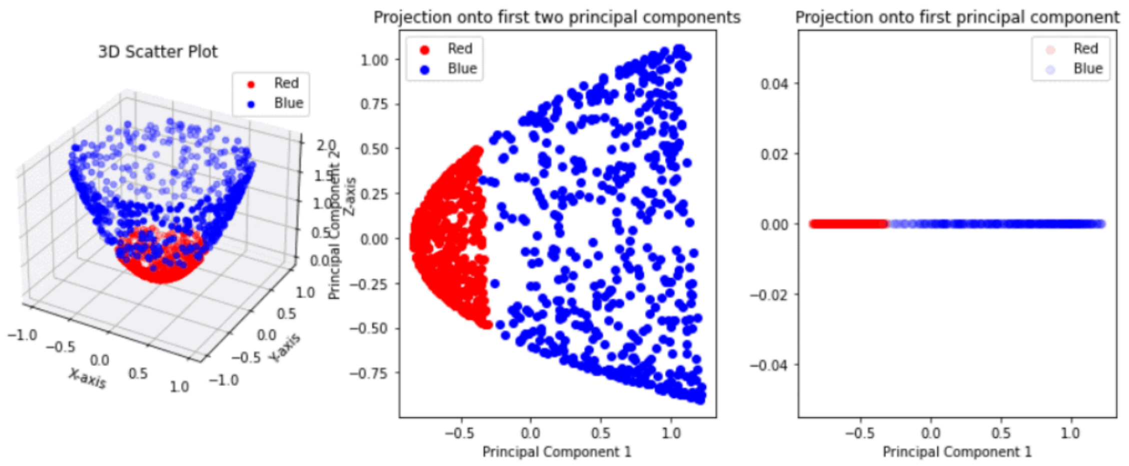
   i. Comparison basic PCA with its chosen variation kernel PCA.

- What is Kernel PCA? It's one of most popular PCA variations which has an advantage over PCA in case of non-linearity of the data. Basic PCA can’t deal with non-linearly separated datasets since its whole objective is to produce the principal components that have the largest variance of data and have most of the data information.
- What is its advantage? it tries to convert the given dataset into higher dimension space that enables it to easily separate the data when it applies the dimensionality reduction.

To gain a better intuition of this process, lets examine the following example.



The previous image shows how basic pca deals with non-linear simple example data.



Kernel-PCA advantage over basic PCA.

alpha (α)	Basic PCA Accuracy (%)	Kernel PCA Accuracy (%)
0.5	77	85
0.6	82	83
0.7	93	94
0.8	94.5	94
0.85	95	95
0.9	95	95
0.95	95	95

we observe that Kernel PCA accuracies are slightly higher or comparable to basic PCA accuracies. This suggests that in the given dataset, Kernel PCA might be more effective in capturing the underlying structure or patterns, especially when dealing with non-linear relationships among variables.

ii- LDA variation using Least Squares solver (lsqr)

- The 'lsqr' solver in LDA utilizes a least-squares approach to achieve this objective. It essentially solves a system of linear equations that optimizes a specific criterion related to class separation.The 'lsqr' solver doesn't directly calculate the determinants. Instead, it leverages the properties of least-squares to find the projections that maximize this ratio. This involves solving a system of linear equations derived from the scatter matrices. The solution to this system provides the optimal direction vectors for class separation.

- Benefits of lsqr:
    - Efficient for Classification: The 'lsqr' solver can be efficient for classification tasks, especially compared to methods that require full eigenvalue decomposition.
    - Supports Shrinkage: Unlike 'svd', 'lsqr' allows you to set a shrinkage parameter to regularize the model and prevent overfitting.
- A model fit with LDA can potentially overfit the training data, especially when dealing with high-dimensional data or a small number of samples. This can lead to poor performance on unseen data. So a shrinkage = 0.1 is used to introduce a penalty term that discourages the model from having overly large coefficients (weights) in its decision function.

The variation model gives accuracy = 96% with split ratio 0.5

The variation model gives accuracy = 99.17% with split ratio 0.7