PA3: Perceptron and PCA

Colab Link: PA3_B22CS090.ipynb

Question 1:

The following are the steps involved:

- The size of the synthetic dataset has been selected to be 10000.
- 4 feature columns are generated using **generateData()** function of class **MakeTxt**, each of size = 10000, from a uniform distribution with minimum and maximum limits as -100 and 100 respectively. Each generation has been made with a random seed for reproducibility of data.
- Random weights and biases are generated from uniform distribution of minimum and maximum limits as -10 and 10 respectively.
- These parameters are used to set the **'actual'** labels of the synthetic dataset by using dot product. A threshold is defined as, class label = 1 if dot product >= 0, else 0.
- Dataset is shuffled and then the first 70% of the data is selected as the training dataset, which is stored in **B22CS090_train.txt**, and the rest in **B22CS090_test.txt** (which do not contain the 'actual' class labels).
- The whole dataset is stored in **B22CS090_data.txt**.

B22CS090_train.py

- Run this python script as '**%run B22CS090_train.py train.txt**' in jupyter notebook / google colab.
- train.txt is read as a command line argument and is used to make the training txt file.
- Reads a certain percentage of training data which is to be used for training. (Eg: 20% of synthetic data = 28.571425% of training data from B22CS090_train.txt)
- Method **standardise()** normalises the read data for training.
- For training, initially, weights and biases are chosen randomly from standard normal distribution.
- **forward()** method in class **Perceptron** calculates the predicted class label for a singular sample and returns it.

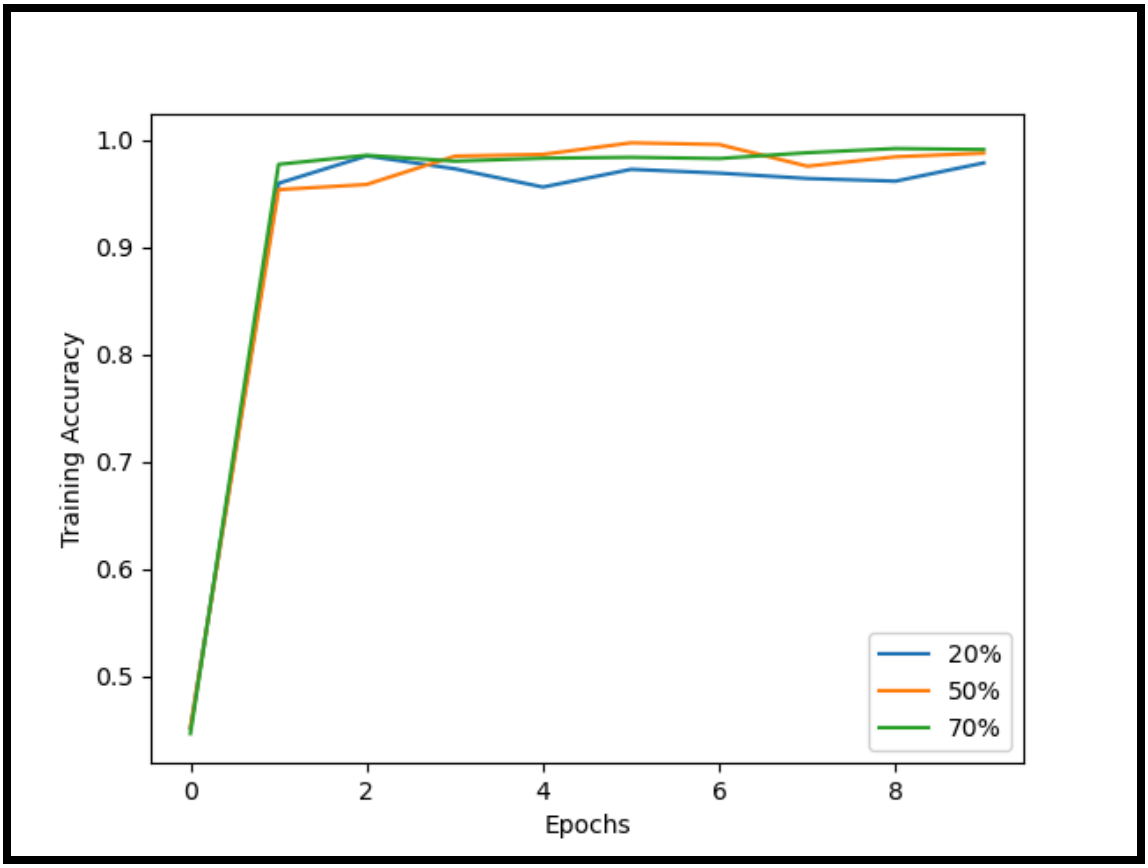
- **backward()** method calculates the error encountered by subtracting predicted label from actual label.
- This error updates the weights and biases in the **train()** method. Update happens after each sample is read, and not batchwise, since that's how the Perceptron algorithm works.
- After given number of epochs, the model weights and biases are saved in **B22CS090_stats.txt** for future reference by B22CS090_test.py. B22CS090_stats.txt will contain training accuracies, weights and biases, means and standard deviations of feature columns of the selected training dataset, and test accuracies of all the 3 cases.
- **Important:** Means and standard deviations of the feature columns for all the 3 cases are being stored since these values will be used to normalise the test dataset later.
- Training was done for 10 epochs, and then training accuracies are calculated using **evaluate()** function

Visualisation

Distribution of Training Data:


Distribution for 50% Training Data
2478
2522
2500
2000
1500
1000
500
0
Count
0
1
Class Labels


Distribution for 70% Training Data
3493
3507
3500
3000
2500
2000
1500
1000
500
0
Count
0
1
Class Labels



Training accuracy vs Epochs graph

Observation: Training accuracy for all the 3 cases, peaked after 2 epochs, and accuracy is very high. The reason for this is that the synthetic dataset is made to be linearly separable, and the updates are done after each sample is read.

We also encounter fluctuations in the curve, which happens because the data cannot be exactly linearly separable as we are using a fraction of the synthetic data. It does not have all of the information. As a result, after almost reaching the peak, the decision boundary oscillates between a few number of points with each update, and results in few correct classifications on one side while misclassifications on the other, causing the rise and fall of the accuracy vs epochs curve.

After 10 epochs:
Training accuracy:
96.95% - 20% of synthetic data

98.76% - 50% of synthetic data
99.09% - 70% of synthetic data

This seems ideal with respect to the percentage considered. However, the order won't be the same if we report accuracy after, say 5 epochs, as seen in the graph. The reason has been mentioned above.

B22CS090_test.py

- Run this python script as '**%run B22CS090_test.py test.txt**' in jupyter notebook / google colab.
- test.txt is read as a command line argument and is used to make the testing txt file.
- Imports Perceptron class from B22CS090_train.py to access some of its functionalities.
- B22CS090_data.txt is read to access the actual class labels of the test set by selecting the last 30% of labels.
- **getStatData()** reads means and standard deviations of the 3 cases.
- Test set is normalised and weights and biases for each case are read.
- Labels are predicted by calculating dot product, and stored in **B22CS090_y_preds.txt**.
- Accuracies are calculated and stored in B22CS090_stats.txt.

Table for Comparison

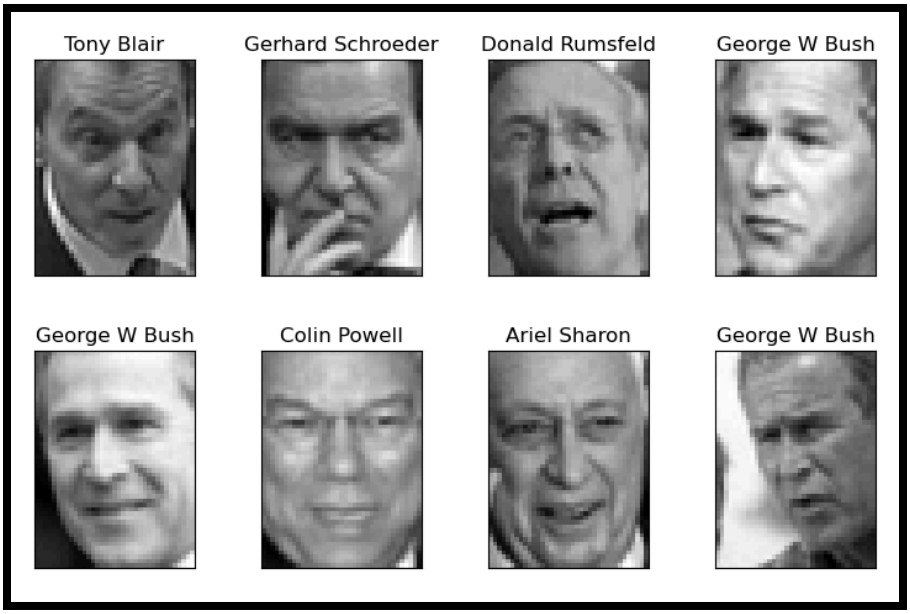
Percentage of Synthetic Data for Training	Training Accuracy	Test Accuracy
20%	96.95%	97.633%
50%	98.76%	98.7%
70%	99.086%	99.3%

We see that test accuracy increases in the increasing order of the percentage of synthetic data used for training. However, it could have been any other order. It depends on the fluctuations and how the decision boundary adjusted itself after the final sample was read.

Question 2:

The following are the steps involved:

- The LFW dataset is loaded from scikit-learn using **fetch_lfw_people()**. **min_faces_per_person** = 50 implies that only those people will be loaded who have a minimum of 50 faces in the dataset. This is done to ensure that there is sufficient training and test data for each label. Suppose, if a person has only 1 picture, it has a very high probability of getting misclassified.
- 40% of the original picture resolution is retained in order to have faster training. [50, 37] is the resolution of every image.

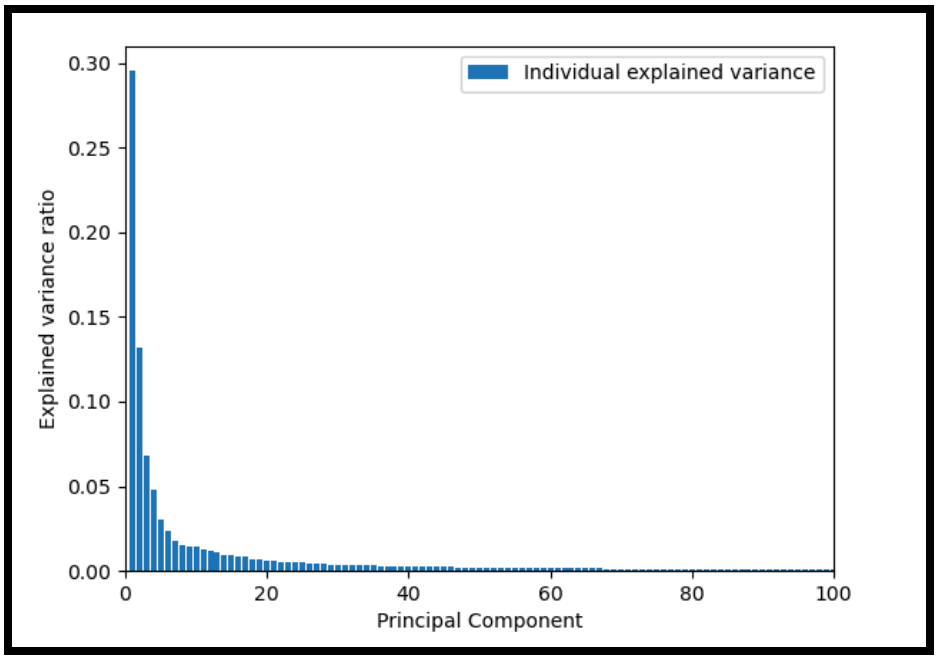


Showing the first 8 images from LFW dataset

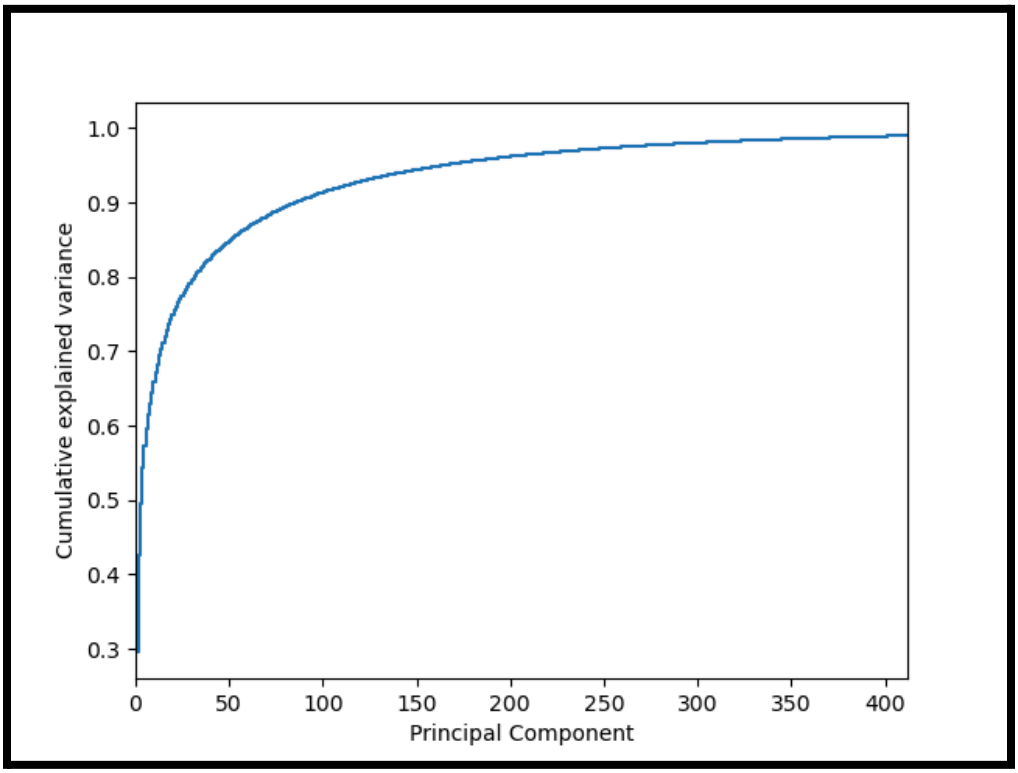
- **train_test_split()** used after flattening the images, to randomly split the dataset into training and test set, and stratification is done based on the class labels.
- Training and test sets are normalised with the means and standard deviations of the training set.
- class **PCA_scratch** is the class containing functionalities related to PCA. I have later compared my implementation with that of scikit-learn and found out that they perform identically.
- **fit_transform()** method is used to transform the training set (dot product) into the PC axes and to retain the eigenvalues and eigenvectors.

- **transform()** method is used to transform the test set (dot product) into the PC axes using the already retained values.
- **eigenval_eigenvec()** method calculates the covariance matrix of the training set and finds the eigenvalues and eigenvectors accordingly.
- **variation_expressed()** calculates the fraction of variation captured by each eigenvector in the training set. It also calculates the cumulative fraction of variation captured, which can be used to find the number of axes desired in order to capture a specific fraction of variance.
  Note: List of fraction of variation captured by each PC axis called **explained_variance_ratio_** in scikit-learn's implementation of PCA.
- **number_components()** calculates the number of axes which capture, say 'x' amount of variance. If x = 0.95, it will return the number of axes beyond which cumulative variance capture becomes > 95%.
- **make_eigenpairs()** creates a list of pairs of eigenvalues and their corresponding eigenvectors. The eigenvalues are sorted in descending order beforehand, in order to ensure that the first axis corresponds to the highest variance captured, then the second highest, and so on.
- **make_projection_matrix()** creates the projection matrix, which is used to transform the dataset into the corresponding PC axes. The dimensions of projection matrix is **[n_features, n_comp]**, where:
  n_features: number of feature columns after flattening the image.
  n_comp: number of PC axes desired.

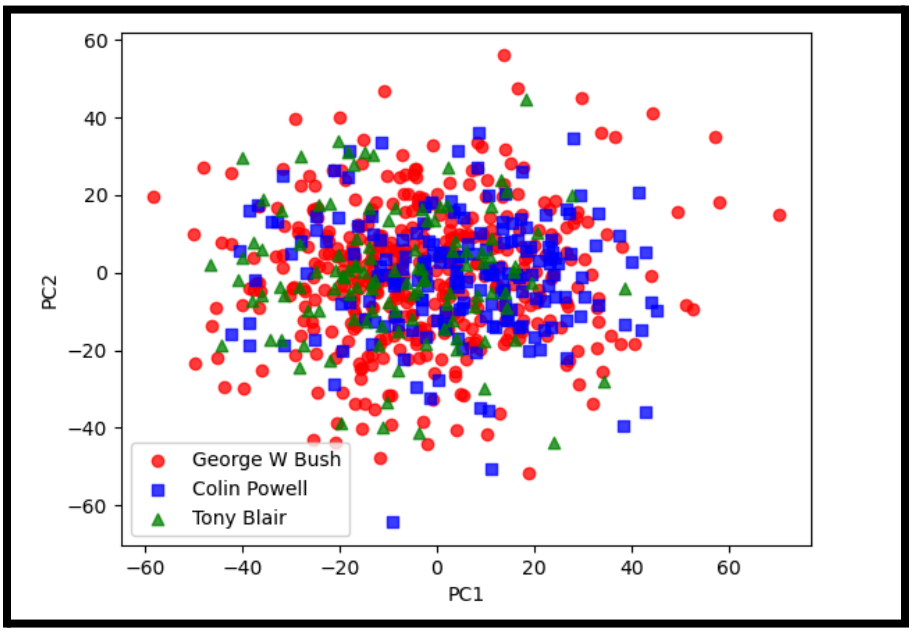
  X' = XW, where:
  X' = transformed dataset in n_comp PC axes
  X = dataset of n_features
  W = projection matrix
- Initially, we capture 99% of the variance, which gives us **412 PC axes**, which is significantly lesser than 50 x 37 = 1850 axes. 99% variation seems ideal as this captures a significant part of the dataset and leaves out the insignificant axes as well. Going beyond this would affect the balance between number of axes and speed of training.
  Note: 412 PC axes is the Scikit-Learn PCA equivalent of **n_components_**.
  From the graph given below, it is evident that capture of variance decreases significantly after 40 PC axes. Yet for 99% capture, we consider 412 PC axes.



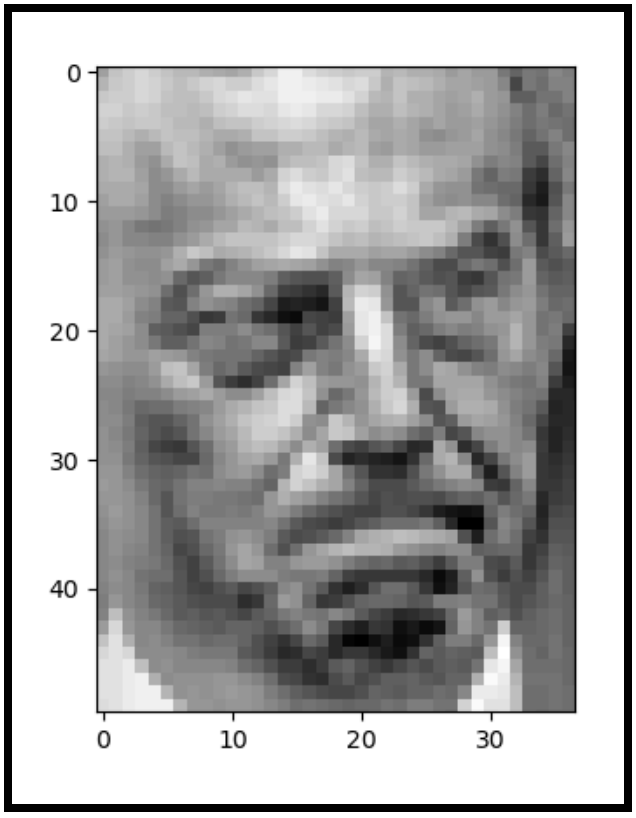
Graph showing fraction of variation captured for the first 100 PCs



- The graph above shows the cumulative fraction of variance for PCs, for the first 412 axes. They capture 99% of the variation. As the PCs increment, the variation fraction decreases significantly, thus taking quite a lot of PCs to reach 100% capture of variance.

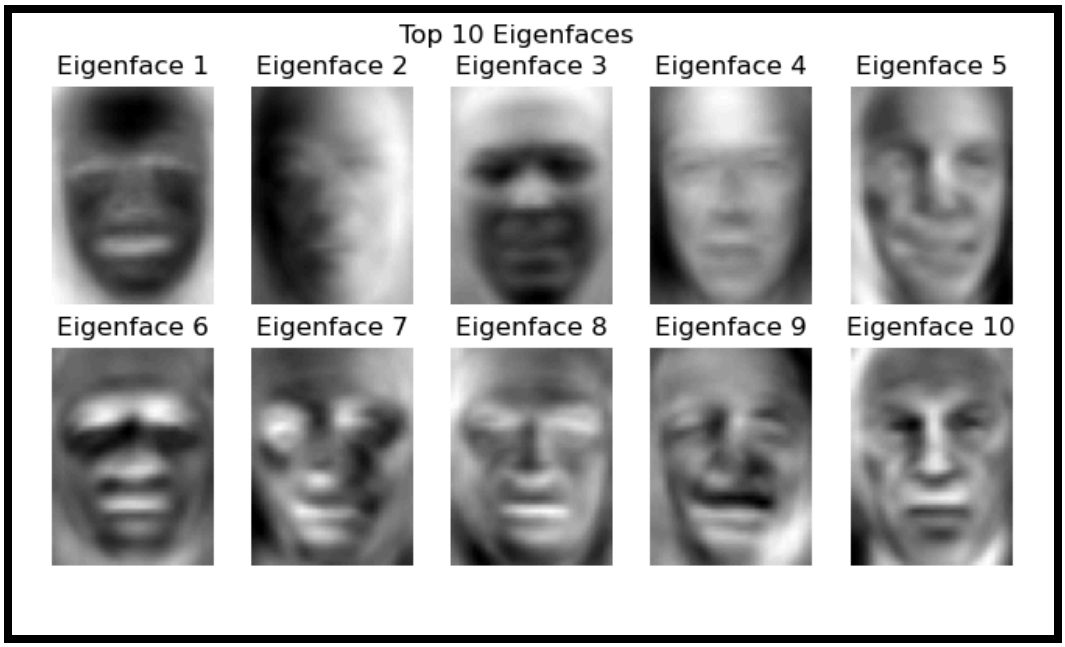


- The graph above shows the separability of the 3 highest occurring class labels in the dataset, plotted using only PC1 and PC2 axes. PC1 and PC2 capture combined variation of around **42.7% of total variation**. Therefore, only 2 axes are not good to separate the samples, which is evident from the graph.
- Logistic regression model is used from scikit-learn to train the dataset.
- **Metrics obtained:**
  Training accuracy: 100%
  Test accuracy: 78.2%
  Clearly, overfitting occurs. The model does well on training data but does not do well on unseen / generalised data. One of the reasons can be the fact that images have locality for each pixel. Basically, each pixel can be related to its surrounding pixels. Traditional ML models cannot capture these patterns because the images are flattened to a 1D array, thereby losing its locality assumption. The model then simply ‘memorises’ the training data and fails to find underlying hidden features.
- **Showing a misclassified sample:**

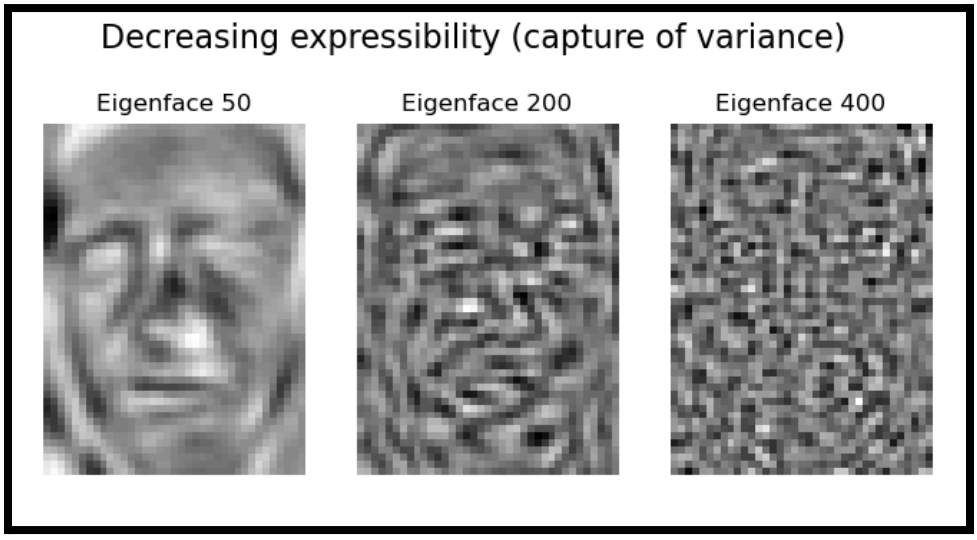


This misclassification may be attributed to the fact that the subject shows a **different facial expression**. Other than this, misclassifications have occurred due to **lack of training examples, non-frontal faces, different background, different ethnicities**, and so on.

- **How can the model be improved?**
    - Hyperparameter tuning using grid search or randomised search.
    - Using a different model like CNN to capture local features to a pixel.
    - Increasing the number of PC axes to capture more variance.
    - Image augmentation like rotation, random crop, scaling, etc.
- Showing the top 10 eigenfaces, ie, the eigenfaces capturing the highest variation in data. Eigenfaces are calculated from the projection matrix.



- Showing how the eigenfaces capture less variation / non interpretable data as we increase the principal component.



- **Experimenting with different values of n_components_ (PC axes)**:
  In my PCA class from scratch, I enter a desired variance to be captured, which gives me the specific number of components. The results are as follows (the table includes the first case as well, ie, 99% variance, which I mentioned above):

Variance	n_components_	Training accuracy	Test accuracy
80%	31	80.128%	69.551%
90%	85	99.439%	75%
92%	108	100%	75.961%
95%	163	100%	76.282%
98%	296	100%	78.846%
99%	412	100%	78.205%
100%	1246	100%	78.525%

Note: 100% variance does not consider all 1850 components because contribution from the last few hundred components become so small that cumulative variation fraction approximates to 1 before reaching the last component.

Overfitting is clearly a problem. High generalisation is not achieved.